# CSE 130 Midterm Solution, Spring 2018

Nadia Polikarpova

May 4, 2018

## Part I. Lambda Calculus [60 pts]

### Q1: Reductions [20 pts]

#### 1.1 [5 pts]

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

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

(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)` **valid**

#### 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)` **valid**

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

(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` **valid**

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

(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)` **valid**

(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)` **valid**

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

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

## Q2: Lists [40 pts]

### 2.1 Repeat [10 pts]

```
let REPEAT =  \n x -> n (PAIR x) FALSE
```

### 2.2 Empty* [20 pts]

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

Alternatively:

```
let EMPTY = \xs -> xs (\x y -> NOT) TRUE
```

### 2.3 Length [10 pts]

```
let LEN = FIX (\rec l -> ITE (EMPTY l) ZERO (INC (rec (SND l))))
```

# Part II. Datatypes and Recursion [50 pts]

## Q3: Binary Search Trees [50 pts]

### 3.1 Size [5 pts]

```
size :: Tree -> Int
size Empty = 0
size (Node _ l r) = 1 + size l + size r
```

### 3.2 Insert [10 pts]

```
insert :: Int -> Tree -> Tree
insert x Empty = Node x Empty Empty
insert x (Node y l r)
  | x == y    = Node y l r
  | x < y     = Node y (insert x l) r
  | otherwise = Node y l (insert x r)
```

### 3.3 Sort [15 pts]

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

```haskell
    fromList (x:xs) = insert x (fromList xs)

    toList :: Tree -> [Int]
    toList Empty = []
    toList (Node x l r) = toList l ++ [x] ++ toList r
```

### 3.4 Tail-recursive size* [20 pts]

```haskell
size :: Tree -> Int
size t = loop 0 [] t
  where
    loop :: Int -> [Tree] -> Tree -> Int
    loop acc []     Empty        = acc
    loop acc (t:ts) Empty        = loop acc       ts     t
    loop acc ts     (Node _ l r) = loop (acc + 1) (r:ts) l
```

Alternatively:

```haskell
size :: Tree -> Int
size t = loop 0 Empty t
  where
    loop :: Int -> Tree -> Tree -> Int
    loop acc Empty        Empty        = acc
    loop acc (Node _ l r) Empty        = loop (acc + 1) l            r
    loop acc t            (Node x l r) = loop acc       (Node x t r) l
```