CMSC 15100 Introduction to Computer Science I

The University of Chicago, Summer 2020

Midterm Examination

July 1, 2020

Please write your name here:

Yifei Li

Instructions:

- Submission deadline is July 10 11:59 pm CST. The exam is out of 56 points. Submit a word doc or pdf similar to HW1 not racket files.
- The exam is open book and open internet. Feel free to google anything
- For the problems that ask you to write Racket code, it's more important that your code has the right ideas than that it compiles. Try and use good coding style if possible (indentation and new lines, correct parens, meaningful variable names).
- You may freely use any functions seen in class. Write helper functions as you see fit.
- There are total 9 questions. The questions 7, 8 and 9 are extra credit
- You don't have to write any tests (`check-expect`, etc.) anywhere on the exam, nor do you have to `require` anything. You don't have to write any comments either, but feel free to add comments if you think your code would be easier to understand.

This box is for grading. Please continue to the next page.

Problem	Points	Score
1	10	
2	11	
3	9	
4	13	
5	9	
6	4	
7	5 (EC)	
8	7 (EC)	
9	7 (EC)	
Total	56	

1. (2 points each) Evaluate each of the given Racket expressions. You don't need to show any work – you can just write the final value. However, if your answer is incorrect, you may get partial credit if you show your work.

(a) `(- 10 15)`

−5

(b) `(or (< 3 3) (and (> 7 6) (string=? "hey" "yo")))`

(or #f (and #t #f))

(or #f #f) ⇒ #f

(c)
```
(+
 (if (> 5 6) 1 0)
 (if (not (= (/ 6 2) 3)) 2 0))
```

0

(d)
```
(cond
  [(empty? (list 0)) "raisins"]
  [(= (length (list 0)) 1) "grapes"]
  [else (error "out of fruit")])
```

"grapes"

(e) `( match (* 2 3) [1 5] [2 7] [n (+ (* n n) 1)])`

(* 2 3) → 6; (* n n) → 36; (+ 36 1) → 37

37

2. A musical melody is a sequence of notes and silences. These both have durations for how long they last in seconds, and notes furthermore have a pitch, which is a letter between A, B, C, ..., G (ignore octaves and sharps and flats, if you know what those are). For this problem you will design type definitions for a melody.

(a) **(5 points)** Give a type definition for a type `Melody`. You may give definitions for auxiliary types as you see fit.

```
(define-type Pitch (U 'A 'B 'C 'D 'E 'F 'G))
(define-struct Note
  ([pitch : Pitch]
   [duration : Integer]))
(define-type Slience Integer)
(define-type Component (U Note Slience))
(define-type Melody (Listof Component))
```

(b) **(3 points)** Define a variable `my-song` for a melody of your choice with at least three notes and one silence. Your melody doesn't have to sound pretty.

```
(: my-song : Melody)
(define my-song (list (Note 'C 2) 1 (Note 'D 1) (Note 'E 2)))
```

(c) **(3 points)** Write a function `melody-length` that takes a `Melody` and computes its total duration.

```
(: melody-length : Melody → Integer)
(define (melody-length melody)
  (local
    {
     (: get-length : Component → Integer)
     (define (get-length c)
  (cond
    [(Note? c) (Note-duration c)]
    [else c]))}
     (cond
    [(= 0 (length melody)) 0]
    [(+ (get-length (first melody)) (melody-length (rest melody)))])))
```

3. Here is a type representing an arrow direction, like on a keyboard:

```
(define-type Arrow (U 'up 'down 'left 'right))
```

Imagine there's a robot initially placed at coordinate (0, 0) of a grid. The robot receives a sequence of arrows, each of which makes it move one unit along the grid. For example, after receiving `'up 'right 'right`, the robot will now be at coordinate (2, 1). Your goal is to implement a function that computes the robot's final location. This location has type `Point`. For your reference, here is the type `Point` from lecture:

```
(define-struct Point
  ([x : Real]
   [y : Real]))
```

(a) **(5 points)** Make a helper function `move-once` that takes a `Point` representing the robot's current location and a single arrow and returns a `Point` with the robot's new location after following the arrow.

```
(: move-once : Arrow Point -> Point)
(define (move-once a p)
  (match a
    ['up (Point (Point-x p) (+ 1 (Point-y p)))]
    ['down (Point (Point-x p) (- (Point-y p) 1))]
    ['left (Point (- (Point-x p) 1) (Point-y p))]
    ['right (Point (+ (Point-x p) 1) (Point-y p))]))
```

(b) **(4 points)** Using the helper function `move-once`, make a function `move` that takes a list of arrow directions and outputs the robot's final location. You should assume that the robot starts at (0, 0).

```
(: move : (Listof Arrow) -> Point)
(define (move arrows)
  (cond
    [(= 0 (length arrows)) (Point 0 0)]
    [(move-once (first arrows) (move (rest arrows)))]))
```

