CMSC 15100 Midterm Exam 2

Zachary Sherman

TOTAL POINTS

87.5 / 100

QUESTION 1

1 p. 2: list->vector 7 / 9

✓ + 0 pts claimed

✓ + 2 pts The correct type is given: (All (a) (Listof a) -> (Vectorof a))

+ 2 pts The function does not crash when the input is an empty list.

✓ + 5 pts The implementation is correct, subject to deductions enumerated below.

+ 0 pts No response.

QUESTION 2

2 p. 3: short evaluations 8.5 / 10

✓ + 0 pts claimed

✓ + 1 pts Expression 1 cannot be evaluated. The second argument to build-list needs to be a function.

✓ + 1 pts Expression 2 evaluates to two concentric circles, or two concentric circles with a dot in the middle.

✓ - 0.5 pts Too many circles.

✓ + 1 pts Expression 3 evaluates to (list 5 17 37).

✓ + 1 pts Expression 4 evaluates to (list #f #f #t #f #f).

✓ + 1 pts Expression 5 cannot be evaluated. The outer map must consume a function as its first argument.

✓ + 1 pts Expression 6 evaluates to (list "ccc").

✓ + 1 pts Expression 7 cannot be evaluated. The first argument to foldr must consume two items; b is unbound and appears here out of nowhere.

+ 1 pts Expression 8 cannot be evaluated. Since the first argument to foldr, the aggregation operator, returns an integer, the second argument to foldr, the base case, must be an integer (and not a string).

✓ + 1 pts Expression 9 evaluates to 0.

✓ + 1 pts Expression 10 evaluates to -3.

QUESTION 3

3 p. 4: types 8 / 10

✓ + 0 pts claimed

✓ + 1 pts (: v : All (a) (Optional a) a -> a)

✓ + 1 pts The behavior v is correctly described, e.g., v unwraps an optional value and returns either the unwrapped value or a default.

+ 0.5 pts A better name for v is suggested, e.g., unwrap or get-value.

+ 0 pts Vague or misleading name for v.

✓ + 1 pts (: smap : All (a b) (a -> b) (Listof (Optional a)) -> (Listof b))

- 0.5 pts The list in smap is a (Listof (Optional a)).

- 0.5 pts Return type of smap is incorrect.

✓ + 1 pts The behavior of smap is correctly described, e.g., smap applied function f to the Some values in a list of optionals, skips the Nones.

+ 0.5 pts A better name for smap is suggested, e.g., some-map, map-some, map-values.

+ 0 pts Vague or misleading name for smap.

✓ + 1 pts (: mapp : All (a b) (a -> (Listof b)) (Listof a) -> (Listof b))

- 0.5 pts The functional argument to mapp produces a (Listof b).

✓ + 1 pts The behavior of mapp is correctly described, e.g., map appends the results of the application of the function f to all the items in the list.

+ 0.5 pts A better name for mapp is suggested, e.g., map-append, flat-map, map-flatten, map-list.

+ 0 pts Vague or misleading name for mapp.

✓ + 1 pts (: omap : All (a b) (a -> b) (Optional a) -> (Optional b))

- 0.5 pts The type of omap should include two type variables.

✓ + 1 pts The behavior of omap is correctly

described, e.g., omap applies f to Some value, leaves None alone.

+ **0.5 pts** A better name for omap is suggested, e.g., map-optional, opt-map, map-some.

+ **0 pts** Vague or misleading name for omap.

QUESTION 4

4 p.5: compose 9 / 9

+ **0 pts** claimed

✓ + **4 pts** **Compose has the correct type, subject to deductions enumerated below.**

- **1 pts** Compose has an extra (non-functional) argument in the type.

- **1 pts** The type of compose should have three type variables; this type is too restrictive.

- **3 pts** The type of compose should be polymorphic; this type includes no type variables.

✓ + **5 pts** **Compose has a correct implementation, subject to deductions enumerated below.**

- **3 pts** Compose does not return a function.

