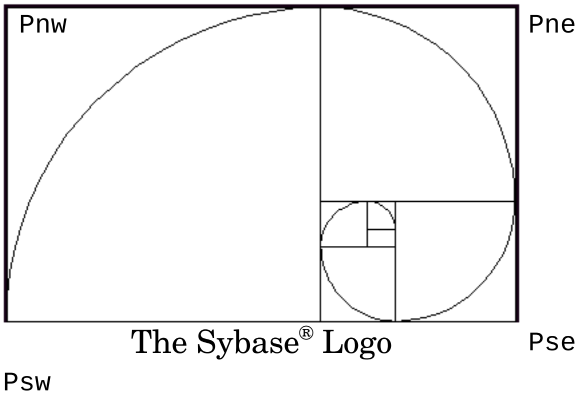
Name: ______________________________

Question 1 – Those clever graphic designers at Sybase®, Inc. (20 pts; 30 min.)

Those of you who have seen the Sybase® logo (shown on the right) realize it's really a fractal in disguise. Your goal is to draw it. Fortunately, *all* of the hard math will be done for us.

First of all, take a careful look at the fractal. Note that the bold lines on the outside left, top and right are only drawn once! Let's assume they're taken care of for us and we only have to draw the inner lines and arcs.

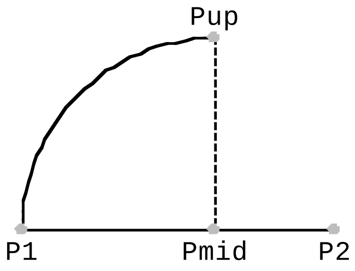


The Sybase® Logo

Your friend provided a really nice graphic interface for you to use; instead of using `x` and `y` coordinates, you just use *points*. The code, as you'll see, becomes much cleaner.

```
;; Look at the figure below for an example
(draw-lineP P1 P2) ;; Draws a line from point P1 to point P2
(draw-arcP  P1 P2) ;; Draws a 90° arc from P1 to Pup centered around Pmid
(get-Pmid   P1 P2) ;; Return Pmid, a little more than half-way from P1 to P2
(get-Pup    P1 P2) ;; Return Pup, a point "above" the P1-P2 line
                    ;; such that the triangle P1-Pmid-Pup is a right triangle.
```

The arc on the right was created with a call to `(draw-arcP P1 P2)`. The arc angle is 90° and is drawn from `P1` to `Pup` as if one end of a compass were at `Pmid`.



a) Fill in the blanks to complete the `sybase` procedure. Use figure to the right to help you understand the temporary variables `Pmid` and `Pup`. (15 points)

```
(define (sybase P1 P2 n)
  (if (= n 0)
      (draw-lineP P1 P2)
      (let ((Pmid (get-Pmid P1 P2))
            (Pup  (get-Pup  P1 P2)))

        _________________________________________________

        _________________________________________________

        _________________________________________________ )))
```

b) Provide the call to `sybase` that generated the fractal in the Sybase® logo at the top. Assume the corner point labels in the diagram are already defined for us to use. (5 points)

```
(sybase  _____  _____  _____ )
```

Question 5 – A ccumulate in tree's clothing… (20 points; 30 minutes)

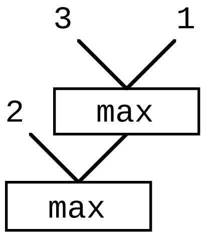
A brilliant CS3 student realizes that the way `accumulate` walks down a sentence can be thought of as a tree, just upside-down! For example, the following call:

```
(accumulate max ‘(2 3 1))
```

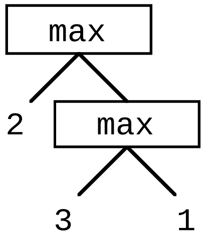
can be visually depicted as the upside-down tree on the immediate right with the bottom-most `max` as the root, 2 as its first child, etc… When we flip it rightside-up, we get the tree on the far right. We'll call these specific trees *accumulate-trees*, or *a-trees* for short. Note that all a-tree inner nodes have *exactly two children*.

Let's now build the a-tree in the far upper-right using the tree interface:

```
(define *max-2-3-1-a-tree*
  (make-tree max
     (list (make-leaf 2)
           (make-tree max
                      (make-leaves
                         ‘(3 1))))))
```



How the `accumulate` calls to `max` on `(2 3 1)` work.



Thinking of the same `accumulate` call as a tree. Cool, huh?!

```
;; Recall the tree interface
(define (make-tree datum children)
  (cons datum children))
(define (make-leaf datum)
  (make-tree datum '()))
(define (make-leaves datum-list)
  (map make-leaf datum-list))
(define (datum tree)
  (car tree))
(define (children tree)
  (cdr tree))
(define (leaf? tree)
  (null? (children tree)))
```

a) If `max` (typed in by itself to scheme) evaluates to `#[procedure max]`, what does scheme return if you just type in `*max-2-3-1-a-tree*` (as defined above)? (10 pts)

b) These a-trees are quite interesting intellectually, but useless if the desired result is the **value** returned by the original `accumulate`. Your job is to fill in the blanks (we started it for you) to write `a-tree->val`, a function that takes in an a-tree and returns the value that would have been calculated by the original `accumulate`. E.g., `(a-tree->val *max-2-3-1-a-tree*)` ➔ `3`. You may assume the original sentence passed into `accumulate` had at least 2 words in it. You **may not use** `begin`, `eval` or `apply`. *We consider this a very hard problem, by far the hardest on the exam.* (10 pts)

```
(define (a-tree->val a-tree)
  (if (leaf? a-tree)
      (datum a-tree)

      __________________________________________________

      __________________________________________________ ))
```

Question 6 – Fractals on the brain (19 points; 30 min)

You *loooooove* fractals so much that you see them absolutely everywhere you go… even in non-graphic functions like the following:

```
(define (fun n)
  (if (= n 0)
      '()
      (cons (fun (- n 1))
            (fun (- n 1)))))
```

a) Given `fun` as defined above, what does Scheme return if you type: `(fun 1)`? (4 pts)

b) Draw the box-and-pointer diagram for `(fun 2)`. (5 points)

c) How many calls to `cons` will there be for `(fun 5)`? (5 points)

d) In Question 4, you might have noticed you were using data structures (namely, *points*) without even being told how they were implemented! **In one word**, how were you able to do this? Hint – this is one BIG IDEA of the course. (5 points)

You're done! Have a great summer break!!