Exercise 2.11.

In passing, Ben also cryptically comments: "By testing the signs of the endpoints of the intervals, it is possible to break `mul-interval` into nine cases, only one of which requires more than two multiplications." Rewrite this procedure using Ben's suggestion.

Answer.

Suppose the two intervals we are about to multiply are X and Y. An advisable practice to implement Ben's strategy would be by categorizing all the cases on two layers: the signs of the endpoints of X and those of Y in each case of X. Thus, the nine cases Ben advocated can be express geometrically, as is shown from Figure 1 to Figure 3.

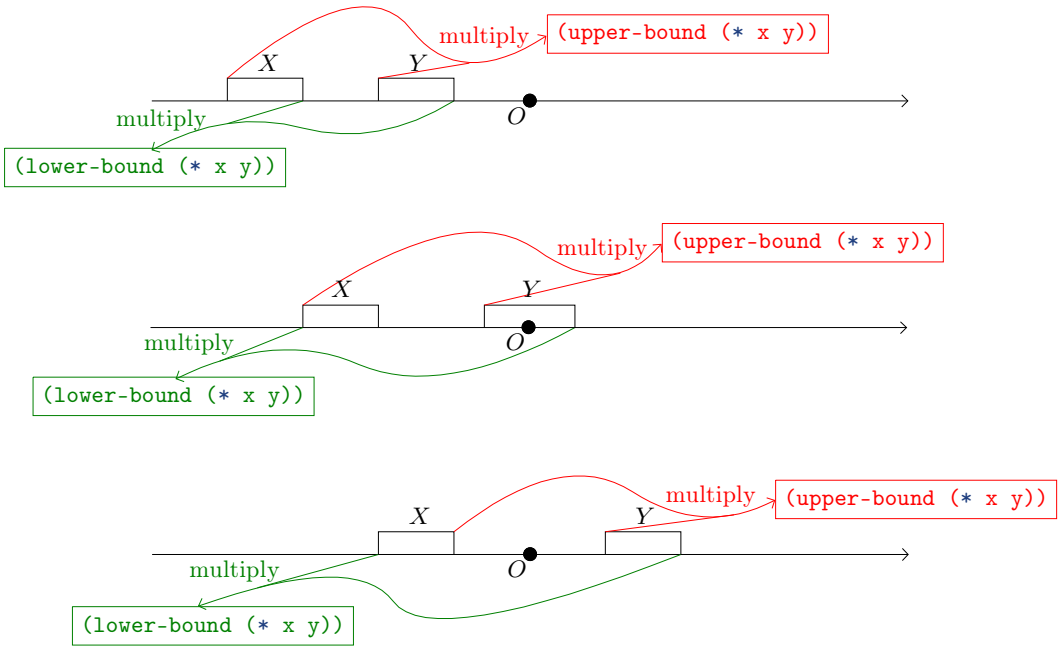

Figure 1. Interval A Locates on The Negtive Part of Number Aixs

Using these graphs, we are now able to rewrite the procedure `mul-interval` as[1]:

```
(define (mul-interval x y)
  (define (locates-negative? i)
    (< (upper-bound i) 0))
  (define (spans-zero? i)
    (and (<= (lower-bound i) 0)
         (>= (upper-bound i) 0)))
  (cond ((locates-negative? x)
         (cond ((locates-negative? y)
```


1. Note that this is a pretty bad way to multiply two intervals, which boggles our minds a lot on redundant cases but is still doubtful in gaining effeciency. Ben ignored that multiplication of intervals, like that of numbers, also satisfies the reflexive property. Hence, what Ben tried to cover are only 6 cases in enumeraion. Even after being improved, this program is still hard to read and mantain.

By comparing this version of `mul-interval` with the original one, we should notice that even using the same data structure, variances on algorithm might cause a world of difference on the complexity of the program. The original one focus on the product of the interval boundaries. Ben, however, cares about a different aspect of this problem: the signs of their boundaries. But this is really trivial, for what really matters is the value of the products of boundaries rather than their signs. Obviously, the original strategy stands on a higher level of abstraction then that of Ben's here, thus makes the program more concise and straightforward.

