Administrative Announcements

Exam

- Covers through middle of Friday's lecture, plus lab lectures
- Take home available Wednesday, due Monday at 5pm
- Closed notes, closed book
- Covers through Intermezzo 4 in the book

- Wednesday night labs as normal

Review

Last class

- We began working with *generative recursion*
  - → New paradigm,
  - → Radical departure from structural recursion

- Built a version of Hoare's classic algorithm, quicksort
  - → Recursion in quicksort comes from insight, not data analysis
  - → Employs a "divide and conquer" strategy
  - → Termination relies on monotonic reduction in subproblem size

Sorting a List of Numbers

Hoare's quicksort

```
;; qsort: list-of-numbers -> list-of-numbers
;; Purpose: return a list containing the input numbers, in ascending order
(define (qsort alon)
   (cond
      [(empty?  alon)           empty ]
      [(cons?    alon)
       (local [  (define pivot (first alon))
                 (define (smaller-items alon threshold)
                    (filter (lambda (n) (< n threshold)) alon))
                 (define (larger-items alon threshold)
                    (filter (lambda (n) (> n threshold)) alon))]
              (append  (qsort (smaller-items alon pivot))      } divide and
                       (list pivot)                            } conquer
                       (qsort (larger-items alon pivot)))]
    )
)
```

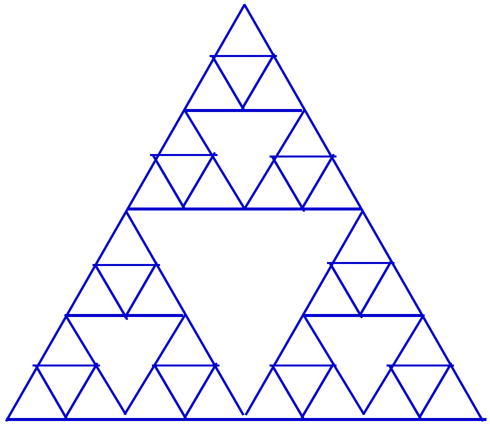

Review

Last class

- We began working with generative recursion
  - → New paradigm,
  - → Radical departure from structural recursion
- Built a version of Hoare's classic algorithm, quicksort
  - → Recursion in quicksort comes from insight, not data analysis
  - → Employs a "divide and conquer" strategy
  - → Termination relies on monotonic reduction in subproblem size
- Today, let's look at another example of generative recursion

Another Example of Generative Recursion

Sierpinski Triangles



4th Sierpinski triangle

and so on …

A Sierpinski triangle is a kind of fractal …
a geometric figure that has the same structure at different scales

How would we generate Sierpinski triangles?

Sierpinski Triangles

Generating Sierpinski triangles

- We run this program for its side effect – drawing lines
- Sierpinski: point point point -> something
  - → Don't really care what it returns, as long as it draws triangle
  - → Make it boolean
  - → Recursively draw i^{th} Sierpinski triangle until sides are too small
- Need a representation for a point

```
;; a posn is a
;; (make-posn x  y) where x and y are numbers
(define-struct posn (x  y))
```

Name "posn" is from the book

- Leads to a contract

  Sierpinski: posn posn posn -> boolean

Sierpinski Triangles

Assume three helper functions

draw-triangle: posn posn posn -> boolean

→ Draws lines: $posn_1$->$posn_2$, $posn_2$->$posn_3$, and $posn_3$->$posn_1$

too-small?: posn posn posn -> boolean

→ Returns true if $posn_1$ is too close to $posn_2$

midpoint: posn posn -> posn

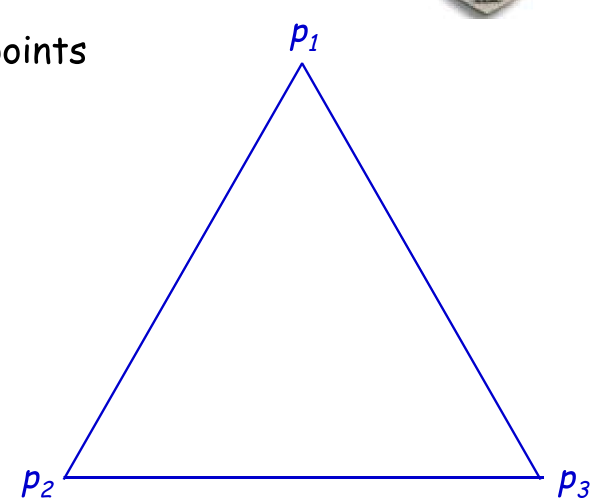
→ returns midpoint of line from two points

