# CSE114A, Fall 2023: Final Exam

Instructor: Owen Arden

December 14, 2023

Student name: ______________________________

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

This exam has 13 questions and 139 total points.

**Instructions**

- Please write directly on the exam.
- For multiple choice questions, **fill in the letter completely**, e.g. from ⓐ to ●
- For short response questions, try to keep your answer within the outlined box.
- **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 problem description, state your interpretation clearly and concisely. *Reasonable interpretations* will be taken into account by graders.

Good luck!

(this page intentionally left blank, you may use it for scratch paper but the contents will not be graded)

## Part 1: Lambda calculus

Question 1 (*5 points*)

Consider the following lambda expression `EXPR1`

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

1.1 (2 points) The free variables of expression `EXPR1` are :

- ⓐ `x` and `y`
- ⓑ `y` and `p`
- ⓒ `y` and `z`
- ⓓ `f` and `z`
- ⓔ None of the above

1.2 (3 points) Choose the best answer for `EXPR1`:

- ⓐ `EXPR1` is in normal form
- ⓑ After one $\beta$-reduction `EXPR1` will be in normal form
- ⓒ After two $\beta$-reductions `EXPR1` will be in normal form
- ⓓ After three $\beta$-reductions `EXPR1` will be in normal form
- ⓔ `EXPR1` does not have a normal form

Question 2 (*10 points*)

2.1 (5 points) What does the following lambda expression evaluate to ?

```
INC ((\x y z -> x (z y)) INC (PAIR ONE TWO) FST)
```

- ⓐ `ONE`
- ⓑ `TWO`
- ⓒ `THREE`
- ⓓ `FOUR`
- ⓔ `FIVE`

2.2 (5 points) What does the following lambda expression evaluate to ?

```
(\x y z -> ITE (FST (PAIR TRUE ONE)) (x z) (y z)) FST SND (PAIR ONE TWO)
```

- ⓐ `ONE`
- ⓑ `TWO`
- ⓒ `THREE`
- ⓓ `FOUR`
- ⓔ `FIVE`

## Part 2: Haskell

Question 3 (*9 points*)

Evaluate Haskell expressions.

3.1 (3 points) Consider the following Haskell expression

```
let sqrFun x = (sqr x) * (sqr x) in
  sqrFun 2
  where
    sqr = \x -> x * x
```

What is the result of evaluating this expression?

ⓐ `4`
ⓑ `8`
ⓒ `16`
ⓓ None of the above
ⓔ Syntax or type error
ⓕ Won't terminate

3.2 (3 points) Consider the following Haskell expression

```
let
  rev :: [Int] -> [Int]
  rev []     = []
  rev (x:xs) = (rev xs) : x
  in
    rev [1,2,3,4,5]
```

What is the result of evaluating this expression?

ⓐ `[1,2,3,4,5]`
ⓑ `[5,4,3,2,1]`
ⓒ `[1,2,3,4,5,5,4,3,2,1]`
ⓓ None of the above
ⓔ Syntax or type error
ⓕ Won't terminate

3.3 (3 points) Consider the following Haskell function:

```
buildList x =
  [ (i,j) | i <- [0..x],
            j <- [0..x]]
```

What does `buildList 2` evaluate to?

ⓐ `[(0,0),(0,1),(0,2),(1,0),(1,1),(1,2),(2,0),(2,1),(2,2)]`
ⓑ `[(0,0),(0,2),(2,0),(2,2)]`
ⓒ `[(0,0),(1,1),(2,2)]`
ⓓ None of the above
ⓔ Syntax or type error
ⓕ Won't terminate

## Part 3: Recursive Data Types

Question 4 (*18 points*)

Consider the following ADT that is used to represent a List

```
data List = Nil | Cons Int List
```

4.1 (3 points) instantiate the following list given the above definition: [1, 4, 3, 2]

4.2 (5 points) implement a function listLength, which returns the length of a given list.

4.3 (5 points) Define a function sumList, which returns the sum of the elements in the list

4.4 (5 points) The function `isListIncreasing` below determines whether a list of integers are sorted in increasing order.

```
isListIncreasing :: List -> Bool
isListIncreasing Nil = True
isListIncreasing (Cons x xs) = helper x xs
where
  helper x Nil = True
  helper x (Cons y ys) = if x > y then False else helper y ys
```

What should be the type signature of the `helper` function?

