Exercise 4.7.

`Let*` is similar to `let`, except that the bindings of the `let*` variables are performed sequentially from left to right, and each binding is made in an environment in which all of the preceding bindings are visible. For example

```
(let* ((x 3)
       (y (+ x 2))
       (z (+ x y 5)))
  (* x z))
```

returns 39. Explain how a `let*` expression can be rewritten as a set of nested `let` expressions, and write a procedure `let*->nested-lets` that performs this transformation. If we have already implemented `let` (exercise 4.6) and we want to extend the evaluator to handle `let*`, is it sufficient to add a clause to `eval` whose action is

```
(eval (let*->nested-lets exp) env)
```

or must we explicitly expand `let*` in terms of non-derived expressions?

Answer.

Typically, a `let*` expression

```
(let* ((<var1> <exp1>)
       (<var2> <exp2>)
       ...
       (<varn> <expn>))
  <body>)
```

is equivalent to

```
(let ((<var1> <exp1>))
  (let ((<var2> <exp2>))
    ...
      (let ((<varn> <expn>))
        <body>)...))
```

To transform a `let*` expression into a set of nested `let` expressions, we need to include syntax procedures that extract the parts of a `let*` expression. A `let*` expression begin with `let*` and has a list of associations and a body. Besides, we also need to reconstruct `let*` expressions when the list of association in the original `let*` expression contains more than one binding.

```
(define (let*? exp) (tagged-list? exp 'let*))

(define (list-of-associations exp) (cadr exp))

(define (let*-body exp) (cddr exp))

(define (make-let assocs body)
  (list 'let assocs body))

(define (make-let* assocs body)
  (list 'let* assocs body))

(define (let*->nested-lets exp)
  (let ((assocs (list-of-associations exp))
        (body (let*-body exp)))
    (if (null? assocs)
```


Email address: informlarry@gmail.com

```
        'false
        (let ((first-assoc (car assocs))
              (rest-assocs (cdr assocs)))
          (if (null? rest-assocs)
              (make-let (list first-assoc) body)
              (make-let (list first-assoc)
                        (let*->nested-lets (make-let* rest-assocs body))))))))
```

If we have already implemented `let` (exercise 4.6) and we want to extend the evaluator to handle `let*`. It is sufficient to add a clause to `eval` whose action is

```
(eval (let*->nested-lets exp) env)
```

Since `let` has already been implemented, all the `let` expressions will surely be evaluated by `eval`. So whenever the evaluator encounters a `let*` expression, all it needs to do is transform that expression into a set of nested `let` expressions using `let*->nested-lets`, the rest part of the evaluation is guaranteed to be handled by `let`.