

# CSE 413 Midterm Exam

---

Nov. 5, 2012

Name \_\_\_\_\_

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

Please place your UW ID on the desk next to you where it can be checked without causing a disruption.

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	/ 20
3	/ 20
4	/ 20
5	/ 20
Total	/ 100

**Question 1.** (20 points) (programming warmup) Write a Racket function `sumof` that computes the sum of the integers in its list argument. The elements of the list can be simple integers, or nested lists that contain integers or other nested lists. Examples:

```
(sumof '(1 2 3)) => 6
(sumof '(1 (2 (3 4) 5))) => 15
(sumof '((((42)))))) => 42
```

Your solution does not need to be tail-recursive. You may define additional helper functions at top-level if you really need them.

You should assume that the only elements of the lists are integers or nested lists. You do not need to deal with other data types.

Hint: Racket has functions to test types of expressions including `number?` `pair?`

**Question 2.** (20 points) Write a tail recursive function `(minpower x target)` that, given two positive integers `x` and `target`, returns the smallest power of `x` that is at least as large as `target`. i.e., the result is `x` or `x2` or ..., or `xn` such that `xn >= target` and no smaller power of `x` is at least that large. Examples:

```
(minpower 2 12) => 16 (since 23 = 8 and 24 = 16, which is the smallest power >= 12)
(minpower 3 9) => 9 (since 32 = 9)
```

You may define any auxiliary functions you need at top level – they don't need to be nested inside `minpower` using `letrec` or anything similar (although you can do that if you want). For full credit your function must be properly tail recursive.

**Question 3.** (20 points) Pictures! Consider the following Racket definitions entered at the top-level of the Racket interpreter.

```
(define g (lambda (f) (lambda (x) (f (f x)))))  
(define p (lambda (x) (+ x x)))  
(define h (g p))
```

(a) Draw a diagram showing the environments, bindings, and closures created by the above definitions. Then answer part (b) below.

(b) What is the value of `(h 3)` if we evaluate it after evaluating the above definitions?

**Question 4.** (20 points) In homework 5 we implemented a memoized version of the `(comb n k)` function, and we saw a memoized version of the Fibonacci function in lecture. For this problem, implement a memoized version of factorial. `(fact n)` should return  $n!$  ( $= 1 * 2 * 3 * \dots * n$ ), except that it should retain previously computed values and reuse those values to avoid recomputing previously calculated answers.

For this problem it's fine to store the values in an association list. An association list is a list of pairs. The function `assoc` can be used to retrieve values. For example, if `lst` is `(cons (cons 1 2) (cons (cons 3 4) null))`, then `(assoc 1 lst)` is `(1 . 2)` and `(assoc 17 lst)` is `#f`.

Complete the following code to implement the memoized factorial function. Your code may not define any additional top-level functions.

```
(define fact
  (letrec((memo null)
          (f (lambda (x)
```

```
              ))
    f))
```

**Question 5.** (20 points) We'd like to add a feature to MUPL to produce E-MUPL (enhanced MUPL). The new feature is an operation to test whether a MUPL expression evaluates to a MUPL int.

(DON'T PANIC!!! The answer is considerably shorter than the question!)

Here is the specification for the new MUPL `isint` expression:

- If  $e$  is a MUPL expression, then `(isint e)` is a MUPL expression. The value of `(isint e)` is either the MUPL value `(int 1)` if evaluating  $e$  produces a MUPL int, or is the MUPL value `(int 0)` if  $e$  is some other kind of MUPL expression (a pair, closure, etc.). In other words, it is similar to the existing `isaunit` function that evaluates to 1 or 0 depending on whether its operand is a MUPL aunit.

On the next page, write the code needed to add this new expression to the MUPL interpreter `eval-prog` function.

You should assume that the following structure has been added to MUPL to represent this expression:

```
(struct isint (e)) ;; evaluate to 1 if e is an int else 0
```

(the `#:transparent` directives have been omitted from the struct declarations in this problem to save space, but that does not change the meaning or use of the struct data types.)

For reference, here are the other structures defined in the original MUPL code (most of which you probably won't need):

```
(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 ifgreater (e1 e2 e3 e4) ) ;; if e1 > e2 then e3 else e4
(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 fst (e) ) ;; get first part of a pair
(struct snd (e) ) ;; get second part of a pair
(struct aunit () ) ;; unit value -- good for ending a list
(struct isaunit (e)) ;; evaluate to 1 if e is aunit else 0
```

```
;; a closure is not in "source" programs; it's what functions evaluate to
(struct closure (env fun))
```

Reminder: the Racket function `(error "message")` can be used to terminate evaluation with the given message.

Write your code on the next page. (You can tear this page out of the exam for reference if that is convenient.)

**Question 5. (cont.)** Write your code to implement the new MUPL `isint` expression below.

```
(struct isint (e)) ;; evaluate to 1 if e is an int else 0

(define (eval-prog p)
  (letrec
    ((f (lambda (env p)
          (cond (...
                ((isint? p) ;; write your code for isint below
                 (

```

)
)
...
;; remainder of eval-prog omitted