Binary Trees

Let's look at another ADT today, a rather common one, and think some more about multiple implementations.

Here is a binary tree ADT:

```
empty : () -> tree
node : (int left right) -> tree

isEmpty : (tree) -> bool
root : (tree) -> int
left : (tree) -> tree
right : (tree) -> tree
```

Here, `empty` and `node` are constructors, and the rest are operations, including a predicate `isEmpty` and three selectors `root`, `left`, `right`.

Note that this ADT, just like the stack ADT from two lectures ago (but unlike the queue ADT), is very much related to data definitions that you saw in 211. Thinking about a data definition for binary trees, you write: a `tree` is either an empty tree `(empty)`, or a `(node int tree tree)`. Of course, this gives you the two constructors in our ADT. To use a data definition, you need a way to distinguish the cases, and `isEmpty` in the ADT plays that role. Finally, we need selectors to extract information for the `(node int tree tree)` case, and our ADT provides those with `root`, `left`, and `right`.

These kind of ADTs, that come from something like a data definition, are sometimes called "algebraic data types" and have a distinctly straightforward algebraic specification:

```
(isEmpty (empty)) = true
(isEmpty (node v l r)) = false
(root (node v l r)) = v
(left (node v l r)) = l
(right (node v l r)) = r
```

Here is the most direct implementation for binary trees. We first need to chose a representation: We represent an empty tree as `nil`, and a `(node v l r)` as a list containing `v`, the representation of `l`, and the representation of `r`. Here are the functions in the implementations, including a predicate for trees and one for trees equality.

```
(defun treep (tr)
  (if (endp tr)
      (= tr NIL)
    (and (integerp (first tr))
         (treep (second tr))
         (treep (third tr)))))

(defun tree-= (t1 t2)  (= t1 t2))

(defun empty () NIL)

(defun node (v l r) (list v l r))

(defun isEmpty (tr) (endp tr))

(defun root (tr)
  (if (endp tr)
      NIL
    (first tr)))

(defun left (tr)
  (if (endp tr)
      NIL
    (second tr)))

(defun right (tr)
  (if (endp tr)
      NIL
    (third tr)))
```

We can prove in ACL2 that this implementation satisfies the ADT specifications, after we express them to ACL2:

`(isEmpty (empty))`

$$\texttt{(integerp v)} \wedge \texttt{(treep l)} \wedge \texttt{(treep r)} \Longrightarrow \neg\texttt{(isEmpty (node v l r))}$$

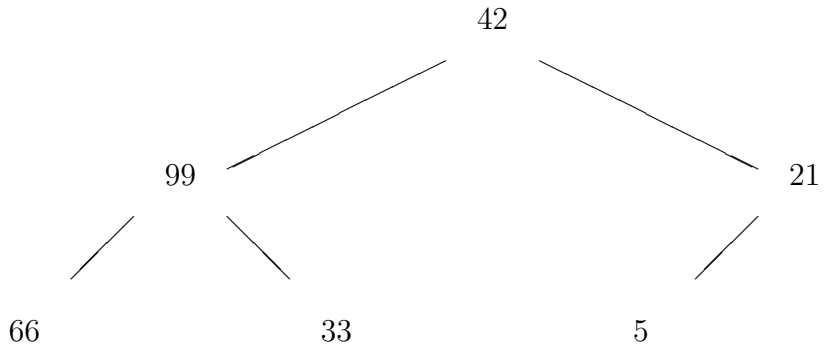
$$\texttt{(integerp v)} \wedge \texttt{(treep l)} \wedge \texttt{(treep r)} \Longrightarrow \texttt{(= (root (node v l r)) v)}$$

$$\texttt{(integerp v)} \wedge \texttt{(treep l)} \wedge \texttt{(treep r)} \Longrightarrow \texttt{(tree-= (left (node v l r)) l)}$$

$$\texttt{(integerp v)} \wedge \texttt{(treep l)} \wedge \texttt{(treep r)} \Longrightarrow \texttt{(tree-= (right (node v l r)) r)}$$

These are all easy to prove, either by hand, or using the theorem prover.

Let's look at a different implementation for trees, this one a bit less obvious. Think about a tree such as:



Label each edge with `L` if the edge leads to a left subtree, or with `R` if the edge leads to a right subtree. Now, look at every node, and record the list of edge labels you need to follow to get to that node (called a path), and the value at the node you get to. For the above tree, we get the paths and node values:

```
() -> 42
(L) -> 99
(L L) -> 66
(L R) -> 33
(R) -> 21
(R L) -> 5
```

