Overview of Lab
The purpose of this lab is to allow you to gain experience in building more advanced list-based data structures and algorithms, to introduce an operator that can help with program efficiency, and to describe the basics of a Prolog tracing mechanism.

Debugging/Tracing
Debugging facilities are built into the Prolog interpreter and can be turned on and off with a small set of commands. The most exhaustive debugging can be turned on by typing `trace.` in the Prolog interpreter and turned off by typing `notrace.`. With trace on, you should see a `[trace]` prompt in the interpreter. In this tracing mode, every goal that is being worked on in answering a query is shown to you. There are four main operations:

- **CALL**: An initial attempt to satisfy a goal.
- **EXIT**: Successful satisfaction of a goal.
- **REDO**: An attempt to re-satisfy a goal.
- **FAIL**: A failed attempt, where satisfaction is not possible.

You should only see one CALL and one EXIT statement (unless you explicitly ask for re-satisfaction with the `;` (semicolon) command). On the other hand, you are likely to see many FAIL and REDO statements. You should hit `enter` to move the interpreter forward one step at a time. When your query has ended, the system will go into a separate debug mode, with a `[debug]` prompt. Ignore this mode for now – it allows you to enter and work with code setpoints (specific single lines you want to follow). To leave this mode and return to the normal Prolog interpreter, type `nodebug`.

Given the following facts:
- `father(george,mary).`
- `father(george,john).`
- `father(harry,sue).`
- `father(edward,george).`

here’s a trace of the query `father(X,george).`:

```prolog
?- trace.
Yes
[trace]  ?- father(X,george).
    Call: (7) father(_G283, george) ? creep
    Exit: (7) father(edward, george) ? creep
    X = edward ;
    Fail: (7) father(_G283, george) ? creep
No
```
Note that it satisfies the original query by matching *edward* against the variable *x*. After I type the semicolon to “request additional answers”, it returns fail, saying no more satisfactions are possible.

**List Based Data Structures and Algorithms**

Given you have a command over the various list operations from last week's lab, one way to expand on that knowledge is to implement list-based data structures and algorithms in Prolog.

Queues and Stacks are fairly easy to design. I will provide you with a full description of the Queue data structure below. The take home questions will ask that you develop a Stack data structure and write a program that takes advantage of the Stack.

The four queue operations of interest and their definitions are:

// returns true iff queue is empty

queueempty([]).

// usually called with variable for NewQueue, appends new element  
// onto back of current queue

queueadd(Queue, Item, NewQueue) :- append(Queue, [Item], NewQueue).

// usually called with variables for second and third  
// arguments FrontQueue and RestQueue  
// holds single element from front of queue in FrontQueue  
// remaining list (remaining queue) in RestQueue  
// [whenever you do a remove, you'll need to catch the  
// updated queue (RestQueue) in a temp variable and then use that]

queueremove([FrontQueue|RestQueue], FrontQueue, RestQueue).

// usually called with variable for second argument, puts  
// copy of front queue in the 2nd argument variable

queuepeek([FrontQueue|_], FrontQueue).

The four definitions above are enough to implement a standard Queue data structure. You can assume that remove and peek on empty queues will automatically fail (return 'No') since there are no rules that match those instances.
An example of the queue in use is as follows (note this is a query I typed in at the terminal):

?- queueAdd([],1,NewQueue),queueAdd(NewQueue,2,NewQueue2),queueRemove (NewQueue2,Item,NewQueue2AfterDeletion).

NewQueue = [1]
NewQueue2 = [1, 2]
Item = 1
NewQueue2AfterDeletion = [2];

Admittedly, this is a bit awkward to use as is. However, one would commonly bundle this inside of other function calls (such as `processComputerJobs(listOfJobs)`) which would then manage the queue for you. As you're aware, this programming style of catching the updated data structures in an output variable occurs frequently in Prolog.

As an example of a more advanced data structure using a Queue, I have implemented in the `lab10Takehome.plg` file a PriorityQueue – a data structure composed of an ordered list of regular Queues. You should consider the queues ordered from priority 0 to n, with 0 being the highest priority. Feel free to play around with the Queue and PriorityQueue operations to see how they work. As an example, you might want to try and see what happens when you make a call such as:

`pqueueInsert([[],[]],a,1,UpdatedPQ)`

This call inserts an a into the subqueue representing priority 1 items (remember they are ordered priority 0, 1, and 2 from left to right in the above picture, and they currently are all empty subqueues).

Besides being useful for data structures, lists can also be useful as their own data structure. A very useful list algorithm is to be able to sort a list. The simplest sorts you run across in introductory CSC courses are insertion sort and selection sort.

