CSE114A, Spring 2025: Midterm Exam 1

Instructor: Lindsey Kuper

April 25, 2025

Student name: _____

CruzID: _____@ucsc.edu

This exam has 13 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.

Good luck!

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

Part 1: Lambda Calculus (68 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 and/or α-renaming steps. Start each line with `=b>` or `=a>`, as if you were using Elsa, and do just one β-reduction step or α-renaming step per line.

   There may be multiple correct ways to reduce the given expression. A correct solution is any solution that Elsa would accept as correct.

   Hint: You will need to do α-renaming in at least one place.

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

Refer to the lambda calculus reference at the end of the exam for definitions used in this section.

2. (6 points) Which of the following lambda calculus expressions is in **normal form**?

   ◯ `(\x -> x x) (\x -> x x)`

   ◯ `\y -> (\x -> y)`

   ◯ `\y -> (\x -> x) y`

   ◯ `(\x -> y y) (\x -> y y)`

   ◯ `\x y -> (\x -> x) y`

3. (6 points) Which of the following lambda calculus expressions evaluates to `TRUE`?

   ◯ `TRUE TRUE TRUE`

   ◯ `TRUE TRUE`

   ◯ `ITE TRUE TRUE`

   ◯ `ITE TRUE FALSE TRUE`

   ◯ `FALSE TRUE FALSE`

4. (6 points) Which of the following lambda calculus expressions evaluates to `ONE`?

   ◯ `ITE TRUE ONE`

   ◯ `FST (PAIR (PAIR ONE ZERO) (PAIR ONE ZERO))`

   ◯ `ONE ZERO`

   ◯ `TRUE ZERO ONE`

   ◯ `TRUE ONE ZERO`

5. (6 points) What is the set of variables that **occur free** in the following lambda calculus expression?

```
(\q r -> r) y (\x y -> x (q r y))
```

   ◯ No variables occur free

   ◯ `q, r`

   ◯ `r, y`

   ◯ `y`

   ◯ `q, r, y`

For the rest of this section, you may use any of the helper functions defined in the provided lambda calculus reference at the end of the exam.

6. (7 points) Define a lambda calculus function `PAIRZERO` that takes a `PAIR` of two Church numerals (or expressions that evaluate to Church numerals) as its argument, and returns `TRUE` if both elements of the pair are equal to `ZERO`, and `FALSE` otherwise. For example, in Elsa:

```
PAIRZERO (PAIR ZERO ZERO) =~> TRUE
PAIRZERO (PAIR ONE ZERO) =~> FALSE
PAIRZERO (PAIR ZERO ONE) =~> FALSE
PAIRZERO (PAIR TWO THREE) =~> FALSE
PAIRZERO (PAIR (DECR ONE) (DECR (DECR TWO))) =~> TRUE
```

You may assume that `PAIRZERO` is only called with `PAIR`s of expressions that evaluate to Church numerals.

```
let PAIRZERO =
```

The next two questions 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 TRUE (CONS TRUE (CONS FALSE NIL))` is the lambda calculus equivalent of the Haskell list `[True, True, False]`,
which is just syntactic sugar for `True : (True : (False : []))`.

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 two questions.

7. (7 points) Define a lambda calculus function `FIRST2TRUE` that takes a list of Boolean expressions, and returns `TRUE` if the first two elements of the list are `TRUE`, and `FALSE` otherwise. You may assume that the argument to `FIRST2TRUE` is a list of *at least* two Boolean expressions. Any list elements beyond the first two should be ignored. For example, in Elsa:

```
FIRST2TRUE (CONS TRUE (CONS TRUE NIL)) =~> TRUE
FIRST2TRUE (CONS TRUE (CONS FALSE NIL)) =~> FALSE
FIRST2TRUE (CONS FALSE (CONS TRUE NIL)) =~> FALSE
FIRST2TRUE (CONS TRUE (CONS TRUE (CONS FALSE NIL))) =~> TRUE
FIRST2TRUE (CONS TRUE (CONS TRUE (CONS TRUE NIL))) =~> TRUE
FIRST2TRUE (CONS TRUE (CONS FALSE (CONS TRUE NIL))) =~> FALSE
```

```
let FIRST2TRUE =
```

