Functional Programming: Multiple choice questions

Basic questions

Given the following examples, give another possible calculation for the result of `double (double 2)`

```
double (double 2)
=   { applying the inner double }
double (2 + 2)
=   { applying + }
double 4
=   { applying double }
4 + 4
=   { applying + }
8
```

```
double (double 2)
=   { applying the outer double }
(double 2) + (double 2)
=   { applying the first double }
(2 + 2) + (double 2)
=   { applying the first + }
4 + (double 2)
=   { applying double }
4 + (2 + 2)
=   { appplying the second + }
4 + 4
=   { applying + }
8
```

Solution:

- [ ]
```
double (double 2)
= { applying the outer double }
(double 2) + (double 2)
= { applying the second double }
(double 2) + (2 + 2)
= { applying double }
(2 + 2) + (2 + 2)
= { applying + }
8
```

- [x]
```
double (double 2)
= { applying the inner double }
double (2 + 2)
= { applying double }
(2 + 2) + (2 + 2)
= { applying the first + }
4 + (2 + 2)
= { applying the second + }
4+4
= { applying + }
8
```

- [ ]
```
double (double 2)
=   { applying the outer double }
(double 2) + (double 2)
=   { applying the first double }
4 + (double 2)
=   { applying the second double }
4 + 4
=   { applying + }
8
```

- [ ]
```
double (double 2)
=   { applying the inner double }
double 4
=   { applying double }
4 + 4
=   { applying + }
8
```

Which of the following are NOT valid lists in Haskell? Select all options that apply.

Select all that apply

- [ ] `[[]]`
- [ ] `[(1,2),(3,4)]`
- [ ] `[[1,2],[3,4]]`
- [x] `[1,[2,3]]`
- [ ] `[(+),(-),(*)]`
- [ ] `[[1],[2,3],[4]]`
- [x] `[1,[2,3],4]`
- [ ] `[[1,2,3,4]]`
- [ ] `[[1],[2],[3],[4]]`

What is the meaning of the type `a -> b -> c -> d`?

Select one answer

- [ ] `a -> (b -> c) -> d`
- [ ] `a -> ((b -> c) -> d)`
- [ ] `(a -> b) -> (c -> d)`
- [x] `a -> (b -> (c -> d))`
- [ ] `((a -> b) -> c) -> d`

Which of the expressions contains a type error?

Select one answer

- [ ] `1 : [2,3,4]`
- [ ] `[] ++ [1,2,3,4]`
- [ ] `[[1,2]] ++ [[3,4]]`
- [x] `[1,2,3] ++ 4`
- [ ] `1 : 2 : 3 : 4 : []`
- [ ] `1 : [2,3,4]`

The library function `last`, which selects the last element of a non-empty list, can be defined in terms of other library functions introduced in the book. Select all correct definitions.

Select all that apply

- [ ] `last xs = drop (length xs - 1) xs`
- [x] `last xs = head (drop (length xs - 1) xs)`
- [ ] `last xs = tail (reverse xs)`
- [ ] `last xs = reverse (head xs)`
- [x] `last xs = xs !! (length xs - 1)`
- [ ] `last xs = head (drop (length xs) xs)`
- [x] `last xs = head (reverse xs)`
- [ ] `last xs = reverse xs !! (length xs - 1)`

List comphrehension questions:

What does `[(x, y) | x <- [1, 2], y <- [1, 2]]` evaluate to?

Select one answer

- [ ] `([1,2],[1,2])`
- [ ] `[(1,2),(1,2)]`
- [ ] `[(1,1),(2,2)]`
- [x] `[(1,1),(1,2),(2,1),(2,2)]`
- [ ] `[(1,1),(2,1),(1,2),(2,2)]`

What does `[x | x <- [1, 2, 3], y <- [1..x]]` evaluate to?

