CSE114A, Spring 2025: Final Exam

Instructor: Lindsey Kuper

June 10, 2025

Student name: _____

CruzID: _____@ucsc.edu

This exam has 20 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 completely fill in the 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. (5 points) Here is a series of β-reduction steps evaluating a lambda calculus expression to normal form:

```
(\f -> f ((\g h -> g g) (\y -> f y))) (\z -> z)
=b> (\z -> z) ((\g h -> g g) (\y -> (\z -> z) y)) -- result of step 1
=b> (\g h -> g g) (\y -> (\z -> z) y)             -- result of step 2
=b> \h -> (\y -> (\z -> z) y) (\y -> (\z -> z) y) -- result of step 3
=b> \h -> (\y -> (\z -> z) y) (\y -> y)           -- result of step 4
=b> \h -> (\y -> y) (\y -> y)                     -- result of step 5
=b> \h -> (\y -> y)                               -- result of step 6
```

For this question, you will write an *alternative reduction sequence* for this same expression (by choosing different redexes than the above reduction sequence chose). For full credit, the result of each step should be *different* from the result of any step in the reduction sequence above (except for the result of the last step, which should be the same!). Start each line with `=b>`, as if you were using Elsa, and do just one β-reduction step per line. Only use β-steps, not α-steps.

```
(\f -> f ((\g h -> g g) (\y -> f y))) (\z -> z)
```

Solution: Here is one possible solution:

```
-- Choose `(\g h -> g g) (\y -> f y)` as the redex
-- result of step 1:
=b> (\f -> f (\h -> (\y -> f y) (\y -> f y))) (\z -> z)
-- Choose `(\y -> f y) (\y -> f y)` as the redex
-- result of step 2:
=b> (\f -> f (\h -> f (\y -> f y))) (\z -> z)
-- The entire expression is the only redex at this point
-- result of step 3:
=b> (\z -> z) (\h -> (\z -> z) (\y -> (\z -> z) y))
-- Choose the entire expression as the redex
 -- result of step 4:
=b> \h -> (\z -> z) (\y -> (\z -> z) y)
-- Choose `(\z -> z) (\y -> (\z -> z) y)` as the redex
 -- result of step 5:
=b> \h -> (\y -> (\z -> z) y)
-- `(\z -> z) y` is the only redex at this point
-- result of step 6:
=b> \h -> (\y -> y)
```

For the next two questions, you may use any of the helper functions defined in the provided lambda calculus reference at the end of the exam.

2. (4 points) For this question, you'll implement a lambda calculus function called `APPLY`. The `APPLY` function takes a *pair* as its argument, and the elements of this pair are as follows:

   - The first element is an operation that takes two Church numerals and returns a Church numeral. (Some examples of operations like this include `ADD`, `SUB`, and `MUL`.)
   - The second element is itself a pair, where both elements are Church numerals.

   APPLY returns the Church numeral that is the result of applying the given operation to the two given Church numerals. For example, `APPLY (PAIR ADD (PAIR ZERO ONE))` will apply `ADD` to the arguments `ZERO` and `ONE`, returning `ONE`.

   Here are a few more example calls to `APPLY` in Elsa:

```
APPLY (PAIR SUB (PAIR THREE ZERO)) =~> THREE
APPLY (PAIR SUB (PAIR ZERO THREE)) =~> ZERO
APPLY (PAIR MUL (PAIR TWO TWO)) =~> FOUR

let APPLY =
```

   Solution: Here is one simple correct answer:

```
\p -> (FST p) (FST (SND p)) (SND (SND p))
```

3. Consider a recursive function $combine$ that is defined as follows:

   - $combine(op, 0, b) = b$
   - $combine(op, n, b) = op(n, combine(op, n-1, b))$

   $combine$ takes three arguments. Its first argument is an operation op that takes two natural numbers and returns a natural number. Its second argument is a natural number n, and its third argument is a natural number b. If n is 0, then $combine$ returns b. Otherwise, $combine$ returns the result of applying op to n and $combine(op, n-1, b)$. (Notice that on the recursive call, n is decremented by 1, and b stays the same.)

