CSE114A, Winter 2025: Midterm Exam

Instructor: Lindsey Kuper

February 18, 2025

Student name: ______________________________

CruzID: ______________________________@ucsc.edu

This exam has 17 questions and 100 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 just 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 would accept as correct.

```
(\p q r -> r (p q)) ((\x y -> x y) (\z -> z)) (\s -> s)
```

Solution: Here is one solution, which deals with the `(\x y -> x y) (\z -> z)` redex first:

```
(\p q r -> r (p q)) ((\x y -> x y) (\z -> z)) (\s -> s)
=b> (\p q r -> r (p q)) (\y -> (\z -> z) y) (\s -> s)
=b> (\q r -> r ((\y -> (\z -> z) y) q)) (\s -> s)
=b> \r -> r ((\y -> (\z -> z) y) (\s -> s))
=b> \r -> r ((\y -> y) (\s -> s))
=b> \r -> r (\s -> s)
```

Alternatively, you could deal with the `(\p q r -> r (p q)) ((\x y -> x y) (\z -> z))` redex first:

```
(\p q r -> r (p q)) ((\x y -> x y) (\z -> z)) (\s -> s)
=b> (\q r -> r (((\x y -> x y) (\z -> z)) q)) (\s -> s)
=b> \r -> r (((\x y -> x y) (\z -> z)) (\s -> s))
=b> \r -> r ((\y -> (\z -> z) y) (\s -> s))
=b> \r -> r ((\z -> z) (\s -> s))
=b> \r -> r (\s -> s)
```

Any correct reduction sequence will end in `\r -> r (\s -> s)`.

For the next three questions, you may use any of the helper functions defined in the provided lambda calculus reference at the end of the exam.

2. (4 points) Define a lambda calculus function `PAIRTRUE` that takes a `PAIR` of two Booleans as its argument and returns `TRUE` if both Booleans are `TRUE`, and `FALSE` otherwise. For example, in Elsa:

```
PAIRTRUE (PAIR TRUE TRUE) =~> TRUE
PAIRTRUE (PAIR TRUE FALSE) =~> FALSE
PAIRTRUE (PAIR FALSE TRUE) =~> FALSE
PAIRTRUE (PAIR FALSE FALSE) =~> FALSE
```

You may assume that `PAIRTRUE` is only called with `PAIR`s of Booleans.

```
let PAIRTRUE =
```

Solution: One simple correct answer is:

```
\p -> AND (FST p) (SND p)
```

3. (4 points) Define a lambda calculus function `PAIRTWO` that takes a `PAIR` of two Church numerals as its argument and returns `TRUE` if they sum to `TWO`, and `FALSE` otherwise. For example, in Elsa:

```
PAIRTWO (PAIR ONE ONE) =~> TRUE
PAIRTWO (PAIR ZERO TWO) =~> TRUE
PAIRTWO (PAIR TWO ZERO) =~> TRUE
PAIRTWO (PAIR ONE ZERO) =~> FALSE
PAIRTWO (PAIR ZERO ONE) =~> FALSE
```

You may assume that `PAIRTWO` is only called with `PAIR`s of Church numerals.

```
let PAIRTWO =
```

Solution: One simple correct answer is:

```
\p -> EQL TWO (ADD (FST p) (SND p))
```

4. We can encode *lists* in lambda calculus using pairs, as follows:

```
let NIL   = \x -> TRUE
let CONS  = PAIR
let HEAD  = FST
let TAIL  = SND
let ISNIL = \lst -> lst (\h t -> FALSE)
```

The list constructors are `NIL` and `CONS`. `NIL` is the empty list, and a non-empty list consists of a pair of a list element and a list. For instance, `(CONS ONE (CONS TWO (CONS THREE NIL)))` is the equivalent of the Haskell list `[1, 2, 3]`. The `ISNIL` function takes a list and returns `TRUE` if it is `NIL` and `FALSE` otherwise.

Define a lambda calculus function `LENGTH` that takes a list encoded in this way and returns its length as a Church numeral. You may assume that `LENGTH` is called with a list constructed with `NIL` or `CONS`. You must use recursion for full credit.

```
let LENGTH1 = \rec -> \lst -> ITE ______(part 4(a))________
                                  ______(part 4(b))________
                                  ______(part 4(c))________

let LENGTH = _______(part 4(d))________
```

a. (5 points) 4(a):

Solution: This is where we check a condition to know whether we are in the base case or not. `(ISNIL lst)` is a correct answer.

b. (5 points) 4(b):

Solution: This is the base case. `ZERO` is a correct answer.

c. (5 points) 4(c):

Solution: This is the recursive case. `(INCR (rec (TAIL lst)))` is a correct answer.

d. (5 points) 4(d):

Solution:

```
Y LENGTH1
```

Part 2: Haskell

The Haskell reference at the end of the exam has information about library functions used in this section.