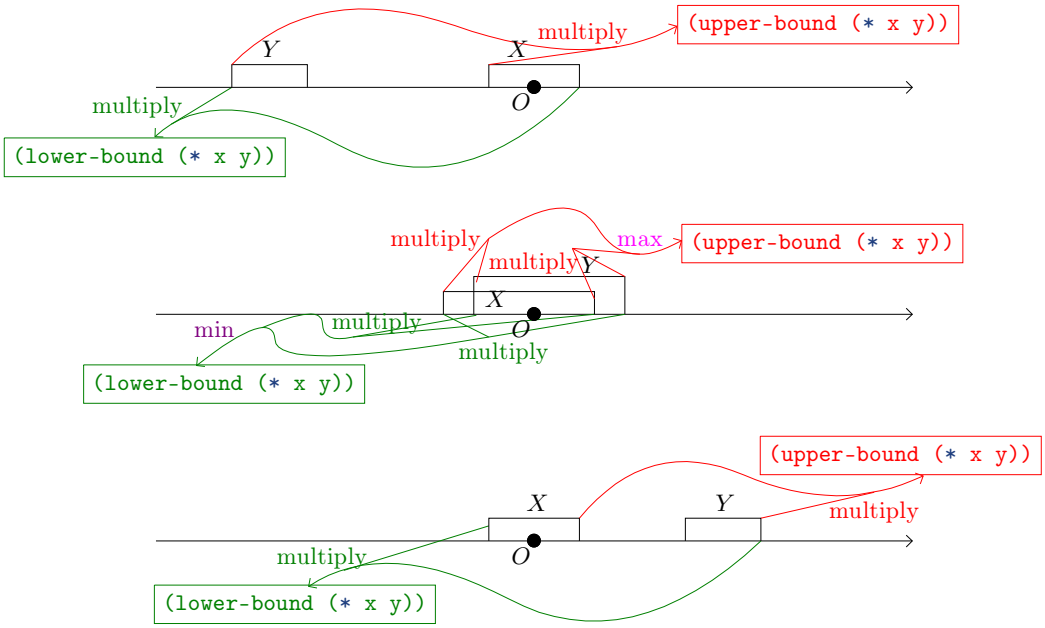

Figure 2. Interval A Spans Zero on The Number Aix

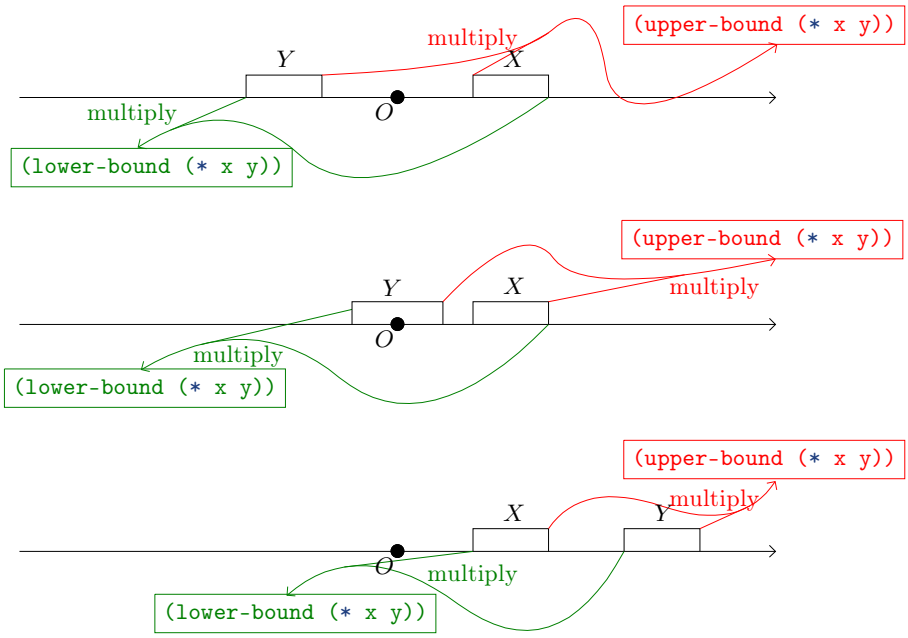

Figure 3. Interval A Locates on The Positive Part of Number Aixs

```
          (make-interval (* (upper-bound x) (upper-bound y))
                         (* (lower-bound x) (lower-bound y))))
        ((spans-zero? y)
         (make-interval (* (lower-bound x) (upper-bound y))
                        (* (lower-bound x) (lower-bound y))))
        (else
          (make-interval (* (lower-bound x) (upper-bound y))
                         (* (upper-bound x) (lower-bound y))))))
  ((spans-zero? x)
   (cond ((locates-negative? y)
          (make-interval (* (upper-bound x) (lower-bound y))
```

```
                    (* (lower-bound x) (lower-bound y))))
       ((spans-zero? y)
        (let ((p1 (* (lower-bound x) (upper-bound y)))
              (p2 (* (upper-bound x) (lower-bound y)))
              (q1 (* (lower-bound x) (lower-bound y)))
              (q2 (* (upper-bound x) (upper-bound y))))
          (make-interval (min p1 p2)
                         (max q1 q2))))
       (else
         (make-interval (* (lower-bound x) (upper-bound y))
                        (* (upper-bound x) (upper-bound y))))))
(else
  (cond ((locates-negative? y)
         (make-interval (* (upper-bound x) (lower-bound y))
                        (* (lower-bound x) (upper-bound y))))
        ((spans-zero? y)
         (make-interval (* (upper-bound x) (lower-bound y))
                        (* (upper-bound x) (upper-bound y))))
        (else
          (make-interval (* (lower-bound x) (lower-bound y))
                         (* (upper-bound x) (upper-bound y)))))))
```