CSE114A, Fall 2024: Midterm Exam

Instructor: Owen Arden

November 1, 2024

Student name: ______________________________

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

Additionally, please write your CruzID at the top of each page.

This exam has 5 questions and 70 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 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.

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 (*12 points*)

Consider the following lambda calculus expression, which we will name `EXPR1`

```
(\x y z -> ITE (OR a b) c
                ((\w -> ITE (ISZ w) (ADD x y) w) FOUR)) ONE TWO
```

1.1 (3 points) What are the free variables of `EXPR1`:

ⓐ `x`, `y`, and `z`

ⓑ `a`, `b`, and `c`

ⓒ `x`, `y`, `z`, and `w`

ⓓ Choices (b) and (c)

ⓔ None of the above

1.2 (4 points) After a single beta-reduction on `EXPR1`, what would the resulting expression be?:

ⓐ
```
(\y z -> ITE (OR a b) c
        ((\w -> ITE (ISZ w) (ADD ONE y) w) FOUR)) TWO
```

ⓑ
```
(\x y z -> ITE (OR a b) c
          (ITE (ISZ FOUR) (ADD x y) FOUR)) ONE TWO
```

ⓒ
```
(\y z -> ITE (OR a b) c
        (ITE (ISZ FOUR) (ADD ONE y) FOUR)) TWO
```

ⓓ Choices (a) and (b)

ⓔ Choices (a) and (c)

1.3 (5 points) What is the normal form of `EXPR1`:

ⓐ `THREE`

ⓑ `FOUR`

ⓒ `\z -> (a TRUE b) c FOUR`

ⓓ `\z -> ITE (OR a b) c FOUR`

ⓔ None of the above

Question 2 (*12 points*)

Recall that =d> in ELSA denotes unfolding a definition. Suppose a new ELSA operator =b~> denotes a sequence of one or more **beta-reductions** ending in an expression that cannot be beta-reduced without expanding a definition.

Reduce the following lambda expression to normal form using =b~> and =d>.

- **Hint:** =b~> is a little like =~> but can only take beta-reduction steps. Use =b~> to reduce expressions that contain a redex (the initial expression below is an example of a redex) until you reach an expression that can't be beta-reduced because the definitions haven't been expanded. Then, expand a definition with =d> and continue to beta-reduce with =b~> until you reach a normal form with no remaining definitions.

```
(\x -> ITE x FIVE TWO) FALSE
```

Solution:

```
(\x -> ITE x FIVE TWO) FALSE
=b~> ITE FALSE FIVE TWO
=d>  (\b x y -> b x y) FALSE FIVE TWO
=b~> FALSE FIVE TWO
=d>  (\x y -> y) FIVE TWO
=b~> TWO
=d> \f x -> f (f x)
```

Rubrics:

- +3 points if there is a step =b~> `ITE FALSE FIVE TWO`
- +4 points if there is a step =b~> `FALSE FIVE TWO`
- +5 points if there is a step =b~> `TWO`

NOTE: Do not deduct points if the final =d> is missing

NOTE: Alternative sequences of reduction are possible. In particular, expanding all definitions and then using a single =b~>. We will give those answers full points for discovering such a loophole in the question.

Part 2: Haskell

Question 3 (*10 points*)

Consider the following Haskell expression

```
let
  acc = (0, 0)
  val = [(0, 1), (2, 3), (4, 5), (6, 7), (8, 9)]
  foldx = foldr f1 acc val
  in
    foldl f2 (12,34) [foldx]
  where
    f1 (x, y) (u, w) = (x + u, y + w)
    f2 (x, y) (u, w) = (x - u, y - w)
```

3.1 (5 points) What is the type of `f1`?

- ⓐ `[Int] -> [Int] -> [Int]`
- ⓑ `(Int, Int) -> (Int, Int) -> (Int, Int)`
- ⓒ `Int -> Int -> (Int, Int)`
- ⓓ None of the above
- ⓔ Syntax or type error

3.2 (5 points) What is the type of `foldx`?