ⓐ `helper :: [Int] -> Bool`

ⓑ `helper :: [List] -> Int -> Bool`

ⓒ `helper :: Int -> List -> Bool`

ⓓ `helper :: [List] -> [Int] -> Bool`

ⓔ None of the above

## Part 4: Higher-order Functions

Question 5 (*16 points*)

Higher-order Functions.

5.1 (5 points) Consider the following Haskell expression:

```
foldr (-) 0 [1,2,3,4,5]
```

What is the result of evaluating this expression?

**Hint**: You may find the implementation of `foldr` in the cheat sheet; evaluate the expression by hand to find the answer.

- ⓐ `-5`
- ⓑ `3`
- ⓒ `-15`
- ⓓ None of the above
- ⓔ Syntax or type error
- ⓕ Won't terminate

5.2 (5 points) Consider the following Haskell expression:

```
foldl (-) 0 [1,2,3,4,5]
```

What is the result of evaluating this expression?

**Hint**: You may find the implementation of `foldl` in the cheat sheet; evaluate the expression by hand to find the answer.

- ⓐ `-5`
- ⓑ `3`
- ⓒ `-15`
- ⓓ None of the above
- ⓔ Syntax or type error
- ⓕ Won't terminate

5.3 (3 points) Consider the following Haskell expression:

```
map (\x -> (x: x * x)) [0,1,2,3,4,5]
```

What is the result of evaluating this expression?

- ⓐ `{0:0,1:1,2:2,3:3,4:4,5:5}`
- ⓑ `{0:0,1:1,2:4,3:9,4:16,5:25}`
- ⓒ `[0,1,4,9,16,25]`
- ⓓ None of the above
- ⓔ Syntax or type error
- ⓕ Won't terminate

5.4 (3 points) Consider the following Haskell function:

```
mapFilter ls = map (filter (\x -> (x `mod` 2) /= 0)) ls
```

What does `mapFilter [[1,2,3,4,5]]` evaluate to?

- ⓐ `[1,3,5]`
- ⓑ `[[1,3,5]]`
- ⓒ `[[1],[3],[5]]`
- ⓓ None of the above
- ⓔ Syntax or type error
- ⓕ Won't terminate

## Part 5: Semantics, scope, environments

Question 6 (*6 points*)

Consider the following Nano program:

```
let a = 1 in
  let b = 2 in
    let f = \x y -> x + y + a + b + c in
      let a = 3 in
        let c = 4 in
          f a b
```

6.1 (3 points) Under **static scope**, what would the above program evaluate to?

- ⓐ `10`
- ⓑ `12`
- ⓒ `14`
- ⓓ error: unbound variable

6.2 (3 points) Under **dynamic scope**, what would the above program evaluate to?

- ⓐ `10`
- ⓑ `12`
- ⓒ `14`
- ⓓ error: unbound variable

Question 7 (*10 points*)

Consider the following Nano program:

```
let a = 1 in
  let b = 2 in
    let f1 = \x y -> x + y + a in
      let f2 = \x y -> x - y - b in
        let a = f1 a b in
          let b = f1 a b in
            f2 a b
```

7.1 (5 points) Under **static scope**, what would the above program evaluate to?

- ⓐ -5
- ⓑ -10
- ⓒ -16
- ⓓ error: unbound variable

7.2 (5 points) Under **dynamic scope**, what would the above program evaluate to?

- ⓐ -5
- ⓑ -10
- ⓒ -16
- ⓓ error: unbound variable

Question 8 (*10 points*)

Consider the following Nano language

```
e ::= x | v | e1 + e2 |
      let x = e1 in e2 |
      \x -> e | e1 e2
v ::= n | \x -> e
where n ∈ ℕ, x ∈ Var
```

and the following operational semantics for the Nano language

```
                  e1 => e1'
[Add-L] -----------------------------
              e1 + e2 => e1' + e2


                  e2 => e2'
[Add-R] -----------------------------
              n1 + e2 => n1 + e2'


[Add] n1 + n2 => n           where n == n1 + n2


                       e1 => e1'
[Let-Def] -----------------------------------------
            let x = e1 in e2 => let x = e1' in e2

[Let] let x = v in e2 => e2[x := v]

              e1 => e1'
[App-L] --------------------
           e1 e2 => e1' e2


              e => e'
[App-R] ------------------
             v e => v e'


[App] (\x -> e) v => e[x := v]
```

(the cases for value substitution are given in the appendices)

8.1 (5 points) Which of the following reductions are valid ?

ⓐ `let x=9+1 in x+1 => let x=10 in x+1`

ⓑ `let x=10 in x+9 => 10+9`

ⓒ `let x=9 in (let y=5+6 in x+y) => let x=9 in (let y=11 in x+y)`

ⓓ a and b

ⓔ All of the above

8.2 (5 points) Which of the following reductions are valid ?

