CSE114A, Fall 2023: Midterm Exam

Instructor: Owen Arden

October 27, 2023

Student name: _____

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

This exam has 5 questions and 95 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 (*15 points*)

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

```
(\x y -> ITE b (ADD x y)
               ((\z -> ITE (ISZ z) x (ADD x z)) FALSE)) FOUR ONE
```

1.1 (5 points) What are the free variables of `EXPR1`?

ⓐ `x`

ⓑ `x, y`

ⓒ `b`

ⓓ Choices (b) and (c)

ⓔ None of the above

1.2 (5 points) After a *single* β-reduction step on `EXPR1`, what would the resulting expression be?

ⓐ
```
(\y -> ITE b (ADD FOUR y)
             ((\z -> ITE (ISZ z) FOUR (ADD FOUR z)) FALSE)) ONE
```

ⓑ
```
(\y -> ITE b (ADD FOUR y)
             (\z -> ITE (ISZ z) FOUR (ADD FOUR z))) ONE
```

ⓒ
```
(\x y -> ITE b (ADD x y)
               (ITE (ISZ FALSE) x (ADD x FALSE))) FOUR ONE
```

ⓓ Choices (a) and (b)

ⓔ Choices (a) and (c)

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

ⓐ `FOUR`

ⓑ `FIVE`

ⓒ `b FIVE FOUR`

ⓓ `(\z -> ITE (ISZ z) FOUR (ADD FOUR z))`

ⓔ None of the above

Question 2 (*15 points*)

Reduce the following expression to normal form step-by-step using `=a>`, `=b>`, and `=d>`.

```
(\x y -> ITE (ISZ y) x (ADD x y)) FIVE ZERO
```

Solution: Rubric:

- no deduction for skipped `=d>` steps
- -1 for invalid step (based on the current term, whether previous steps were correct or not)
- -5 if not normal form or incorrect answer

```
=b>
(\y -> ITE (ISZ y) FIVE (ADD FIVE y)) ZERO
=b>
(ITE (ISZ ZERO) FIVE (ADD FIVE ZERO))
=d>
(ITE ((\n -> n (\z -> FALSE) TRUE) ZERO) FIVE (ADD FIVE ZERO))
=b>
(ITE (ZERO (\z -> FALSE) TRUE) FIVE (ADD FIVE ZERO))
=d>
(ITE ((\f x -> x) (\z -> FALSE) TRUE) FIVE (ADD FIVE ZERO))
=b>
(ITE ((\x -> x) TRUE) FIVE (ADD FIVE ZERO))
=b>
(ITE TRUE FIVE (ADD FIVE ZERO))
=d>
((\b x y -> b x y) TRUE FIVE (ADD FIVE ZERO))
=b>
((\x y -> TRUE x y) FIVE (ADD FIVE ZERO))
=b>
((\y -> TRUE FIVE y) (ADD FIVE ZERO))
=b>
TRUE FIVE (ADD FIVE ZERO)
=d>
(\x y -> x) FIVE (ADD FIVE ZERO)
=b>
(\y -> FIVE) (ADD FIVE ZERO)
=b>
FIVE
```

Question 3 (*15 points*)

Consider the following Haskell function:

```
listFun xs = helper ([], []) xs
  where
    helper (arr1, arr2) [] = (arr1, arr2)
    helper (arr1, arr2) (x:xs)
      | x `mod` 2 == 0 = helper (arr1 ++ [x], arr2) xs
      | otherwise = helper (arr1, arr2 ++ [x]) xs
```

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

ⓐ `Int -> [Int]`

ⓑ `[Int] -> [Int]`

ⓒ `Int -> (Int, Int)`

ⓓ `[Int] -> ([Int], [Int])`

ⓔ None of the above

3.2 (5 points) What does `listFun [1, 4, 9, 12, 21, 33, 4, 52]` evaluate to?

ⓐ `[1, 4, 4, 9, 12, 21, 33, 52]`

ⓑ `149122133452`

ⓒ `([52, 4, 12, 4], [33, 21, 9, 1])`

ⓓ `([4, 12, 4, 52], [1, 9, 21, 33])`

ⓔ None of the above

3.3 (5 points) Is `listFun` tail recursive?

ⓐ No, but a tail recursive implementation **is possible**.

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

ⓒ Yes

Question 4 (*28 points*)

Consider the binary search tree below where each internal node has a value i and two child subtrees. The left child contains all nodes with values less than or equal to i, and the right child contains all nodes with values strictly greater than i.