- ⓐ `(Int, Int)`
- ⓑ `[Int]`
- ⓒ `Int`
- ⓓ None of the above
- ⓔ Syntax or type error

Question 4 (*24 points*)

Recall that the Fibonacci sequence is a sequence in which each number is the sum of the two preceding ones. Starting from 0 and 1, the sequence will be $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \cdots$

Please complete the implementation below of a function that returns the N-th number in Fibonacci sequence that starts from 0 and 1.

```
fib :: Int -> Int
fib n
    | blank_1  = 0
    | blank_2  = 1
    | blank_3  = (fib (n - 1)) + (fib (n - 2))
```

4.1 (3 points) Please fill in the blanks ***blank_1*** in the box below:

Solution:

```
n == 0
```

NOTE: `n <= 0` is also acceptable.

4.2 (3 points) Please fill in the blanks ***blank_2*** in the box below:

Solution:

```
n == 1
```

NOTE: `n <= 1` is also acceptable.

4.3 (3 points) Please fill in the blanks ***blank_3*** in the box below:

Solution:

```
otherwise
```

NOTE: `n > 1` is also acceptable.

4.4 (5 points) Next, please complete the implementation below of the same function but using tail recursion

```
fibTR :: Int -> Int
fibTR n = helper n (0,1)
    where
        helper :: Int -> (Int, Int) -> Int
        helper 0 (a,_) = a
        helper n (a,b) = blank_1
```

Please fill in the blank ***blank_1*** in the box below:

Solution:

```
helper (n-1) (b, a+b)
```

4.5 (10 points) In addition to the Fibonacci sequence, Factorial sequence is another famous number sequence. Below is the function that returns the N-th number in the Factorial sequence that starts from 0.

```
fac :: Int -> Int
fac n
    | n <= 1 = 1
    | otherwise = n * fac (n - 1)
```

Now, please implement the function:

```
seqArray :: (Int -> Int) -> Int -> [Int]
```

`seqArray` accepts a number sequence function and an integer representing N, and returns an array of the given number sequence from 0 to the N-th number.

For example, `seqArray fib 10` should returns `[0,1,1,2,3,5,8,13,21,34,55]`

And `seqArray fac 5` should returns `[1,1,2,6,24,120]`

Please implement `seqArray` using tail recursion if you can. 7 points for correct implementation, 3 points for correct tail recursive implementation, total 10 possible points.

Solution:

```
seqArray f n = helper n []
  where
      helper 0 acc = (f 0):acc
      helper n acc = helper (n - 1) ((f n):acc)
```

Rubrics:

- -10 points if no answer is provided, or the answer is not in something close to Haskell (eg Python)
- -3 to -7 points if the code does not provide the functionality we asked for
- -3 points if the function is **not tail recursive**
- **minor syntax errors are acceptable**
- -2 points if the intended meaning of the code is ambiguous (due to major syntax errors etc), but likely indicates an understanding of the correct implementation.

Question 5 (*12 points*)

5.1 (6 points) Consider the following data type and function definition. What typeclass instances, if any, are required for the following function to typecheck? Just list names, no definitions are required.

```
data Triangle =
    ASA Int Int Int
  | SSS Int Int Int
  | SAS Int Int Int

maxTriangle :: Triangle -> [Triangle] -> Triangle
maxTriangle m [] = m
maxTriangle m (t:ts) | m >= t = maxTriangle m ts
                     | otherwise = maxTriangle t ts
```

Solution:

- +3 points for `Eq`
- +3 points for `Ord`
- -1 point for one or more unnecessary typeclasses

5.2 (6 points) Consider the following function

```
sameSide :: Triangle ->  Triangle -> Bool
sameSide (ASA _ y _) (ASA _ y' _) = y == y'
sameSide (SSS x y z) (ASA _ y' _) = x == y' || y == y' || z == y'
sameSide (SAS x _ z) _ = True
sameSide _ (ASA _ _ _) = False
```

What would the following expression evaluate to?

```
sameSide (ASA 1 2 3) (SSS 2 2 2)
```

Ⓐ **True**
Ⓑ **False**
Ⓒ Type error
Ⓓ Runtime error
Ⓔ None of the above

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

class Eq a  where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool

class (Eq a) => Ord a where
  (<)  :: a -> a -> Bool
  (<=) :: a -> a -> Bool
  (>)  :: a -> a -> Bool
  (>=) :: a -> a -> Bool
  ...
```