1. Generic/Binairy Trees

If all of our data structures (and even our programs!) are ultimately expression trees, then we should expect to spend a lot of time manipulating trees. Indeed, inasmuch as natural numbers in successor notation and lists are expression trees, we have already spent some time doing just that. But lists and natural numbers are rather special cases of trees. These structures are of course *technically* trees, but they're really not very `tree-like'. What does this mean? What does it mean to be ``essentially tree-like?'' Let's begin by comparing the technical recursive/inductive definitions of successor-notation and lists, and see how we can make the concepts we find there more general.

- Definition of *NN*:
    1. *Basic = {0 },*
    2. *Gen = {F_s },* where we define F_s *a*s follows*:*

$F_s(t) = s(t)$.

- Definition of *Lists*:
    1. *Basic = {[] },*
    2. *Gen = {F_{exp} | exp* is an expression tree }, where we *define F_{exp} as follows:*

$$F_{exp}(t) = [exp|t] .$$

We observed in the last section that these structures are similar in that their generator functions take only a single argument. To obtain trees, we simply relax this restriction and allow generator functions which take multiple arguments. So with successor notation and lists we have trees with an essentially *linear* structure, because the functions which generate them are recursive in only one argument position. Trees, whose generator functions are *polyadic*, not necessarily monadic, are *multiply recursive* and essentially nonlinear because their generator functions are recursive in several argument places. So we have essentially the following hierarchy of generalizations:

- The natural numbers form a mathematical structure with a single monadic function.
- The lists form a mathematical structure with *many* monadic functions.
- The trees form a mathematical structure with *many polyadic* functions.

For our purposes it is perhaps best to think of a tree as being like a `sequence' with one head and multiple tails, each of which is (recursively) itself a `sequence' with one head and multiple tails, each of which is ...

```
list( L ) :- empty( L ).
list( L ) :-
        head( L, _ ),
        tail( L, LTail ),
        list( LTail ).


tree( T ) :- leaf( T ).

tree( T ) :-
        root( T, _ ),           % ``head''
        subtrees( T, SubTrees ), % ``tails''
        forest( SubTrees ).      % Is each tail a tree?


forest( LT ):- empty_forest( LT ).
forest( LT ) :-
        first( LT, LT_First ),
        rest( LT, LT_Rest ),
        tree( LT_First ),
        forest( LT_Rest ).
```

Essentially, we treat the root of a tree as we would treat the head of a list: this is the main similarity between trees and lists. On the other hand, the multiply recursive nature of trees means that a whole new dimension is introduced into our treatment of a tree's tails. That is, the definition of the tree data type given here is *doubly recursive*. Intuitively, we must travel recursively *down* the tree and also *across* the sequence of subtrees at any given node.

1.1 How to write trees

In the case of lists, there was a special representation already built into Prolog. In the case of trees we are on our own. So the way of encoding trees which I will present to you here should in no way be considered the absolute and inviolable standard way to do things. There is much more variety among programs and programmers in representations for trees than there is in representations of sequences. We will use the following scheme to encode trees.

First, we define a forest to be a *list* of trees. So we can define empty_forest/1 as the empty list, [], and first/2 and rest/2 as the head and tail of a list, respectively. So the definition of forest/1 can be rewritten as follows.

```
forest( [] ).
forest( [T1|RestTs] ) :-
    tree( T1 ),
    forest( RestTs ).
```

A tree is the pairing of a root with a (possibly empty) forest of subtrees. So we need some 2-place functor to tie these two data items together. Here we will use `/'/2, the division symbol. Because this can be written infixed between its arguments, it makes tree terms a little more readable. Once again: there is nothing sacred in this representation; in particular, one might prefer a prefixed functor like t or even tree. Using the scheme we have adopted here, though, we would implement tree/1 as follows.

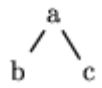
```
tree( _Root/[] ).\quad\% A leaf has no subtrees.

tree( _Root/SubTrees ) :-

    SubTrees = [_|_],\quad\% SubTrees is non-empty;

    forest( SubTrees ).
```

For example, we would represent the tree



as the Prolog term a/[b/[], c/[]]. The proof that this is a tree is straightforward and is summarized in figure 1

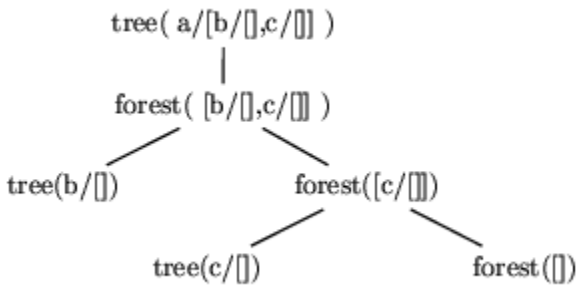


Figure 1: Proof tree for the goal: ?- tree(a/[b/[],c/[]]).

The definition I have given in tree/1 is very general. Often we can take advantage of particular circumstances to derive more restricted definitions. One of the most common subtypes of tree is the binary tree. A binary tree is a tree in which every node has exactly two or zero subtrees. We have already seen one example of this structure: lists are a sort of binary tree, except that they are only recursive in their second subtree. We can represent binary trees as 3-place functions, with one argument for the root, as usual, and the other two argument places for the left subtree and the right subtree.

```
bintree( bt( Root, [], [] ) ).
bintree( bt( Root, LTree, RTree ) ) :-
   bintree( LTree ),
   bintree( RTree ).
```

Clearly we can generlize this to trinary, quaternary and in general *n*-ary trees, for any fixed *n*, as follows.

```
 ntree( n( Root, [], ... , [] ) ).
ntree( n( Root, T1, ... , Tn ) ) :-
      ntree( T1 ),
       ... ,
       ntree( Tn ).
```

Our original example was a binary tree. Recoding it as a bintree would yield the following term.

bt(a, bt(b, [], []), bt(c, [], []))

The representation of the data structure is less readable, perhaps (though no `one-dimensional' representation of a tree will ever be really readable!), but we can process binary trees without ever calling anything like forest/1. That is, instead of one call to tree/1 and one to forest/1, we have two calls to bintree/1. The proof tree corresponding to figure 1 is given in figure 2. Note that the proof tree in this case much more closely reflects the structure of the underlying tree. With our more flexible slash-notation we have to focus on the calls to tree/1 and ignore the calls to forest/1, treating them as essentially just some sort of glue.

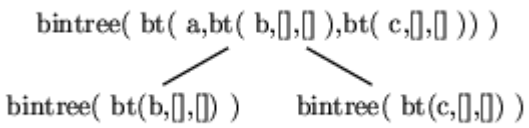


Figure 2: Proof tree for the goal ?- bintree (bt (a, bt (b,[],[]), bt (c,[],[]))).