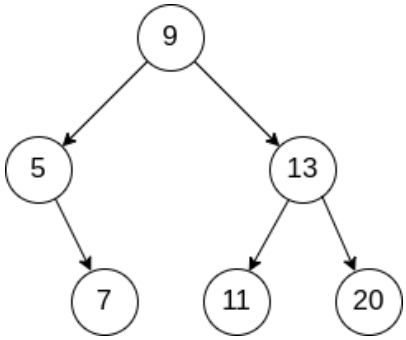

Figure 1: A binary search tree

Given the following ADT that is used to represent the tree above, answer the questions below. **You may not use library functions except for `(+)` and `(++)`.**

```
data Tree = Leaf | Node Int Tree Tree
```

4.1 (7 points) Define a function sumTree, which returns the sum of all of the values in the nodes within a tree *without using tail recursion*. For example, if t is the tree in the figure above, then `sumTree t` evaluates to 65.

```
sumTree :: Tree -> Int
```

Solution:

```
sumTree Leaf = 0
sumTree (Node x left right) = x + sumTree left + sumTree right
```

4.2 (7 points) Define a function sumTreeTR, which returns the sum of all the values in the nodes within a tree *using tail recursion*. For example, if t is the tree in the figure above, then `sumTree t` evaluates to 65.

```
sumTreeTR :: Tree -> Int
```

Solution:

```
sumTreeTR tree = helper tree 0
  where
    helper :: Tree -> Int -> Int
    helper Leaf sum = sum
    helper (Node x left right) sum =
        helper left (helper right (sum + x))
```

4.3 (7 points) Implement the function `inOrder`, which returns the list of nodes from smallest to largest. For example, if t is the tree in the figure above, then `inOrder t` evaluates to `[5,7,9,11,13,20]`. (It may be head or tail recursive.)

```
inOrder :: Tree -> [Int]
```

Solution:

```
inOrder Leaf = []
inOrder (Node x left right) = inOrder left ++ [x] ++ inOrder right
```

4.4 (7 points) Now use `inOrder` to implement `sumTree` and `sumTreeTR` using `foldr` and `foldl`. Your implementation should use the correct fold for head or tail recursion. The implementation of `inOrder` above does not matter, even if incomplete.

Solution:

```
sumTree t = foldr (+) 0 (inOrder t)

sumTreeTR t = foldl (+) 0 (inOrder t)
```

Question 5 (*22 points*)

Haskell has some functions that let you convert `Char` values to `Int` values and vice-versa. Function `ord :: Char -> Int` converts a `Char` to an `Int` corresponding to its position in the alphabet (starting at 0), so (ord 'a') is 0 and (ord 'z') is 25. Function `chr :: Int -> Char` converts a `Int` to the `Char` corresponding to the letter at that position (mod 26) in the alphabet. So (chr 0) is 'a', (chr 25) is 'z', and (chr 26) is 'a'.

Consider the specifications for the higher-order functions `enc` and `dec`.

```
-- 'enc f txt' applies f to each character of 'txt' to encode a string.
enc :: (Char -> Char) -> String -> String
-- 'dec g txt' applies g to each character of 'txt' to decode 'txt'.
dec :: (Char -> Char) -> String -> String
-- REQUIRED: if '(g (f c)) = c' then 'dec g (enc f str) = str'
```

5.1 (5 points) Given the above specifications, what does the following Haskell expression evaluate to?

```
let k c = chr ((ord c) + 13) in
  enc k "abc"
```

ⓐ `"xyz"`
ⓑ `"nmo"`
ⓒ `"abc"`
ⓓ Type Error
ⓔ None of the above

5.2 (5 points) Given the above specifications, what does the following Haskell expression evaluate to?

```
let k c = chr ((ord c) + 2) in
  dec k "rpgai"
```

ⓐ `"trick"`
ⓑ `"treat"`
ⓒ `"candy"`
ⓓ Type Error
ⓔ None of the above

5.3 (5 points) Given the above specifications, what does the following Haskell expression evaluate to?

```
let k  n c  = chr ((ord c) + n)
    k' n c = chr ((ord c) - n)
in
  dec (k' 5) (enc (k 5) "foo")
```

ⓐ `"ktt"`
ⓑ `"ajj"`
ⓒ `"oof"`
ⓓ `"foo"`
ⓔ Type Error

5.4 (7 points) Give an implementation for `enc` and `dec`. You may use any library functions you wish.

Solution:

```
enc = map
dec = map
```

(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]
```

(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)