CSE114A, Spring 2023: Midterm Exam

Instructor: Lindsey Kuper

May 11, 2023

Student name: ______________________________

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

This exam has 10 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 completely fill in the circle for the correct choice.
- **You have 95 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 problem description, 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. (5 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.

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

```
-- Church numerals
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))))

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

-- Arithmetic
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)

-- Note: Since ZERO is the smallest Church numeral,
-- calls to DECR and SUB bottom out at ZERO.
-- For example, DECR ZERO evaluates to ZERO,
-- and SUB TWO THREE evaluates to ZERO.

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

2. (5 points) Fill in the blank below to define a lambda calculus function `LEQ` that takes two Church numerals `n` and `m` as arguments, and returns `TRUE` if `n` is less than or equal to `m` and `FALSE` otherwise. You may use any of the functions defined above.

   `let LEQ = \n m -> ____(write your answer in the box below)____`

3. In the famous Fibonacci number sequence, each number is the sum of the two preceding ones. If we begin with 0 and 1, the sequence is $0, 1, 1, 2, 3, 5, 8, 13, \ldots$. Fill in the blanks in the program below to define a recursive lambda calculus function `FIB`, where `FIB n` returns the nth number (indexed from 0) in the above sequence. For example:

```
FIB ZERO  =~> ZERO
FIB ONE   =~> ONE
FIB TWO   =~> ONE
FIB THREE =~> TWO
FIB FOUR  =~> THREE
FIB FIVE  =~> FIVE
```

You may assume that `FIB` is only called with non-negative integers represented as Church numerals. You may use any of the functions defined on the previous page, and you may also use `LEQ` from the previous question. Any other helper functions you must define yourself. You must use recursion for full credit.

```
let FIB1 = \rec -> \n -> ITE _____(part 3(a))_______
                             _____(part 3(b))_______
                             _____(part 3(c))_______

let FIB = ______(part 3(d))_______
```

a. (5 points) 3(a):

b. (5 points) 3(b):

c. (5 points) 3(c):

d. (5 points) 3(d):

Part 2: Haskell

4. 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. (5 points) `True : [False, True, False]`

   b. (5 points) `\x y -> "charizard"`

   c. (5 points) `foldr (++) "" ["mew", "lucario", "squirtle"]`

   d. (5 points) `\x -> if x then [x] else [False]`

   e. (5 points)
   ```
   \x -> case x of
     Just s  -> s
     Nothing -> "Sorry, there's nothing here!"
   ```

   f. (5 points) `map (\x y -> x) (foldr (++) [] [[True, False], [True]])`

5. For each part of this question, write a Haskell *expression* that has the specified type. There may be many correct answers, but each of these questions can be answered with a short one-liner, so for full credit, aim for simplicity. There is no need to use Haskell library functions here.

   a. (5 points) `[(Bool, String)]`

   b. (5 points) `Bool -> String`

   c. (5 points) `(Bool -> String) -> [String]`

   d. (5 points) `a -> a`

6. The Haskell library function `toUpper :: Char -> Char` converts characters to upper case.

   a. (5 points) What does

      ```
      map (\x -> map toUpper x) ["foo", "bar", "baz"]
      ```

      evaluate to?

      ◯ Syntax error
      ◯ Type error
      ◯ `["FOO","BAR","BAZ"]`
      ◯ `["Foo","Bar","Baz"]`

   b. (5 points) Suppose we want to translate the expression from part (a) to use *list comprehensions* instead of `map`. Which of the following is an accurate translation?

      ◯ `[ [ toUpper c | c <- x ] | x <- ["foo", "bar", "baz"]]`
      ◯ `[ toUpper x | x <- ["foo", "bar", "baz"]]`
      ◯ `[ [ c | c <- toUpper x ] | x <- ["foo", "bar", "baz"]]`
      ◯ `[ toUpper c | c <- [ x | x <- ["foo", "bar", "baz"]]]`

Part 3: Working with Abstract Syntax Trees

Consider the following data type for abstract syntax trees of lambda calculus expressions:

```
data LCExpr = Var String | Lam String LCExpr | App LCExpr LCExpr
```

For example, we would represent the lambda calculus expression `\f -> \y -> g y` with the `LCExpr`

```
Lam "f" (Lam "y" (App (Var "g") (Var "y")))
```

7. (5 points) Translate the following lambda calculus expression into the corresponding `LCExpr`:

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

8. (15 points) **For this problem, you may use the Haskell library functions described in the Haskell Reference on the last page of the exam.**

Let us define the *depth* of a lambda calculus expression as follows:

- The depth of a variable is 1.
- The depth of a lambda abstraction `\x -> e` is 1 + the depth of `e`.
- The depth of an application `e1 e2` is 1 + the maximum of the depth of `e1` and the depth of `e2`.

Write a Haskell function `depth` that takes an `LCExpr` and returns its depth. Here are some sample calls to `depth`:

```
> depth (Var "x")
1
> depth (Lam "x" (Var "x"))
2
> depth (Lam "f" (Lam "y" (App (Var "g") (Var "y"))))
4
> depth (App (Lam "x" (Lam "y" (Var "y"))) (App (Var "q") (Var "z")))
4
```

The `depth` type signature is provided below for you. Hint: You can solve this problem in 3 lines of code, and the arithmetic operations on the last page of the exam will be a big help.