5. (4 points) What is the **type** of the following Haskell expression?

```
\x y -> ["oliver", "angie", x, y]
```

- ◯ Type error
- ◯ `[String]`
- ◯ `[String] -> [String] -> [String]`
- √ `String -> String -> [String]`
- ◯ `[a] -> [a] -> [String]`
- ◯ `a -> a -> [String]`

6. (4 points) What is the **type** of the following Haskell expression?

```
(\x y -> ["sunny", "coco", x, y]) ""
```

- ◯ Type error
- ◯ `[String]`
- √ `String -> [String]`
- ◯ `[String] -> [String]`
- ◯ `a -> [String]`
- ◯ `[a] -> [String]`

7. (4 points) What is the **type** of the following Haskell expression?

```
\x -> if x == "waldo" then Just "loki" else Nothing
```

- ◯ Type error
- ◯ `String -> String`
- ◯ `Maybe String -> Maybe String`
- ◯ `Maybe String -> String`
- √ `String -> Maybe String`

8. (4 points) What is the **type** of the following Haskell expression?

```
map (\x -> [x, "kira", "juno"])
```

- ◯ Type error
- √ `[String] -> [[String]]`
- ◯ `String -> [[String]]`
- ◯ `[String] -> [String]`
- ◯ `String -> [String]`

9. (4 points) What is the **type** of the following Haskell expression?

```
\x y -> (x + x) == 3
```

- ◯ `Eq a => a -> b -> Bool`
- √ `(Eq a, Num a) => a -> b -> Bool`
- ◯ `Eq a => a -> a -> Bool`
- ◯ `(Eq a, Num a) => a -> a -> Bool`

10. (4 points) What is the **type** of the following Haskell expression?

```
map (\x -> [Nothing, x, Nothing]) [False, True, False]
```

- √ Type error
- ◯ `[Bool]`
- ◯ `[Maybe Bool]`
- ◯ `[[Bool]]`
- ◯ `[[Maybe Bool]]`

11. (5 points) What does the following Haskell expression **evaluate to**?

```
map (\x y -> x == 3) [1, 2, 3]
```

Solution: The idea here is that since `\x y -> x == 3` takes two arguments, mapping it over the list will give us a list of functions of one argument. We would accept either

```
[\y -> False, \y -> False, \y -> True]
```

or

```
[\y -> 1 == 3, \y -> 2 == 3, \y -> 3 == 3]
```

as answers. (The expressions `1 == 3`, `2 == 3`, and `3 == 3` evaluate to `False`, `False`, and `True` respectively, so for the purposes of this question, the two answers are essentially equivalent modulo details about evaluation order.)

12. (5 points) What does the following Haskell expression **evaluate to**?

```
foldr (\x y -> "hello" ++ x ++ y) "" ["cupid", "mischief"]
```

Solution:

```
"hellocupidhellomischief"
```

Part 3: Working with Abstract Syntax Trees

For the next four questions, we'll use the following `LCExpr` data type, which defines a grammar of abstract syntax trees (ASTs) for lambda calculus. `LCExpr`s are constructed using three constructors: `LCVar`, `LCLam`, and `LCApp`.

```
data LCExpr = LCVar String
            | LCLam String LCExpr
            | LCApp LCExpr LCExpr
```

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

```
LCLam "x" (LCLam "y" (LCApp (LCVar "x") (LCVar "y")))
```

13. (4 points) Be the parser! Translate the following lambda calculus expression into its corresponding `LCExpr`.

```
(\z -> z) (\q -> \r -> r (r q))
```

Solution: The key here is to realize that the whole expression is an application of the function `(\z -> z)` to the argument `(\q -> \r -> r (r q))`, so we have an `LCApp` constructor on the outside.

```
LCApp (LCLam "z" (LCVar "z"))
      (LCLam "q" (LCLam "r" (LCApp (LCVar "r")
                                   (LCApp (LCVar "r")
                                          (LCVar "q")))))
```

14. (6 points) The *size* of an `LCExpr` is the number of `LCExpr` constructors that it has. For instance:
`LCVar "x"` is size 1,
`LCLam "x" (LCVar "x")` is size 2, and
`LCApp (LCLam "x" (LCVar "x")) (LCVar "y")` is size 4.

Define a Haskell function `size` that takes an `LCExpr` and returns its size as an `Int`. You can use the library functions in the Haskell reference at the end of the exam, but no other library functions. The type signature of `size` is provided for you below; fill in the rest of the definition.

```
size :: LCExpr -> Int
```

Solution: A simple implementation is:

```
size (LCVar _) = 1
size (LCLam _ e) = 1 + size e
size (LCApp e1 e2) = 1 + size e1 + size e2
```

15. (6 points) Is your implementation of `size` in the previous question tail-recursive? If so, what makes it tail-recursive? If not, what makes it not tail-recursive, and would it be possible to implement tail-recursively? (Answer in 2-3 sentences)

Solution: My implementation is not tail-recursive, because a recursive call to `size` is not the last thing that happens in the body of the function; rather, a call to `(+)` is.