- **1 pts** Your function definition includes type-incorrect applications.

+ **0 pts** There is no response.

- **1 pts** The functions should be applied in the other order.

+ **0 pts** Incorrect response.

QUESTION 5

5 p. 6: BST sketches 8 / 8

✓ + **0 pts** **claimed**

✓ + **4 pts** **Most balanced tree is correct.**

- **2 pts** Not as balanced as possible

- **2 pts** Balanced tree is not a valid BST; elements are out of order.

✓ + **4 pts** **Most unbalanced tree is correct.**

- **2 pts** Not as unbalanced as possible

- **2 pts** Unbalanced tree is not a valid BST; elements are out of order.

+ **0 pts** No response.

QUESTION 6

6 p. 7: LogicalLoc order 9 / 9

✓ + **0 pts** **claimed**

✓ + **3 pts** **The order type is correct (LogicalLoc LogicalLoc -> Boolean), subject to deductions enumerated below.**

✓ + **3 pts** **The order definition is correct, subject to deductions enumerated below.**

- **2 pts** The order does not distinguish distinct items. (LogicalLoc 3 4) should not equal (LogicalLoc 4 3).

- **1 pts** The ordering doesn't maintain transitivity. If a < b and b < c, then it must be that a < c.

+ **0 pts** Incorrect response.

✓ + **3 pts** **There are three correctly-written tests, subject to deductions enumerated below.**

+ **0 pts** There is no response.

QUESTION 7

7 p. 8: LogicalLoc hash 9 / 9

✓ + **0 pts** **claimed**

✓ + **3 pts** **The type (LogicalLoc -> Integer) is correct.**

✓ + **3 pts** **The definition is correct and distinguishes between (1,2) and (2,1) and (1,11) and (11,1).**

✓ + **3 pts** **There are three correct tests.**

+ **0 pts** No response.

QUESTION 8

8 p. 9: Alarm data structure 9 / 9

✓ + **0 pts** **claimed**

✓ + **3 pts** **Can represent either a specific date or a weekly alarm.**

+ **1.5 pts** Can represent specific dates, but not weekly alarms.

✓ + **3 pts** **Can represent a time.**

✓ + **3 pts** **Can include an optional reminder.**

- **2 pts** Did not define an Alarm struct.

+ **0 pts** No response.

QUESTION 9

9 p. 10: pass board, count stones on board 6 / 8

✓ + **0 pts** **claimed**

✓ + **4 pts** **The implementation of pass is correct, subject to the deductions enumerated below.**

- **2 pts** Underscores in pass have the effect of discarding necessary information (and are not proper arguments to constructors).

+ **0 pts** No response for pass.

✓ + **4 pts** **The implementation of stones1 or stones2 is correct, subject to the deductions enumerated below.**

+ **0 pts** No response for stones1 or stones2.

- 2 Point adjustment

If stones have been captured, the length of the history will not necessarily equal the number of stones on the board. Clever idea, though.

QUESTION 10

10 p. 11: retro 8 / 10

+ **0 pts** claimed

✓ + **2 pts** **Correctly populates new structure with board dimension.**

✓ + **1 pts** **Correctly populates new structure with next player.**

✓ + **7 pts** **Correctly populates new structure with locations of black and white stones.**

✓ - **2 pts** **Does not properly separate black and white stones.**

+ **0 pts** No response/incomplete response.

+ **0 pts** Click here to replace this description.

- **1 pts** Board stores (Optional Stone) not Stone

+ **2.5 pts** This addresses the inverted problem of building a Go2 given a Go1. You were supposed to consume a Go2 and build a Go1.

You never test if the stone returned by bd-ref is the actually color you're working on

QUESTION 11

11 p. 12: thing 6 / 9

✓ + **0 pts** **claimed**

✓ + **9 pts** **The drawing is essentially correct, subject to deductions enumerated below.**

✓ - **2 pts** **The circles are aligned at the bottom rather than in the middle.**

- **1 pts** There are too many circles in the figure.

