CS242 Final
Fall 2023

- Please read all instructions (including these) carefully.
- There are 5 questions on the exam, all with multiple parts. You have 3 hours to work on the exam.
- The exam is open note. You may use laptops, phones and e-readers to read electronic notes, but not for computation or access to the internet for any reason.
- Please write your answers in the space provided on the exam, and clearly mark your solutions. Do not write on the back of exam pages or other pages.
- Solutions will be graded on correctness and clarity. Each problem has a relatively simple and straightforward solution. You may get as few as 0 points for a question if your solution is far more complicated than necessary. Partial solutions will be graded for partial credit.

NAME: ____

In accordance with both the letter and spirit of the Honor Code, I have neither given nor received assistance on this examination.

SIGNATURE: ____

Problem	Max points	Points
1	18	
2	24	
3	24	
4	14	
5	24	
TOTAL	104	

1. **Lifetimes** (18 points)

Consider the following Rust programs. Each program has several holes labeled '?'. For each proposed signature with the holes filled in, answer **T** if the program would type check with that signature and F otherwise. Assume that '...' is some valid boolean expression.

(a)
```
fn swap<?>(x: &? i32, y: &? i32) -> (&? i32, &? i32) {
    (y, x)
}
```

i. `swap<'a, 'b>(x: &'a i32, y: &'b i32) -> (&'a i32, &'b i32)`

T/F: F

ii. `swap<'a, 'b>(x: &'a i32, y: &'b i32) -> (&'b i32, &'a i32)`

T/F: T

iii. `swap<'a>(x: &'a i32, y: &'a i32) -> (&'a i32, &'a i32)`

T/F: T

(b)
```
fn swap<?>(x: &? i32, y: &? i32) -> (&? i32, &? i32) {
    if ... { (x, y) } else { (y, x) }
}
```

i. `swap<'a, 'b>(x: &'a i32, y: &'b i32) -> (&'a i32, &'b i32)`

T/F: F

ii. `swap<'a, 'b>(x: &'a i32, y: &'b i32) -> (&'b i32, &'a i32)`

T/F: F

iii. `swap<'a>(x: &'a i32, y: &'a i32) -> (&'a i32, &'a i32)`

T/F: T

(c)
```
fn swap<?>(x: &? str, y: &? str) -> (&? str, &? str) {
    (y, x)
}
fn main() {
    let r1: &'static str = "hello";
    let r3 = {
        let s = String::new("world");
        // String::as_str has the following signature:
        // as_str<'a>(&'a self) -> &'a str
        let (r2, _) = swap(s.as_str(), r1);
        r2
    };
    println!("{r3}");
}
```

i. `swap<'a, 'b>(x: &'a str, y: &'b str) -> (&'a str, &'b str)`

T/F: F

ii. `swap<'a, 'b>(x: &'a str, y: &'b str) -> (&'b str, &'a str)`

T/F: T

iii. `swap<'a>(x: &'a str, y: &'a str) -> (&'a str, &'a str)`

T/F: F

2. **Ownership** (24 points)

Much research has been dedicated to the study of ownership type systems as a tool to manage resources such as memory. The type rules below show an early ownership system known as *affine types* for the simply typed lambda calculus (STLC). The basic idea behind affine types is that *a variable can be used at most once.*

The type judgements have the form $\Gamma \vdash e : t \,;\, \Gamma'$, so each rule results in a new context as well as a type. Rules are read: given the unused variables Γ, evaluating expression e yields a value of type t and leaves variables Γ' unused. We use $\Gamma \setminus \{x\}$ to denote the context Γ with any binding for x removed, should it exist. For simplicity, the typing rules enforce that there is no variable *shadowing*—any variable bound by a lambda abstraction is not bound by any other lambda abstraction nested inside it (see Abs).

