CSE114A, Spring 2025: Midterm Exam 2

Instructor: Lindsey Kuper

May 16, 2025

Student name: _____

CruzID: _____@ucsc.edu

This exam has 11 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 65 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.

Recent research has shown that viewing cute images can improve people's performance on tasks requiring focused attention, so here's Bhagavan's dog.

Good luck!

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

Part 1: Lambda Calculus (30 points)

1. (6 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.

```
(\x y -> (\f -> f x y)) (\r q -> q) (\s -> s) (\b c -> b)
```

The next question will involve a lambda calculus encoding of **lists**. 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 TWO (CONS THREE (CONS ONE NIL))` is the lambda calculus equivalent of the Haskell list `[2, 3, 1]`, which is just syntactic sugar for `2 : (3 : (1 : []))`.

The `ISNIL` function takes a list and returns `TRUE` if the list is `NIL` and `FALSE` otherwise.

You can (and should!) use the `NIL`, `CONS`, `HEAD`, `TAIL`, and `ISNIL` functions in your answers to the next question.

2. Define a lambda calculus function `INCRLIST` that takes a list of Church numerals as its input, and returns a list in which all of the Church numerals in the input list have been incremented by one. For example, in Elsa:

```
INCRLIST NIL =~> NIL

INCRLIST (CONS ONE NIL) =*> CONS TWO NIL

INCRLIST (CONS ZERO (CONS ZERO NIL)) =*> CONS ONE (CONS ONE NIL)

INCRLIST (CONS ONE (CONS ZERO (CONS TWO NIL)))
=*> CONS TWO (CONS ONE (CONS THREE NIL))
```

You may assume that `INCRLIST` is called with a list constructed with `NIL` or `CONS`, and that all elements of the list are Church numerals. You must use recursion for full credit.

```
let INCRLIST1 = \rec -> \lst -> ITE _____(part 2(a))_______
                                   _____(part 2(b))_______
                                   _____(part 2(c))_______

let INCRLIST = ______(part 2(d))_______
```

a. (6 points) 2(a):

b. (6 points) 2(b):

c. (6 points) 2(c):

d. (6 points) 2(d):

Part 2: Haskell Types (24 points)

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

3. (6 points) What is the **type** of the following Haskell expression? (Choose only one answer.)

```
\x -> case x of
  Just _  -> "something"
  Nothing -> "nothing"
```

◯ This expression is ill-typed; it doesn't have a type
◯ `Maybe String`
◯ `String`
◯ `Maybe String -> String`
◯ `Maybe a -> String`

4. (6 points) Consider the following two expressions:

```
foldr (:) [] [True, False, True]
```

```
foldl (:) [] [True, False, True]
```

Which of the following statements is accurate? (Choose only one answer.) Hint: keep in mind that the type of `(:)` is `a -> [a] -> [a]`.

◯ Both expressions are well-typed
◯ Only the expression using `foldr` is well-typed
◯ Only the expression using `foldl` is well-typed
◯ Neither expression is well-typed

5. (6 points) What is the **type** of the following Haskell expression? (Choose only one answer.)

```
\x y -> if x == y then Nothing else Just y
```

◯ This expression is ill-typed; it doesn't have a type
◯ `Eq a => Maybe a -> Maybe a -> a`
◯ `Eq a => Maybe a -> Maybe a -> Maybe a`
◯ `Eq a => a -> a -> a`
◯ `Eq a => a -> a -> Maybe a`

6. (6 points) What is the **type** of the following Haskell expression? (Choose only one answer.)

```
\f -> map f ["rainbow", "sprinkles"]
```

◯ This expression is ill-typed; it doesn't have a type
◯ `(String -> b) -> [b]`
◯ `String -> b -> [b]`
◯ `(String -> String) -> [String]`
◯ `String -> String -> [String]`

Part 3: Working with Abstract Syntax Trees (46 points)

On section worksheet 5, we saw the following `Expr` data type, which defines a grammar of abstract syntax trees for a tiny language of strings. We will use this data type for most of the questions in this section.

```
data Expr = Var String    -- Variable
          | Str String    -- String literal
          | Cat Expr Expr -- Concatenate strings: `(++)`
          | Rev Expr      -- Reverse a string: `reverse`
```

For example, we would could represent the expression `(reverse (s ++ (reverse "cookie")))` with the following `Expr`:

```
Rev (Cat (Var "s") (Rev (Str "cookie")))
```

7. (8 points) Let us define the *depth* of an `Expr` as follows:

   - The depth of a variable is 1.
   - The depth of a string literal expression is 1.
   - The depth of a concatenation expression `Cat e1 e2` is 1 + the *maximum* of the depths of `e1` and `e2`.
   - The depth of a reverse expression `Rev e` is 1 + the depth of `e`.

