CS61A Notes – Disc 6: Scheme1, Data Directed Programming (solutions)

You Are Scheme – and don't let anyone tell you otherwise

QUESTIONS

1. **Try it yourself: trace each call to `eval-1` and `apply-1` for the expression `(+ 3 (- 8 5))`.**

```
eval-1 with exp = (+ 3 (- 8 5))
   eval-1 with exp = +; returns #[closure ... 0]
   eval-1 with exp = 3; returns 3
   eval-1 with exp = (- 8 5)
      eval-1 with exp = -; returns #[closure ... 1]
      eval-1 with exp = 8; returns 8
      eval-1 with exp = 5; returns 5
      apply-1 with proc = #[closure ... 1], args = (8 5); returns 3
   eval-1 returns 3
   apply-1 with proc = #[closure ... 0], args = (3 3); returns 6
eval-1 returns 6
```

2. **How about for `(+ 3 ((lambda (x) (- x 1)) 5))`?**

```
eval-1 with exp = (+ 3 ((lambda (x) (- x 1)) 5))
   eval-1 with exp = +; returns #[closure ... 0]
   eval-1 with exp = 3; returns 3
   eval-1 with exp = ((lambda (x) (- x 1)) 5)
      eval-1 with exp = (lambda (x) (- x 1)); returns (lambda (x) (- x 1))
      eval-1 with exp = 5; returns 5
      apply-1 with proc = (lambda (x) (- x 1)), args = (5)
         eval-1 with exp = (- 5 1)
            eval-1 with exp = -; returns #[closure ... 1]
            eval-1 with exp = 5; returns 5
            eval-1 with exp = 1; returns 1
            apply-1 with proc = #[closure ... 1], args = (5 1); returns 4
         eval-1 returns 4
      apply-1 returns 4
   eval-1 returns 4
   apply-1 with proc = #[closure ... 0], args = (3 4); returns 7
eval-1 returns 7
```

3. **If I type this into STk, I get an unbound variable error:**
   `(eval-1 'x)`

This surprises me a bit, since I expected `eval-1` to return `x`, unquoted. Why did this happen? What should I have typed in instead?

The quote in front of x protects x from STk, but not from Scheme1. Recall that eval-1 is just a procedure, and for STk to make that procedure call, it first evaluates all its arguments – including (quote x) – before passing the argument to eval-1. Then, when eval-1 sees the symbol x, it tries to call eval on it, throwing an unbound variable error.

The problem, then, is that the expression 'x is evaluated TWICE – once by the STk evaluator, and once by eval-1. Thus, to protect it twice, you need two quotes:

```
(eval-1 ''x)

Note that if you type just 'x into Scheme1, it works:

Scheme1: 'x
x
```

Make sure you understand the difference, and why you don't need double-quote there (because STk is never told to evaluate 'x, unlike the previous case).

4. **Hacking `Scheme1`: For some reason, the following expression works:**
   `('(lambda(x) (* x x)) 3)`

Note the quote in front of the `lambda` expression. Well, it's not supposed to! Why does it work? What fact about `Scheme1` does this exploit?

When eval-1 sees the procedure call and tries to evaluate the procedure, it sees that it is a quoted expression, and unquotes it. Then, the procedure is passed to apply-1, which sees that it is a lambda expression, and uses it as such. This exploits the fact that in Scheme1, a compound procedure is represented as a list that looks exactly like the lambda expression that created it. Thus, even though we never evaluated the lambda expression into a procedure value, Scheme1 is still fooled into thinking it's a valid procedure.

QUESTION: The TAs have broken out in a cold war; apparently, at the last midterm-grading session, someone ate the last potsticker and refused to admit it. It is near the end of the semester, and Colleen really needs to enter the grades. Unfortunately, the TAs represent the grades of their students differently, and refuse to change their representation to someone else's. Colleen is far too busy to work with five different sets of procedures and five sets of student data, so for educational purposes, you have been tasked to solve this problem for her. The TAs have agreed to type-tag each student record with their first name, conforming to the following standard:

```
(define type-tag car)
(define content cdr)
```

It's up to you to combine their representations into a single interface for Colleen to use.

IN CLASS:

1. **Write a procedure, `(make-tagged-record ta-name record)`, that takes in a TA's student record, and type-tags it so it's consistent with the `type-tag` and `content` accessor procedures defined above.**

```
(define make-tagged-record cons)
Why not list?
```

2. **A student record consists of two things: a "name" item and a "grade" item. Each TA represents a student record differently. Hamilton uses a list, whose first element is a name item, and the second element the grade item. Eric uses a `cons` pair, whose `car` is the name item, and the `cdr` the grade item. Write `get-name` and `get-grade` procedures that take in a student record and return the name or grade items, respectively. We will do this in three different ways:**

a. Conventional-style type tagging

```
(define (get-name tagged-record)
   (cond ((equal? (type-tag tagged-record) 'Hamilton)
          (car (content tagged-record)))
         ((equal? (type-tag tagged-record) 'Eric)
          (car (content tagged-record)))))

(define (get-grade tagged-record)
   (cond ((equal? (type-tag tagged-record) 'Hamilton)
          (cadr (content (tagged-record)))
         ((equal? (type-tag tagged-record) 'Eric)
          (cdr (content tagged-record)))))
```

