; As fun as a binary of monkeys.

(require picturing-programs)

; There's a function rectangle from picturing-programs.
; It takes four arguments: two numbers for width and height,
; a string "solid" or "outline", and a string name of a color.
; It produces an image.

; We can note that information with a datatype "contract":
; rectangle : number number string string -> image

; An example use of the function.
(rectangle 20 40 "solid" "red")

; We can document the datatype of the name of a value as well.
; monkey : image

(define monkey )

; one : image
(define one (scale 1/6
    (crop-right monkey
        (// (image-width monkey)
            3)))))

; That used a few functions from picturing-programs.
; Here are their contracts:
; scale : number image -> image
; crop-right : image number -> image
; image-width : image -> number
; / : number number -> number

; Let's make a place-holder image for "no monkey".
; zero : image
(define zero
    (rectangle (image-width one) 2 "solid" "black"))

; Let's stack the compound monkeys vertically, i.e. use
; above instead of beside.
; The contract for the above and beside functions are:
;    above : image image ... -> image
;    beside : image image ... -> image

; two : image
(define two (above one one))
; four : image
(define four (above two two))
; eight : image
(define eight (above four four))

; Let's make our own function, that takes four images and makes
; one with them side-by-side, sitting on the same "baseline"

; There's a function beside/align from picturing-programs.
;    beside/align : string image image ... -> image

; Here's a comparison of it with plain beside.
;
; a-star : image
(define a-star (star 25 "solid" "blue"))
; big-triangle : image
(define big-triangle (triangle 100 "outline" "purple"))
;
(beside a-star big-triangle)
(beside/align "top" a-star big-triangle)
(beside/align "middle" a-star big-triangle)
(beside/align "bottom" a-star big-triangle)

; EXERCISE: what are the contracts for star and triangle?

; row : image image image image image -> image
(define (row image-0 image-1 image-2 image-3)
    ; Original version:
    ; (beside/align "bottom" image-0 image-1 image-2 image-3)
    ; Decided later to join them in reverse order:
    (beside/align "bottom" image-3 image-2 image-1 image-0))

(row zero zero zero zero)
; (beside/align "bottom" zero zero zero zero one)
(row zero zero zero one)
(row zero zero two zero)
(row zero zero two one)
(row zero four zero zero)
(row zero four zero one)
(row zero four two zero)
(row zero four two one)

; EXERCISE: put in the remaining eight.
Why bother with making our own function rather than using beside/align with "bottom"?

Less to write, less error-prone.

Easier to read and understand. Although it's not much of a problem here, since they're all together and lined up, scanning something repetitive to see which things are always the same and which ones vary is a cognitive cost that adds up in complex situations (when you can least afford it!).

And we can tell whether other uses of beside/align are meant to be for similar reasons, or coincidences.

Easier to change: it's not an accident that we did a similar thing eight times, and when we wanted to reverse the order it was only necessary to change in one place: the definition of row.

If we tried to do it with a search-and-replace of beside/align it could accidentally change other uses that had nothing to do with the intended one.

(beside/align "bottom" zero zero zero zero zero)
(beside/align "bottom" zero zero zero one)
(beside/align "bottom" zero zero two zero)
(beside/align "bottom" zero zero two one)
(beside/align "bottom" zero four zero zero)
(beside/align "bottom" zero four zero one)
(beside/align "bottom" zero four two zero)
(beside/align "bottom" zero four two one)