A Tool for Defining Recursive Data Types

Exercise 2.21

```
(define-datatype env env?
  (empty-env)
  (extended-env
   (saved-var identifier?)
   (saved-val schemeval?)
   (saved-env env?)))

(define has-binding?
  (lambda (s e)
    (cases env e
      (empty-env () #f)
      (extended-env (var val env)
                    (if (eqv? s var)
                        #t
                        (has-binding? s env))))))

(define identifier? symbol?)
(define schemeval?
  (lambda (a)
    #t))
```

Exercise 2.22

```
(define-datatype stack stack?
  (empty-stack)
  (non-empty-stack
   (first schemeval?)
   (rest stack?)))

(define push
  (lambda (v s)
```

```
    (non-empty-stack v s)))

(define pop
  (lambda (s)
    (cases stack s
      (empty-stack () (report-stack-is-empty 'pop))
      (non-empty-stack (first rest)
                       rest))))

(define top
  (lambda (s)
    (cases stack s
      (empty-stack () (report-stack-is-empty 'top))
      (non-empty-stack (first rest)
                       first))))

(define empty-stack?
  (lambda (s)
    (cases stack s
      (empty-stack () #t)
      (non-empty-stack (f r) #f))))

(define report-stack-is-empty
  (lambda (observer)
    (eopl:error 'empty-stack "Called ~s on an empty stack"
                observer)))
```

Exercise 2.23

```
(define identifier2?
  (lambda (i)
    (and (symbol? i)
         (not (eqv? i 'lambda)))))
```

Exercise 2.24

```
(define bintree-to-list
  (lambda (btree)
    (cases bintree btree
      (leaf-node (num) (list 'leaf-node num))
      (interior-node (key left right)
                     (list 'interior-node
                           key
                           (bintree-to-list left)
                           (bintree-to-list right))))))
```

Exercise 2.25

```
(define sum-tree
  (lambda (btree)
    (cases bintree btree
      (leaf-node (num) num)
      (interior-node (k l r)
                     (+ (sum-tree l)
                        (sum-tree r))))))

(define max-tree
  (lambda (btree)
    (cases bintree btree
      (leaf-node (num) btree)
      (interior-node
       (k l r)
       (let ((lmax-tree (max-tree l))
             (rmax-tree (max-tree r)))
         (let ((lmax-sum (sum-tree lmax-tree))
               (rmax-sum (sum-tree rmax-tree))
               (t-sum (sum-tree btree)))
           (cond ((and (>= t-sum lmax-sum)
```

```
                        (>= t-sum rmax-sum))
                   btree)
                  ((and (>= lmax-sum t-sum)
                        (>= lmax-sum rmax-sum))
                   lmax-tree)
                  (else rmax-tree))))))))

(define max-interior
  (lambda (btree)
    (cases bintree btree
      (leaf-node (num) 'fail)
      (interior-node (k l r)
                     (let ((max-btree (max-tree btree)))
                       (cases bintree max-btree
                         (leaf-node (n) 'impossible)
                         (interior-node (key left right)
                                        key)))))))
```

Exercise 2.26

```
(define-datatype red-blue-tree red-blue-tree?
  (a-red-blue-tree
   (rb-tree red-blue-subtree?)))

(define-datatype red-blue-subtree red-blue-subtree?
  (red-subtree
   (left red-blue-subtree?)
   (right red-blue-subtree?))
  (blue-subtree
   (blue-subtrees (list-of red-blue-subtree?)))
  (rb-leaf-node
   (num integer?)))

(define mark
```

```
  (lambda (tree n)
    (cases red-blue-subtree tree
      (red-subtree (left right)
                   (red-subtree (mark left (+ n 1))
                                (mark right (+ n 1))))
      (blue-subtree (blue-subtrees)
                    (blue-subtree (map (lambda (bt) (mark bt n))
                                       blue-subtrees)))
      (rb-leaf-node (num)
                    (rb-leaf-node n)))))

(define mark-with-red-leaves
  (lambda (rb-tree)
    (cases red-blue-tree rb-tree
      (a-red-blue-tree (rbt)
                       (mark rbt 0)))))
```