- At this point, we are not that much closer to drawing the actual triangles …
  - → Let's get down to details

Sierpinski Triangles

The Basic Idea

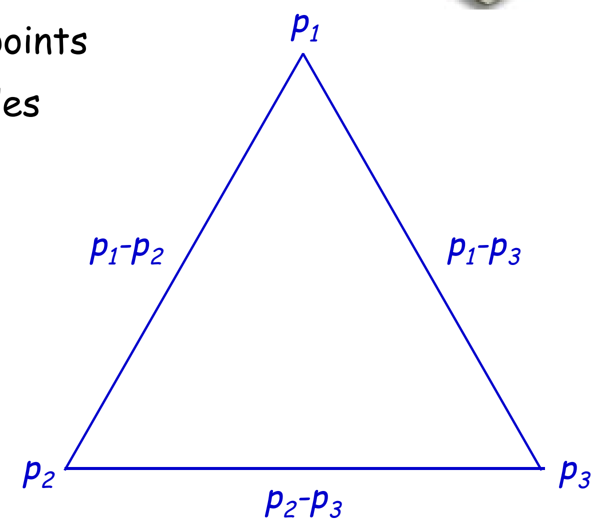
1. Draw a triangle for three points



Sierpinski Triangles

The Basic Idea

1. Draw a triangle for three points
2. Find midpoints of three sides

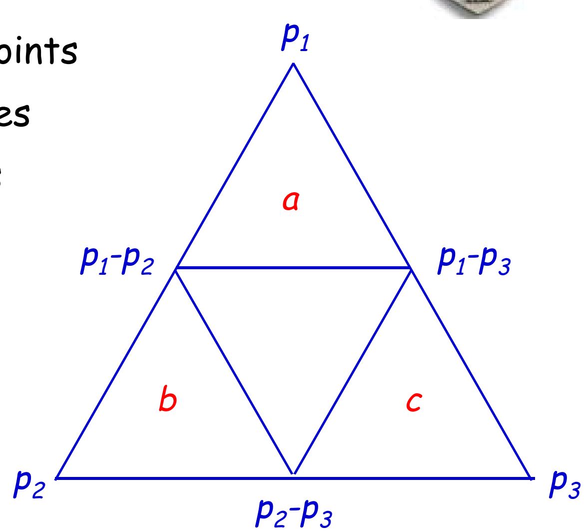


Divide

Sierpinski Triangles

The Basic Idea

1. Draw a triangle for three points
2. Find midpoints of three sides
3. Recur in the outer triangles
   - a, b, and c



Divide and conquer

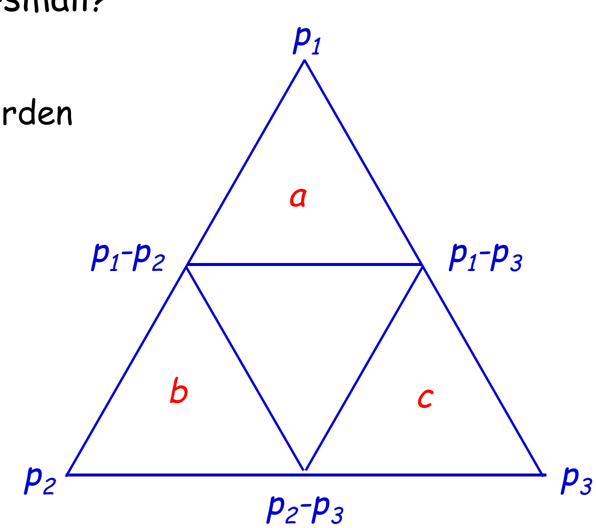
Sierpinski Triangles

```
;; sierpinski: posn  posn  posn -> boolean
;; Purpose: draw Sierpinski triangle to a resolution defined by
;;       the (external) function too-small?
(define (sierpinski  p1  p2  p3)
   (cond
     [(too-small? p1 p2 p3)     true]   ;; value forced by use of and
     [else
       (local [ (define p1-p2 (midpoint p1 p2))            ;; Step 2
                (define p1-p3 (midpoint p1 p3))            ;; Step 2
                (define p2-p3 (midpoint p2 p3)) ]          ;; Step 2
             (and    (draw-triangle p1  p2  p3)            ;; Step 1
                     (sierpinski p1  p1-p2  p1-p3)         ;; Step 3
                     (sierpinski p1-p2  p2  p2-p3)         ;; Step 3
                     (sierpinski p2-p3  p1-p3  p3)         ;; Step 3
             ) ) ]
   ))
```


