A3 - sol.

(win 2023)

① Cannot use short circuit evaluation for XOR because its value is never determined by one argument. Must always evaluate both.

② original code

```
before_loop
while (condition) do
    body
after_loop
```

equivalent code

```
f (before_vars)
    if (not condition) then
        after_loop
        return
    else
        body
        return f (before_vars)

before_loop
f (before_vars)
```

The goal of this exercise is to help thinking recursively, for the purpose of programming in Scheme.

Example (not required for your solution)

```
int p[n], i = 1, count = 0
while (i ≤ n) do
    if p[i] = i then
        count ← count + 1
    i ← i + 1
print (count)
```

```
f(p, n, i, count)
    if (i > n) then
        print (count)
        return
    else
        if p[i] = i then
            count ← count+1
        i ← i+1
        return f(p, n, i, count)

f(p, n, 1, 0)
```

$T \equiv \lambda pq.p$

$F \equiv \lambda pq.q$

$NOT \equiv \lambda pqr.prq$

$XOR \equiv \lambda pq.p\,(NOT\,q)\,q$

③ ⓐ - call by value

$$XOR\ p\ (NOT\ p)$$

$$\equiv (\lambda pq.p\,(NOT\,q)\,q)\ p\ (NOT\ p)$$

$$\equiv (\lambda pq.p\,((\lambda pqr.prq)\,q)\,q)\ p\ ((\lambda pqr.prq)\,p)$$

$$\Rightarrow_\alpha (\lambda pq.p\,((\lambda pkr.prk)\,q)\,q)\ p\ ((\lambda pqr.prq)\,p)$$

$$\overset{A}{\Rightarrow}_\beta (\lambda pq.p\,(\lambda kr.qrk)\,q)\ p\ ((\lambda pqr.prq)\,p)$$

$$\overset{A}{\Rightarrow}_\beta (\lambda q.p\,(\lambda kr.qrk)\,q)\ ((\lambda pqr.prq)\,p)$$

$$\overset{A}{\Rightarrow}_\beta (\lambda q.p\,(\lambda kr.qrk)\,q)\ (\lambda qr.prq)$$

$$\overset{A}{\Rightarrow}_\beta p\,(\lambda kr.(\lambda qr.prq)\,r\,k)\,(\lambda qr.prq)$$

$$\overset{A}{\Rightarrow}_\alpha p\,(\lambda kr.(\lambda qs.psq)\,r\,k)\,(\lambda qr.prq)$$

$$\overset{A}{\Rightarrow}_\beta p\,(\lambda kr.(\lambda s.psr)\,k)\,(\lambda qr.prq)$$

$$\overset{A}{\Rightarrow}_\beta p\,(\lambda kr.pkr)\,(\lambda qr.prq)$$

- <u>call by name</u>

$$XOR\ p\ (NOT\ p)$$

$$\equiv (\lambda pq.p\ (NOT\ q)\ q)\ p\ (NOT\ p)$$

$$\equiv (\lambda pq.p\ ((\lambda pqr.\ p\ r\ q)\ q)\ q)\ p\ ((\lambda pqr.prq)\ p)$$

$$\Rightarrow_\beta (\lambda q.p\ ((\lambda pqr.\ prq)\ q)\ q)\ ((\lambda pqr.prq)\ p)$$

$$\Rightarrow_\beta p\ ((\lambda pqr.prq)\ ((\lambda pqr.prq)\ p))\ ((\lambda pqr.prq)\ p))$$

$$\Rightarrow_\beta p\ (\lambda qr.((\lambda pqr.prq)\ p)\ r\ q\ ((\lambda pqr.\ prq)\ p))$$

$$\Rightarrow_\beta p\ (\lambda qr.(\lambda qr.prq)\ r\ q\ ((\lambda pqr.prq)\ p))$$

$$\Rightarrow_\alpha p\ (\lambda qr.(\lambda qs.\ psq)\ r\ q\ ((\lambda pqr.prq)\ p))$$

$$\Rightarrow_\beta p\ (\lambda qr.(\lambda s.\ psr)\ q\ ((\lambda pqr.prq)\ p))$$

$$\Rightarrow_\beta p((\lambda qr.pqr)\ ((\lambda pqr.prq)\ p))$$

$$\Rightarrow_\beta p\ (\lambda qr.pqr)\ (\lambda qr.prq)$$

ⓑ We need to test if the computation at ⓐ is consistent with the known XOR behaviour. For that, we need to replace p with boolean values, T and F.

ⓐ says: $XOR\ p\ (NOT\ p) \Rightarrow^*_\beta p\ (\lambda qr.pqr)(\lambda qr.prq)$

For $p = T$:

$$p\ (\lambda qr.pqr)(\lambda qr.prq)$$

$$\equiv \underline{\underline{T}}\ (\lambda qr.Tqr)(\lambda qr.Trq) \qquad (T \text{ chooses first})$$

$$\Rightarrow_\beta \lambda qr.\underline{\underline{T}}qr$$

$$\Rightarrow_\beta \lambda qr.q$$

$$\equiv T$$

For $p = F$:

$$p\ (\lambda qr.pqr)(\lambda qr.prq)$$

$$\equiv \underline{\underline{F}}\ (\lambda qr.Fqr)(\lambda qr.Frq) \qquad (F \text{ chooses second})$$

$$\Rightarrow_\beta \lambda qr.\underline{\underline{F}}rq$$

$$\Rightarrow_\beta \lambda qr.q$$

$$\equiv T$$

In both cases, XOR behaves as expected.

4

```
(define count-inversions
  (lambda (l)
    (if (null? l)
        0
        (+ (count-smaller (car l) (cdr l)) (count-inversions (cdr l))))))

(define count-smaller
  (lambda (x l)
    (if (null? l)
        0
        (+ (if (> x (car l)) 1 0) (count-smaller x (cdr l))))))
```