   Define a Haskell function `depth` that takes an `Expr` and returns its depth 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 `depth` and one of the cases is provided for you below; fill in the rest of the definition.

```
depth :: Expr -> Int
depth (Var _) = 1
```

8. (6 points) Here is a function that computes the total depth of all the `Exprs` in a list (assuming that `depth` from the previous question is implemented):

```
depthAll :: [Expr] -> Int
depthAll []     = 0
depthAll (x:xs) = depth x + depthAll xs
```

Which of the following statements is accurate? (Choose only one answer.)

- ◯ `depthAll` is tail-recursive.
- ◯ `depthAll` is not tail-recursive, and it is not possible to implement tail-recursively.
- ◯ `depthAll` is not tail-recursive, but it could be implemented tail-recursively using an accumulator argument of type `Int`.
- ◯ `depthAll` is not tail-recursive, and it could *not* be implemented tail-recursively using an accumulator argument of type `Int`, but it *could* be implemented tail-recursively using continuation-passing style.

9. (6 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 `Expr` type. We will say that `Exprs` are equal **if they have the same depth**, and not equal otherwise.[1]

Implement an `Eq` instance for `Expr` 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, 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 Expr where
  (==) :: Expr -> Expr -> Bool
```

[1] This is a pretty strange definition of program equivalence, but let's roll with it!

10. (6 points) Here is the type of abstract syntax trees for a tiny language of arithmetic expressions, supporting integers, addition and subtraction operations, and variables, similar to what we've seen in lecture:

```
data ArithExpr = ANum Int
               | AAdd ArithExpr ArithExpr
               | ASub ArithExpr ArithExpr
               | AVar String
```

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

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

```
ghci> show (ANum 3)
"3"
ghci> show (AAdd (ANum 3) (AVar "x"))
"(3+x)"
ghci> show (AAdd (ASub (AVar "x") (AVar "y")) (ANum 2))
"((x-y)+2)"
ghci> show (AVar "y")
"y"
ghci> show (ASub (ASub (ANum 5) (ANum 3)) (AVar "z"))
"((5-3)-z)"
ghci> show (AAdd (ASub (ANum 5) (ANum 3)) (AAdd (ANum 2) (AVar "x")))
"((5-3)+(2+x))"
```

The implementation of `show` is below, with two cases already written for you. Fill in the remaining cases. (Hint: This is a lot like the pretty-printer for arithmetic expressions that you implemented for Assignment 2. Since strings in Haskell are just lists of characters, you can use the `(++)` function to concatenate strings together. Take note of how expressions are parenthesized in the examples above.)

```
instance Show ArithExpr where
  show :: ArithExpr -> String
  show (ANum n) = show n
  show (AVar s) = s
```

Finally, let's return to our tiny language of strings, and write an interpreter for it. Once again, here is the `Expr` type that this section began with:

