

COMP 356
Programming Language Structures
Class 26: Additional Prolog notes

The main topic for today's class is "Deficiencies of Prolog." For this topic, please refer to section 16.7 of the textbook. These notes provide some background on several small additional topics, which may help with the homework.

1 Comma notation in list heads

We already discussed the meaning of Prolog fragments like `[A|B]`. This notation can be extended using commas to match multiple items at the head of the list. For example, `[A,B|C]` matches a list whose first two elements are A, B, with the remainder of the list being C.

2 The "not proven" operator

The Prolog operator `\+` negates the meaning of its argument. But it's not the same thing as the Boolean operator `!` in C or Java. For example, the query `\+(student(sophie))` will terminate with "success" or "true" if the system cannot prove that Sophie is a student. If the system can prove that Sophie is a student, this query terminates with "failure" or "false". Important notes:

1. You need to be especially careful when using `\+` with variables. For certain technical reasons that will not be covered in this course, `\+` should only be used with variables that have already been assigned a value. So an expression like `\+(student(X))` should only be used after some other expression that will have already assigned a value to X. For example, `takingCourse(X,progLang), \+(student(X))` is acceptable, because Prolog will always assign some value to X in `takingCourse(X,progLang)` before moving on to prove the goal `\+(student(X))`. In practice, this means you should put `\+` clauses at the *end* of any rule.
2. Many Prolog systems permit the use of a built-in relation `not()` in place of `\+`. In particular, the examples in the textbook use `not()`. However, XGP does not permit this, so you will need to use `\+` instead of `not()`.

3 More sophisticated data structures

Prolog is capable of representing sophisticated data structures. As a simple example¹ of this, the code given in Figure 1 can be used to represent, manipulate, and query a sorted binary tree that stores a single integer data value at each node. The basic idea is to define a relation `node(L, D, R)`, where L represents the left child of the node (which could be the special value `empty`), R represents the right child of the tree, and D represents the data value stored at the node. Please see the accompanying file `tree.pl` to experiment with these definitions.

¹This example is based closely on Professor Wahls' lecture notes.

```

/* relation to check whether some value occurs in the tree */
isin(K, node(_, K, _)).
isin(K, node(L, D, R)) :- K < D, isin(K, L).
isin(K, node(L, D, R)) :- K > D, isin(K, R).

/* adding a node at the proper position in the tree with no attempt at
   balancing the tree */
insert(K, empty, node(empty, K, empty)).
insert(K, node(L, D, R), node(L2, D, R)) :- K < D, insert(K, L, L2).
insert(K, node(L, D, R), node(L, D, R2)) :- K > D, insert(K, R, R2).

/* do an inorder traversal of the tree, accumulating the node values in
   a list */
inorder(empty, []).
inorder(node(L, D, R), Z) :- inorder(L, LL), inorder(R, RL),
                             append(LL, [D], Z1),
                             append(Z1, RL, Z).

```

Figure 1: Prolog code for a sorted binary tree storing integers.