

CSc 4980/6980 Functional Programming Practice Exam 1 Questions, Fall 2026

1. Reduce the following expression by applying beta-reductions repeatedly until no more reductions are possible (show work in the form of expression trees or intermediate beta-reductions):

$((\lambda q. (\lambda x. (q\ x))) (\lambda y. (p\ y)))\ j)$

2. Show the result of applying the following substitutions:

$(\lambda y. ((\lambda f. (f\ x))\ y))\ [x := (f\ y)]$

$((f\ (\lambda x. (x\ y))) (\lambda z. ((x\ y)\ z)))\ [x := g]$

3. Using the Lambda Calculus implementation in Python, write a lambda expression, `msplit`, for splitting a list of Church encoded numbers into a pair of two lists of equal lengths (the lengths may differ by a maximum of 1). Then, write a lambda expression, `splitlist`, that will use the `ycomb` operator to split a list. Example of a list is given below:

```
listA = cons(five)(cons(three)(cons(one)(cons(nine)(cons(seven)(nil)))))
```

A sample run is shown below:

```
>>> displayListOfNumbers(listA)
[5, 3, 1, 9, 7]
>>> displayListOfNumbers(first(splitlist(listA)))
[5, 1, 7]
>>> displayListOfNumbers(second(splitlist(listA)))
[3, 9]
```

4. Choose the correct answer:

a) The expression `['a', ['b', 'c']]`:

- (a) has type `[Char]`
- (b) has type `[Char, [Char]]`
- (c) has type `[Char, [Char, Char]]`
- (d) contains a type error

b) Which of the following defines a higher-order function?

- (a) `swap (x,y) = (y,x)`
- (b) `second (x:y:zs) = y`
- (c) `apply f [x,y] = [f x, f y]`
- (d) `sumsqrs xs = sum (map (^2) xs)`

c) Evaluating `filter even (map (*2) [1..5])` gives:

- (a) `[]`
- (b) `[4,8]`
- (c) `[2,6,10]`
- (d) `[2,4,6,8,10]`

5. Implement the following functions **only** with “folds” (if you cannot fit the operator in the fold expression, you may choose to implement the operator with a name and use the name instead).

a) `factorial` takes as input a positive integer, `n`, and returns the factorial of `n`.

```
factorial :: Int -> Int
```

b) `wordValue` takes a word (string with just upper- or lower-case letters) as input and generates a value (number) as output. The word's value is computed by adding the ordinal values of the individual letters in the word, with 'a' having an ordinal value of 1, 'b' having a value of 2, etc. Lower and upper case of the same letter have the same ordinal value. For example: `wordValue "Attitude" = 100`

```
wordValue :: String -> Int
```

c) `minmax`, that takes as input a list of integers and returns a pair (min,max) containing the smallest and the largest number in the list. In case of empty list, return `maxBound::Int` or `minBound::Int` appropriately. For example, `minmax [20,10,80,60,70] = [10,80]`.

```
minmax :: [Int] -> (Int,Int)
```

6. Consider the following function definition:

```
g :: Int -> [Int]
g n = [i | i <- [1..n], n `mod` i == 0]
```

What is the value of `(g 7)`?

What is the value of `(g 12)`?

What does the function `g` do?

7. Two numbers are **amicable** if each is the sum of the distinct proper factors of the other. For example, 220 and 284 are amicable because the factors of 284 are 1, 2, 4, 71 and 142 and these add up to 220 and because the factors of 220 are 1, 2, 4, 5, 10, 11, 20, 22, 44, 55 and 110 and these add up to 284. Write a function `amicable` which takes as input 2 positive integers and returns `True` if the two inputs are amicable and `False` otherwise.

```
amicable :: Int -> Int -> Bool
```

8. What are the values of the following Haskell expressions?

```
foldl (-) 2 [1,7,2,9]
```

```
foldr (-) 0 [1,7,2,9]
```

```
foldr (/) (last [1,2,4,8,16]) (init [1,2,4,8,16])
```

9. Consider the following definitions:

```
second xs = head (tail xs)
```

```
swap (x,y) = (y,x)
```

```
pair x y = (x,y)
```

What are the Haskell types of `second`, `swap`, and `pair`?

10. Suppose we are given the (recursive) definition of a function such as `length`, as follows:

```
length :: [a] -> Int
length [] = 0
length (_:xs) = 1 + length xs
```

The step-by-step evaluation of `length [1,2,3]` would then be:

```
length [1,2,3]
= 1 + length [2,3]
= 1 + (1 + length [3])
= 1 + (1 + (1 + length []))
= 1 + (1 + (1 + 0))
= 3
```

The definition of `init` is as follows:

```
init :: [a] -> [a]
init [] = []
init (x:xs) = x : init xs
```

Show the step-by-step evaluation of `init [1,2,3]`.