6.090 Building Programming Experience

Lecture 4

1/16/2007

Outline

- List Procedures
- Compound Data Types

Finding the length of a list

- Define a procedure `length`
- What's the recursive case?
- What's the base case?

Finding the length of a list

- Define a procedure `length`
- What's the recursive case?
  - Add 1 to length of "smaller list"
- What's the base case?
  - Empty list -> 0

Checking for the empty list

- Base case is an empty list
- Check for it using the **null?** Predicate
- (null? lst) returns #t if lst is the empty list, and #f otherwise

length

```
(define (length lst)
  (if (null? lst)
      0
      (+ 1 (length (cdr lst)))))
```

List-copy

- Define `list-copy` which takes a list and returns an identical new list
  - Cons up a new list, don't return the original
  - `(list-copy (list 1 2 3))`
    `;Value: (1 2 3)`

List-copy

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

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: ()
(n-copies (list 3) 3)
Value: ((3) (3) (3))
```

n-copies

```
(define (n-copies v n)
  (if (= n 0)
      null
      (cons v (n-copies
                v
                (- n 1)))))
```

append

- Write append, which takes two lists and returns a new list with the elements of the first list and second list
- `(append (list 3 4) (list 1 2)`
- `Value: (3 4 1 2)`
- `(append null (list 1 2))`
- `Value: (1 2)`

append

```
(define (append l1 l2)
 (if (null? l1)
     l2
     (cons (car l1)
           (append (cdr l1) l2))))
```

reverse

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

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

Reverse

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

Reverse

- Is this iterative or recursive?
- Write the alternate version

Iterative Reverse

```
(define (reverse-iter lst)
  (reverse-iter-helper lst null))

(define (reverse-helper l r)
  (if (null? l)
      r
      (reverse-helper (cdr l)
                      (cons (car l) r))))
```

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

List-ref

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

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

```
(define (list-range a b)
  (if (> a b)
      null
      (cons a
            (list-range (+ a 1)
                        b))))
```

Max-list

- Write max-list, which takes in a list of numbers and returns the largest 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
```

Max-list

```
(define (max-list lst)
  (if (null? (cdr lst))
      (car lst)
      (let ((m (max-list (cdr lst))))
        (if (> m (car lst))
            m
            (car lst)))))
```

Data Abstraction

- Scheme Provides us with a set of data types
- We may want to create more useful 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 these data types
- Define a **point** data structure as an example
  - Point has coordinates x and y

Constructor

- Procedure to create the data structure

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

Selectors

- Get the pieces out of the data structure
- Usually correspond to the arguments for the contructor
- What selectors does a point need?

Contract

- 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

- Once a data abstraction is defined, it doesn't matter how a point is stored
- Only constructor and selectors are used

```
(car (make-point 5 7))
```

Abstraction Violation

Add-point

- Write a procedure that take two points at (x1,y1) and (x2,y2) and creates a new point at (x1+x2, y1+y2)

Left-of?

- Write a procedure that takes two points, p1 and p2, and returns #t if 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?

Segment-length

- Write a procedure that computes the length of a segment