4. A set of integers is a collection of some integers; for example, here are some sets,

$\{2, 7\}$ $\{-1, 10, 151, -99\}$ $\{0, 2, 4, 6, 8, 10, \dots\}$ $\{\dots, -3, -2, -1, 0, 1, 2, 3, \dots\}$

The second-to-last set is infinite, and the last set is the set of all the integers. We define the type

```
(define-type Int-Set (Integer -> Boolean))
```

We can view a set of integers as a function `Integer -> Boolean` where the function is true if the input is in the set. For example, the following function represents the set $\{2, 7\}$:

```
(: little-set : Int-Set)
(define (little-set n)
  (match n
    [2 #t]
    [7 #t]
    [_ #f]))
```

For the set of all integers $\{\dots, -3, -2, -1, 0, 1, 2, 3 \dots\}$, the corresponding `Int-Set` always returns true.

(a) **(4 points)** Create an `Int-Set` variable `negatives` for the set of negative integers, and another variable `less-than-100` for the integers that are between -100 and 100, inclusive.

```
(: negatives : Int-Set)
(define (negatives n)
  (cond
    [(< n 0) #t]
    [else #f]))

(: less-than-100 : Int-Set)
(define (less-than-100 n)
  (cond
    [(and (⩽ n 100) (⩾ n -100)) #t]
    [else #f]))
```

(b) **(4 points)** One interesting thing we can do with sets is complement them, which flips which elements are in and out of the set. For example, the complement of the even integers is the odd integers. Write the `complement` function.

```
(: complement : Int-Set -> Int-Set)
(define (complement set)
  (local
    {
     (: neg : Int-Set)
     (define (neg n)
       (not (set n)))}
    neg))
```

(c) **(5 points)** Using this representation of sets, if someone gives us an `Int-Set`, it's not even clear if that `Int-Set` has anything in it! Make a function `find-int` that outputs any `Integer` in the input set. If the input set doesn't have any elements, `find-int` should run forever.

```
(: find-int : Int-Set → Integer)
(define (find-int set)
  (local
    {
     (: n : Integer)
     (define n (random -2147483543 2147483544))}
     (if (set n) n (find-int set))))
```

5. (3 points each) Write the polymorphic type of the following functions:

`length`

```
(: length : (All (A) (Listof A) → Integer))
```

`foldl`

```
(: foldl : (All (A B) (A B → B) B (Listof A) → B))
```

`map`

```
(: map : (All (A B) (A → B) (Listof A) → (Listof B)))
```

6. (4 points) Consider the function $f(n)$ defined by the following recurrence

$$f(n) = f(n-1) + 2f(n-2). \tag{1}$$

We are also given the base cases $f(0) = 0$ and $f(1) = 2$. Write racket function (recursive) that takes a natural number n as input and outputs the value of $f(n)$.

```
(: func : Integer → Integer)
(define (func n)
  (match n
    [0 0]
    [1 2]
    [_ (+ (func (- n 1)) (* 2 (func (- n 2))))]))
```

7. (Extra Credit 5 points) Write a racket function takes a list of integers as input and outputs the number of even integers in the list. Feel free to write as many helper functions you want. For example if the input list is $[2, 7, 8, 3]$ the output should be 2.

```
(: count-even : (Listof Integer) → Integer)
(define (count-even l)
  (cond
    [(zero? (length l)) 0]
    [(= (modulo (first l) 2) 0) (+ 1 (count-even (rest l)))]
    [else (count-even (rest l))]))
```

8. (Extra Credit 7 points) Write a recursive implementation of the function `foldl`. Recall that `foldl` takes three inputs a function, an initial value and an list. When the list is empty the output should be just the initial value (this is the base case).

```
(: my-foldl : (All (A B) (A B → B) B (Listof A) → B))
(define (my-foldl func init list)
  (cond
    [(zero? (length list)) init]
    [else (my-foldl func (func (first list) init) (rest list))]))
```

9. (Extra Credit 7 points) A person is running up a staircase with n steps and can hop either 1 step, 2 steps, or 3 steps at a time. Write a racket function to count how many possible ways the person can run up the stairs. The input to the function is n the number of steps and output should be the number of ways he can run up the stairs

```
(: stairs : Integer → Integer)
(define (stairs n)
  (local
    {
     (: part : Integer Integer → Integer)
     (define (part n m)
       (cond
         [(= n m) (+ 1 (part n (- m 1)))]
         [(or (zero? m) (< n 0)) 0]
         [(zero? n) 1]
         [else (+ (part n (- m 1)) (part (- n m) m))]))}
    (part n 3)))
```