- **2 pts** There are too few circles in the figure.

- **3 pts** The figure is not symmetric around the center.

- **1 pts** There should not be whitespace around the objects.

+ **0 pts** No response.

- 1 Point adjustment

Pattern of the circles is partially incorrect.

Midterm Exam 2
CMSC 15100 Autumn 2018
Monday, December 3, 2018

Please write your name here:

Zach Sherman

We do not answer questions from students once the exam has begun. Please read the directions carefully and follow them as best you can. If you have trouble interpreting a question, you can write us a note about your interpretation of it on the test itself along with your response.

You may use the functions you write on this test anywhere on this test. You may not refer to functions you may have written at some earlier time (such as a homework exercise) without rewriting them here. Wherever you design your own helper function, write its purpose and type above its definition.

We will be scanning your exams and grading digital versions of them. Please do your best to write all responses in the given spaces. Write your initials on each page, and please try not to write too close to the margins. Material at the margins may not be successfully scanned. Having said that, all your exams will be read by actual people and we can consult the paper copies if we must.

Some common built-in operations and their types are as follows:

```
cons     : All (A) A (Listof A) -> (Listof A)

first    : All (A) (Listof A) -> A
rest     : All (A) (Listof A) -> (Listof A)

empty?   : All (A) (Listof A) -> Boolean

length   : All (A) (Listof A) -> Integer
reverse  : All (A) (Listof A) -> (Listof A)
```

map : All (a b) (Listof a) (a->b) ->(L b

foldr : All (a b) (Listof a) b (a->b)-> b

Throughout the exam, assume

```
(define-struct (Some a)
  ([value : a]))

(define-type (Optional a)
  (U 'None (Some a)))
```

These data definitions are common to project1 and project2.

```
(define-type Stone
  (U 'black 'white))

(define-struct LogicalLoc
  ([col : Integer]
   [row : Integer]))
```

ZS

There are various utility functions built in to Racket to convert between one kind of data structure and another. Write the type and definition of the function list->vector, whose name clearly says what it should do.

```
(: list->vector : All (a) (Listof a) -> (Vectorof a))
(define (list->vector as)
  (local {(define len (length as))
          (: lp : (Listof a) (Vectorof a) Integer -> (Vectorof a))
          (define (lp xs acc i)
            (match xs
              ['() acc]
              [(cons h t) (begin (vector-set! i h acc)
                                 (lp t acc (add1 i)))]))}
    (lp as (make-vector len (first as)) 0)))
```

may have confused order of args

position

value

vector to modify

length of new vector

value of each slot in vector

order of args may be wrong here

ZS

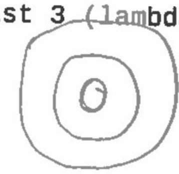
Evaluate each expression, or identify, in a few words, why it cannot be evaluated.

```
(foldr + 0 (build-list 10 20))
```

cannot evaluate → (: build-list : All (a) Integer (Natural -> a) -> (Listof a)

type-conflict with [20]
expected: (Natural->a) given: Inte

```
(foldr overlay
       empty-image
       (build-list 3 (lambda ([i : Integer]) (circle (* i 10) 'outline 'black))))
```



```
(map (lambda ([i : Integer]) (add1 (sqr i))) (list 2 4 6))
```

(list 5 17 37)

```
(map (lambda ([s : String]) (> (string-length s) 2))
     (list "a" "bb" "ccc" "dd" "e"))
```

(list #f #f #t #f #f)

```
(map (map add1 (list 1 2 3)) (list 2 3 4))
```

cannot evaluate → in map (outer one) first argument, expected: (All (a b) a->b)
given: (Listof a)

```
(filter (lambda ([s : String]) (> (string-length s) 2))
        (list "a" "bb" "ccc" "dd" "e"))
```

