

CS61A Notes – Week 6: Generic operators (solutions)

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. Ahmed uses a list, whose first element is a name item, and the second element the grade item. Jerry 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 'ahmed 'get-name car)
(put 'ahmed 'get-grade cadr)
(put 'jerry 'get-name car)
(put 'jerry '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. George uses a cons pair, whose `car` is the last name and whose `cdr` is the first. Michael 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 'george 'get-first-name (lambda (r) (cdr (get-name r))))
(put 'george 'get-last-name (lambda (r) (car (get-name r))))
(put 'michael 'get-first-name (lambda (r) (first (get-name r))))
(put 'michael '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 'george 'get-first-name cdr)
(put 'george 'get-last-name car)
(put 'michael 'get-first-name first)
(put 'michael '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. Justin is lazy, so his grade item is just the total number of points for the student. Fares 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 takes in a tagged student record and return the total number of points the student has.**

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

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

```
(put 'fares '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-creator, (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))
```