Higher-order functions and currying

Recursionless programming

LYaH: Chapter 6 *Higher Order Functions*

Encapsulating common recursion patterns

Compare two defs of a function adding `1` to every element of a list.

```
add1 :: [Int] -> [Int]
add1 [] = []
add1 (x : l) = (x+1) : add1 l

add1' :: [Int] -> [Int]
add1' l = map (+1) l
```

```
add1 [1,2,3] = add1 (1 : [2,3]) = (1+1) : add1 [2,3] = ... = [2,3,4]

add1' [1,2,3] = [(+1) 1, (+1) 2, (+1) 3] = [2,3,4]
```

Understanding `add1` needs recursion.

Understand `add1'` does not: we understand `map` directly.

Recursionless programming

$[x_1, \ldots, x_m] :: [a]$

$f : a \to b \to c$

$[y_1, \ldots, y_m] :: [b]$

result: $[f\ x_1\ y_1, \ldots, f\ x_m\ y_m] :: [c]$

Idea:

- design small library of functions over lists; a list "API"
- implement list programs using combos of API functions, without recursion

Partial listing below. Exercise: use their types to guess what the last five do.

```
map     :: (a -> b) -> [a] -> [b]
filter  :: (a -> Bool) -> [a] -> [a]
all     :: (a -> Bool) -> [a] -> Bool
any     :: (a -> Bool) -> [a] -> Bool
elem    :: Eq a => a -> [a] -> Bool
take    :: Int -> [a] -> [a]
drop    :: Int -> [a] -> [a]
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
foldr   :: (a -> b -> b) -> b -> [a] -> b
foldr1  :: (a -> a -> a) -> [a] -> a
```

zipWith $f\ l_1\ l_2 \to [c]$

special case of foldr / there's also foldl sum [1,2,3] = foldr1 (+) [1,2,3]

$\underset{1}{x}$ $\underset{\text{`op`}}{+}$ foldr1 (+) [2,3] = 1 + 2 + foldr1 (+) [3]

= 1 + 2 + 3 = 6

Fold and zipWith

```
-- zipWith f [x1,...,xn] [y1,...,yn] = [f x1 y1, ..., f xn yn]
zipWith f [] l = []
zipWith f l [] = []
zipWith f (x1:l1) (y1:l1) = f x1 y1 : zipWith f l1 l2
```

into (a -> a -> a)

recursive call

```
-- foldr1 op [x1,...,xn] = x1 `op` (x2 `op`(... `op` xn)...)
foldr1 op [] = error "foldr1: empty list"
foldr1 op [x] = x
foldr1 op (x : l) = x `op` (foldr1 op l)
        -- or: = op x (foldr1 op l)
```

sum l = foldr1 (+) l

(OR sum = foldr1 (+))

Example

Dot product of "vectors": *(x1,..,xn) · (y1,...,yn) = x1*y1 + ... + xn*yn*

```
dot :: Num a => [a] -> [a] -> a
dot u v = foldr1 (+) (zipWith (*) u v)
```

```
dot [1,2,3] [4,5,6] =
foldr1 (+) (zipWith (*) [1,2,3] [4,5,6]) =
foldr1 (+) [(*) 1 4, (*) 2 5, (*) 3 6] =
foldr1 (+) [4,       10,       18] =
32
```

Note: didn't make any (new) recursive definitions.

Just used specification of `foldr1` and `zipWith` to "compute".

foldr

result for rest of list

x `op` z
∈a ∈b

Compare types:

```
foldr   :: (a -> b -> b) -> b -> [a] -> b
foldr1  :: (a -> a -> a) ->      [a] -> a
```

x `op` z
∈a ∈a

Spec:

```
foldr op z [x1, x2, ..., xn] == x1 `op` (x2 `op` ... (xn `op` z)...)
```

Examples:

```
foldr (+) (-1) [1,2,3] = 1 + (2 + (3 + (-1))) = 5
foldr (:) [4] [1,2,3] = 1 : (2 : (3 : [4])) = [1,2,3,4]
```

common usage

x (:) ∈a ∈[a]

`op`

z

Exercises

Use `foldr` to

1. implement `map`,
2. implement `filter`,
3. reverse a list,
4. split a list according to a predicate.

