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

A data structure representation where all lists are tagged.

```
(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)))
```

A procedural representation.

```
(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)))
```

Exercise 2.18

```
(define number->sequence
  (lambda (num)
    (list num '() '())))

(define current-element
  (lambda (seq)
    (car seq)))

(define move-to-left
  (lambda (seq)
    (if (at-left-end? seq)
        (report-at-left-end seq)
        (list (first-of-left seq)
              (rest-of-left seq)
              (cons (current-element seq) (all-of-right seq))))))

(define move-to-right
  (lambda (seq)
    (if (at-right-end? seq)
        (report-at-right-end seq)
        (list (first-of-right seq)
              (cons (current-element seq) (all-of-left seq))
              (rest-of-right seq)))))

(define insert-to-left
  (lambda (num seq)
    (list (current-element seq)
          (cons num (all-of-left seq))
          (all-of-right seq))))

(define insert-to-right
  (lambda (num seq)
    (list (current-element seq)
          (all-of-left seq)
          (cons num (all-of-right seq)))))

(define at-left-end?
  (lambda (seq)
    (null? (all-of-left seq))))

(define at-right-end?
  (lambda (seq)
    (null? (all-of-right seq))))

(define first-of-left
  (lambda (seq)
    (car (all-of-left seq))))

(define rest-of-left
  (lambda (seq)
    (cdr (all-of-left seq))))

(define all-of-left
  (lambda (seq)
    (cadr seq)))

(define first-of-right
  (lambda (seq)
    (car (all-of-right seq))))
```

```
(define rest-of-right
  (lambda (seq)
    (cdr (all-of-right seq))))

(define all-of-right
  (lambda (seq)
    (caddr seq)))

(define report-at-left-end
  (lambda (seq)
    (eopl:error 'move-to-left
                "Cannot move to the left in ~s"
                seq)))

(define report-at-right-end
  (lambda (seq)
    (eopl:error 'move-to-right
                "Cannot move to the right in ~s"
                seq)))
```

Exercise 2.19

```
(define number->bintree
  (lambda (num)
    (list num '() '())))

(define current-element
  (lambda (bintree)
    (car bintree)))

(define move-to-left-son
  (lambda (bintree)
    (if (at-leaf? bintree)
        (report-at-leaf bintree)
        (cadr bintree))))

(define move-to-right-son
  (lambda (bintree)
    (if (at-leaf? bintree)
        (report-at-leaf bintree)
        (caddr bintree))))

(define at-leaf?
  (lambda (bintree)
    (null? bintree)))

(define insert-to-left
  (lambda (num bintree)
    (list (current-element bintree)
          (list num
                (cadr bintree)
                '())
          (caddr bintree))))

(define insert-to-right
  (lambda (num bintree)
    (list (current-element bintree)
          (cadr bintree)
          (list num
                (caddr bintree)
                '()))))
```

Exercise 2.20

Where the current node is not the root, then the current node has a parent. Where a node has a parent, then the parent has as its left or right son the symbol `hold` depending on where the current node is relative to the parent.

If the bintree is a leaf, then we cannot traverse the tree any longer as a leaf is only the empty list.

```
(define number->bintree
  (lambda (num)
    (list num '() '() '())))

(define current-element
  (lambda (bintree)
    (car bintree)))

(define move-to-left-son
  (lambda (bintree)
    (cond ((at-leaf? bintree)
           (report-illegal-move 'move-to-left-son 'leaf))
          ((at-leaf? (left-son bintree)) '())
          (else (list (current-element (left-son bintree))
                      (left-son (left-son bintree))
                      (right-son (left-son bintree))
                      (list (current-element bintree)
                            'hold
                            (right-son bintree)
                            (parent bintree)))))))

(define move-to-right-son
  (lambda (bintree)
    (cond ((at-leaf? bintree)
           (report-illegal-move 'move-to-right-son 'leaf))
          ((at-leaf? (right-son bintree)) '())
          (else (list (current-element (right-son bintree))
                      (left-son (right-son bintree))
                      (right-son (right-son bintree))
                      (list (current-element bintree)
                            (left-son bintree)
                            ’hold
                            (parent bintree)))))))

(define insert-to-left
  (lambda (num bintree)
    (list (current-element bintree)
          (list num
                (left-son bintree)
                ’()
                ’())
          (right-son bintree)
          (parent bintree))))

(define insert-to-right
  (lambda (num bintree)
    (list (current-element bintree)
          (left-son bintree)
          (list num
                ’()
                (right-son bintree)
                ’())
          (parent bintree))))

(define move-up
  (lambda (bintree)
    (if (at-root? bintree)
        (report-illegal-move ’move-up ’root)
        (if (eqv? (left-son (parent bintree)) ’hold)
            (list (current-element (parent bintree))
                  (list (current-element bintree)
                        (left-son bintree)
                        (right-son bintree)
                        '())
                  (right-son (parent bintree))
                  (parent (parent bintree)))
            (list (current-element (parent bintree))
                  (left-son (parent bintree))
                  (list (current-element bintree)
                        (left-son bintree)
                        (right-son bintree)
                        '())
                  (parent (parent bintree)))))))

(define at-root?
  (lambda (bintree)
    (null? (parent bintree))))

(define at-leaf?
  (lambda (bintree)
    (null? bintree)))

(define left-son
  (lambda (bintree)
    (cadr bintree)))

(define right-son
  (lambda (bintree)
    (caddr bintree)))

(define parent
  (lambda (bintree)
    (cadddr bintree)))
```

```
(define report-illegal-move
  (lambda (proc-name node-type)
    (eopl:error proc-name
                "Cannot ~s from ~s"
                proc-name
                node-type)))
```