Functional programming

Part two!

Higher-order function basics (1)

- Functions with multiple arguments are usually defined using the notion of currying
- That is, arguments are taken one at a time by exploiting the fact that functions can return functions
- Example:

```
add :: Int -> Int -> Int
add x y = x + y
```

Actually means

```
add :: Int -> (Int -> Int)
add = \x -> (\y -> x + y)
```

Higher-order function basics (2)

- In Haskell, you can also define functions that take functions as arguments.
- For example a function that applies a function twice to an argument:

```
twice :: (a -> a) -> a -> a
twice f x = f (f x)
```

- Considering that this is a multi-argument function (and therefore curried), it can be used to define other functions. For example:

```
quadruple :: Num a => a -> a
quadruple x = twice (*2)
```

Higher-order function basics (3)

- Formally, functions that either take or return other functions are called *higher-order functions*.
- Usually we use the term *curried* for functions that return functions
- In practice, with *higher-order* we refer to functions that take functions
- Higher-order functions allow common programming patterns to be encapsulated as functions within the language itself
- Higher-order functions can be used to define ‘domain-specific’ languages

Processing lists (1) - map

- There are a number of useful higher-order functions in the prelude
- The function map applies a function to each element of a list

map :: (a -> b) –> [a] -> [b]

- map is polymorphic, so it applies to any list
- map can be applied in a nested manner, for example:
  map (map (+1)) [[1,2,3],[4,5]]

Processing lists (2) - filter

- The function filter selects all elements from a list that satisfy the provided predicate.

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

- Again, this is a polymorphic function

```
filter even [1..10]
```

- Often, map and filter are used in a combination, for example:

```
sumSqrEven :: [Int] -> Int
sumSqrEven ns = sum (map (^2)(filter even ns))
```

Processing lists (3) – more handy functions

```
all :: (a -> Bool) -> [a] -> Bool
any :: (a -> Bool) -> [a] -> Bool
takeWhile :: (a -> Bool) -> [a] -> [a]
dropWhile :: (a -> Bool) -> [a] -> [a]
```

The *foldr* function (1)

- Many functions on lists can be defined using the following simple pattern:

f [] = v
f (x:xs) = x ⊕ f xs

- Examples:

sum [] = 0
sum (x:xs) = x + sum xs

or [] = False
or (x:xs) = x || or xs

The *foldr* function (2)

- The library function *foldr* makes it very convenient to express such functions
- *Foldr* is an abbreviation for ‘fold right’
- The previous definitions can be rewritten using *foldr* as follows:

```
sum  = foldr (+) 0
or = foldr (||) False
```

- The arguments are implicit in the above definitions, but can be made explicit:

```
sum xs = foldr (+) 0 xs
```

The *foldr* function (3)

- The *foldr* function is defined as follows:

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

- This recursive definition can be slightly mind-bending
- What is a simpler way to think of the behavior of foldr?

The *foldr* function (4)

- The foldr function defines a rather simple pattern of recursion
- However, it can define a larger amount of functions than expected
- Consider length:
  ```
  length :: [a] -> Int
  length [] = 0
  length (_:xs) = 1 + length xs
  ```
- Can be defined as:
  ```
  length = foldr (\_ n -> 1 + n) 0
  ```

The *foldr* function (5)

- Now let's try reverse, which is slightly more complex:
  reverse :: [a] -> [a]
  reverse [] = []
  reverse (x:xs) = reverse xs ++ [x]
- Applying reverse to [1,2,3] basically gives
  (([] ++ [3]) ++ [2]) ++ [1]
- How do we foldr this?
- Basically we perform an 'inversed cons' operation for each element, adding to the end of the list instead of to the start

The *foldr* function (6)

- Let's define this reversed cons:
  ```
  snoc :: a -> [a] -> [a]
  snoc x xs = xs ++ [x]
  ```
- Now we can define reverse as:
  ```
  reverse [] = []
  reverse (x:xs) = snoc x (reverse xs)
  ```
- And this can be changed to foldr trivially
  ```
  reverse = foldr snoc []
  ```

The *foldl* function (1)

- The *foldr* function *folds right*. The right refers to right-associative
- There is also a left-associative variant, called *foldl (fold left)*.
- This one can be slightly harder to get your head around
- When used with associative functions, foldr and foldl produce the same result
- Evaluation order might result in different performance

