CSE 114A, Fall 2021 Final

Section	Points	Score
Part I	40 points	
Part II	40 points	
Part III	100 points	
Total	180 points	

Instructions

- **You have 180 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. **[5pts]** Circle the free variables of this lambda calculus expression. For each occurrence of a bound variable, draw an arrow from the occurrence to its binder.

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

2. **[15pts]** Use β-reductions to evaluate the following lambda terms to a normal form.

2.1. `(\n m -> m n) (\x y -> x (x y)) (\z w -> z (z w))`
Possible answer:

- `(\n.\m.m n) (\x.\y.x (x y)) (\z.\w.z (z w))`
  ```
  _____
  ```
  `(\m.m (\x.\y.x (x y))) (\z.\w.z (z w))`
  ```
  _____
  ```
  `(\z.\w.z (z w)) (\x.\y.x (x y))`
  ```
  _____
  ```
  `\w.(\x.\y.x (x y)) ((\x.\y.x (x y)) w)`
  ```
      _____
  ```
  `\w.\y.(\x.\y.x (x y)) w ((\x.\y.x (x y)) w y)`
  ```
          _____
  ```
  `\w.\y.(\y.w (w y)) ((\x.\y.x (x y)) w y)`
  ```
                    _____
  ```
  `\w.\y.w (w ((\x.\y.x (x y)) w y))`
  ```
                  _____
  ```
  `\w.\y.w (w ((\y.w (w y)) y))`
  ```
                  _____
  ```
  `\w.\y.w (w (w (w y)))`

2.2. (\n -> (\a -> \a b -> a) n (\a b -> b)) ((\x -> x x) (\y -> y y)) (\z -> z)

Possible answer:

- (\n.(\a.\a.\b.a) n (\a.\b.a)) ((\x.x x) (\y.y y)) (\z.z)
  =b> (\a.\a.\b.a) ((\x.x x) (\y.y y)) (\a.\b.a) (\z.z)
  =b> (\a.\b.a) (\a.\b.a) (\z.z)
  =b> (\b.\a.\b.a) (\z.z)
  =b> \a.\b.a

2.2. (\n -> (\a -> \a b -> a) n (\a b -> b)) ((\x -> x x) (\y -> y y)) (\z -> z)

- (\n.(\a.\a.\b.a) n (\a.\b.a)) ((\x.x x) (\y.y y)) (\z.z)

3. **(20 pts)** Fill in a lambda calculus expression for each blank in the program below to define a function RANGE where (RANGE n m) returns a "list" (encoded as nested pairs) containing numbers n through **and including** m, and terminated by FALSE:

(PAIR n (PAIR (n+1) (PAIR ... (PAIR (m-1) (PAIR m FALSE)))))

for any Church-encoded numerals n and m where n <= m. (You can assume RANGE is only called with n <= m.) For example,

(RANGE ONE FIVE) =~> (PAIR ONE (PAIR TWO (PAIR THREE (PAIR FOUR (PAIR FIVE FALSE))))).

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 RANGE1 = \f n m -> ITE _____(3.1)_____
                            _____(3.2)_____
                            _____(3.3)_____

let RANGE = _____(3.4)_____
```

3.1

          EQL n m

3.2

          (**PAIR** m **FALSE**)

3.3

          (**PAIR** n (f (**DECR** n) m))

3.4

          FIX RANGE1

4

Part II: Formalization

Consider the following grammar for a language that supports arithmetic functions on integer pairs (and only pairs) of the form `(n1,n2)` for any integer `n`.

Grammar

```
e ::= (n1, n2) | x | (\x -> e) | (e1 e2) | (e1 + e2)
```

(x is an alphanumeric string representing a variable name)

For example, the function `(\x -> (\y -> (x + y)))` could be applied to the pairs `(1, 2)` and `(2, 4)` as follows:

```
(((\x -> \y -> x + y) (1, 2)) (2, 4)) =~> (3,6)
```

Types

Types are represented by the following grammar:

```
T := (Int, Int) | T1 -> T2
```

Type system

Below is a partial type system for this language.

```
[T-Var] -------------------          [T-Pair] ---------------------------
          [x:T] |- x :: T                       G |- (n1, n2) :: (Int, Int)


          G, x:T1 |- e :: T2                     G |- e1 :: T1 -> T2   G |- e2 :: T1
[T-Abs] ------------------------     [T-App] ------------------------------------
          G |- (\x -> e) :: T1 -> T2             G |- (e1 e2) :: T2
```

4. **[5pts]** The above rules are missing a rule for typing add expressions. Fill in the missing parts of the `T-Add` rule below.