Taking a step back, here's my version of selection sort in ML (based on list recursion) - we'll see that a Prolog implementation is quite similar because we also only can do list recursion.
fun smallest(L:int list) =
    let
        val hvalue = hd(L);
        val tvalue = tl(L)
    in
        if (null(tvalue)) then hvalue
        else if (hvalue < hd(tvalue)) then smallest(hvalue::tl(tvalue))
        else smallest(tvalue)
    end;

fun remove(nil, x:int) = nil
| remove(z::zs, x:int) = if (x = z) then zs else z::remove(zs,x);

fun ssort(nil) = nil
| ssort((x:int)::xs) =
    let
        val z = smallest(x::xs)
    in
        z::(ssort(remove(x::xs,z)))
    end;

Now, here it is in Prolog (note its similarity to the ML implementation):

smallest([H],H).
smallest([H|T],H) :- smallest(T,S), H =< S.
smallest([H|T],S) :- smallest(T,S), H > S.

remove([],_,[]).
remove([H|T],H,T).
remove([H|T],S,Result) :- S <= H, remove(T,S,TempResult),
append([H],TempResult,Result).

selectionSort([],[]).
// note that the back end of this rule is one long statement
selectionSort([H|Tail],Result) :- smallest([H|Tail],RealSmallest),
remove([H|Tail],RealSmallest,UpdatedTail),
selectionSort(UpdatedTail,TempResult),append([RealSmallest],TempResult,Result).

One key idea to notice is how all of the nesting in the ML call has been linearly ordered in Prolog (smallest first, then remove, then selection sort, then append).
The Cut Operator
The cut operator is a special operator that can be used in Prolog to control how backtracking and goal satisfaction works in the language. The syntax for the cut operator is a single exclamation point, !, which can be used anywhere on the RHS of a Prolog rule.

The cut operator is always satisfiable (i.e. if a conjunction includes the cut operator, it will be taken as a “Yes” or “True” in the overall satisfaction).

This operator can play a role in improving the efficiency of Prolog programs. The technical definition of a cut is as follows:

When a cut is encountered as a goal, the system thereupon becomes committed to all choices made since the parent goal was invoked. All other alternatives are discarded. An attempt to re-satisfy any goal between the parent goal and the cut goal will fail. (Clocksin and Mellish, Programming in Prolog, 1994).

To gain an intuitive idea of how this cut operator works in Prolog, think of the satisfaction of a rule’s RHS that is made up of 2 or more subgoals. Visualise these subgoals as being represented by a slot machine. The system sweeps left to right, trying to find a combination of facts that will satisfy each of the subgoals. When the first subgoal is satisfied, the system moves to the next wheel in the slot machine. If the second subgoal fails, the first wheel (representing the first subgoal) is rotated a little more, to see if another satisfying set of variables for the first subgoal might allow the rest of the subgoals to be satisfied. What the cut operator does is to say: once you have passed over the cut operator, you can’t go back and spin the wheels in front of the cut operator any more. You’re committed to those entries.

In addition, the cut, if reached and satisfied, does not allow alternative definitions of the same high level goal be looked at – it commits you to just that particular definition as well.

This lab will introduce two uses of the cut operator.

Use 1 – Telling the Prolog system that it has found the right rule for a goal.

Assume you want to recursively implement a sum_to definition so that it acts as a summing function: sum_to(X,Y) should sum all numbers from 1 to X and store the result in Y. As an example, sum_to(1,Y) should set Y = 1, and sum_to(3,Y) should set Y = 6 (1+2+3).

A first try, naive definition would just list two different definitions for sum_to:

sum_to(1,1).
sum_to(X,Y) :- X1 is X-1, sum_to(X1,Y1), Y is X+Y1.
A query for `sum_to(1,X)` will return 1, and then if I hit ; to continue searching for answers, I will receive an out of stack error (the `sum_to` definition was attempted, and it got into a recursive loop).

A fix that works is as follows:

```
sum_to(1,1).
sum_to(X,Y) :- X > 1, X1 is X-1, sum_to(X1,Y1), Y is X+Y1.
```

(We discussed this fix idea in lecture yesterday; basically it is - use a constraint on the 2\textsuperscript{nd} rule that ensures that the 2\textsuperscript{nd} rule doesn’t match if the first rule has already matched).

This 2\textsuperscript{nd} definition requires that the $X>1$ test is executed because that constraint is the first part of the conjunction to be satisfied.

