Quiz 6b

1. (2 + 3 + 1 points)
   1. Louis Reasoner looked at the Scheme1 interpreter, and was shocked to see that we use a massive cond clause in order to figure out what type of expression the user typed in. He thinks that instead we should use data-directed programming. He wants each expression to know how to evaluate itself. When he gets an expression, he will look up the corresponding “evaluation procedure” in the get/put table, and then execute it. Thus, eval-1 would be:

(define (eval-1 exp)

((get exp ‘evaluate)))

Explain why this cannot be done.

The problem is that there are infinitely many different possibilities for exp, for example, every single number is a possible exp. We cannot put in procedures into the get/put table to deal with all of these infinite cases.

2 points – use best judgment. Mentioning that there are too many cases gets at least 1 point.

* 1. Louis Reasoner was actually right, we can (sort of) use data-directed programming. We will assume that we have a typeof function that given an expression, returns a word that represents the type of the expression:

> (typeof ‘(lambda (x) (+ x 2)))

lambda

> (typeof ‘(+ 2 3))

call

> (typeof ‘x)

symbol

Show how we could implement eval-1 using data-directed programming. You should also show the calls to put corresponding to lambda and call, but you do **not** need to show the other calls to put. Apply-1 will not change.

(put ‘lambda ‘eval (lambda (exp) exp))

(put ‘call ‘eval (lambda (exp) (apply-1 (eval-1 (car exp))

(map eval-1 (cdr exp)))))

(define (eval-1 exp)

((get (typeof exp) ‘eval) exp))

1 point for correct arguments to get in eval-1

1 point for using the return value of get correctly in eval-1

1 points for the calls to put.

* 1. Fill in the blank. Make sure your answer is specific to Scheme-1 (as opposed to something that is true for any data-directed program).

When we use data-directed programming in Scheme-1, we don’t have to modify existing code when we add new special forms.

1. (1+3 points) We’re going to write a program for the game Snakes and Ladders! Well, almost. In Snakes and Ladders, we have a board with 100 different cells, each with their own unique position. A cell can be a “snake”, a “ladder”, or a “normal” cell. We’ll make it more interesting by adding a “teleport” cell.

We have a position ADT which represents a position on the board. You shouldn’t need to know its constructors or selectors for this problem.

We also have tagged versions for all of the cells:

;; pos is the position of the cell, nextpos is the next position that we go to (for snake/ladder).

(define (make-normal pos) (attach-tag ‘normal pos))

(define (make-snake-or-ladder-cell pos nextpos)

(attach-tag ‘movement (cons pos nextpos)))

(define (make-teleport pos) (attach-tag ‘teleport pos))

We now want to write a procedure find-next-pos that takes as input a tagged cell that a piece moved to, and returns the new position that the piece is on. If the cell is a normal cell, it returns the position of that cell. If the cell is a movement cell (snake or ladder), it returns the next position (the cdr of the *untagged* cell). If the cell is a teleport cell, we return a random position. Conveniently for us, (get-random-position) will give us a random legal position.

* 1. In the future, we’ll probably add even more types of cells. Knowing this, what style of programming should we use? (No explanation necessary.)
     1. Conventional
     2. Data-directed, using the get/put table
     3. Message passing

1 point for either ii or iii, or both. 0 for anything else.

* 1. Implement find-next-pos using the style of programming you chose in part a. If you use the get/put table, make sure you show all calls to put. (Remember to use type-tag and contents!)

Answer if they said data-directed for a):

(put ‘normal ‘pos (lambda (cell) cell))

(put ‘movement ‘pos (lambda (cell) (cdr cell)))

(put ‘teleport ‘pos (lambda (cell) (get-random-position)))

(define (find-next-pos cell)

((get (type-tag cell) ‘pos) (contents cell)))

Of course, there are different answers depending on their answer to part a). If the answer to b) doesn’t match the style they answered for a), -1 point.

Note that if they said “conventional” in a), they lose that point, but could still get full credit for b).

Don’t take off points if they had only 2 arguments to put and 1 argument to get (as long as the uses were consistent).

Use your judgment for assigning partial credit.

Reference: Relevant Scheme-1 code:

(define (scheme-1)

(display "Scheme-1: ")

(flush)

(print (eval-1 (read)))

(scheme-1))

(define (eval-1 exp)

(cond ((constant? exp) exp)

((symbol? exp) (eval exp)) ; use underlying Scheme's EVAL

((quote-exp? exp) (cadr exp))

((if-exp? exp)

(if (eval-1 (cadr exp))

(eval-1 (caddr exp))

(eval-1 (cadddr exp))))

((lambda-exp? exp) exp)

((pair? exp) (apply-1 (eval-1 (car exp)) ; eval the operator

(map eval-1 (cdr exp))))

(else (error "bad expr: " exp))))

(define (apply-1 proc args)

(cond ((procedure? proc) ; use underlying Scheme's APPLY

(apply proc args))

((lambda-exp? proc)

(eval-1 (substitute (caddr proc) ; the body

(cadr proc) ; the formal parameters

args ; the actual arguments

'()))) ; bound-vars, see below

(else (error "bad proc: " proc))))