```
data Expr = Var String
          | Str String
          | Cat Expr Expr
          | Rev Expr
```

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 variable names (which are `String`s) and their values (which are also `String`s, since every expression in this language evaluates to a string), using the following type alias:

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

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 (Str "larry") []
Just "larry"

ghci> eval (Rev (Str "larry")) []
Just "yrral"

ghci> eval (Rev (Var "x")) [("x", "the creature")]
Just "erutaerc eht"

ghci> eval (Rev (Var "x")) [("y", "cali")]
Nothing

ghci> eval (Cat (Rev (Var "x")) (Rev (Var "y"))) [("x", "mo"), ("y", "mo")]
Just "omom"

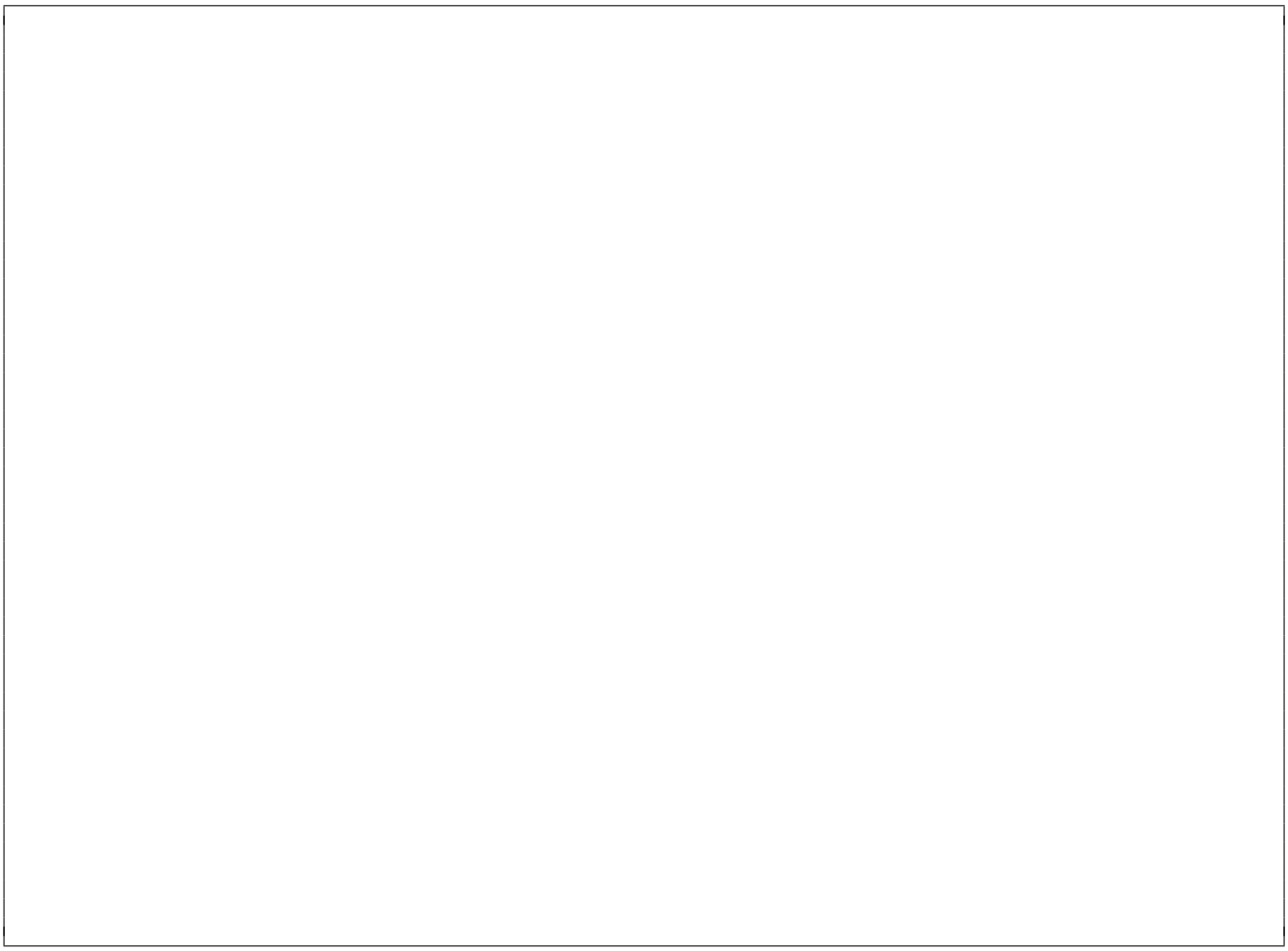
ghci> eval (Cat (Rev (Var "x")) (Rev (Var "y"))) [("x", "cookie")]
Nothing

ghci> eval (Rev (Rev (Rev (Rev (Var "s"))))) [("s", "phoebe")]
Just "phoebe"
```

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

11. (20 points) The type signature of our interpreter is provided below, with one case already written for you. Fill in the remaining cases. You can use library functions from the Haskell reference at the end of the exam, and you can also use the helper function at the bottom of the page.

```
eval :: Expr -> Env -> Maybe String
eval (Rev e) env = case eval e env of
  Just v -> Just (reverse v)
  _      -> Nothing
```



```
lookupInEnv :: String -> Env -> Maybe String
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
  ```

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

  The list constructor operator. Takes an element and a list, and constructs a new list with the specified element as its head and the specified list as its tail.

  ```
  > 3 : [4, 5]
  [3, 4, 5]
  > 3 : (4 : (5 : []))
  [3, 4, 5]
  ```

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

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

  ```
  > 3 + 4
  7
  ```

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

  Appends two lists, e.g.,

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

- `reverse :: [a] -> [a]`

  Returns the elements of its argument list in reverse order, e.g.,

  ```
  > reverse [2,5,7]
  [7,5,2]
  > reverse "hello"
  "olleh"
  ```
  ```

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

  Compares two arguments for equality, e.g.,

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

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

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

  ```
  > max 3 4
  4
  ```