Handling Trees in Prolog

Binary Trees in Prolog

- A *binary tree* is a finite set of elements that is either empty or is partitioned into three disjoint subsets.
  - The first subset contains a single element called the *root* of a tree.
  - The other two subsets are binary trees themselves called *left subtree* and *right subtree* of the original binary tree.
- Each element of a binary tree is called a *node* having three arguments namely, Value, Left_subtree and Right_subtree.

Representation of Binary Tree

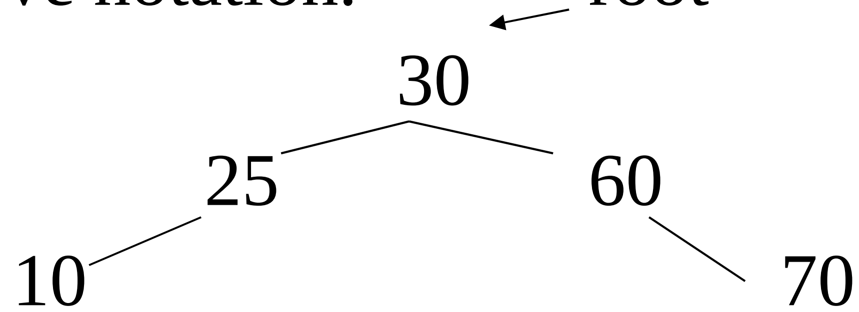
- The binary tree in Prolog is represented by ternary functor, say,

  b_tree(Value, Left_subtree, Right_subtree),

- The empty binary tree is represented by an atom called *void*.

Contd..

- Let us represent the following binary tree using above notation. root



- Prolog representation of binary tree

b_tree(30, b_tree(25, b_tree(10, void, void), void),
b_tree(60, void, b_tree(70, void, void)))

Binary Tree Manipulation

- Binary trees are manipulated in the similar way as the lists are done.
- Binary tree can also be used as an argument of a predicate just as list and object are used.
- Operations on Binary tree
  - Traversal,
  - Copying,
  - Counting nodes,
  - Swapping the branches etc.

Traversal of Binary Trees

- Binary trees are traversed using recursion, mainly, in preorder, inorder and postorder.
  - ***Preorder Traversal***: visit root, traverse left subtree in preorder, right subtree in preorder.
  - ***Inorder Traversal*****:** traverse left subtree in inorder, visit root and right subtree in inorder.
  - ***Postorder Traversal*****:** traverse left subtree in postorder, right subtree in postorder and visit root.

Prolog programs

- Prolog program for preorder traversal.

/* pre_btree(T, Y)- Y is unified with a list of nodes obtained by traversing T in preorder.*/

```
pre_btree( void, [ ]).                                   (1)
pre_btree(b_tree(X, L, R), Y):-
pre_btree(L, L1),  pre_btree(R, R1),
                   append([X|L1], R1, Y).   (2)
```

- Membership program

/* mem_btree(X, T)- succeeds if X is member of tree T */

```
mem_btree(X, b_tree(X, L, R)).
mem_btree(X, b_tree(Y, L, R)) :- mem_btree(X, L).
mem_btree(X, b_tree(Y, L, R)) :- mem_btree(X, R).
```

Goal: ?- mem_btree(10, b_tree(35, b_tree(25, b_tree(10,void,void), void), b_tree(62, void, void))).

Negation as Failure

- Horn clauses are incomplete version of FOPL because of the limitation of one positive literal in a clause.
- Here we describe an extension to the LP computation model that allows a limited use of negative information in the program.
- A goal ***not(G)*** is said to be a *logical consequence* of a program P if G is not a logical consequence of P. In other words, if goal G can not be shown to be true, then infer the truth of not(G).

Contd…

- A goal *not(G)* is implied by a program P by the *negation as failure* rule.
- The cut-fail combination can be used to implement a version of negation as failure.
- It is difficult to implement negation both efficiently and correctly.

```
not(G)    :-    G , !, fail.
not(G).
```

- The cut ensures that if G succeeds, the second clause will not be attempted and if G fails, then cut is not activated and so second clause is tried and it succeeds.

Contd…

- Therefore, if goal G succeeds, then not(G) fails and if G fails, then not(G) succeeds.

Goals:

?- not(2 < 4).	Answer: No
?- 2 < 4	Answer: No

- It is a good programming style to replace *cut* by the use of *not* if possible because the programs containing cuts are generally harder to understand.
- The rule using *not* predicate is more readable and gives clear semantic interpretation of the rule but at times could be computationally expensive.

Contd..

- The Prolog rule P :- Q_1, Q_2, ...,Q_n expresses only if condition for P and it says nothing about other conditions under which P can be true.
- *Negation as failure* introduces a closed world in the limited sense. Every thing not stated is taken to be false.
- Predicate *if_then_else* can be implemented using using **not** predicate instead of cut.

  if_then_else(P, Q, R) :- P, Q.

  if_then_else(P, Q, R) :- not(P), R.

  - Here the goal P have to be computed again while trying second rule of if_then_else.

