Interfaces for Recursive Data Types

Exercise 2.15

```
(define var-exp
  (lambda (var)
    var))

(define lambda-exp
  (lambda (var lc-exp)
    (list 'lambda (list var) lc-exp)))

(define app-exp
  (lambda (lc-exp1 lc-exp2)
    (list lc-exp1 lc-exp2)))

(define var-exp?
  (lambda (lc-exp)
    (symbol? lc-exp)))

(define lambda-exp?
  (lambda (lc-exp)
    (if (var-exp? lc-exp)
        #f
        (eqv? (car lc-exp) 'lambda))))

(define app-exp?
  (lambda (lc-exp)
    (and (not (var-exp? lc-exp))
         (not (lambda-exp? lc-exp)))))

(define var-exp->var
  (lambda (lc-exp)
    lc-exp))
```

```
(define lambda-exp->bound-var
  (lambda (lc-exp)
    (caadr lc-exp)))

(define lambda-exp->body
  (lambda (lc-exp)
    (caddr lc-exp)))

(define app-exp->rator
  (lambda (lc-exp)
    (car lc-exp)))

(define app-exp->rand
  (lambda (lc-exp)
    (cadr lc-exp)))
```

Exercise 2.16

```
(define lambda-exp
  (lambda (var lc-exp)
    (list 'lambda var lc-exp)))

(define lambda-exp->bound-var
  (lambda (lc-exp)
    (cadr lc-exp)))
```

Exercise 2.17

```
(define var-exp
  (lambda (var)
    var))

(define lambda-exp
  (lambda (var lc-exp)
```

```
    (list 'lambda (list 'bound-variable var) lc-exp)))

(define app-exp
  (lambda (lc-exp1 lc-exp2)
    (list 'application lc-exp1 lc-exp2)))

(define var-exp?
  (lambda (lc-exp)
    (symbol? lc-exp)))

(define lambda-exp?
  (lambda (lc-exp)
    (if (var-exp? lc-exp)
        #f
        (eqv? (car lc-exp) 'lambda))))

(define app-exp?
  (lambda (lc-exp)
    (if (var-exp? lc-exp)
        #f
        (eqv? (car lc-exp) 'application))))

(define var-exp->var
  (lambda (lc-exp)
    lc-exp))

(define lambda-exp->bound-var
  (lambda (lc-exp)
    (cadr (cadr lc-exp))))

(define lambda-exp->body
  (lambda (lc-exp)
```

```
    (caddr lc-exp)))

(define app-exp->rator
  (lambda (lc-exp)
    (cadr lc-exp)))

(define app-exp->rand
  (lambda (lc-exp)
    (caddr lc-exp)))

(define var-exp
  (lambda (var)
    (lambda (pred/extr observer)
      (if (eqv? pred/extr 'predicate)
          (eqv? observer 'var?)
          var))))

(define lambda-exp
  (lambda (var lc-exp)
    (lambda (pred/extr observer)
      (if (eqv? pred/extr 'predicate)
          (eqv? observer 'lambda?)
          (if (eqv? observer 'bound-var)
              var
              lc-exp)))))

(define app-exp
  (lambda (lc-exp1 lc-exp2)
    (lambda (pred/extr observer)
      (if (eqv? pred/extr 'predicate)
          (eqv? observer 'app?)
          (if (eqv? observer 'rator)
              lc-exp1
```

```
              lc-exp2)))))

(define var-exp?
  (lambda (lc-exp)
    (lc-exp 'predicate 'var?)))

(define lambda-exp?
  (lambda (lc-exp)
    (lc-exp 'predicate 'lambda?)))

(define app-exp?
  (lambda (lc-exp)
    (lc-exp 'predicate 'app?)))

(define var-exp->var
  (lambda (lc-exp)
    (lc-exp 'extractor 'var)))

(define lambda-exp->bound-var
  (lambda (lc-exp)
    (lc-exp 'extractor 'bound-var)))

(define lambda-exp->body
  (lambda (lc-exp)
    (lc-exp 'extractor 'body)))

(define app-exp->rator
  (lambda (lc-exp)
    (lc-exp 'extractor 'rator)))

(define app-exp->rand
  (lambda (lc-exp)
```

```
(lc-exp 'extractor 'rand)))
```