CSE114A, Spring 2024: Final Exam

Instructor: Lindsey Kuper

June 10, 2024

Student name: ______________________________

CruzID: ______________________________@ucsc.edu

This exam has 17 questions and 100 total points.

Instructions

- Please write directly on the exam.
- For short answer questions, please write your answer in the provided boxes. You can use space outside of the boxes as scratch space, but we won't see or grade it.
- For multiple choice questions, please circle the correct choice.
- **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 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.
- **We will give partial credit for partially correct answers when it makes sense to do so.** A partially correct answer is better than leaving an answer blank.

Good luck!

This page is for your use as scratch space. Anything you write here will be ungraded.

Part 1: Lambda Calculus

1. (4 points) A lambda calculus expression is in *normal form* if it cannot be further reduced. Evaluate the following lambda calculus expression to normal form using a series of β-reduction steps (and only β-reduction steps – you shouldn't need anything else). Start each line with `=b>`, as if you were using Elsa, and do just one β-reduction step per line.

   Note: There may be multiple correct ways to reduce the expression. A correct solution is any solution that Elsa will accept as correct.

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

   Solution:

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

2. We've seen how the `PAIR` lambda calculus function takes two arguments and creates a data structure with two elements that can be accessed via the `FST` and `SND` accessor functions:

```
let PAIR   = \x y -> (\b -> ITE b x y)
let FST    = \p -> p TRUE
let SND    = \p -> p FALSE
```

Suppose we want to create a data structure with *three* elements. Here's one way to define such a data structure, which we'll call a *triple*, using `PAIR`:

```
let TRIPLE = \x y z -> PAIR x (PAIR y z)
```

So, `TRIPLE x y z` is a `PAIR` where the first element is `x` and the second element is another `PAIR` containing `y` and `z`.

We can write accessor functions `TFST`, `TSND`, and `TTHD` for accessing the first, second, and third elements of a triple, respectively. For example, in Elsa:

```
TFST (TRIPLE ONE TWO THREE) =~> ONE
TSND (TRIPLE ONE TWO THREE) =~> TWO
TTHD (TRIPLE THREE TWO ONE) =~> ONE
TTHD (TFST (TRIPLE (TRIPLE FALSE FALSE TRUE) FALSE FALSE)) =~> TRUE
```

`TFST` is simple; its implementation is just like `FST`:

```
let TFST = \t -> t TRUE
```

Now it's your turn to implement `TSND` and `TTHD`. Feel free to use anything from the lambda calculus reference at the end of the exam.

a. (3 points) `let TSND =`

Solution: There are multiple correct answers.

`\t -> (t FALSE) TRUE`

or

`\t -> FST (SND t)`

both work.

b. (3 points) `let TTHD =`

Solution: There are multiple correct answers.

`\t -> (t FALSE) FALSE`

or

`\t -> SND (SND t)`

both work.

3. 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 in the lambda calculus reference at the end of the exam. Any other helper functions you must define yourself. You must use recursion for full credit.

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

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

   a. (3 points) 3(a):

      Solution:
      `(ISZ n)`

   b. (3 points) 3(b):

      Solution:
      `ONE`

   c. (4 points) 3(c):

      Solution:
      `(MUL n (rec (DECR n)))`

   d. (3 points) 3(d):

      Solution:
      `Y FACT1`

Part 2: Haskell

For this section, you may wish to use the Haskell reference at the end of the exam.

4. (4 points) Consider the following Haskell expression:

```
let f = \x -> x in
  let y = f "sylveon" in
    let z = f True in
      [y, z]
```

Which of the following statements is true?

(a) The expression is ill-typed in Haskell, because the function `f` is being applied to arguments of two different types.

(b) The expression is ill-typed in Haskell, since `y` has type `(a -> a) -> String` and `z` has type `(a -> a) -> Bool`, so they can't be in a list together.

(c) The expression is ill-typed in Haskell, since `y` has type `String` and `z` has type `Bool`, so they can't be in a list together.

(d) The expression is well-typed in Haskell and has the polymorphic type `[a]`, because `f` is polymorphic and can be applied to arguments of arbitrary type.