$$\begin{aligned} t \in \text{Type} &::= \mathsf{int} \mid t \rightarrow t \\ i \in \text{Int} &::= 0 \mid 1 \mid \cdots \\ e \in \text{Expr} &::= i \mid x \mid \lambda x{:}\,t.\,e \mid e_1\,e_2 \\ \Gamma \in \text{Context} &::= x_1 : t_1, \ldots, x_n : t_n \end{aligned}$$

$$\frac{}{\Gamma \vdash i : \mathsf{int} \,;\, \Gamma}\ [\textsc{Int}] \qquad \frac{x \notin \Gamma \quad \Gamma, x : t_1 \vdash e : t_2 \,;\, \Gamma'}{\Gamma \vdash \lambda x{:}\,t_1.\,e : t_1 \rightarrow t_2 \,;\, \Gamma' \setminus \{x\}}\ [\textsc{Abs}]$$

$$\frac{x : t \in \Gamma}{\Gamma \vdash x : t \,;\, \Gamma \setminus \{x\}}\ [\textsc{Var}] \qquad \frac{\Gamma \vdash e_1 : t_1 \rightarrow t_2 \,;\, \Gamma' \quad \Gamma' \vdash e_2 : t_1 \,;\, \Gamma''}{\Gamma \vdash e_1\,e_2 : t_2 \,;\, \Gamma''}\ [\textsc{App}]$$

(a) Write a lambda expression that is well-typed in the ordinary STLC (without shadowing), but not well-typed in the affine STLC.

$\lambda f{:}\,\mathsf{int} \rightarrow \mathsf{int}.\ f\ (f\ 0)$

(b) Consider adding `let` expressions to the affine STLC:

$$
\begin{aligned}
t \in \text{Type} &::= \mathsf{int} \mid t \to t \\
i \in \text{Int} &::= 0 \mid 1 \mid \cdots \\
e \in \text{Expr} &::= i \mid x \mid \lambda x{:}\, t.\, e \mid e_1\, e_2 \mid \text{let } x = e_1 \text{ in } e_2 \\
\Gamma \in \text{Context} &::= x_1 : t_1, \ldots, x_n : t_n
\end{aligned}
$$

Give an affine type rule for let compatible with the other rules (given above) for an affine type system. Ensure there is no shadowing.

$$
\frac{x \notin \Gamma \quad \Gamma \vdash e_1 : t_1 \,;\, \Gamma' \quad \Gamma', x : t_1 \vdash e : t_2 \,;\, \Gamma''}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : t_2 \,;\, \Gamma'' \setminus \{x\}} \; [\textsc{Let}]
$$

3. **Continuations** (24 points)

The "with"-pattern is a popular resource management abstraction in functional languages. Consider a hypothetical Scheme/Racket dialect that provides a higher-order function `with-open-file`, which accepts 2 arguments: a `path` and a function `f`. When called, `with-open-file` opens a file object at the given file `path`, then calls function `f` with this file object, and finally closes the file object after `f` returns.

For example, the following code uses `with-open-file` to open `123.txt` and reads its contents. `read-file` is a built-in function that reads the contents of an *open* file object.

```
(with-open-file "123.txt"
  (lambda (file) (read-file file)))
```

The following is an implementation of `with-open-file`. Built-in function `open-file` opens a file object at a given path, and `close-file` closes a file object. Once `close-file` is called on a file object, it is no longer open and cannot be used for file operations, such as `read-file`.

```
(define (with-open-file path f)
  (let* ((file (open-file path))
         (return-value (f file)))
    (close-file file)
    return-value))
```

In this example, `with-open-file` calls each of `open-file` and `close-file` exactly once. In fact, `with-open-file` is guaranteed to issue matching pairs of `open-file` and `close-file` calls no matter what computation we do in `f`, *as long as* `call/cc` *is not involved.*

For each of the following expressions, answer how many times `open-file` and `close-file` are called during evaluation. Assume that erroneous file operations such as using `read-file` on a closed file object are no-ops and do not terminate the program:

(a)
```
(let* ((count 0)
       (k #f))
  (call/cc (lambda (k1) (set! k k1)))
  (with-open-file "123.txt"
    (lambda (file) (read-file file)))
  (set! count (+ count 1))
  (if (< count 10)
      (k #f)
      #f))
```

`open-file` is called | 10 | times, `close-file` is called | 10 | times.