Sierpinski Triangles

Termination

- Recursion cuts off when too-small? returns true
- Some tolerance built into too-small?
  - → Related to screen resolution
  - → Account for computational burden



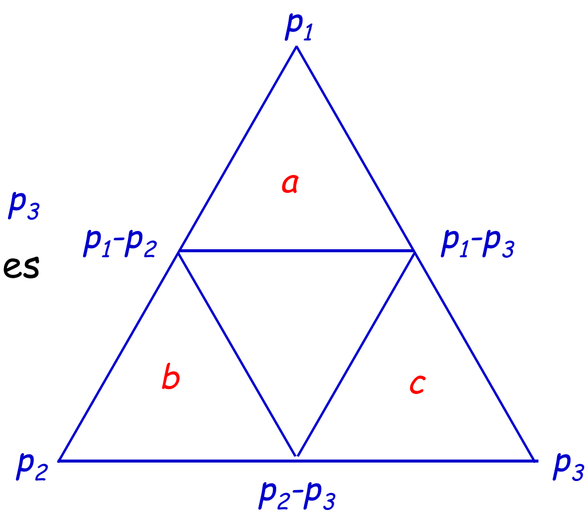
Sierpinski Triangles

Termination

- Recursion cuts off when too-small? returns true
- Some tolerance built into too-small?

Correctness

- Draws an outer triangle p_1, p_2, p_3
- Recurs on three corner triangles
  a, b, and c
- Fairly simple argument



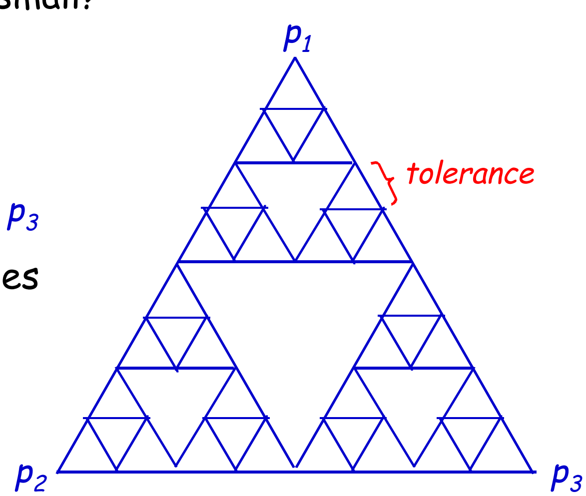
Sierpinski Triangles

Termination

- Recursion cuts off when too-small? returns true
- Some tolerance built into too-small?

Correctness

- Draws an outer triangle p_1, p_2, p_3
- Recurs on three corner triangles
  a, b, and c
- Fairly simple argument
- Leads to this $\Rightarrow$

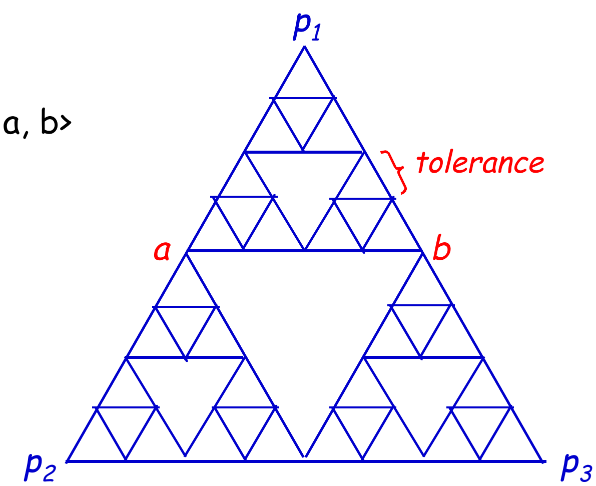


So, we're done … or are we?

