CSE 130 Midterm, Winter 2019

Nadia Polikarpova

Feb 11, 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 + 5 extra]

Q1: Reductions [10 pts]

For each λ-term below, check the box next to **each** valid reduction 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, and the right-hand side is in *normal form*

1.1 [5 pts]

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

(A) `=b> \x y -> y x` [ ]

(B) `=b> \z x -> x z` [ ]

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

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

(E) `=a> \x y -> (\z y -> y z) (x y)` [ ]

1.2 [5 pts]

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

(A) `=b> (\x -> x) (apple (\z -> z))` [ ]

(B) `=b> (\y -> apple y) (\z -> z)` [ ]

(C) `=a> (\z -> z) (\y -> apple y) (\z -> z)` [ ]

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

(E) `=~> apple (\z -> z)` [ ]

Q2: Factorial [10 pts + 5 extra]

In this task you will implement the *factorial function* in lambda calculus. Your implementation of `FACT` should satisfy the following test cases:

```
eval fact0 :     eval fact1 :     eval fact2 :     eval fact3 :
  FACT ZERO        FACT ONE         FACT TWO         FACT THREE
  =~> ONE          =~> ONE          =~> TWO          =~> SIX
```

You can use any function defined in the "Lambda Calculus Cheat Sheet" at the end of this exam, including the fixpoint combinator `FIX`. You are allowed (but not required) to define a helper function `STEP`.

You will get **5 extra points** if your implementation *does not* use the fixpoint combinator. *Hint:* to implement this version, define the helper function `STEP` that takes in a pair, similarly to `SKIP1` from the homework.

```
let STEP = ______________________________________________________

           ______________________________________________________

let FACT = ______________________________________________________

           ______________________________________________________
```

Part II. Datatypes and Higher-Order Functions [30 pts]

Q3: Files and Directories [30 pts]

We can represent a directory structure using the following Haskell datatype:

```
data Entry =
    File String Int       -- file: name and size
  | Dir String [Entry]    -- directory: name and child entries
```

For example, the value:

```
homedir = Dir "home"
            [ File "todo" 256
            , Dir  "HW0" [ File "Makefile" 575 ]
            , Dir  "HW1" [ File "Makefile" 845, File "HW1.hs" 3007]
            ]
```

represents the following directory structure:

```
home
 |---todo (256 bytes)
 |
 |---HW0
 |     |---Makefile (575  bytes)
 |
 |---HW1
       |---Makefile (845  bytes)
       |---HW1.hs   (3007 bytes)
```

In your solutions you can use any library functions on integers (e.g. arithmetic operators), but **only the following** functions on lists:

```
(==)   :: String -> String -> Bool       -- equality on strings
(++)   :: [a] -> [a] -> [a]              -- append on any lists
map    :: (a -> b) -> [a] -> [b]
filter :: (a -> Bool) -> [a] -> [a]
foldr  :: (a -> b -> b) -> b -> [a] -> b
foldl  :: (b -> a -> b) -> b -> [a] -> b
```

3.1 Size [10 pts]

Implement the function `size` that computes the total size of an entry in bytes. You are *allowed* to introduce a helper function using a `where` clause, although we encourage you to use a higher-order function instead.

Your implementation must satisfy the following test cases

```
size (File "todo" 256)
  ==> 256
size (Dir "haskell-jokes" [])
  ==> 0
size homedir
  ==> 4683    -- 256 + 575 + 845 + 3007
```

```
size :: Entry -> Int
```

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

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

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

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

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

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

3.2 Find [20 pts]

Implement the function `find` that finds all *files* with a given name inside a given entry. More precisely, `find path e f` finds all files with name `f` inside the entry `e`, where `path` is the full path to `e`, and returns the list of full paths to those files.

Your implementation must satisfy the following test cases

```
find "./home" (File "todo" 256) "todo"
  ==> ["./home/todo"]
find "./home" (Dir "todo" []) "todo"
  ==> []
find "." homedir "Makefile"
  ==> ["./home/HW1/Makefile", "./home/HW0/Makefile"]
      -- ^ order is irrelevant
```

You are **not allowed** to introduce recursive helper functions and **must use** higher-order functions instead.

```
find :: String -> Entry -> String -> [String]

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

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

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

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

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

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

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

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

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

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))

-- Recursion ------------------------------

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