

# Generative Recursion

---

- Most of the functions we have written recurse on data:
  - process elements of a list (until empty).
- Such functions are "structurally recursive"
  - same pattern is followed regardless of the exact data.
- Functions that recurse based on something other than the structure of the data are called "generative recursive".

# Algorithms and Generative Recursion

---

- Consider the TSP homework. Most of us did a brute-force approach:
  - generate all possible tours.
  - find cost of each tour.
  - find tour with minimum cost.
- We could consider other approaches that depend on the actual tour costs:
  - find lowest cost leg of a tour that involves a city we have not yet visited (until no cities remain unvisited).

# Ball animation

---

- We want to move a ball until it hits the edge of the window.
  - we need to recurse until some condition is met that has nothing to do with the structure of the data.
  -

# Support code

---

```
(define-struct ball (x y delta-x delta-y))

(define (draw-and-clear a-ball)
  (and
    (draw-solid-disk
      (make-posn (ball-x a-ball) (ball-y a-ball))
      5 'red)
    (sleep-for-a-while DELAY)
    (clear-solid-disk
      (make-posn (ball-x a-ball) (ball-y a-ball))
      5 'red)))
```

# More Support code

---

```
(define (move-ball a-ball)
  (make-ball (+ (ball-x a-ball) (ball-delta-x a-ball))
            (+ (ball-y a-ball) (ball-delta-y a-ball))
            (ball-delta-x a-ball)
            (ball-delta-y a-ball)))
```

;; Dimension of canvas

```
(define WIDTH 100)
```

```
(define HEIGHT 100)
```

```
(define DELAY .1)
```

# What we need to write

---

- We need some way of knowing whether the ball has reached the edge of the window:
  - `(out-of-bounds? a-ball)` returns true/false
  - `(move-until-out a-ball)`
    - animates the ball movement and stops once the ball hits the edge of the window.

# out - of - bounds?

---

- The ball is *out of bounds* if either:
  - x value is  $\leq 0$  or x value is  $>$  than WIDTH
    - (not ( $\leq 0$  (ball-x a-ball) WIDTH))
  - y value is  $\leq 0$  or y value is  $>$  HEIGHT
    - (( $\leq 0$  (ball-y a-ball) HEIGHT))

# out-of-bounds?

---

```
(define (out-of-bounds? a-ball)
  (not
   (and
    (<= 0 (ball-x a-ball) WIDTH)
    (<= 0 (ball-y a-ball) HEIGHT) ) ) )
```

# move-until-out

---

```
(define (move-until-out a-ball)
  (cond
    [(out-of-bounds? a-ball) ...]
    [else ...]))
```

need to draw the ball, move the  
ball, and call move-until-out

# move-until-out

---

```
(define (move-until-out a-ball)
  (cond
    [(out-of-bounds? a-ball) true]
    [else
     (and
      (draw-and-clear a-ball)
      (move-until-out (move-ball a-ball)))]))
```

# Exercise

---

- Start with the code "ball.scm"
  - animates a ball bouncing around inside a window.
  - the function never stops!
- Extend the code to support multiple balls
  - a list of balls.
  - `(bounce-lots-o-balls alob)`
  - Consider using abstract functions like `map` !

# Sorting

---

- We can develop sorting *algorithms* based on structural or generative recursion.
  - scan the list  $n$  times, moving the largest element to the end each time.
  - scan the list  $n$  times, swapping adjacent elements when it makes sense.
- Sorting can be based on generative recursion.
  - number of steps involved may depend on the *values*, not just on the structure of the data.

# Generative Sorting: Quicksort

---

- *Divide and conquer*:
  - create a number of *subproblems*.
  - solve each *subproblem*.
  - combine solutions to *subproblems*.
- There must be some end to the recursion!
  - there must be some kind of *subproblem* for which we don't apply the same strategy.

# Quicksort

---

- The first element in the list is called the *pivot*.
- Create a list of all elements that are less than the pivot.
  - sort this list using quicksort.
- Create a list of all elements that are greater than the pivot.
  - sort this list using quicksort.
- Put everything together to create a sorted list.

# Example: ' ( 5 2 8 1 6 3 10 )

---

- pivot is 5.
- create list of all elements less than 5:
  - ' ( 2 1 3 ) sort this to get ' ( 1 2 3 )
- create list of all elements greater than 5:
  - ' ( 8 6 10 ) sort this to get ' ( 6 8 10 )
- (append ' ( 1 2 3 ) ' ( 5 ) ' ( 6 8 10 ) )

# Create list of elements less than some threshold

---

- We can use the abstract function `filter`:

```
(define (smaller-items alon thres)
  (local
    ((define (lessthantres x) (< x thres)))
    (filter lessthantres alon)))
```

- We can define `greater-items` in the same way.

# Quicksort base case

---

- We need to end the recursion: in what case do we not use the divide and conquer strategy?
  - an empty list is easy to sort.
  - a list with only one element is also easy.
  - some quicksort implementations use another sorting algorithm once the list size is below some threshold.
- We can use "an empty list" for now...

# quick-sort

---

```
(define (quick-sort alon)
  (cond
    [(empty? alon) empty]
    [else
     (append
      (quick-sort (smaller-items alon (first alon)))
      (list (first alon))
      (quick-sort (larger-items alon (first alon)))
      )]))
```

# quick-sort vs. quicksort

---

- The scheme `quicksort` function uses the algorithm we just implemented, although it does not force comparisons to be numeric.
  - give `quicksort` a function that can tell whether one *thing* is smaller than another.
  - used to create lists of elements smaller/greater than the pivot.

# Exercises

---

- What happens if there are duplicates?
  - `(quick-sort '(1 3 4 3 1 2))`
  - **FIX THIS!**
- Revise our quick-sort function so that it works like quicksort.
  - `(quick-sort somelist less-than-operator)`  
`(quick-sort '(1 4 2 8 5 9) <)`