Finding the length of a list

• Define a procedure \texttt{length}

• What’s the recursive case?

• What’s the base case?

Checking for the empty list

• Base case is an empty list

• Check for it using the \texttt{null?} Predicate

\begin{verbatim}
  (def len lst)
  (if (null? lst)
       0
       (+ 1 (len (cdr lst))))
\end{verbatim}
**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)`

```lisp
(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

```lisp
(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

```lisp
(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 \((a, b : a \leq b)\), and returns a list containing the numbers from \(a\) to \(b\) inclusive.

\[
\begin{align*}
\text{list-range 1 5} & \quad \text{Value: (1 2 3 4 5)} \\
\text{list-range 2 5} & \quad \text{Value: (2 3 4 5)} \\
\text{list-range 42 42} & \quad \text{Value: (42)} \\
\text{list-range 207 5} & \quad \text{Value: ()}
\end{align*}
\]

List-range

\[
\text{(define (list-range a b)} \\
\quad \text{(if (> a b)} \\
\quad\quad \text{null)} \\
\quad\quad \text{(cons a)} \\
\quad\quad \text{(list-range (+ a 1) b))})
\]

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.

\[
\begin{align*}
\text{max-list (list 1)} & \quad \text{Value: 1} \\
\text{max-list (list 1 3 5)} & \quad \text{Value: 5} \\
\text{max-list (list 2 56 8 43 21)} & \quad \text{Value: 56}
\end{align*}
\]

Max-list

\[
\text{(define (max-list lst)} \\
\quad \text{(if (null? (cdr lst)} \\
\quad\quad \text{(car lst)} \\
\quad\quad \text{(let ((m (max-list (cdr lst)))} \\
\quad\quad\quad \text{(if (> m (car lst))} \\
\quad\quad\quad\quad m)} \\
\quad\quad\quad \text{(car lst)))})
\]

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

\[
\text{(define (make-point x y)}
\text{(list x y))}
\]

Selectors

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

Contract

- Selectors and constructors must be written to enforce a contract:

\[
\text{(get-x (make-point 5 7)) => 5}
\text{(get-y (make-point 5 7)) => 7}
\]

Abstraction Violation

\[
\text{(car (make-point 5 7))}
\text{Abstraction Violation}
\]

Add-point

- Write a procedure that take two points at \((x_1,y_1)\) and \((x_2,y_2)\) and creates a new point at \((x_1+x_2, y_1+y_2)\)

Left-of?

- Write a procedure that takes two points, \(p_1\) and \(p_2\), and returns \#t if \(p_1\) is to the left of \(p_2\).
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