CSE114A, Spring 2022: Final Exam

Instructor: Lindsey Kuper

June 6, 2022

Student name: _____

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

This exam has 21 questions and 200 total points.

Instructions

- Please write directly on the exam.
- **You have 180 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 additional 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!

(Blank page for use as scratch paper)

Part 1: Lambda calculus (50 points)

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

```
\f x -> f ((\g y -> g y) f x)
```

   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".

Solution:

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

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

```
(\y -> y y) (\z -> z z)
```

   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".

Solution:

```
(Note: Any solution that is alpha-equivalent to
this is also acceptable)
(\z -> z z) (\z -> z z)
```

For the next two questions, the following lambda calculus definitions will be useful:

```
-- Church numerals
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)))

-- Booleans
let TRUE  = \x y -> x
let FALSE = \x y -> y
let ITE   = \b x y -> b x y

-- Pairs
let PAIR  = \x y b -> ITE b x y
let FST   = \p     -> p TRUE
let SND   = \p     -> p FALSE

-- Arithmetic
let SUC   = \n f x -> f (n f x)
let ADD   = \n m -> n SUC m
let MUL   = \n m -> n (ADD m) ZERO
let SKIP1 = \f p -> PAIR TRUE (ITE (FST p)
                                   (f (SND p))
                                   (SND p))
let DEC   = \n -> SND (n (SKIP1 SUC) (PAIR FALSE ZERO))
let SUB   = \n m -> m DEC n
let ISZ   = \n -> n (\z -> FALSE) TRUE

-- The Y combinator
let FIX   = \s -> (\x -> s (x x)) (\x -> s (x x))
```

3. (5 points) What does the following lambda calculus term evaluate to?

```
SUC (SUC (\f x -> f ((\g y -> g y) f x)))
```

(a) `FALSE`

(b) `TWO`

(c) `THREE`

(d) `FOUR`

(e) None of the above

4. (5 points) What does the following lambda calculus term evaluate to?

```
FST (SND ((\x y b -> b x y) INC (PAIR FALSE TWO)))
```

(a) `FALSE`

(b) `PAIR FALSE TWO`

(c) `THREE`

(d) `INC`

(e) None of the above

5. The factorial of a positive integer n is the product of all positive integers less than or equal to n. For this question, you will fill in the blanks in the program below to define a function `FACT` where `FACT n` returns the factorial of `n`. (You can assume that `FACT` is only called with positive integers.)

   You may use any of the functions defined on the previous page. Any other helper functions you must define yourself. You must use recursion for full credit.

```
let FACT1 = \f n -> ITE ______(3(a))________
                        ______(3(b))________
                        ______(3(c))________

let FACT = _______(3(d))________
```

   a. (5 points) 3(a):

   Solution:
   ```
   (ISZ n)
   ```

   b. (5 points) 3(b):

   Solution:
   ```
   ONE
   ```

   c. (5 points) 3(c):

   Solution:
   ```
   (MUL n (f (DEC n)))
   ```

   d. (5 points) 3(d):

   Solution:
   ```
   FIX FACT1
   ```

Part 2: Haskell (55 points)

For this section, you may wish to use the Haskell Reference on the last page.

6. (5 points) Which of the following Haskell expressions has the type `[(String, Bool)]`?

   (a) `[("apple", False)]`

   (b) `("orange", True) : []`

   (c) `[("apple", False)] : [("orange", True)] : []`

   (d) (a) and (b)

   (e) (a), (b), and (c)

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

```
let f = foldl (.) (\x -> x) in
  let g = f [(\x -> x - 1), (\x -> x - 2)] in
    g 2
```

   (a) `[1, 0]`

   (b) `-1`

   (c) `0`

   (d) `[-1]`

   (e) Type error

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

```
let l = filter (\n -> n <= 3) [1, 2, 3, 4] in
  case l of
    []     -> (\y -> True)
    (x:xs) -> (\y -> False)
```

   (a) `a -> Bool`

   (b) `[a] -> Bool`

   (c) `[a] -> a -> Bool`

   (d) `[a] -> b -> Bool`

   (e) Type error

9. (5 points) What does the following Haskell expression evaluate to?

```
map ((\x -> "x + 5") . show) [1, 2, 3]
```

