Problem 1: Binary Trees: Flatten

Recall the data definition for binary trees:

A **binary-tree** is either:

- #f
- (make-bt val left right)
  where val is a number,
  and left and right are binary-trees.

- (define-struct bt (val left right))

Write a function **flatten** that returns a list of the values of all the nodes in the tree.

Here are some examples (be sure to use these as tests when developing your function):

```
(define n1 (make-bt 1 #f #f))
(define n2 (make-bt 2 #f #f))
(define n3 (make-bt 3 #f #f))
(define m12 (make-bt 12 n1 n2))
(define r123 (make-bt 123 m12 n3))

(flatten r123) should be (123 12 1 2 3)
;; Note: This is simply a sample output. Any order of the same
;; numbers would be acceptable.
(flatten n1) should be (1)
```

Adding Data

Add new binary tree nodes and modify as few existing nodes as possible to produce a binary tree with four nodes at depth 2.

Submit your revised set of tree definitions.

Template

Write the template that you will follow in constructing this function.

Contract & Purpose

Write the contract specifying the input/output types of **flatten** and the purpose statement specifying its behavior.

Implementation

Write the code to implement **flatten**. Confirm that it produces the desired behavior on both the original examples and your modified examples.

Problem 2: Following Directions

In Homework 4, we explored the notion of paths in family trees, where the path was the sequence of mom or dad relations between the initial family tree node and an ancestor with a desired trait. Here we will generalize the notion of paths to describe paths through binary trees, rather than family trees in particular.

We can view a path in this sense as specifying a route through given binary tree. We will define a function **(follow-directions abt path)** that returns the value of the node reached by following the input path in the input tree. For example, using the example binary trees defined above:

```
(follow-directions r123 '()) should be 123.
(follow-directions m12 '(l r r)) should be #f.
(follow-directions r123 '(l r)) should be 2.
```

Data Definition

Here is the data definition for a path:

a Path is either:

- '()
- (cons 'l path)
- (cons 'r path)

Template

The follow-directions function's arguments are both interesting from the point of view of the template. So, rather than just having two conditions from a binary tree or three conditions from the path, we have 6 conditions, one for each combination of family trees and paths:

```
(define (bt-path-template a-bt a-path)
  (cond
    [(and (boolean? a-bt)
          (null? a-path))
     ...]
    [(and (bt? a-bt)
          (null? path))
     ...]
    [(and (boolean? a-bt)
          (eq? 'l (car path)))
     ...]
    [(and (bt? a-bt)
          (eq? 'l (car path)))
     ...]
    [(and (boolean? a-bt)
          (eq? 'r (car path)))
     ...]
    [(and (bt? a-bt)
          (eq? 'r (car path)))
     ...]))
```

Then, when we split out the pieces and do the natural recursions, we end up with many more pieces that we would have. Of course, you won't usually need all of them – the best way is to look at each case and think about what pieces you will need. Here they are:

```
(define (bt-path-template a-bt a-path)
  (cond
    [(and (boolean? a-bt)
          (null? a-path))
     ...]
    [(and (bt? a-bt)
          (null? a-path))
     (bt-number a-bt)
     (bt-path-template (bt-left a-bt) a-path)
     (bt-path-template (bt-right a-bt) a-path)]
    [(and (boolean? a-bt)
          (eq? 'l (car a-path)))
     (bt-path-template a-bt (cdr a-path))]
    [(and (bt? a-bt)
          (eq? 'l (car a-path)))
    (bt-path-template (bt-left a-bt) a-path)
     (bt-path-template (bt-left a-bt) (cdr a-path))

     (bt-path-template (bt-right a-bt) a-path)
     (bt-path-template (bt-right a-bt) (cdr a-path))

     (bt-path-template a-bt (cdr a-path))

     (bt-number a-bt)]
    [(and (boolean? a-bt)
      (eq? 'r (car a-path)))
     (bt-path-template a-path (cdr a-path))]
    [(and (bt? a-bt)
          (eq? 'r (car a-path)))
     (bt-path-template (bt-left a-bt) a-path)
     (bt-path-template (bt-left a-bt) (cdr a-path))

     (bt-path-template (bt-right a-bt) a-path)
     (bt-path-template (bt-right a-bt) (cdr a-path))

     (bt-path-template a-bt (cdr a-path))

     (bt-number a-bt)]))
```

Contract & Purpose

Write the contract specifying the input/output types of **follow-directions** and the purpose statement specifying its behavior.

Implementation

Write the scheme code to implement **follow-directions**.

Problem 3: Sets: Binary Search Trees

Recall the data definition for binary search trees:

A **binary-search-tree** is either:

- #f
- (make-bt val left right)
  where val is a number,
  and left and right are binary-search-trees.

- INVARIANT: for each node, n, in a tree, (bt-val n) is bigger than all numbers in (bt-left n) and smaller than all numbers in (bt-right n).
- (define-struct bt (val left right))

Example Set

Write Scheme code that yields a valid binary search tree for the set {8,4,2,6,10,14,12}.

Template

Create a template for functions on binary search trees.

Union

Write a function called **union** that computes the union of two sets, represented as binary search trees. The union of two sets is a set that contains all of the elements in both sets.

Here is a function header for **union**:

```
;; union : set-of-numbers set-of-numbers set-of-numbers
;; builds a set of the numbers contained in both s1 and s2
(define (union s1 s2)
  ...)
```

Invariants

Explain briefly (2-3 lines) how your function enforces the invariant on binary search trees.

Problem 4: Binary Search Trees: Ordered Output

Write a function **inorder** that produces a list of the values in a binary search tree ordered from smallest to largest.

Contract and Purpose

Write the contract and purpose statement for the function **inorder**.

Implementation

Write the function `(inorder bst)` . Demonstrate its output the sample trees you constructed for Problem 3.

Analysis

Explain **BRIEFLY** how the structure of the binary search tree facilitates the creation of an ordered output list of elements.