Lists of Lists

CS135 Lecture 11

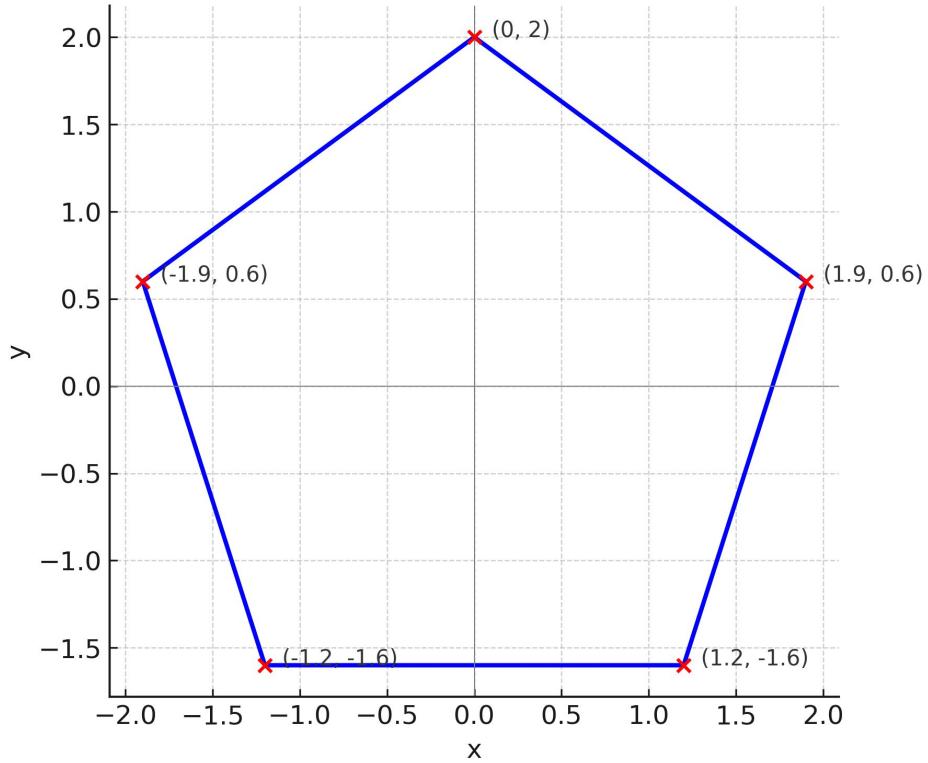
Reminder: Rules of recursion

1. Change at least one argument closer to termination while recurring
2. When recurring on a natural number use `(sub1 n)` and test termination with `zero?`
3. When recurring on a list use `(rest lst)` and test termination with `empty?`
4. Arguments that aren't involved in recursion can change.

Almost all problems in CS135 can be solved under these rules.

L11.00 Polygons

A polygon is a list of points




```
(list
 (list 1.6 0.6)
 (list 1.2 -1.6)
 (list -1.2 -1.6)
 (list -1.9 0.6)
 (list 0 2))
```

In our examples we will assume all polygons are *simple*, i.e., the segments don't cross

How to compute the perimeter?

From L05: Helper functions for points

```
;; a Point is a (x,y) point in the Cartesian coordinate system
;; a Point is a (cons Num (cons Num empty)
;;
;; mk-point consumes an x and y coordinate and produces a Point
;; mk-point: Num Num -> Point
(define (mk-point x y) (cons x (cons y empty)))

;; get-x consumes a Point and produces its x coordinate
;; get-x: Point -> Num
(define (get-x point) (first point))

;; get-y consumes a Point and produces its y coordinate
;; get-y: Point -> Num
(define (get-y point) (first (rest point)))
```