5. (3 points) What is the **type** of the following Haskell expression?

```
\y -> (y : map (\x -> True) "charmander")
```

(a) Type error

(b) `Bool -> [Bool]`

(c) `a -> [Bool]`

(d) `Bool -> String`

(e) `a -> [String]`

6. (3 points) What is the **type** of the following Haskell expression?

```
foldr (\x y -> x ++ y) "" ["xatu", "mew", "pineco"]
```

(a) Type error

(b) `String`

(c) `[String]`

(d) `String -> String`

(e) `[String] -> String`

7. (3 points) What is the **type** of the following Haskell expression?

```
\s t -> (s, t, s == t)
```

   (a) Type error
   (b) `a -> a -> (a, a, Bool)`
   (c) `a -> b -> (a, b, Bool)`
   (d) `Eq a => a -> a -> (a, a, Bool)`
   (e) `(Eq a, Eq b) => a -> b -> (a, b, Bool)`

8. (3 points) Consider the following Haskell expression:

```
map (\x -> True)
```

Which of the following statements is true about this expression?

   (a) The expression is ill-typed in Haskell, because `map` takes two arguments, but here it's only being applied to one argument.
   (b) The expression is ill-typed in Haskell, because `map` takes a function of type `a -> b` as its first argument, but the function `\x -> True` has type `a -> Bool`.
   (c) It evaluates to a function that takes a list of elements of type `a` and returns a `Bool`.
   (d) It evaluates to a function that takes a list of elements of type `a` and returns a **list** of `Bool`s.

9. (4 points) Consider the following Haskell expression:

```
map (==) ["pikachu", "togepi", ["fletchling", "charizard"]]
```

Which of the following statements is true about this expression?

   (a) The expression is ill-typed in Haskell, because a list cannot contain both elements of type `String` and elements of type `[String]`.
   (b) The expression is ill-typed in Haskell, because `map` takes a one-argument function as its first argument, but `(==)` takes two arguments.
   (c) The expression has type `[Eq a => a -> Bool]`. It evaluates to a list of functions that all take arguments comparable with `(==)` and return values of `Bool` type.
   (d) The expression has type `[Bool]`. It evaluates to a list of `False` values.
   (e) (a) and (b)

Part 3: Abstract Syntax Trees, Interpreters, Environments, and Scope

For the questions in this section, we will use the following `Expr` data type. It defines the grammar of abstract syntax trees for a language with: Booleans, integers, variables, a couple binary operators, lambda (function definition) expressions, application (function call) expressions, and `let`-expressions.

```
data Expr = EBool Bool | EInt Int | EVar Id | EBin BinOp Expr Expr
          | ELam Id Expr | EApp Expr Expr | ELet Id Expr Expr

data BinOp = And | Plus

type Id = String
```

For example, we would represent the expression

```
let f = \x -> True && False in f (3 + 4)
```

with the `Expr`

```
ELet "f" (ELam "x" (EBin And (EBool True) (EBool False)))
         (EApp (EVar "f") (EBin Plus (EInt 3) (EInt 4)))
```

10. Be the parser! Translate the following expressions into their corresponding `Expr`s.

   a. (4 points)

```
let n = True in
  let m = \x -> n && x in
    m True
```

Solution:

```
ELet "n" (EBool True)
         (ELet "m" (ELam "x" (EBin And (EVar "n") (EVar "x")))
                   (EApp (EVar "m") (EBool True)))
```

   b. (4 points)

```
(\x -> (\y -> x + y) 3) 4
```

Solution:

```
(EApp (ELam "x" (EApp (ELam "y" (EBin Plus (EVar "x") (EVar "y")))
                                (EInt 3)))
      (EInt 4)
```

This page is for your use as scratch space. Anything you write here will be ungraded.