(list "ccc")

```
(foldr (lambda ([a : Integer]) (+ a b)) 0 (list 1 2 3))
```

cannot evaluate → never defined [b] / gave it a type

```
(foldr (lambda ([s : String] [t : Integer]) (+ t (string-length s)))
       ""
       (list "a" "b" "ccc"))
```

5

```
(foldr max 0 (list -1 -2 -3))
```

0

```
(foldr min 0 (list -1 -2 -3))
```

-3

ZS

Write the type of each function, and explain clearly, in a sentence or two, what each one does. Note that writing the types of these polymorphic functions entails inferring the names of the relevant type variables from the given code. Also, suggest, for each function, a properly descriptive name for it; the names given here are all terse to a fault. None of these functions is ill-typed or does not compile.

(: v : All (a) (Optional a) a —> a)

```
(define (v opt def)
  (match opt ['None def] [(Some x) x]))
```

takes in Optional a and a base definition,
if a has a value associated with it, returns the value,
otherwise (if a is 'None) returns base definition

```
(define (smap f opts)
  (foldr (lambda ([opt : (Optional a)] [ys : (Listof b)])
           (match opt ['None ys] [(Some x) (cons (f x) ys)]))
         '()
         opts))
```

(: smap : All (a b) (a -> b) (Listof (Optional a)) —> (Listof b))

smap takes in a list of optional alpha and an operation (type a->b)
and maps the operation onto all the values of the given list
that actually have a value (not 'None) and returns the resulting list of b,

```
(define (mapp f xs)
  (foldr (lambda ([x : a] [ys : (Listof b)]) (append (f x) ys)) '() xs))
```

(: mapp : All (a b) (a —> (Listof b)) (Listof a) —> (Listof b))

mapp takes in 1) an operation that converts values of type a into
a list of values of type b and 2) a list of values of type a
and returns the resulting appended lists of values of type b, once
the operation is mapped to the list of a

(: omap : All (a b) (a->b) (Optional a) -> (Optional b))

```
(define (omap f opt)
  (match opt ['None 'None] [(Some x) (Some (f x))]))
```

omap applies operation of type (a->b) to an optional a,
returning 'None if the given Optional a is 'None or Some b if
Optional a is Some value

ZS

The built-in compose function takes a variable number of functions and combines them together into a new function. We restrict this question to an implementation of compose that consumes exactly two functional arguments. It should return a function that behaves as the successive application of those two functions, with the function on the right's application first.

These examples show compose in use:

```
((compose sqr add1) 10) --> 121
((compose add1 sqr) 10) --> 101
((compose sqr sqr)   3) -->  81
```

Define compose for two arguments, writing its type and its definition. Do not write a purpose or tests.

```
(: compose : All (a b c) (b -> c) (a -> b) -> (a -> c))
(define (compose bc ab)
  (lambda ([x : a]) (bc (ab x))))
```

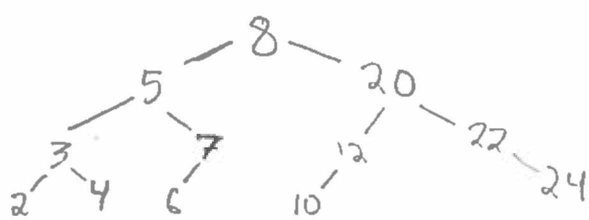
ZS

Consider this list of numbers:

