CS 3520/6520 Fall 2023
Practice Midterm Exam 2

Name: _____

Instructions: You have eighty minutes to complete this open-book, open-note exam. Electronic devices are allowed only to consult notes, slides, or books (not videos) from local storage. Network use to look up technical information is prohibited, you can use DrRacket only as an editor (not running any programs), and use of other interpreters or large language models is not allowed. **Write only on the front side of each page**, and ask the proctor for extra pages if needed.

Note on actual exam: Like this practice exam, the actual exam will refer to the `"lambda_k.rhm"` interpreter for the questions that are not about eager versus lazy evaluation. If you need the interpreter for reference to answer the questions, please have a copy ready.

For each of the following, show the results produced by an eager variant of Moe and a lazy variant of Moe and (redundantly) circle whether the results are the **same** or **different**. When showing results, write "error" for an error result, write "function" for a function result, and write a specific number for a number result. Two function results are always the **same**, even if the two functions might behave differently when applied. Similarly, two errors are the **same** even if they might come with different messages.

1. `1 + 2` 5 points

Eager: _____ Lazy: _____ same / different

2. `(fun (y): 12)(1(2))` 5 points

Eager: _____ Lazy: _____ same / different

3. `fun (x): ((fun (y): 12)(1(2)))` 5 points

Eager: _____ Lazy: _____ same / different

4. `1 + ((fun (y): 12)(1(2)))` 5 points

Eager: _____ Lazy: _____ same / different

5. `1 + ((fun (x): 1 + 13)(1 + (fun (z): 12)))` 5 points

Eager: _____ Lazy: _____ same / different

6. `1 + ((fun (x): x + 13)(1 + (fun (z): 12)))` 5 points

Eager: _____ Lazy: _____ same / different

7. Suppose a garbage-collected interpreter uses the following three kinds of records: 15 points

- Tag `1`: a record containing two pointers
- Tag `2`: a record containing one pointer and one integer
- Tag `3`: a record containing one integer

The interpreter has one register, which always contains a pointer, and a memory pool of size 22. The allocator/collector is a two-space copying collector, so each space is of size 11. Records are allocated consecutively in to-space, starting from the first memory location, 0.

The following is a snapshot of memory just before a collection where all memory has been allocated:

Register: `8`

To space: `1 3 8 3 0 2 3 7 2 0 8`

What are the values in the register and the new to-space (which is also addressed starting from 0) after collection? Assume that unallocated memory in to-space contains 0.

Register:

To space:

In "lambda_k.rhm", what final result will the following `continue` calls produce? Show your answer as a Shplait `Value`, or write *error* if the `continue` call leads to an error instead of a result `Value`.

The actual exam will have fewer of these.

8. 5 points

```
continue(doPlusK(intV(8),
                 doneK()),
         intV(-1))
```

9. 5 points

```
continue(doAppK(closV(#'x,
                      parse('x * x'),
                      mt_env),
                doPlusK(intV(1),
                        doneK())),
         intV(3))
```

10. 5 points

```
continue(appArgK(parse('fun (f): f(y)'),
                 extend_env(bind(#'y, intV(5)), mt_env),
                 doneK()),
         intV(3))
```

11. 5 points

```
continue(appArgK(parse('fun (f): f(y)'),
                 extend_env(bind(#'y, intV(5)), mt_env),
                 doneK()),
         closV(#'g,
               parse('g(fun (q): q + (-1 * y))'),
               extend_env(bind(#'y, intV(7)), mt_env)))
```

Each remaining question shows an expression plus a candidate trace of `interp` and `continue` using the `"lambda_k.rhm"` interpreter. The trace should show all calls to `interp` and `continue` in the right order with the right arguments. If `interp` or `continue` eventually reports an error, the trace should show *error* at the end of the trace, and without omitting any calls to `interp` or `continue` that are made. A final, non-error result does not need to be shown.

For each question, mark the trace as "correct" if it correctly shows the complete `interp` and `continue` trace. For an incorrect trace, identify the first place where the trace is wrong (which would be the end if the trace is incomplete) and provide the correct next term—either a full `interp` call or a full `continue` call—that should appear at that position.

Keep in mind that `parse` desugars `let`, so `parse('let x = 1: x')` is interchangeable with `parse('(fun (x): x)(1)')`, for example.

The actual exam will have fewer of these.

