Exercise 3.18

```
(letproc-exp
 (name identifier?)
 (var identifier?)
 (proc-body expression?)
 (body expression?))

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

Exercise 3.19

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

Exercise 3.20

```
(define-datatype proc proc?
  (procedure
   (vars (list-of identifier?))
   (body expression?)
   (saved-env environment?)))

(define apply-procedure
  (lambda (proc1 vals)
    (cases proc proc1
      (procedure (vars body saved-env)
                 (value-of body (extend-env* vars
                                             vals
                                             saved-env))))))
```

```
(proc-exp
 (formal-parameters (list-of identifier?))
 (body expression?))
(call-exp
 (rator expression?)
 (rands (list-of expression?)))

(proc-exp (params body)
          (proc-val (procedure params 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)))
```

Exercise 3.21

```
(define-datatype proc proc?
  (procedure
   (vars (list-of identifier?))
   (body expression?)
   (saved-env environment?)))

(define apply-procedure
  (lambda (proc1 vals)
    (cases proc proc1
      (procedure (vars body saved-env)
                 (value-of body (extend-env* vars
                                             vals
                                             saved-env))))))

(proc-exp
 (formal-parameters (list-of identifier?))
```

```
 (body expression?))
(call-exp
 (rator expression?)
 (rands (list-of expression?)))

(proc-exp (params body)
          (proc-val (procedure params 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)))
```

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

```
(proc-exp (var body)
          (proc-val
           (procedure
            var
            body
            (let ((vars (no-repeats (all-occurs-free body))))
              (extend-env* vars
                           (map (lambda (v) (apply-env env v))
                                vars)
                           (empty-env))))))

(define all-occurs-free
  (lambda (exp)
    (cases expression exp
      (const-exp (num) '())
      (var-exp (var) (list var))
      (diff-exp (exp1 exp2)
                (append (all-occurs-free exp1)
                        (all-occurs-free exp2)))
      (zero?-exp (exp1)
                 (all-occurs-free exp1))
      (if-exp (exp1 exp2 exp3)
```

```
              (append (all-occurs-free exp1)
                      (append (all-occurs-free exp2)
                              (all-occurs-free exp3))))
      (let-exp (var exp1 body)
               (append (all-occurs-free exp1)
                       (remove var (all-occurs-free body))))
      (proc-exp (var body)
                (remove var (all-occurs-free body)))
      (call-exp (rator rand)
                (append (all-occurs-free rator)
                        (all-occurs-free rand))))))
```

Exercise 3.27

```
(traceproc
 (var identifier?)
 (body expression?)
 (saved-env environment?))

(traceproc-exp
 (var identifier?)
 (body expression?))

(traceproc-exp (var body)
               (proc-val (traceproc var body env)))

(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 "Enter")
                 (let ((val1 (value-of body (extend-env
```

```
                                                    var
                                                    val
                                                    saved-env))))
                        (display "Exit")
                        val1)))))
```

Exercise 3.28

```
(define-datatype proc proc?
  (procedure
   (var identifier?)
   (body expression?)))

(define apply-procedure
  (lambda (proc1 val env)
    (cases proc proc1
      (procedure (var body)
                 (value-of body (extend-env var val env))))))

(define procedure
  (lambda (var body)
    (lambda (val env)
      (value-of body (extend-env var val env)))))

(define apply-procedure
  (lambda (proc arg env)
    (proc arg env)))

(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)))
```

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.