DelftX: FP101x Introduction to Functional Programming

EXERCISE 0 (1 point possible)

Choose the equivalent of the following list comprehension `[f x | x <- xs, p x]` expressed using higher-order functions.

- ( ) `map p (map f xs)`
- ( ) `filter p (map f xs)`
- (•) `map f (filter p xs)`
- ( ) `map f (takeWhile p xs)`

You have used 1 of 1 submissions

EXERCISE 1 (1 point possible)

Choose **all** options that implement the Prelude function

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

taking into account only finite, non-partial input lists with non-bottom values and where the predicate p always returns either True, or False, but not bottom.

- [x] `all p xs = and (map p xs)`
- [ ] `all p xs = map p (and xs)`
- [x] `all p = and . map p`
- [x] `all p = not . any (not . p)`

- [ ] `all p = map p . and`
- [x] `all p xs = foldl (&&) True (map p xs)`
- [ ] `all p xs = foldr (&&) False (map p xs)`
- [x] `all p = foldr (&&) True . map p`

For additional understanding, try to experiment with infinite and partial lists and see if you can spot any differences in behaviour for the various implementations.

You have used 0 of 1 submissions

EXERCISE 2 (1 point possible)

Choose **all** options that implement the Prelude function

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

taking into account only finite, non-partial input lists with non-bottom values and where the predicate p always returns either True, or False, but not bottom.

- [ ] `any p = map p . or`
- [x] `any p = or . map p`
- [x] `any p xs = length (filter p xs) > 0`
- [x] `any p = not . null . dropWhile (not . p)`
- [ ] `any p = null . filter p`
- [x] `any p xs = not (all (\ x -> not (p x)) xs)`

- [x]
```haskell
any p xs = foldr (\ x acc -> (p x) || acc) False xs
```

- [ ]
```haskell
any p xs = foldr (||) True (map p xs)
```

For additional understanding, try to experiment with infinite and partial lists and see if you can spot any differences in behaviour for the various implementations.

You have used 0 of 1 submissions

EXERCISE 3 (1 point possible)

Choose the option that implements the Prelude function

```haskell
takeWhile :: (a -> Bool) -> [a] -> [a]
```

taking into account only finite, non-partial input lists with non-bottom values and where the predicate p always returns either True, or False, but not bottom.

- ( )
```haskell
takeWhile _ [] = []
takeWhile p (x : xs)
  | p x = x : takeWhile p xs
  | otherwise = takeWhile p xs
```

- (•)
```haskell
takeWhile _ [] = []
takeWhile p (x : xs)
  | p x = x : takeWhile p xs
  | otherwise = []
```

- ( )
```haskell
takeWhile _ [] = []
takeWhile p (x : xs)
  | p x = takeWhile p xs
  | otherwise = []
```

- ( )
```haskell
takeWhile p = foldl (\ acc x -> if p x then x : acc else acc) []
```

For additional understanding, try to experiment with infinite and partial lists and see if you can spot any differences in behaviour for the various implementations.

You have used 0 of 1 submissions

EXERCISE 4 (1 point possible)

Choose the option that implements the Prelude function

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

taking into account only finite, non-partial input lists with non-bottom values and where the predicate p always returns either True, or False, but not bottom.

- (selected)
```
dropWhile _ [] = []
dropWhile p (x : xs)
  | p x = dropWhile p xs
  | otherwise = x : xs
```

-
```
dropWhile _ [] = []
dropWhile p (x : xs)
  | p x = dropWhile p xs
  | otherwise = xs
```

-
```
dropWhile p = foldr (\ x acc -> if p x then acc else x : acc) []
```

-
```
dropWhile p = foldl add []
  where add [] x = if p x then [] else [x]
        add acc x = x : acc
```

For additional understanding, try to experiment with infinite and partial lists and see if you can spot any differences in behaviour for the various implementations.

You have used 0 of 1 submissions

EXERCISE 5 (1 point possible)

Choose the option that implements the Prelude function

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

taking into account only finite, non-partial input lists with non-bottom values and where the mapping function does not return bottom.

- ( ) `map f = foldr (\ x xs -> xs ++ [f x]) []`
- ( ) `map f = foldr (\ x xs -> f x ++ xs) []`
- ( ) `map f = foldl (\ xs x -> f x : xs) []`
- (•) `map f = foldl (\ xs x -> xs ++ [f x]) []`

For additional understanding, try to experiment with infinite and partial lists and see if you can spot any differences in behaviour for the various implementations.

