Important notes about grading:

Late assignments: Please carefully review the course website's policy on late assignments, as all assignments handed in after the deadline will be considered late. Submitting the incorrect version before the deadline and realizing that you have done so after the deadline will be counted as a late submission.

How to Start:

1. Download the source code here.
2. You will need to implement your code in `assignment4/assignment4.pl`. `assignment4/assignment4.plt` is the test file. And `assignment4/plunit.pl` is for unit testing. We give detailed instructions on the functions that you need to implement below.
3. To run your implementation, you have two options:
   Option (1): Download and install the SWI-Prolog interpreter here. After the interpreter GUI is openned, you can use it to launch your program and make queries. In the installed GUI, type `consult("/path/to/your/assignment4/assignment4.pl").` after the `?-` query symbol to load the soure file. Please specify the path correctly to where you saved `assignment4/assignment4.pl`. You will see a number of warnings. Please ignore them. Type "range(1,2,0)." (see Problem 1) after the `?-` query symbol to make your first query. The result should be `false`. The installation process is quite simple but we cannot offer too much help remotely if you do encouter problems. So we recommend you to use `iLab`.

   Option (2): Use `iLab`. Please see the document here.
4. Unit testing:
   (1) In iLab, suppose you are in the folder that contains `assignment4.pl`, `assignment4.plt` and `plunit.pl`.

   (2) swipl -t "use_module(plunit), load_test_files([]), run_tests." -s assignment4

   (3) You will then be able to see how many test-cases you passed.

Some "strange" things about SWI-Prolog you need to know:

I cannot quit the SWI-Prolog interpreter.

Type "halt." after the `?-` query symbol to quite the prolog interpreter.

I get a number of singleton variables warnings.

Warning: /../../../assignment4.pl:14: Singleton variables: [X]

This is because you introduced some variables in your program that are ever unused which could be replaced with `_`. You can ignore this type of warnings.

I get one solution when I expected multiple solution.

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

X = 1

You expect to see X = 2 and X = 3 but it seems the interperter just gets stuck there.

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

X = 1; <-- waits for me to type the ;

X = 2; <-- waits for me to type the ;

X = 3

Put a semicolon there to resuem the search.

I get true, then false when I expected just true.

Sometimes after the last solution, when you expect prolog to just stop, it instead waits for you to type ; one more time. So something like

?- member(1, [1,2,3]).

true ; <-- waits for me to type the ;

false.

This is just because Prolog's not infinitely smart about knowing if it's got another solution. A story to illustrate:

A woman walks into a hardware store and says to the owner "I'm driving in fence posts, I need a really big hammer" The owner shows her a hammer. She says "No, I need a bigger hammer than that". The owner says "hold on, I have a bigger one in back".

He comes back a couple minutes later and says "Sorry, must have sold it. That's the biggest we have." The first solution is offered (the first hammer). She rejects it, and the storeowner tries again. This time he fails. He thought he might have another solution, but he doesn't.

Students often try to eliminate these. They have no effect on execution.

Built-in Predicates

Some of the predicates we learn in class are actually built-in. You are free to use them in your assignment. Some examples are `append` and `length`. Arithmetic predicates include `+`, `*`, `/`, `<`, `=<`, `>`, `>=`, etc. The predicate `is` should be used for equivalence checking, e.g., `A is B`. Negation is like `not(A is B)`.

Warmup

Problem 1

Implement a predicate `range(S,E,M)` which holds if the integer `M` is within the range `S` to `E` including `S` and `E`.

In [1]:

```
/* YOUR CODE HERE (delete the following line) */
range(S,E,M) :- false.
```

```
Added 1 clauses(s).
```

Test-cases

In [2]:

```
?- range(1,2,2).
/* expected output: true. */

?- not(range(1,2,3)).
/* expected output: true. */
```

```
true.
true.
```

Programming with Lists

Let's implement some programs over lists.

Problem 2

Implement reverse of a list using the predicate `reverseL(X,RevX)` where `RevX` is the reverse of the list `X`. You might want to use `append`.

