Homework Assignment #4

Due: Before you get a Schein

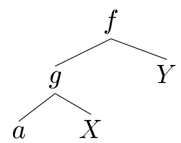
Problem 4.1. Consider the following definition of `flatten/2`, a relation which takes a list of lists of lists of ... and "flattens" them, i.e., converts them to a list of atoms.

```
flatten( [], [] ).
flatten( X, [X] ):- atomic( X ).
flatten( [L1|Ls], FlatList ):-
        flatten( L1, Flat1 ),
        flatten( Ls, Flat2 ),
        append( Flat1, Flat2, FlatList ).
```

The SICStus builtin `atomic/1` succeeds if its argument is either a number or an atom. That is, it fails whenver its argument is bound to a structure or a variable. In this example we are using it to pick out the things that aren't lists. (Note: we'll get in trouble if our lists of lists of lists contain trees or any other non-list structure, or any unbound variables. For this assignment, assume that the ultimate atomic elements of these lists are indeed all simple atoms and numbers.)

Reimplement `flatten/2` so that it uses difference lists to get rid of the call to `append/3`.

Problem 4.2. We know that the only real data structure built into Prolog is expression trees. We also have our own way of representing trees and, if you think back to the boolean algebra problem, our own way of representing variables. So the abstract tree



can be represented in either of the following ways:

`f(g(a,X),Y)` `f/[g/[-a,+1],+2]`

The latter way of representing a term, which I will refer to from now on as the *ground representation*, allows us to manipulate it in arbitrary ways *without ever leaving the logical core of pure Prolog*. In particular, it allows us to write pure Prolog implementations of unification algorithms, for example.

One reason you might want to do that is to implement a variant of unification which includes an occurs check, which is the topic of this exercise.

There are other reasons as well: Prolog is a very good language for developing other languages, such as grammar formalisms, knowledge representation systems, database systems or more advanced logic programming systems. Developing a programming language means developing programs (in Prolog) that manipulate programs in that language. So you need to be able to take terms apart and put them back together again at will.

In this exercise, however, we will just focus on implementing the occurs check, for terms in the ground representation.

Implement this: Implement a predicate `g_occurs_in(GVar,GTerm)`, which succeeds just in case there is an instance of the "variable" (i.e., a Prolog term of the form '`?_`') `GVar` in the (ground representation) `GTerm`. For example,

```
?- g_occurs_in(?1,f/[g/[a/[],?1/[]],?2/[]]).
```

should succeed, but

```
?- g_occurs_in(?3,f/[g/[a/[],?1/[]],?2/[]]).
```

should fail.

Don't forget to include the following code at the beginning of your program!

```
:- op(200,fx,?).
```

Problem 4.3. The tree structures we have looked at so far are all very well if you only need to travel down into the tree: you just use a recursive search predicate. But what if you are looking at a particular subtree and you discover you need to find some node in the tree above it? This is not so easy!

To make the example concrete, suppose you are building a parser for some natural language. All natural languages contain certain kinds of elements which require the presence of other licensing elements higher up in the tree. The classic example is gaps, as in *Which book do you think John will read __ next?*. The "filler" for this gap is the phrase *which book*, which appears way above and to the left of the gap position in the phrase structure tree. But gaps are just the tip of the iceberg. Reflexive pronouns require antecedents; "bound variable" pronouns also must have antecedents within the sentence; negative polarity items need special kinds of licensing contexts (compare *John doubts that anyone will come* with **John hopes that anyone will come*); the list goes on. This, then, is a case where you might well want to be able to climb up a tree as easily as you can climb down it.

One way to do this is to use a graph to represent a tree. Then, with every node record, we store the name of that node's parent. We are already storing the names of that node's descendants; the addition of these "back pointers" makes travel in all directions possible. For example, instead of just using tree addresses to find our way around in a tree we could now use lists like `[up, up, left, down]`. With the help of parent-pointers we can follow such a path from any node in a tree. (Note that this idea is the basis of some very interesting approaches to grammar based on Modal Logic: I'm not just making this up!)