You have used 0 of 1 submissions

EXERCISE 6 (1 point possible)

Choose the option that implements the Prelude function

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

taking into account only finite, non-partial input lists with non-bottom values and where the predicate p always returns either True, or False, but not bottom.

- ( ) `filter p = foldl (\ xs x -> if p x then x : xs else xs) []`
- (•) `filter p = foldr (\ x xs -> if p x then x : xs else xs) []`
- ( ) `filter p = foldr (\ x xs -> if p x then xs ++ [x] else xs) []`
- ( ) `filter p = foldl (\ x xs -> if p x then xs ++ [x] else xs) []`

For additional understanding, try to experiment with infinite and partial lists and see if you can spot any differences in behaviour for the various implementations.

You have used 0 of 1 submissions

EXERCISE 7 (1 point possible)

Choose a definition for the function `dec2int :: [Integer] -> Integer` that converts a finite, non-partial list of non-bottom Integer digits, that represents a decimal number, into the non-bottom Integer this list represents. For example:

```
> dec2int [2, 3, 4, 5]
2345
> dec2int []
0
> dec2int [0, 0, 0, 0]
0
```

- ( ) `dec2int = foldr (\ x y -> 10 * x + y) 0`
- ( ) `dec2int = foldl (\ x y -> x + 10 * y) 0`
- (•) `dec2int = foldl (\ x y -> 10 * x + y) 0`
- ( ) `dec2int = foldr (\ x y -> x + 10 * y) 0`

For additional understanding, try to experiment with infinite and partial lists and see if you can spot any differences in behaviour for the various implementations.

You have used 0 of 1 submissions

EXERCISE 8 (1 point possible)

Choose an explanation for why the following definition of `sumsqreven` is invalid:

```
sumsqreven = compose [sum, map (^ 2), filter even]
```

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

- ( ) The definition of compose doesn't typecheck.

- ( ) A tuple must have values of different types.
- ( ) This code is not valid Haskell syntax.
- (•) The definition of sumsqreven doesn't even typecheck.

You have used 0 of 1 submissions

EXERCISE 9 (1 point possible)

Choose the correct definition for the Prelude function `curry :: ((a, b) -> c) -> a -> b -> c`, that converts a function that takes its arguments as a pair into a function that takes its arguments one at a time. For this exercise assume that bottom does not exist.

- ( ) `curry f = \ x y -> f x y`
- ( ) `curry f = \ x y -> f`
- (•) `curry f = \ x y -> f (x, y)`
- ( ) `curry f = \ (x, y) -> f x y`

For additional understanding, try to experiment with undefined and partial tuples, and see if you can spot any differences in behaviour for the various implementations.

You have used 0 of 1 submissions

EXERCISE 10 (1 point possible)

Choose the definition for the Prelude function `uncurry :: (a -> b -> c) -> (a, b) -> c`, that converts a function that takes its arguments one at a time into a function that takes its arguments as a pair. For this exercise assume that bottom does not exist.

- (•) `uncurry f = \ (x, y) -> f x y`

- ( ) `uncurry f = \ x y -> f (x, y)`
- ( ) `uncurry f = \ (x, y) -> f`
- ( ) `uncurry f = \ x y -> f`

For additional understanding, try to experiment with undefined and partial tuples, and see if you can spot any differences in behaviour for the various implementations.

You have used 0 of 1 submissions

EXERCISE 11 (1 point possible)

Consider the following higher-order function `unfold :: (b -> Bool) -> (b -> a) -> (b -> b) -> b -> [a]` that encapsulates a simple pattern of recursion for producing a list.

```
unfold p h t x
  | p x = []
  | otherwise = h x : unfold p h t (t x)
```

The function `unfold p h t x` produces the empty list if the predicate `p x` is True. Otherwise it produces a non-empty list by applying the function `h x` to give the head of the generated list, and the function `t x` to generate another seed that is recursively processed by unfold to produce the tail of the generated list.