ⓐ `(\x y -> let z=y+1 in x+z)(3+4)(5+6)`
`=> (\y -> let z=y+1 in 3+4+z)(5+6)`

ⓑ `(\x y -> let z=y+1 in x+z)(3+4)(5+6)`
`=> (\x y -> let z=y+1 in x+z)(7)(5+6)`

ⓒ `(\y -> let z=y+1 in 7+z) (5+6)`
`=> (let z=(5+6)+1 in 7+z)`

ⓓ `(\y -> let z=y+1 in y+z)(5+6)`
`=> (\y -> let z=y+1 in y+z)(11)`

ⓔ b and d

Question 9 (*10 points*)

Consider the following grammar for Nano1

**Grammar**

```
e ::= x | v
| e1 + e2
| let x = e1 in e2
v ::= n
where n ∈ ℕ, x ∈ Var
```

Let the sizes for the terms in our grammar be the:

**Term Size**

```
size n = 1
size x = 1
size (e1 + e2) = 1 + size e1 + size e2
size (let x = e1 in e2) = size e1 + size e2
```

9.1 (5 points) Consider the Lemma and its corresponding proof below

**Lemma:** For any e, size e $>$ 0

**Proof:** By induction on the term e

- Base case 1: size n = 1 $>$ 0
- Base case 2: size x = 1 $>$ 0
- Inductive case 1: size (e1 + e2) = 1 + size e1 + size e2 $>$ 0 because size e1 $>$ 0 and size e2 $>$ 0 by IH
■

What is the inductive hypothesis (IH)?

ⓐ `size e1 > 0 and size e2 > 0`

ⓑ `size e = 1`

ⓒ `size e1 + size e2 > 0`

ⓓ `size n = 1 and size x = 1`

ⓔ None of the above

9.2 (5 points) The above proof is missing the `let` case. In the space below, complete the proof using the same format as the other cases above.

## Part 6: Type, type-inference, type-classes

Question 10 (*15 points*)

General Unifiers

10.1 (5 points) What is a unifier for the following types?

`a -> b` and `c -> Int -> String`

- ⓐ `[a / c, b / Int -> String]`
- ⓑ `[a / c -> Int, b / String]`
- ⓒ `[a / Bool, b / Int -> String, c / Bool]`
- ⓓ `(a) and (b)`
- ⓔ `(a) and (c)`
- ⓕ `(b) and (c)`
- ⓖ Cannot unify

10.2 (5 points) What is a unifier for the following types?

`a -> Int` and `b -> Int -> Int`

- ⓐ `[a / Int, b / Int -> Int]`
- ⓑ `[a / Int -> Int, b / Int]`
- ⓒ `[a / Int, b / Int]`
- ⓓ `[a / Int -> Int, b / Int -> Int]`
- ⓔ Cannot unify

10.3 (5 points) Consider the following types: `a -> Int -> Int` and `b -> c`.

Is the following unifier a **most** general unifier? `[a / Int, b / Int, c / Int -> Int]`

- ⓐ Yes
- ⓑ No, a most general unifier is `[b / a, c / Int -> Int]`
- ⓒ No, a most general unifier is `[a / Int, b / Int -> Int, c /Int]`
- ⓓ Cannot unify
- ⓔ None of the above

Question 11 (*6 points*)

Let us extend our grammar for Nano1 to be

### Grammar

```
e ::= x | v
| e1 + e2
| e1 * e2
| let x = e1 in e2
v ::= n
where n ∈ ℕ, x ∈ Var
```

### Types

Types are represented by the following grammar:

```
T := Int | T1 -> T2
```

### Type system

Below is a partial type system for this language.

The above rules are missing a rule for typing multiply expressions. Fill in the missing parts of the `T-Mul` rule below.

```
[T-Num] --------------------        [T-Var] ----------------------------
           G |- n :: Int                                 G |- x :: T


           G |- e1 :: Int  G |- e2 :: Int
[T-Add] ------------------------------------
           G |- e1 + e2 :: Int


           G |- e1 :: T1 G,x:T1 |- e2 :: T2
[T-Let] ------------------------------------
           G |- let x = e1 in e2 :: T2



           G |- (a)        G |-  (b)
   [T-Mul] --------------------------
                      (c)
```

11.1 (2 points) (a)

11.2 (2 points) (b)

11.3 (2 points) (c)

Question 12 (*9 points*)

