CSE 130 Midterm, Fall 2019

Nadia Polikarpova

Nov 1, 2019

NAME ______________________________

SID ______________________________

- DO NOT TURN THIS PAGE OVER BEFORE WE TELL YOU TO
- 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.
- 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 [20 pts]

Q1: Reductions [10 pts]

Note: the correct answer might have none, some, or all of the boxes checked.

1.1 [5 pts]

Check the box next to **each** term that contains **exactly one** redex (i.e. there is one and only one way to apply a beta step to this term).

```
(A) (\x -> x) (\x -> x)                     [ ]
(B) \x -> x (\x -> x)                       [ ]
(C) f (\x -> x) (\x -> x)                   [ ]
(D) (\x -> x) f (\x -> x)                   [ ]
(E) (\f x -> f (f x)) y z                   [ ]
```

1.2 [5 pts]

Check the box next to **each** valid reduction.

Reminder:

- `=a>` stands for an α-step (α-renaming)
- `=b>` stands for a β-step (β-reduction)

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

(A) =b> (\x y -> (\y -> y) x) apple banana       [ ]
(B) =a> (\x y -> (\x x -> x) y x) apple banana   [ ]
(C) =a> (\x y -> (\x z -> x) y x) apple banana   [ ]
(D) =b> (\y -> (\x y -> x) y apple) banana       [ ]
(E) =b> (\y -> (\x y -> apple) y apple) banana   [ ]
```

Q2: 2048 [10 pts]

Write a lambda term that evaluates to the Church representation of the number 2048.

Your term has to fit on **one line** and be a complete, syntactically valid lambda term without abbreviations. You **cannot** define new names, but you **can** use any function defined in the "Lambda Calculus Cheat Sheet" at the end of this exam.

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

Part II. Datatypes and Recursion [30 pts]

Q3: Reverse Polish Notation [30 pts]

Imagine you are implementing an arithmetic calculator in Haskell, and you are using a datatype `Expr` to represent expressions and an auxiliary datatype `Op` to represent binary operations:

```
data Op = Plus | Minus | Times

data Expr = Num Float         -- floating point constant
          | Bin Op Expr Expr  -- binary operation
```

Recall that *reverse Polish notation* (RPN), also known *postfix notation*, is a mathematical notation in which operators follow their operands. Its main advantage is that you can write any arithmetic expression in RPN unambiguously without any parentheses. For example, here are some expressions and their RPN versions:

```
Expression                   RPN

3 + 5                        3 5 +
(3 + 5) * 2                  3 5 + 2 *
3 + 5 * 2                    3 5 2 * +
3 + 2 * (5 - 4)              3 2 5 4 - * +
```

We can represent an RPN expression in Haskell as a *list of tokens*, where each token is either an operand or an operation:

```
data Token = Operand Float | Operation Op
```

For example, the RPN expression `3 5 +` is represented as

```
[Operand 3, Operand 5, Operation Plus]
```

Some lists do not represent valid RPN expressions, e.g. the following list is an *invalid* RPN:

```
[Operand 5, Operation Plus]
```

In your solutions you **can use** list constructors (`[]` and `(:)`), bracket notation (e.g. `[1, 2, 3]`) and the list append function (`(++)`).

3.1 To RPN [10 pts]

Implement the function `toRPN` that converts an expression into RPN.

```
toRPN :: Expr -> [Token]
```

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

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

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

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

3.2 From RPN [20 pts]

Implement the function `fromRPN` that converts from RPN into an expression. Your function must be **tail-recursive**. If given an invalid RPN as input, your function **must throw an error** using `error "Invalid RPN"`.

Hint: Introduce an auxiliary function with an extra parameter that stores all current top-level expressions (i.e. expressions that are still waiting for their parent binary operation).

```
fromRPN :: [Token] -> Expr
```

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

-- 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))
let FOUR  = \f x -> f (f (f (f x)))
let FIVE  = \f x -> f (f (f (f (f x))))
let SIX   = \f x -> f (f (f (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
let SKIP1 = \f p -> PAIR TRUE (ITE (FST p) (f (SND p)) (SND p))
let DEC   = \n   -> SND (n (SKIP1 INC) (PAIR FALSE ZERO))
```