In principle, though, one could write a tail-recursive implementation using continuation-passing style.

(If anyone actually wrote a correct CPS implementation on the exam, then "yes, my implementation is tail-recursive" would be the correct answer here! However, no one attempted this.)

16. (5 points) In lecture, we saw how we can implement custom instances of the `Eq` type class. Let's implement an instance of `Eq` for the `LCExpr` type. We will say that `LCExpr`s are equal if they have the same size, and not equal otherwise.[1]

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 `size` function you wrote for the previous question as well as the library functions from the Haskell reference at the end of the exam but no other library functions. (Hint: Your answer should be one short line of code.)

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

Solution:

```
  (==) e1 e2 = size e1 == size e2
```

[1] This might not be a very good notion of program equivalence, but let's roll with it.

For the next question, we'll use the following `Expr` data type, which defines a grammar of abstract syntax trees for a small arithmetic language.

```
data Expr = EPlus Expr Expr
          | EMinus Expr Expr
          | ENum Int
          | EVar String
```

We want to write an interpreter for `Expr`s. Since `Expr`s may contain variables, our interpreter will need to be an *environment-passing* interpreter. We will represent an environment as a list of pairs of `String`s and `Value`s, using the following type alias:

```
type Env = [(String, Int)]
```

If an expression contains variables that are not bound in the environment, we won't be able to interpret it, so we'll use a `Maybe` type as the return type of our interpreter. If we try to evaluate an expression containing a variable that does not have a binding in the provided environment, our interpreter should return `Nothing`.

Here are some example calls to `eval`:

```
ghci> eval (EPlus (ENum 3) (ENum 5)) []
Just 8
ghci> eval (EPlus (ENum 3) (EVar "x")) [("x", 5)]
Just 8
ghci> eval (EMinus (EVar "y") (EVar "x")) [("x", 5)]
Nothing
ghci> eval (EMinus (EVar "y") (EVar "x")) [("x", 5), ("y", 6)]
Just 1
```

Note: Although the above examples are simple, in general our interpreter should be able to handle arbitrarily deeply nested `Expr`s.

17. (12 points) The type signature of our interpreter is provided below. Fill in the definition of `eval`. You can use library functions from the Haskell reference at the end of the exam, and you can also use the two helper functions at the bottom of the page. (Hint: If you use the helper functions, you can do this in four pretty short lines of code.)

```
eval :: Expr -> Env -> Maybe Int
```

Solution: This question is easy if you understand what the two helpers do. The only potentially tricky parts are (1) remembering to use `Just` in the base case instead of just returning `n` (because `eval` must return a `Maybe Int`), and (2) remembering to call `eval` on the second and third arguments to `evalNumOp` because they need to be `Maybe Ints`.

```
eval (ENum n)        _   = Just n
eval (EVar s)        env = lookupInEnv s env
eval (EPlus e1 e2)   env = evalNumOp (+) (eval e1 env) (eval e2 env)
eval (EMinus e1 e2) env = evalNumOp (-) (eval e1 env) (eval e2 env)
```

```
evalNumOp :: (Int -> Int -> Int) -> Maybe Int -> Maybe Int -> Maybe Int
evalNumOp f (Just n) (Just m) = Just (f n m)
evalNumOp f _        _        = Nothing

lookupInEnv :: String -> Env -> Maybe Int
lookupInEnv _ []          = Nothing
lookupInEnv s ((k,v):kvs) = if s == k then Just v else lookupInEnv s kvs
```

Lambda Calculus Reference

```
-- 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))

-- Booleans
let TRUE  = \x y -> x
let FALSE = \x y -> y
let ITE   = \b x y -> b x y
let AND   = \b1 b2 -> ITE b1 b2 FALSE

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

-- Lists
let NIL   = \x -> TRUE
let CONS  = PAIR
let HEAD  = FST
let TAIL  = SND
let ISNIL = \lst -> lst (\h t -> FALSE)

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

-- The definitions of DECR, SUB, ISZ, and EQL 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)
let EQL   = \n m -> -- (return TRUE if n == m 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))
```

Haskell Reference

- ```
  map :: (a -> b) -> [a] -> [b]
  map f []     = []
  map f (x:xs) = f x : map f xs
  ```

- ```
  foldr :: (a -> b -> b) -> b -> [a] -> b
  foldr f b []     = b
  foldr f b (x:xs) = f x (foldr f b xs)
  ```

- ```
  foldl :: (b -> a -> b) -> b -> [a] -> b
  foldl f acc []     = acc
  foldl f acc (x:xs) = foldl f (f acc x) xs
  ```

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

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

  ```
  > 3 + 4
  7
  ```

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

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

  ```
  > 5 - 4
  1
  ```

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

  Appends two lists, e.g.,

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

- `(==) :: Eq a => a -> a -> Bool`

  Compares two arguments for equality, e.g.,

  ```
  > False == True
  False
  > "apple" == "apple"
  True
  ```
  ```