In [3]:

```
/* YOUR CODE HERE (delete the following line) */
```

```
/* YOUR CODE HERE (delete the following line) */
reverseL(X,RevX) :- false.
```

```
Added 2 clauses(s).
```

Test-cases

In [4]:

```
?- reverseL([],X).
/* expected output: X = [  ]. */

?- reverseL([1,2,3],X).
/* expected output: X = [ 3, 2, 1 ]. */

?- reverseL([a,b,c],X).
/* expected output: X = [ c, b, a ]. */
```

```
X = [  ] .
X = [ 3, 2, 1 ] .
X = [ c, b, a ] .
```

Problem 3

Implement list membership predicate `memberL(X,L)` which holds if $X \in L$. Addtionally, if $L = \emptyset$, return `false`.

In [5]:

```
/* YOUR CODE HERE (delete the following line) */
memberL(X,L) :- false.
```

```
Added 3 clauses(s).
```

Test-cases

In [6]:

```
?- not(memberL(1, [])).
/* expected output: true. */

?- memberL(1,[1,2,3]).
/* expected output: true. */

?- not(memberL(4,[1,2,3])).
/* expected output: true. */

?- memberL(X, [1,2,3]).
/* expected output: see below */
```

```
true.
true.
true.
X = 1 ;
X = 2 ;
X = 3 .
```

Problem 4

Implement the predicate `zip(Xs, Ys, XYs)` that relates three lists, where the third list `XYs` contains pairs whose elements are the elements at the same indices of the first two lists `Xs` and `Ys`. We will use `A-B` to represent a pair of element `A` and `B`. For example, `1-2` is like a pair `(1,2)`

The third list `XYs` will always be as long as the shorter of the first two lists; additional elements in the longer list are discarded.

In [7]:

```
/* YOUR CODE HERE (delete the following line) */
zip(Xs, Ys, XYs) :- false.
```

```
Added 3 clauses(s).
```

Test-cases:

In [8]:

```
?- zip([1,2],[a,b],Z).
/* expected output: Z = [1-a, 2-b]. */

?- zip([a,b,c,d], [1,X,y], Z).
/* expected output:: Z = [a-1, b-X, c-y]. */

?- zip([a,b,c],[1,X,y,z], Z).
/* expected output:: Z = [a-1, b-X, c-y]. */

?- length(A,2), length(B,2), zip(A, B, [1-a, 2-b]).
/* expected output:: A = [1,2], B = [a,b]. */
```

```
Z = [ 1-a, 2-b ] .
X = _1908, Z = [ a-1, b-_1908, c-y ] .
X = _1926, Z = [ a-1, b-_1926, c-y ] .
A = [ 1, 2 ], B = [ a, b ] .
```

Problem 5

Implement the predicate `insert(X, Ys, Zs)` that is a relation between an integer `X`, a sorted list of integers `Ys`, and a second sorted list of integers `Zs` that contains the first argument `X` and all the elements of the second argument `Ys`.

In [9]:

```
/* YOUR CODE HERE (delete the following line) */
insert(X, Ys, Zs) :- false.
```

```
Added 3 clauses(s).
```

Test-cases:

In [10]:

```
?- insert(3, [2,4,5], L).
/* expected output: L = [2,3,4,5]. */

?- insert(3, [1,2,3], L).
/* expected output: L = [1,2,3,3]. */

?- not(insert(3, [1,2,4], [1,2,3])).
/* expected output: true. */

?- insert(3, L, [2,3,4,5]).
/* expected output: L = [2,4,5]. */

?- insert(9, L, [1,3,6,9]).
/* expected output: L = [1,3,6]. */

?- insert(3, L, [1,3,3,5]).
/* expected output: L = [1,3,5]. */
```

```
L = [ 2, 3, 4, 5 ] .
L = [ 1, 2, 3, 3 ] .
true.
L = [ 2, 4, 5 ] .
L = [ 1, 3, 6 ] .
```

```
L = [ 1, 3, 5 ] .
```

Problem 6

Write a Prolog predicate `remove_duplicates(L1,L2)` that is true if `L2` is equal to the result of removing all duplicate elements from `L1`. In the result, the order of the elements must be the same as the order in which the (first occurences of the) elements appear in `L1`.

