;READ THE FOLLOWING VERY CAREFULLY BEFORE PROCEEDING ON TO THE REST OF THE EXAM.

For the rest of the exam, you will be working on problems related to provided problem description and data definition. Your work for the subsequent problems must use our data definitions.

PROBLEM DESCRIPTION:

In a drawing program, it is common for the program to represent the image in terms of shapes and their relation, rather than in terms of the actual displayed image. That is what makes it possible for the program to support commands that re-align images for example.

Below we are giving you data definitions that might be part of a VERY simple graphical editor. In problem 3 you will annotate the data definitions. In problems 4, 5, 6, and 7 you will design functions that operate on that data.

;Problem 3

On the data definitions below, you should do four things:
- On the TYPE COMMENTS, draw any appropriate reference, self-reference, and mutual reference arrows and label each arrow with R, SR or MR.
- On the TEMPLATES, draw any appropriate natural helper, natural recursion, and natural mutual recursion arrows and label NH, NR or NMR.
- Number each arrow with a number (like 1, 2, 3) to show which type comment arrows correspond to which template arrows. Arrows that correspond should have the same number.

```
;; Alignment is one of:
;; - "above"
;; - "beside"                    R1
;; - "overlay"
;; interp. the alignment by which to combine images


(define-struct sq (side color))
(define-struct ci (radius color))
(define-struct aligned (style loi))
;; BasicImage is one of:
;; - false
;; - (make-sq Natural String)
;; - (make-ci Natural String)
;; - (make-aligned Alignment ListOfBasicImage)
;; interp. an empty (false) image
;;         or square with side length (in pixels) and color
;;         or circle with radius (in pixels) and color
;; MR3     or multiple images set in one of the alignments
;;                          MR2   (above, beside, overlay)


;; ListOfBasicImage is one of:   SR4
;; - empty
;; - (cons BasicImage ListOfBasicImage)
;; interp. a list of images
```

```
(define EMPTY-IMAGE false)
(define IMAGE-SQUARE (make-sq 5 "red"))
(define IMAGE-CIRCLE (make-ci 12 "blue"))

(define IMAGE-ABOVE (make-aligned "above" (list IMAGE-SQUARE IMAGE-CIRCLE)))
(define IMAGE-BESIDE (make-aligned "beside" (list IMAGE-SQUARE IMAGE-CIRCLE)))
(define IMAGE-OVERLAY (make-aligned "overlay" (list IMAGE-SQUARE IMAGE-CIRCLE)))

(define BIG-IMAGE (make-aligned "overlay" (list IMAGE-SQUARE
                                                IMAGE-ABOVE
                                                IMAGE-BESIDE
                                                IMAGE-OVERLAY)))

#;
(define (fn-for-bi bi)

  (local [(define (fn-for-alignment a)

            (cond [(string=? a "above") ...]

                  [(string=? a "beside") ...]

                  [(string=? a "overlay") ...]))              NH1

          (define (fn-for-bi bi)

            (cond [(false? bi) ...]

                  [(sq? bi) (... (sq-side bi)

                                 (sq-color bi))]

                  [(ci? bi) (... (ci-radius bi)

                                 (ci-color bi))]
   NMR3
                  [else (... (fn-for-alignment (aligned-style bi))

                             (fn-for-lobi (aligned-loi bi)))]))
                                  NMR2
          (define (fn-for-lobi lobi)

            (cond [(empty? lobi) (...)]

                  [else                      NR4

                   (... (fn-for-bi (first lobi))

                        (fn-for-lobi (rest lobi)))]))]

    (fn-for-bi bi)))
```

; Problem 4

Complete the design of a function that consumes a basic image and produces the number of squares that are contained in that image.

