CSC 231 Spring 2015 **Wake Forest University**
Lab 10: Lists, Recursion, and Backtracking in Prolog

Overview of Lab

This lab is intended to introduce you to more advanced topics in Prolog. It will focus on representations and algorithms for list data-structures. It will also discuss issues to be concerned with when writing recursive code in the backtracking-search environment which Prolog employs.

Lists in Prolog

A list in Prolog is an ordered collection of items. The items can be primitive items, such as constants and variables, or high-level data-structures, such as other lists. The basic syntax for a list is the same as in Erlang – open and close square brackets, with comma separated items within the brackets.

```
[]                          // an empty list
[1,2,3]                     // a three element list
[the, cat, sat, [on, the mat]]   // a four element list, one of those elements is a
                            // three element list itself
```

A special pattern can be used to grab the head (the first element of the list) and the tail (the remainder of the list, when the head is taken away, still viewed as a list) into variables. This is as follows:

[X | Y]

X would match the head, Y the tail.

Examples of facts with lists and and examples of queries that cause matches to those facts are:

```
somePredicate([1,2,3]).
somePredicate([the, cat, sat, [on, the mat]]).

?- somePredicate([X|Y]).
X = 1          Y = [2,3] ;
X = the        Y = [cat, sat, [on, the, mat]]

?- somePredicate([_,_,_,[_|X]])      // note how the _ are anonymous
X = [the, mat]
```

The empty list has no head or tail. A one item list has a head and an empty tail.

Given a list, functions on the list can be defined, typically using recursion.

As an example, a set *member* function requires determining if an item is in the list. A recursive implementation would be to:

> See if the head matches the item being searched for
> If the head doesn't match, see if the item being searched for is a member of the tail of the list.

Corresponding definitions in Prolog are:

```
memberOfSet(X,[X|_]).
memberOfSet(X,[_|Y]) :- memberOfSet(X,Y).
```

Think about what this definition is telling us… there is a clause that says that a variable X is a member of a list if X is the first item on the list, and a separate clause that says we can infer X is a member of a list whose tail is Y (no matter what the head of that list is) if we can show X is a member of the tail Y.

I called this function *memberOfSet* since there is already a list *member* function built into Prolog.

Note this style of programming is a lot like using Erlang patterns. Using Erlang, we could have written the following to define the member function:

member(X,[ ]) → false;
member(X,[X|_]) → true;
member(X,[_|Y] → member(X,Y).

Note that we do not have to write the false (empty set) case when writing in Prolog. In Prolog, that case is handled by omission. The only facts and rules defined in Prolog so far are those for showing something true. If something can't be proved/inferred true in Prolog, it eventually returns false.

If desired to use, there is a *not* predicate built into Prolog. It is satisfied when the predicate and objects it is passed are not satisfied in any way. An example of its usage is:

notInSet(X,L) :- not(memberOfSet(X,L)).

Make sure you understand the meaning of what you are writing. For example, given a database of *friend(A,B)* relationships, a query such as *not(friend(jane,X))* is true only when *jane* has no friends at all [i.e. if there are no *friend* relationships listed in the database as facts or that can be inferred that satisfy the goal *friend(jane, X)*].

Going back to list functions, finding the length of a list requires:
Returning zero if the list is empty
If the list is not empty, returning 1 + size of the list without the head.

Remember that "returning a value" is actually setting an output variable.

listlength([ ],0).
listlength([_|Y],L) :- listlength(Y,Z), L is 1+Z.

These rules are essentially are telling us that the length of an empty list is 0, and that we can infer the length of a list with any head item and a tail Y from finding the length of Y and adding 1.

The listlength example above could also have been written using another technique – an accumulator variable. The accumulator variable is an explicit variable keeping track of the current number of items counted so far. Using this approach, the list shortens while our accumulator variable increases. At the end, the base case (for the empty list) stores the accumulated count into an output variable.

listlength2(L,N) :- lengthAccumulator(L,0,N).

lengthAccumulator([ ],A,A).
lengthAccumulator([H|T],A,B) :- LargerSize is A+1, lengthAccumulator(T,LargerSize,B).

Let L be the list of interest, N be the size of the list, and the middle column of the argument to lengthAccumulator be the current length seen so far. These statements are then essentially telling us that we can satisfy our 'goal' listlength2(L,N) by satisfying lengthAccumulator(L,0,N) [the 0 saying we haven't seen any parts of the list yet]. lengthAccumulator itself can be satisfied in two ways: We can infer that with an empty list, the output length is the same as the length accumulated up to this point (thus the [],A,A parameters], and second, we can infer the length of a list with a head and tail by adding 1 to the current length and then satisfying the lengthAccumulator function on just the tail.

