CS107 Handout 51
Autumn 2000 November 29, 2000

A Larger LISP Example: Computing Statistics

Original work written by Todd Feldman.

During our LISP lectures, I'll be showing snippets of the language, introducing the syntax and usage of various functions, but we won't have much of a chance to develop "big" programs in the short lecture time we have. To give you a little more cohesive feel about the language, this handout walks you through developing a LISP program composed of several functions and tries to guide you in the way of bottom-up development. The strategy taken here is akin to the way you want to attack writing any large program in LISP.

The Problem

We want to write a program that computes various statistics about a course given the grades of each student. Our program will accept input a single list of the form:

```
(<student1 grades> <student2 grades> . . . <studentN grades>)
```

(for any `N`) where each set of student grades is a list of the form:

```
((<assignment scores>) <midterm score> <exam score>)
```

and `<assignment scores>` is a list of integers and both the midterm and exam scores are integers. 100 is a perfect score on each course component. The formula for computing each student's course grade is 40% of the assignment average, 20% midterm, and 40% exam. We wish to compute the high, low, and average course grades for the entire class. Here might be a set of course grades, expressed as a data list:

```
          '(((77 87 95 75) 80 65)      ;; first student's grades
             ((88 78 99 80) 71 90)      ;; next student's grades
         ...
             ((91 75 69 83) 65 100))    ;; last student's grades
```

Working From the Bottom Up

If we start from the bottom (often a good approach in LISP), we can begin by computing the course grade for one student. Start with `course-grade` itself, and don't be afraid to just make calls to functions like `average` that you haven't even written yet, you'll get back to that in a minute. (Although the interpreter may warn that you're using a function that hasn't yet been defined, you don't get into trouble unless you try to call a function before you finish implementing the underlying components.)

```
? (defun course-grade (scores)
   (+ (* (average (first scores)) 0.40)      ;; first is synonym for car
      (* (second scores) 0.20)               ;; second - tenth also defined
      (* (third scores) 0.40)))
COURSE-GRADE
```

The `course-grade` function will expect a list with one student's grades, in the format of above. It's probably a good idea to comment the function with this expectation so we'll remember how to use it later.

Okay, so `course-grade` needs the function `average`, which in turn requires the function `sum`. So we take care of those:

```
? (defun average (list)
     (/ (sum list) (length list)))
AVERAGE

? (defun sum (list)
     (if (null list) 0
         (+ (car list) (sum (cdr list)))))
SUM
```

Before we go on, we should stop and test out what we have and make sure it works. LISP programs can quickly turn into a dense, gnarled forest of code and your best strategy is to write very small functions in isolation and test them thoroughly before moving on to incorporate them in a larger program. So, give `sum` a little workout straight from the interpreter and see how it performs. Be sure to try out a variety of cases to discover all its behaviors.

```
? (sum '(4 5 6))
15
? (sum '())
0
? (sum '(4.5 7.5 4 3 12/3))
23.0
? (sum '(1 4 "hi"))
> Error: value "hi" is not of the expected type NUMBER.
> While executing: SUM
> Type Command-. to abort.
```

And now do the same thing with `average`:

```
? (average '(4 5 6))
5
? (average '(4.5 7.5 4 3 12/3))
4.6
? (average '())
> Error: DIVISION-BY-ZERO detected
>        performing TRUNCATE on (0 0)
> While executing: CCL::DIVIDE-BY-ZERO-ERROR
> Type Command-. to abort.
? (average '(1 4 "hi"))
> Error: value "hi" is not of the expected type NUMBER.
> While executing: AVERAGE
> Type Command-. to abort.
```

Note the attempt to average an empty list raises an error. We could design the function to handle an empty list as a special case, but would this really be an improvement? What value should the function return in that case, 0? If averaging an empty list returned 0, that would allow the original mistake to be covered up and would propagate bad results to somewhere else in the program, which is really not desirable. Think back to our strategy of working comprehensive asserts into our C code, and realize we are getting some of the same protections, but from the language itself without extra work on our part.

And now `course-grade` can be tested, by making up some scores to pass to it:

```
? (course-grade '((80 80 80 80) 80 80))  ;; easy to verify
80.0
? (course-grade '((88 78 99 80) 71 90))
84.7
```

Only after we're sure that our helper functions all work properly (and are nicely commented) do we want to move on. From here, we can quickly put `course-grade` to use in a function `compute-all-grades` that calls it for each student to build a new list of computed course grades.

```
? (defun compute-all-grades (students)
  (if (null students) '()
      (cons (course-grade (car students))    ;; usual car-cdr recursion
            (compute-all-grades (cdr students)))))
COMPUTE-ALL-GRADES

Again, a run of testing...
? (compute-all-grades '(((95 87 95 75) 80 65)
                         ((88 78 99 80) 71 90)))
(77.2 84.7)
```

Note that `compute-all-grades` returns a list of final scores, one for each student. It is a good idea to add a comment for the function describing the expected input format, as well as the generated output format, in the function comments. The untyped nature of LISP can make it hard for the reader to sort out what is expected without some direction from the code's author.

