CSE 114A, Fall 2021 Midterm

Section	**Points**	**Score**
Part I	58 points	
Part II	80 points	
Total	138 points	

Instructions

- **You have 95 minutes to complete this exam.**
- 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.
- 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.

NAME: ____

CruzID: ____________________________ @ucsc.edu

Part I: Lambda calculus

1. Consider the following lambda calculus expression:

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

   (A) **(5 pts)** Recall that the *redex* of a lambda expression is the is a lambda term of the form `(\x -> e1) e2`. Identify a redex in the above expression and write it below. If there are multiple, you may choose any valid redex. If there are no redexes, write "none."

   (B) **(8 pts)** Does the above expression have a normal form? If so, what is it?

   (a) `apple`

   (b) `\c -> c`

   (c) `TRUE`

   (d) `\x y -> y`

   (e) Does not have a normal form

2. Now consider the following lambda calculus expression:

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

(A) **(5 pts)** Which variables are **bound** in the above expression?

(a) All occurrences of `a` and `b`

(b) All occurrences of `x`, `y` and `z`

(c) All occurrences of `a`, `b`, `x`, `y` and `z`

(d) The first occurrences of `x`, `y` and `z`

(e) All but the first occurrences of `x`, `y` and `z`

(B) **(10 pts)** Which of the following expressions can be obtained from the above expression with one or more beta reductions?

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

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

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

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

(e) `(\y -> y y b) (\y -> y y b) b b b b a`

3. **(20 pts)** Fill in a lambda calculus expression for each blank in the program below to define a function POW where (POW x n) returns x raised to the n^{th} power. For example POW TWO THREE =~> EIGHT. You may use any of the functions defined on the Lambda Calculus Cheat Sheet on the back page. Any other helper functions you must define yourself. You must use recursion for full credit.

```
let POW1 = \f x n -> ITE _____(A)_______
                         _____(B)_______
                         _____(C)_______

let POW = ______(D)_______
```

(A)

(B)

(C)

(D)

4. For each bound occurrence of a variable in the following lambda terms, draw an arrow pointing to its binder. For each free occurrence, draw a circle around the variable.

(A) **(5 pts)** `(\x -> \y -> \z -> x (\y -> z (\z -> y z)) y)`

(B) **(5 pts)** `(\x y z -> (\a b c -> x y z)) (\x y z -> a b c)`

Part II: Haskell

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

```
map (\x y -> x + y) [1, 2, 3, 4, 5]
```

   (a) `Int`

   (b) `[Int]`

   (c) `Int -> [Int]`

   (d) `[Int -> Int]`

   (e) Type error

6. **(5 pts)** What does the following Haskell expression evaluate to?

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

   (a) 0

   (b) -2

   (c) 2

   (d) -6

   (e) Type error

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

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

   (a) 0

   (b) -2

   (c) 2

   (d) -6

   (e) Type error

For the next questions, consider the data type defined below.

```
data Expr = Abs String Expr
          | App Expr Expr
          | Var String
```

8. **(15 pts)** The function `show` converts an `Expr` value to a String in lambda expression syntax. For example: `show (App (Abs "x"(Var "x")) (Var "y"))` returns "`(\x -> x) y`". Fill in the blanks to complete the implementation.

```
show :: Expr -> String

show ______(A)_______ = "(\" ++ x ++ " -> " ++ (show e) ++ ")"

show (App e1 e2)      = __________(B)_________

show ________________(C)______________________
```

(A)

(B)

(C)

9. **(10 pts)** Create an `Expr` value that, when passed to `show`, returns:

```
(\stp -> (\x -> stp (x x)) (\x -> stp (x x)))
```

10. **(30 pts)** The `subst` function implements capture-avoiding substitution for `Expr` values. For example, `subst (Abs a (Var b)) b (Abs x (Var x))` returns `(Abs a (Abs x (Var x)))` since `(\a -> b)[b:=(\x -> x)]` equals `(\a -> (\x -> x))`.

    The helper function `fv :: Expr -> [String]` returns a list of strings containing the name of each free variable in an `Expr` value.

    The guard expression `notElem x (fv e2)` returns `True` if the string `x` is **not** in the list of free variable names of `e2` returned by `(fv e2)`, otherwise it returns `False`.

```
subst :: Expr -> String -> Expr -> Expr
subst (Var x)      y e   = ____________(A)______________
subst (App e1 e2) y e3 = ____________(B)______________
subst (Abs x e1)  y e2
  | x == y               = _________________(C)_______________
  | notElem x (fv e2)    = _________________(D)_______________
```

(A)

(B)

(C)

(D)

11. **(10 pts)** Are the equations for `subst` exhaustive? If they are, write "exhaustive"; if not, write a call to `subst` with arguments that are not matched by any of `subst`'s equations.