Implement this: First, you are to implement a predicate

`tree2graph(Tree,Graph),`

which takes a tree in our standard `Root/Subtrees` notation and converts it to a graph. Nodes in the graph are represented as terms of the form

`(ParentName*Label)/Neighbors.`

You are to implement the graph this time using *arrays*, instead of lists. A node's name will be the position in which it is stored. So, if we find that node 2 is a neighbor of node 3, then to find the information about node 2 we need to find the array cell at index 2. This is a little tricky! Remember that you have to keep track of the current parent node while you are processing its daughters, since the name of the parent is part of the label of each daughter. You will also need to keep track of the next available array cell. So, after you add the root of the tree to the graph, the next available cell will be cell 2. As you create nodes, the name of each new node will just be the index of the next available array cell. Also remember that you can store a `(P*L)/Nbrs` record in the array *before you know what the names of the neighbors are.* Later processing can fill in values for `Nbrs`.

By convention, the parent of the root will be 0.

This one is a little tricky, but it can be done in less than a dozen lines of code. If you find yourself writing much more than that, you may want to get away from the keyboard and sit down somewhere with a pencil and paper.

Implement this: Now implement a predicate

`wander(Path,Start,TreeGraph,End),`

where `Path` is a list with elements from the set {`up`,`down`,`left`,`right`}. (Feel free to add other instructions if you like!) Note that in a tree the `down` instruction is nondeterministic. (You might consider extending the language with an instruction `down(N)`, where the integer `N` indicates which branch to descend. By `left` and `right` I mean the *immediate* left and right sisters of a node, so these are deterministic instructions, as is `up`. `Start` is a node name, namely the node at which the first instruction in `Path` is executed. `End` should wind up being unified with the name of the node at the end of the path. Obviously, if the path falls off the edge of the tree, the predicate should fail.

Optional: If you are feeling ambitious, you might consider some ways of extending this "path language". For example, you might want to add something like `repeat(D)`, where `D` is one of the four directions, which allows you to move in direction `D` one or more times, or something like `or(D1,D2)`, which allows you to move in either direction `D1` or `D2`, or something which allows you to test the label on a node. With extensions like these, you should have no trouble defining the c-command relation, for example.

Problem 4.4. Consider the graph

```
G = [1-2,2-3,1-3]
```

In a sense the arc from node 1 to node 3 is superfluous. If we needed to get from node 1 to node 3 we could always go via node 2.

One instance of such a problem is the following. Suppose we have a collection of statements of the form "node x dominates node y" (in some tree). By "dominates" we mean here strict dominance: x is either the parent of y or its grandparent or its great-grandparent or .... But it is not equal to y.

It is quite possible to have a collection of such dominance assertions which is inconsistent. For example, if I claim that node 1 dominates node 2 and also that node 2 dominates node 1, then it is provable that there is no tree that makes both these assertions true.

Obviously, such a collection of constraints can be represented by a graph, which contains an edge for every dominance constraint in the input. One way to tell if the set of dominance constraints is consistent is by actually exhibiting a tree that satisfies them. In such a tree:

1. All the nodes in the graph appear as nodes in the tree.
2. Every arc in the original graph is a path in the derived tree.

So there is a fairly obvious nondeterministic solution to this problem:

1. Nondeterministically select a root from among the nodes in the input graph.
2. Construct a traversal tree rooted at that node such that it satisfies the conditions above.

Implement a predicate `witness/2` such that `witness(Graph,Tree)` succeeds just in case `Graph` is a consistent set of dominance constraints which is satisfied by `Tree`. Assume that the input *Graph* is represented as a list of arcs `X-Y` interpreted as meaning that `X` dominates `Y`. For example, the goal

```
?- witness([1-2,3-1,3-2],3/[-1,-2]).
```

should succeed. But the following goal should fail.

```
?- witness([1-2,2-3,3-1],X).
```