   For example, here's a call to $combine$ where op is addition ($+$), $n = 3$, and $b = 0$:

$$\begin{aligned} combine(+, 3, 0) &= 3 + combine(+, 2, 0) \\ &= 3 + (2 + combine(+, 1, 0)) \\ &= 3 + (2 + (1 + combine(+, 0, 0))) \\ &= 3 + (2 + (1 + 0)) \\ &= 6 \end{aligned}$$

   For this question, you will define a lambda calculus function `COMBINE` that implements the behavior of $combine$. For example, in Elsa, the above example would evaluate like this:

```
COMBINE ADD THREE ZERO =~> SIX
```

If we change the operation to `SUB`, since $3 - (2 - (1 - 0))$ evaluates to 2, we would get `COMBINE SUB THREE ZERO =~> TWO`. Remember that if its second argument is `ZERO`, `COMBINE` just returns its third argument, e.g., `COMBINE MUL ZERO TWO =~> TWO`.

You may assume that the first argument to `COMBINE` is an operation that takes two Church numerals (like `ADD`, `SUB`, or `MUL`, for instance) and returns a Church numeral, and you may further assume that `COMBINE`'s second and third arguments are Church numerals. You must use recursion for full credit.

```
let COMBINE1 = \rec -> \op -> \n -> \b -> ITE _____(part 3(a))_______
                                             _____(part 3(b))_______
                                             _____(part 3(c))_______

let COMBINE = _______(part 3(d))_______
```

a. (4 points) 3(a):

Solution: This is where we check a condition to know whether we are in the base case or not. `(ISZ n)` is a correct answer.

b. (4 points) 3(b):

Solution: This is the base case. `b` is a correct answer.

c. (4 points) 3(c):

Solution: This is the recursive case. `op n (rec op (DECR n) b)` is a correct answer.

d. (4 points) 3(d):

Solution:

```
Y COMBINE1
```

Part 2: Haskell

The Haskell reference at the end of the exam has information about library functions used in this section.

4. (3 points) What is the **type** of the following Haskell expression? (Choose only one answer.)

```
\x -> case x of
  Just y -> "larry"
  Nothing -> Nothing
```

- √ This expression is ill-typed; it doesn't have a type
- ◯ `Maybe a -> String`
- ◯ `Maybe a -> Maybe String`
- ◯ `Maybe String -> String`
- ◯ `Maybe String -> Maybe String`

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

```
\x -> case x of
  Just y  -> y
  Nothing -> Nothing
```

- ◯ This expression is ill-typed; it doesn't have a type
- ◯ `Maybe a -> a`
- ◯ `Maybe a -> Maybe a`
- √ `Maybe (Maybe a) -> Maybe a`
- ◯ `Maybe a -> Maybe (Maybe a)`

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

```
\g -> [g "walter", "rainbow", "sprinkles"]
```

- ◯ This expression is ill-typed; it doesn't have a type
- ◯ `(a -> String) -> [String]`
- ◯ `(String -> a) -> [a]`
- √ `(String -> String) -> [String]`
- ◯ `(String -> a) -> [String]`

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

```
\g -> [g "walter", g "rainbow", g "sprinkles"]
```

- ◯ This expression is ill-typed; it doesn't have a type
- ◯ `(a -> String) -> [a]`
- ◯ `(a -> String) -> [String]`
- √ `(String -> a) -> [a]`
- ◯ `(String -> String) -> [String]`

8. (3 points) What is the **type** of the following Haskell expression? (Choose only one answer.)

```
\x y -> [x, y "walter"]
```

- ◯ This expression is ill-typed; it doesn't have a type
- ◯ `a -> (a -> String) -> [String]`
- ◯ `String -> (String -> String) -> [String]`
- ◯ `a -> (String -> b) -> [b]`
- √ `a -> (String -> a) -> [a]`

9. (3 points) What is the **type** of the following Haskell expression? (Choose only one answer.)

```
\x y -> (x == y, x == y)
```

- ◯ This expression is ill-typed; it doesn't have a type
- √ `Eq a => a -> a -> (Bool, Bool)`
- ◯ `(Eq a, Eq b) -> a -> b -> (Bool, Bool)`
- ◯ `Eq a => a -> a -> (a, a)`
- ◯ `(Eq a, Eq b) -> a -> b -> (a, b)`

10. (3 points) What does the following Haskell expression **evaluate to**? (Choose only one answer.)

```
foldr (:) [] ["h", "e", "l", "l", "o"]
```

- ◯ This expression is ill-typed; it doesn't evaluate to anything
- √ `["h","e","l","l","o"]`
- ◯ `"hello"`
- ◯ `["hello"]`
- ◯ `["h","e","l","l","o",""]`

11. (5 points) Consider again the *combine* function from question 3. Here's its definition again:

    - $combine(op, 0, b) = b$
    - $combine(op, n, b) = op(n, combine(op, n-1, b))$

    Implement a version of the *combine* function as a Haskell function `combine` that operates on `Int`s instead of Church numerals.