Distance between two points: $d = \sqrt{(x_1 - x_0)^2 + (y_1 - y_0)^2}$

```
;; distance between two points
;; d: Point Point -> Num
(define (d p0 p1)
  (sqrt (+ (sqr (- (get-x p1) (get-x p0)))
           (sqr (- (get-y p1) (get-y p0))))))

(check-within (d (mk-point 1 5) (mk-point 4 1))
              5 0.00001)
(check-within (d (mk-point -1 -1) (mk-point -2 -2))
              (sqrt 2) 0.00001)
```

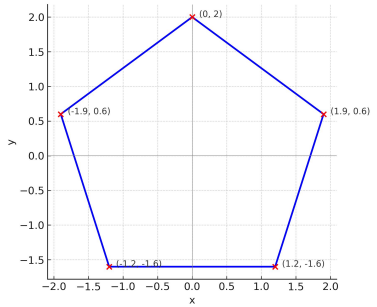
Add the lengths of segments between successive points

```
;; Add successive line segments in a list of point
;; add-segments: (listof Point) -> Num
(define (add-segments lop)
  (cond [(or (empty? lop) (empty? (rest lop))) 0]
        [else (+ (d (first lop) (second lop))
                 (add-segments (rest lop)))]))
(define square
  (list (mk-point 1 1) (mk-point -1 1)
        (mk-point -1 -1) (mk-point -1 1)))
(check-within (add-segments square) ??? 0.00001)
```

What's the problem?

Computing the perimeter of a polygon

```
;; Compute the perimeter of a polygon
;; perimeter: (listof Point) -> Num
;; Requires: points form a simple polygon
(define (perimeter polygon)
  (+ (d (first polygon) (last polygon))
     (add-segments polygon)))
(check-within (perimeter square) 8 0.00001)
(check-within
 (perimeter
  (list (list 1.6 0.6) (list 1.2 -1.6)
        (list -1.2 -1.6) (list -1.9 0.6) (list 0 2)))
 11.43 0.01)
```



Perimeter of a polygon: Alternative solution

In the previous solution, we had to “traverse” the list twice: once to add up the segments and once to find the last point. Can we do better?

Idea: A version of add-segment with the initial point as an extra argument.

```
;; Add successive line segments with wrapping
;; add-segments/wrap: (listof Point) -> Num
(define (add-segments/wrap p0 lop)
  (cond [(empty? lop) 0]
        [(empty? (rest lop)) (d p0 (first lop))]
        [else (+ (d (first lop) (second lop))
                 (add-segments/wrap p0 (rest lop)))]))
```

Perimeter of a polygon: Alternative solution

```
;; Compute the perimeter of a polygon
;; perimeter/v2: (listof Point) -> Num
;; Requires: points form a simple polygon
(define (perimeter/v2 polygon)
  (add-segments/wrap (first polygon) polygon))

(check-within (perimeter/v2 square) 8 0.00001)
(check-within
 (perimeter/v2
  (list (list 1.6 0.6) (list 1.2 -1.6)
        (list -1.2 -1.6) (list -1.9 0.6) (list 0 2)))
 11.43 0.01)
```

Which version is “better”?

L11.01 Tables

Example: A multiplication table as a list of lists

×	1	2	3	4	5	6	7	8	9	10	11	12
1	1	2	3	4	5	6	7	8	9	10	11	12
2	2	4	6	8	10	12	14	16	18	20	22	24
3	3	6	9	12	15	18	21	24	27	30	33	36
4	4	8	12	16	20	24	28	32	36	40	44	48
5	5	10	15	20	25	30	35	40	45	50	55	60
6	6	12	18	24	30	36	42	48	54	60	66	72
7	7	14	21	28	35	42	49	56	63	70	77	84
8	8	16	24	32	40	48	56	64	72	80	88	96
9	9	18	27	36	45	54	63	72	81	90	99	108
10	10	20	30	40	50	60	70	80	90	100	110	120
11	11	22	33	44	55	66	77	88	99	110	121	132
12	12	24	36	48	60	72	84	96	108	120	132	144

