COMP 200: Elements of Computer Science
Fall 2004
Lecture 7: September 8, 2004

Structures and Aggregates

On the Board

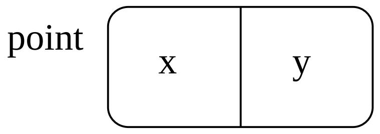
Homework 2 due 9/15/2004
Read Chapter 2 — as of today, you have the background

Structures — A First Example

Computer displays are (now) filled with windows, menus, popup menus, and so on. Each of these forms is (roughly) a rectangle. To represent the myriad rectangles on the screen, we can use the notion of points on a Cartesian plane. A window (or a menu pane) is simply two opposite corners; by convention, the upper left and lower right corners. We must establish an origin (by convention, the lower left corner.

To represent these points in programs that manipulate screen images, we can use a structure. The structure needs an x coordinate and a y coordinate. In our model, it needs no other data, although a real program might associate more information with the point (color, intensity, kind of rectangle, other objects that it overlays or underlies on the screen, etc.)

If I asked you to draw a point, it might look like:



In Scheme, we create such a structure as follows:

```
; a point is a structure
;   (make-point x y)
;  where x and y are numbers
(define-struct point (x y))
```

Comments for the methodology and later readers …

Actual Scheme command

When this command executes (type it into the Definitions pane of Dr. Scheme and click the ***Run*** button), it creates a set of functions that a Scheme program (or expression) can use to manipulate ***points***. These functions include:

make-point : number number → point ← Constructor

point-x : point → number
point-y : point → number ← Selectors

make-point creates a new ***point*** and sets its values for x and y to the values of its two arguments, in the order used in the **define-struct** construct. The new ***point*** is an object in Scheme's workspace that persists until it can no longer be seen (end of the session, or loss of the last reference to it). Once you can no longer name it, Dr. Scheme is free to reclaim any space it occupies. Since you cannot name it, you cannot tell if Dr. Scheme maintains it or recycles it.

point-x and point-y take as their sole argument a ***point***. They return the value of the appropriate field — x or y — inside the point.

define-struct creates other functions for manipulating points, but we won't get to discuss those for a while.

With these functions, we can write programs that create and manipulate points. For example, to add two points,

```
; PointAdd:  point point → point
; Purpose:     Add two points to produce a new point
(define (PointAdd p1 p2)
    (make-point (+ (point-x p1) (point-x p2))
                      (+ (point-y p1) (point-y p2))
     ))
```

To move a point on the screen, we "translate" it by an offset in the x direction and an offset in the y direction, precisely what PointAdd does.

Writing Programs with Structures

When we start using structures, the methodology for writing programs becomes more complex in two ways. First, we need to think (actively) about the kind of data that the program will encounter (will need?) and how to represent that data. This requirement adds a step before "contract, header, and purpose" called "***data analysis***." Second, to remind ourselves of the possible selectors that we may need, we also write a template for each structure — a skeleton of a function that accesses that structure. It includes all the selector functions that we may need in manipulating instances of that structure. Now, the methodology is as follows:

1. Analyze the data and develop any structures that are needed; for each structure write a function templatte
2. Write a contract, purpose, and header for the program
3. Develop a table of test data and expected answers
4. Work out the expressions required for the body of the function

   HINT: This problem requires a number line, as did FirstClassMail.
5. Code up the program
6. Test the program on your data from step 3

Applying the methodology to another program, we can develop a program that computes the area of a rectangle (as in a window on your computer screen) from two points that define its opposite corners. Assume that the program takes as input the upper left and lower right corners of the rectangle, represented as points.

Step 1: *Analyze the Data*

Input is two points. We can use the definition of point given earlier in the lecture

```
; a point is a structure
;   (make-point x y)
;  where x and y are numbers
(define-struct point (x y))

; Template for functions that deal with instances of point
(define (ApointFunction ThePoint)
       ( … (point-x ThePoint) …
          … (point-y ThePoint) … )
```

Step 2: *Contract, Purpose, Header, Template*

```
; RectangleArea: point point → number
; Purpose: given points that define the upper left and lower right
;                corners of a rectangle, compute the rectangle’s area.
(define (RectangleArea ul lr) … )
```

Step 3: *Test Data*

Inputs	Results
(make-point 0 0) (make-point 2 4)	8
(make-point 0 0) (make-point –2 –4)	8
(make-point 1 1) (make-point 1 1)	0

Step 4: *Work out the code body*

Need to compute the sides and multiply them. Subtract x coordinates, then y coordinates. Placement of the origin affects the order in which we should subtract them. Can use absolute value to eliminate the entire order issue.

The function works with points, so we can refer back to the template to remind us of what selectors are available inside the point structure.

Something along the lines of

```
(abs (- (point-x ul) (point-x lr)))
```

yields the size of the x dimension. Similar expression yields the size in the y dimension. Thus, the area should be

```
(*  (abs  (-  (point-x ul) (point-x lr)) )
    (abs  (-  (point-y  ul) (point-y lr)) ) )
```

Step 5: *Code up the program*

```
; RectangleArea: point point → number
; Purpose: given points that define the upper left and lower right
;              corners of a rectangle, compute the rectangle's area.
(define (RectangleArea ul lr)
   (*  (abs (- (point-x ul) (point-x lr) ) )
       (abs (- (point-y ul) (point-y lr) ) )
       ))
```

Step 6: *Test the program on the data from Step 3*

Try it yourself in Dr. Scheme or hand evaluate the expressions using the rewriting semantics. (Expect to have a hand evaluation on the first test. The rewriting semantics for a selector function are simple — replace the selector and its argument, such as (point-x ul), with the value of that field in the structure.