CSE114A, Spring 2023: Final Exam

Instructor: Lindsey Kuper

June 13, 2023

Student name: ______________________________

CruzID (the part before the "@" in your UCSC email address): ______________

This exam has 18 questions and 140 total points.

Instructions

- Please write directly on the exam.
- For short answer questions, please write your answer in the provided boxes. You can use space outside of the boxes as scratch space, but we won't see or grade it.
- For multiple choice questions, please circle the correct choice.
- **You have 180 minutes to complete this exam.** You may leave when you are finished.
- This exam is **closed book**. You may use one double-sided page of notes, but no other materials.
- Avoid seeing anyone else's work or allowing yours to be seen.
- Please, no talking. No notes, books, laptops, phones, or other electronic devices. Do not communicate with anyone but an exam proctor.
- To ensure fairness (and the appearance thereof), **proctors will not answer questions about the content of the exam**. If you are unsure of how to interpret a question, state your interpretation clearly and concisely. *Reasonable interpretations* will be taken into account by graders.
- **We will give partial credit for partially correct answers when it makes sense to do so.** A partially correct answer is better than leaving an answer blank.

Good luck!

This page is for your use as scratch space. Anything you write here will be ungraded.

Part 1: Lambda Calculus

1. (4 points) A lambda calculus expression is in *normal form* if it cannot be further reduced. Evaluate the following lambda calculus expression to normal form using a series of β-reduction steps (and only β-reduction steps – you shouldn't need anything else). Start each line with `=b>`, as if you were using Elsa, and do one β-reduction step per line.

   Note: There may be multiple correct ways to reduce the expression. A correct solution is any solution that Elsa will accept as correct.

```
(\b f g -> b f g) (\x y -> x) (\z -> z) (\f x -> (\q -> q) x)
```

Solution: There are several correct options here. For example, you could work on the outer redex first:

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

Or you could reduce inside the body of `(\f x -> (\q -> q) x)` first:

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

A combination of the above approaches could also work.

2. (3 points) Which of the following lambda calculus expressions is in normal form?

   (a) `(\x -> x x) (\x -> x x)`

   (b) `\step -> (\x -> step (x x)) (\x -> step (x x))`

   (c) `\s z -> s z`

   (d) (a), (b), and (c)

   (e) (b) and (c)

   (f) None of the above

```
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 TRUE  = \x y -> x
let FALSE = \x y -> y
let ITE   = \b x y -> b x y

let PAIR  = \x y b -> ITE b x y
let FST   = \p     -> p TRUE
let SND   = \p     -> p FALSE

let SUC   = \n f x -> f (n f x)
let ADD   = \n m -> n SUC m

-- The definitions of DECR, SUB, and ISZ are elided
-- but you can still use them:
let DECR  = \n ->   -- (decrement n by one)
let SUB   = \n m -> -- (subtract m from n)
let ISZ   = \n ->   -- (return TRUE if n == 0 and FALSE otherwise)

-- The Y combinator
let Y = \step -> (\x -> step (x x)) (\x -> step (x x))
```

3. For this question, the definitions above may be helpful.

   a. (3 points) What does the lambda calculus expression

      `ADD (SND ((\x y b -> ITE b x y) TRUE THREE)) (\f x -> f x)`

      evaluate to?

      (a) Syntax error
      (b) `ONE`
      (c) `TWO`
      (d) `THREE`
      (e) `FOUR`

   b. (3 points) What does the lambda calculus expression

      `ADD (FST ((\x y b -> ITE b x y) FALSE THREE)) (\f x -> f x)`

      evaluate to?

      (a) Syntax error
      (b) `ONE`
      (c) `TWO`
      (d) `THREE`
      (e) `FOUR`

4. In the *triangular* number sequence, the zeroth entry is 0, and the nth entry (indexed from 0) is the sum of n and the $(n-1)$th entry. Therefore the sequence is $0, 1, 3, 6, 10, 15, \ldots$.

   (In other words, the zeroth entry is 0, the first entry is equal to $1 + 0$, the second entry is equal to $2 + 1 + 0$, the third entry is equal to $3 + 2 + 1 + 0$, and so on.)

