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

(test 0 (p))
```

When Ben uses an interpreter that uses applicative-order evaluation, the interpreter will first evaluate the operator (the test procedure) and then the operands (0 and (p)), before applying the resulting procedure to the resulting arguments. This means the interpreter will evaluate 0 and (p) before proceeding to apply test, which under our assumption of the substitution model of procedure application, would result in their substitution to an if special form. The interpreter can evaluate the expression 0 as normal, then based on the definition of the procedure p, it will evaluate the procedure p to be (p). This is a problem because the test procedure cannot be applied until the operands are evaluated, and the final operand (p) has been evaluated to (p), which requires further evaluation. This only yields (p) again, so if there is an endless loop, then Ben will know that the interpreter uses applicative-order evaluation.

When Ben uses an interpreter that uses normal-order evaluation, the interepter does not evaluate operands until their values are needed. It instead will first substitute the operand expressions for parameters until only primitive operators remain. Therefore, the test procedure will be applied and 0 will be substitued for x and (p) will be substituted for y. This thus yields,

```
(if (= 0 0) 0 (p))
```

Because 0 equals 0, the procedure will evaluate to the consequent, 0, which means the alternative, (p), never needs to be evaluated by the interpreter, thus avoiding the infinite loop problem that was illustrated above. Therefore, if the interpreter yields 0, Ben can be sure that the interpreter uses normal-order evaluation.