CSE 413 Midterm Exam

November 3, 2014

Name _____

The exam is closed book, closed notes, no electronic devices, signal flags, tin-can telephones, smoke signals, telepathy, tattoos, 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	/ 15
2	/ 15
3	/ 18
4	/ 18
5	/ 18
6	/ 16
Total	/ 100

Question 1. (15 points) (programming warmup) Write a Racket function `clone` that returns an exact duplicate of its argument. Examples:

```
(clone '()) => '()
(clone 'thing) => 'thing
(clone '(1 foo 3)) => '(1 foo 3)
(clone '(1 (2 (iii 4) (((5))))) => '(1 (2 (iii 4) (((5)))))
(clone '(there (are "no" (numbers)) here) =>
                                '(there (are "no" (numbers)) here)
```

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

Question 2. (15 points) Many Racket programs (including MUPL) use *association lists* as data structures to store key-value pairs, much like a hash table or dictionary in other languages. An association list is a list whose elements are key-value pairs formed with `cons`. For instance, a list that stores the key-value pairs `apple,1` and `cow,2` is `(cons (cons 'apple 1) (cons (cons 'cow 2) '())` and would print as `'((apple . 1) (cow . 2))`.

For this problem, write a function `(insert key value assoc)` that returns a list where the pair `(key . value)` has been added to the list `assoc`. If `key` duplicates a key that already appears in `assoc`, the old `(key . value)` pair should be replaced in the new list by the new pair. You should use `equal?` to test whether two key values are the same. You may assume that there are no duplicate key values in the original list. Since an association list is not ordered, the ordering of key, value pairs in the resulting list is not specified, in case that matters for your solution.

Question 3. (18 points) (chasing our tail) Write a tail-recursive, self-contained function `(avg lst)` that computes the average value of a list of numbers `lst`. The average should be computed using `(/ sum nitems)`, where `sum` is the sum of the items in the list and `nitems` is the number of items in the list. Don't worry about whether this returns an integer, rational, or floating-point value. That will depend on the numbers in the list and you do not need to do anything to handle different types of numbers in any special way.

Examples:
```
(avg '(1 2 3))    => 2
(avg '(1 2 4))    => 2 1/3
(avg '(1 2.0 4)) => 2.3333333
```

Simplifying assumptions: You may assume that the function argument is a simple list of numbers with no non-numeric data or nested lists, and that there is at least one number in the list. You do not need to check for errors or unexpected items in the list.

Complications: For full credit your solution must be properly *tail-recursive* and must *not* define any other functions or other values besides `avg` in the top-level global scope. However, you may, of course, include any local function definitions or other bindings inside the scope of the `avg` function itself.

Question 4. (18 points) Pictures! Suppose we execute the following code at the top level of a Racket interpreter:

```
(define puzzle
  (lambda (x)
    (let ((w (cons x x)))
      (lambda (p)
        (cons p w)))))

(define alist '(a b))
(define p (puzzle alist))
```

(a) (12 points) Draw a diagram showing the environments, bindings, and closures created by the above definitions. Then answer part (b) below.

(b) (6 points) What is the printed value of `(p 'x)` if we evaluate that expression after the above definitions have been made?

Question 5. (18 points) Recall that we can implement a stream in Racket as a thunk (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.

(a) (12 points) Define a stream `squares` that produces the sequence 2, 4, 16, 256, 65536, … . In other words each element of the stream is the square of the previous element.

(b) (6 points) Write a Racket expression that produces the second element of the stream `squares` (i.e., 4) when it is evaluated. Your answer must, of course, include appropriate operations involving `squares` and functions like `car` and `cdr` to produce this element – it is not sufficient to just write 4.

Question 6. (16 points) MUPL we must. It's time to add a new operation to MUPL. (DON'T PANIC!!! The answer is considerably shorter than the question!) The operation we want to add is a `swap` function that interchanges the two elements of a MUPL pair. The specification is:

- If *e* is a MUPL expression, then `(swap e)` is a MUPL expression. If `e` is not a MUPL pair `(apair?)` then it is an error. Otherwise if `e` is the pair `(apair e1 e2)`, the value of `(swap e)` is the pair `(apair e2 e1)`.

On the next page, write the code needed to add this new expression to the MUPL interpreter `eval-under-env` function. Hint: the `swap` operation should do no more and no less evaluation of its argument than is done by the `fst` and `snd` operations that extract components of a MUPL pair.

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

```
(struct swap (e) #:transparent) ;; swap two parts of a pair
```

For reference, here are the other structures defined in the original MUPL code (most of which you probably won't need). The `#:transparent` directives have been omitted from the struct declarations 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 fst  (e)    ) ;; get first part of a pair
(struct snd  (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 unit else 0


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

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 6. (cont.) Write your code to implement the new MUPL `swap` expression below.

```
(struct swap (e) #:transparent) ;; swap two parts of a pair

(define (eval-under-env e env)
  (cond [(var? e)
         (envlookup env (var-string e))]
        ;; remaining cases omitted...

        ;; CHANGE add your code for swap here









        [#t (error (format "bad MUPL expression: ~v" e))]))

;; Do NOT change
(define (eval-exp e)
  (eval-under-env e null))
```