

Introduction

This lab has two main purposes: (1) to give you a bit more practice working with lists in Scheme and (2) to illustrate one method for implementing abstract data types in Scheme. You'll need to remember a little about trees from your data-structures class (CIS 351); if you have trouble with the terminology, check with a labmate or the TA or me.

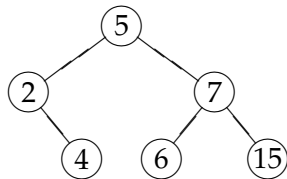
The lab is pretty straightforward; so overly complex answers indicate something is probably amiss.

Binary Trees

A binary tree is a tree where every node has either no children, a left child, a right child, or both a left and right child. Under this definition, the empty tree (i.e., the tree with no nodes) is also a binary tree. If we assume that every node has a numeric id, then binary trees can be specified by the following BNF specification:

$$\langle \text{bin-tree} \rangle ::= () \mid (\langle \text{number} \rangle \langle \text{bin-tree} \rangle \langle \text{bin-tree} \rangle) \quad (1)$$

For example, `(5 (2 () (4 () ())) (7 (6 () ()) (15 () ())))` is a valid $\langle \text{bin-tree} \rangle$ (convince yourself before continuing), corresponding to the (abstract) tree that we might sketch as follows:



Note that the role of `()` in $\langle \text{bin-tree} \rangle$ is similar to that of NULL pointers in C, indicating the absence of any nodes at all. Therefore, a *leaf* is a $\langle \text{bin-tree} \rangle$ with form `(⟨number⟩ () ())`.

For any nonempty $\langle \text{bin-tree} \rangle$, we can extract its left child using `cadr` and

its right child using `caddr`. It's good to introduce mnemonic names for these procedures:

```
(define left-child cadr)
(define right-child caddr)
```

These new names make it easier to think of each $\langle \text{bin-tree} \rangle$ as an abstract data structure rather than as a list. Moreover, they can make writing and debugging programs easier: it's much easier to remember the purpose of a `right-child` procedure call than a call to `caddr` (as, `caddr` gets used in lots of contexts!), thereby making the task of locating logical (and typographical) errors more straightforward. Similarly, we can introduce new names for the procedures that indicate whether a $\langle \text{bin-tree} \rangle$ is empty and what a nonempty node's id (i.e., number) is:

```
(define empty? null?)
(define id car)
```

To play with these definitions a bit (and to minimize the pain of typing in sample trees), introduce the following definition (it's the tree illustrated earlier):

```
(define tree1 '(5 (2 () (4 () ()))
                 (7 (6 () ()) (15 () ())))
              ))
```

Now evaluate the following expressions, and make sure you understand how they work:

```
(right-child tree1)
(left-child (right-child tree1))
(id tree1)
(empty? (left-child tree1))
(empty? (left-child (left-child tree1)))
```

Traversals

A *traversal* is a scheme for visiting each node of a tree exactly once in a particular order—typically so that some type of operation can be done at each node. For example, if we want to create a list of the node ids for some purpose, we need to decide on the order in which to list the node ids. There are three standard (left-to-right) traversal schemes: *preorder*, *inorder*, and *postorder*.

In a **preorder traversal**, a node's id is listed, followed by a preorder traversal of its left child and then a preorder traversal of its right child. For example, the preorder traversal of `tree1` can be represented as the list `(5 2 4 7 6 15)`. The Scheme code for creating this traversal list follows:

```
(define preorder
  (lambda (tree)
    (if (empty? tree)
        '()
        (append (list (id tree))
                 (preorder (left-child tree))
                 (preorder (right-child tree))))))
```

In an **inorder traversal**, the traversal of a node's left child occurs *before* that node's id is listed; the traversal of the node's right child still occurs last. Thus the inorder traversal of `tree1` gives `(2 4 5 6 7 15)`.

Finally, in a **postorder traversal**, a node's id is listed (you guessed it!) *after* the traversals of its left and right children (in that order). The postorder traversal of `tree1` yields `(4 2 6 15 7 5)`.

Your problems

1. Write a Scheme procedure `inorder` that takes a $\langle bin-tree \rangle$ `tree` and returns the list representing its inorder traversal. *Hint:* Modify `preorder` in an obvious way.
2. Write a Scheme procedure `bin-tree?` that takes an arbitrary Scheme expression and determines whether or not it is a $\langle bin-tree \rangle$.

For example:

```
> (bin-tree? '())
#t
> (bin-tree? 5)
#f
> (bin-tree? '(5 (2 () (4 () ())) (7 () ())))
#t
> (bin-tree? '(5 (2 () (4 () ())) ))
#f
> (bin-tree? '(a () ()))
#f
```

Hint: Consider equation (1).

3. A *binary search tree* is a binary tree in which every node has the following property:

Its id value is greater than that of every node of its left child and less than every node of its right child.

(That is, every node that occurs to the left of it has a smaller id; every node that occurs to the right of it has a larger id.)

Write a Scheme procedure `bst?` that takes a binary tree `tree` and returns `#t` if `tree` is a binary search tree and `#f` otherwise. *Hint:* What's special about an inorder traversal of a binary search tree?

What to Hand In Hand in your source code (*including contracts and purposes for each procedure you're asked to write*), along with verified test cases that demonstrate that the code works as required.

You **do not** have to submit anything electronically for this lab.

Due: This lab is due, in the CST 3-212 bin, at noon Friday, February 1.