Here’s a trace of that process in action:

```
[trace]  ?- sum_to(1,X).
   Call: (7) sum_to(1, _G284) ? creep
   Exit: (7) sum_to(1, 1) ? creep
   X = 1 ; // me hitting a semicolon to request another answer
   Redo: (7) sum_to(1, _G284) ? creep
     ^ Call: (8) 1>1 ? creep
     ^ Fail: (8) 1>1 ? creep
     Fail: (7) sum_to(1, _G284) ? creep
   No
```

It’s at this point where we could take advantage of the cut operator. By changing the definition to the following:

```
sum_to(1,1) :- !.
sum_to(X,Y) :- X1 is X-1,sum_to(X1,Y1), Y is Y1 + X.
```

the 2\textsuperscript{nd} definition will never be looked at, even if a search for an additional answer is requested. The cut goal is reached and satisfied on the first rule and thus no other definition for the same function will be looked at. The trace is now as follows:

```
[trace]  ?- sum_to(1,X).
   Call: (7) sum_to(1, _G284) ? creep
   Exit: (7) sum_to(1, 1) ? creep
   X = 1 ;
   Fail: (7) sum_to(1, 1) ? creep
   No
```
While this may not seem like a huge improvement in this example, imagine if the first subgoal (the constraint) for alternative definitions was complex (requiring satisfying other subgoals through other definitions) or if there were multiple alternative definitions for the rule of interest. These could potentially require significant complexity in testing the constraint. However, the cut operator now lets us bypass the test.

In most of uses of the cut operator in this fashion, there is an equivalent implementation using the not predicate or a comparison (\(\neq\), >, <, etc) as the first subgoal. The tradeoff here comes down to efficiency (use of the !) versus readability (use of not or \(\neq\), >, <, etc).

A different example, with more subgoals on the RHS, is as follows: Assume, A, B, C, and D are all different clauses to be satisfied and that A is defined as follows.

\[
\begin{align*}
A & \iff B, C \\
A & \iff \text{not}(B), D.
\end{align*}
\]

This can be simplified for efficiency reasons as:

\[
\begin{align*}
A & \iff B, !, C \\
A & \iff D.
\end{align*}
\]

If B is ever satisfied, and then the cut is immediately satisfied afterwards, the first definition is forced to be the only one looked at. This is great, as it says “I know B is satisfiable, so the 2\textsuperscript{nd} rule won’t ever matter anyway and there’s no reason to try it and go through the work of trying to re-satisfy B in proving not(B).”

This implementation could also affect whether or not A gets satisfied at all however. Note that the placement of ! fixes our choice for B as well (re-read page 5 where this was stated). If satisfying C fails, no other choices for B will be looked at beyond the first match. You need to make sure this early stopping of looking at B is what you really mean for it to do.

**Use 2** – Telling the Prolog system to fail immediately a particular goal and not search for alternative solutions.

First, another new predicate needs to be introduced: the fail predicate. The fail predicate immediately fails, and causes the specific definition that is being checked to also fail (as the RHS conjunctions that include fail can’t possibly now all be found to be true).

Given the fail predicate, the following example will demonstrate the use of cut and fail together to tell Prolog to stop trying to satisfy a goal.
Imagine a system where we want to define an average taxpayer. We might have some constraints on the amount of income the person has, whether or not they are married, and so on that define what it means to be an average taxpayer. A rule that might be implemented for this is:

\[
\text{average_taxpayer}(X) :- \text{income}(X,Y), Y < 50000, \text{married}(X).
\]

Now imagine that we need to exclude foreign citizens that work in the U.S. as they are required to handle their taxes in a different way to satisfy both governments. We would need to extend our definition as follows:

\[
\text{average_taxpayer}(X) :- \text{not(foreigner}(X)), \text{income}(X,Y), Y < 50000, \text{married}(X).
\]

where foreigner\(X\) is appropriately defined.

Let's consider an alternative implementation:

\[
\text{average_taxpayer}(X) :- \text{foreigner}(X), \text{fail}.
\]
\[
\text{average_taxpayer}(X) :- \text{income}(X,Y), Y < 50000, \text{married}(X).
\]

The first definition will say that if the person represented by the variable \(X\) is a foreigner, then average_taxpayer fails (which is correct). Prolog, using these rules as they are written, however, will try to backtrack and see if there is another way of determining if the person represented by \(X\) is an average taxpayer. If the person fits the income and marriage requirements, then the average_taxpayer rule will be satisfied. So, this implementation doesn’t quite preserve our old definition.