b. Data-directed programming

```
(put 'Hamilton 'get-name car)
(put 'Hamilton 'get-grade cadr)
(put 'Eric 'get-name car)
(put 'Eric 'get-grade cdr)

(define (get-name tagged-record)
   ((get (type-tag tagged-record) 'get-name) (content tagged-record)))
(define (get-grade tagged-record)
   ((get (type-tag tagged-record) 'get-grade) (content tagged-record)))

As you can see, the above two look very similar, so we can define a
   generic, "operate" procedure:

(define (operate op tagged-record)
   ((get (type-tag tagged-record) op) (content tagged-record)))

Then, we can just do:

(define (get-name tagged-record) (operate 'get-name tagged-record))
(define (get-grade tagged-record) (operate 'get-grade tagged-record))
```

c. Message passing – this one's a little different. Instead of going off an existing TA's implementation, you now have the chance to create your own. Create an implementation of student records using message passing, which can accept the messages `get-name` and `get-grade`.

```
(define (make-student-record name grade)
   (define (dispatch m)
     (cond ((eq? m 'name) name)
           ((eq? m 'grade) grade)
           ((eq? m 'student-info) (list name grade))
           (else (error "Bad Message"))))
   dispatch)

(define (get-name record)
  (record 'name))

(define (get-grade record)
  (record 'grade))
```

;note here the number of messages you could have is endless, in this situation the messages are somewhat trivial, but it is not hard to see how this could be useful, for instance, say we wanted to be able to scale a student's grade, we could easily add a more non-trivial message that would do this for us by returning a procedure, that takes a scale factor and uses it appropriately.

```
   ((eq? m 'scale-grade) (lambda (factor) (* grade factor)))

;it is examples like this that show the power of message passing.
```

EXTRA PRACTICE:

3. **Each TA represents names differently. Kevin uses a `cons` pair, whose `car` is the last name and whose `cdr` is the first. Phill is so cool that a "name" is just a word of two letters, representing the initials of the student (so George Bush would be `gb`). Write generic `get-first-name` and `get-last-name` procedures that take in a tagged student record and return the first or last name, respectively. Try this using conventional-style type tagging, then with data-directed programming.**

   ;with type-tagging

```
(define (get-first-name tagged-record)
   (cond ((equal? (type-tag tagged-record) 'Kevin)
          (cdr (get-name tagged-record)))
         ((equal? (type-tag tagged-record) 'Phill)
          (first (get-name tagged-record)))))

(define (get-last-name tagged-record)
   (cond ((equal? (type-tag tagged-record) 'Kevin)
          (car (get-name tagged-record)))
         ((equal? (type-tag tagged-record) 'Phill)
          (last (get-name tagged-record)))))
```

;with DDP

```
   There are a few ways to organize this. The obvious way is to put into the
      table procedures that take in a full student record and returns the
      first name:

   (put 'Kevin 'get-first-name (lambda (r) (cdr (get-name r))))
   (put 'Kevin 'get-last-name (lambda (r) (car (get-name r))))
```

```
(put 'Phill 'get-first-name (lambda (r) (first (get-name r))))
(put 'Phill 'get-last-name (lambda (r) (last (get-name r))))

(define (get-first-name tagged-record)
    (operate 'get-first-name tagged-record))
(define (get-last-name tagged-record)
    (operate 'get-last-name tagged-record))
```

Or, we can instead, put into the table procedures that take in a NAME ITEM rather than the whole record:

```
(put 'Kevin 'get-first-name cdr)
(put 'Kevin 'get-last-name car)
(put 'Phill 'get-first-name first)
(put 'Phill 'get-last-name last)

(define (get-first-name tagged-record)
    ((get (type-tag tagged-record) 'get-first-name)
     (get-name tagged-record)))
```

Unfortunately, this would mean we can no longer use operate. So the first way of doing this is neater.

4. **Each TA represents grades differently. Eric is lazy, so his grade item is just the total number of points for the student. Stephanie is more careful, so her grade item is an association list of pairs; each pair represents a grade entry for an assignment, so the `car` is the name of the assignment, and the `cdr` the number of points the student got. Write a generic `get-total-points` procedure that takes in a tagged student record and return the total number of points the student has. Try this using conventional-style type tagging, then with data-directed programming.**

;With type-tagging

```
(define (get-total-points tagged-record)
    (cond ((equal? (type-tag tagged-record) 'Eric)
           (get-grade tagged-record))
          ((equal? (type-tag tagged-record) 'Stephanie)
           (accumulate + 0 (map cdr (get-grade tagged-record))))))
```

;With DDP

```
(put 'Eric 'get-total-points get-grade)
```

Note that for Eric, the total-points is just the grade item, so we can just use get-grade as our procedure.

```
(put 'Stephanie 'get-total-points
     (lambda (r) (accumulate + 0 (map cdr (get-grade r)))))
```

We use map to get a list of points, and accumulate to add them up.

```
(define (get-total-points tagged-record)
    (operate 'get-total-points tagged-record))
```

5. **Now Colleen wants you to convert all student records to the format she wants. She has supplied you with her record-constructor, `(make-student-record name grade)`, which takes in a name item and a grade item, and returns a student record in the format Colleen likes. She also gave you `(make-name first last)`, which creates a name item, and `(make-grade total-points)`, which takes in the total number of points the student has and creates a grade item. Write a procedure, `(convert-to-colleen-format records)`, which takes in a list of student records, and returns a list of student records in Colleen's format, each record tagged with `'Colleen`.**

```
(define (convert-to-colleen-format records)
      (map (lambda(r)
                (make-tagged-record 'Colleen
                    (make-student-record
                        (make-name (get-first-name r)
                                   (get-last-name r))
                        (make-grade (get-total-points r)))))
           records))
```