So far, so good. Now let's work on the function `compute-course-stats` which computes the various statistics of the `compute-all-grades` results. Here's our first attempt, again we are calling as-yet-non-existent functions that we'll define a bit later:

```
? (defun grade-stats (students)
     (list 'high (find-max (compute-all-grades students))
           'low (find-min (compute-all-grades students))
           'average (average (compute-all-grades students))))
```

```
GRADE-STATS
```

We make a list of the results, but rather than just return a list of three numbers, we label each with a symbol name (quoted so the interpreter doesn't attempt to look up values) so that the result is a little clearer.

We already have an `average` function, here is our first stab at `find-max`:

```
? (defun find-max (list)
   (if (null list) 0          ;; reasonable value for this case?
       (if (= (length list) 1) (car list)
          (if (> (car list) (find-max (cdr list)))
              (car list) (find-max (cdr list))))))
FIND-MAX
```

In testing the code, (as you definitely should, before moving on), you're likely to stumble across things you didn't quite see the first time. For example, why do we have two base cases here? I claim if we take out the second base case (the test for a list of length 1) the function will behave poorly under certain circumstances. Do you see why? Try testing it some on your own to find out what is going on. But doesn't having the second base case mean the first base case is never reached? The first base case is hit in one situation— what it is? What is the right thing to do in a case such as that one? Make a choice and document it so you remember!

Note that `find-max` is pretty inefficient in that it finds the max of the `cdr` twice— once to use it in the comparison operation, and perhaps again to return it. Similarly in the `grade-stats` function we compute all of the students' course grades for each course statistic! This is very wasteful.

Using `let`

To fix this, we use `let` to **bind** the values to temporary variables after computing them just once.

```
? (defun find-max (list)
    (if (null list) 0
      (if (= (length list) 1) (car list)
         (let ((firstElem (car list))
               (maxOfRest (find-max (cdr list))))
               (if (> firstElem maxOfRest) firstElem maxOfRest)))))
FIND-MAX
```

Similarly `find-min` will be:

```
? (defun find-min (list)
    (if (null list) 0
      (if (= (length list) 1) (car list)
        (let ((firstElem (car list))
              (minOfRest (find-min (cdr list))))
              (if (< firstElem minOfRest) firstElem minOfRest)))))
```

```
   FIND-MIN
```

(Notice the above two functions are identical except for the comparator used in the very last expression. After we've learned about using functions as data, you could revisit this example and unify these functions.)

And finally, a more efficient form of `grade-stats` using `let` :

```
   ? (defun grade-stats(students)
       (let ((all-grades (compute-all-grades students)))
         (list 'high (find-max all-grades)
               'low (find-min all-grades)
               'average (average all-grades))))
   GRADE-STATS
```

Before we move on, thoroughly testing `find-min`, `find-max` and `grade-stats` is definitely in order. Adding some comments about how they work and how to use them is a good idea as well.

```
   ? (find-max '(4 0 10 23 56 101 -4 -5))
   101
   ? (find-min '(4 0 10 23 56 101 -4 -5))
   -5
   ? (find-min '())
   0
   ? (grade-stats '(((95 87 95 75) 80 65)
                    ((88 78 99 80) 71 90)))
   (HIGH 84.7 LOW 77.2 AVERAGE 80.95)
```

Using the built-in sort

What if we wanted to include the median to our report? The median is defined as the middle value— half the values are above, half below. The easiest way to compute the median is to sort the list and pull out the middle element. If the length is odd, there is exactly one middle element. If the length is even, the median is the average of the middle two elements. We use `let` to avoid re-evaluating expressions.

```
   ? (defun median (list)
       (let ((len (length list))
             (sortedList (sort list #'<)))
         (if (evenp len)            ;; evenp is a built-in predicate
           (/ (+ (nth (- (/ len 2) 1) sortedList)     ;; average middle two
                 (nth (/ len 2) sortedList))
              2)
           (nth (/ (- len 1) 2) sortedList))))  ;; get middle elem
   MEDIAN
```

A few new LISP functions used here:

- `nth` takes two arguments: an integer index and a list. It returns the element of the specified list with the specified index. Note that indices begin at 0, as you're used to from arrays in C.

· The built-in sort function which sorts a list by some two-argument predicate. Note that this predicate is a function being passed as a parameter, like the way we used function pointers in C to supply the comparator function to qsort (but functional parameters are much cooler in LISP— we'll see more of this in the upcoming lectures). For today, just accept we are passing the usual less-than function symbol. The prefacing #' is used to access the functional object bound to the symbol. sort returns the list with its elements rearranged so that the predicate applied to any two consecutive elements returns T.

Try median out on its own to make sure it works properly and then you can incorporate it into the grade report to get our final result.

```
? (grade-stats '(((95 87 95 75) 80 65)
                 ((88 78 99 80) 71 90)
                 ((94 77 84 91) 83 80)))
(HIGH 84.7 LOW 77.2 AVERAGE 81.7 MEDIAN 83.2)
```