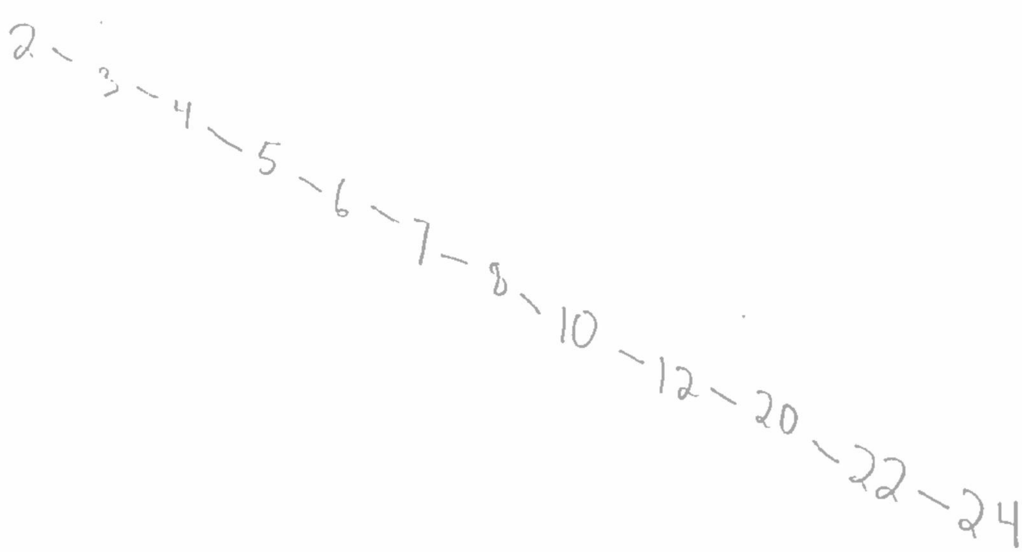
(list 3 5 7 2 4 6 8 10 12 20 22 24)

Draw two binary search trees containing these numbers. In the two drawings, the numbers can be in any legal BST arrangement. One should depict a binary search tree that is as well balanced as possible. The other should depict a binary search tree that is as poorly balanced as possible. Note there is not a single correct response to this question.

well-balanced



poorly-balanced



ZS

Generic BSTMaps were the subject of your most recent lab exercise. Assume that you need to build a BSTMap whose keys were LogicalLoc structs as defined in the Go projects. Write an ordering function that would enable you to use LogicalLocs as keys in a BSTMap. You can choose its name. You need not write a purpose for it, but you must write its type, its definition, and at least three check-expect tests.

```
(: ord-LL : LogicalLoc LogicalLoc -> Boolean)
(define (ord-LL a b)
  (match* (a b)
   [((LogicalLoc xa ya)
     (LogicalLoc xb yb))
    (cond
      [(> (+ xa ya) (+ xb yb)) #t]
      [(< (+ xa ya) (+ xb yb)) #f]
      [(> (xa xb)) #t]
      [else #f])]))

(check-expect (ord-LL (LogicalLoc 10 10)
                      (LogicalLoc 0 0)) #t) ✓

(check-expect (ord-LL (LogicalLoc 10 10)
                      (LogicalLoc 11 10)) #f) ✓

(check-expect (ord-LL (LogicalLoc 10 10)
                      (LogicalLoc 11 9)) #f) ✓

(check-expect (ord-LL (LogicalLoc 10 10)
                      (LogicalLoc 8 12)) #t) ✓
```

ZS

Along similar lines, write a hash function to enable use of LogicalLocs as keys in a hash map. It need not be an industrial-strength hash function, but it should be not obviously bad in the following specific way: it must compute distinct values for (LogicalLoc 1 2) and (LogicalLoc 2 1), and for (LogicalLoc 1 11) and (LogicalLoc 11 1). Write its type, its definition, and at least three check-expect tests.

```
(: hash-LL : LogicalLoc -> Integer)
(define (hash-LL LL)
  (match LL
    [(LogicalLoc x y) (abs (- (* 5 x) y))]))
```

this is absolute value right?

```
(check-expect (hash-LL (LogicalLoc 1 2)) 3)
(check-expect (hash-LL (LogicalLoc 2 1)) 9)
(check-expect (hash-LL (LogicalLoc 1 11)) 6)
                                  11 1)) 54)
```

ZS

Assume you are working on a team designing a Racket-driven alarm clock. Alarms are scheduled either to happen once at an exact date and time, or weekly on a particular day at some given time. Alarms need to be able to have events associated with them, in which case they can serve as specific reminders (such as "walk the dog" or "call Mom"). If they are associated with no such event, they are pure alarms which just blast a sound at the appointed time.

