Part I.

Data-structures.scm

First, we need to add the data structures, we need to add the nested procedure:

```
(nested-procedure
     (bvar symbol?)
     (count number?)
     (name symbol?)
     (body expression?)
     (env environment?))
```

Then the extension of the environment:

```
    (extend-env-rec-nested
     (id symbol?)
     (bvar symbol?)
     (count number?)
     (body expression?)
     (saved-env environment?))
```

Environments.scm

First of all we need to extend out environment with our count variable which will be n in my case:

```
(extend-env
       'n (num-val 0)
```

Then we need to handle the values in apply-env:

```
              (cases expval val
                (proc-val (procval)
                  (cases proc procval
                    (nested-procedure (bvar count name body env)
                       (proc-val (nested-procedure bvar count var body env)))
                    (else procval)))
                (else val))
```

And now we can handle the new environment extension:

```
        (extend-env-rec-nested (id bvar count body saved-env)
          (if (eqv? search-sym id)
              (proc-val (nested-procedure bvar count id body env))
              (apply-env saved-env search-sym)))
```

Interp.scm

Now we can start handling the interpreter

```
(proc-nested-exp (var count name body)
                         (proc-val (nested-procedure var (expval->num (value-of (var-exp count)
env)) name body env)))

        (call-nested-exp (rator rand count)
                  (let ((rnd (value-of rand env))
                        (procedure (expval->proc (value-of rator env)))
                        (newcount (expval->num (value-of count env))))
                    (apply-procedure
                     (cases proc procedure
                       (nested-procedure (var count name body env)
                                         (nested-procedure var newcount name body env))
                       (else procedure))
                     rnd)))

        (letrec-nested-exp (p-name b-var b-count p-body letrec-body)
                           (let ((count (expval->num (value-of (var-exp b-count) env))))
                             (value-of letrec-body
                                       (extend-env-rec-nested p-name b-var count p-body env))))
```

And we also need to add the nested-procedure to apply-procedure:

```
(nested-procedure (var count name body saved-env)
                          (begin
                            (recursive-displayer name count)
                            (value-of body (extend-env 'n (num-val count)
                                                       (extend-env var arg saved-env)))
                            ))
```

Lang.scm

Now we can add them to our grammar:

```
(expression
       ("proc-nested" "(" identifier "," identifier "," identifier ")" expression)
       proc-nested-exp)

      (expression
       ("call-nested" "(" expression expression "," expression ")")
       call-nested-exp)

      (expression
       ("letrec-nested" identifier "(" identifier "," identifier ")" "=" expression "in"
expression)
       letrec-nested-exp)
```

Translator.scm

Now we need to translate all of the newly added expressions

```
(proc-exp (var body)
                  (proc-nested-exp var 'n 'anonym
                                   (translation-of body env))
        )

(call-exp (rator rand)
                  (let* ((operator (translation-of rator env))
                         (operand (translation-of rand env))
                         (count (cases expression operator
                                  (var-exp (var) (diff-exp
                                                  (var-exp 'n)
                                                  (const-exp -1)))
                                  (else (const-exp 1)))))
                    (call-nested-exp operator operand count))
        )

(letrec-exp (p-name b-var p-body letrec-body)
                    (letrec-nested-exp p-name b-var 'n
                                       (translation-of p-body env)
                                       (translation-of letrec-body env))
        ) (letrec-exp (p-name b-var p-body letrec-body)
                    (letrec-nested-exp p-name b-var 'n
                                       (translation-of p-body env)
                                       (translation-of letrec-body env))
        )
```

Tests.scm

Extra test cases:

Test 1: here we evaluate a function based on the output of another function. Notice how b is unused, it is there only to check how let handles letrec in in its body

```
      (double-letrec-1
       "let b = 1 in letrec func1(y) = -(10,y) in letrec func2(z) = if zero?(z) then 1 else 2 in
(func1 (func2 0))"
       9)
```

Test 2: here we have an anonymous function which gets constructed on the fly and its result is passed to the letrec which will get called a max. of 2 times.

```
      (letrec-forever
       "letrec l(y) = if zero?(y) then 1 else (l 0) in (l (proc(x) -(x,-1) 4))"
       1)
```

Test 3: here a letrec constructs different anonymous functions on the fly depending on its input.

```
      (letrec-notletrec-letrec
       "letrec fun1(r) = if zero?(r) then
         (proc(x) if zero?(r) then -(r,-(0,r)) else 0 r)
         else
         (proc(x) if zero?(-(r, 1)) then 1 else 0 r)
         in (fun1 1)"
       1)
```

Part II.

Translator.scm

The translation of var-exp:

```
(let ((count (apply-senv-number senv var)))
                   (if (> count 0) (var-exp (string->symbol
                                             (string-append (symbol->string var) (number->string
count))))
                       (eopl:error 'translation-of "unbound variable in code: ~s" var)))
```

Translation of let:

```
(let ((count (apply-senv-number senv var)))
                   (if (> count 0) (var-exp (string->symbol
                                             (string-append (symbol->string var) (number->string
count))))
                       (eopl:error 'translation-of "unbound variable in code: ~s" var)))
```

Translation of proc:

```
(let* ((count (apply-senv-number senv var))
                         (var-string (symbol->string var))
                         (old-var
                          (string-append var-string (number->string count)))
                         (new-var
                          (string-append var-string (number->string (+ 1 count))))
                         (message (if (> count 0)
                                      (string-append var-string " has been reinitialized. " new-
var " is created and shadows " old-var ".")
                                      ""))
                         (var-field (string->symbol (string-append new-var " " message))))
                    (proc-exp var-field
                              (translation-of body (extend-senv var senv))))
```

Senv:

```
(lambda (senv var)
      (cond
        ((null? senv) 0)
        ((eqv? var (car senv))
         (+ 1 (apply-senv-number (cdr senv) var)))
        (else
         (+ 0 (apply-senv-number (cdr senv) var)))))
```