   (a) `[6, 7, 8]`

   (b) `["6", "7", "8"]`

   (c) `["x + 5","x + 5","x + 5"]`

   (d) `["15","25","35"]`

   (e) Type error

10. (10 points) Consider the function `naiveLength`, which returns the length of a list:

```
naiveLength :: [a] -> Int
naiveLength []     = 0
naiveLength (x:xs) = 1 + naiveLength xs
```

For this question, you will define a function `lengthTR` that is a **tail-recursive** version of `naiveLength`. You may use any of: a separately defined helper function, a `where` clause in the definition of `lengthTR`, or `foldl`. Do not use any other library functions other than those used in the above definition of `naiveLength`.

```
lengthTR :: [a] -> Int
```

Solution:

```
lengthTR l = helper l 0
  where helper []     n = n
        helper (x:xs) n = helper xs (n + 1)

(or using a separate helper, or foldl)
```

The next two questions will use the following data type definitions for lambda calculus expressions:

```
type Id = String
data LExpr = Var Id | Lam Id LExpr | App LExpr LExpr
```

11. (10 points) For this question, you will define a Haskell function `occurs` that takes an `Id` and a `LExpr` as arguments and returns a `Bool`. `occurs x e` returns `True` if `x` occurs in `e`, and `False` otherwise. **For the purposes of this question, a formal parameter does *not* count as a variable occurrence.** Here are some sample calls to `occurs`:

```
> occurs "x" (Var "x")
True
> occurs "x" (Lam "x" (Var "y"))
False
> occurs "y" (App (Var "f") (Var "x"))
False
> occurs "g" (Lam "f" (Lam "y" (App (Var "g") (Var "y"))))
True
```

Complete the below definition of `occurs`. The type signature and one of the cases are already filled in for you. You may use any of the Haskell library functions described in the Haskell Reference on the last page of the exam, but no other library functions.

```
occurs :: Id -> LExpr -> Bool
occurs id1 (Var id2) = id1 == id2
```

Solution:

```
occurs id1 (Lam id2 expr) = occurs id1 expr
occurs id1 (App e1 e2)    = occurs id1 e1 || occurs id1 e2
```

12. (15 points) For this question, you will define a Haskell function `freeVars` that takes a `LExpr` as defined above and returns a list of its free variables (in any order). Here are some sample calls to `freeVars`:

```
> freeVars (Var "x")
["x"]
> freeVars (Lam "y" (Var "y"))
[]
> freeVars (App (Var "f") (Var "x"))
["f","x"]
> freeVars (Lam "f" (Lam "y" (App (Var "g") (Var "y"))))
["g"]
> freeVars (App (Var "x") (Lam "x" (Var "x")))
["x"]
```

For full credit, a variable that occurs free more than once in an expression should only appear once in the list returned by `freeVars`. Thus `freeVars (App (Var "y") (Var "y"))` should evaluate to `["y"]`.

Complete the below definition of `freeVars`. The type signature is already filled in for you. You may use any of the Haskell library functions described in the Haskell Reference on the last page of the exam, but no other library functions.

```
freeVars :: LExpr -> [Id]
```

Solution:

```
freeVars (Var id)       = [id]
freeVars (Lam id expr) = freeVars expr \\ [id]
freeVars (App e1 e2)    = nub (freeVars e1 ++ freeVars e2)
```

Part 3: Scope, environments, and interpreters (63 points)

13. Consider the following Nano program:

```
let a = 1 in
  let b = 2 in
    let f = \x y -> x + y + a + b in
      let a = 3 in
        let b = 4 in
          f a b
```

a. (5 points) Under **static scope**, what would the above program evaluate to?

(a) 6

(b) 10

(c) 14

(d) error: multiple declarations of a variable

(e) error: unbound variable

b. (5 points) Under **dynamic scope**, what would the above program evaluate to?

(a) 6

(b) 10

(c) 14

(d) error: multiple declarations of a variable

(e) error: unbound variable

14. Consider the following Nano program:

```
let a = 1 in
  let b = 2 in
    let f = \x -> x + a + b + c in
      let a = 2 in
        let c = 1 in
          f a
```

a. (5 points) Under **static scope**, what would the above program evaluate to?