Since this is CS135, we start at 0:

```
(list
 (list 0 0 0 0 0 0 0 0 0 0 0 0 0)
 (list 0 1 2 3 4 5 6 7 8 9 10 11 12)
 (list 0 2 4 6 8 10 12 14 16 18 20 22 24)
 (list 0 3 6 9 12 15 18 21 24 27 30 33 36)
 (list 0 4 8 12 16 20 24 28 32 36 40 44 48)
 (list 0 5 10 15 20 25 30 35 40 45 50 55 60)
 (list 0 6 12 18 24 30 36 42 48 54 60 66 72)
 (list 0 7 14 21 28 35 42 49 56 63 70 77 84)
 (list 0 8 16 24 32 40 48 56 64 72 80 88 96)
 (list 0 9 18 27 36 45 54 63 72 81 90 99 108)
 (list 0 10 20 30 40 50 60 70 80 90 100 110 120)
 (list 0 11 22 33 44 55 66 77 88 99 110 121 132)
 (list 0 12 24 36 48 60 72 84 96 108 120 132 144))
```

Generating n×m multiplication table

```
;; Generate an nXm multiplication table
;; times: Nat Nat -> (listof (listof Nat))
(define (times n m)(...)


(check-expect (times 3 5)
              (list (list 0 0 0 0 0 0)
                    (list 0 1 2 3 4 5)
                    (list 0 2 4 6 8 10)
                    (list 0 3 6 9 12 15)))
```

Generating one row of an n×m multiplication table

```
;; Generate a times table row from n*(m-i) to n*m
;; times/row: Nat Nat Nat -> (listof Nat)
(define (times/row n m i)
  (cond [(zero? i) (list (* n m))]
        [else (cons (* n (- m i)) (times/row n m (sub1 i)))]))

(check-expect (times/row 3 5 5)
              (list 0 3 6 9 12 15))
```

We follow the Rules of Recursion by recursion on `i`.

Keeping `n` and `m` fixed allows us to generate the numbers in increasing order.

Generating several rows of an n×m multiplication table

```
;; Generate m-times table rows from (n - i) to n
;; times/rc: Nat Nat Nat -> (listof (listof Nat))
(define (times/rc n m i)
  (cond [(zero? i) (list (times/row n m m))]
        [else (cons (times/row (- n i) m m)
                    (times/rc n m (sub1 i)))]))

(check-expect (times/rc 3 5 1)
              (list (list 0 2 4 6 8 10)
                    (list 0 3 6 9 12 15)))
```

Generating n×m multiplication table

```
;; Generate an nXm multiplication table
;; times: Nat Nat -> (listof (listof Nat))
(define (times n m)
  (times/rc n m n))

(check-expect (times 3 5)
              (list (list 0 0 0 0 0 0)
                    (list 0 1 2 3 4 5)
                    (list 0 2 4 6 8 10)
                    (list 0 3 6 9 12 15)))
```

Example: Indexing into a table

```
(define (index-table n m table)...)

(check-expect (index-table 2 3 (list
                                (list 1 2 3 4 5)
                                (list 'a 'b 'c 'd 'e 'f)
                                (list 'u 'v 'w 'x 'y 'z)
                                (list 'red 'green 'blue)))
              'x)
(check-expect (index-table 100 100 empty) empty)
```

If **n** or **m** are larger than lengths of the corresponding lists, we produce **empty**.

Note that **empty** is a list of lists.

Recall **`index`** from lecture L10

```
(define (index n lst)
  (cond [(empty? lst) empty]
        [(zero? n) (first lst)]
        [else (index (sub1 n) (rest lst))]))

(check-expect (index 3 (list 'a 'b 'c 'd 'e)) 'd)
(check-expect (index 100 (list 'a 'b 'c 'd 'e)) empty)
```

How do we extend to a table?