A built in function called **append** is used to concatenate two lists together. Append has the following syntax: **append(A,B,C).** C will hold the result of adding all of the elements from B onto the back of A.

Using this, we can generate lists that are the concatenation of two lists:
?- append([a,b],[c,d],C).
C = [a, b, c, d]

You might find it helpful to see the actual definition for append – it is defined recursively as follows:
append([ ],L,L).
append([X|L1],L2,[X|L3]) :- append(L1,L2,L3).

Note that this says:
Base Case: Appending a list L onto the back of an empty list gives the original list L.
Recursive Case: Appending a second list onto the back of a first list is the first element of the first list hooked to a tail that is (the output from appending the 2nd list to the tail of the first list). This essentially reverse prepends all of the elements from the first list onto the 2nd list.

A Larger List Example: Generating English Sentences
In this example, we will exploit the **append** implementation and **lists** to investigate the process of generating English-like sentences. In essence, what we will need to do here is to map a grammar (a sentence generator) into a set of Prolog statements. We'll start with generating very simple sentences such as "john likes mary" – these all have the form *noun, verb, noun*. The goal is to have Prolog return us a list which is a sentence constructed appropriately according to our grammar rules.

Describe the terminals in the language (things that can't be broken down further).
word(john).
word(likes).
word(hates).
word(mary).
word(wine).

Describe valid decompositions (grammar rules/productions).
compose(noun, [john]).
compose(noun, [mary]).
compose(noun, [wine]).
compose(verb, [likes]).
compose(verb, [hates]).
compose(verb_phrase, [verb, noun]).
compose(sentence, [noun, verb_phrase]).

We also need code to describe a general technique for doing decompositions in a top to bottom, left to right per level, manner. This code is described below:

Let's define a function *build(X,P)* that says to take a one step derivation in the grammar, determining what X decomposes into. The base case is if you are at a terminal (a word that can't be broken down any further), wherein you should just return the word in a list. The other case is if X decomposes *(which you find by compose(X,Subtype) matching)* into a list of non-terminals, wherein that list needs to be handled piece by piece.

build(X,[X]) :- word(X).
build(X,P) :- compose(X, Subtype), buildlist(Subtype,P).

The next set of rules detail how to move left to right through a list of decompositions to perform. The results are generated by pre-pending the leftmost decomposition onto the front of the decompositions from the rest of the list. The base case manages when you run out of further decompositions (on the same level, left to right) to execute.

buildlist([ ],[ ]).
buildlist([H|T],Result) :- build(H,HeadComposition), buildlist(T,TailComposition),
append(HeadComposition, TailComposition, Result).

Finally, we need one more rule to say how to derive a sentence – A sentence X is derived if we can build it starting from the top-level grammar item *sentence*.

deriveASentence(X) :- build(sentence,X).

Given the grammar we defined above, using deriveASentence(X) inside the Prolog interpreter will start the process of generating all words from this grammar.

Strings

In Prolog, strings are really just integer lists – that is, a string literal is actually mapped back to a list of integers, where the integers represent the ASCII characters that make up the string. To write a string literal, such as the word *John* in a list (which Prolog would normally see as a variable because it's uppercase), put double quotes around the string. Since strings are worked with as lists, we can write a string length function by passing off work to the listlength function (written earlier in the lab).

stringlength(X,L) :- listlength(X,L).
?- stringlength("helloworld",L).
L = 10

A method for checking if two strings are equal can be defined by exploiting the fact that they are integer lists and testing their integer list representations for equality:

stringEquals (X,Y) :- intEquals(X,Y).
intEquals([ ],[ ]).
intEquals([H|T],[X|Y]) :- H=X, intEquals(T,Y).

Recursion Hints

Many Prolog definitions require recursion (we also saw this *feature* a good bit in Erlang). Be careful when writing recursion in Prolog, however, as the search mechanism that Prolog employs can easily be led into an infinite loop. In particular, make sure you don't implement circular definitions:

parent(X,Y) :- child(Y,X).
child(A,B) :- parent(B,A).

If our query was to find **parent(X,Y)**, then the Prolog search through the database would generate the subgoal child**(Y,X)**. To satisfy this subgoal, the query **parent(X,Y)** would be generated, and the system would be stuck in a loop.

