CSE 413 Midterm Exam

November 3, 2008

Name _____Sample Solution_____

The exam is closed book, except that you may have the Scheme language definition and a single page of hand-written notes for reference.

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 many 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	/ 12
3	/ 17
4	/ 17
5	/ 17
6	/ 17
7	/ 5
Total	/ 100

Question 1. (15 points) For each of the following, what value is printed?

(a)
```
(define x 10)
(define y 5)
(let ((x 3)
      (y (+ x 1)))
  (+ x y))
```

14

(b)
```
(map odd?  '(2 17 5 6))
```

(#f #t #t #f)

(c)
```
(map car  '(((1 2) 3) (4 5) (((6)))))
```

((1 2) 4 ((6)))

Question 2. (12 points) Suppose we enter the following function definitions at the top level of a Scheme interpreter.

```
(define gadget
  (lambda (g)
    (lambda (x)
      (g (g x)))))

;; return n squared
(define square (lambda (n) (* n n)))
```

(a) What value do we get if we evaluate the following?

```
((gadget square) 2)
```

16

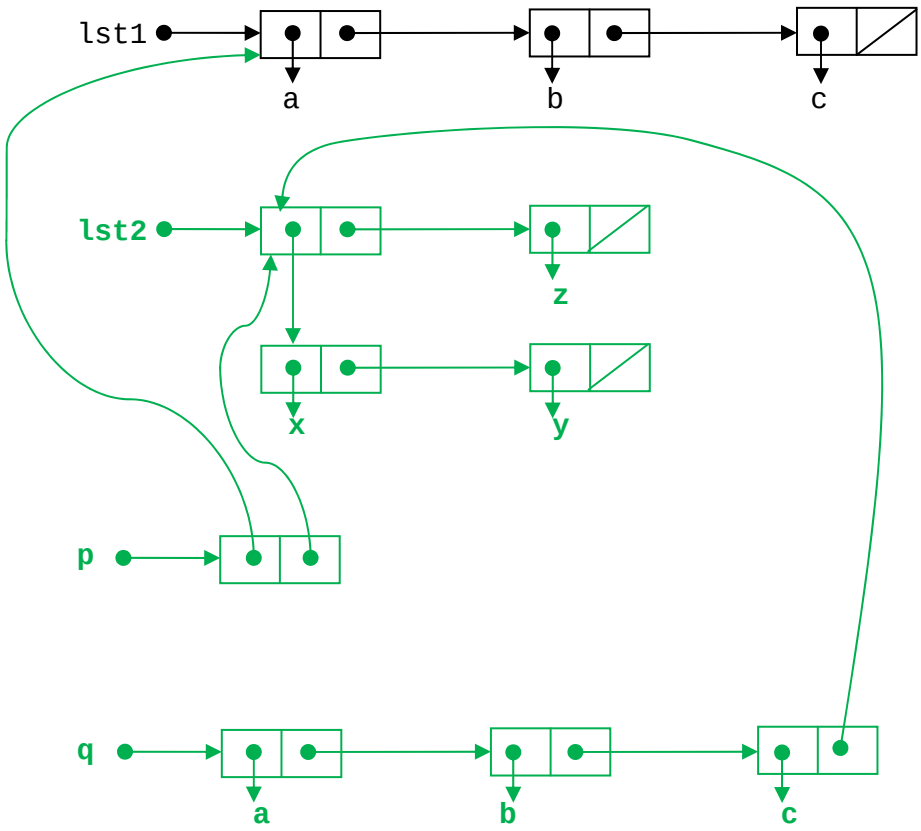
(b) Describe the result returned by evaluating `(gadget foo)`. What value is returned when function `gadget` is applied to its argument? (This question is about the value of `(gadget foo)`, independent of how that value is later used.) If the value returned is a function closure, be sure you describe the function as well as the environment bindings (which names are bound to what) in the closure. Keep your answer brief, if possible.

Evaluating `(gadget foo)` returns a closure. The code part of the closure is the function `(lambda (x) (g (g x)))`, and the environment binds `g` to `foo`.

Question 3. (17 points) Suppose we have the following definitions in a Scheme program

```
(define lst1 '(a b c))
(define lst2 '((x y) z))
(define p (cons lst1 lst2))
(define q (append lst1 lst2))
```

(a) Draw a diagram showing the result of evaluating these definitions as a single group (i.e., `lst1` and `lst2` should appear once, and p and q should reference these values as needed – don't draw `lst1` and `lst2` a second time). `lst1` is drawn for you.



(b) What are the values of p and q when these are printed by Scheme?

p **((a b c) (x y) z)**

q **(a b c (x y) z)**

Question 4. (17 points) For this problem, write a Scheme function `(merge lst1 lst2)` that has as input two lists of integers sorted in ascending order and produces a single sorted list containing all of the numbers from the input lists. For example, evaluation of

```
(merge  '(-7 1 5 12 19 413)  '(2 3 10 12 15))
```

should produce

```
(-7 1 2 3 5 10 12 12 15 19 413)
```

You should assume that the arguments are proper lists containing integers only, and that the values are correctly sorted.

