CMPS 112, Spring 2019 Midterm (Solutions)

Section	**Points**	**Score**
Reductions	10 points	
Lists	15 points	
Snoc lists	20 points	
Matrices	50 points	
Total	95 points	

Instructions

- **You have 95 minutes to complete this exam.**
- This exam is **closed book**. You may use one double-sided page of notes, but no other materials.
- Where space is provided for answers, use only the space provided (or as close as possible). Only the provided space will be considered for grading.
- Questions marked with * are difficult; we recommend solving them last.
- Avoid seeing anyone else's work or allowing yours to be seen.
- Do not communicate with anyone but an exam proctor.
- To ensure fairness (and the appearance thereof), **proctors will not answer questions about the content of the exam**. If you are unsure of how to interpret a problem description, state your interpretation clearly and concisely. *Reasonable interpretations* will be taken into account by graders.
- Good luck! Remember that a good score on the final can replace a poor midterm grade. So take a breath, do your best, and show me what you know.

Part I: Lambda Calculus

1 Reductions [10 pts]

For each λ-term below, circle all valid reductions of that term. It is possible that none, some, or all of the listed reductions are valid. Reminder:

- `=a>` stands for an α-step (α-renaming)
- `=b>` stands for a β-step (β-reduction)
- `=*>` stands for a sequence of zero or more steps, where each step is either an α-step or a β-step

1.1 [5 pts]

```
(\x y z -> x z (y z)) (\a b -> a) c d
```

[✓ **(A)** `=*> d`]

(B) `=*> (c d)`

(C) `=*> (\x y -> x) z (y z)`

(D) `=a> (\x y z -> x z (y z)) (\x y -> x) x y`

(E) `=*> (\x y -> x) c d`

1.2 [5 pts]

```
(\x -> \y -> y) ((\x -> x x) (\x -> x x))
```

[✓ **(A)** `=b> \y -> y`]

(B) `=b> (\y -> y y)`

(C) `=b> (\x -> (\y -> y y))`

(D) `=b> (\x -> x x) (\x -> x x)`

[✓ **(E)** `=b> (\x -> \y -> y) ((\x -> x x) (\x -> x x))`]

2 Lists [15 pts]

One way of encoding lists in λ-calculus is to use `FALSE` as the empty list, and nested pairs for non-empty lists where the head of the list is the first element of the pair, and the tail of the list is the second element. For example, the Haskell list $[1, 2, 3]$, could be represented as

```
PAIR ONE (PAIR TWO (PAIR THREE FALSE))
```

Helpful list functions: The following function, `EMPTY`, returns `TRUE` when applied to an empty list, and `FALSE` otherwise.

```
let EMPTY = \xs -> xs (\x y z -> FALSE) TRUE
```

The `EMPTY` function is useful for defining the base case of recursive functions on lists. Recall that we can define recursive functions in λ-calculus using the fixed point combinator `FIX`:

```
let FIX = \stp -> (\x -> stp (x x)) (\x -> stp (x x))
```

For example, we can define the function `APPEND` that uses recursion to append two lists:

```
let APPEND = FIX (\rec l1 l2 ->
                   ITE (EMPTY l1)
                     l2
                     (PAIR (FST l1) (rec (SND l1) l2)))
```

`APPEND` passes the following test case:

```
eval append:
  APPEND (PAIR ONE FALSE) (PAIR TWO FALSE)
  =~> (PAIR ONE (PAIR TWO FALSE))
```

See the "Lambda Calculus Cheat Sheet" at the end of the exam for definitions of variables. You may use any of these functions (as well as the functions you define yourself) for the problems below.

2.1 Get [5 pts]

Implement the function `GET`. Given a non-empty list `l` and a church numeral `i`, `GET l i` returns the `i`th element of list `l`. **Hint:** *You do not need recursion to implement* `GET`*.*

```
let GET = \l i -> FST (i SND l)
```

Your implementation must pass the following test case:

```
eval get1:
  GET (PAIR ONE (PAIR TWO (PAIR THREE FALSE))) ONE
  =~> TWO
```

2.2 Reverse [10pts]

Now implement the function `REVERSE`. Given a list `l`, `REVERSE` returns a list with the elements of `l` in reverse order.

```
let REVERSE = FIX (\rec l -> ITE (EMPTY l) FALSE
         (APPEND (rec (SND l)) (PAIR (FST l) FALSE)))
```

Your implementation must pass the following test cases:

```
eval reverse1:
  REVERSE (PAIR ONE (PAIR TWO (PAIR THREE FALSE)))
  =~> PAIR THREE (PAIR TWO (PAIR ONE FALSE))

eval reverse2:
  REVERSE FALSE
  =~> FALSE
```

3 Snoc lists [20 pts]

The lists in the previous problems are sometimes called "cons lists" after the Lisp keyword for constructing pairs. (Haskell's `List` type also represents lists this way.) An alternative representation of a list also uses `FALSE` as the empty list and pairs for non-empty lists, but uses the first element to represent the front of the list and the second element to represent the last element of the list. These lists are sometimes called "snoc lists" since the pairs are nested differently from cons lists. For example, the Haskell list $[1, 2, 3]$, would be represented as

```
PAIR (PAIR (PAIR FALSE ONE) TWO) THREE
```

Notice that the elements of the list are *not* reversed, just how the pairs are nested has changed.