Another problem area is in using left recursion – having recursion as the first sub-goal of a conjunctive definition. As an example, the following statements will loop forever if a query for **person(X)** is issued:

//something is a person if their mother is a person
person(X) :- person(Y), mother(Y,X)
person(adam). // adam and eve don't have a mother, so they are the
person(eve). **//** base case

Even though there is a person defined ("adam"), this fact will never be encountered because in searching through the list of facts and rules, the system will always hit the rule **person(X) :- person(Y), mother(Y,X)** first and the system will always try to satisfy this new sub-goal for **person(Y)** first.

If the facts are reordered as shown below, the fact that *adam* is a person is returned, but the system still gets stuck in a loop and eventually crashes. Note that the problem here is that we try to satisfy *person(Y)* first, for an abstract variable Y, when satisfying *person(X)*, leading to an infinite loop.

person(adam).
person(X) :- person(Y), mother(Y,X).

This version

person(adam).
person(X) :- mother(Y,X), person(Y).

will work correctly, as it forces Y to be instantiated to some value (one of the mothers of X) before **person(Y)** is called. Once the system runs out of ways of discovering mothers, this 2nd definition of *person(X)* can no longer be satisfied.

Take Home Problems

The key examples shown in this lab can be found on the class webpage in the file **lab10Takehome.plg**.

1. Set operations – A set member function has already been written as one of the examples at the start of the lab. Write the remaining *set* functions:
    a. **insertSet**, with three parameters – an atom, a list representing a set, and the output list representing the set with the atom inserted. Remember, this is for sets, so the atom should be inserted only if it's not already in the set.
    b. **deleteSet**, with three parameters – an atom, a list representing a set, and the output list with the atom deleted if it was present in the set.

2. For general lists (not sets), write a function called **count**, which has three parameters - an atom, a list, and an output parameter which holds the number of times the atom passed as a parameter was found in the list passed as a parameter.

3. Write two functions, both of which can sum over a list of integers. The first function, called **listSum**, should have two parameters, the list and the output sum, and should implement the summing directly (similar to how listlength works) while the second, called **listSum2**, should also have two parameters (list and output sum), but should pass off the work to a three parameter function which takes the list, an accumulator sum, and the output sum and which uses the accumulator approach. Look at the examples listlength and listlength2 for inspiration. Assume summing over an empty list returns a value of 0.

4. Write a function called **stringLessThan** which takes two parameters (two strings) and which is satisfied if the first parameter is before the second parameter in the dictionary. There are three cases which should satisfy stringLessThan:
    a. If the first letter in the first word is "before" (in a dictionary ordering) the first letter in the second word. (bat < cat, because b < c)
    b. When the first word and second word have matched so far, but the letter in the current position in the first word is "before" the letter in the same position in the second word (car < cat, because r < t)
    c. When the first word and second word have matched so far, but the first word has run out of letters (car < cart)

5. Write a function called **strcat** which, when given a list of strings, generates the single string which is the concatenation of the original strings. You should use recursion in your solution. I have included in the lab10Takehome.plg document a printString method which will print to the screen as readable characters a string's underlying int list representation. To test your method, employ **strcat** and **printString** as follows:

    ?- strcat(["Hey","You","Guys"],A), printString(A).
    HeyYouGuys // the printed string, shown by printString
    A = [72, 101, 121, 89, 111, 117, 71, 117, 121|...].
    // the int list representation which is shown by the matching algorithm

6. Revise the grammar introduced in the lab so that the sentence, noun_phrase, and verb_phrase non-terminals are now represented as follows (implement these in place of their original decompositions):

    sentence → noun_phrase verb_phrase
    noun_phrase → determinant noun | determinant adjective noun
    verb_phrase → verb noun_phrase

    and such that the following additional rules are added:

    determinant → a | the
    adjective → smelly

    where *a, the*, and *smelly* are terminals that can't be decomposed further.

    Remember the | symbol means OR (one-or-the-other selection) in a grammar specification.

Due Date: Thursday, 4/16/2015 – midnight
To submit via Sakai:

- **An updated *lab10Takehome.plg* file which contains the facts and definitions required to answer the above questions.**
- **No documentation required this time**

Testing is left up to you – you don't have to turn in anything about your testing process, but you should try to thoroughly test each of your answers. Many of these problems we have encountered in the Erlang labs before, so you may find some good test cases in those labs.

Grading information is included below.

CSC 231 Lab 10 Gradesheet Name: _____

Task:	Available Points:	Earned Points:
Programming		
12.5 points each - insertSet, deleteSet functions	**25**	
12.5 points each - count function	**12.5**	
12.5 points each - listSum, listSum2 functions	**25**	
12.5 points each - stringLessThan function	**12.5**	
12.5 points each - strcat function	**12.5**	
12.5 points each - augmented sentence grammar	**12.5**	

of 100