```
depth :: LCExpr -> Int
```

9. In lecture, we saw how we can implement custom instances of the `Eq` and `Ord` typeclasses for any type we want. Let's implement instances of `Eq` and `Ord` for the `LCExpr` type.

   Now that we can compute the depth of `LCExprs`, let's say that `LCExprs` are equal if they have the same depth, and not equal otherwise. Likewise, for `LCExprs` `e1` and `e2`, let's say that `e1 <= e2` if `e1`'s depth is less than or equal to `e2`'s depth.[1]

   a. (5 points) Implement an `Eq` instance for `LCExpr` that will give us the behavior described above. The `Eq` instance declaration and `(==)` type signature are provided below for you. You may use the `depth` function you wrote for the previous question. Hint: You can solve this problem in 1 line of code.

```
instance Eq LCExpr where
  (==) :: LCExpr -> LCExpr -> Bool
```

   b. (5 points) Now implement an `Ord` instance for `LCExpr` that will give us the behavior described above. The `Ord` instance declaration and `(<=)` type signature are provided below for you. You may use the `depth` function you wrote for the previous question. Hint: You can solve this problem in 1 line of code.

```
instance Ord LCExpr where
  (<=) :: LCExpr -> LCExpr -> Bool
```

   c. (5 points) Haskell's list library has a `sort` function of type `Ord a => [a] -> [a]`. The `sort` function arranges the elements of the input list from smallest to largest. Any elements that are considered equal remain in the order they appeared in the input.

   Now that we have our `Eq` and `Ord` instances in place for `LCExpr`, what would the following expression evaluate to?

```
sort [Var "z", App (Var "x") (Var "z"), Lam "y" (Var "y"), Var "x"]
```

[1] These are admittedly strange definitions of program equality and ordering, but let's just roll with it.

10. (15 points) **For this problem, you may use the Haskell library functions described in the Haskell Reference on the last page of the exam.**

An occurrence of a variable in a lambda calculus expression is *free* if it's not in the scope of an enclosing lambda abstraction. For example, in the expressions `\x -> x` and `\x -> (\y -> x)`, the variable `x` is bound by a `\x` binder. In the following expressions, `x` is free:

```
x y              -- no binders at all (both x and y occur free)
\y -> x          -- no binder for x
\z -> (\y -> x)  -- no binder for x
(\x -> y) x      -- x occurs outside of the \x binder
```

For this question, you will define a Haskell function `freeVars` that takes an `LCExpr` and returns a list of variables that occur free in it (in any order). Here are some sample calls to `freeVars`:

```
> freeVars (Var "x")
["x"]
> freeVars (Lam "y" (Var "y"))
[]
> freeVars (App (Var "f") (Var "x"))
["f","x"]
> freeVars (Lam "f" (Lam "y" (App (Var "g") (Var "y"))))
["g"]
> freeVars (App (Var "x") (Lam "x" (Var "x")))
["x"]
> freeVars (App (Lam "x" (Var "y")) (Var "x"))
["y","x"]
```

For full credit, a variable that occurs free more than once in an expression should only appear once in the list returned by `freeVars`. Thus `freeVars (App (Var "y") (Var "y"))` should evaluate to `["y"]`. The `freeVars` type signature is provided below for you. Hint: You can solve this problem in 3 lines of code, and the list operations on the last page of the exam will be a big help.

```
freeVars :: LCExpr -> [String]
```

Haskell Reference

- `(+) :: Num a => a -> a -> a`

  Returns the sum of its two arguments, e.g.,

  ```
  > 3 + 4
  7
  ```

- `max :: Ord a => a -> a -> a`

  Returns the maximum of its two arguments, e.g.,

  ```
  > max 3 4
  4
  ```

- `(++) :: [a] -> [a] -> [a]`

  Append two lists, e.g.,

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

- `nub :: [a] -> [a]`

  Remove duplicate elements from a list, e.g.,

  ```
  > nub [1,2,3,4,3,2,1,2,4,3,5]
  [1,2,3,4,5]
  ```

- `(\\) :: [a] -> [a] -> [a]`

  Compute the difference of two lists. In the result of `xs \\ ys`, the first occurrence of each element of `ys` in turn (if any) has been removed from `xs`. Thus `((xs ++ ys) \\ xs) == ys`, e.g.,

  ```
  > ["a","b","c"] \\ ["a"]
  ["b","c"]
  > ["a","b","c","a"] \\ ["a","c"]
  ["b","a"]
  ```