8. Define a lambda calculus function ANDLIST that takes a list of Boolean expressions, and returns TRUE if all elements of the list evaluate to TRUE, and FALSE otherwise. For example, in Elsa:

```
ANDLIST NIL =~> TRUE
ANDLIST (CONS TRUE NIL) =~> TRUE
ANDLIST (CONS FALSE NIL) =~> FALSE
ANDLIST (CONS TRUE (CONS FALSE (CONS TRUE NIL))) =~> FALSE
ANDLIST (CONS TRUE (CONS TRUE (CONS FALSE NIL))) =~> FALSE
ANDLIST (CONS TRUE (CONS TRUE (CONS TRUE NIL))) =~> TRUE
```

You may assume that ANDLIST is called with a list constructed with NIL or CONS and that all elements in the list are Boolean expressions that evaluate to either TRUE or FALSE. You must use recursion for full credit.

```
let ANDLIST1 = \rec -> \lst -> ITE _____(part 8(a))_______
                                  _____(part 8(b))_______
                                  _____(part 8(c))_______

let ANDLIST = _______(part 8(d))_______
```

a. (6 points) 8(a):

b. (6 points) 8(b):

c. (6 points) 8(c):

d. (6 points) 8(d):

Part 2: Haskell (32 points)

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

9. (6 points) What is the **type** of the following Haskell expression?

```
[not (not (not False)), True] ++ [False, True]
```

- ◯ This expression is ill-typed; it doesn't have a type
- ◯ `Bool -> Bool -> Bool`
- ◯ `[Bool] -> [Bool] -> [Bool]`
- ◯ `Bool`
- ◯ `[Bool]`

10. (6 points) What is the **type** of the following Haskell expression?

```
(\x -> if x then ["rainbow"] else ["sprinkles"]) True
```

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

11. (6 points) What does the following Haskell expression **evaluate to**?

```
(\x -> x : x) True
```

- ◯ This expression is ill-typed; it doesn't evaluate to anything
- ◯ `[True, True]`
- ◯ `[True, [True]]`
- ◯ Infinite list: `[True, True, True, ...]`
- ◯ Infinite list: `[True, [True, [True, ...]]]`

For the next two questions, you can use the library functions provided in the Haskell reference at the end of the exam.

Haskell's standard library provides a list data type with two constructors, `[]` (pronounced "nil") and `(:)` (pronounced "cons"). For instance, the list `["phoebe","sam","larry"]` can be constructed with calls to the list constructors:

```
"phoebe" : ("sam" : ("larry" : []))
```

But we could also define our own list data type. For instance, here is a type of lists of `String`s:

```
data StrList = Nil | Cons String StrList
```

To construct a `StrList` with the strings `"phoebe"`, `"sam"`, and `"larry"`, in that order, we would use the `StrList` constructors as follows:

```
Cons "phoebe" (Cons "sam" (Cons "larry" Nil))
```

12. (7 points) Write a Haskell function `strListLength` that takes a `StrList` and returns its length as an `Int`. For example, in GHCi:

```
ghci> strListLength Nil
0

ghci> strListLength (Cons "phoebe" (Cons "sam" (Cons "larry" Nil)))
3
```

The type signature is provided for you. Hint: use pattern matching to deal with the two `StrList` constructors, `Nil` and `Cons`.

```
strListLength :: StrList -> Int
```

13. (7 points) Write a Haskell function `strListAppend` that takes two `StrLists` and returns a `StrList` that appends the provided `StrLists` together. For example, in GHCi:

```
ghci> strListAppend Nil Nil
Nil

ghci> strListAppend (Cons "james" (Cons "nella" Nil)) Nil
Cons "james" (Cons "nella" Nil)

ghci> strListAppend (Cons "james" (Cons "nella" Nil)) (Cons "mo" Nil)
Cons "james" (Cons "nella" (Cons "mo" Nil))

ghci> strListAppend (Cons "james" Nil) (Cons "nella" (Cons "mo" Nil))
Cons "james" (Cons "nella" (Cons "mo" Nil))
```

The type signature is provided for you.

Hint: use pattern matching to deal with the two `StrList` constructors, and use the constructors to match against the *first* argument to `strListAppend`.

```
strListAppend :: StrList -> StrList -> StrList
```

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

- `(:) :: 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
  ```

- `not :: Bool -> Bool`

  Boolean negation.

  ```
  > not True
  False
  > not (not True)
  True
  ```

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

  Appends two lists, e.g.,

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