Select one answer

- [ ] `[]`
- [ ] `[1,2,3]`
- [x] `[1,2,2,3,3,3]`
- [ ] `[1,1,2,1,2,3]`
- [ ] `[1,1,1,2,2,3]`
- [ ] `[1,2,3,2,3,3]`

What does `sum [x | x <- [1..10], even x]` evaluate to?

Select one answer

- [ ] `10`
- [ ] `25`
- [x] `30`
- [ ] `55`
- [ ] An error
- [ ] An infinite loop

List questions:

Which of the following properties about lists is false?

Select one answer

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

Which of the following is true for all values of `f`, `g`, `p`, and `xs` for which the given expressions are well defined?

Select one answer

- [ ] `reverse xs == xs`
- [ ] `map f (map g xs) == map g (map f xs)`
- [ ] `reverse (reverse xs) == reverse xs`
- [x] `reverse (map f xs) == map f (reverse xs)`
- [ ] `map f (map f xs) == map f xs`

Which of the following equations is true for all lists `xs` and `ys` for which the given expressions are well defined?

Select all that apply

- [ ] `(reverse xs) ++ ys == ys ++ (reverse xs)`
- [ ] `reverse (xs ++ xs) == xs ++ xs`
- [ ] `reverse (reverse xs) == reverse xs`
- [ ] `xs ++ (reverse ys) == (reverse ys) ++ xs`
- [x] `reverse (xs ++ ys) == reverse ys ++ reverse xs`
- [ ] `reverse (xs ++ ys) == reverse xs ++ reverse ys`
- [ ] `reverse (xs ++ ys) == ys ++ xs`

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

Select one answer

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

Type Questions:

What is the type of `['a', 'b', 'c']`?

Select all that apply

- [x] [Char]
- [ ] (Char,Char,Char)
- [ ] (Char)
- [ ] [a]
- [x] String
- [ ] [String]

What is the type of `('a', 'b', 'c')`?

Select one answer

- [ ] [Char]
- [x] (Char,Char,Char)
- [ ] (Char)
- [ ] [Char, Char, Char]

What is the type of `[(False,'0'),(True,'1')]`?

Select one answer

- [x] [(Bool,Char)]
- [ ] ([Bool],[Char])
- [ ] [(Bool, Char), (Bool, Char)]
- [ ] [(Bool, String)]

What is the type of `([False,True],['0','1'])`?

Select one answer

- [ ] [(Bool,Char)]
- [x] ([Bool],[Char])
- [ ] ((Bool,Char))
- [ ] [[Bool],[Char]]
- [ ] [[(Bool,Char)]]
- [ ] [Bool,Char]

What is the type of `[tail, init, reverse]`?

Select one answer

- [x] [[a] -> [a]]
- [ ] [a -> a]
- [ ] [a -> [a]]
- [ ] [[a] -> a]
- [ ] [a] -> [a]

What is the type of the function `second xs = head (tail xs)`?

Select one answer

- [ ] [a] -> [a]
- [ ] [a] -> Bool
- [x] [a] -> a
- [ ] Eq a => [a] -> a
- [ ] a -> [a]
- [ ] [a -> a]

What is the most general type of the function `swap (x, y) = (y, x)`?

Select one answer

- [ ] `Tuple a => a -> a`
- [ ] `(a, a) -> (a, a)`
- [ ] `(a, b) -> (a, b)`
- [x] `(a, b) -> (b, a)`
- [ ] `(Int, Bool) -> (Bool, Int)`
- [ ] `(Int, a) -> (a, Int)`
- [ ] `(a, Int) -> (Int, a)`

What is the type of the function `pair x y = (x, y)`?

Select one answer

- [ ] `(a, b) -> (a, b)`
- [ ] `a -> b -> a -> b`
- [ ] `a -> b -> c`
- [x] `a -> b -> (a, b)`
- [ ] `(a, b) -> a -> b`

What is the most general type of the function `double x = x * 2`?

