COMP90048 Declarative Programming
Semester 1, 2018
Peter J. Stuckey

Declarative Programming

Answers to workshop exercises set 5.

QUESTION 1
Define the function

```
maybeApply :: (a -> b) -> Maybe a -> Maybe b
```

that yields Nothing when the input Maybe is Nothing, and
applies the supplied function to the content of the Maybe when it is
Just some content.

Try, for example, computing

```
maybeApply (+1) (Just 41)
maybeApply (+1) Nothing
```

This function is defined in the standard prelude as fmap.

ANSWER

```
>maybeApply f Nothing = Nothing
>maybeApply f (Just x) = Just $ f x
```

QUESTION 2
Define the function

```
zWith :: (a -> b -> c) -> [a] -> [b] -> [c]
```

that constructs a list of the result of applying the first argument to
corresponding elements of the two input lists. If the two list
arguments are different lengths, the extra elements of the longer one
are ignored. For example,

```
zWith (-) [1,4,9,16] [1,2,3,4,5] = [0,2,6,12]
```

This function is defined in the standard library as 'zipWith'.

ANSWER

```
>zWith f [] _ = []
>zWith f _ [] = []
>zWith f (x:xs) (y:ys) = f x y : zWith f xs ys
```

QUESTION 3
Define the function

```
    linearEqn :: Num a => a -> a -> [a] -> [a]
```

that constructs a list of the result of multiplying each element in the third argument by the first argument, and then adding the second argument. For example,

```
    linearEqn 2 1 [1,2,3] = [2*1+1, 2*2+1, 2*3+1] = [3,5,7]
```

Write the simplest defintion you can, remembering the material covered recently.

ANSWER

```
>linearEqn :: Num a => a -> a -> [a] -> [a]
>linearEqn m n = map (\x -> m*x + n)
```

QUESTION 4
The following function takes a number and returns a list containing the positive and negative square roots of the input (assume non-zero input)

```
>sqrtPM :: (Floating a, Ord a) => a -> [a]
>sqrtPM x
>  | x  > 0    = let y = sqrt x in [y, -y]
>  | x == 0    = [0]
>  | otherwise = []
```

Using this function, define a function allSqrts that takes a list and returns a list of all the positive and negative square roots of all the numbers on the list. For example:

```
    allSqrts [1,4,9] = [1.0,-1.0,2.0,-2.0,3.0,-3.0]
```

Include a type declaration for your function.

ANSWER

```
>allSqrts :: (Floating a, Ord a) => [a] -> [a]
>allSqrts xs = foldl (++) [] (map sqrtPM xs)
```

Here's a simpler definition:

```
>allSqrts1 :: (Floating a, Ord a) => [a] -> [a]
>allSqrts1 xs = concat (map sqrtPM xs)
```

Simpler still:

```
>allSqrts2 :: (Floating a, Ord a) => [a] -> [a]
>allSqrts2 xs = concatMap sqrtPM xs
```

QUESTION 5
Lectures have given the definitions of two higher order functions in the
Haskell prelude, filter and map:

```
    filter :: (a -> Bool) -> [a] -> [a]
    map :: (a -> b) -> [a] -> [b]
```

Filter returns those elements of its argument list for which the given function
returns True, while map applies the given function to every element of the
given list.

Suppose you have a list of numbers, and you want to (a) filter out all the
negative numbers, and (b) apply the sqrt function to all the remaining integers.

(a) Write code to accomplish this task using filter and map.
(b) Write code to accomplish this task that does only one list traversal, without any higher order functions.
(c) Transform (b) to (a).

ANSWER
Filter + map version. The inner call to filter traverses xs and
produces an intermediate list, which is traversed by map.

```
>sqrt_pos1 :: (Ord a, Floating a) => [a] -> [a]
>sqrt_pos1 ns = map sqrt (filter (>=0) ns)
```

Single-traversal version - we just do more complex processing for each element, combining the test for >=0 and sqrt.

```
>sqrt_pos2 :: (Ord a, Floating a) => [a] -> [a]
>sqrt_pos2 [] = []
>sqrt_pos2 (x:xs) =
>   if x >= 0 then sqrt x : sqrt_pos2 xs
>   else sqrt_pos2 xs
```

The definitions from lectures are

```
filter :: (a -> Bool) -> [a] -> [a]
filter _ [] = []
filter f (x:xs) =
    if f x == True then x:fxs else fxs
    where
        fxs = filter f xs

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

The definition of sqrt_pos ns is map sqrt (filter (>=0) ns). We can use a form of structural induction on the list ns and use the definitions of
filter and map above to simplify instances of this expression, as follows:

```
For ns=[] we have
    map sqrt (filter (>=0) [])
=   map sqrt []
=   []
```

Thus we can derive the equation

```
sqrt_pos [] = []
```

```
For ns = (x:xs) we have
    map sqrt (filter (>=0) (x:xs))
=   map sqrt (if (>=0) x == True then x:fxs else fxs)
        where
        fxs = filter (>=0) xs
=   map sqrt (if x >= 0 then x:filter (>=0) xs else filter (>=0) xs)
=   if x >= 0 then map sqrt (x:filter (>=0) xs)
```

```
    else map sqrt (filter (>=0) xs)
=   if x >= 0 then sqrt x : map sqrt (filter (>=0) xs)
    else map sqrt (filter (>=0) xs)
=   if x >= 0 then sqrt x : sqrt_pos xs
    else sqrt_pos xs
```

Thus we can derive the equation

```
sqrt_pos (x:xs) =
       if x >= 0 then sqrt x : sqrt_pos xs
       else sqrt_pos xs
```