Next, we will be writing an interpreter for `Expr`s, but first we need to set up some machinery. We'll be making use of the following Haskell definitions:

```
data Val a = VClos a Id Expr | VBool Bool | VInt Int
  deriving Show

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

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

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)
```

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 the environment interface is with `ListEnv`, which represents an environment as a list of pairs that map identifiers to values. For example, here is 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"))]
```

The interpreter you write ought to work with any environment representation that correctly implements the `Env` interface. In other words, **the `eval` function you write for the next question should call `lookupInEnv` and `extendEnv`**, rather than doing something that would depend on the environment representation.

11. Fill in the blanks in the following definitions to implement an interpreter for our language. Note: Your interpreter does *not* need to support recursive functions.

```
eval :: Env a => a -> Expr -> Val a
eval _   (EBool b)       = VBool b
eval _   (EInt n)        = ____(11(a))____
eval env (EVar id)       = ____(11(b))____
eval env (EBin op e1 e2) = binHelper op (eval env e1) (eval env e2)
eval env (ELam id e)     = VClos env id e
eval env (EApp e1 e2)    = appHelper (eval env e1) (eval env e2)
eval env (ELet id e1 e2) = eval extendedEnv e2
  where extendedEnv = extendEnv id (____(11(c))____) env

appHelper :: Env a => Val a -> Val a -> Val a
appHelper (VClos cEnv id e) argVal = eval (____(11(d))____) e
appHelper _                 _      = error "type error!"

binHelper :: Env a => BinOp -> Val a -> Val a -> Val a
binHelper And  (VBool b1) (VBool b2) = VBool (b1 && b2)
binHelper Plus (VInt n1)  (VInt n2)  = ____(11(e))____
binHelper _    _          _          = error "type error!"
```

a. (4 points) 11(a):

Solution:

```
VInt n
```

b. (4 points) 11(b):

Solution:

```
lookupInEnv id env
```

c. (4 points) 11(c):

Solution:

```
eval env e1
```

d. (4 points) 11(d):

Solution:

```
extendEnv id argVal cEnv
```

e. (4 points) 11(e):

Solution:

```
VInt (n1 + n2)
```

Part 4: Unification and Type Inference

12. (3 points) Which of the following substitutions is *a unifier* for the types `Int -> a` and `b -> Int`?
    - (a) `[(a, Int), (b, Int)]`
    - (b) `[(a, b)]`
    - (c) `[(a, Int), (b, Int), (c, Int)]`
    - (d) (a) and (b)
    - (e) (a) and (c)
    - (f) Cannot unify

13. (3 points) Which of the following substitutions is *a unifier* for the types `(Int -> Int) -> a` and `b -> c`?
    - (a) `[(c, a), (b, Int -> Int)]`
    - (b) `[(Int -> Int, b), (c, a)]`
    - (c) `[(a, Int), (b, Int), (c, Int -> Int)]`
    - (d) (a) and (b)
    - (e) (a), (b), and (c)
    - (f) Cannot unify

14. (3 points) Which of the following substitutions is *a unifier* for the types `a` and `b -> Bool -> a`?
    - (a) `[(b -> Bool -> a, a)]`
    - (b) `[(b, c), (a, c -> Bool -> a)]`
    - (c) `[(a, b -> Bool -> a)]`
    - (d) (a) and (b)
    - (e) (a), (b), and (c)
    - (f) Cannot unify

15. (3 points) Which of the following substitutions is *a unifier* for the types `a -> c` and `b -> Bool -> a`?
    - (a) `[(a, b), (c, Bool -> b)]`
    - (b) `[(b, a), (c, Bool -> a)]`
    - (c) `[(a, Bool), (c, Bool -> a)]`
    - (d) (a) and (b)
    - (e) (a), (b), and (c)
    - (f) Cannot unify

16. In section 3 of this exam, we wrote an interpreter for a small language, but under some circumstances this interpreter will throw a type error at run time. For example, here's what would happen if you tried to evaluate the expression `False + 3`:

```
> eval (emptyEnv::ListEnv) (EBin Plus (EBool False) (EInt 3))
*** Exception: type error!
```

To catch these kinds of errors prior to run time, we can define a *type system* for our language and then implement a *type checker*.

Below are some of the typing rules we'd like to use, with a couple of blanks left for you to fill in. Fill in the blanks to complete the typing rules.