We have provided the signature, purpose, check-expects, and template. You will need to complete the function body by NEATLY editing the template provided.

```
;; BasicImage -> Natural
;; counts the number of squares in that image
(check-expect (count-squares false) 0)
(check-expect (count-squares IMAGE-SQUARE) 1)
(check-expect (count-squares IMAGE-CIRCLE) 0)
(check-expect (count-squares IMAGE-OVERLAY) 1)
(check-expect (count-squares BIG-IMAGE) 4)

; Template from BasicImage (+ MR cycle), encapsulated by local

#;
(define (count-squares bi)

  (local [~~(define (fn-for-alignment a)~~

            ~~(cond [(string=? a "above") ...]~~

                  ~~[(string=? a "beside") ...]~~

                  ~~[(string=? a "overlay") ...]))~~

          (define (fn-for-bi bi)

            (cond [(false? bi) 0 ...]

                  [(sq? bi) (. 1 . ~~(sq-side bi)~~

                                ~~(sq-color bi)~~)]

                  [(ci? bi) (. 0 . ~~(ci-radius bi)~~

                                ~~(ci-color bi)~~)]

                  [else
                   ~~(... (fn-for-alignment (aligned-style bi))~~

                        (fn-for-lobi (aligned-loi bi))~~)~~]))

          (define (fn-for-lobi lobi)

            (cond [(empty? lobi) (. 0 .)]

                  [else

                   (. + . (fn-for-bi (first lobi))

                        (fn-for-lobi (rest lobi)))]))]

    (fn-for-bi bi)))
```

```
; Problem 5

Design a function that takes a basic image and produces the width of that
image.

Remember, the width of a square is just the side length. The width of a
circle is 2 * radius.

When a group of images are placed in an "above" alignment, the width of the
combined image will be the width of the widest image in the group. This is
also the case for an "overlay" alignment.

When a group of images are placed in a "beside" alignment, the width of the
combined image will be the total width of all images.

You must provide the signature, purpose, and check-expects.

We have provided a copy of the encapsulated template. Do your function
definition by NEATLY editing the template we are giving you -- edit it,
do not rewrite it!
```

```
;; BasicImage → Natural
;; [illegible] produce width of given image
(check-expect (width EMPTY-IMAGE) 0)
(check-expect (width IMAGE-SQUARE) 5)
(check-expect (width IMAGE-CIRCLE) 24)
(check-expect (width IMAGE-ABOVE) 24)
(check-expect (width IMAGE-BESIDE) 29)
(check-expect (width IMAGE-OVERLAY) 24)
(check-expect (width BIG-IMAGE) 29)


  ; template from types


#;
```

```
(define (fn-for-bi bi)
  (local [(define (fn-for-alignment a)        ; → (Natural Natural → Natural)
            (cond [(string=? a "above") max]
                  [(string=? a "beside") .+.]
                  [(string=? a "overlay") max]))

          (define (fn-for-bi bi)   ; →.Natural
            (cond [(false? bi) .0.]
                  [(sq? bi) (— (sq-side bi)
                               ~~(sq-color bi)~~)]
                  [(ci? bi) (.*. (ci-radius bi) 2)
                               ~~(ci-color bi)~~)]
                  [else (foldr (fn-for-alignment (aligned-style bi))
                               0
                               (fn-for-lobi (aligned-loi bi)))]))

          (define (fn-for-lobi lobi)   ; → (listof Natural)
            (cond [(empty? lobi) empty]
                  [else
                   (cons (fn-for-bi (first lobi))
                         (fn-for-lobi (rest lobi)))]))]
    (fn-for-bi bi)))
```

; Problem 6

Design an abstract fold function for BasicImage. You must include the signature, purpose, one check-expect that produces a copy of the argument, the origin of the template statement, and the function defintion.

We have provided a copy of the template. You should NEATLY edit it for your function definition.

```
;; (Natural String -> Y) (Natural String -> Y) (X Z -> Y) (Y Z -> Z)
        X  X  X  Y  Z    -> Y

;; abstract fold function for BasicImage

(check-expect (fold-bi make-sq make-ci make-aligned cons
                       "above" "beside" "overlay" false empty BIG-IMAGE)

              BIG-IMAGE)


; template from types

#;              fold-bi
(define (~~fn-for-bi~~ c1 c2 c3 c4 b1 b2 b3 b4 b5 bi)

  (local [(define (fn-for-alignment a)                ; -> X
            (cond [(string=? a "above") b1]
                  [(string=? a "beside") b2]
                  [(string=? a "overlay") b3]))

          (define (fn-for-bi bi)                      ; -> Y
            (cond [(false? bi) b4]
                  [(sq? bi) (c1 (sq-side bi)
                                (sq-color bi))]
                  [(ci? bi) (c2 (ci-radius bi)
                                (ci-color bi))]
                  [else (c3 (fn-for-aligment (aligned-style bi))
                            (fn-for-lobi (aligned-loi bi)))]))

          (define (fn-for-lobi lobi)                  ; -> Z
            (cond [(empty? lobi) (b5)]
                  [else
                   (c4 (fn-for-bi (first lobi))
                       (fn-for-lobi (rest lobi)))]))]
    (fn-for-bi bi)))
```