For example, the function `int2bin`, that converts a non-negative integer into a binary number, **with the least significant bit first**, can be defined as:

For example:

```
type Bit = Int

int2bin :: Int -> [Bit]
int2bin 0 = []
int2bin n = n `mod` 2 : int2bin (n `div` 2)
```

```
> int2bin 13
[1, 0, 1, 1]
> int2bin (-0) -- Yes, 0 can be negative!
[]
```

This function can be rewritten more compactly using `unfold` as follows:

```
int2bin = unfold (== 0) (`mod` 2) (`div` 2)
```

Next consider the function `chop8 :: [Bit] -> [[Bit]]` that takes a list of bits and chops it into lists of at most eight bits (assuming the list is finite, non-partial, and does not contain bottom):

```
chop8 :: [Bit] -> [[Bit]]
chop8 [] = []
chop8 bits = take 8 bits : chop8 (drop 8 bits)
```

Choose an implementation of `chop8` using `unfold`.

- ( ) `chop8 = unfold [] (drop 8) (take 8)`
- (•) `chop8 = unfold null (take 8) (drop 8)`
- ( ) `chop8 = unfold null (drop 8) (take 8)`
- ( ) `chop8 = unfold (const False) (take 8) (drop 8)`

You have used 0 of 1 submissions

EXERCISE 12 (1 point possible)

Following the previous question, choose an implementation of `map :: (a -> b) -> [a] -> [b]` using `unfold`.

taking into account only finite, non-partial input lists with non-bottom values, and where the mapping function does not return bottom.

- ( ) `map f = unfold null (f) tail`
- ( ) `map f = unfold null (f (head)) tail`

- (selected) `map f = unfold null (f . head) tail`
- `map f = unfold empty (f . head) tail`

For additional understanding, try to experiment with infinite and partial lists and see if you can spot any differences in behaviour for the various implementations.

You have used 0 of 1 submissions

EXERCISE 13 (1 point possible)

Choose an implementation of the Prelude function `iterate :: (a -> a) -> a -> [a]` using `unfold`.

- (selected) `iterate f = unfold (const False) id f`
- `iterate f = unfold (const False) f f`
- `iterate f = unfold (const True) id f`
- `iterate f = unfold (const True) f f`

You have used 0 of 1 submissions

EXERCISE 14 (1 point possible)

Assuming `f`, `g` and `h` are not bottom, the following equality holds for all `f`, `g` and `h` of the correct type:

- `f . f = f`
- `f . g = g . f`
- `f . g = f . h`

- (selected) `f . (g . h) = (f . g) . h`

You have used 0 of 1 submissions

EXERCISE 15 (1 point possible)

Which of the following properties about lists is false:

- `x : xs = [x] ++ xs`
- `[] ++ xs = xs`
- `x : (xs ++ ys) = (x : xs) ++ ys`
- (selected) `[x] : xs = [x, xs]`
- `x : [] = [x]`

You have used 0 of 1 submissions

EXERCISE 16 (1 point possible)

Which of the following properties about `map` and `filter` is true for all `f`, `g` and `p` of the correct type:

- `filter p . map f = map f . filter p`
- `filter p = filter (not . p)`
- (selected) `filter p . filter p = filter p`
- `map f . map g = map g . map f`
- `map f . map f = map f`

You have used 0 of 1 submissions

EXERCISE 17 (1 point possible)

Which of the following is true for all non-bottom `f`, `g` and `p` of the correct type, and finite, non-partial input lists `xs` that contain no bottom values:

- ( ) `reverse xs = xs`
- ( ) `map f (map g xs) = map g (map f xs)`
- ( ) `reverse (reverse xs) = reverse xs`
- (•) `reverse (map f xs) = map f (reverse xs)`
- ( ) `map f (map f xs) = map f xs`

You have used 0 of 1 submissions

EXERCISE 18 (1 point possible)

Which of the following equations is true for all finite, non-partial lists `xs` and `ys`, with non-bottom values:

- ( ) `(reverse xs) ++ ys = ys ++ (reverse xs)`
- ( ) `reverse (xs ++ xs) = xs ++ xs`
- ( ) `reverse (reverse xs) = reverse xs`
- ( ) `xs ++ (reverse ys) = (reverse ys) ++ xs`
- (•) `reverse (xs ++ ys) = reverse ys ++ reverse xs`

You have used 0 of 1 submissions

EXERCISE 19 (1 point possible)

Which of the following expressions produces a finite list:

- ( ) `takeWhile (> 0) [1..]`
- ( ) `dropWhile (< 10) [1..]`
- (•) `take 10 [1..]`
- ( ) `iterate (+1) 0`
- ( ) `filter even [1..]`

You have used 0 of 1 submissions

EXERCISE 20 (1 point possible)

Which of the following statements about the Prelude function `sum :: Num a => [a] -> a` is false:

- (•) sum is a higher-order function
- ( ) sum is defined on the empty list
- ( ) sum is an overloaded function (in the Haskell sense)
- ( ) sum is a polymorphic function (in the Haskell sense)

You have used 0 of 1 submissions

EXERCISE 21 (1 point possible)

Which of the following statements about the Prelude function `map :: (a -> b) -> [a] -> [b]` is false:

- ( ) map is a curried function

- ○ map is a higher-order function
- ○ map is a function with two arguments
- ○ map is a polymorphic function
- ◉ map is an overloaded function

You have used 0 of 1 submissions

EXERCISE 22 (1 point possible)

Which of the following statements about the Prelude function `foldr :: (a -> b -> b) -> b -> [a] -> b` is false:

- ○ foldr is a higher-order function
- ◉ foldr is an overloaded function
- ○ foldr is a curried function
- ○ foldr is a polymorphic function

You have used 0 of 1 submissions

EXERCISE 23 (1 point possible)

Which of the following statements about various Prelude functions is true:

- ○ sum is a higher-order function
- ◉ take is a polymorphic function
- ○ filter is an overloaded function
- ○ length is a curried function

- ○ head can only be defined using recursion

You have used 0 of 1 submissions

EXERCISE 24 (1 point possible)

Which equation defines a function f that is overloaded:

- ○ `f = \x -> x`
- ○ `f (g,x) = g x`
- ○ `f x = x : f x`
- ◉ `f x = x > 3`
- ○ `f xs = xs ++ xs`

You have used 0 of 1 submissions

EXERCISE 25 (1 point possible)

Which of the following expressions is equal to [1, 2, 3, 4]:

- ○ `1:2:3:4`
- ○ `1++2++3++4`
- ○ `map (<= 4) [1..]`
- ○ `filter (<= 4) (repeat 1)`
- ◉ `take 4 (iterate (+1) 1)`

You have used 0 of 1 submissions

EXERCISE 26 (1 point possible)

Evaluating `takeWhile even [2, 4, 5, 6, 7, 8]` gives:

- ( ) `[]`
- ( ) `[2]`
- (x) `[2,4]`
- ( ) `[2,4,6]`
- ( ) `[2,4,6,8]`

You have used 0 of 1 submissions

EXERCISE 27 (1 point possible)

Evaluating `zip [1, 2] ['a', 'b', 'c']` gives:

- ( ) `An error`
- (x) `[(1,'a'),(2,'b')]`
- ( ) `[(1,'a'),(2,'b'),(2,'c')]`
- ( ) `[(1,'a'),(2,'b'),(3,'c')]`
- ( ) `([1,2],['a','b','c'])`

You have used 0 of 1 submissions

EXERCISE 28 (1 point possible)

Evaluating `foldr (-) 0 [1, 2, 3, 4]` gives:

- ( ) −10
- ( ) −8
- (•) −2
- ( ) 0
- ( ) 10

You have used 0 of 1 submissions

EXERCISE 29 (1 point possible)

Evaluating `filter even (map (+1) [1..5])` gives (Note: you can copy and paste this expression directly from edX intro GHCi!):

- ( ) `[]`
- ( ) `[3,5]`
- ( ) `[1,3,5]`
- ( ) `[2,4]`
- (•) `[2,4,6]`

You have used 0 of 1 submissions

EXERCISE 30 (1 point possible)

Which of the following expressions is equal to `filter p (map f xs)`, for all finite, non-partial lists `xs` with no bottom values, and for all non-bottom `f` and `p` of the correct type:

- ( ) `map f (filter p xs)`

- ( ) `f [x|x <- xs, p x]`
- ( ) `[p (f x) | x <- xs]`
- (•) `[f x|x <- xs, p (f x)]`
- ( ) `[f x | x <- xs, p x]`

You have used 0 of 1 submissions

EXERCISE 31 (1 point possible)

After watching the jam session about Church Numerals, what could be a possible implementation for exponentiation? (Note: you have very many attempts to get this question correct)

`cExp :: CNat -> CNat -> CNat`

- (•) `cExp (CNat a) (CNat b) = CNat (a ^ b)`
- ( ) `cExp (CNat a) (CNat b) = CNat (a b)`
- ( ) `cExp (CNat a) (CNat b) = CNat (b a)`
- ( ) `cExp (CNat a) (CNat b) = CNat (a . b)`

You have used 0 of 666 submissions