3.1 S_GET [5 pts] EXTRA CREDIT

Implement the function `S_GET` for snoc lists that corresponds to `GET` for cons lists.

```
let S_GET = \l i -> GET (CONVERT l) i
```

Your implementation must pass the following test cases.

```
eval snoc_get1:
  S_GET (PAIR (PAIR (PAIR FALSE ONE) TWO) THREE) ONE
  =~> TWO
```

3.2 CONVERT* [15 pts]

Now implement `CONVERT`, which converts a snoc list to a cons list. You may use the extra space below to define a helper function if desired. Hint: `EMPTY` has the same behavior for cons or snoc lists.

```
let SWAP = FIX (\rec l -> ITE (EMPTY l) FALSE
                       (PAIR (SND l) (rec (FST l))))
let CONVERT = \l -> REV (SWAP l)
```

Your implementation should satisfy the following test cases

```
eval convert1:
  CONVERT (PAIR (PAIR (PAIR FALSE ONE) TWO) THREE)
  =~> PAIR ONE (PAIR TWO (PAIR THREE FALSE))

eval convert2:
  CONVERT FALSE
  =~> FALSE
```

Part II: Recursive data structures

For this part, you may use any Haskell functions you wish to solve the following problems, but only basic arithmetic, boolean, and list operations (e.g., `length` and `(:)`) are necessary.

4 Matrices [50 pts]

Matrices can be represented (inefficiently) in Haskell using nested lists.[1] For example, each row of a matrix is stored as a list, and each row is stored as an element of a list. In this scheme, the matrix

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

is represented as the Haskell list

```
[ [1,2], [3,4] ]
```

[1]The module `Data.Matrix` has a more efficient representation.

4.1 isMatrix [10 pts]

A drawback of this representation is that it is possible to construct an invalid matrix where the rows have different lengths. Implement the function `isMatrix` to check that all rows of a matrix are equal.

```
isMatrix :: [[a]] -> Bool
isMatrix [] = True
isMatrix (r:rs) = check (length r) rows
  where
    check n [] = True
    check n (r:rs) = (length r == n) && check n rs
```

Your implementation must pass the following test cases.

```
isMatrix []
  ==> True

isMatrix [ [1,2], [3,4], [5,6] ]
  ==> True

isMatrix [ [1,2,3], [4,5] ]
  ==> False
```

4.2 diagonal [20 pts]

The diagonal of a matrix is the list of matrix elements where the index of the column and row are equal. For example, the diagonal of the matrix

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

is $[1, 4]$ since 1 is in (0,0) position and 4 is in the (1,1) position. Implement the function `diagonal` to return the diagonal of a matrix.

```
diagonal :: [[a]] -> [a]
diagonal (r:rs) | isMatrix (r:rs) =
    diag (length r) (r:rs)
  where
    get (x:xs) i = if i == 0 then x else get xs (i-1)
    diag n _ [] = []
    diag n m (r:rs) | n >= m = []
    diag n m (r:rs) = (get r n) : (diag (n+1) rs)
diagonal _ = []
```

Your implementation must pass the following test cases.

```
diagonal [ [1,2], [3,4] ]
  ==>[1,4]

diagonal [ [1,2,3], [4,5,6] ]
  ==>[1,5]

diagonal [ [1,2], [3,4], [5,6] ]
  ==> [1,4]
```

4.3 max [20 pts]

Implement the function `max` that returns the maximum element of an `Int` matrix or 0 if the argument is not a valid matrix.

```
max :: [[Int]] -> Int
max [] = 0
max mat | isMatrix mat = max' 0 mat
  where
    rowmax m [] = m
    rowmax m (x:xs) | m >= x = rowmax m xs
    rowmax m (x:xs) = rowmax x xs
    max' m [] = m
    max' m (r:rs) | m >= rowmax m r = max' m rs
    max' m (r:rs) = max' (rowmax m r) rs
max mat = 0
```

Your implementation must pass the following test cases.

```
max [ [1,2], [3,4] ]
  ==> 4

max []
  ==> 0

max [[1,2], [3,4,5]]
  ==> 0
```

```
-- Booleans ---------------------------------
let TRUE =\x y -> x
let FALSE = \x y -> y
let ITE = \b x y -> b x y
let NOT = \b x y -> b y x
let AND = \b1 b2 -> ITE b1 b2 FALSE
let OR = \b1 b2 -> ITE b1 TRUE b2

-- Pairs ------------------------------------
let PAIR = \x y b -> b x y
let FST = \p -> p TRUE
let SND = \p -> p FALSE

-- Recursion --------------------------------
let FIX = \stp -> (\x -> stp (x x)) (\x -> stp (x x))

-- Lists ------------------------------------
let EMPTY = \xs -> xs (\x y z -> FALSE) TRUE
let APPEND = FIX (\rec l1 l2 ->
                   ITE (EMPTY l1)
                     l2
                     (PAIR (FST l1) (rec (SND l1) l2)))

-- Numbers ----------------------------------
let ZERO = \f x-> x
let ONE = \f x -> f x
let TWO = \f x -> f (f x)
let THREE = \f x -> f (f (f x))

-- Arithmetic -------------------------------
let INC = \n f x -> f (n f x)
let ADD = \n m -> n INC m
let MUL = \n m -> n (ADD m) ZERO
let ISZ = \n -> n (\z -> FALSE) TRUE
```