To fix this, the cut operator needs to be put in. An implementation that actually matches the original definition in meaning is:

\[
\text{average_taxpayer}(X) :- \text{foreigner}(X), !, \text{fail}.
\]
\[
\text{average_taxpayer}(X) :- \text{income}(X,Y), Y < 50000, \text{married}(X).
\]

If the cut operator is satisfied (by having \text{foreigner}(X) satisfied), it forces only the first average_taxpayer definition to be looked at (backtracking will never look at the other taxpayer definition rules) and the fail indicates that a No answer for average_taxpayer\(X\) will be returned for that particular person.

Similar to case 1 of the use of the cut operator, this example demonstrates that it is possible to interchange the use of a cut operator with that of a \text{not} predicate or another comparison based predicate.
Take Home Problems:

1. Implement a Stack data-structure using a list as the base data-structure so that it works as follows. You should have implementations for:

   `stackempty`, taking one argument, a stack, and returning true if the stack is empty.

   `stackpush`, taking three arguments in this order: the current stack, a variable holding the item to push on top of the stack, and the resulting stack after the push.

   `stackpop`, taking three arguments in this order: the current stack, a result variable which will hold the item popped from the top of the stack, and the resulting stack after the pop.

   `stackpeek`, taking two arguments in this order, the current stack and a result variable which will hold the item sitting on top of the stack (but the item isn’t popped off, so the stack doesn’t change).

2. Using the stack data-structure you completed in part 1, write a function called `postfix` which can compute the expression value for a postfix expression. `postfix` will take two parameters – a list representing a postfix expression and a result variable for holding the returned expression value. Your input expression will be in the form of a list and should not be considered as a Stack itself, just a list. Your evaluation of the expression should make use of a Stack to hold intermediate data in the evaluation of the expression. Postfix should only interact with the Stack being used for evaluation through the Stack operations written above, not through direct manipulation of the lists underlying the Stack. Calls to `postfix` will appear as follows:

   
   ?- postfix([2,3,'+'],Z).
   Z = 5 ;
   No
   |- postfix([2,3,'+',2,'*'],Z).
   Z = 10 ;
   No
   |- postfix([2,3,'+',4,6,'-', '*'],Z).
   Z = -10 ;
   No

   Your code should be able to handle, the +, -, *, and / operators. The operators will be written in single quotes as characters (which you can, for example, test against with statements like `X = '+' or `X \= '+'), while the numbers will be written plain (without single quotes). You can assume that you will get a valid postfix expression (it can’t start with a ‘+’ for example) with at least one element in the expression (so the smallest
expression is a numerical value by itself). Keep your operands in left to right order – for example, if you get \([a, b, '-']\), do \(a-b\), not \(b-a\).

If you are unfamiliar with postfix notation or the use of a stack in evaluating postfix expression, please stop by my office and I’ll catch you up.

3. Implement *ascending* insertion sort (lowest at the front, highest at the back) using Prolog. A function called `insertionSort` should be written that takes two arguments – an input list to be sorted, and a variable to hold the resulting sorted output list. Write `insertionSort` recursively using the following approach:
   a. First call `insertionSort` recursively on the input list without the head.
   b. `insertionSort` of an empty list is just the empty list
   c. As the recursive functions complete and are returning values, insert the element that was pulled off before the recursive call at the right spot in the array (this requires an auxiliary function to drop the item into the right spot).

Here’s an example of my `insertionSort` running:

```prolog
?- insertionSort([2,4,1,3,5],SortedList).
SortedList = [1, 2, 3, 4, 5] ;
No
```

4. Copy and paste and then edit the `insertSet` and `count` functions found in `lab10Takehome.plg` so that they make use of the cut (!) operator to prevent an alternative definition from being tried when the user hits ‘;’ to ask for another answer answer and to optimize out unnecessary ‘filters’ (any checks, such as less-thans, used to verify the inputs are valid for the function). Rename your copied/pasted/revised functions as `insertSet2` and `count2`. If you use recursion, make sure your recurse on `insertSet2` and `count2` instead of the original functions.

Use the ‘trace.’ command on the original functions and your updated functions to verify the improvement. Copy and paste the results of the trace into a file called `lab10Answers.txt` and submit this along with `lab10Takehome.plg` (You should have 4 entries in `lab10Answers.txt` – both functions traced in the regular mode (insertSet and count) and in the ‘improved with cut’ mode (insertSet2 and count2)).
Due Date: Thursday, April 1st, 9:00pm (note the Thursday due date, so finishing the lab doesn’t interfere with the test).

Submit on Sakai:
- An updated lab10Takehome.plg file which contains the facts and definitions required to answer the above take-home questions.
- A file called lab10Answers.txt which holds your trace results from problem 4.
- A file called lab10Description.txt which holds a brief explanation and justification of your approach for implementing a solution to each question.