(a) 5

(b) 6

(c) 7

(d) error: multiple declarations of a variable

(e) error: unbound variable

b. (5 points) Under **dynamic scope**, what would the above program evaluate to?

(a) 5

(b) 6

(c) 7

(d) error: multiple declarations of a variable

(e) error: unbound variable

(Blank page for use as scratch paper)

For the next two questions, carefully consider the following Haskell definitions:

```
type Id = String

data Expr = EInt Int | EVar Id | ELam Id Expr
  | EApp Expr Expr | ELet Id Expr Expr

data Val a = VClos a Id Expr | VInt Int

class Env a where
  emptyEnv :: a
  extendEnv :: Id -> Val a -> a -> a
  lookupInEnv :: Id -> a -> Val a

data ListEnv = ListEnv [(Id, Val ListEnv)]

instance Env ListEnv where
  emptyEnv :: ListEnv
  emptyEnv = ListEnv []

  extendEnv :: Id -> Val ListEnv -> ListEnv -> ListEnv
  extendEnv id val (ListEnv env) =
    ListEnv ((id, val) : env)

  lookupInEnv :: Id -> ListEnv -> Val ListEnv
  lookupInEnv id (ListEnv []) =
    error ("unbound variable: " ++ id)
  lookupInEnv id (ListEnv ((x,v):xs))
    | id == x   = v
    | otherwise = lookupInEnv id (ListEnv xs)
```

15. The code above defines (among other things) an `Env` *type class* that defines the interface that an environment should implement: an `emptyEnv` value, and `extendEnv` and `lookupInEnv` operations. One way to implement this interface is with `ListEnv`, which represents an environment as a list of pairs that map identifiers to values. The following is an example of an environment represented with `ListEnv` that maps the variable `a` to the value `3` and maps the variable `b` to a closure.

```
ListEnv [("a",VInt 3),("b",VClos emptyEnv "x" (EVar "x"))]
```

An alternative to `ListEnv` is `FunEnv`, which represents an environment as a *function* that takes an identifier as an argument and returns a value:

```
data FunEnv = FunEnv (Id -> Val FunEnv)
```

As a `FunEnv`, the above example could be (conceptually) thought of as:

```
FunEnv (\x -> if x == "a"
              then VInt 3
              else if x == "b"
                   then VClos emptyEnv "x" (EVar "x")
                   else error ("unbound variable: " ++ x))
```

For this question, you will fill in the blanks in the below code to define a new instance of the `Env` type class for the `FunEnv` type. The definition of `emptyEnv` is already provided for you, and you'll need to fill in two blanks to complete the definitions of `extendEnv` and `lookupInEnv`. (Hint: these are one-liners.)

```
instance Env FunEnv where
  emptyEnv :: FunEnv
  emptyEnv =
    FunEnv (\x -> error ("unbound variable: " ++ x))

  extendEnv :: Id -> Val FunEnv -> FunEnv -> FunEnv
  extendEnv id val (FunEnv f) = _____(15(a))_______

  lookupInEnv :: Id -> FunEnv -> Val FunEnv
  lookupInEnv id (FunEnv f) = _____(15(b))_______
```

a. (13 points) 15(a):

Solution:

```
FunEnv (\x -> if x == id then val else f x)
```

b. (10 points) 15(b):

Solution:

```
f id
```

16. The following is part of an implementation of an interpreter for the little language of `Exprs`.

```
eval :: Env a => a -> Expr -> Val a
eval _   (EInt n)      = VInt n
eval env (EVar id)     = _____(16(a))_______
eval env (ELam id e)   = VClos env id e
eval env (EApp e1 e2) = case eval env e1 of
  (VClos cEnv cId cExpr) -> _____(16(b))_______
  _ -> error "type error!"
eval env (ELet id e1 e2) = eval env (EApp (ELam id e2) e1)
```

Instead of a concrete type for the `env` argument, `eval`'s type signature type variable `a` that's constrained by the `Env a` type class constraint. So, `eval` ought to work with any environment representation that correctly implements the `Env` interface. (In other words, **`eval` should call `lookupInEnv` and `extendEnv`** rather than doing something that would depend on the environment representation.)

a. (5 points) 16(a):

Solution:

```
lookupInEnv id env
```

b. (15 points) 16(b):

Solution:

```
eval env' cExpr
    where env' = extendEnv cId v2 cEnv
          v2   = eval env e2
```

Part 4: Types and type inference (32 points)

17. (5 points) Which of the following is *a unifier* for the types `a -> b` and `Int -> c`?