Indexing into a table

```
;; index into a list of lists
;; index-table: Nat Nat (listof (listof Any)) -> Any
(define (index-table n m table)
  (index m (index n table)))

(check-expect (index-table 2 3 (list
                                (list 1 2 3 4 5)
                                (list 'a 'b 'c 'd 'e 'f)
                                (list 'u 'v 'w 'x 'y 'z)
                                (list 'red 'green 'blue)))
              'x)
(check-expect (index-table 100 100 empty) empty)
```

L11.02 Subsets

Sets of symbols

A data definition for a set of symbols is as follows:

```
;; A Set is a (listof Sym)
;; Requires: symbols must be unique
```

Given a set, write a function that produces a list of all its subsets.

```
(define (subsets set) ...)

(check-expect (subsets (list 'a 'b 'c))
              (list (list 'c 'b 'a) (list 'b 'a)
                    (list 'c 'a) (list 'a) (list 'c 'b)
                    (list 'b) (list 'c)))
```

Producing a list of subsets from a set

Breaking the problem down into smaller problems…

1. Suppose we have a set of subsets and we want to add a new symbol. We can write a function to expand the set of subsets. For each subset we produce two subsets, one with the symbol and one without.
2. Start with the empty set (which is a set of subsets of the empty set). Using the function in #1, add each symbol in turn… Maybe use an accumulator for the expanded set of subsets?
3. Write a wrapper for the accumulative recursive in #2.

#1 Expanding a list of subsets with a new symbol

```
;; Expand a list of subsets with a new symbol
;; expand/sym: Sym (listof Set) -> (listof Set)
;; Requires: new symbol must not appear in the subsets
(define (expand/sym sym subsets)
  (cond [(empty? subsets) (cons (list sym) empty)]
        [else (cons (cons sym (first subsets))
                    (cons (first subsets)
                          (expand/sym sym (rest subsets))))]))
(check-expect
  (expand/sym 'i (list (list 'a 'b) (list 'a) (list 'b)))
  (list (list 'i 'a 'b) (list 'a 'b) (list 'i 'a) (list 'a)
        (list 'i 'b) (list 'b) (list 'i)))
```

#2 Expanding a list of subsets with a list of new symbols

```
;; Expand a list of subsets with a list of new symbols
;; expand/list: (listof Sym) (listof Set) -> (listof Set)
;; Requires: new symbols must not appear in the subsets
(define (expand/list lst subsets)
  (cond [(empty? lst) subsets]
        [else (expand/list (rest lst)
                           (expand/sym (first lst) subsets))]))

(check-expect (expand/list (list 'a 'b 'c) empty)
              (list (list 'c 'b 'a) (list 'b 'a)
                    (list 'c 'a) (list 'a) (list 'c 'b)
                    (list 'b) (list 'c)))
```

#3 Wrapper for the accumulative recursion

```
;; Produce a list of subsets from a Set
;; subsets: Set -> (listof Set)
(define (subsets set)
  (expand/list set empty))

(check-expect (subsets (list 'a 'b 'c))
              (list (list 'c 'b 'a) (list 'b 'a)
                    (list 'c 'a) (list 'a) (list 'c 'b)
                    (list 'b) (list 'c)))
```

Lecture 11 Summary

L11: You should know

How the techniques you learned in previous lectures (L00 to L10) generalize to lists of lists.

In some sense, there's nothing "new" in this lecture. The lecture is just series of examples demonstrating how lists can contain lists, in addition to values like **`Nat`**, **`Int`**, **`Num`**, **`Sym`**, etc., and how to work with these lists.

L11: Allowed constructs

Newly allowed constructs:
none

Previously allowed constructs:

```
( ) [ ] + - * / = < > <= >= ;
abs acos add1 and append asin atan check-expect check-within
cond cons cons? cos define e else empty empty? exp expt
false first inexact? integer? length list list? log max min
not number? or pi quotient rational? remainder rest reverse
second sin sqr sqrt sub1 symbol? symbol=? tan third true
zero?
listof Any anyof Bool Int Nat Num Rat Sym
```