Lecture 5: List Procedures and Data Structures

List shorthand

- (cadr lst) = (car (cdr lst))
- (caar lst) = (car (car lst))
- (caadar lst) = (car (car (cdr (car lst))))

Writing a simple list procedure

- Define **list-length** that computes the length of a list.
- What's the recursive case?

- What's the base case?

Checking for the empty list

- The base case is an empty list.
- How do we check for an empty list?
  - **null?** Predicate
- (null? lst) evaluates to #t if lst is the empty list

length

- Plan:
  - Base Case: Empty list -> 0
  - Recursive Case: 1 + length of “rest of the list”

- (define (length list)
    to-be-completed)

list-copy

- Write **list-copy**, which takes a list and returns an identical new list
  - (Do not just return the original list, cons up a new list).

```
(list-copy (list 1 2 3))
;Value: (1 2 3)
```

n-copies

- Write n-copies, which takes a value and a number of copies, and returns a list with the appropriate number of copies.

```
(n-copies 7 5)
;Value: (7 7 7 7 7)
(n-copies "yay" 1)
;Value: ("yay")
(n-copies 7 0)
;Value: () ; or #f
(n-copies (list 3) 3)
;Value: ((3) (3) (3))
```

reverse

- Write reverse, which takes a list and returns new list with the order of the elements reversed.

```
(reverse (list 1 2 3))
;Value: (3 2 1)
(reverse (list 1))
;Value: (1)
```

reverse

- Is this iterative or recursive?
- Let’s write the alternate version

append

- Write **append**, which takes two lists and returns a new list with the elements of the first list and the second list.

```
(append (list 3 4) (list 1 2))
;Value: (3 4 1 2)
(append nil (list 1 2))
;Value: (1 2)
```

list-ref

- Write **list-ref**, which takes a list and an index (starting at 0), and returns the nth element of the list. You may assume that the index is less than the length of the list.

```
(list-ref (list 17 42 35 "hike") 0)
;Value: 17
(list-ref (list 17 42 35 "hike") 1)
;Value: 35
(list-ref (list 17 42 35 "hike") 2)
;Value: 35
```

list-range

- Write **list-range**, which takes two numbers (a,b : a < b) and returns a list containing the numbers from a to b, inclusive.

```
(list-range 1 5)
;Value: (1 2 3 4 5)
(list-range 2 5)
;Value: (2 3 4 5)
(list-range 42 42)
;Value: (42)
(list-range 207 5)
;Value: ()
```

list-range

- Write **max-list,** which takes in a list of numbers and returns the maximum element. You may assume that the list is non-empty.

```
(max-list (list 1))
;Value: 1
(max-list (list 1 3 5))
;Value: 5
(max-list (list 2 56 8 43 21))
;Value: 56
```

Data Abstraction

- Scheme provides us with a a decent set of data types
- We may want to create more complicated types of data
  - Points
  - Vectors
  - Matrices

Compound Data

- We can use specially formed lists to put data together into data structures
- We will define special procedures to work with this data type
- We will define a **point** data structure as an example
  - A point has coordinates x and y

Constructor

- This procedure creates the data structure

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

Selectors

- These procedures select out data from the data structure.
- Usually, the selectors correspond to the arguments in the constructor
- What selectors will we need for a point?

Contract

- The selectors and constructors must be written to enforce a contract:

  (get-x (make-point 5 7)) => 5

  (get-y (make-point 5 7)) => 7

Abstraction Barrier

- Now that we have defined the data abstraction, it doesn't matter how a point is defined.
- Only the constructors and selectors are used

Add-point

- Write a procedure that takes two points and creates a new point that is the sum of the two points.

left-of

- Write a procedure that takes two points, p1 and p2, and returns true if the p1 is to the left of p2

Defining a new abstraction

- Define the abstraction **segment** that represents a line segment
- Consists of two end-points
- What functions do we need to define?

segment-length

- Write a procedure that computes the length of a segment.