CS61A Notes 06 – Data-Directed Programming (with some Scheme-1) [Solutions v1.0]

Data-Directed Programming

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 Professor Harvey 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. Professor Harvey 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 him. 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 Professor Harvey to use.

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. Min uses a list, whose first element is a name item, and the second element the grade item. Justin uses a `cons` pair, whose `car` is the name item, and the `cdr` the grade item. Make calls to `put` and `get`, and write generic `get-name` and `get-grade` procedures that take in a tagged student record and return the name or grade items, respectively.**
   ```
   (put 'Min 'get-name car)
   (put 'Min 'get-grade cadr)
   (put 'Justin 'get-name car)
   (put 'Justin '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))
   ```

3. **Each TA represents names differently. Darren uses a `cons` pair, whose `car` is the last name and whose `cdr` is the first. Jerry 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`). Make calls to `put` and `get` to prepare the table, then 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.**

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 'Darren 'get-first-name (lambda(r) (cdr (get-name r))))
(put 'Darren 'get-last-name (lambda(r) (car (get-name r))))
(put 'Jerry 'get-first-name (lambda(r) (first (get-name r))))
(put 'Jerry '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 'Darren 'get-first-name cdr)
(put 'Darren 'get-last-name car)
(put 'Jerry 'get-first-name first)
(put 'Jerry '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. Ahmed is lazy, so his grade item is just the total number of points for the student. Justin is more careful, so his 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. Make calls to `put` and `get` to prepare the table, and write a generic `get-total-points` procedure that take in a tagged student record and return the total number of points the student has.**

```
(put 'Ahmed 'get-total-points get-grade)
   Note that for Ahmed, the total-points is just the grade item, so we
   can just use get-grade as our procedure.

(put 'Justin 'get-total-points
   (lambda(r) (apply + (map cdr (get-grade r)))))

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

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

5. **Now Professor Harvey wants you to convert all student records to the format he wants. He has supplied you with his 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 Professor Harvey likes. He 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-harvey-format records)`, which takes in a list of student records, and returns a list of student records in Professor Harvey's format, each record tagged with `'Brian`.**

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

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

QUESTIONS

1. **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 cal 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).

2. **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.