Logical Limitations of Prolog

- Prolog does not allow disjunction ('or') of facts or conclusion such as

  "If car does not start and the light does not come on, then either battery is down or problem with ignition or some electric fault"

- Such rules can not be expressed straight away in Prolog.
- Prolog does not allow negative facts or conclusions e.g., not(a) :- b ; not(c) etc are not valid in Prolog.
- Prolog does not allow facts, rules having existential quantifications.

Incomplete Data Structure

- Data structures which are incomplete or having holes are useful in many applications.
  - Applications will be discussed later.
- Incomplete list is an example of such structures.
  - For example, [1,2,3 | X] is an incomplete list whereas [1,2,3,4] is a complete list.
- First we will discuss difference list, an alternative data structure for representing a list.
- Consider a complete list [1, 2, 3]. We can represent it as the difference of the following pair of lists.

Examples

- Examples of difference list.
  - [1, 2, 3, 5, 8] and [5, 8]
  - [1, 2, 3, 6, 7, 8, 9] and [6, 7, 8, 9]
  - [1,2,3] and [ ].
- Each of these are instances of the pair of two incomplete lists [1,2,3 | X] and X.
- We call such pair a *difference-list*.
- We denote the difference list by A-B, where A is the first argument and B is the second argument of a difference-list A-B.
- A list [1,2,3] is represented using difference-list as [1,2,3|X] – X
- Such representation of list facilitates some of list operations more efficiently.

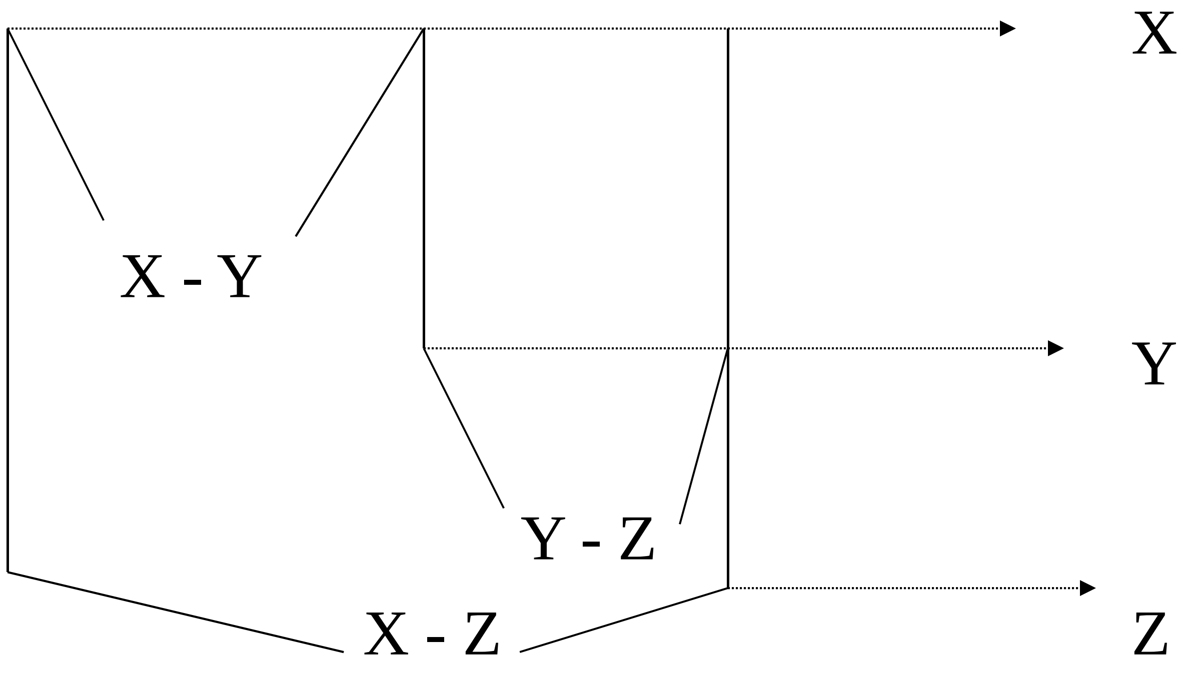
Example–Concatenation of two lists

- Concatenating two lists represented in the form of difference lists.
- When two lists represented as difference lists are concatenated (or appended), then we get appended list by simply unifying the appropriate arguments as given below: One line program.

diff_append (A - B, B - C, A - C).

Graphical representation

- Graphical representation of append program for difference lists:



Query for append predicate

- Let us append two lists [1,2,3] and [4,5,6] using above rule.
- Represent both complete lists using difference list notation.

  [1,2,3] = [1,2,3 | X] – X

  [4,5,6] = [4,5,6 | Y] - Y
- Append Rule

  diff_append (A - B, B - C, A - C).

Goal: ?- diff_append([1,2,3 | X] - X , [4,5,6 | Y] - Y, N).

Search Tree

Append Rule: *diff_append (A - B, B - C, A - C).*

?- diff_append([1,2,3 | X] - X , [4,5,6 | Y] - Y, N).

{A = [1,2,3 | X], B = X = [4,5,6 | Y],

C =Y, N = A - C=[1,2,3,4,5,6 |Y] - Y}

succeeds

Answer: N = [1,2,3,4,5,6 |Y] - Y

- Note:
  - This program can not be used for concatenating two complete lists.
  - Here each list is to be represented using difference-list notation. There are nontrivial limitations to this representation because the first list gets changed.