See Haskell file.

Handy supporting functions: `compose` and `flip`

```
f . g =
  \x -> f (g x)

flip f = \x y -> f y x
```

```
((+1) . (+1)) 17 =
(\x -> (+1) ((+1) x)) 17 =
(+1) ((+1) 17) =
(+1) 18 =
19
```

```
flip (-) 3 2 =
(\x y -> (-) y x) 3 2 =
(-) 2 3 =
2 - 3 = -1
```

Handy supporting concept: currying

Let's digress into math. A function is a set of ordered pairs.

Consider $f(x,y) = x+y$.

> $f = \{ ((x,y), x+y) \mid x,y \in \mathbb{Z} \}$

For each $x \in \mathbb{Z}$, let

> $f_x = \{ (y, x+y) \mid y \in \mathbb{Z} \}$

so f_x is a function and $f_x(y) = x+y$ for $y \in \mathbb{Z}$.

Facts:

1. For all $x,y \in \mathbb{Z}$, $f_x(y) = f(x,y)$
2. $f \in \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$
3. For all $x \in \mathbb{Z}$, $f_x \in \mathbb{Z} \to \mathbb{Z}$
4. The mapping $x \mapsto f_x$ is in $\mathbb{Z} \to (\mathbb{Z} \to \mathbb{Z})$.

Currying in Haskell

Consider a typical 2-argument function in Haskell.

```
f :: Double -> Double -> Double  -- Double is floating-point numbers.
f x y = (x+y)/2
```

We can turn this into a function on pairs of numbers:

```
f' :: (Double, Double) -> Double
f' (x,y) = (x+y)/2
```

Difference:

1. To apply `f`, need to supply both *x* and *y* to make the pair *(x,y)*.
2. In contrast, `f x` is an expression in Haskell.
   It's the function that maps `y` to `(x+y)/2`. E.g. `(f 17) 1 = 9`.

Pseudo-definition of currying in Haskell

The "curried" form of a function is the default in Haskell. E.g.

1. `f 1 2 3` is the same as `((f 1) 2) 3`.
2. `Int -> Int -> Int -> Int` is the same as `Int -> (Int -> (Int -> Int))`.

The "uncurried" form of a function uses pairs/tuples. E.g.

1. Instead of `f 1 2 3` have `f' (1,2,3)`.
2. Instead of `f :: Int -> Int -> Int -> Int` have `f' :: (Int,Int,Int) -> Int`.

Currying/uncurrying

```
ghci> :ty curry
curry :: ((a, b) -> c) -> a -> b -> c
ghci> :ty uncurry
uncurry :: (a -> b -> c) -> (a, b) -> c
```

```
curry f x y = f (x,y)
uncurry f (x,y) = f x y
-- OR, equivalently
curry f = \x y -> f (x,y)
uncurry f = \(x,y) -> f x y  -- note: can use pattern-matching in lambda exps
```

Some "symbolic" computation, supposing `f :: a -> b -> c`:

```
curry (uncurry f) =
curry (\(x,y) = f x y) =
\x y -> (\(x,y) = f x y) (x,y) =
\x y -> f x y  -- semantically the same as f: same results on all inputs
```

Practical use of curried form of functions

We've already seen how it's useful to "partially apply" functions, e.g.

> apply `(+) :: Int -> Int -> Int` to `1` to get `(+ 1) :: Int -> Int`

Example from Assignment 3 sample solution:

```
data DB = DB [[Int]]
equivDB :: DB -> DB -> Bool
equivDB (DB lss1) (DB lss2) =
  length lss1 == length lss2
    && all (`elem` lss2) lss1
    && all (`elem` lss1) lss2
```

Exercise: check the types of `all` and `elem` and verify they work in the above.

See Haskell file for another example.