(b)
```
(call/cc
  (lambda (k)
    (with-open-file "123.txt"
       (lambda (file)
         (k (read-file file))))))
```
`open-file` is called [1] times, `close-file` is called [0] times.

(c)
```
(let* ((count 0)
       (k #f))
  (with-open-file "123.txt"
    (lambda (file)
      (read-file file)
      (call/cc (lambda (k1) (set! k k1)))))
  (set! count (+ count 1))
  (if (< count 10)
      (k #f)
      #f))
```
`open-file` is called [1] times, `close-file` is called [10] times.

(d)
```
(let* ((count 0)
       (k #f))
  (call/cc (lambda (k1) (set! k k1)))
  (with-open-file "123.txt"
    (lambda (file)
      (read-file file)
      (set! count (+ count 1))
      (if (< count 10)
          (k #f)
          #f))))
```
`open-file` is called [10] times, `close-file` is called [1] times.

In the following questions, we explore how we can guarantee `with-open-file` executes a matching close for every open even in the presence of `call/cc`. Fill in the blank boxes to make each expression:

- Maintain matching pairs of calls to `open-file` and `close-file`, i.e. close is only called on open files and all open file objects are eventually closed.
- Guarantee that `read-file` is only ever called with an open file object.

If the expression already satisfies the above properties, fill in `#f`. The reference answer contains one line of code for each blank box.

Hint: the following expressions correspond one-to-one with the expressions we considered in the first part of this problem. The only modification is that we replace some captured continuations `k` with "wrapped" continuations `(lambda (v) [ ] (k v))`, so that you can run custom setup/cleanup code when these "wrapped" continuations are invoked.

(e)

```
(let* ((count 0)
       (k #f))
  (call/cc (lambda (k1)
             (set! k (lambda (v)
                       [ #f                                    ]
                       (k1 v)))))
  (with-open-file "123.txt"
    (lambda (file) (read-file file)))
  (set! count (+ count 1))
  (if (< count 10)
      (k #f)
      #f))
```

(f)

```
(call/cc
  (lambda (k)
    (with-open-file "123.txt"
      (lambda (file)
        ((lambda (v)
           [ (close-file file)                          ]
           (k v))
         (read-file file))))))
```

(g)
```
(let* ((count 0)
       (k #f))
  (with-open-file "123.txt"
    (lambda (file)
      (read-file file)
      (call/cc (lambda (k1)
                 (set! k (lambda (v)
                           (set! file (open-file "123.txt"))
                           (k1 v)))))))
  (set! count (+ count 1))
  (if (< count 10)
      (k #f)
      #f))
```

(h)
```
(let* ((count 0)
       (k #f))
  (call/cc (lambda (k1) (set! k k1)))
  (with-open-file "123.txt"
    (lambda (file)
      (read-file file)
      (set! count (+ count 1))
      (if (< count 10)
          ((lambda (v)
             (close-file file)
             (k v))
           #f)
          #f))))
```

4. **Monads & Haskell** (14 points)

(a) T/F: the following function types are equivalent to "`a -> b -> c -> d`":

i.	T	`a -> b -> c -> d` (example)
ii.	F	`a -> ((b -> c) -> d)`
iii.	T	`(a -> (b -> c -> d))`
iv.	F	`((((a -> b) -> c -> d)))`
v.	F	`(a -> b) -> (c -> d)`

