6.090, Building Programming Experience
Ben Vandiver
Lecture 5: List Procedures and Data Abstraction

Review of list procedures:

cons:	(cons elem lst)
car:	(car lst) → first element
cdr:	(cdr lst) → the rest of the elements (all but first)
cadr:	(car (cdr lst))→ second element
list-ref:	(list-ref lst k)→ kth element of lst (remember zero-based indexing)
list:	(list elements)→ (elements)
append:	(append lst1 lst2)→list of all elements in lst1 and lst2, ordered

list-copy: given a list, it makes a copy of the exact list and returns it.

Create a procedure called list-copy.
Write a plan:
Base case: nil→nil
Recursive case: (cons (car lst) (list-copy (cdr lst)))

Consider (cons 1 (2 3)) → (1 2 3) when trying to understand the above.

Now we define.

```
(define (list-copy lst)
  (if (null? lst)
      nil
      (cons (car lst) (list-copy (cdr lst)))))
```

The last primitive procedure above, cdr, is "cdr-ing" down the list lst.

Create a procedure called n-copies that will make n copies of an element and enclose them within a list.

Write a plan:
Base case: copies = 0 → nil
Recursive: (cons elem (n-copies elem (- copies 1)))

Let's analyze from the innermost procedure out. We are taking one less than the amount of copies we need, keeping the element unchanged, performing the n-copies procedure on that unchanged element for 1 fewer copies, and adding one identical element to the front of the result of the n-1 copies list.

```
(define (n-copies elem copies)
  (if (= copies 0)
      nil
      (cons elem (n-copies elem (- copies 1)))))
```

Create a procedure called reverse to reverse a list.
(1 2 3) => (3 2 1)

Write a plan:
Base case: nil→ nil
Recursive case: stick (car lst) on the end of (reverse (cdr lst))
In scheme terms, (append (reverse (cdr lst)) (list (car lst)))

```
(define (reverse lst)
  (if (null? lst)
      nil
      (append (reverse (cdr lst))
              (list (car lst)))))
```

Change the above recursive process to an iterative process.

```
(define (reverse-helper lst answer)
  (if (null? lst)
      answer
      (reverse-helper (cdr lst) (cons (car lst) answer))))

(define (reverse lst)
        (reverse-helper lst nil))
```

The following is the iterative process that occurs through this procedure:

```
(reverse (list 1 2 3))
(reverse-helper (1 2 3) ( ))
(reverse-helper (2 3) (1))
(reverse-helper (3) (2 1))
(reverse-helper ( ) (3 2 1))
```

Create the procedure append that glues two lists together.

Write a plan:
Base case: lst1 is null → lst2
Recursive: (cons (car lst1) (append (cdr lst1) lst2))

```
(define (append lst1 lst2)
  (if (null? lst1)
      lst2
      (cons (car lst1) (append (cdr lst1) lst2))))
```

Create the procedure list-ref that takes in a list and a number n and returns the nth element of that list in the zero-based index system.

Write a plan:

Base case: (list-ref lst 0)→car lst
Recursive case: (list-ref (cdr lst) (- n 1))

```
(define (list-ref lst n)
  (if (= n 0)
      (car lst)
      (list-ref (cdr lst) (- n 1))))
```

Side note: some of the coding done involves defining functions that are already programmed in Scheme. If ever there is a need to get the original version of the procedure back, evaluate the following in Scheme:

(define *name of procedure you defined* (access *name of procedure you defined* #f))
This gets the primitive procedure back

Create a procedure list-range that takes in two numbers and returns the range of numbers between the first and second, inclusive.

See the solutions to lecture 5 on the course website for the solution to this problem.

Abstraction

Abstraction is the basic concept of isolating details of implementation from use. (Example given in class: when we use a door, i.e., open and close it, we don't need to know its internal mechanics in order to use it properly and successfully. It just works. This is analogous to abstraction in programming, procedural abstraction, where the user (or client) does not know what is behind the functionality of a program…he only knows how to use the program successfully.

A data abstraction has several parts:
Constructors create the data abstraction from its component parts
Selectors retrieve information
Every abstraction has a name
Values have types: numbers, Booleans, strings, lists, etc. Abstractions have derived types, types that only exist in our heads, but that are not formal. This idea will be mentioned later in the notes.*

Point Data Abstraction

It is important to implement the user-defined type in the following order:

1. Constructor: procedure which takes in x and y (a point) in scheme
   The procedure defined below makes a point in the Cartesian coordinate plane:

```
(define (make-point x y)
  (list x y))
```

   Takes in two numbers and returns a point, so the types are

   number, number → point

   point is a user-defined type

   (define p1 (make-point 5 7)) → p1 is a point at (5,7)
   p1 prints out as: (5 7)
2. Selectors: select out parts of the abstraction, often values supplied to the constructor.
   in this case, we need two selectors:

```
(define (get-x p)
  (car p))
(define (get-y p)
  (cadr p))
```

3. Contract: between constructor and selector that makes data abstraction
   (get-x (make-point 5 7)) => 5
   (get-y (make-point 5 7)) => 7

We want to add p1 and p2 together.

```
(define (add-point p1 p2)
  (make-point (+ (get-x p1) (get-x p2))
              (+ (get-y p1 (get-y p2)))))
```

```
(define v (add-point p1 p2)
v => (10 14)
```

Type of add-point is: point, point -> point

"client" of data abstraction, as mentioned before, is the user. This user is not aware of lists used to create the function add-point or the function get-x. This is important to recognize.

Create a procedure that determines if a point is to the left of another point:

```
(define (left-of? p1 p2 )
  (< (get-x p1) (get-x p2)))
```

Evaluating left-of? in an example:
(left-of? (make-point 3 4) (make-point 1 2))
;Value: #f

append: list, list → list
left-of?: point, point → Boolean

*Above is the second mention of derived types. Point is a type we "derived" to talk about the function left-of?, which determines if the first point is left of the second point. The return value is a Boolean. Types help to write code and to check code.

Abstraction Violation

```
(define (left-of? p1 p2)
  (< (car p1) (car p2)))
```

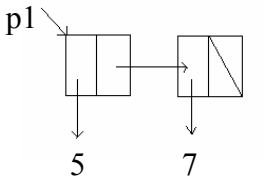
The above depends on specific implementation. We cannot use car, because car could have been predetermined as a different procedure used to find the y value if the coordinates in the definition of get-y had been written as (y x) instead of (x y).

A line must be imagined to exist between already-created constructors and selectors, and the contract between the two.

If you assume you know the implementation of constructor/selector, problems arise, because the coding in constructors and selectors may change. This change is unknown to "user", so use contract…not abstraction violation.

Here's a visual:

When we know what the list looks like, we can picture it like this:



When we don't know what the list looks like, we can picture as a cloud:

The rest of the notes consists of defining stacked abstractions (defining new abstractions with previously-defined abstractions). The solutions are posted at the solutions to lecture 5, posted on the server.

Layer abstraction: we have the definition of points, but we want to keep going. Construct an abstraction entitled "segment" to define a line segment, which is constructed from a pair of end points.