In [11]:

```
/* YOUR CODE HERE (delete the following line) */
remove_duplicates(L1,L2) :- false.
```

```
Added 4 clauses(s).
```

Test-cases:

In [12]:

```
?- remove_duplicates([1,2,3,4,2,3],X).
/* expected output: X = [1, 2, 3, 4] */

?- remove_duplicates([1,4,5,4,2,7,5,1,3],X).
/* expected output: X = [1, 4, 5, 2, 7, 3] */

?- remove_duplicates([], X).
/* expected output: x = [] */
```

```
X = [ 1, 2, 3, 4 ] .
X = [ 1, 4, 5, 2, 7, 3 ] .
X = [  ] .
```

Problem 7

Write a Prolog predicate `intersectionL(L1,L2,L3)` that is true if `L3` is equal to the list containing intersection of the elements in `L1` and `L2` without any duplicates. In other words, `L3` should contain the elements that both in `L1` and in `L2`. As for union the predicate must be true for some order of elements of the intersection (but not necessarily all).

In [13]:

```
/* YOUR CODE HERE (delete the following line) */
intersectionL(L1,L2,L3) :- false.
```

```
Added 4 clauses(s).
```

Test-cases:

In [14]:

```
?- intersectionL([1,2,3,4],[1,3,5,6],[1,3]).
/* expected output: true. */

?- intersectionL([1,2,3,4],[1,3,5,6],X).
/* expected output: X = [1,3]. */

?- intersectionL([1,2,3],[4,3],[3]).
/* expected output: true. */
```

```
true.
X = [ 1, 3 ] .
true.
```

Problem 8

Implement `partition(L,P,S)` such that

- `P` is the prefix of `L` and
- `S` is the suffix of `L` and
- `append(P,S,L)` holds
- If `L` is `[]`, then `P` and `S` are `[]`.
- If `L` is `[H]`, then `P` is `[H]` and `S` is `[]`.
- Otherwise,
  - let length of `L` be `N`. Then length of `P` is `div(N,2)`. Use Prolog's built-in integer division.
  - length of `S` is `N - div(N,2)`.

You may need to use the `length`, `prefix`, `suffix`, `append` predicates that we have seen in class.

In [15]:

```
prefix(P,L) :- append(P,X,L).
suffix(S,L) :- append(X,S,L).

/* YOUR CODE HERE (delete the following line) */
partition(L,P,S) :- false.
```

```
Added 5 clauses(s).
```

Test-cases

In [16]:

```
?- partition([a],[a],[]).
/* expected output: true. */

?- partition([1,2,3],[1],[2,3]).
/* expected output: true. */

?- partition([a,b,c,d],X,Y).
/* expected output: Y = [ c, d ], X = [ a, b ]. */
```

```
true.
true.
Y = [ c, d ], X = [ a, b ] .
```

Problem 9

Implement the predicate `merge(X,Y,Z)` where `X` and `Y` are sorted, and `Z` contains the same elements as `U` where `append(X,Y,U)` but `Z` is also additionally sorted.

In [17]:

```
/* YOUR CODE HERE (delete the following line) */
merge(X,Y,Z) :- false.
```

```
Added 4 clauses(s).
```

Test-cases

In [18]:

```
?- merge([],[1],[1]).
/* expected output: true. */

?- merge([1],[],[1]).
/* expected output: true. */
```

```
?- merge([1,3,5],[2,4,6],X).
/* expected output: X = [ 1, 2, 3, 4, 5, 6 ]. */
```

```
true.
true.
X = [ 1, 2, 3, 4, 5, 6 ] .
```

Problem 10

Implement predicate `mergesort(L,SL)` where `SL` is the sorted version of `L`. Use the predicate to partition the list `L` into two, sort each on separately (using `mergesort`) and combine the individual sorted list using `merge`.

In [19]:

```
/* YOUR CODE HERE (delete the following line) */
mergesort(L,SL) :- false.
```

```
Added 3 clauses(s).
```

Test-cases

In [21]:

```
?- mergesort([3,2,1],X).
/* expected output: X = [ 1, 2, 3 ]. */

?- mergesort([1,2,3],Y).
/* expected output: Y = [ 1, 2, 3 ]. */

?- mergesort([],Z).
/* expected output: Z = [  ]. */

?- mergesort([1,3,5,2,4,6],X).
/* expected output: X = [ 1, 2, 3, 4, 5, 6 ]. */
```

```
X = [ 1, 2, 3 ] .
Y = [ 1, 2, 3 ] .
Z = [  ] .
X = [ 1, 2, 3, 4, 5, 6 ] .
```