   Fill in the blanks in the program below to define a recursive lambda calculus function `TRI`, where `TRI n` returns the nth number (indexed from 0) in the above sequence, represented as a Church numeral. For example:

```
TRI ZERO  =~> ZERO
TRI ONE   =~> ONE
TRI TWO   =~> THREE
TRI THREE =~> \f x -> f (f (f (f (f (f x))))) -- 6
TRI FOUR  =~> \f x -> f (f (f (f (f (f (f (f (f (f x))))))))) -- 10
```

   You may assume that `TRI` is only called with non-negative integers represented as Church numerals. You may use any of the functions defined on page 2. Any other helper functions you must define yourself. You must use recursion for full credit.

```
let TRI1 = \rec -> \n -> ITE _____(part 5(a))_______
                             _____(part 5(b))_______
                             _____(part 5(c))_______

let TRI = ______(part 5(d))_______
```

   a. (4 points) 5(a):

      Solution: This is where we check a condition to know whether we are in the base case or not. The simplest correct answer here is `(ISZ n)`.

   b. (4 points) 5(b):

      Solution: This is the base case. The simplest correct answer here is `ZERO`.

   c. (4 points) 5(c):

      Solution: This is the recursive case. The simplest correct answer here is `(ADD n (rec (DECR n)))`.

   d. (4 points) 5(d):

      Solution:

      `Y TRI1`

Part 2: Haskell

5. For each part of this question, write the *type* of the specified Haskell expression. Your answer should be the same as what GHCi's `:t` would say, modulo names of type variables. (For example, if GHCi would say an expression has type `p1 -> p2`, then answers like `a -> b` or `b -> a` would be correct, but `a -> a` would be incorrect since `p1` and `p2` are different type variables.)

   a. (4 points) `\x y -> [True, False, x]`

   Solution: `Bool -> a -> [Bool]`

   b. (4 points) `map (\x -> if x then "xatu" else "mew") [True, False]`

   Solution: `[String]`

   c. (4 points) `map (\x -> if x then "vaporeon" else "espeon")`

   Solution: `[Bool] -> [String]`

   d. (4 points)

   ```
   \x -> case x of
           Just val -> (val, x)
           Nothing  -> ("cramorant", x)
   ```

   Solution: `Maybe String -> (String, Maybe String)`

   e. (4 points)

   ```
   \x -> case x of
           Just val -> val
           Nothing  -> ["dodrio", "delphox", "dragonite"]
   ```

   Solution: `Maybe [String] -> [String]`

6. (8 points) For this question, you will implement a Haskell function `foo` that takes three arguments: a default value of type `b`, a function of type `a -> b`, and a value of type `Maybe a`. If the `Maybe a` value is `Nothing`, then `foo` returns the default value. Otherwise, it applies the provided function to the value inside the `Just` and returns the result.

   Hint: You can implement `foo` in two lines of code, one for each of the two cases to handle.

```
foo :: b -> (a -> b) -> Maybe a -> b
```

   Solution: A concise solution is:

```
foo n _ Nothing  = n
foo _ f (Just x) = f x
```

   Incidentally, this function is part of the Haskell standard library, where it's called `maybe` (not be confused with upper-case `Maybe`)!

7. (3 points) Which of the following expressions does *not* have type `String`? Hint: If you need them, the type signatures of `map`, `foldl`, and `(++)` are in the Haskell Reference on the last page of the exam.

   (a) `map (\x -> []) "lucario"`

   (b) `foldl (++) "pikachu" []`

   (c) `foldl (++) "" []`

   (d) `foldl (\x y -> x) [] ["ninetales"]`

   (e) `foldl (\x y -> "aipom") "" [1, 2, 3]`

   (f) (a) and (e)

8. (3 points) Which of the following list comprehensions does *not* have type `[Bool]`?

   (a) `[ if x == 3 then False else True | x <- [1, 2, 3] ]`

   (b) `[ (\y -> if y then False else True) x | x <- [True, False] ]`

   (c) `[ if x then False else True | x <- [1, 2, 3] ]`

   (d) `[ (\y -> if y == 3 then False else True) 3 | x <- [True] ]`

   (e) (c) and (d)

Part 3: Abstract Syntax Trees, Interpreters, Environments, and Scope

For the questions in this section, we will use the following `Expr` data type. It defines the grammar of abstract syntax trees for a little language with numbers, variables, addition expressions, lambda (function definition) expressions, application (function call) expressions, and `let`-expressions.

```
data Expr = ENum Int | EVar Id | EPlus Expr Expr
          | ELam Id Expr | EApp Expr Expr | ELet Id Expr Expr

type Id = String
```

For example, we would represent the expression

```
let f = \x -> x in
  f (3 + 4)
```

with the `Expr`

```
ELet "f" (ELam "x" (EVar "x"))
         (EApp (EVar "f") (EPlus (ENum 3) (ENum 4)))
```

9. Be the parser! Translate the following expressions into their corresponding `Expr`s.

   a. (4 points)

   ```
   let n = 2 in
     let m = 3 + n in
       \x -> m + x
   ```

   Solution:

   ```
   ELet "n" (ENum 2)
            (ELet "m" (EPlus (ENum 3) (EVar "n"))
                      (ELam "x" (EPlus (EVar "m")
                                       (EVar "x"))))
   ```

   b. (4 points)

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

   Solution:

   ```
   EApp (ELam "x" (EVar "x"))
        (ELam "z" (ELam "y" (EApp (EVar "y") (EVar "z"))))
   ```

10. If we typed in an `Expr` at the GHCi prompt, we'd get an error saying that there is no instance of the `Show` typeclass for the `Expr` type. Let's fix that by implementing a custom instance of `Show` for `Expr`s. To do so, we need to implement a function `show` with type signature `Expr -> String`.