    Here are some example calls to `combine`:

```
ghci> combine (*) 3 1
6
ghci> combine (+) 3 0
6
ghci> combine (+) 1 2
3
ghci> combine (+) 0 4
4
ghci> combine (-) 3 5
-3
```

    (Fun fact: because the Haskell function `combine` operates on `Int`s instead of Church numerals, it'll behave differently than the lambda calculus function `COMBINE` from question 3. For example, `combine (-) 3 5` will evaluate to `-3`, because `3 - (2 - (1 - 5))` evaluates to `-3` in Haskell, but `COMBINE SUB THREE FIVE` will evaluate to `ONE`, because `SUB ONE FIVE` evaluates to `ZERO`!)

    The type signature of `combine` is provided for you below; fill in the rest of the definition. Use pattern matching, and *do not* use an `if`-expression. (You can use pattern guards, but it's not required.)

```
combine :: (Int -> Int -> Int) -> Int -> Int -> Int
```

Solution: A simple implementation is:

```
combine op 0 b = b
combine op n b = op n (combine op (n-1) b)
```

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

In this section, you may use any of the helper functions defined in the provided lambda calculus reference at the end of the exam.

Let's implement a cute little programming language that we'll call *Smol*. The Smol language has string literals, integer literals, lambda abstractions, variables, a string concatenation operation, an integer addition operation, `let`-expressions, and function calls. Accordingly, we'll define the following `Expr` data type, which defines a grammar of abstract syntax trees (ASTs) for Smol (using the type alias `Id` for `String`):

```
type Id = String

data Expr
  = EStr String          -- String literal
  | EInt Int             -- Integer literal
  | ELam Id Expr         -- Lambda abstraction: `\x -> e`
  | EVar Id              -- Variable
  | EConcat Expr Expr    -- String concatenation: `e1 ++ e2`
  | EAdd Expr Expr       -- Integer addition: `e1 + e2`
  | ELet Id Expr Expr    -- let-expression: `let x = e1 in e2`
  | EApp Expr Expr       -- Function call: `e1 e2`
```

For example, the Smol program `3 + (4 + x)` has the AST

```
EAdd (EInt 3) (EAdd (EInt 4) (EVar "x"))
```

and the Smol program `let f = \x -> x ++ " is cute" in f "mo"` has the AST

```
ELet "f" (ELam "x" (EConcat (EVar "x") (EStr " is cute")))
  (EApp (EVar "f") (EStr "mo"))
```

12. (3 points) Be the parser: translate the following Smol program into its corresponding `Expr`.

```
(\s -> s) (let name = "larry" in "hello" ++ name)
```

Solution:

```
EApp (ELam "s" (EVar "s"))
     (ELet "name" (EStr "larry")
       (EConcat (EStr "hello") (EVar "name")))
```

Next, we'll write a Smol interpreter. Ideally, an `Expr` will evaluate to a `Value`, as defined by the following data type:

```
data Value = VStr String | VInt Int | VClos Id Expr Env
```

where `Env` is a list representation of an environment that maps program identifiers to their `Values`:

```
type Env = [(Id, Value)]
```

We can straightforwardly add new bindings to an environment with the `extendEnv` function:

```
extendEnv :: Id -> Value -> Env -> Env
extendEnv id value env = (id, value):env
```

There will be two kinds of errors that can arise when we're evaluating a Smol program: *unbound variable errors*, like when we try to evaluate the program `EAdd (EInt 4) (EVar "x")` and we don't have a binding for `"x"`, and *type errors*, like if (for example) a program tries to add strings or concatenate functions. We'll define an `Error` type for these situations, with two constructors:

```
data Error = UnboundError | TypeError
```

Next, we need to define a function `lookupInEnv` that takes an identifier and an environment, and returns either the value to which the identifier is bound in the environment, or returns an error value indicating that the identifier is unbound. Because we want to return *either* one thing or the other, we'll use Haskell's `Either` type. Recall that the definition of `Either` is:

```
data Either a b = Left a | Right b
```

and to construct values of `Either` type, we have to use the `Left` and `Right` constructors.