We are going to take this as a representation of a tree: the list of all nodes, where a node is described by a path of `L`s and `R`s and an integer value. Let's look at the corresponding implementation. I've suffixed `*` to these operations, just so that we can compare them to the original implementation above.

```
;; pathp : (any) -> bool
;; check if a value is a true list of path symbols
(defun pathp (p)
  (if (endp p)
      (= p NIL)
```

```
    (and (or (= (car p) 'L)
             (= (car p) 'R))
         (pathp (cdr p)))))

(defun treep* (tr)
  (if (endp tr)
      (= tr NIL)
    (and (pathp (first (car tr)))
         (integerp (second (car tr)))
         (treep* (cdr tr)))))

(defun tree*-= (t1 t2) (= t1 t2))

(defun empty* () NIL)

;; add-edge : (L|R tree) -> tree
;; add a symbol in front of the path of every node in the tree
(defun add-edge (sym tr)
  (if (endp tr)
      NIL
    (cons (list (cons sym (first (car tr)))
                (second (car tr)))
          (add-edge sym (cdr tr)))))

(defun node* (v l r)
  (cons (list NIL v)
        (append (add-edge 'L l)
                (add-edge 'R r))))

(defun isEmpty* (tr) (endp tr))

;; the root node is always on top!
(defun root* (tr)
  (if (endp tr)
      NIL
    (second (car tr))))

;; extract all the nodes whose path start with L
;; remove that L from each path
(defun left* (tr)
  (if (endp tr)
      NIL
```

```
    (if (= (car (first (car tr))) 'L)
        (cons (list (cdr (first (car tr))) (second (car tr)))
              (left* (cdr tr)))
      (left* (cdr tr)))))

;; extract all the nodes whose path start with R
;; remove that R from each path
(defun right* (tr)
  (if (endp tr)
      NIL
    (if (= (car (first (car tr))) 'R)
        (cons (list (cdr (first (car tr))) (second (car tr)))
              (right* (cdr tr)))
      (right* (cdr tr)))))
```

I claim that the above implementation satisfies the tree ADT specification. It is straightforward (albeit a bit less easy) to prove that.

Remember last lecture when we talked about establishing observational equivalence between objects with different implementations without going through the algebraic specification, by coming up with a suitable equivalence between representations that is invariant with respect to the respective implementations? Well, we can do the same here, although the equivalence is more painful to write down. Roughly the equivalence is the one we implicitly used above to come up with the alternate representation: a standard tree `t` and a path-based tree `t*` are equivalent if `t*` lists all the nodes in `t` obtained by following their corresponding path in `t`. The trick to writing down this equivalence is to figure out a way to transform a standard tree representation in to a path-based tree representation. Here is a function that does that. It uses the helper function `add-edge` in the path-based tree implementation:

```
;; expand : tree -> tree*
;; takes a tree in the standard representation and produces a
;; a path-based representation for it
(defun expand (tr)
  (if (endp tr)
      NIL
    (cons (list NIL (first tr))
          (append (add-edge 'L (expand (second tr)))
                  (add-edge 'R (expand (third tr)))))))
```

This function looks very similar to `node*`, and there's a good reason for that. (Very roughly speaking, it's because the tree ADT is an algebraic data type, and we are trying to convert the standard representation of a tree. We don't have the math background to understand what's going on here, but there's something fundamental at work here.)

With this in hand, we can write a function `equiv` capturing the required equivalence between the two implementations:

```
(defun equiv (tr tr*)
  (= (expand tr) tr*))
```

I'll leave it to you as an exercise to show that the two implementations' operations preserve this equivalence, and if you have two trees that satisfy the equivalence, then observations on those two trees yield the same result, just like we did for queues.

Anyways, we now have two (equivalent) implementation of binary trees, pretty different at that. To illustrate this equivalence, consider defining functions based on the tree ADT. Consider the problem of binary tree *traversals*. There are three kinds of traversals you can define: preorder, inorder, and postorder traversals. A traversal is a process by which you look at every node in a tree, in some specified order. Let's say that we want to accumulate in a list all the values stored in the binary tree. The traversal order will affect the order in which the values appear in the list.

- In a preorder traversal, when you reach a node, you record the value in the node, then you recursively traverse the left subtree, then you recursively traverse the right subtree.
- In an inorder traversal, when you reach a node, you first recursively traverse the left subtree, then you record the value in the node, then you recursively traverse the right subtree.
- In a postorder traversal, when you reach a node, you first recursively traverse the left subtree, then you traverse the right subtree, and then you record the value in the node.

The prefix pre-, in-, and post- indicate when the recording of the value occurs.

Here is code implementing the above traversals, written against the ADT.

```
(defun preorder (tr)
  (if (isEmpty tr)
      NIL
    (cons (root tr)
          (append (preorder (left tr))
                  (preorder (right tr))))))

(defun inorder (tr)
  (if (isEmpty tr)
      NIL
    (append (inorder (left tr))
            (cons (root tr)
```

```
                  (inorder (right tr))))))

(defun postorder (tr)
  (if (isEmpty tr)
      NIL
    (append (postorder (left tr))
            (append (postorder (right tr))
                    (list (root tr))))))
```

Because they are written again the tree ADT, these functions work for all tree implementations that satisfy the ADT specifications.

In fact, you can use the specification to predict exactly what the result of calling `postorder` on a tree constructed using `empty`s and `node`s is. It just requires a bit of symbolic evaluation.