CSE114A, Fall 2022: Midterm Exam

Instructor: Owen Arden

October 27, 2022

Student name: _____

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

This exam has 6 questions and 123 total points.

Instructions

- Please write directly on the exam.
- For multiple choice questions, **fill in the letter completely**, e.g. from (a) to ●
- **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.

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

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

```
(\b -> ITE b trick treat) (OR (\x y -> x) (\x y -> y)) candy prank
```

   a. (5 points) What are the free variables of `EXPR1`?
      (a) `b, x, y`
      (b) `candy, prank`
      (c) `trick, treat`
      (d) Choices (b) and (c)
      (e) Choices (a) and (b)

   b. (5 points) After a *single* β-reduction step on `EXPR1`, what would the resulting expression be?
      (a) `(\b -> ITE b trick treat) (OR candy (\x y -> y)) prank`
      (b) `(ITE (OR (\x y -> x) (\x y -> y)) trick treat) candy prank`
      (c) `(\b -> ITE b trick treat) (\x y -> x) candy prank`
      (d) Choices (b) or (c)
      (e) None of the above

   c. (5 points) What is the *normal form* of `EXPR1`?
      (a) `trick`
      (b) `treat`
      (c) `treat candy`
      (d) `trick candy prank`
      (e) None of the above

2. Consider the following lambda calculus expression, which we will name `EXPR2`:

```
let F = (\rec l -> rec (PAIR (PLUS (FST l) (FST (SND l))) l))
(FIX F) (PAIR ONE (PAIR ZERO FALSE))
```

   a. (5 points) What does `EXPR2` evaluate to?
      (a) `(PAIR TWO (PAIR ONE (PAIR ONE (PAIR ZERO FALSE))))`
      (b) `F (FIX F) (PAIR ONE (PAIR ZERO FALSE))`
      (c) `STEP (FIX STEP)`
      (d) `EXPR2` does not have a normal form.
      (e) `EXPR2` does not type check.

   b. (5 points) Which of the following reductions of `EXPR2` is valid? (Recall `=*>` denotes any sequence of `=a>`, `=b>`, or `=d>` reductions.)
      (a)
```
=*> F (F (F (FIX F)))
      (PAIR TWO (PAIR ONE (PAIR ONE (PAIR ZERO FALSE))))
```
      (b)
```
=*> (FIX F)
      (PAIR TWO (PAIR ONE (PAIR ONE (PAIR ZERO FALSE))))
```
      (c)
```
=*> F (FIX F)
      (PAIR FIVE (PAIR THREE
       (PAIR TWO (PAIR ONE (PAIR ONE (PAIR ZERO FALSE))))))
```
      (d) All of the above.
      (e) None of the above.

Part 2: Haskell

3. (8 points) What does the following Haskell expression evaluate to?

```
let
    f x = x + 1
    g = \f x -> f (f x)
    h f = \y -> y (y f)
    i k = k
    k = \a b -> a i b
in
  k g (flip h) f 0
```

   (a) 0

   (b) 2

   (c) 4

   (d) Infinite loop

   (e) Type Error

4. Consider the following Haskell function:

```
mkpal s = s ++ (helper s)
  where
    helper []     = ""
    helper (c:cs) =  (helper cs) ++ [c]
```

   a. (5 points) What is the type of `mkpal`?

      (a) `String -> [String]`

      (b) `[String] -> [String]`

      (c) `String -> String`

      (d) `[String] -> [Char]`

      (e) None of the above

   b. (5 points) What does `mkpal "foobarbaz"` evaluate to?

      (a) `"foobarbaz"`

      (b) `"zabraboof"`

      (c) `"foobarbazbazbarfoo"`

      (d) `"foobarbazzabraboof"`

      (e) None of the above

   c. (5 points) Is `mkpal` tail recursive?

      (a) Yes

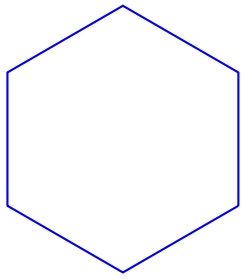
      (b) No, but a tail recursive implementation **is not possible**.