The *foldl* function (2)

- Consider a slightly more complex implementation of *sum*, which uses a helper *sum’* with an *accumulator value* to accumulate the final result (note that we again use implicit argument passing)

```
sum xs = sum’ 0 xs
         where
           sum’ v [] = v
           sum’ v (x:xs) = sum’ (v + x) xs
```

The *foldl* function (3)

- How does this work?

```
 sum [1,2,3]
= { apply sum }
 sum’ 0 [1,2,3]
= { apply sum’ }
 sum’ (0+1) [2,3]
= { apply sum’ }
 sum’ ((0+1) + 2) [3]
= { apply sum’ }
 sum’ ((0+1) + 2) + 3
= { apply + }
 6
```

The *foldl* function (4)

- Again, we can generalize and define many functions on list using the pattern:

f v [] = v
f v (x:xs) = f (v ⊕ x) xs

- Map the empty list to the accumulator value v
- Recursively create a new accumulator value by adding the current accumulator value to the head of the list, and recursively processing the tail with the new accumulator

The *foldl* function (5)

- There’s a function for that!
- *foldl* applies this pattern
- Again, we can redefine sum and or:

```
sum = foldl (+) 0
or = foldl (||) False
```

- And length and reverse:

```
length = foldl (\n _ -> n + 1) 0
reverse = foldl (\xs x -> x:xs) []
```

Function composition (1)

- The higher-order library function . returns the composition of two functions

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
f . g = \x -> f (g x)
```

- Read f . g as ‘f composed with g’
- f . g is the function that takes an argument x, applies g to it, and then applies f to the result of that
- Function composition can be used to simplify nested function applications, by reducing parenthesis and avoiding references to arguments

Function composition (2)

- Examples

```
odd n = not (even n)
odd = not . even
twice f x = f (f x)
twice f = f . f
sumSqrEven ns = sum (map (^2) (filter even ns))
sumSqrEven = sum . map (^2) . filter even
```

- The last definition uses the fact that composition is associative

Function composition (3)

- Composition has an identity, given by the *identity function*

id :: a -> a
id = \x -> x

- The identify function has the property that id . f = f and f . id = f for any f
- What can we do with this?
- Use foldr to compose lists of functions

```
compose :: [a -> a] -> (a -> a)
compose = foldr (.) id
```

Type declarations (1)

- Haskell allows you to define new types
- The simplest way of declaring a new type is to introduce a new name for an existing type
- Type declarations use the type keyword
- Example from the prelude:

```
type String = [Char]
```

Type declarations (2)

- Type declarations can be nested, in the sense that one type can be declared in terms of another
- Consider a game board (e.g. Go)
- A type for such a board can be defined as follows:

```
type Pos = (Int, Int)
type Board = [Pos]
```

Type declarations (3)

- How would you declare a Tree type?

type Tree = (Int, [Tree])

- This isn't allowed in Haskell, simple type declarations cannot be recursive

Type declarations (4)

- Type declarations can be parametrized
- Consider our lookup table, we can define a parametrized type for it

```
type Assoc k v = [(k, v)]
```

- And we can use this in our definition of find:

```
find     :: Eq k => k -> Assoc k v -> v
find k t = head [v | (k', v) <- t, k == k']
```

Data declarations (1)

- Type declarations basically create ‘type aliases’
- It is possible to define entirely new types using *data declarations*
- Data declarations use the data keyword
- For example, consider the definition of Bool from the prelude:

data Bool = True | False

- The | is read as *or*
- True and False are *constructors*
- Note that both the type and the constructors have no meaning yet

data A = B | C would be exactly the same

Data declarations (2)

- Values of new types can be used exactly as those of built-in types

```
data Move = Left | Right | Up | Down

move :: Move -> Pos -> Pos
move Left (x, y) = (x – 1, y)
move Right (x, y) = (x + 1, y)
move Up (x, y) = (x, y - 1)
move Down (x, y) = (x, y + 1)
```

Data declarations (3)

