

CS61A Notes – Week 6: Generic operators

Generic Operators

A generic operator, abstractly, is a procedure that can interface over several different kinds of representations of arguments. When you use '+' in mathematics, for example, you often use it to add integers to integers, integers to reals, vectors to vectors, etc. There are three basic ways to implement generic operators in Scheme:

1. Explicit dispatch
2. Message passing
3. Data directed

Explicit dispatch is essentially one big `cond` clause. In other words, it implements an operator that behaves differently for each object type. This is the conventional approach.

Message passing implements some object as a procedure that takes a “message” as an argument. If we want to perform an operation, simply pass the desired message to the object; the generic operator for this, then, may look something like:

```
(define (apply-generic op arg)
  (arg op))

(define (make-fraction num den)
  (define (dispatch op)
    (cond ((eq? op 'numerator) num)
          ((eq? op 'denominator) den)
          (else (error "Unknown op - MAKE-FRACTION" op))))
  dispatch)
```

`apply-generic` simply gives some argument (which could be a `make-fraction` as defined above) the desired operation (which could be the word `numerator`). In this case, the program is “smart”, but the data is “dumb”; the program has all the information on how to execute, and all it has to be given is a word that represents what operation to perform. For now, this is all there is to message-passing, but in just a few weeks we'll

Data directed as an approach to generic operators is, as we'll see, quite meaty. This approach can be considered the rough “opposite” of the message passing approach; instead of smart procedures, we'll create a dumb `apply-generic` (different than the one above!) and give it a smart operation.

Data Directed Programming

Up until now, we have been writing “smart programs” – programs that know what to do given the arguments. But with data-directed programming, we're slowly moving toward the paradigm where programs are dumb, but data are smart. We will store relevant information about object types and operators in some sort of “lookup table” (which often but does not necessarily imply the use of `put` and `get`). Objects are tagged using the `tagged-data` ADT (`attach-tag`, `type-tag`, `contents`) to keep track of which operators to use.

The advantage is simple: maintainability. Remember, programs or procedures are complicated beasts, and you should cringe every time you need to modify a working procedure. Yet data is simple – putting new items into a global table doesn't require you to alter existing code. Adding new features will rarely break old, working code.

The essentials of SICP's data-directed generic operator:

```
(define (apply-generic op . args)
  (let ((proc (get op (map type-tag args))) ) ;; look up operator
        (apply proc (map contents args)))) ;; use operator
```

The period is like the period in pair notation, which allows us to put the rest of the arguments as a list of `args`:

```
(apply-generic 'add num1 num2)
;; op = add, args = (num1 num2)
```

Finally, `STk`'s `apply` works as follows:

```
(apply + '(1 2 3)) => (+ 1 2 3)
```

As an example of real-life data-directed “stuff”, think airports. They'll tag your baggage with a destination (`type`), and when they ship (`operator`) your bag, they check the type, look up where to send it, and place it in the appropriate luggage container. Let's look at another real-life example:

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.
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.

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.

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.

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`.