    Here are some examples of the behavior we should see in GHCi after implementing `show`:

```
ghci> EPlus (ENum 3) (EPlus (ENum 4) (EVar "z"))
(3 + (4 + z))
ghci> EApp (EVar "x") (EVar "y")
(x y)
ghci> EApp (EApp (EVar "x") (EVar "y")) (EPlus (ENum 3) (EVar "z"))
((x y) (3 + z))
ghci> ELet "x" (ENum 3) (EPlus (EVar "x") (ENum 2))
let x = 3 in (x + 2)
ghci> ELet "f" (ELam "x" (EVar "x")) (EApp (EVar "f") (ENum 3))
let f = (\x -> x) in (f 3)
```

    The implementation of `show` is below, with some blanks for you to fill in. Hint: The `EPlus`, `EApp`, and `ELet` cases are not that different from the provided `ELam` case.

```
instance Show Expr where
  show :: Expr -> String
  show (ENum n)         = show n
  show (EVar s)         = s
  show (EPlus e1 e2)    = ________(11(a))________
  show (ELam id body)   = "(\\" ++ id ++ " -> " ++ show body ++ ")"
  show (EApp e1 e2)     = ________(11(b))________
  show (ELet id e1 e2)  = ________(11(c))________
```

    a. (4 points) 11(a):

    Solution:
    `"(" ++ show e1 ++ " + " ++ show e2 ++ ")"`

    b. (4 points) 11(b):

    Solution:
    `"(" ++ show e1 ++ " " ++ show e2 ++ ")"`

    c. (4 points) 11(c):

    Solution:
    `"let " ++ id ++ " = " ++ show e1 ++ " in " ++ show e2`

11. We will be writing an interpreter for `Exprs`, but first, we need to set up some machinery. First, we'll define a type of `Values` that expressions can evaluate to:

```
data Value = VNum Int | VClos Env Id Expr
```

We can now define a type of *environments* that associate variable identifiers with values. We will represent an environment as a list of pairs of `Id` and `Value`:

```
type Env = [(Id, Value)]
```

a. (4 points) The Haskell function `lookupInEnv` takes as arguments a variable identifier and an environment, and returns the value that the specified identifer is associated with in the environment. Fill in the blank below to complete the definition of `lookupInEnv`.

```
lookupInEnv :: Id -> Env -> Value
lookupInEnv id []           = error "unbound variable"
lookupInEnv id ((x,val):xs) = ________(11(a))________
```

Solution: There are a couple of correct ways to write this, but a concise answer is:

```
if id == x
  then val
  else lookupInEnv id xs
```

b. (4 points) Write a Haskell function `extendEnv` that takes as arguments a variable identifier, a value, and an environment, and returns a new environment that extends the old one with a binding for the specified identifier and value. The type signature is provided for you. Hint: With the way we are representing environments, this is a one-liner.

```
extendEnv :: Id -> Value -> Env -> Env
```

Solution:

```
extendEnv id val env = (id, val) : env
```

12. We can now define an interpreter `eval` that takes an environment of type `Env` and an expression of type `Expr` and returns a value of type `Value`. Fill in the blanks in the following definition of `eval`.

```
eval :: Env -> Expr -> Value
eval env (ENum n)       = VNum n
eval env (EVar s)       = lookupInEnv s env
eval env (EPlus e1 e2)  = case (eval env e1, eval env e2) of
  (VNum n1, VNum n2) -> VNum (n1 + n2)
  _                  -> error "type error: not a number"
eval env (ELam id body) = VClos env id body
eval env (EApp e1 e2)   = case ________(12(a))________ of
  VClos ce id e -> let argVal      = ________(12(b))________
                       extendedEnv = ________(12(c))________
                   in eval extendedEnv e
  _             -> error "type error: not a function"
eval env (ELet id e1 e2) =
  let v1          = eval env e1
      extendedEnv = ________(12(d))________
    in ________(12(e))________
```

a. (4 points) 12(a):

Solution:
`eval env e1`

b. (4 points) 12(b):

Solution:
`eval env e2`

c. (4 points) 12(c):

Solution:
`extendEnv id argVal ce`

d. (4 points) 12(d):

Solution:
`extendEnv id v1 env`

e. (4 points) 12(e):

Solution:
`eval extendedEnv e2`

13. Consider the following Nano program:

```
let a = 3 in
  let f = \x y -> x + y + a in
    let x = 4 in
      let a = 5 in
        f x a
```

a. (3 points) Under **static scope**, what would the above program evaluate to?

