CSE 413 Midterm Exam

November 7, 2016

Name ___________________________________

The exam is closed book, closed notes, no electronic devices, signal flags, tin-can telephones, smoke signals, telepathy, tattoos, implants, or other signaling or communications apparatus.

Style and indenting matter, within limits. We're not overly picky about details like an extra or a missing parenthesis, but we do need to be able to follow your code and understand it.

If you have questions during the exam, raise your hand and someone will come to you. ***Don't*** leave your seat.

Please wait to turn the page until everyone has their exam and you have been told to begin.

Advice: The solutions to several of the problems are quite short. Don't be alarmed if there is a lot more room on the page than you actually need for your answer.

More gratuitous advice: Be sure to get to all the questions. If you find you are spending a lot of time on a question, move on and try other ones, then come back to the question that was taking the time.

1	/ 20
2	/ 18
3	/ 9
4	/ 9
5	/ 18
6	/ 16
7	/ 10
Total	/ 100

Question 1. (20 points) Suppose we have the following definitions in a Racket program:

```
(define a '(a (b c) d))
(define b (cons 'x (cadr a)))
(define c (list (cddr a) '(p q)))
```

(a) (14 points) Draw a diagram showing the combined results of evaluating these definitions together in the given order in a newly reset Racket environment.

(b) (6 points) What are the values displayed if `a`, `b`, and `c` are printed by Racket?

`a:`

`b:`

`c:`

Question 2. (18 points) (programming) Write a Racket function `sum` that returns the sum of all of the numbers in its argument, ignoring any non-numeric data that is encountered. The argument may have an arbitrary number of sub-lists, cons cells, and data with other, non-numeric types. Your solution does not need to be tail-recursive. You may define additional helper functions at top-level if you need them. Examples:

```
(sum '(1 2.5 3))  => 6.5
(sum '(1 (cons 2 3) "hello" (list "bye" '(false '()) ((4))))) => 10
(sum "thing") => 0
(sum '("no" nil ("numbers") "here" true)) => 0
```

Hint: Racket has several predicate functions you might find useful including `boolean? number? pair? list? null?`

```
(define (sum x)

   ;; write your code below













)
```

Question 3. (9 points, 3 each) Scope and bindings. Suppose we have the following definitions in a Racket program.

```
(define x 8)
(define y 2)

(define (f1 x)
  (let ([x (+ x y)]
        [y (* x y)])
    (cons x y)))

(define (f2 x)
  (let* ([x (+ x y)]
         [y (* x y)])
    (cons x y)))

(define (f3 x)
  (letrec ([x (+ x y)]
           [y (* x y)])
    (cons x y)))
```

For each of the following expressions, if the function call executes successfully, write down the value that is the result of the call. If evaluation fails because of some sort of error(s), give a brief description of the error(s). There are no syntactic errors (e.g., mismatched parentheses, etc.) in the code – if there are errors it is because of something that is incorrect during evaluation.

(a) `(f1 4)`

(b) `(f2 4)`

(c) `(f3 4)`

Question 4. (9 points, 3 each) As we've seen, sometimes there are several different ways to write a function that differ in the amount of execution space or time needed to compute the same value. For example, we saw that a tail-recursive version of factorial could compute *n*! in constant space compared to a simple recursive version that required space proportional to *O*(*n*).

Each of the following three functions computes the same result given the same argument value. For each function, decide whether a straight-forward implementation can compute the result in a constant amount of space or whether it requires space that varies depending on the argument value. Circle "yes" or "no" below each function definition to indicate your answer. (All three functions are proper Racket functions that can be executed.)

Hint: These functions can take a long time to execute, even for small argument values. Instead of doing an exhaustive trace, it may be more productive to look carefully at the structure of the code.