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

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

    (c) `[a / Int, c / Bool]`

    (d) (a) and (b)

    (e) Cannot unify

18. (5 points) Which of the following is *a unifier* for the types `Int -> Int -> a` and `b -> c`?

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

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

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

    (d) (a) and (b)

    (e) Cannot unify

19. (5 points) Which of the following is *the most general unifier* for the types `Int -> a` and `b -> String`?

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

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

    (c) `[a / b]`

    (d) (a) and (b)

    (e) Cannot unify

20. (5 points) Which of the following is *the most general unifier* for the types `a` and `b -> Bool -> a`?

    (a) `[a / Bool -> a, a / b]`

    (b) `[a / Bool -> a, b / a]`

    (c) `[a / b -> Bool -> a]`

    (d) (a) and (b)

    (e) Cannot unify

21. Below are some of the typing rules for the Nano language:

$$\text{[T-Lam]}\ \frac{G, (x, T1) \vdash e :: T2}{G \vdash (\backslash x \to e) :: T1 \to T2}$$

$$\text{[T-App]}\ \frac{G \vdash e1 :: T1 \to T2 \qquad G \vdash e2 :: T1}{G \vdash (e1\ e2) :: T2}$$

$$\text{[T-Int]}\ \frac{}{G \vdash n :: Int} \qquad \text{[T-Var]}\ \frac{(x, T) \text{ in } G}{G \vdash x :: T}$$

Below is a partial typing derivation that shows that the Nano expression `(\x -> x) 3` has type `Int`. For each blank, fill in a type or the name of a typing rule to complete the typing derivation.

$$\dfrac{\dfrac{\dfrac{}{[\_21(a)\_]\ (x, Int) \text{ in } [(x, Int)]}\quad \dfrac{}{[(x, Int)] \vdash x :: \_21(b)\_}\ [\_21(c)\_]}{[] \vdash \backslash x \to x :: \_21(e)\_} \qquad \dfrac{}{[\_21(d)\_]\ [] \vdash 3 :: \_21(f)\_}}{[\_21(g)\_]\ [] \vdash (\backslash x \to x)\ 3 :: Int}$$

a. (2 points) 21(a):

Solution:
T-Var

b. (2 points) 21(b):

Solution:
Int

c. (2 points) 21(c):

Solution:
`T-Lam`

d. (2 points) 21(d):

Solution:
`T-Int`

e. (2 points) 21(e):

Solution:
`Int -> Int`

f. (2 points) 21(g):

Solution:
`T-App`

Haskell Reference

- `(.) :: (b -> c) -> (a -> b) -> a -> c`
  `(.) f g = \x -> f (g x)`

  Function composition.

- `(||) :: Bool -> Bool -> Bool`

  Boolean "or".

- `(++) :: [a] -> [a] -> [a]`

  Append two lists, e.g.,

```
> [1,2,3] ++ [4,5]
[1,2,3,4,5]
> "apple" ++ "orange"
"appleorange"
```

- `nub :: [a] -> [a]`

  Remove duplicate elements from a list, e.g.,

```
> nub [1,2,3,4,3,2,1,2,4,3,5]
[1,2,3,4,5]
```

- `(\\) :: [a] -> [a] -> [a]`

  Compute the difference of two lists. In the result of `xs \\ ys`, the first occurrence of each element of `ys` in turn (if any) has been removed from `xs`. Thus `(xs ++ ys) \\ xs == ys`, e.g.,

```
> ["a","b","c"] \\ ["a"]
["b","c"]
> ["a","b","c","a"] \\ ["a","c"]
["b","a"]
```

- `show :: a -> String`

  Convert a value to a `String`, e.g.,

```
> show 1
"1"
```