Define an Alarm data structure that enables these features. You need not write any code for this problem outside of data definitions, but if you refer to other data structures such as Date, etc., please include their data definitions here as well.

```
                  -> (Sunday)
(define-type Day (U 'U 'M 'T 'W 'R 'F 'S))

(define-struct Date
  ([d : Integer]
   [m : Integer]
   [y : Integer]))

(define-struct Time  -> 24-hr time (no AM/PM)
  ([sec : Integer]
   [min : Integer]
   [hour : Integer]))

(define-struct Alarm
  ([time : Time]   -> would write a function to check if valid Alarm (one of day or date present).
   [date : (Optional Date)] -> if 'None, day must not be 'None (repeats weekly)
   [msg : (Optional String)] -> if 'None, no message
   [day : (Optional Day)])) -> if 'None, doesn't repeat, else repeats every
                               week on given day
```

ZS

```
;; === Go from project1, renamed to Go1 (Stone and LogicalLoc are defined on p. 1)

(define-struct Go1
  ([dimension : Integer]
   [black-stones : (Listof LogicalLoc)]
   [white-stones : (Listof LogicalLoc)]
   [next-to-play : Stone]))

;; === Go from project2, renamed to Go2

(define-type Board
  (Vectorof (Vectorof (Optional Stone))))

(define-struct Go2
  ([board : Board]
   [next-to-play : Stone]
   [history : (Listof Board)]))
```

a) Implement a function to pass to the next player. You need only write the definition.

```
(: pass : Go1 -> Go1)
(define (pass go)
 (match go
  [(Go1 d bs ws 'black) (Go1 d bs ws 'white)]
  [(Go1 d bs ws _) (Go1 d bs ws 'black)]))
```

b) Write one of these two functions to count the number of stones in play. You need only write the definition.

```
(: stones1 : Go1 -> Integer)
(: stones2 : Go2 -> Integer)
```

```
(define (stones2 go)
 (match go
  [(Go2 _ _ hist) (sub1 (length hist))]
```

* assuming the programmer does not add duplicate boards to hist on passes and assuming hist includes blank board

25

c) Write a function "retro" to read a Go2 value and produce the closest possible corresponding Go1 value. In your implementation, you may assume the existence of a function with the following name and type:

```
(: bd-ref : Board Integer Integer -> (Optional Stone))
```

Any other helper functions in your response, you must define as part of your response. No purpose or tests are needed.

```
(: retro : Go2 -> Go1)
(define (retro go)
  (match go
    [(Go2 board next _)
     (Go1 (vector-length board)
          (retro-stones 'black board)
          (retro-stones 'white board)
          next)]))


(: retro-stones : Stone Board -> (Listof LogicalLoc))
(define (retro-stones s bd)
  (local {(define len (vector-length bd))
          (: lp : Integer Integer -> (Listof LogicalLoc))
          (define (lp x y)
            (cond
              [(= y len) '()]
              [(= x len) (lp 0 (add1 y))]
              [else (match* (bd-ref bd x y) s)
                      [('None _) (lp (add1 x) y)] -> technically don't need
                      [((Some st) st) (cons (LogicalLoc x y)
                                            (lp (add1 x) y))]

                      [(_ _) (lp (add1 x) y)])]))}

    (lp 0 0)))
```

```
(: thing : Integer -> Image)
(define (thing i)
  (if (<= i 1)
      (square 1 'solid 'black)
      (local
        {(define t (thing (quotient i 3)))}
        (beside t
                (circle (quotient i 3) 'outline 'black)
                t))))
```

Draw (thing 81).

ZS