```scheme
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;; Structure and Interpretation of Computer Programs, 2. ed.    ;;;;;
;;;;; Section 1.1, Exercise 1.5                                     ;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;; Student: Abrantes Araújo Silva Filho                           ;;;;;
;;;;; Date: 2019-02-11                                              ;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


;;;; QUESTION:
;;;; Ben Bitdiddle has invented a test to determine whether the
;;;; interpreter he is faced with is using applicative-order
;;;; evaluation or normal-order evaluation. He defines the following
;;;; two procedures:

(define (p) (p))

(define (test x y)
  (if (= x 0)
      0
      y))

;;;; Then he evaluates the expression

(test 0 (p))

;;;; What behavior will Ben observe with an interpreter that uses
;;;; applicative-order evaluation? What behavior will he observe with
;;;; an interpreter that uses normal-order evaluation? Explain your
;;;; answer. (Assume that the evaluation rule for the special form if
;;;; is the same whether the interpreter is using normal or applicative
;;;; order: The predicate expression is evaluated first, and the result
;;;; determines whether to evaluate the consequent or the alternative
;;;; expression.)



;;;; WRITE YOUR ANSWER HERE:
;
; The first DEFINE it's used to create a procedure "p" without any
; parameters. This procedure just return the value of procedure "p",
; that is, procedure p is "recursivelly" returning the value of "p"
; itself.

(define (p) (p))
p
; compound procedure

; The second DEFINE it's used to create a procedure "test" with 2
; arguments, "x" and "y". If the value of x is 0, return 0, else
; return y value.

(define (test x y)
  (if (= x 0)
      0
      y))

(test 0 1)
; 0
```

```scheme
(test 1 1)
; 1

; The "magic" is on the third expression:
(test 0 (p))

; The result of this expression it's dependent on the type of
; substitution model used by the interpreter: applicative-order or
; normal-order.
;
; a) If applicative-order, the interpreter FIRST EVALUATES the
;    parameters, replacing it by the value of the subexpressions. But
;    evaluating (p) just return "p" itself, so the interpreter
;    continues to evaluate and ends in an infinite loop:
;    1) Evaluate (test 0 (p))
;    2) Evaluate subexpression 0, and return 0. Now we have:
;           (test 0 (p))
;    3) Now evaluate subexpression (p): this return the value of
;       procedure p, which is p itself. Now we have:
;           (test 0 (p))
;    4) Woops! It's need to evaluate the (p) subexpression again
;    because of applicative-order. Evaluating (p) again, we have:
;           (test 0 (p))
;    5) And so on... we have an infinite loop. The problem was the
;       applicative-order: THE ARGUMENTS ARE EVALUATED FIRST AND THEN
;       THESE VALUES ARE SUBSTITUTED IN FURTHER EXPRESSION. The
;       important point to note is this:
;
;       APPLICATIVE-ORDER: the arguments are evaluated first, and only
;       after obtaining the value of the subexpression, the
;       substitution continues.
;
; b) If normal-order: the interpreter FIRST EXPAND the procedures and
;    the subexpressions are NOT EVALUATED until it's needed. We have:
;    1) Evaluate (test 0 (p))
;    2) Expand to: (if (= 0 0) 0 (p))
;    3) Now the interpreter evaluates (= 0 0), which is #t, and so the
;       procedure return 0:
;           0
;    4) The important point is:
;
;       NORMAL-ORDER: the interpreter FIRST EXPAND the procedures and
;       DO NOT EVALUATE subexpression until when it's needed.
```