Exercise 3.19 procedural representation

```
;; the-grammar
(expression
 ("letproc" identifier "(" identifier ")" "="  expression "in"
            expression)
 letproc-exp)

;; value-of
(letproc-exp (name var proc-body body)
             (let ((pval (proc-val (procedure var proc-body env)))
               (value-of body
                         (extend-env name pval env))))
```

Exercise 3.20

```
proc (x) proc (y) -(x, -(0, y))
```

Exercise 3.21 procedural representation

```
;; the-grammar
(expression
 ("proc" "(" (separated-list identifier ",") ")" expression)
 proc-exp)

(expression
 ("(" expression (arbno expression) ")")
 call-exp)

;; value-of
(proc-exp (vars body)
          (proc-val (procedure vars body env)))
(call-exp (rator rands)
          (let ((proc (expval->proc (value-of rator env)))
                (args (map (lambda (rand) (value-of rand env))
```

```
                              rands)))
              (apply-procedure proc args)))

;; procedure : Listof(Var) * Exp * Env -> Proc
(define procedure
  (lambda (vars body env)
    (lambda (vals)
      (value-of body (extend-env* vars vals env)))))

;; apply-procedure : Proc * Listof(ExpVal) -> ExpVal
(define apply-procedure
  (lambda (proc vals)
    (proc vals)))
```

Exercise 3.23

```
(times 4 3)
((makemult makemult) 3)
-(((makemult makemult) 2), -4)
-(-(((makemult makemult) 1), -4), -4)
-(-(-(((makemult makemult) 0), -4), -4), -4)
-(-(-(0, -4), -4), -4)
-(-(4, -4), -4)
-(8, -4)
12
```

Let $a$ be any nonnegative integer.

```
let times = proc (maker)
             proc (y)
              proc (x)
               if zero?(x)
               then 0
               else -((((maker maker) y) -(x,1)), -(0, y))
in let fact = proc (maker)
```

```
                proc (x)
                 if zero?(x)
                 then 1
                 else (((times times) ((maker maker) -(x, 1))) x)
in ((fact fact) a)
```

Exercise 3.24

Let $a$ be any nonnegative integer.

```
let makeeven = proc (this)
                proc (next)
                 proc (x)
                  if zero?(x)
                  then 1
                  else (((next next) this) -(x, 1))
in let makeodd = proc (this)
                  proc (next)
                   proc (x)
                    if zero?(x)
                    then 0
                    else (((next next) this) -(x, 1))
in let odd = proc (x) (((makeodd makeodd) makeeven) x)
in let even = proc (x) (((makeeven makeeven) makeodd) x)
in (even a)
```

Exercise 3.25

```
(times4 3)
((makerec maketimes4) 3)
((maketimes4 (d d)) 3)
((maketimes4 proc (z) ((maketimes4 (d d)) z)) 3)
-((proc (z) ((maketimes4 (d d)) z) 2), -4)
-(((maketimes4 (d d)) 2), -4)
-(((maketimes4 proc (z) ((maketimes4 (d d)) z)) 2), -4)
-(-((proc (z) ((maketimes4 (d d)) z) 1), -4), -4)
```

```
-(-(((maketimes4 (d d)) 1), -4), -4)
-(-(((maketimes4 proc (z) ((maketimes4 (d d)) z)) 1), -4), -4)
-(-(-((proc (z) ((maketimes4 (d d)) z) 0), -4), -4), -4)
-(-(-(((maketimes4 (d d)) 0), -4), -4), -4)
-(-(-(((maketimes4
        proc (z) ((maketimes4 (d d)) z)) 0), -4), -4), -4)
-(-(-(0, -4), -4), -4)
-(-(4, -4), -4)
-(8, -4)
12
```