```
            G,(x,T1) |- e :: T2                   (x,T) in G
[T-Lam] --------------------------     [T-Var] -----------
        G |- (\x -> e) :: T1 -> T2             G |- x :: T


        G |- e1 :: Int       G |- e2 :: Int
[T-Add] -----------------------------------
          G |- e1 + e2 :: ____(16(a))____


        G |- e1 :: T1 -> T2       G |- e2 :: T1
[T-App] ---------------------------------------
            G |- (e1 e2) :: ____(16(b))____

        G |- e1 :: T1    G,(x,T1) |- e2 :: T2
[T-Let] ----------------------------------------    [T-Int] -------------
        G |- let x = e1 in e2 :: ____(16(c))____             G |- n :: Int
```

a. (2 points) 16(a):

Solution:
`Int`

b. (2 points) 16(b):

Solution:
`T2`

c. (2 points) 16(c):

Solution:
`T2`

17. Finally, let's use our typing rules to make sure that an expression in our little language is well-typed. The expression we'll consider is `let f = \x -> x + 1 in f 2`.

For each blank below, fill in a type or the name of a typing rule to complete the typing derivation.

We are using the following abbreviations for type environments:

```
G1 = [(x,Int)]
G2 = [(f,Int -> Int)]
```

```
     (x,Int) in G1
[17a]------------- [17b]-----------
     G1 |- x::Int       G1 |- 1::Int      (f,Int->Int) in G2
[17c]------------------------------- [17d]------------------  [17e]--------------
              G1 |- x+1 :: [17f]          G2 |- f :: Int->Int      G2 |- 2 :: Int
       [17g]----------------------   [17h]---------------------------------------
            [] |- \x -> x+1 :: [17i]                 G2 |- f 2 :: [17j]
       [17k]-----------------------------------------------------------
                  [] |- let f = \x -> x + 1 in f 2 :: Int
```

a. (1 point) 17(a):

Solution:
`T-Var`

b. (1 point) 17(b):

Solution:
`T-Int`

c. (1 point) 17(c):

Solution:
`T-Add`

d. (1 point) 17(d):

Solution:
`T-Var`

e. (1 point) 17(e):

Solution:
`T-Int`

f. (1 point) 17(f):

Solution:
`Int`

g. (1 point) 17(g):

Solution:
`T-Lam`

h. (1 point) 17(h):

Solution:
`T-App`

i. (1 point) 17(i):

Solution:
`Int -> Int`

j. (1 point) 17(j):

Solution:
`Int`

k. (1 point) 17(k):

Solution:
`T-Let`

Lambda Calculus Reference

```
-- 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))

-- 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

-- The definitions of DECR, SUB, and ISZ are elided
-- but you can still use them:
let DECR  = \n ->   -- (decrement n by one)
let SUB   = \n m -> -- (subtract m from n)
let ISZ   = \n ->   -- (return TRUE if n == 0 and FALSE otherwise)

-- Note: Since ZERO is the smallest Church numeral,
-- calls to DECR and SUB bottom out at ZERO.
-- For example, DECR ZERO evaluates to ZERO,
-- and SUB TWO THREE evaluates to ZERO.

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

Haskell Reference

-
  ```
  map :: (a -> b) -> [a] -> [b]
  map f []     = []
  map f (x:xs) = f x : map f xs
  ```

-
  ```
  foldr :: (a -> b -> b) -> b -> [a] -> b
  foldr f b []     = b
  foldr f b (x:xs) = f x (foldr f b xs)
  ```

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

  The list constructor operator. Takes an element and a list, and constructs a new list with the specified element as its head and the specified list as its tail.

  ```
  > 3 : [4, 5]
  [3, 4, 5]
  ```

- `(+) :: Num a => a -> a -> a`

  Returns the sum of its two arguments, e.g.,

  ```
  > 3 + 4
  7
  ```

- `(&&) :: Bool -> Bool -> Bool`

  The logical ‘and’ operation.

  ```
  > True && False
  False
  > True && True
  True
  ```

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

  Append two lists, e.g.,

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

- `(==) :: Eq a => a -> a -> Bool`

  Compare arguments for equality, e.g.,

  ```
  > False == True
  False
  > "apple" == "apple"
  True
  ```