```
            G |- e1 :: _4.1_      G |- e2 :: _4.2_
   [T-Add] ------------------------------------------
                   G |- e1 + e2 :: (Int, Int)
```

4.1

```
┌─────────────────────────────────────────────┐
│                                               │
│                                               │
│                                               │
│                                               │
└─────────────────────────────────────────────┘
```

4.2

```
┌─────────────────────────────────────────────┐
│                                               │
│                                               │
│                                               │
│                                               │
└─────────────────────────────────────────────┘
```

5. **[20pts]** Below is a partial proof that `(\x -> \y -> x + y) (1,2) (3,4)` is well-typed. For each blank, fill in a typing rule or type to complete the proof.

```
[_5.1_] --------------------------------------     [_5.2_]--------------------------------
        G,x:(Int,Int),y:(Int,Int)|- y :: _5.3_    G,x:(Int,Int),y:(Int,Int)|- x :: _5.4_
[_5.5_] -----------------------------------------------------------------------------------
        G,x:(Int,Int),y:(Int,Int) |- (x + y) :: _5.6_
[_5.7_] ----------------------------------------------
        G,[x:(Int,Int)] |- (\y -> x + y) :: _5.8_
[_5.9_] --------------------------------     ------------[_5.10_]
        G |- (\x -> \y -> x + y) :: _5.11_  G |- (1,2) :: _5.12_
[_5.13_] ----------------------------------------------------- [_5.14_]----------------
         G |- (\x -> \y -> x + y) (1,2) :: _5.15_                G |- (3,4) :: _5.16_
[_5.17_] -------------------------------------------------------------------------------------
         G |- (\x -> \y -> x + y) (1,2) (3,4) :: (Int, Int)
```

5.1

5.2

5.3

5.4

5.5

5.6

5.7

5.8

5.9

5.10

5.11

5.12

5.13

5.14

5.15

5.16

5.17

Unification

Fill in the letter of the **one** item that best answers the questions below.

6. **[5pts]** What is the most general unifier of the following types?

$$(a \rightarrow b \rightarrow c) \qquad\qquad (x \rightarrow y)$$

(a) [x / (a -> b), y / c]

(b) [(a -> b) / x, c / y]

(c) [a / x , (b -> c) / y]

(d) [x / a, y / (b -> c)]

(e) None of the above

(f) Cannot unify

7. **[5pts]** What is the most general unifier of the following types?

$$(Int \rightarrow a) \qquad\qquad (a \rightarrow b)$$

(a) [a / Int, b / Int]

(b) [b / Int, a / b]

(c) [a / b, a / Int]

(d) [a / b, b / a, a / Int]

(e) None of the above

(f) Cannot unify

8. **[5pts]** What is the most general unifier of the following types?

$$(\text{Int} \to \text{Int} \to b) \qquad\qquad (a \to b)$$

(a) [a / Int, b / (Int -> b)]

(b) [a / (Int -> Int)]

(c) [a / Int, b / Int, b / (Int -> Int)]

(d) [a / Int, Int / b, b / (Int -> Int)]

(e) None of the above

(f) Cannot unify

Part III: Haskell

9. **[5pts]** Given the following definition of `nats`:

```
nats = 0 : next 1
  where
    next n = n : next (n+1)
```

What does the expression `(head (tail (tail (tail (map (* 2) nats)))))` evaluate to?

ⓐ 3

ⓑ 6

ⓒ 8

ⓓ runtime error

ⓔ infinite loop

ⓕ type error

10. **[5pts]** What does this Haskell expression evaluate to? (See Haskell cheat sheet for definition of foldl.)

```
foldl (flip (:) . reverse) [] [1, 2, 3]
```

ⓐ `[1, 2, 3]`

ⓑ `[3, 2, 1]`

ⓒ `[2, 1, 3]`

ⓓ `[3, 1, 2]`

ⓔ Type error

ⓕ None of the above

For the following questions, carefully consider the Haskell definitions below

```haskell
data Gift = Clothes Size Color
          | Toy String
          | Cash Int
          | Cookies Flavor
  deriving (Show, Eq)

data Size = S | M | L
  deriving (Show, Eq)
data Color = Red | Blue | Green
  deriving (Show, Eq)
data Flavor = Chocolate | Oatmeal | Sugar
  deriving (Show, Eq)

data WrappedGift = Card String String Gift
                 | Bag  String String Gift
                 | Box  String String Gift

unwrap :: String -> [WrappedGift] -> [Gift]
unwrap _ [] = []
unwrap name (w:ws) =
  case w of
    Card to from g | to == name -> g:(unwrap name ws)
    Bag  to from g | to == name -> g:(unwrap name ws)
    Box  to from g | to == name -> g:(unwrap name ws)
    _ -> unwrap name ws

keep :: [Gift] -> [Gift]
keep = filter isKeeper
  where
    isKeeper (Clothes _ _) = True
    isKeeper (Clothes S Green) = False
    isKeeper (Cookies _) = False
    isKeeper (Toy name) = isPrefixOf "Lego" name
    isKeeper _ = True
```

11. **[25pts]**