Exercise 3.26 data-structure representation

```
;; value-of
(proc-exp (var body)
          (proc-val
           (procedure
            var
            body
            (all-occurs-free body (list var) env))))

;; append-env : Env * Env -> Env
(define append-env append)

;; all-occurs-free : Exp * Listof(Var) * Env -> Env
(define all-occurs-free
  (lambda (exp bound-vars env)
    (cases expression exp
      (const-exp (num) (empty-env))
      (var-exp (var)
               (if (memq var bound-vars)
                   (empty-env)
                   (extend-env var (apply-env env var)
                               (empty-env))))
```

```
      (diff-exp (exp1 exp2)
                (append-env
                 (all-occurs-free exp1 bound-vars env)
                 (all-occurs-free exp2 bound-vars env)))
      (zero?-exp (exp1)
                 (all-occurs-free exp1 bound-vars env))
      (if-exp (exp1 exp2 exp3)
              (append-env
               (all-occurs-free exp1 bound-vars env)
               (append-env
                (all-occurs-free exp2 bound-vars env)
                (all-occurs-free exp3 bound-vars env))))
      (let-exp (var exp1 body)
               (append-env
                (all-occurs-free exp1 bound-vars env)
                (all-occurs-free body (cons var bound-vars) env)))
      (proc-exp (var body)
                (all-occurs-free body (cons var bound-vars) env))
      (call-exp (rator rand)
                (append-env
                 (all-occurs-free rator bound-vars env)
                 (all-occurs-free rand bound-vars env))))))
```

Exercise 3.27 data-structure representation

```
;; the-grammar
(expression
 ("traceproc" "(" identifier ")" expression)
 traceproc-exp)

;; value-of
(traceproc-exp (var body)
               (proc-val (traceproc var body env)))
```

```
;; apply-procedure : Proc * ExpVal -> ExpVal
(define apply-procedure
  (lambda (proc1 val)
    (cases proc proc1
      (procedure (var body saved-env)
                 (value-of body (extend-env var val saved-env)))
      (traceproc (var body saved-env)
                 (display (list "entering with" var "=" val))
                 (newline)
                 (let ((val1 (value-of body (extend-env
                                             var
                                             val
                                             saved-env))))
                   (display (list "exiting with" val1))
                   (newline)
                   val1)))))

;; proc? : SchemeVal -> Bool
;; procedure : Var * Exp * Env -> Proc
(define-datatype proc proc?
  (procedure
   (var symbol?)
   (body expression?)
   (env environment?))
  (traceproc
   (var symbol?)
   (body expression?)
   (env environment?)))
```

Exercise 3.28 data-structure representation

```
;; value-of
(proc-exp (var body)
          (proc-val (procedure var body)))
```

```
(call-exp (rator rand)
          (let ((proc (expval->proc (value-of rator env)))
                (arg (value-of rand env)))
            (apply-procedure proc arg env)))

;; apply-procedure : Proc * ExpVal -> ExpVal
(define apply-procedure
  (lambda (proc1 val env)
    (cases proc proc1
      (procedure (var body)
                 (value-of body (extend-env var val env))))))

;; proc? : SchemeVal -> Bool
;; procedure : Var * Exp * Env -> Proc
(define-datatype proc proc?
  (procedure
   (var symbol?)
   (body expression?)))
```

Exercise 3.28 procedural representation

```
;; value-of
(proc-exp (var body)
          (proc-val (procedure var body)))
(call-exp (rator rand)
          (let ((proc (expval->proc (value-of rator env)))
                (arg (value-of rand env)))
            (apply-procedure proc arg env)))

;; procedure : Var * Exp -> Proc
(define procedure
  (lambda (var body)
    (lambda (val env)
      (value-of body (extend-env var val env)))))
```

```
;; apply-procedure : Proc * ExpVal * Env -> ExpVal
(define apply-procedure
  (lambda (proc val env)
    (proc val env)))
```

Exercise 3.29

When `(f 2)` is called, the formal parameter `a` is bound to 2. Then `(p 0)` binds `z` to 0 in an environment where `a` is bound to 2, and thus `proc (z) a` returns 2.