12. 10 points

```
  2 + 1
```

```
[1]  interp(parse('2 + 1'),
            mt_env,
            doneK())
[2]  interp(parse('2'),
            mt_env,
            K1 = plusSecondK(parse('1'), mt_env, doneK()))
[3]  continue(K1,
              intV(2))
[4]  interp(parse('3'),
            mt_env,
            K2 = doPlusK(intV(2), doneK()))
[5]  continue(K2,
              intV(3))
[6]  continue(doneK(),
              intV(5))
```

13. 10 points

```
fun (x):
  5
```

```
[1] interp(parse('fun (x): 5'),
           mt_env,
           doneK())
[2] continue(doneK(),
             closV(#'x, parse('5'), mt_env))
```

14. 10 points

```
let f = (fun (x):
           x + 1):
  f(10)
```

```
[1]  interp(parse('(fun (f): f(10))(fun (x): x + 1)'),
            mt_env,
            doneK())
[2]  interp(parse('fun (f): f(10)'),
            mt_env,
            K1 = appArgK(parse('fun (x): x + 1'), mt_env, doneK()))
[3]  continue(K1,
              V1 = closV(#'f, parse('f(10)'), mt_env))
[4]  interp(parse('fun (x): x + 1'),
            mt_env,
            K2 = doAppK(V1, doneK()))
[5]  continue(K2,
              V2 = closV(#'x, parse('x + 1'), mt_env))
[6]  interp(parse('f(10)'),
            E1 = extend_env(bind(#'f, V2), mt_env),
            doneK())
[7]  interp(parse('f'),
            E1,
            K3 = appArgK(parse('10'), E1, doneK()))
[8]  continue(K3,
              V2)
[9]  interp(parse('10'),
            E1,
            K4 = doAppK(V2, doneK()))
[10] continue(K4,
              intV(10))
[11] interp(parse('x + 1'),
            E2 = extend_env(bind(#'x, intV(10)), mt_env),
            doneK())
[12] interp(parse('x'),
            E2,
            K5 = plusSecondK(parse('1'), E2, doneK()))
[13] continue(K5,
              intV(10))
[14] interp(parse('1'),
            E2,
            K6 = doPlusK(intV(10), doneK()))
[15] continue(K6,
              intV(1))
[16] continue(doneK(),
              intV(11))
```

15. 10 points

```
let f = (fun (x):
           x + 1):
  f
```

```
[1] interp(parse('(fun (f): f)(fun (x): x + 1)'),
           mt_env,
           doneK())
[2] interp(parse('fun (f): f'),
           mt_env,
           K1 = appArgK(parse('fun (x): x + 1'), mt_env, doneK()))
[3] continue(K1,
             V1 = closV(#'f, parse('f'), mt_env))
[4] interp(parse('fun (x): x + 1'),
           mt_env,
           K2 = doAppK(V1, doneK()))
[5] continue(K2,
             closV(#'x, parse('x + 1'), mt_env))
[6] interp(parse('x + 1'),
           mt_env,
           doneK())
[7] interp(parse('x'),
           mt_env,
           plusSecondK(parse('1'), mt_env, doneK()))
[8] error
```

16. 10 points

```
(fun (x):
   fun (y):
     fun (x):
       x)(1)(2)(0)
```

```
[1]  interp(parse('(fun (x): fun (y): fun (x): x)(1)(2)(0)'),
            mt_env,
            doneK())
[2]  interp(parse('(fun (x): fun (y): fun (x): x)(1)(2)'),
            mt_env,
            K1 = appArgK(parse('0'), mt_env, doneK()))
[3]  interp(parse('(fun (x): fun (y): fun (x): x)(1)'),
            mt_env,
            K2 = appArgK(parse('2'), mt_env, K1))
[4]  interp(parse('fun (x): fun (y): fun (x): x'),
            mt_env,
            K3 = appArgK(parse('1'), mt_env, K2))
[5]  continue(K3,
              V1 = closV(#'x, parse('fun (y): fun (x): x'), mt_env))
[6]  interp(parse('0'),
            mt_env,
            K4 = doAppK(V1, K2))
[7]  continue(K4,
              intV(0))
[8]  interp(parse('fun (y): fun (x): x'),
            E1 = extend_env(bind(#'x, intV(0)), mt_env),
            K2)
[9]  continue(K2,
              V2 = closV(#'y, parse('fun (x): x'), E1))
[10] interp(parse('2'),
            mt_env,
            K5 = doAppK(V2, K1))
[11] continue(K5,
              intV(2))
[12] interp(parse('fun (x): x'),
            E2 = extend_env(bind(#'y, intV(2)), E1),
            K1)
[13] continue(K1,
              V3 = closV(#'x, parse('x'), E2))
[14] interp(parse('1'),
            mt_env,
            K6 = doAppK(V3, doneK()))
[15] continue(K6,
              intV(1))
```

```
[16]  interp(parse('x'),
             extend_env(bind(#'x, intV(1)), E2),
             doneK())
[17]  continue(doneK(),
               intV(1))
```

