COMP301 Project 4

Project completed by Altun Hasanli. Code passess all tests provided.

Part 1

Add new `proc` type in `data-structures.rkt`

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

Handle the new `proc` type in `apply-procedure` in the file `interp.rkt`

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

Add new `environment` type in `data-structures.rkt`

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

Handle the new `environment` type in `apply-env` in `environments.scm`.

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

Handle the new `proc` type in `apply-env` in `environments.scm`. Keep the bound environment, but modify the name from `'anonym` to the variable name associated with the procedure in the environment.

```
...
(extend-env (var val saved-env)
        (if (eqv? search-sym var)

                ; #####################################################
```

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

                  ; #####################################################

                  (apply-env saved-env search-sym)))
...
```

Extend the initial environment to hold the `count` variable in `init-env` in `environments.scm`

```
(define init-env
        (lambda ()
                (extend-env
       'count (num-val 0)
                        ...
```

Modify the grammar in `lang.rkt` to include the new nested expressions.

```
(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)
```

Add the new nested expressions to the `translator.scm`. Pass `'count` variable symbol to `proc-nested-exp` and `letrec-nested-exp`. For `call-nested-exp`, pass the following **LETREC** expression: `-(count, -1)`.

```
...
(proc-exp (var body)
    (proc-nested-exp var 'count '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) (difference-exp
                                   (var-exp 'count)
                                   (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 'count
                        (translation-of p-body env)
                        (translation-of letrec-body env)))
...
```

Handle the new nested expressions in `interp.rkt`. For `proc-nested-exp` and `letrec-nested-exp`, retrieve count from the environment. For `call-nested-exp`, evaluate the translated count expression and update it in the `nested-procedure` type.

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

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

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

Add extra tests to `tests.scm`. These test procedures passed as arguments, nested `letrec` bound to different scopes, variable shadowing, and nested anonymous procedures.

```
(nested-procs-3
       "let a = 1 in
                                 let a = 2 in
          let f = proc(func) ((func a) b) in
            let g = proc(a) proc(b) -(a, b) in
              -(a, -(b, (f g)))"
       -2)

Recursive Print:
f --> 1
....func --> 2
anonym --> 1

(nested-letrecs-1
       "let m = 0 in let b = -1 in
          letrec f(func1) = ((func1 m) b) in
            letrec g(func2) = ((func2 m) b) in
              let u = (f proc(a) proc(b) -(a, b)) in
                let v = (g proc (a) proc(b) -(-(a,b),-10)) in
                  letrec double(x) = if zero?(x) then 0 else -((double -(x,1)), -2) in
                    -((double u), -(0, (double v)))"
       24)

Recursive Print:
f --> 1
....func1 --> 2
anonym --> 1
g --> 1
....func2 --> 2
anonym --> 1
double --> 1
```

```
....double --> 2
double --> 1
....double --> 2
........double --> 3
............double --> 4
................double --> 5
....................double --> 6
........................double --> 7
............................double --> 8
................................double --> 9
....................................double --> 10
........................................double --> 11
............................................double --> 12

(nested-letrecs-2
                        "let m = 0 in let n = -1 in let o = 5 in
                          letrec f(func1) = ((func1 o) n) in
                            letrec g(func2) = ((func2 m) n) in
                              letrec h(func3) = ((func3 o) m) in
                                let u = (f proc(a) proc(b) -(a, b)) in
                                  let v = (g proc (a) proc(b) -(-(a,b), -10)) in
                                    let w = (h proc (a) proc(b) -(-(a,b), -5)) in
                                      letrec sum(x) = if zero?(x) then 0 else -(x, -(0, (sum -
(x,1)))) in
                                        letrec increment(x) = if zero?(x) then 1 else -
((increment -(x,1)), -1) in
                                          letrec double(x) = if zero?(x) then 0 else -((double
-(x,1)), -2) in
                                            -((increment u), -((sum v), (double w)))"
      -39)

Recursive Print:
f --> 1
....func1 --> 2
anonym --> 1
g --> 1
....func2 --> 2
anonym --> 1
h --> 1
....func3 --> 2
anonym --> 1
increment --> 1
....increment --> 2
........increment --> 3
............increment --> 4
................increment --> 5
....................increment --> 6
........................increment --> 7
sum --> 1
....sum --> 2
........sum --> 3
............sum --> 4
................sum --> 5
....................sum --> 6
........................sum --> 7
............................sum --> 8
................................sum --> 9
....................................sum --> 10
........................................sum --> 11
............................................sum --> 12
double --> 1
....double --> 2
```

```
........double --> 3
............double --> 4
................double --> 5
....................double --> 6
........................double --> 7
............................double --> 8
................................double --> 9
....................................double --> 10
........................................double --> 11
```

Part 2

Implement the procedure `apply-senv-number`, which finds the number of occurrences of a variable `var` in a static environment `senv`.

```
(define apply-senv-number
  ; ############################################################
  ; ###### define apply-senv-number, a procedure that applies
  ; ###### the environment and finds the occurences of variable
  ; ###### var in the environment senv
  ; ############################################################
    (lambda (senv var)
      ; find number of occurences of var in senv
      (cond
        ((null? senv) 0)
        ((eqv? var (car senv))
          (+ 1 (apply-senv-number (cdr senv) var)))
        (else
          (+ 0 (apply-senv-number (cdr senv) var)))))
   )
```

Implement the translator for `var-exp`, which references a variable `x` in the lexical scope. Variable `x` is renamed, based on the number of occurrences using the procedure `apply-senv-number`. If no occurrences are found, translator is interrupted and “unbound variable” error is thrown.

```
(var-exp (var)
 ; ################################################
 ; ############ implement translation of var-exp here
 ; ################################################
  (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 program: ~s" var)))
 )
```

Implement the translator for `let-exp`, which declares and possibly, shadows a variable in the scope. If a variable is shadowed, the variable name is appended with an appropriate message.

```
(lett-exp (var exp1 body)
        ; ################################################
        ; ############ implement translation of let-exp here
        ; ################################################
        (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))))
            (lett-exp var-field
                      (translation-of exp1 senv)
                      (translation-of body (extend-senv var senv))))
          )
```

Implement the translator for `proc-exp` with the same logic as `let-exp`.

```
 (proc-exp (var body)
  ; ################################################
  ; ############ implement translation of proc-exp here
  ; ################################################
   (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))))
  )
```