**Computer Science I**  
**Week 6**

Professor Jenny Walter  
October 2 and 3

---

**Terminology**

- **Flat recursion**: Recursion over the top-level items of a list (items that are not inside a nested list).

  First top-level item on list _first_ treated as a unit and recursive step applies same procedure to the _rest_ of the list.

---

**Terminology**

- **Numeric recursion**: Recursion over all integers from _n_ down to 1.

  First procedure applied to input integer _n_ and subsequent recursive calls on _n-1_, _n-2_, _n-3_, …, 1.

---

**Problem definition**: Create a recursive procedure to multiply a number by itself _n_ times

```
(define (power base exp) …?…)
```

Welcome to DrRacket

```
> (power 2 0)
1
> (power 2 1)
2
> (power 2 2)
4
> (power 2 3)
8
```

---

**Constructing Solution to Larger Problem from Solution to Smaller One**

- **Nesting level**: Top level items in a list have nesting level 1. E.g., (a b c).

  Items not in parentheses have nesting level 0. E.g., 'a 'b 'c.

\[
\begin{align*}
\mathbf{b^n} &= \mathbf{b} \times \mathbf{b^{n-1}} \\
\mathbf{b^0} &= \mathbf{1} \\
\end{align*}
\]
Recursive definition of power

(define (power base exp)
  (if (= exp 0)
      1
      (* base (power base (- exp 1)))))

• Note that in this example, the actual argument substituted for exponent decreases by one on each recursive call until the base case (exponent = 0) is reached. (The value entered for exponent in this function must be positive…why?.)

Tail-recursive definition of power using local

(define (power base exp)
  (local
   [(define (pow-acc exp acc)
         (if (= exp 0)
             acc
             (pow-acc (- exp 1) (* base acc))))
    (pow-acc exp 1))

• Note that there is no need to pass base into local function because it does not change and is visible everywhere inside function power.
Problem definition: Remove all negative numbers from a list of numbers

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

Welcome to DrRacket
> (remove-negatives '() )
empty
> (remove-negatives '(-7) )
empty
> (remove-negatives '(-3 2 4 -5 6) )
(2 4 6)

Recursive Definition of REMOVE-NEGATIVES

(define (remove-negatives lst)
  (cond [(empty? lst) empty]
        [(< (first lst) 0) ;; first is negative
         (remove-negatives (rest lst))] ;; omit it in return list
        [else (cons (first lst) ;; first is positive
                    ;; include it in return list
                    (remove-negatives (rest lst)))]))

• Here, there is a base case and 2 recursive cases.

Stopping conditions & inductive case

Base Case 1: Empty list.
Result: Return #f

Base Case 2: (equal? (first input_list) item)
Result: Return #t

Inductive case: (member? item (rest input_list))

Examine the FIRST element

Case 1: first = p (positive)
answer for rest = (p1 p2 p3 p4 ... pN)
Return: (p p1 p2 p3 p4 ... pN)

Case 2: first = p (negative)
answer for rest = (p1 p2 p3 p4 ... pN)
How many base cases: ? How many inductive case(s): ?

Define a predicate to determine whether a specified symbol is in a list of symbols

(define (member? item lst) ...?...)

Welcome to DrRacket
> (member? 'a '() )
#f
> (member? 'a '(z a z) )
#t
> (member? 'a '(a z a) )
#t
> (member? 'a '(x y z) )
#t

Recursive definition of member?

(define (member? item lst)
  (cond [(empty? lst) #f]
        [(equal? item (first lst)) #t]
        [else (member? item (rest lst))]))
Alternate definition of member?

\[
\text{(define (member? item lst)} \rightarrow \text{(and (not (empty? lst)) (or (equal? item (first lst)) (member? item (rest lst)))))}
\]

Define a procedure to insert an item to the left of another item in a list

\[
\text{(define (insert-left new old lst) …?…)}
\]

Welcome to DrRacket

> (insert-left 'a 'b '())
empty

> (insert-left 'b 'c d b )
(list a b c d a b)

> (insert-left 'cute 'kitty '(the kitty ran away) )
'(the cute kitty ran away)

Stopping conditions & inductive cases

Arguments = x y '((x y1 y2 y3 ... yN))
Case 1: y = x
Return: (x y1 y2 y3 y4 ... yN)
Recursive case: (cons x (cons x (return from rest of list)))
Case 2: y ≠ x
Return: (x y1 y2 y3 y4 ... yN)
Recursive case: (cons x (return from rest of list))
Base case: empty list, return empty

Recursive definition of insert-left

\[
\text{(define (insert-left new old lst)} \rightarrow \text{(cond (empty? lst) '()) [(equal? (first lst) old) (cons new (insert-left new old (rest lst)))) [else (cons (first lst) (insert-left new old (rest lst))))])}
\]

Define a procedure deepen-1 that wraps a pair of parenthesis around each top-level item in a list

\[
\text{(define deepen-1 (lambda (lst) …?…))}
\]

Welcome to DrRacket

> (deepen-1 '())
empty

> (deepen-1 '(b c d b))
(list (b) (c) (d) (b))

> (deepen-1 '((0) 10))
(((0)) (10))

Stopping conditions & inductive case

Arguments = '(x y1 y2 y3 ... yN)
Return: (x y1 y2 y3 y4 ... yN)
Base case: empty
Recursive step:
(cons (first lst) (return from rest of list))
Recursive definition of deepen-1

(define (deepen-1 lst)
  (if (empty? lst)
      empty
      (cons (cons (first lst) empty) (deepen-1 (rest lst)))))

How would we change deepen-1 to create a list from any input list?

(define (deepen-1 ls)
  (cond;; if ls is empty, return empty list
  ([(empty? ls) empty]
   ;; if ls is a consed list
   ([(cons? ls) (cons (cons (first ls) '()) (deepen-1 (rest ls)))]
    [else (cons ls empty)]))

Define a recursive procedure to return all but the nth element of a list

(define (all-but-n n lst) ...?...)

Welcome to DrRacket
> (all-but-n 1 '())
empty
> (all-but-n 2 '(a b c d b))
(list b d b)
> (all-but-n 4 '(9 10))
(list 9 10)

Cases

Case 1: length of list \(\geq\) n
Return whole list minus the nth element.

Case 2: length of list < n
Return whole list.

E.g. (all-but-n 4 '(a b c d e f))
Could you write this with string functions and no recursion?

Procedure all-but-n

(define (all-but-n n lst)
  (cond
   [(empty? lst) empty]
   [(= n 1) (rest lst)]
   [(> n 1) (cons (first lst) (all-but-n (- n 1) (rest lst)))]))

Base cases: \(n = 1\) or lst is empty
Recursive case: (cons (car lst) (result from rest))

Alternate procedure all-but-n

(define (all-but-n n lst)
  (if (not (empty? lst))
      (if (= n 1)
          (rest lst)
          (cons (first lst) (all-but-n (- n 1) (rest lst))))))
Yet another alternate all-but-n

(define (all-but-n n lst)
  (if (empty? lst)
      empty
      (if (= n 1)
          (rest lst)
          (cons (first lst) (all-but-n (- n 1) (rest lst))))))