Sierpinski Triangles

What about efficiency? (*even if this is 210*)

- How many times does $\langle p_1,p_2 \rangle$ get drawn?
  - → Once for $\langle p_1,p_2,p_3 \rangle$
  - → Again for $\langle p_1, a, b \rangle$
  - → Again for subtriangles of $\langle p_1, a, b \rangle$
  - → Again …
  - → Until too-small? returns true
  - → $\log_2(\text{length of } \langle p_1,p_2 \rangle)$



- What about other lines?
  - → Same behavior
  - → $\langle a,b \rangle$ gets drawn whole, in halves, in quarters, …

Sierpinski Triangles

We have seen this kind of thing before

- Remember max-of-list ?
- We used a local to escape the problem
  - → Exponential time to linear time

Can we do a similar thing with sierpinski?

- What do we need to preserve?
- How can we preserve it?

Sierpinski Triangles

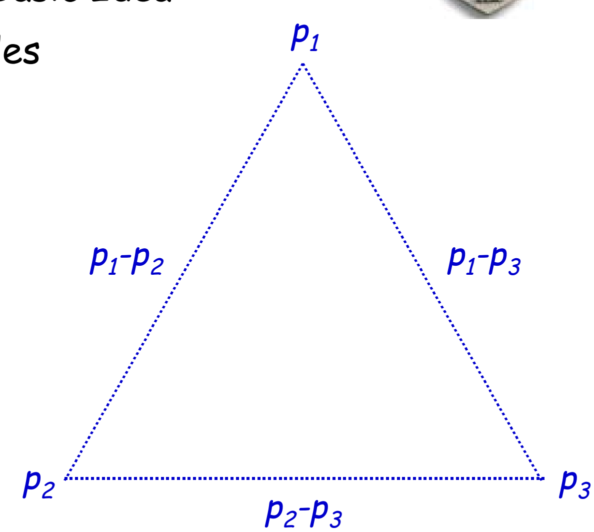
```
;; sierpinski: posn  posn  posn -> boolean
;; Purpose: draw Sierpinski triangle to a resolution defined by
;;       the (external) function too-small?
(define (sierpinski  p1  p2  p3)
   (cond
     [(too-small? p1 p2 p3)    true]   ;; value forced by use of and
     [else
       (local [ (define p1-p2 (midpoint p1 p2))             ;; Step 2
                (define p1-p3 (midpoint p1 p3))             ;; Step 2
                (define p2-p3 (midpoint p2 p3)) ]           ;; Step 2
             (and   (draw-triangle p1  p2  p3)              ;; Step 1
                    (sierpinski p1  p1-p2  p1-p3)           ;; Step 3
                    (sierpinski p1-p2  p2  p2-p3)           ;; Step 3
                    (sierpinski p2-p3  p1-p3  p3)           ;; Step 3
             ) ) ]
   ))
```

The problem (pointing to `(draw-triangle p1  p2  p3)`)