One possible solution:

```
(define (merge lst1 lst2)
  (cond ((null? lst1) lst2)
        ((null? lst2) lst1)
        ((< (car lst1) (car lst2))
              (cons (car lst1) (merge (cdr lst1) lst2)))
        (else (cons (car lst2) (merge lst1 (cdr lst2))))))
```

Question 5. (17 points) Write a tail-recursive Scheme function `(power x n)` that computes the value x^n, i.e., x multiplied by itself n times. You should assume that the exponent n is a non-negative integer value and, as usual, if n is 0, then `(power x n)` should evaluate to 1.
For full credit,

- Your implementation must be properly tail-recursive, and
- You may not define any additional functions or variables at top-level (i.e., `define`). You are, of course, free to create any additional functions or variables you wish nested inside the scope of `power`, and
- You should calculate the result directly with appropriate arithmetic operations; you may not use the Scheme library function `expt` or anything similar.

One possible solution:

```
(define (power x n)
  (letrec ((aux (lambda (x n acc) ;; return acc * x^n
                  (if (< n 1)
                      acc
                      (aux x (- n 1) (* x acc))))))
    (aux x n 1)))
```

Question 6. (17 points) One of the advantages of an interpreter is that it is easy to add and change language features. For this question we would like to extend MUPL to add a new kind of conditional expression, in addition to the "ifgreater" one that is already included in the language.
(DON'T PANIC!!! The answer is considerably shorter than the question!)

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

- If e_1, e_2, and e_3 are MUPL expressions, then `(make-m-if` e_1 e_2 e_3`)` is a MUPL expression: a conditional evaluating to e_2 if e_1 is "true", otherwise to e_3. The expression e_1 is considered to be "false" if it is the MUPL unit value `(make-m-unit)`. Any other MUPL expression of any type is considered to be "true".

On the next page, write the code to add this new expression to the MUPL `eval-prog` function. Your implementation should evaluate the condition e_1 *only once*, and should evaluate either e_2 or e_3, but *not both*, depending on the value of e_1.

You should assume that the following structure has been added to MUPL to represent these conditional expressions:

```
(define-struct m-if (e1 e2 e3))  ;; if e1 is anything other than
                                 ;;   m-unit then e2 else e3
```

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

```
(define-struct var    (string)) ;; a variable, e.g., (make-var "foo")
(define-struct int    (num))    ;; a number, e.g., (make-int 17)
(define-struct add    (e1 e2))  ;; add two expressions
(define-struct ifgreater (e1 e2 e3 e4));;if e1 > e2 then e3 else e4
(define-struct fun    (nameopt formal body))
                           ;; a recursive(?) 1-argument function
(define-struct app    (funexp actual)) ;; function application
(define-struct m-pair (e1 e2))  ;; make a new pair
(define-struct fst    (e))      ;; get first part of a pair
(define-struct snd    (e))      ;; get second part of a pair
(define-struct m-unit ())       ;; unit
(define-struct is-m-unit (e))   ;; evaluate to 1 if e is unit else 0

(define-struct closure (env fun))
```

Write your code on the next page…

(You can also tear this page out of the exam if it saves you time by avoiding page flipping.)

Question 6. (cont.) Write your code to implement the new MUPL conditional expression in the space provided.

```
(define-struct m-if (e1 e2 e3))  ;; new conditional expression

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

                      (let ((v1 (f env (m-if-e1 p))))
                         (if (m-unit? v1)
                             (f env (m-if-e3 p))
                             (f env (m-if-e2 p))))

                    )
             )
           ...
```

Notes: Because the value of `e1` is used only once , it could have been evaluated in the `(m-unit? …)` expression directly without using a `let` binding. It was separated out in this sample solution only because it seems a little easier to read that way.

Several solutions tried to use MUPL's `is-m-unit` to decide whether `e1` was a MUPL unit value. That is a problem because it is confusing code from MUPL abstract programs with the Scheme code in the interpreter that examines MUPL code and carries out the computation described by it. While it may be possible to use `is-m-unit` somehow to decide whether `e1` is a MUPL unit, it is not the appropriate way to evaluate `m-if?`.

Question 7. (5 points) In class we surveyed some of the concepts behind garbage collection. Most modern garbage collectors are generational garbage collectors. What is the main idea behind this kind of collector and why is it effective? Keep your answer brief – it should only take a couple of sentences.

The key idea is that most functional and object-oriented programs tend to allocate a large number of objects with very short lifetimes. If new allocations are done from a small portion of the heap, and that portion is garbage-collected frequently, then these collections will be very effective in recovering a large portion of the unreachable objects in the heap. It is still necessary to perform a collection on the rest of the heap occasionally, but these larger collections will not have nearly as high a yield of new free space for the effort expended.