Below is a partial typing derivation that shows that a Nano1 expression `5 + (let x = 9 in x+1)` has type `Int`. For each blank, fill in a type, the name of a typing rule, or the whole typing judgement (premise) to complete the typing derivation.

```
                                                          [T-Var]----------   [(d)]-----------
                                                             x:Int |- x::Int    x:Int |- 1:: Int
                                       [T-Num]-----------   [T-Add]---------------------
                                          [] |- 9 :: Int                  (c)
 [T-Num]-----------------  [T-Let]---------------------------------------
        [] |-  5 :: Int            [] |- let x = 9 in x+1 :: (b)
[(a)]----------------------------------------------------------------------
                                       [] |- 5 + (let x = 9 in x+1) :: Int
```

12.1 (2 points) (a)

12.2 (2 points) (b)

12.3 (3 points) (c)

12.4 (2 points) (d)

Question 13 (*15 points*)

Consider the three data types as follows

```
data Circle = Circle{r::Double}
data Rectangle = Rectangle{w::Double, l::Double}
data Triangle = Triangle{b::Double, h::Double}
```

and the following `ShapeArea` class

```
class ShapeArea a where
    area :: a -> Double
```

13.1 (10 points) Create instances for the typeclass `ShapeArea` for each data type `Circle`, `Rectangle` and `Triangle`. The `area` function returns area of the given shape. The area of a circle is calculated as (3.14*`radius`*`radius`), the area of a rectangle is calculated as (`width * height`), and the area of a triangle is calculated as (`0.5 * base * height`).

13.2 (5 points) Write a Haskell function named `sumArea` that takes a list of type `a`, where `a` is an instance of `ShapeArea`, and returns sum of the areas.
E.g. `sumArea [(Rectangle 2.0 3.0),(Rectangle 10.0 2.0)]` returns `26.0`,
`sumArea [(Triangle 2.0 3.0),(Triangle 10.0 2.0)]` returns `13.0`.

(this page intentionally left blank, you may use it for scratch paper but the contents will not be graded)

# 1 Lambda calculus cheat sheet

```
-- Booleans --------------------------------
let TRUE =\x y -> x
let FALSE = \x y -> y
let ITE = \b x y -> b x y
let NOT = \b x y -> b y x
let AND = \b1 b2 -> ITE b1 b2 FALSE
let OR = \b1 b2 -> ITE b1 TRUE b2

-- Numbers ---------------------------------
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))))

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

-- Arithmetic ------------------------------
let INC   = \n f x -> f (n f x)
let ADD   = \n m -> n INC m
let MUL   = \n m -> n (ADD m) ZERO
let ISZ   = \n -> n (\z -> FALSE) TRUE
let DECR  = \n ->   -- decrement n by one --
let EQL   = \a b -> -- return TRUE if a == b, otherwise FALSE --

-- Recursion -------------------------------
let FIX = \stp -> (\x -> stp (x x)) (\x -> stp (x x))
```

## 2 Haskell cheat sheet

```
data  Maybe a  =  Nothing | Just a

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 b xs          = helper b xs
  where
    helper acc []     = acc
    helper acc (x:xs) = helper (f acc x) xs

filter :: (a -> Bool) -> [a] -> [a]
filter p []    = []
filter p (x:xs)
  | p x         = x : filter p xs
  | otherwise     = filter p xs

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

flip :: (a -> b -> c) -> b -> a -> c
flip f x y = f y x

(.) :: (b -> c) -> (a -> b) -> a -> c
(.) f g x = f (g x)

(++) :: [a] -> [a] -> [a]
(++) []     ys = ys
(++) (x:xs) ys = x : xs ++ ys

-- returns the elements of a list in reverse order.
reverse :: [a] -> [a]

-- Extract the first element of a list, which must be non-empty.
head :: [a] -> a

-- Extract the elements after the head of a list, which must be non-empty.
tail :: [a] -> [a]

-- Extract the first n elements of a list.
take :: Int -> [a] -> [a]
```

## 3 Value substitution cheat sheet

```
x[x := v] = v
y[x := v] = y -- assuming x /= y
n[x := v] = n
(e1 + e2)[x := v] = e1[x := v] + e2[x := v]
(let x = e1 in e2)[x := v] = let x = e1[x := v] in e2
(let y = e1 in e2)[x := v] = let y = e1[x := v] in e2[x := v]
```

(this page intentionally left blank, you may use it for scratch paper but the contents will not be graded)

(this page intentionally left blank, you may use it for scratch paper but the contents will not be graded)