

CSc 4980 Functional Programming**Exam I, Fall 2026****12 October 2026****NAME:**

PROBLEM	MAX POINTS	POINTS EARNED
1	10	
2	20	
3	10	
4	20	
5	10	
6	10	
7	20	
8	10	
9	10	
10	10	
TOTAL	130	

Problem 1 (10 Points)

In the lambda expression $(\lambda x. xy) (\lambda y. xy)$, which statement correctly identifies the free occurrences of variables in the bodies of the lambda abstractions?

1. The first y and the second x
2. Both occurrences of the variable x
3. The first x and the second y
4. Both occurrences of the variable y

Which of the following lambda expressions is α -equivalent to $\lambda x. \lambda y. x (yz)$?

1. $\lambda y. \lambda x. x (yz)$
2. $\lambda a. \lambda b. a (bz)$
3. $\lambda z. \lambda y. z (yx)$
4. $\lambda a. \lambda b. a (ba)$

What is the result of performing capture-avoiding substitution $(\lambda y. xy) [x = yz]$?

1. $\lambda y. (yz) y$
2. $\lambda w. (yz) w$
3. $\lambda y. x (yz)$
4. $\lambda w. x (yz)$

Problem 2 (20 Points)

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

$((\lambda x. (\lambda y. (x\ y))) (\lambda z. z))\ a)$

$((\text{lambda } x\ (\text{lambda } y\ (x\ y)))\ (\text{lambda } z\ z))\ a);$

BETA: $((\text{LAMBDA } Y\ ((\text{LAMBDA } Z\ Z)\ Y))\ A)$

BETA: $((\text{LAMBDA } Z\ Z)\ A)$

BETA: A

$((((\lambda x. (\lambda y. (x\ y)))\ (\lambda z. z))\ (\lambda r. (r\ r)))\ c)$

$((((\text{lambda } x\ (\text{lambda } y\ (x\ y)))\ (\text{lambda } z\ z))\ (\text{lambda } r\ (r\ r)))\ c);$

BETA: $((((\text{LAMBDA } Y\ ((\text{LAMBDA } Z\ Z)\ Y))\ (\text{LAMBDA } R\ (R\ R)))\ C)$

BETA: $((((\text{LAMBDA } Z\ Z)\ (\text{LAMBDA } R\ (R\ R)))\ C)$

BETA: $((\text{LAMBDA } R\ (R\ R))\ C)$

BETA: $(C\ C)$

Problem 3 (10 Points)

Show the result of applying the following substitutions:

$$(\lambda x. (\lambda y. ((f\ x)\ y)))\ [y := x]$$
$$(\text{lambda } x\ (\text{lambda } y\ ((f\ x)\ y)))\ [y = x];$$
$$= (\text{lambda } x'\ (\text{lambda } y\ ((f\ x')\ y)))\ [y = x];$$
$$= (\text{lambda } x'\ (\text{lambda } y\ ((f\ x')\ y)))$$
$$((\lambda x. (f\ x)) (\lambda f. (f\ x)))\ [f := (g\ x)]$$
$$((\text{lambda } x\ (f\ x)) (\text{lambda } f\ (f\ x)))\ [f = (g\ x)];$$
$$((\text{lambda } x'\ (f\ x')) (\text{lambda } f\ (f\ x)))\ [f = (g\ x)];$$
$$= ((\text{lambda } x'\ ((g\ x)\ x')) (\text{lambda } f\ (f\ x)))$$
$$(\lambda f. (\lambda y. ((f\ x)\ y)))\ [x := (f\ y)]$$
$$(\text{lambda } f\ (\text{lambda } y\ ((f\ x)\ y)))\ [x = (f\ y)];$$
$$(\text{lambda } f\ (\text{lambda } y'\ ((f\ x)\ y')))\ [x = (f\ y)];$$
$$= (\text{lambda } f'\ (\text{lambda } y'\ ((f'\ (f\ y))\ y')))$$

Problem 4 (20 points)

Using the Lambda Calculus in Python library, write Lambda expressions for the `map` and `filter` higher order functions. As usual, you will build two lambda expressions for each with the second one employing `ycomb` to achieve recursion.

```
# map list of numbers using function

map_step = lambda f: lambda g: lambda xs: \
    isempty(xs) \
        (lambda _: nil) \
        (lambda _: cons(g(head(xs))) (f(g) (tail(xs)))) (zero)

mapp = lambda g: lambda xs: ycomb(map_step) (g) (xs)

# filter list of numbers using predicate

filter_step = lambda f: lambda g: lambda xs: \
    isempty(xs) \
        (lambda _: nil) \
        (g(head(xs)) \
            (lambda _: cons(head(xs)) (f(g) (tail(xs)))) \
            (lambda _: f(g) (tail(xs)))) \
        ) (zero)

filterr = lambda g: lambda xs: ycomb(filter_step) (g) (xs)
```

A sample run is shown below:

```
>>> displayListOfNumbers(mylist1)
[1, 2, 3, 4, 5]
>>> displayListOfNumbers(mapp(lambda x: poww(x) (church(2))) (mylist1))
[1, 4, 9, 16, 25]
>>> displayListOfNumbers(filterr(lambda x: neq(mod(x) (two)) (zero)) (mylist1))
[1, 3, 5]
```

Problem 5 (10 points)

Choose the correct answer:

(1) A function in Haskell is polymorphic if:

- (a) Its type contains one or more type variables
- (b) It takes multiple arguments one at a time
- (c) It takes a function as an argument
- (d) It returns a tuple type

(2) The function `inc x = x+1` has type:

- (a) `a -> a`
- (b) `int -> int`
- (c) `Num -> Num`
- (d) `Num a => a -> a`

(3) Suppose we have the following (higher order) functions:

```
twice f x = f (f x)
thrice f x = f (f (f x))
```

What do the following expressions evaluate to? Show the simplification steps.

```
twice thrice (+1) 0
= (((twice thrice) (+1)) 0)
= (thrice (thrice (+1)) 0)
= (thrice (+3) 0)
= ((+9) 0)
= 9
```

```
thrice twice (+1)
= (((thrice twice) (+1)) 0)
= ((twice (twice (twice (+1)))) 0)
= ((twice (twice (+2))) 0)
= ((twice (+4)) 0)
= ((+8) 0)
= 8
```

Problem 6 (10 Points)

Give a definition of the function, called `disjoint`, that takes two lists in ascending order and determines whether they have an element in common.

```
disjoint :: (Ord a) => [a] -> [a] -> Bool

disjoint [] _ = True

disjoint _ [] = True

disjoint (x:xs) (y:ys)
    | x == y = False
    | x < y = disjoint xs (y:ys)
    | x > y = disjoint (x:xs) ys
```

Problem 7 (20 Points)

Eratosthenes' sieve is an ancient method for finding prime numbers. Start by writing out all the numbers from 2 to (say) 100. The first number (2) is prime. Now cross out all multiples of 2. The first remaining number (3) is also prime. Cross out all multiples of 3. The first remaining number (5) is also prime... and so on. When no numbers remain, you have found all the prime numbers in the range you started with.

Complete the following function definitions (all of these are one or two liners!):

```
-- isPrime returns True if input positive integer is prime; False otherwise
isPrime :: Int -> Bool
isPrime n = all (\d -> n `mod` d /= 0) [2.. (n `div` 2 + 1)]

-- crossout takes as parameters an integer m and a list of integers ns and returns a subset of ns by
-- removing from ns all multiples of m.
-- e.g. crossout 3 [3,4,5,6,7,8,9,10] = [4,5,7,8,10]
crossout :: Int -> [Int] -> [Int]
crossout n = filter (\d -> d `mod` n /= 0)

-- sieve returns a list of primes in the input list ns. ns can be assumed to be initialized to [2..n] given n.
-- e.g. for n=10, sieve [2,3,4,5,6,7,8,9,10] = [2,3,5,7]
sieve :: [Int] -> [Int]
sieve [] = []
sieve (x:xs) = x:sieve (crossout x xs)

-- isSumOfTwoPrimes returns True if input n is a sum of two primes, False otherwise.
-- e.g. isSumOfTwoPrimes 11 = False (because 11 is 1+10, 2+9, 3+8, 4+7, 5+6 and none of the sums involve
-- two primes) and isSumOfTwoPrimes 45 = True (because 45 = 2 + 43)
isSumOfTwoPrimes :: Int -> Bool
isSumOfTwoPrimes n = or [x+y == n | x <- sieve [2..n], y <- sieve [2..n]]

-- Write a Haskell expression that evaluates to True if all even numbers between 4 and 100 inclusive can
-- be expressed as sum of two primes and false otherwise
even4To100SumOfPrimes :: Boolean
even4To100SumOfPrimes = all isSumOfTwoPrimes (filter (\d -> d `mod` 2 == 0) [4..100])
```

Problem 8 (10 Points)

Implement the following functions **only** with “folds” (if you cannot fit the operator in the fold, you may give it a name and use the name in the fold).

- (a) `longest`, which takes as input a non-empty list of strings and returns the longest string in the list. In case of tie for longest length, you may return any one of the strings.

For example, `longest ["al", "errol", "christina", "sachin"] = "christina"`.

```
longest :: [String] -> String
```

```
longest =
```

```
    foldr (\x acc -> if length x > length acc then x else acc) ""
```

- (b) `rmDups` removes adjacent duplicates from a list. For example,

```
rmDups [1, 2, 2, 3, 3, 3, 1, 1] = [1, 2, 3, 1].
```

```
rmDups :: [Int] -> [Int]
```

```
rmDups = foldr (\x acc -> if null acc then [x]
```

```
                    else if x == head acc then acc
```

```
                    else x:acc ) []
```

Nansw2navd5

Problem 9 (10 Points)

(a) Consider the following function definition:

```
f :: Ord a => [a] -> [a]
f xs
  | null xs    = xs
  | otherwise = head xs : f [x | x <- tail xs, x /= head xs]
```

What is the value of $(f\ [1,2,1,3,1,4,2])$?

$f\ [1,2,1,3,1,4,2] = [1,2,3,4]$

What does the function f do?

f removes duplicates from the input list

Problem 10 (10 Points)

Using `map` and `filter`, define a function `wiv` (without internal vowels) which takes a list of strings as its argument and removes every occurrence of a vowel from each string. You can use the built-in function `notElem` that takes two parameters, an item and a list of items and returns `True` if item is not in list of items and `False` otherwise. For example:

```
wiv ["The", "End", "Is", "nigh"] = ["Th","nd","s","ngh"]
```

```
wiv :: [String] -> [String]
wiv = map (filter (\x -> notElem x "aeiouAEIOU"))
```