CSE 130 Midterm, Spring 2018

Nadia Polikarpova

May 4, 2018

NAME ______________________________

SID ______________________________

- You have **50 minutes** to complete this exam.
- Where limits are given, **write no more** than the amount specified.
- You may refer to a **double-sided cheat sheet**, but no electronic materials.
- 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.
- If you have a question, raise your hand.
- **Good luck!**

Part I. Lambda Calculus [60 pts]

Q1: Reductions [20 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 -> x (\x -> x)) (f x)
```

(A) `=b> f x (\x -> x)`

(B) `=b> f x (\x -> f x)`

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

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

(E) `=a> (\x -> x (\y -> y)) (f x)`

1.2 [5 pts]

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

(A) `=a> \x -> (\y x -> x x) (\x -> x)`

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

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

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

(E) `=b> \y z -> (\x -> x) y`

1.3 [5 pts]

```
(\f g x -> f (g x)) (\x -> g x) (\z -> z)
```

(A) `=b> (\f g x -> f (g x)) (g (\z -> z))`

(B) `=b> (\g x -> (\x -> g x) (g x)) (\z -> z)`

(C) `=*> \x -> g x`

(D) `=*> \y -> g y`

(E) `=*> \x -> f x`

1.4 [5 pts]

```
(\x y -> \b u v -> b v u) (\x y -> y) (\x y -> y) (\x y -> x)
```

(A) `=b> (\y -> \b u v -> b v u) (\x y -> y) (\x y -> x)`

(B) `=b> (\y -> \b u v -> b v u) (\x y -> y) (\x y -> y)`

(C) `=*> (\b u v -> b v u) (\x y -> x)`

(D) `=*> \x y -> y`

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

Q2: Lists [40 pts]

We can encode lists in λ-calculus by representing

- an *empty* list as `FALSE`
- a *non-empty* list as a `PAIR` of its head and tail

For example, the list `[1,2,3]` would be represented as:

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

You can find the definitions of all variables used in this question in the "Lambda Calculus Cheat Sheet" at the end of this exam.

2.1 Repeat [10 pts]

Implement the function `REPEAT`, which, given a Church numeral `n` and any value `x`, returns a list with `n` copies of `x`. You can use any function defined in the "Lambda Calculus Cheat Sheet".

```
let REPEAT =  ___________________________________________________
```

The function should satisfy the following test cases:

```
eval repeat1 :
  REPEAT ZERO x
  =~> FALSE

eval repeat2 :
  REPEAT TWO ONE
  =~> PAIR ONE (PAIR ONE FALSE)
```

2.2 Empty* [20 pts]

Implement the function `EMPTY`, which, given a list represented as defined above, determines whether the list is empty. You can use any function defined in the "Lambda Calculus Cheat Sheet".

```
let EMPTY =  ___________________________________________________
```

The function should satisfy the following test cases:

```
eval empty1 :
  EMPTY FALSE
  =~> TRUE

eval repeat2 :
  EMPTY (PAIR ZERO FALSE)
  =~> FALSE
```

2.3 Length [10 pts]

Recall that recursion can be implemented in λ-calculus using *fixpoint* combinators like this one:

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

Using the fixpoint combinator, implement the function `LEN`, which, given a list represented as defined above, computes its length. You can use `FIX`, `EMPTY` from 2.2, *and* any function defined in the "Lambda Calculus Cheat Sheet".

```
let LEN =  _______________________________________________________
```

The function should satisfy the following test cases:

```
eval len1 :
  LEN FALSE
  =~> ZERO

eval repeat2 :
  LEN (PAIR ONE (PAIR TWO FALSE))
  =~> TWO
```

Part II. Datatypes and Recursion [50 pts]

Q3: Binary Search Trees [50 pts]

Recall that a **binary search tree** (BST) is a binary tree, where every node stores an integer *key* `x`, and the keys are *ordered* such that all keys in the node's left sub-tree are *strictly less* than `x`, and all keys in the node's right sub-tree are *strictly greater* than `x`. An example BST is depicted below.

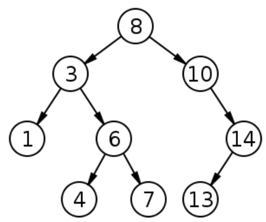

Figure 1: Binary Search Tree

In this question, you will implement several Haskell functions that operate on BSTs. We will represent BSTs using the following datatype:

```
data Tree = Empty | Node Int Tree Tree
```

In your implementations, you can use any library functions on integers (arithmetic operators and comparisons), and the append function on lists `(++)`.

3.1 Size [5 pts]

Implement the fuction `size` that computes the size of a tree.

```
size :: Tree -> Int
```

```
---------------------------------------------------------------

---------------------------------------------------------------

---------------------------------------------------------------
```

Your implementation must satisfy the following test cases

```
size Empty
  ==> 0
size (Node 8 (Node 3 Empty Empty) (Node 10 Empty Empty))
  ==> 3
```

3.2 Insert [10 pts]

Implement the fuction `insert` that inserts a key into a BST. More precisely, `insert x t` may *assume* that `t` is a BST (i.e., elements are ordered as defined above) and must *ensure* that its result is a BST that contains a key `x`.

```
insert :: Int -> Tree -> Tree
```

```
----------------------------------------------------------------

----------------------------------------------------------------

----------------------------------------------------------------

----------------------------------------------------------------

----------------------------------------------------------------

----------------------------------------------------------------
```

Your implementation must satisfy the following test cases

```
insert 8 Empty
  ==> Node 8 Empty Empty
insert 6 (Node 8 (Node 3 Empty Empty) (Node 10 Empty Empty))
  ==> Node 8
        (Node 3 Empty (Node 6 Empty Empty))
        (Node 10 Empty Empty)
```

3.3 Sort [15 pts]

Implement the helper fuctions `fromList` and `toList` so that the `sort` function below sorts a list of integers (you can assume the input list has no duplcate elements). You implementation can use the function `insert` from question 3.2.

```
sort :: [Int] -> [Int]
sort xs = toList (fromList xs)
  where
    fromList :: [Int] -> Tree

    ---------------------------------------------------------

    ---------------------------------------------------------

    ---------------------------------------------------------

    toList :: Tree -> [Int]

    ---------------------------------------------------------

    ---------------------------------------------------------

    ---------------------------------------------------------
```

3.4 Tail-recursive size* [20 pts]

Re-implement the function `size` from question 3.1 so that it's *tail-recursive.*

Hint: introduce an auxiliary function with extra arguments that keep track of what's already done and what is still left to do.

```
size :: Tree -> Int

----------------------------------------------------------------

----------------------------------------------------------------

----------------------------------------------------------------

----------------------------------------------------------------

----------------------------------------------------------------

----------------------------------------------------------------

----------------------------------------------------------------
```

Lambda Calculus Cheat Sheet

Here is a list of definitions you may find useful for Q2

```
-- 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


-- 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
```