      (c) No, but a tail recursive implementation **is possible**.

5. *Turtle graphics*[1] are an approach to drawing figures by directing a cursor (or "turtle") around a canvas using simple set of commands. For example, the program below would draw a (regular) hexagon with sides of length 50 and internal angles of 120 degrees since turning the turtle left 60 degrees creates an angle of $180 - 60 = 120$.

```
repeat 6 {
  forward 50
  rotate_left 60
}
```

(a) Turtle program



(b) Program output

For the next questions, consider the following definitions that describe a simple language for turtle graphics.

```
data TurtleCmd = Home                -- return to starting position and orientation
         | Forward Int               -- move turtle forward by Int pixels
         | Back Int                  -- move turtle forward by Int pixels
         | RotateL Int               -- rotate left by Int degrees
         | RotateR Int               -- rotate right by Int degrees
         | Repeat Int [TurtleCmd]    -- repeat a list of commands Int times
type Plan = [TurtleCmd]
```

a. (5 points) In the box below, write a `Plan` corresponding to the hexagon-drawing program above.

[1] Turtle graphics were a central part of the *Logo* programming language. Look it up!

b. (20 points) Now help complete an interpreter for a `Plan` program. For each basic TurtleCmd there is a `do` function that applies the command to a `Canvas` value representing the state of the drawing and turtle. You don't have to worry about the definition of `Canvas`, just that you must provide the current canvas to each `do` function. You may use the second box for any helper functions.

Hint: Note that there is (intentionally) no `do` function for `Repeat`: you must write Haskell code to evaluate `Repeat` commands. You do not need the definition of `Canvas` to accomplish this.

```
doHome     :: Canvas -> Canvas
doForward  :: Canvas -> Int -> Canvas
doBack     :: Canvas -> Int -> Canvas
doRotateL  :: Canvas -> Int -> Canvas
doRotateR  :: Canvas -> Int -> Canvas

eval :: Canvas -> Plan -> Canvas
eval c [] =                          -- fill in the base case here

eval c (cmd:rest) =
  let c' = case cmd of -- fill in the body of the case function below
```

```
  in
    eval c' rest   -- use the below box for helper functions or where clauses
```

6. (25 points) Consider the following higher-order function `gen`:

```
gen :: (b -> Maybe (a, b)) -> b -> [a]
gen f b0 =
  let go b = case f b of
               Just (a, new_b) -> a : go new_b
               Nothing         -> []
  in go b0
```

a. (5 points) Choose the best answer:
   (a) `gen` always terminates (on any input)
   (b) `gen` never terminates (on any input)
   (c) `gen` may or may not terminate
   (d) `gen` terminates if and only if `f b` terminates
   (e) None of the above

b. (5 points) What does
   `gen (\b if b >= 5 then Nothing else Just (b,b+1)) 10` evaluate to?
   (a) [10,9,8,7,6]
   (b) [6,7,8,9,10]
   (c) []
   (d) Does not type check
   (e) Does not terminate

c. (5 points) What does
   `gen (\b if b >= 10 then Nothing else Just (b,b+1)) 5` evaluate to?
   (a) [9,8,7,6,5]
   (b) [5,6,7,8,9]
   (c) []
   (d) Does not type check
   (e) Does not terminate

d. (5 points) What does
   `take 5 (gen (\b if b < 5 then Nothing else Just (b,b+1)) 10)` evaluate to?
   (a) [5,6,7,8,9]
   (b) [10,11,12,13,14]
   (c) []
   (d) Does not type check
   (e) Does not terminate

e. (5 points) What does

```
helper []     = Nothing
helper (x:xs) = Just (x:xs)
gen helper (foldr (:) [] [6,7,8,9,10])
```

   evaluate to?
   (a) [10,9,8,7,6]
   (b) [6,7,8,9,10]
   (c) []
   (d) Does not type check
   (e) Does not terminate

(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 ->   -- return TRUE if n == 0 --
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

```haskell
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]
```

(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)