CSE114A, Spring 2022: Midterm Exam

Instructor: Lindsey Kuper

May 3, 2022

Student name: ______________________________

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

This exam has 11 questions and 120 total points.

Instructions

- Please write directly on the exam.
- **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)

Part 1: Lambda calculus

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

```
(\x y z -> y (x y z)) (\a b -> a b)
```

   a. (5 points) Choose the best answer:
      (a) `EXPR1` is in normal form
      (b) After 1 β-reduction step, `EXPR1` will be in normal form
      (c) After 2 β-reduction steps, `EXPR1` will be in normal form
      (d) After 3 or more β-reduction steps, `EXPR1` will be in normal form
      (e) `EXPR1` does not have a normal form

   b. (5 points) After a *single* β-reduction step on `EXPR1`, what would the resulting expression be? Write your answer in the box below. If no β-reduction step is possible, write "no β-reduction possible".

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

```
(\a b -> a b) (\f x -> f (f x))
```

   a. (5 points) Choose the best answer:
      (a) `EXPR2` is in normal form
      (b) After 1 β-reduction step, `EXPR2` will be in normal form
      (c) After 2 β-reduction steps, `EXPR2` will be in normal form
      (d) After 3 or more β-reduction steps, `EXPR2` will be in normal form
      (e) `EXPR2` does not have a normal form

   b. (5 points) After a *single* β-reduction step on `EXPR2`, what would the resulting expression be? Write your answer in the box below. If no β-reduction step is possible, write "no β-reduction possible".

3. Consider the following lambda calculus expression, which we will name `EXPR3`:

```
\a b -> (\f x y -> f x y) a b (\x y -> z y)
```

a. (5 points) Choose the best answer:
   - (a) No variables occur free in `EXPR3`
   - (b) `a` and `b` occur free in `EXPR3`
   - (c) `a`, `b`, and `z` occur free in `EXPR3`
   - (d) `z` occurs free in `EXPR3`

b. (7 points) Which of the following expressions can be obtained from `EXPR3` with *one or more* β-reductions?
   - (a) `\a b -> a b (\x y -> z y)`
   - (b) `a b (\x y -> z y)`
   - (c) `\a b -> a (b (\x y -> z y))`
   - (d) `\b -> (\f x y -> f x y) b (\x y -> z y)`

c. (3 points) Which of the following is a correct α-renaming of `EXPR3`?
   - (a) `\q r -> (\f x y -> f x y) a b (\x y -> z y)`
   - (b) `\q r -> (\f x y -> f x y) q r (\g h -> z h)`
   - (c) `\a b -> (\f x y -> f x y) a b (\x y -> x y)`
   - (d) `\a b -> (\f g h -> f g h) a b (\g h -> z y)`

d. (10 points) Evaluate

   `EXPR3 (\x -> x) (\y -> y)`

   to normal form with a series of β-reduction steps. Show your work in the box below.

4. (10 points) Consider the following lambda calculus combinators (a *combinator* is just a lambda calculus term with no free variables):

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

Use the above combinators to define a new combinator `SAME` with the following behavior:

- `SAME b1 b2` should evaluate to `TRUE` if *both* `b1` and `b2` evaluate to `TRUE`.
- Likewise, `SAME b1 b2` should evaluate to `TRUE` if *both* `b1` and `b2` evaluate to `FALSE`.
- `SAME b1 b2` should evaluate to `FALSE` if one of `b1` and `b2` evaluates to `TRUE` and the other evaluates to `FALSE`.

You may use any of the above predefined combinators in your definition of `SAME`. You may assume that the arguments to `SAME` are expressions whose value is either `TRUE` or `FALSE`. Write your answer in the box below.

```
let SAME =
```

Part 2: Haskell

5. (5 points) What is the type of the following Haskell expression?

```
map (\s -> "hello " ++ s) ["apple", "orange"]
```

   (a) `String -> String`
   (b) `[String] -> String`
   (c) `String`
   (d) `[String]`
   (e) Type error

6. (5 points) Suppose `subtractThree` is defined as follows:

```
subtractThree :: Int -> Int
subtractThree x = x - 3
```

   What is the type of the following Haskell expression?

```
subtractThree (foldr (+) 0 [1,2,3])
```

   (a) `Int -> Int`
   (b) `[Int] -> Int`
   (c) `Int`
   (d) `[Int]`
   (e) Type error

7. (5 points) What does the following Haskell expression evaluate to?

```
let f = \x -> x + 1
    g = filter (\y -> y > 1)
  in g (map f [0,1,2,3])
```

   (a) `[2,3]`
   (b) `[2,3,4]`
   (c) `[3,4]`
   (d) Type error

8. (5 points) What is the type of the following Haskell expression?

```
let b = 3 < 5 in
  case b of
    True  -> (\s -> "hello, " ++ s)
    False -> (\s -> "goodbye, " ++ s)
```

   (a) `String -> String`

   (b) `Bool -> String`

   (c) `String -> Bool -> String`

   (d) `Bool -> String -> String`

   (e) Type error

9. For this question, you will define a Haskell function `listify` that takes as arguments an `Int` and some element, and returns a list of the specified number of repetitions of that element. For example, `listify 3 5` evaluates to `[5,5,5]` and `listify 2 "hi"` evaluates to `["hi","hi"]`.

   a. (5 points) What is the type signature of `listify`? Write your answer in the box below.

   b. (10 points) Complete the below definition of `listify`. The base case is already filled in for you. Write the remaining part of the definition in the box below. Don't change the existing code, and don't use library functions other than the list constructors and simple arithmetic on `Int`s.

```
listify n x
    | n <= 0 = []
    <YOUR CODE HERE>
```

For the next two questions, consider the following data type:

```
data AExp = Num Int | Plus AExp AExp | Minus AExp AExp
```

10. (20 points) An `AExp` can represent an arithmetic expression. For example:

- 3 is represented by `Num 3`
- $3 + 4$ is represented by `Plus (Num 3) (Num 4)`
- $7 - (3 + 12)$ is represented by
  `Minus (Num 7) (Plus (Num 3) (Num 12))`

In the box below, define a Haskell function `evalAExp` that takes an `AExp` and returns the `Int` value of the arithmetic expression it represents. Here are some sample calls to `evalAExp`:

```
> evalAExp (Num 3)
3
> evalAExp (Plus (Num 3) (Num 4))
7
> evalAExp (Minus (Num 7) (Plus (Num 3) (Num 12)))
-8
```

The type signature of `evalAExp` is provided below for you. Do not use library functions other than simple arithmetic on `Int`s.

```
evalAExp :: AExp -> Int
```

11. (10 points) Instead of evaluating an `AExp`, we might want to simply look at a more readable representation of it. For this question, you will define a Haskell function `showAExp` that takes an `AExp` and returns a nicely formatted `String` of the expression it represents. Here are some sample calls to `showAExp`:

```
> showAExp (Num 3)
"3"
> showAExp (Plus (Num 3) (Num 4))
"(3 + 4)"
> showAExp (Minus (Num 7) (Plus (Num 3) (Num 12)))
"(7 - (3 + 12))"
```

Complete the below definition of `showAExp`. The type signature and base case are already filled in for you. Write the remaining part of the definition in the box below. Don't change the existing code, and don't use library functions other than the append `(++)` function.

The base case of `showAExp` uses the Haskell library function `show` to convert an `Int` to a `String`. (For example, `show 3` evaluates to `"3"`.) Do not use `show` elsewhere in the definition of `showAExp`.

```
showAExp :: AExp -> String
showAExp e = case e of
  Num i -> show i
  <YOUR CODE HERE>
```