```
presents :: [WrappedGift]
presents = [ Box "Alice" "Owen" (Clothes M Blue)
           , Box "Owen" "Alice" (Clothes M Blue)
           , Bag "Owen" "Bob" (Cookies Sugar)
           , Bag "Bob" "Owen" (Cookies Chocolate)
           , Card "Owen" "Carol" (Cash 25)
           , Bag "Owen" "Carol" (Clothes L Red)
           , Box "Owen" "Dave" (Toy "Lego Mindstorms Robot")
           , Box "Dave" "Owen" (Toy "Lego Hogwarts Castle")
           , Box "Owen" "Eve" (Toy "Largo Minestorms Robot")
           , Box "Owen" "Eve" (Clothes S Green )
           , Card "Eve" "Owen" (Cash 0)
           ]
```

Consider the above list called presents. What does the following program print?

```
main :: IO ()
main = putStrLn ( show (( keep . (unwrap "Owen") ) presents ))
```

```
[Clothes M Blue,Cash 25,Clothes L Red,Toy "Lego Mindstorms Robot",Clothes S Green]
```

12. **[15pts]** Write an instance for the typeclass `Show` for `WrappedGift` that returns a string with the kind of container, the recipient, and the sender, **but not the contents of the present** (of course!). For example, `show (Card "Owen" "Carol" (Cash 25))` should return something like `"[Card -- To: Owen, From: Carol]"`.

```
instance Show WrappedGift where

  show (Card to from _) = "[Card] To:" ++ to ++ " From:" ++ from
  show (Bag  to from _) = "[Bag] To:" ++ to ++ " From:" ++ from
  show (Box  to from _) = "[Box] To:" ++ to ++ " From:" ++ from
```

13. **[20pts]** Implement a function `wrap` that takes a `String` for the receiver , a `String` for the sender, and a `Gift` and returns a `WrappedGift`. Gifts should be wrapped according to what kind of gift they are. Clothes and toys go in boxes, cookies go in bags, and cash goes in cards. For example,

`wrap "Alice" "Bob" [Clothes L Red, Cash 20, Toy "train", Cookies Sugar]`

should return the list

```
[ Box  "Alice" "Bob" (Clothes L Red)
, Card "Alice" "Bob" (Cash 20)
, Box  "Alice" "Bob" (Toy "train")
, Bag  "Alice" "Bob" (Cookies Sugar)
]

 wrap :: String -> String -> Gift -> Wrapped

  wrap to from (Cookies f)   = Bag  to from (Cookies f)
  wrap to from (Clothes sz c) = Box  to from (Clothes sz c)
  wrap to from (Cash d)       = Card to from (Cash d)
  wrap to from (Toy n)        = Box  to from (Toy n)
```

14. **[30pts]** Now write a function `regift` that unwraps all unwanted (i.e., according to the function `keep`) gifts for a recipient and re-wraps them for a new recipient. For example,

```
regift "Bob" "Alice" ([ Bag "Bob" "Owen" (Cookies Chocolate)
                       , Box "Bob" "Eve"  (Toy "Lego X-Wing")
                       , Box "Owen" "Bob" (Toy "Pet Rock")
                       ])
```

should return `[Bag "Alice" "Bob" (Cookies Chocolate)]` since Bob received `(Cookies Chocolate)` from Owen, but they were not part of the list returned by `keep`. You may find the functions `elem` and/or `notElem` useful. See the Haskell Cheat sheet for details.

```
regift :: String -> String -> [WrappedGift] -> [WrappedGift]

regift oldTo newTo wrapped = regift_helper gifts
 where
   gifts = unwrap oldTo wrapped
   keepers = keep gifts

   regift_helper [] = []
   regift_helper (g:gs) | elem g keepers = regift_helper gs
                        | otherwise      = wrap g : regift_helper gs
```

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

-- Lists -------------------------------
let EMPTY = \xs -> xs (\x y z -> FALSE) TRUE
let CONS  = PAIR
let NIL   = FALSE
let HEAD  = FST
let TAIL  = SND

-- 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
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 pred []     = []
filter pred (x:xs)
  | pred x          = x : filter pred xs
  | otherwise       = filter pred 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]

-- Does the element occur in the list?
elem     :: (Eq a) => a -> [a] -> Bool
-- Negation of elem: does the element *not* occur in the list?
notElem :: (Eq a) => a -> [a] -> Bool

-- 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]
```