(a)

```
(define (mystery1 x)
  (let ([mod2 (modulo x 2)])
    (+ 1 (cond
           [(eq? x 1) 0]
           [(eq? mod2 0) (mystery1 (/ x 2))]
           [(eq? mod2 1) (mystery1 (+ (* x 5) 1))]))))
```

Constant space? Yes No

(b)

```
(define (mystery2 x)
  (let ([mod2 (modulo x 2)])
    (cond
      [(eq? x 1) 1]
      [(eq? mod2 0) (+ 1 (mystery2 (/ x 2)))]
      [(eq? mod2 1) (+ 1 (mystery2 (+ (* x 5) 1)))])))
```

Constant space? Yes No

(c)

```
(define (mystery3 x)
  (letrec ([fun (lambda (x arg)
                  (let ([mod2 (modulo x 2)])
                    (cond
                      [(eq? x 1) arg]
                      [(eq? mod2 0) (fun (/ x 2) (+ 1 arg))]
                      [(eq? mod2 1) (fun (+ (* x 5) 1)
                                         (+ 1 arg))])))])
    (fun x 1)))
```

Constant space? Yes No

Question 5. (18 points) Streams. Recall that we can implement a stream in Racket as a thunk (a 0-argument function) that, when called, returns a pair whose `car` is the current item in the stream and whose `cdr` is a stream (thunk) that will return the next element of the stream when used appropriately.

Write a Racket function `pair-stream` whose argument is a stream `s` and whose result is a stream of pairs from the original stream `s`. For example, suppose the original stream is the `nats` stream from lecture that produces the stream of values 1, 2, 3, 4, 5, 6, … . The result of `(pair-stream nats)` should be a stream whose values are the pairs (1 . 2), (2 . 3), (3 . 4), etc. In other words the *i*th value in the stream produced by `(pair-stream s)` is the `cons` pair (*x* . *y*) where *x* is element *i* from the original stream `s` and *y* is element *i*+1. Of course, the stream function itself will return a pair that contains this pair value (the data) plus a thunk to produce the next value-thunk pair.

```
(define (pair-stream s)
    ;; write your solution here

)
```

Question 6. (16 points) Function composition and pictures! The following function has two functions f and g as parameters and returns a new function that applies g to its argument and then applies f to the result of g (i.e., it composes the two functions f and g into a single function):

```
(define (compose f g)
  (lambda (x) (f (g x))))
```

Now suppose we have the following additional definitions and code in our program:

```
(define y 1)
(define fun
  (let ([b 3])(compose (lambda (a) (* a b))
                       (lambda (x) (+ x y)) )))
(fun 2)
```

(a) (14 points) Draw a diagram showing the environments, bindings, and closures that exist when x has been bound to 2 at the beginning of evaluating (fun 2), i.e., right before evaluating the body of the closure expression that is bound to fun. Then answer part (b) below. You need to show the bindings in the global environment that are used in this code, but you do not need to show any additional standard functions or other global values.

(b) (2 points) What value is produced when we evaluate (fun 2)?

Question 7. (10 points) Compose in MUPL. Recall from the previous question that one way to write function composition in Racket is as follows:

```
(define (compose f g)
  (lambda (x) (f (g x))))
```

For this question, write an equivalent MUPL function (i.e., use the Racket `struct` definitions from the MUPL assignment to write a MUPL function that is equivalent to the Racket `compose`). Complete the following Racket `define` so that `mupl-compose` is bound to an appropriate MUPL data structure that represents the MUPL version of `compose`. The main change you will need to make is to curry the function, since the Racket `compose` had two arguments, while MUPL only supports single-argument functions. (Hint: the sample solution is 4 lines and is not particularly tricky)

```
(define mupl-compose
   ;; write your code below
















)
```

For reference, here are the structures defined in the original MUPL code (most of which you probably won't need). The `#:transparent` directives have been omitted to save space, but that does not change the meaning or use of the struct data types.

```
(struct var  (string))  ;; a variable, e.g., (var "foo")
(struct int  (num)   )  ;; a constant number, e.g., (int 17)
(struct add  (e1 e2) )  ;; add two expressions
(struct isgreater (e1 e2)) ;; evaluate to 1 if e1>e2 else 0
(struct ifnz (e1 e2 e3)  ) ;; if e1 is not 0 then e2 else e3
(struct fun  (nameopt formal body) ) ;; a recursive(?) 1-argument function
(struct call (funexp actual)       ) ;; function call
(struct mlet (var e body) ) ;; a local binding (let var = e in body)
(struct apair (e1 e2)     ) ;; make a new pair
(struct first (e)    ) ;; get first part of a pair
(struct second(e)    ) ;; get second part of a pair
(struct munit ()    ) ;; unit value -- good for ending a list
(struct ismunit (e) ) ;; evaluate to 1 if e is munit else 0
```