CSE 130 Final, Fall 2019

Nadia Polikarpova

December 12, 2019

NAME _____

SID _____

- DO NOT TURN THIS PAGE OVER BEFORE WE TELL YOU TO
- You have **180 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!**

Q1: Lambda Calculus: Fibonacci [10 pts]

Your task is to implement the function `FIB` in lambda calculus, which computes the `n`-th Fibonacci number. Recall that the *Fibonacci numbers* is a sequence of natural numbers `0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...` such that each number (except the 0th and 1st) is the sum of the two preceding ones.

Your implementation should satisfy the following test cases:

```
eval fib0 :    eval fib1 :    eval fib2 :    eval fib5 :
  FIB ZERO       FIB ONE        FIB TWO        FIB FIVE
  =~> ZERO       =~> ONE        =~> ONE        =~> FIVE
```

You can use any function defined in the "Lambda Calculus Cheat Sheet" at the end of this exam.

Hint: There are two solution: one uses `FIX`, and the other one does not. We encourage you to implement the latter, using an auxiliary function `STEP` that takes as input a pair of numbers and returns a pair of numbers.

```
let STEP = ____________________________________________________


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


let FIB = _____________________________________________________
```

Q2: Datatypes and Recursion [30 pts]

Consider a simplified version of Nano that only has numbers, addition, variables, and let-bindings. Expressions in this language can be represented using the following Haskell datatype `Expr`:

```
type Id = String

data Expr
  = Num Int           -- numeral
  | Var Id            -- variable
  | Add Expr Expr     -- addition
  | Let Id Expr Expr -- let-binding
```

You can use the following library functions in your solutions below:

- equality: `(==) :: (Eq a) => a -> a -> Bool`
- append two lists: `(++) :: [a] -> [a] -> [a]`
- check list membership: `elem :: (Eq a) => a -> [a] -> Bool`

2.1 Tail-Recursive Delete [10 pts]

Implement a function `delete` that deletes an identifier from a list of identifiers. Your function **must be tail-recursive** and use the provided auxiliary function `loop`.

Your implementation must satisfy the following test cases

```
delete "x" ["foo", "x", "y"]
  ==> ["foo", "y"]
delete "x" ["foo", "y"]
  ==> ["foo", "y"]
```

```
delete :: Id -> [Id] -> [Id]

delete x xs = loop ___________________________________________

  where

    loop :: [Id] -> [Id] -> [Id]

    ____________________________________________________________

    ____________________________________________________________

    ____________________________________________________________

    ____________________________________________________________

    ____________________________________________________________

    ____________________________________________________________
```

2.2 Free Variables [10 pts]

Using `delete` from 2.1, implement a function `freeVars` that computes the set of *free variables* in a Nano expression. Reminder: a variable is free in `e` if at least one of its occurrences is not bound by any enclosing `let`-binding.

```
freeVars :: Expr -> [Id]
```

2.3 Optimize [10 pts]

Using `freeVars` from 2.2, implement a function `optimize` that removes *unused* `let`-bindings from a Nano expression. Your implementation must satisfy the following test cases:

```
-- let y = 2 in 1  ==>  1
optimize (Let "y" (Num 2) (Num 1))
  ==> Num 1
-- let x = 1 in (let y = 2 in x)  ==>  let x = 1 in x
optimize (Let "x" (Num 1) (Let "y" (Num 2) (Var "x")))
  ==> Let "x" (Num 1) (Var "x")
-- let x = 1 in (let y = x in 3)  ==>  3
optimize (Let "x" (Num 1) (Let "y" (Var "x") (Num 3)))
  ==> Num 3
-- let x = 1 in (let y = x in y)  ==> itself
optimize (Let "x" (Num 1) (Let "y" (Var "x") (Var "y")))
  ==> (Let "x" (Num 1) (Let "y" (Var "x") (Var "y")))

optimize :: Expr -> Expr
```

Q3: Higher-Order Functions [30 pts]

Consider the following implementation of *Bucket sort*. The main function `sort` sorts a list of integers `xs` by partitioning its elements into separate lists (*buckets*), one per each integer between the minimum and the maximum element of `xs`, and then concatenating the buckets together:

```
-- | Bucket sort
-- | sort [4, 1, 1, 2]  ==>  [1, 1, 2, 4]
sort :: [Int] -> [Int]
sort [] = []
sort xs = concat (bucket xs [minimum xs .. maximum xs])

-- | Minimum element of a non-empty list
-- | minimum [4, 1, 1, 2]  ==> 1
minimum :: [Int] -> Int
minimum [x]      = x
minimum (x:y:ys) = minimum (min x y : ys)

-- | Maximum element of a non-empty list
-- | maximum [4, 1, 1, 2]  ==> 4
maximum :: [Int] -> Int -- similar to minimum

-- | bucket xs bs distributes elements from `xs` into `bs`:
-- | bucket [4, 1, 1, 2] [1, 2, 3, 4]  ==>  [[1, 1], [2], [], [4]]
bucket :: [Int] -> [Int] -> [[Int]]
bucket _  []     = []
bucket xs (b:bs) = pick b xs : bucket xs bs
  where
    pick b []     = []
    pick b (y:ys) = if y == b then y:(pick b ys) else pick b ys

-- | Concatenate a list of lists
-- | concat [[1, 1], [2], [], [4]]  ==>  [1, 1, 2, 4]
concat :: [[Int]] -> [Int]
concat []       = []
concat (xs:xss) = xs ++ concat xss
```