(b) Recall the following identity for `do`-notation:

```
do  a <- e1
    e2
```

is equivalent to

```
e1 >>= \a -> e2
```

Convert the following Haskell expression from `do`-notation to use `(>>=)` and `return`: Recall that a lambda is declared in Haskell through the syntax `(\arg1 arg2 arg3 -> ...)`

```
convertMe f y g h m k = do x <- f y
                           let z = g x y
                            in if z
                                  then do h
                                          m x
                                  else return ()
                           n <- return x
                           n k
```

```
convertMe f y g h m k =
```

```
(f y) >>= (\x ->
  (let z = g x y
    in if z then (h >>= (\_ -> m x))
              else (return ())
    ) >>= (\_ ->
      (return x) >>= (\n -> n k)))
```

5. **Applicative vs. Monad** (24 points)

```
putStrStderr :: String -> IO () -- prints a string to stderr
putStrStderr s = hPutStr stderr s

putStrStdout :: String -> IO () -- prints a string to stdout
putStrStdout s = hPutStr stdout s

not :: Bool -> Bool
not True = False
not False = True

trueVal :: IO Bool
trueVal = putStrStdout "trueVal," >> return True

falseVal :: IO Bool
falseVal = putStrStdout "falseVal," >> return False

trueBranch :: IO Char
trueBranch = putStrStdout "trueBranch," >> return 'T'

falseBranch :: IO Char
falseBranch = putStrStdout "falseBranch," >> return 'F'

cond :: Bool -> a -> a -> a
cond True t f = t
cond False t f = f

-- prints a char to stderr (remember that String is an alias for [Char])
printResult :: Char -> IO ()
printResult x = putStrStderr [x]

class Functor m where
  fmap :: (a -> b) -> m a -> m b

-- infix shorthand for fmap
(<$>) :: Functor m => (a -> b) -> m a -> m b
(<$>) = fmap

class Functor m => Applicative m where
  (<*>) :: m (a -> b) -> m a -> m b
  pure :: a -> m a

class Applicative m => Monad m where
  (>>=) :: m a -> (a -> m b) -> m b
  return :: a -> m a
```