17. 10 points

```
let f = (fun (x):
           fun (y):
             x(y)):
  f(fun (z):
      z)(1)
```

```
[1]  interp(parse('(fun (f): f(fun (z): z)(1))(fun (x): fun (y): x(y))'),
            mt_env,
            doneK())
[2]  interp(parse('fun (f): f(fun (z): z)(1)'),
            mt_env,
            K1 = appArgK(
                          parse('fun (x): fun (y): x(y)'),
                          mt_env,
                          doneK()
                        ))
[3]  continue(K1,
              V1 = closV(#'f, parse('f(fun (z): z)(1)'), mt_env))
[4]  interp(parse('fun (x): fun (y): x(y)'),
            mt_env,
            K2 = doAppK(V1, doneK()))
[5]  continue(K2,
              V2 = closV(#'x, parse('fun (y): x(y)'), mt_env))
[6]  interp(parse('f(fun (z): z)(1)'),
            E1 = extend_env(bind(#'f, V2), mt_env),
            doneK())
[7]  interp(parse('f(fun (z): z)'),
            E1,
            K3 = appArgK(parse('1'), E1, doneK()))
[8]  interp(parse('f'),
            E1,
            K4 = appArgK(parse('fun (z): z'), E1, K3))
[9]  continue(K4,
              V2)
[10] interp(parse('fun (z): z'),
            E1,
            K5 = doAppK(V2, K3))
[11] continue(K5,
              V3 = closV(#'z, parse('z'), E1))
[12] interp(parse('fun (y): x(y)'),
            E2 = extend_env(bind(#'x, V3), mt_env),
            K3)
[13] continue(K3,
              V4 = closV(#'y, parse('x(y)'), E2))
```

```
[14] interp(parse('1'),
            E1,
            K6 = doAppK(V4, doneK()))
[15] continue(K6,
              intV(1))
[16] interp(parse('x(y)'),
            E3 = extend_env(bind(#'y, intV(1)), E2),
            doneK())
[17] interp(parse('x'),
            E3,
            K7 = appArgK(parse('y'), E3, doneK()))
[18] continue(K7,
              V3)
[19] interp(parse('y'),
            E3,
            K8 = doAppK(V3, doneK()))
[20] continue(K8,
              intV(1))
[21] interp(parse('z'),
            extend_env(bind(#'z, intV(1)), E1),
            doneK())
[22] continue(doneK(),
              intV(1))
```

18. This question is too mean to be on an exam, but if you check every detail, you should be able to find a mistake. Hint: the number of the step that is wrong is part of the expression for question 6. 10 points

```
let f = (fun (x):
           -1 * x):
  f(10) + 8
```

```
[1]  interp(parse('(fun (f): f(10) + 8)(fun (x): -1 * x)'),
            mt_env,
            doneK())
[2]  interp(parse('fun (f): f(10) + 8'),
            mt_env,
            K1 = appArgK(parse('fun (x): -1 * x'), mt_env, doneK()))
[3]  continue(K1,
              V1 = closV(#'f, parse('f(10) + 8'), mt_env))
[4]  interp(parse('fun (x): -1 * x'),
            mt_env,
            K2 = doAppK(V1, doneK()))
[5]  continue(K2,
              V2 = closV(#'x, parse('-1 * x'), mt_env))
[6]  interp(parse('f(10) + 8'),
            E1 = extend_env(bind(#'f, V2), mt_env),
            doneK())
[7]  interp(parse('f(10)'),
            E1,
            K3 = plusSecondK(parse('8'), E1, doneK()))
[8]  interp(parse('f'),
            E1,
            K4 = appArgK(parse('10'), E1, K3))
[9]  continue(K4,
              V2)
[10] interp(parse('10'),
            E1,
            K5 = doAppK(V2, K3))
[11] continue(K5,
              intV(10))
[12] interp(parse('-1 * x'),
            E2 = extend_env(bind(#'x, intV(10)), E1),
            K3)
[13] interp(parse('-1'),
            E2,
            K6 = multSecondK(parse('x'), E2, K3))
[14] continue(K6,
              intV(-1))
```

```
[15] interp(parse('x'),
            E2,
            K7 = doMultK(intV(-1), K3))
[16] continue(K7,
              intV(10))
[17] continue(K3,
              intV(-10))
[18] interp(parse('8'),
            E1,
            K8 = doPlusK(intV(-10), doneK()))
[19] continue(K8,
              intV(8))
[20] continue(doneK(),
              intV(-2))
```

Answers

1. Eager: `3`; Lazy: `3`; **Same**

2. Eager: *error*; Lazy: `12`; **Different**

3. Eager: *function*; Lazy: *function*; **Same**

4. Eager: *error*; Lazy: `13`; **Different**

5. Eager: *error*; Lazy: `15`; **Different**

6. Eager: *error*; Lazy: *error*; **Same**

7. Register: `0`, To space: `2 3 8 1 6 0 3 0 0 0 0`

8. `intV(7)`

9. `intV(10)`

10. *error*, because 3 is not a function

11. `intV(-2)`

12. Step [4] should have a `1` instead of `3`: `interp(parse('1'), mt_env, K2)`.

13. Correct.

14. Correct.

15. The body expression `x + 1` should not be `interp`ed. Step [6] should be

```
interp(parse('f'),
       extend_env(bind(#'f, V2),
                  mt_env),
       done_k())
```

where `V2` is the value argument to `continue` in step [5].

16. Starting at step [6], the expressions/values `0` and `1` are backwards. The final answer should be `intV(0)`. Step [6] should be

```
interp(parse('1'),
       mt_env,
       doAppK(V1, K2))
```

17. Correct.

18. Step 12 should have `mt_env` in place of `E1`.