Select one answer

- [x] `Num a => a -> a`
- [ ] `Int a => a -> a`
- [ ] `Int -> Int`
- [ ] `Num -> Num`

What is the most general type of the function `twice f x = f (f x)`?

Select one answer

- [ ] `a -> a -> a -> a`
- [x] `(a -> a) -> a -> a`
- [ ] `a -> (a -> a) -> a`
- [ ] `a -> a -> (a -> a)`

What is the most general type of the function `palindrome xs = reverse xs == xs`?

Select one answer

- [ ] `[a] -> [a]`
- [ ] `String -> Bool`
- [x] `Eq a => [a] -> Bool`
- [ ] `[a] -> a`
- [ ] `[a] -> Bool`
- [ ] `Eq a => [a] -> [a]`
- [ ] `Eq a => [a] -> a`
- [ ] `[Int] -> Bool`

What is the type of the expression `("1,2","3,4")`?

Select one answer

- [ ] `[Int]`
- [ ] `(Int,Int)`
- [ ] `(Int,Int,Int,Int)`
- [x] `(String,String)`
- [ ] `[String]`
- [ ] `String`

What is the type of the expression `["False","True"]`?

Select one answer

- [ ] [a]
- [ ] [Bool]
- [x] [String]
- [ ] [Bool,Bool]
- [ ] [String,String]
- [ ] [Char]

What is the type of the expression `([False, True], False)`?

Select one answer

- [x] `([Bool],Bool)`
- [ ] `([Bool,Bool],Bool)`
- [ ] `(Bool,Bool,Bool)`
- [ ] `([Bool],[Bool])`
- [ ] `((Bool,Bool),Bool)`
- [ ] `[(Bool,Bool)]`

What is the type of the expression `[(1,True), (0,False)]` ?

Select one answer

- [ ] `String`
- [ ] `[(Int,Bool),(Int,Bool)]`
- [ ] `(Int, bool)`
- [ ] `[Int,Bool]`
- [x] `[(Int,Bool)]`
- [ ] `([Int,Bool])`
- [ ] `([Int],[Bool])`

What is the most general type of the function `f xs = take 3 (reverse xs)` ?

Select one answer

- [ ] `[a]`
- [x] `[a] -> [a]`
- [ ] `Int`
- [ ] `Int -> [a]`
- [ ] `Int -> [a] -> [a]`
- [ ] `[Int] -> [Int]`

Higher-order questions:

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

Select one answer

- [ ] `f . f = f`
- [ ] `f . g = g . f`
- [ ] `f . g = f . h`
- [x] `f . (g . h) = (f . g) . h`

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

Select one answer

- [ ] `filter p . map f = map f . filter p`
- [ ] `filter p = filter (not . p)`
- [x] `filter p . filter p = filter p`
- [ ] `map f . map g = map g . map f`
- [ ] `map f . map f = map f`

Which of the following expressions produces a finite list:

Select one answer

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

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

Select one answer

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

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

Select one answer

- [ ] -10
- [ ] -8
- [x] -2
- [ ] 0
- [ ] 10

Evaluating `filter even (map (+1) [1..5])` gives

Select one answer

- [ ] `[]`
- [ ] `[3,5]`
- [ ] `[1,3,5]`
- [ ] `[2,4]`
- [x] `[2,4,6]`

Which of the following expressions is equal to `filter p (map f xs)` for all values of `p`, `f`, and `xs` for which this expression is well-defined?

Select one answer

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

Is it possible for the function type `a -> b` to be an instance of the `Eq` class? In other words, is it possible to implement the function `(==) :: (a -> b) -> (a -> b) -> Bool` that returns `True` if and only if the two given functions always return equal results for equal arguments?

Select one answer

- [ ] Possible for all types `a` and `b`
- [ ] Impossible for all types `a` and `b`
- [x] Possible only for certain combinations of `a` and `b`
- [ ] Possible only for all functions that terminate

Datatype Questions:

The expression `Node (Leaf 1) (Leaf 2)` is a value of which datatype?