Sierpinski Triangles

A Different Approach to the Basic Idea

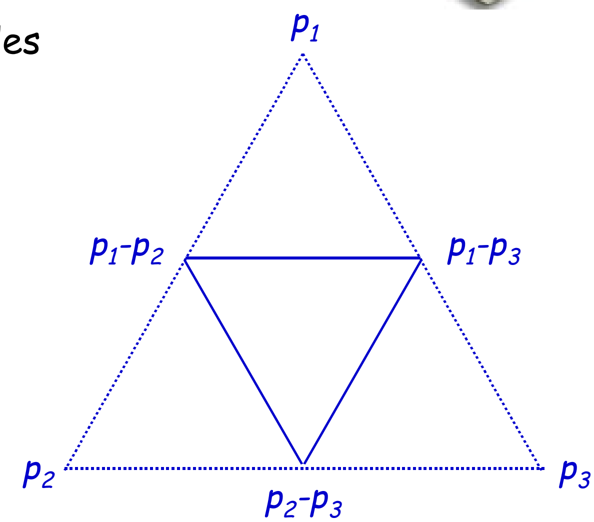
1. Find midpoints of three sides



Sierpinski Triangles

A Different Approach to the Basic Idea

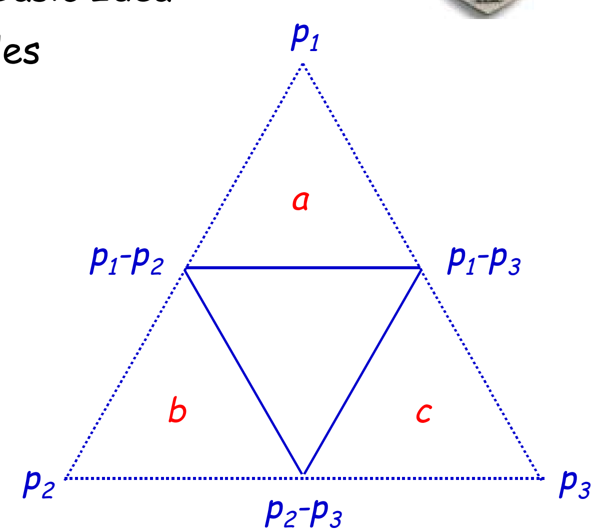
1. Find midpoints of three sides
2. Draw the inside triangle



Sierpinski Triangles

A Different Approach to the Basic Idea

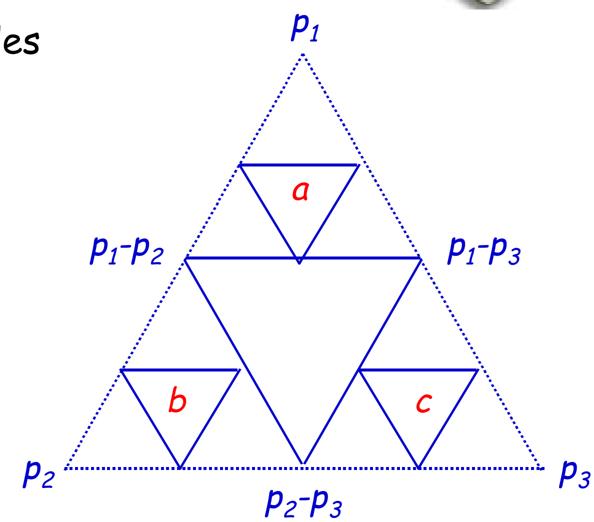
1. Find midpoints of three sides
2. Draw the inside triangle
3. Recur on a, b, and c



Sierpinski Triangles

A Different Approach to the Basic Idea

1. Find midpoints of three sides
2. Draw the inside triangle
3. Recur on a, b, and c



- *And so on …*

- But, …
- This never draws the outside
- Need to handle that separately

Sierpinski Triangles

```
;; sierpinski: posn  posn  posn -> boolean
(define (sierpinski p1 p2 p3)
   (local
      [(define (sierp  p1  p2  p3)       ;; workhorse routine for recurrence
         (cond
            [(too-small? p1 p2 p3)        true]
            [else
              (local [ (define p1-p2 (midpoint p1 p2))
                       (define p1-p3 (midpoint p1 p3))
                       (define p2-p3 (midpoint p2 p3)) ]
                      (and (draw-triangle p1-p2 p1-p3 p2-p3)
                           (sierp p1  p1-p2  p1-p3)
                           (sierp p1-p2  p2  p2-p3)
                           (sierp p2-p3  p1-p3  p3) ) ) ]
      (and (draw-triangle p1 p2 p3)    ;; draw current triangle
           (sierp p1 p2 p3))            ;; and recur
   ))
```

Wrap Up

- Used same divide and conquer strategy
- Built one solution *and* analyzed it
- Consequences of generating too many solutions
  - → Needed to revise our solution to ensure reasonable behavior
  - → Think about big picture issues

- Next Class
  - → Exam will be handed out
  - → Templates for Generative Recursion