```
moves :: [Move] -> Pos -> Pos
moves [] p = p
moves (m:ms) p = moves ms (move m p)

flip :: Move -> Move
flip Left = Right
flip Right = Left
flip Up = Down
flip Down = Up
```

Data declarations (3)

- Constructors in data declarations can have arguments
- Consider shapes:

data Shape = Circle Float | Rect Float Float

- Shape has values of the forms:
  - Circle r, where r is a floating point number
  - Rect x y, where x and y are floating point numbers

Data declarations (4)

- What is the use?
- These constructors can be used to define functions on shapes
- Examples:

```
square :: Float -> Shape
square n = Rect n n
```

```
area :: Shape -> Float
area (Circle r) = pi * r ^2
area (Rect x y) = x * y
```

Data declarations (5)

- Data declarations can be parameterized as well
- The Prelude declares the following type

```
data Maybe a = Nothing | Just a
```

- A value of type Maybe a is either Nothing, or of the form Just a
- Nothing represents ‘failure’
- Just represents ‘success’

Data declarations (6)

- Maybe can be used to define safe versions of div and head

```
safediv :: Int -> Int -> Maybe Int
safediv _ 0 = Nothing
safediv m n = Just (m ‘div’ n)

safehead :: [a] -> Maybe a
safehead [] = Nothing
safehead xs = Just (head xs)
```

Recursive types (1)

- Types declared using the data mechanism can be recursive
- Consider the following type

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

- Nat is either Zero or Succ n for some n of type Nat

```
Zero
Succ Zero
Succ (Succ Zero)
Succ (Succ (Succ Zero))
```

- Natural numbers with Zero = 0 and Succ = 1 + Nat
  Succ (Succ (Succ Zero)) = 1 + (1 + (1 + 0)) = 3

Recursive types (2)

- We can define the following conversions to/from integer for this

```
nat2int :: Nat -> Int
nat2int Zero = 0
nat2int (Succ n) = 1 + nat2int n

int2nat :: Int -> Nat
int2nat 0 = Zero
int2nat n = Succ (int2nat (n-1))
```

Recursive types (3)

- We can define add on this type as well
- Obviously by conversion

```
add :: Nat -> Nat -> Nat
add m n = int2nat (nat2int m + nat2int n)
```

- But more efficiently by recursion

```
add Zero n = n
add (Succ m) n = Succ (add m n)
```

Recursive types (4)

- How does it work?
- 2 + 1 = 3

```
 add (Succ (Succ Zero)) (Succ Zero)
= {applying add}
 Succ (add (Succ Zero) (Succ Zero))
= {applying add}
 Succ ( Succ ( add Zero (Succ Zero)))
= {applying add}
 Succ (Succ (Succ Zero))
```

Recursive types (5)

- Let’s define our own List

```
data List a = Nil | Cons a (List a)
```

- A List is either Nil (empty)
- Or of the form Cons x xs form some values x :: a and xs :: List a (non-empty)
- Now we can also calculate the length of such a list:

```
len :: List -> Int
len Nil = 0
len (Cons _ xs) = 1 + len xs
```

Recursive types (6)

- And finally, let’s define a tree (for Ints this time)
- Our tree contains values in both the nodes and leaves

```
data Tree = Leaf Int | Node Tree Int Tree
```

- How do we use it?

```
t :: Tree
t = Node (Node (Leaf 1) 3 (Leaf 4)) 5 (Node (Leaf 6) 7 (Leaf 9))
```

Recursive types (7)

- Some convenient functions
- Does a certain value occur in our tree?

```
occurs :: Int -> Tree -> Bool
occurs m (Leaf n) = m == n
occurs m (Node l n r) = m == n || occurs m l || occurs m r
```

- Convert (flatten) a tree to a list?

```
flatten :: Tree -> Int
flatten (Leaf n) = [n]
flatten (Node l n r) = flatten l ++ [n] ++ flatten r
```

Recursive types (8)

- If our tree is *sorted* we can do occurs more efficiently

```
occurs m (Leaf n) = m == n
occurs m (Node l m r)
    | m == n      = True
    | m < n       = occurs m l
    | otherwise = occurs m r
```

- And we have a binary search tree

Recursive types (9)

- To really conclude it all, some alternative trees

```
data Tree a = Leaf a | Node (Tree a) (Tree a)
data Tree a = Leaf | Node (Tree a) a (Tree a)
data Tree a b = Leaf a | Node (Tree a b) b (Tree a b)
data Tree a = Node a [Tree a]
data Tree a b = Leaf a | Node b [Tree a b]
```