COMP301 Project 4

Ahmet Uyar
ID: 72847

Part 1: Translator for LETREC

Adding new expression variants to *lang.scm*

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

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

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

We now add the nested-procedure to *data-structures.scm* and handle it in *interp.rkt*. Same process for *extend-env-rec-nested* now handle it in *environments.scm*.

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

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

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

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

Handle nested procedure in *appy-env*:

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

      ; ######################################################
      ; ###### ENTER YOUR CODE HERE, YOU MAY DELETE
      ; ###### THE CODE BELOW, IT IS PUT TO PROVIDE A RUNNING
      ; ###### CODE BASELINE.
      ; ######
      ; ###### You need to check the given value, and take
      ; ###### care of the case where the given value is a
      ; ###### nested-procedure. If it is a nested-procedure,
      ; ###### a proc-val with a nested-procedure should be
      ; ###### returned. Otherwise, it should behave
      ; ###### as it normally does.
      ; ######################################################
      (cases expval val
        (proc-val (procval)
          (cases proc procval
            (nested-procedure (bvar body env name count)
              (proc-val (nested-procedure bvar body env var count))) ;; return nested proc
            (else procval)))
        (else val))

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

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

We add count to *init-env*:

```
  ; """"""""""""""""""""""""""""""""""""""

  (extend-env
   'count (num-val 0) ;; add count
```

Now we add translation instructions for proc-exp, call-exp, and letrec-exp.

```
(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 (if (eq? (car operator) 'var-exp)
                            (difference-exp (var-exp 'count) (const-exp -1))
                            (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))
```

Finally we handle value-of statements for these three operations in *interp.rkt*

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

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

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

Now adding additional test cases:

1) **Fibonacci:** finds the nth fibonacci number.

```
(fibonacci
 "letrec fib(x) = if zero?(x)
      then 0
      else if zero?(-(x,1)) then 1
          else -((fib -(x,1)), -(0,(fib -(x,2))))
 in (fib 5)"
 5)
```

Testing for the 5th fibonacci number gives the correct result of 5.

```
test: fibonacci
fib --> 1
....fib --> 2
........fib --> 3
............fib --> 4
................fib --> 5
................fib --> 5
............fib --> 4
........fib --> 3
............fib --> 4
............fib --> 4
....fib --> 2
........fib --> 3
............fib --> 4
............fib --> 4
........fib --> 3
correct
```

2) **Linear Sum:** finds the linear sum until n

```
(linear-sum
 "letrec linear-sum(x) = if zero?(x)
      then 0
      else -((linear-sum -(x,1)), -(0,x)) in (linear-sum 10)" 55)
```

Testing for 10 which is 55.

```
test: linear-sum
linear-sum --> 1
....linear-sum --> 2
........linear-sum --> 3
............linear-sum --> 4
................linear-sum --> 5
....................linear-sum --> 6
........................linear-sum --> 7
............................linear-sum --> 8
................................linear-sum --> 9
....................................linear-sum --> 10
........................................linear-sum --> 11
correct
```

3) **Times five**: finds the number n multiplied with 5

```
   (times-five
    "let a = 1
        in let b = 5
          in letrec times-five(x) = if zero?(x) then 0
            else -((times-five -(x,1)), -(0,5)) in (times-five 20)"
     100)
```

```
test: times-five
times-five --> 1
....times-five --> 2
........times-five --> 3
............times-five --> 4
................times-five --> 5
....................times-five --> 6
........................times-five --> 7
............................times-five --> 8
................................times-five --> 9
....................................times-five --> 10
........................................times-five --> 11
............................................times-five --> 12
................................................times-five --> 13
....................................................times-five --> 14
........................................................times-five --> 15
............................................................times-five --> 16
................................................................times-five --> 17
....................................................................times-five --> 18
........................................................................times-five --> 19
............................................................................times-five --> 20
................................................................................times-five --> 21
correct
```

All tests pass and are identical to the expected print-out.

Part 2: Modify Translator for LEXADDR

First we add *apply-senv-number* to the translator, this will find the occurrences of var in the static environment.

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

 )
```

Then add *var-exp, let-exp,* and *proc-exp*:

```
(var-exp (var)
 ; ################################################
 ; ############ implement translation of var-exp here
 ; ################################################
        (if (> (apply-senv-number var senv) 0)
            (var-exp (string->symbol (string-append (symbol->string var) (number->string (apply-senv-number var senv)))))
            (eopl:error "~s is an unbound variable" var))

 )

(let-exp (var expl body)
  ; ################################################
  ; ############ implement translation of let-exp here
  ; ################################################
         (let* ((var-str (symbol->string var))
                (count (apply-senv-number var senv))
                (old-var (string-append var-str (number->string count)))
                (new-var (string-append var-str (number->string (+ 1 count))))
                (body-translation (translation-of body (extend-senv var senv)))
                (expression-translation (translation-of expl senv))
                (message (if (> count 0)
                             (string-append var-str " has been reinitialized. " new-var " is created and shadows " old-var ".")
                             "")))
           (let-exp (string->symbol (string-append new-var " " message)) expression-translation body-translation))


 )

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


 )
```

var-exp checks if the variable is bound in the current environment senv. If the variable is bound, it returns the number of occurrences of that variable in the environment, indicating its lexical address. If not, it raises an error.

let-exp handles let expressions where the variable is bound within the local scope. We recursively call *translation-of* for the bounding expression and the body thus extending the environment.

proc-exp handles procedure expressions where we translate the body extended with the bound variable var. Similar to *let-exp* we recursively call *translation-of* for body translation.

All tests pass and are identical to the expected printouts.