600.432/632 Declarative Methods - Prof. Jason Eisner
Answers to "Prolog Continued" discussion problems

1. The following solution gets partial credit: it works on all pairs of Peano integers, but it also mistakenly thinks that geq(three,z) is true:

```
geq(_,z).
geq(s(A),s(B)) :- geq(A,B).
```

This version gets full credit by patching that bug directly, but it is somewhat ugly:

```
geq(A,z) :- peano(A).
geq(s(A),s(B)) :- geq(A,B).

peano(z).
peano(s(A)) :- peano(A).
```

This prettier version also gets full credit. (It is related to the previous version by a program transformation.)

```
geq(z,z).
geq(s(A),z) :- geq(A,z).
geq(s(A),s(B)) :- geq(A,B).
```

There are other solutions. In particular, a good approach is to reduce it to a previously solved problem:

```
geq(A,B) :- sum(B,C,A).   % A >= B iff there is some C >= 0 with A=B+C
```

But then for full credit, you have to give an appropriate definition of sum/3. This standard definition won't work because it will allow sum(z,three,three) and therefore geq(three,z) again:

```
sum(z,B,B).
sum(s(A),B,s(C)) :- sum(A,B,C).
```

You'd need this version of sum for full credit:

```
sum(z,z,z).
sum(z,s(B),s(C)) :- sum(z,B,C).
sum(s(A),B,s(C)) :- sum(A,B,C).
```

2. a.
```
nth(z, [X|_], X).
nth(s(N), [_|Rest], X) :- nth(N, Rest, X).
```

b. It should give 2 solutions:

```
N = s(s(z))
N = s(s(s(z)))
```

c. Yes, it finds these two solutions then terminates.
Nice to be working in pure Prolog, eh?

d. This also behaves as expected, giving [_, b|_] as the solution.

3. a. Arc consistency. By assigning a vowel to X5, arc consistency restricts X4's domain to consonants, and then arc consistency restricts X3's domain to vowels, etc.

b.
```
alternating([]).
alternating([_]).
alternating([X,Y|Rest]) :- vowel(X) #\= vowel(Y), alternating([Y|Rest]).
```

4. The answers are:

```
X = a ;
X = a ;
X = d ;
X = d ;
No
```

To understand this, let's look first at foo. Ordinarily the subgoal member(A, List) would try to bind A to each element of list in turn. However, the cut prevents it from backtracking, so A is only bound to the first element of List. Thus, foo is just a (silly) method for getting the first element of a list. It is equivalent to

```
foo(List,A) :- List=[A|_].
```

or to

```
foo([A|_],A).
```

Now, how about bar(X)? The backtracking proceeds like this:

```
goal: bar(X)
    subgoal: member(List, [[a,b,c],[d,e,f],[g,h,i]])
    1.  succeeds with List=[a,b,c]
        subgoal: foo([a,b,c], X)
        1.  succeeds with X=a
            subgoal: member(a, [a,a,b,b,c,c,d,d,e,e,f,f])
            1.  succeeds
            2.  succeeds again (another copy of a)
            No (no more copies of a)
        No  (the cut prevents foo from finding more members)
    2.  succeeds with List=[d,e,f]
        subgoal: foo([d,e,f], X)
        1.  succeeds with X=d
            subgoal: member(d, [a,a,b,b,c,c,d,d,e,e,f,f])
            1.  succeeds
            2.  succeeds again (another copy of d)
            No (no more copies of d)
        No (the cut prevents foo from finding more members)
    3.  succeeds with List=[g,h,i]
        subgoal: foo([g,h,i], X)
        1.  succeeds with X=g
            subgoal: member(g, [a,a,b,b,c,c,d,d,e,e,f,f])
            No (g is not a member)
        No (the cut prevents foo from finding more members)
```

5. a. B=8, Answer=[1,5,8,3].

   b. Bs=[1,8,0,9,_G100], Cs=_G100, Answer=g([9,2,4,1,8,0,9|_G100], _G100).

   c. This is basically a technique for quickly appending two lists. How would you do that in a procedural language? Imagine a Java or C++ linked list object that stores a pointer not only to its first element but also to its last element. This pointer makes it easy to jump instantly to the end of the list, making it fast to destructively append by pointing the end of the list to the start of another list.

      In Prolog, the analogous technique uses a special representation of lists, called "difference lists." Instead of [9,2,4], we want [9,2,4|Bs] so that we have room for expansion at the end. We can fill in the rest of the list just by unifying Bs with something -- perhaps [], perhaps a long list. (What's more, we can backtrack if that destructive modification doesn't work

out.)

But as in Java or C++, we don't want to have to iterate down the whole list to find and modify Bs. (E.g., we don't want to have to remove 9,2,4 and then add them back again -- that would take O(n) time.) We want our list object (term) to store something like a quickly accessible pointer that points right to the end of the list.

In Prolog, instead of a pointer to Bs, we just store another *copy* of Bs. That "shallow copy" of Bs (available rapidly as the second argument of g/2) is *unified* with the deep copy of Bs (buried at the end of the list). So unifying a new value with the shallow copy will also unify it with the deep copy!

Thus, our final encoding of the list is g([9,2,4|Bs],Bs). This is basically an object consisting of the incomplete list [9,2,4|Bs] together with a "pointer" Bs to the end of that list.

engima/3 is written so that all three arguments are difference lists. So we encode the second list also as a difference list, g([1,8,0,9|Cs],Cs). And the result is also a difference list, g([9,2,4,1,8,0,9|Cs],Cs) -- ready to be appended to something else.