Select one answer

- [ ] `data Tree = Node | Leaf | Int`
- [ ] `data Tree = Leaf Int | Node Int Int`
- [ ] `data Tree = Leaf Tree | Node Int Int`
- [x] `data Tree = Leaf Int | Node Tree Tree`
- [ ] `data Tree = Leaf Tree | Node Tree Tree`

The expression `Node (Node Leaf Leaf) Leaf` is a value of which datatype?

Select one answer

- [ ] `data Tree = Node | Leaf`
- [ ] `data Tree = Leaf Tree | Node`
- [x] `data Tree = Leaf | Node Tree Tree`
- [ ]

Given the type declaration `data Tree = Leaf Int | Node Tree Tree`, what can be said about the number of `Leaf` and `Node` constructors in such a tree? Choose the most specific answer that applies.

Select one answer

- [ ] The number of `Leaf` constructors is always equal to the number of `Node` constructors.
- [ ] The number of `Leaf` constructors is always strictly greater than the number of `Node` constructors.
- [ ] The number of `Leaf` constructors is always strictly less than the number of `Node` constructors.
- [x] There is always exactly one more `Leaf` constructor than there are `Node` constructors.
- [ ] There is always exactly one more `Node` constructor than there are `Leaf` constructors.

How many proper values are there in the type `(Bool, Either (Bool,Bool) Bool)` (not counting `undefined` or `error`)?

Select one answer

- [ ] 6
- [ ] 8
- [x] 12
- [ ] 16
- [ ] 36

How many proper values are there in the type `Bool -> Either Bool Bool` (not counting `undefined` or `error`)?

Select one answer

- [ ] 4
- [ ] 6
- [ ] 8
- [x] 16
- [ ] 64

How many proper values are there in the type `Either Ordering (Bool -> Bool)` (not counting `undefined` or `error`)?

Select one answer

- [x] 7
- [ ] 12
- [ ] 27
- [ ] 64
- [ ] 81

Lazy evaluation questions:

How does Haskell evaluate the expression `fst (1+2,3+4)` ?

Select one answer

- [ ] fst (1+2,3+4)
  —>
  fst (3,3+4)
  —>
  fst (3,7)
  —>
  3

- [ ] fst (1+2,3+4)
  —>
  fst (1+2,7)
  —>
  fst (3,7)
  —>
  3

- [x] fst (1+2,3+4)
  —>
  1+2
  —>
  3

- [ ] fst (1+2,3+4)
  —>
  fst (3,3+4)
  —>
  3

Innermost reduction (also known as call-by-value reduction)…

Select all that apply

- [ ] Is the reduction strategy used by Haskell
- [ ] Ensures that evaluation always terminates
- [x] May require fewer steps than outermost reduction
- [ ] Exploits sharing to avoid duplicating work
- [ ] Encourages programming with infinite structures

Outermost reduction (also known as call-by-name reduction)…

Select all that apply

- [ ] Only applies to recursive functions
- [ ] Does not form part of lazy evaluation
- [ ] Prohibits programming with infinite structures
- [ ] Never requires more steps than innermost reduction
- [x] May terminate when innermost reduction does not

Using lazy evaluation (also known as call-by-need reduction)…

Select all that apply

- [x] Makes some programs more efficient than when using outermost reductio
- [ ] Makes all programs terminate
- [ ] Will fully evaluate all programs
- [ ] Requires more reductions than innermost reduction
- [ ] Makes some programs loop that would otherwise terminate

Is it possible to implement a function of type `{A : Set} → List A → Nat → A` in Agda?

Select one answer

- [ ] Yes, this is possible as a total function.
- [ ] Yes, this is possible, but the function will be partial (i.e. it can possibly throw an error)
- [x] No, because any implementation would be rejected by either the coverage checker or termination checker.
- [ ] No, because any implementation would be rejected by the type checker.