Your task is to rewrite each recursive helper function of Bucket sort into a function that *always returns the same result* but doesn't directly use recursion. Instead, your functions can use the following higher-order functions from the standard library:

```
map    :: (a -> b) -> [a] -> [b]
filter :: (a -> Bool) -> [a] -> [a]
foldr  :: (a -> b -> b) -> b -> [a] -> b
foldl  :: (b -> a -> b) -> b -> [a] -> b
```

Apart from these four functions, your implementation can **only** use the library functions and syntactic sugar that the original implementation uses, in particular:

- minimum between two integers: `min :: Int -> Int -> Int`
- append two lists: `(++) :: [a] -> [a] -> [a]`
- range syntax: `[n .. m]`

3.1 List minimum [10 pts]

```haskell
-- | Minimum element of a non-empty list
-- | minimum [4, 1, 1, 2]  ==> 1
minimum :: [Int] -> Int
```

3.2 Bucket [10 pts]

```haskell
-- | bucket xs bs distributes elements from `xs` into `bs`:
-- | bucket [4, 1, 1, 2] [1, 2, 3, 4]  ==>  [[1, 1], [2], [], [4]]
bucket :: [Int] -> [Int] -> [[Int]]
```

3.3 Concatenation [10 pts]

```haskell
-- | Concatenate a list of lists
-- | concat [[1, 1], [2], [], [4]]  ==>  [1, 1, 2, 4]
concat :: [[Int]] -> [Int]
```

Q4: Semantics and Type Systems [20 pts]

In this part, you will answer questions about the operational semantics and the type system of Nano, both given in Appendix II at the end of the exam. In each question, mark **all** the answers that apply; it is possible that *none*, *some*, or *all* of the answers are correct.

4.1 Evaluation 1 [5 points]

Which of these evaluation relations are valid according to the operational semantics of Nano?

(A) `[] ; 1 + x ==> 1` [ ]

(B) `[] ; (\x -> 1) ==> 1` [ ]

(C) `[] ; (\x -> 1) (2 + 3) ==> 1` [ ]

(D) `[] ; (\x -> x x) (\x -> x x) ==> <[], x, x x>` [ ]

(E) `[f := <[x:=5],y,x + y>] ; f 1 ==> 6` [ ]

4.2 Evaluation 2 [5 points]

Which of the following rules are used in the derivation of the reduction

```
[] ; (\x y -> x + y) 5  ==>  <[x:=5], y, x+y>
```

(A) `E-Num` [ ]

(B) `E-Var` [ ]

(C) `E-Add` [ ]

(D) `E-Lam` [ ]

(E) `E-App` [ ]

4.3 Typing 1 [5 points]

Which of the following typing judgments are valid according to the type system of Nano?

(A) `[x:Int,y:Int] |- x :: Int` [ ]

(B) `[x:Int] |- x + y :: Int` [ ]

(C) `[] |- \x y -> x :: Int -> Int -> Int` [ ]

(D) `[] |- \x y -> x :: Int -> (Int->Int) -> Int` [ ]

(E) `[] |- \x y -> x :: Int -> Int -> Int -> Int` [ ]

4.4 Typing 2 [5 points]

Which of the following rules are used in the derivation of the typing judgment

`[] |- (\x y -> x + y) 5 :: Int -> Int`

(A) `T-Num` [ ]

(B) `T-Var` [ ]

(C) `T-Add` [ ]

(D) `T-Lam` [ ]

(E) `T-App` [ ]

Appendix I: Lambda Calculus Cheat Sheet

```
-- 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))
let FOUR  = \f x -> f (f (f (f x)))
let FIVE  = \f x -> 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))
let SUB   = \n m -> m DEC n
let EQL   = \n m -> AND (ISZ (SUB n m)) (ISZ (SUB m n))
```

Appendix II: Syntax and Semantics of Nano

Expressions:

```
e ::= n          -- numeral
    | x          -- variable
    | e1 + e2    -- addition
    | \x -> e    -- abstraction
    | e1 e2      -- application
```

Values:

```
v ::= n           -- numeral
    | <E, x, e>   -- closure
```

Environments:

```
E ::= []          -- empty
    | x := v, E   -- value binding and rest
```

Evaluation Rules

```
[E-Num]-------------
       E ; n  ==>  n


[E-Var]--------------------
       (x:=v,E) ; x  ==>  v


       E ; e1 ==> n1   E ; e2 ==> n2   n == n1 + n2
[E-Add]-----------------------------------------
                  E ; (e1 + e2) ==> n


[E-Lam]---------------------------
       E ; (\x -> e) ==> <E, x, e>

       E ; e1 ==> <E', x, e>   E ; e2 ==> v2   (x:=v2,E') ; e ==> v
[E-App]-------------------------------------------------------------
                            E ; (e1 e2) ==> v
```

Types:

```
T ::= Int        -- integers
    | T1 -> T2   -- function types
```

Contexts:

```
G ::= []       -- empty
    | x:T, G   -- type binding and rest
```

Typing Rules

```
[T-Num]-------------
       G |- n :: Int

[T-Var]----------------
       x:T, G |- x :: T

       G |- e1 :: Int   G |- e2 :: Int
[T-Add]-------------------------------
            G |- e1 + e2 :: Int


         x:T1, G |- e :: T2
[T-Lam]------------------------
       G |- \x -> e :: T1 -> T2

       G |- e1 :: T1 -> T2   G |- e2 :: T1
[T-App]-----------------------------------
               G |- e1 e2 :: T2
```