Implement the following functions in a way that satisfies the given type signature and example result. You may only use core language constructs of Haskell (lambdas, if-then, etc.), i.e., no external functions outside of those defined above. No solutions require let-in or where clauses, so these should also not be used. If the solution is the same as a previous part (for example, part 3), write "same as part 3". If no such function can be implemented, write "not possible". To improve readability, everything that matches part 1 has been marked in light gray in subsequent parts.

```
part1 :: Applicative m => m Bool -> m b -> m b -> m b
description: applicative default
(part1 trueVal trueBranch falseBranch) >>= printResult
-- stdout: "trueVal,trueBranch,falseBranch,"
-- stderr: "T"
(part1 falseVal trueBranch falseBranch) >>= printResult
-- stdout: "falseVal,trueBranch,falseBranch,"
-- stderr: "F"
```

```
part1 cc tt ff = cond <$> cc <*> tt <*> ff
```

```
part2 :: Monad m => m Bool -> m b -> m b -> m b
description: monad default
(part2 trueVal trueBranch falseBranch) >>= printResult
-- stdout: "trueVal,trueBranch,falseBranch,"
-- stderr: "T"
(part2 falseVal trueBranch falseBranch) >>= printResult
-- stdout: "falseVal,trueBranch,falseBranch,"
-- stderr: "F"
```

```
part2 cc tt ff = Same as part 1
```

```
part3 :: Applicative m => m Bool -> m b -> m b -> m b
description: description: applicative fixed return
(part3 trueVal trueBranch falseBranch) >>= printResult
-- stdout: "trueVal,trueBranch,falseBranch,"
-- stderr: "F"
(part3 falseVal trueBranch falseBranch) >>= printResult
-- stdout: "falseVal,trueBranch,falseBranch,"
-- stderr: "F"
```

```
part3 cc tt ff = (\_ _ ff' -> ff') <$> cc <*> tt <*> ff
```

```
part4 :: Monad m => m Bool -> m b -> m b -> m b
description: monad fixed return
(part4 trueVal trueBranch falseBranch) >>= printResult
-- stdout: "trueVal,trueBranch,falseBranch,"
-- stderr: "F"
(part4 falseVal trueBranch falseBranch) >>= printResult
-- stdout: "falseVal,trueBranch,falseBranch,"
-- stderr: "F"
```

```
part4 cc tt ff = Same as part 3
```

```
part5 :: Applicative m => m Bool -> m b -> m b -> m b
description: applicative conditional side effect
(part5 trueVal trueBranch falseBranch) >>= printResult
-- stdout: "trueVal,trueBranch,"
-- stderr: "T"
(part5 falseVal trueBranch falseBranch) >>= printResult
-- stdout: "falseVal,falseBranch,"
-- stderr: "F"
```

```
part5 cc tt ff = Not possible
```

```
part6 :: Monad m => m Bool -> m b -> m b -> m b
description: monad conditional side effect
(part6 trueVal trueBranch falseBranch) >>= printResult
-- stdout: "trueVal,trueBranch,"
-- stderr: "T"
(part6 falseVal trueBranch falseBranch) >>= printResult
-- stdout: "falseVal,falseBranch,"
-- stderr: "F"
```

```
part6 cc tt ff = cc >>= (\c -> cond c tt ff)
```

```
part7 :: Applicative m => m Bool -> m b -> m b -> m b
description: applicative fixed return & conditional side effect
(part7 trueVal trueBranch falseBranch) >>= printResult
-- stdout: "trueVal,trueBranch,"
-- stderr: "F"
(part7 falseVal trueBranch falseBranch) >>= printResult
-- stdout: "falseVal,falseBranch,"
-- stderr: "F"
```

```
part7 cc tt ff = Not possible
```

```
part8 :: Monad m => m Bool -> m b -> m b -> m b
description: monad fixed return & conditional side effect
(part8 trueVal trueBranch falseBranch) >>= printResult
-- stdout: "trueVal,trueBranch,"
-- stderr: "F"
(part8 falseVal trueBranch falseBranch) >>= printResult
-- stdout: "falseVal,falseBranch,"
-- stderr: "F"
```

```
part8 cc tt ff = Not possible
```

```
part9 :: Applicative m => m Bool -> m b -> m b -> m b
description: applicative no cond
(part9 trueVal trueBranch falseBranch) >>= printResult
-- stdout: "trueBranch,falseBranch,"
-- stderr: "T"
(part9 falseVal trueBranch falseBranch) >>= printResult
-- stdout: "trueBranch,falseBranch,"
-- stderr: "F"
```

```
part9 cc tt ff = Not possible
```

```
part10 :: Monad m => m Bool -> m b -> m b -> m b
description: monad no cond
(part10 trueVal trueBranch falseBranch) >>= printResult
-- stdout: "trueBranch,falseBranch,"
-- stderr: "T"
(part10 falseVal trueBranch falseBranch) >>= printResult
-- stdout: "trueBranch,falseBranch,"
-- stderr: "F"
```

```
part10 cc tt ff = Not possible
```

```
part11 :: Applicative m => m Bool -> m b -> m b -> m b
description: applicative conditional side effect order
(part11 trueVal trueBranch falseBranch) >>= printResult
-- stdout: "trueVal,trueBranch,falseBranch,"
-- stderr: "T"
(part11 falseVal trueBranch falseBranch) >>= printResult
-- stdout: "falseVal,falseBranch,trueBranch,"
-- stderr: "F"
```

```
part11 cc tt ff = Not possible
```

```
part12 :: Monad m => m Bool -> m b -> m b -> m b
description: monad conditional side effect order
(part12 trueVal trueBranch falseBranch) >>= printResult
-- stdout: "trueVal,trueBranch,falseBranch,"
-- stderr: "T"
(part12 falseVal trueBranch falseBranch) >>= printResult
-- stdout: "falseVal,falseBranch,trueBranch,"
-- stderr: "F"
```

```
part12 cc tt ff = cc >>= (\c -> if c
                    then (tt >>= (\t -> ff >>= (\_ -> return t)))
                    else (ff >>= (\f -> tt >>= (\_ -> return f))))
```