COMP 200: Elements of Computer Science
Fall 2004
Lecture 6: September 3, 2004

Finish up Conditionals, On to the Matter of Names

On the Board

Homework 2 available on the web site; homework 1 returned
No class Monday

Back to FirstClassMail

At the end of last class, we had written the program

```
; FirstClassMail: posnum → num
; Purpose: Given the weight of a letter in ounces,
;      compute the first class, domestic postage
(define (FirstClassMail Ozs)
    (cond
        ((= 0 Ozs)  0)
        ((= 1 Ozs)  37)
        ((<= 2 Ozs) (+ 37 (* (- Ozs 1) 23)))
     ))
```

and the observation that this program computed the correct answer for non-negative integers (counting numbers, starting with zero). For fractions of an ounce, it produces preposterous answers that do not match any denomination of stamp sold by the post office.

I tried to fix it quickly and violated one clear cut rule. I didn't go back to the design methodology. The problem is that the contract is not tight enough. It requires a positive number, a category that includes fractions. The program that we designed works correctly only for inputs drawn from the set 0, 1, 2, 3, 4, … —a set that we will call the ***natural numbers*** in COMP 200.

Some people, particularly mathematicians, will take issue with including zero in the natural numbers. Their stance is justifiable. My stance is based on the needs of COMP 200. We will consider zero as the lowest valued natural number.

But all this discussion ignores the critical point. Not all letters weigh an integral number of ounces. If our program is to be useful, it must work for

an arbitrary positive number as input. Two solutions are easy to envision. The first solution creates a "helper" function that converts the posnum to the next largest natural number. We encountered this function in our discussion of binary search — when we computed the midpoint, we took the ***ceiling*** of (low + high)/2. Our helper function is straightforward; it invokes FirstClassMail with the ceiling of its argument. (See Scheme Code for the lecture; program FirstFix.)

The second option is to fix up the original code so that it works over natural numbers. (See Scheme Code for the lecture, program SecondFix.) Notice that we had to use a fairly complex expression to find the range (0,1].

Another Example

In algebra, we have the notion of a number's absolute value. Develop a small Scheme program that computes the absolute value of a number. Follow all the steps of the design methodology:

1. Write a contract, purpose, and header
2. Develop a table of test data and expected answers
3. Work out the expressions required for the body of the function

   HINT: This problem requires a number line, as did FirstClassMail.
4. Code up the program
5. Test the program on your data from step 2

Work with the person sitting next to you. Someone will volunteer to share their code with the rest of us.

My Solution

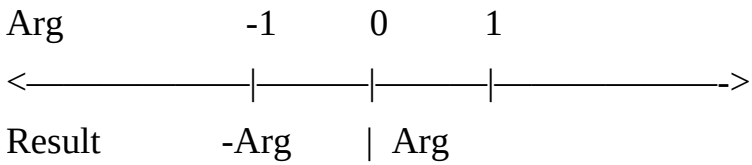
1. Contract, Purpose, Header

   ; ABSVAL: num → posnum
   ; Purpose: return the absolute value of its input argument
   (define (ABSVAL Arg) …)

2. Table of test data

Argument	Result
0	0
1	1
-1	1

Looking at this problem on the number line, it is clear that the problem has two regions of interest.



3. Expression for the code body

```
(cond
   ((<   Arg 0) (* Arg –1))
   ((>= Arg 0)  Arg)) )
```

4. The entire program

```
; ABSVAL:  num → posnum
; Purpose: return the absolute value of its input argument
(define (ABSVAL Arg)
   (cond
      ((<   Arg 0) (* Arg –1))
      ((>= Arg 0)  Arg)) ) )
```

Moving Forward

Assume that you are teaching a class with 25 students and have just graded their first homework. You would like to compute some elementary statistics on their grades, such as the average grade. How would we go about developing a program to perform this computation?

The difficult issue that we face goes back to the second day of class — names for data and names for values. We have 25 students, each with some number of homework grades — with a more organized professor, we might know in advance how many homeworks, three test grades, and a final project grade. How do we name each of these grades in a way that we can record them, that we can manipulate them, and that we can remember the names? (After all, programming is a human activity and our limited memories play a large role in our ability to develop programs.)

Practical Answer – A Spreadsheet

See digression on spreadsheets.

Programmer's Answer – Structures, Arrays, and Lists

What do we need? That is the key question in designing a program that must handle a large amount of data. For the COMP 200 grading program,

we need the ability to handle multiple students, each with multiple grades, and the ability to write simple programs that can operate on this data to produce statistics such as averages, summations, and ranks in class.

To handle such data, programming languages (in general) offer three kinds of abstractions:

- Structures — data (of a potentially dissimilar nature) grouped together because it is related
- Arrays — multidimensional tables of similar items
- Lists — an unbounded list of similar items

Most languages permit combinations of these structures — an array of structures, a structure containing an array, a list of anything, a structure that contains a list.

In COMP 210, the class delves into the intricacies of structures in great detail. In COMP 200, much of that practice is immaterial. We will skim over structures to reach lists. Scheme provides relatively poor support for arrays, so we will deal with them only in the abstract (for now).