CMPU 101 § 55

Computer Science I

Problem-Solving and Abstraction

Prof. Jonathan Gordon

Lecture 8

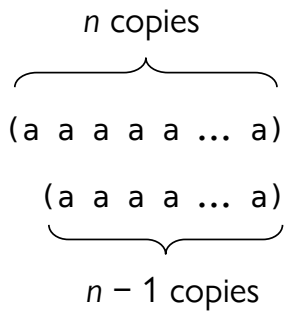
Recursively constructing lists

Problem: Make a list with *n* copies of something

```
(define replicate
  (lambda (item n)
    ...?...))
```

```
> (replicate 'foo 0)
'()
> (replicate 'foo 1)
'(foo)
> (replicate 'foo 2)
'(foo foo)
> (replicate 'foo 3)
'(foo foo foo)
```

Constructing solution to larger problem from solution to smaller one



Recursive definition of **replicate**

```
(define replicate
  (lambda (item n)
    (if (<= n 0)
        '()
        (cons item
              (replicate item (- n 1))))))
```

Problem: Remove the negative numbers from a list

```
(define remove-negatives
  (lambda (lst) ...?...))
```

```
> (remove-negatives '())
()
> (remove-negatives '(-7))
()
> (remove-negatives '(7))
(7)
> (remove-negatives '(-3 2 4 -5 6))
(2 4 6)
```

Constructing solution to larger problem from solution to smaller one

Case 1: `first` = p (positive)

Answer for `rest` = (p_1 p_2 p_3 p_4 … p_n)

Return: (p p_1 p_2 p_3 p_4 … p_n)

Case 2: `first` = n (negative)

Answer for `rest` = (p_1 p_2 p_3 p_4 … p_n)

Return: (p_1 p_2 p_3 p_4 … p_n)

Recursive definition of **remove-negatives**

```
(define remove-negatives
  (lambda (lst)
    (cond
      ;; Base case: reached the end of lst
      ((null? lst) '())

      ;; Recursive case 1:
      ;;   First element is negative; skip it.
      ((negative? (first lst))
       (remove-negatives (rest lst)))

      ;; Recursive case 2:
      ;;   First element is positive; include it.
      (else
       (cons (first lst)
             (remove-negatives (rest lst)))))))
```

Problem: Removing specified elements from a list

```
(define remove-all
  (lambda (item lst)
    ...?...))
```

```
> (remove-all 'a '())
()
> (remove-all 'a '(z a z))
(z z)
> (remove-all 'a '(a z a))
(z)
> (remove-all 'a '(a a a))
()
```

Constructing solution to larger problem from solution to smaller one

Case 1: `lst` = $(x_0\ x_1\ x_2\ \ldots\ x_n)$ where x_0 = `item`

Answer for `rest` = $(y_1\ y_2\ \ldots\ y_n)$

Return: $(y_1\ y_2\ \ldots\ y_n)$

Case 2: `lst` = $(x_0\ x_1\ x_2\ \ldots\ x_n)$ where $x_0 \neq$ `item`

Answer for `rest` = $(y_1\ y_2\ \ldots\ y_n)$

Return: $(x_0\ y_1\ y_2\ \ldots\ y_n)$

Recursive definition of **`remove-all`**

```
(define remove-all
  (lambda (elt lst)
    (cond
      ;; Base case:
      ;;   lst is empty; return ()
      ((null? lst) '())
      ;; Recursive case 1:
      ;;   first elt of lst is supposed to be removed.
      ((equal? elt (first lst))
       (remove-all elt (rest lst)))
      ;; Recursive case 2:
      ;;   first elt of lst should NOT be removed
      (else
       (cons (first lst)
             (remove-all elt (rest lst)))))))

(tester '(remove-all 5 '(1 5 2 5 3 5 4 5)))

(tester '(remove-all 'a '(a b a c a d a z)))
```

What if we want to remove only the first instance of the specified element from a list?

Recursive definition of **`remove-first`**

```
(define remove-first
  (lambda (elt lst)
    (cond
      ;; Base case 1:
      ;;   lst is empty; return ()
      ((null? lst) '())

      ;; Base case 2:
      ;;   first elt of lst is supposed to be removed; we're done!
      ((equal? elt (first lst))
       (rest lst))

      ;; Recursive case:
      ;;   first elt of lst should NOT be removed
      (else
       (cons (first lst)
             (remove-first elt (rest lst)))))))

(tester '(remove-first 5 '(1 5 2 5 3 5 4 5)))

(tester '(remove-first 'a '(a b a c a d a z)))
```

Take every second element of a list

```
(define every-second-element
  (lambda (lst)
    ...?...))
```

```
> (every-second-element '())
'()

> (every-second-element '(a))
'()

> (every-second-element '(a b))
'(b)

> (every-second-element '(1 2 3 4 5 6 7))
'(2 4 6)
```

Constructing solution to larger problem from solution to smaller one

Argument = (a_1 **b_1** a_2 **b_2** a_3 **b_3** … a_n **b_n**)

`first` = a_1

`rest` = (**b_1** a_2 **b_2** a_3 **b_3** … a_n **b_n**)

`rest` of `rest` (`cddr`) = (a_2 **b_2** a_3 **b_3** … a_n **b_n**)

Answer for `rest` of `rest` (`cddr`) = (**b_2** **b_3** … **b_n**)

Return: (**b_1** **b_2** **b_3** … **b_n**)

Recursive definition of **`every-second-element`**

```
(define every-second-element
  (lambda (lst)
    (cond ((null? lst) '())
          ((null? (rest lst)) '())
          (else
           (cons (first (rest lst))
                 (every-second-element
                   (rest (rest lst))))))))
```

Recursive predicates

When writing predicates that test whether something holds of all elements of a list – or at least *one* element of a list – you can often make the recursive function call with an `and` or `or`.

Problem: Write a function that returns `#t` if every element of a list is a positive number.

```
(define all-positive?
  (lambda (lst)
    ;; Base case: List is empty; return #t
    (or (null? lst)
        ;; Recursive case: first element positive...
        (and (number? (first lst))
             (positive? (first lst))
             ;; ... and so are the rest.
             (all-positive? (rest lst))))))

(tester '(all-positive? '(3 8 2 4 9 6)))

(tester '(all-positive? '(3 8 2 -4 9 6)))
```

We can modify `all-positive?` to return `#t` if *any* of the numbers in the list is positive.

```
(define any-positive?
  (lambda (lst)
    ;; Base case: List is empty; return #f
    (and (not (null? lst))
         ;; Recursive case: first element is positive...
         (or (and (number? (first lst))
                  (positive? (first lst)))
             ;; ... or one of the rest of them is.
             (any-positive? (rest lst))))))

(tester '(any-positive? '(-3 -2 #t #f () -8)))

(tester '(any-positive? '(-3 -2 #t #f 4 -8)))
```


Acknowledgments

This lecture incorporates material from:

Tom Ellman

Luke Hunsberger