   (a) error: multiple declarations of a variable
   (b) error: unbound variable
   (c) 14
   (d) 12
   (e) 10

b. (3 points) Under **dynamic scope**, what would the above program evaluate to?

   (a) error: multiple declarations of a variable
   (b) error: unbound variable
   (c) 14
   (d) 12
   (e) 10

Part 4: Types, Unification, and Type Inference

14. (3 points) Which of the following is *a unifier* for the types `String` and `a -> b`?
    (a) `[(a, String), (b, String)]`
    (b) `[(a -> b, String)]`
    (c) `[(String, a -> b)]`
    (d) (b) and (c)
    (e) Cannot unify

15. (3 points) Which of the following is *a unifier* for the types `Int -> Int` and `a -> b`?
    (a) `[(Int, a), (Int, b)]`
    (b) `[(a, Int), (b, Int)]`
    (c) `[(a, Int), (b, Int), (c, String)]`
    (d) (a), (b), and (c)
    (e) (b) and (c)
    (f) Cannot unify

16. (3 points) Which of the following is *a unifier* for the types `a -> b` and `(b -> Bool) -> c`?
    (a) `[(b, c), (a, c -> Bool)]`
    (b) `[(a, Bool -> Bool), (b, Bool), (c, Bool -> Bool)]`
    (c) `[(a, Bool -> Bool), (b, Bool), (c, Bool)]`
    (d) (a) and (c)
    (e) Cannot unify

17. (3 points) Which of the following is *the most general unifier* for the types `a -> b` and `Int -> c -> Int`?
    (a) `[(a, Int), (b, c -> Int)]`
    (b) `[(a, Int -> c), (b, Int)]`
    (c) `[(a, Int), (b, Int -> Int), (c, Int)]`
    (d) `[(a, Int), (b, c -> Int), (c, Int)]`
    (e) Cannot unify

18. Here are some of the typing rules for the Nano language:

```
              G,(x,T1) |- e :: T2                  (x,T) in G
[T-Lam] --------------------------     [T-Var] -----------
        G |- (\x -> e) :: T1 -> T2             G |- x :: T

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

        G |- e1 :: T1    G,(x,T1) |- e2 :: T2
[T-Let] -------------------------------------    [T-Int] -------------
               G |- let x = e1 in e2 :: T2                 G |- n :: Int
```

Below is a partial typing derivation for the Nano expression `let y = 3 in (\x -> x) y`. We are using the following abbreviations for type environments:

```
G1 = [(y,Int)]
G2 = [(y,Int),(x,Int)]
```

For each blank below, fill in a type or the name of a typing rule to complete the typing derivation.

```
                                            (x,Int) in G2
                                    [_18a_]----------------
                                          G2 |- x :: _18b_            (y,Int) in G1
                                [_18c_]------------------------ [_18d_]----------------
                                      G1 |- (\x -> x) :: _18e_        G1 |- y :: _18f_
[_18g_]---------------- [_18h_]------------------------------------------------------
       [] |- 3 :: _18i_                  G1 |- (\x -> x) y :: _18j_
[_18k_]-------------------------------------------------------------------------
                        [] |- let y = 3 in (\x -> x) y :: Int
```

a. (1 point) 18(a):

Solution:
`T-Var`

b. (1 point) 18(b):

Solution:
`Int`

c. (1 point) 18(c):

Solution:
`T-Lam`

d. (1 point) 18(d):

Solution:
`T-Var`

e. (1 point) 18(e):

Solution:
`Int -> Int`

f. (1 point) 18(f):

Solution:
`Int`

g. (1 point) 18(g):

Solution:
`T-Int`

h. (1 point) 18(h):

Solution:
`T-App`

i. (1 point) 18(i):

Solution:
`Int`

j. (1 point) 18(j):

Solution:
`Int`

k. (1 point) 18(k):

Solution:
`T-Let`

Haskell Reference

- `map :: (a -> b) -> [a] -> [b]`
- `foldl :: (b -> a -> b) -> b -> [a] -> b`
- `(++) :: [a] -> [a] -> [a]`

  Append two lists, e.g.,

  ```
  > [1,2,3] ++ [4,5]
  [1,2,3,4,5]
  > "apple" ++ "orange"
  "appleorange"
  ```