13. (4 points) The type signature of `lookupInEnv` and one of the cases is provided for you below; fill in the rest of the definition.

```
lookupInEnv :: Id -> Env -> Either Value Error
lookupInEnv id [] = Right UnboundError
```

Solution: A simple implementation is:

```
lookupInEnv id ((s,v):rest) = if id == s
                              then Left v
                              else lookupInEnv id rest
```

14. We're now ready to write our Smol interpreter. Fill in the blanks in the following definition of `eval`. Your interpreter does *not* need to support recursive functions. The `EApp` case is missing on purpose; we'll deal with it in the next question!

```
eval :: Expr -> Env -> Either Value Error
eval (EStr s) _ = Left (VStr s)
eval (EInt n) _ = Left (VInt n)
eval (ELam id e) env = ____(14(a))____
eval (EVar id) env = lookupInEnv id env
eval (EConcat e1 e2) env = case (eval e1 env, eval e2 env) of
    (Right err     , _             ) -> Right err
    (_             , Right err     ) -> Right err
    (Left (VStr s1), Left (VStr s2)) -> ____(14(b))____
    (Left _        , Left _        ) -> Right TypeError
eval (EAdd e1 e2) env = case (eval e1 env, eval e2 env) of
    (Right err     , _             ) -> Right err
    (_             , Right err     ) -> Right err
    (Left (VInt n1), Left (VInt n2)) -> ____(14(c))____
    (Left _        , Left _        ) -> Right TypeError
eval (ELet id e1 e2) env = case eval e1 env of
    Left v    -> ____(14(d))____
    Right err -> Right err
```

a. (4 points) 14(a):

Solution:
```
Left (VClos id e env)
```

b. (4 points) 14(b):

Solution:
```
Left (VStr (s1 ++ s2))
```

c. (4 points) 14(c):

Solution:
```
Left (VInt (n1 + n2))
```

d. (4 points) 14(d):

Solution:
```
eval e2 (extendEnv id v env)
```

15. Lindsey implemented the `EApp` case of the Smol interpreter like this:

```
eval (EApp e1 e2) env = case (eval e1 env, eval e2 env) of
    (Right err, _) -> Right err
    (_, Right err) -> Right err
    (Left (VClos id e cEnv), Left arg) -> eval e (extendEnv id arg env)
    (Left (VStr _), _) -> Right TypeError
```

Everything compiled and ran fine, but when Lindsey tried evaluating this Smol program, she didn't get the result she expected!

```
let n = 0 in
  let f = \x -> n + x in
    let n = 6 in
      f n
```

a. (3 points) What does the above Smol program evaluate to under **static scope**?

Solution:
`6`

b. (3 points) What does the above Smol program evaluate to under **dynamic scope**?

Solution:
`12`

c. (4 points) Lindsey accidentally implemented *dynamic scope* in her Smol interpreter. **What small change** does she need to make to the `EApp` case above to implement *static scope* instead, and **how will this small change fix the problem**? (Answer in 2-3 sentences)

Solution: In the call `eval e (extendEnv id arg env)`, `env` should be `cEnv` instead. This will fix the problem by using the environment saved in the function's closure, instead of the dynamic environment.

(Alternatively, when pattern-matching on the closure, one could use `env` instead of verb `cEnv` as the pattern variable. Then the closure's `env` would overshadow the dynamic `env`, so extending `env` would be correct.)

Part 4: Unification and Type Inference

In the context of unification and type inference, we've discussed the concept of *substitutions*. A substitution is a mapping from *type variables* to *types*, where the types may themselves contain type variables (or be type variables). For example,

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

is a substitution that maps the type variable `a` to the type `Bool`, the type variable `b` to the type `Int -> c`, and the type variable `d` to the type `e`.

16. (3 points) Which of the following substitutions is *a unifier* for the types `(Bool -> Bool) -> a` and `b -> c`?

- √ (A) `[(c, a), (b, Bool -> Bool)]`
- ◯ (B) `[(Bool -> Bool, b), (c, a)]`
- ◯ (C) `[(a, Bool), (b, Bool), (c, Bool -> Bool)]`
- ◯ (A) and (B)
- ◯ (A), (B), and (C)
- ◯ Cannot unify

17. (3 points) Which of the following substitutions is *a unifier* for the types `Bool -> Bool -> a` and `a -> c`?

- ◯ (A) `[(a, Bool -> Bool), (c, Bool -> Bool)]`
- √ (B) `[(a, Bool), (c, Bool -> Bool)]`
- ◯ (C) `[(a, Bool -> Bool), (c -> Bool)]`
- ◯ (A) and (B)
- ◯ (A), (B), and (C)
- ◯ Cannot unify

18. (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)]`
- √ (A) and (B)
- ◯ (A), (B), and (C)
- ◯ Cannot unify

19. (4 points) If possible, write down a substitution that is *a unifier* for the types `a -> b` and `c -> Bool -> d`. If these types do not unify, write "Cannot unify".

Solution: One correct solution is:

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

Here, `c -> Bool -> d` is syntactic sugar for `c -> (Bool -> d)`, so `[(a, c -> Bool), (b, d)]` would be an incorrect answer. However, other solutions are possible.

20. In the previous section of this exam, we wrote an interpreter for the Smol language. Here's what would happen if we tried to evaluate the Smol expression `(\x -> x + 3) "mo"`:

```
ghci> eval (EApp (ELam "x" (EAdd (EVar "x") (EInt 3))) (EStr "mo")) []
Right TypeError
```

This is a run-time type error. To catch these kinds of errors *prior* to run time, we could define a *type system* for Smol and then implement a *type checker*. Below are the typing rules we'll use for Smol, all of which are standard:

```
             (x,T1):G |- 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] ----------------------------------     [T-Int] -------------
                G |- e1 + e2 :: Int                    G |- n :: Int

           G |- e1 :: Str       G |- e2 :: Str
[T-Concat] ----------------------------------    [T-Str] -------------
                G |- e1 ++ e2 :: Str                    G |- s :: Str


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

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

Let's use our typing rules to make sure that the Smol expression

```
(\x -> x ++ " is cute") "phoebe"
```

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

```
      (x,Str) in [(x,Str)]
[20a]--------------------  [T-Str]-----------------------------
     [(x,Str)] |- x :: Str          [(x,Str)] |- " is cute" :: Str
[20b]-------------------------------------------------------------
            [(x,Str)] |- x ++ " is cute" :: [20c]
       [20d]-------------------------------------
             [] |- \x -> x ++ " is cute" :: [20e]     [] |- "phoebe" :: Str
       [20f]-------------------------------------------------------------
                  [] |- (\x -> x ++ " is cute") "phoebe" :: Str
```

a. (½ point) 20(a):

Solution:
`T-Var`

b. (½ point) 20(b):

Solution:
`T-Concat`

c. (½ point) 20(c):

Solution:
`Str`

d. (½ point) 20(d):

Solution:
`T-Lam`

e. (½ point) 20(e):

Solution:
`Str -> Str`

f. (½ point) 20(f):

Solution:
`T-App`

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))
let FOUR  = \f x -> f (f (f (f x)))
let FIVE  = \f x -> f (f (f (f (f x))))
let SIX   = \f x -> f (f (f (f (f (f x)))))

-- Booleans
let TRUE  = \x y -> x
let FALSE = \x y -> y
let ITE   = \b x y -> b x y
let AND   = \b1 b2 -> ITE b1 b2 FALSE

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

-- Arithmetic
let INCR  = \n f x -> f (n f x)
let ADD   = \n m -> n INCR m
let MUL   = \n m -> n (ADD m) ZERO

-- The definitions of DECR, SUB, ISZ, and EQL 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)
let EQL   = \n m -> -- (return TRUE if n == m 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

- ```
  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]
  > 3 : (4 : (5 : []))
  [3, 4, 5]
  ```

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

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

  ```
  > 3 + 4
  7
  ```

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

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

  ```
  > 3 - 4
  -1
  ```

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

  Appends two lists, e.g.,

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

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

  Compares two arguments for equality, e.g.,

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