---

1. Write a predicate `weird_sum(List, Result)` which takes a `List` of numbers, and computes the sum of the squares of the numbers in the list that are greater than or equal to 5, minus the sum of the absolute values of the numbers that are less than or equal to 2. For example:

   ```
   ?- weird_sum([3,6,2,-1], Result).

   Result = 33
   ```

   That's 6×6 - 2 - 1.

   Note:
   (a) it is the *value* of the item, not its position in the list, that should be tested to see if it is greater than or equal to 5, or less than or equal to 2.
   (b) you are not required to check whether the items in the list are numbers.

   Think carefully about how the predicate should behave on the *empty* list — should it fail or is there a reasonable value that `Result` can be bound to?

---

2. Suppose that a set of family relationships have been loaded into Prolog using the same format as `family.pl`

   ```
   parent(jim, brian).
   parent(brian, jenny).
   parent(pat, brian).
   female(pat).
   female(jenny).
   male(jim).
   male(brian).
   ```

   NOTE: do not include these in your solution file.

   We assume that each person will have the same family name as their father, but that married women retain their original birth name.

   Write a predicate `same_name(Person1,Person2)` that succeeds if it can be deduced from the facts in the database that `Person1` and `Person2` will have the same family name. (It is ok if your code returns `true` multiple times). For example:

   ```
   ?- same_name(pat, brian).
   false.

   ?- same_name(jenny, jim).
   true
   ```

   Note that your `same_name` predicate will be tested with different facts to those in `family.pl`

---

3. Write a predicate `log_table(NumberList, ResultList)` that binds `ResultList` to the list of pairs consisting of a number and its log, for each number in `NumberList`. For example:

   ```
   ?- log_table([1,3.7,5], Result).
   Result = [[1, 0.0], [3.7, 1.308332819650179], [5, 1.6094379124341003]].
   ```

   Note that the Prolog built-in function `log` computes the natural logarithm, and that it needs to be evaluated using `is` to actually compute the log:

```
?- X is log(3.7).
X = 1.308332819650179.

?- X = log(3.7).
X = log(3.7).
```

COMP9814 only: write a predicate `function_table(+N, +M, +Function, -Result)` that binds `Result` to the list of pairs consisting of a number X and Function(X), from `N` down to `M`. For example:

```
?- function_table(7, 4, log, Result).

Result = [[7, 1.94591], [6, 1.79176], [5, 1.60944], [4, 1.38629]] ;

false.
```

`log` computes the natural logarithm of it's argument.

Hint: look up `univ` or `=..` in the Prolog Dictionary.

---

4. Any list of integers can (uniquely) be broken into "parity runs" where each run is a (maximal) sequence of consecutive even or odd numbers within the original list. For example, the list

   ```
   List = [8,0,4,3,7,2,-1,9,9]
   ```

   can be broken into `[8, 0, 4]`, `[3, 7]`, `[2]` and `[-1, 9, 9]`

   Write a predicate `paruns(List, RunList)` that converts a list of numbers into the corresponding list of parity runs. For example:

   ```
   ?- paruns([8,0,4,3,7,2,-1,9,9], RunList).
   RunList = [[8, 0, 4], [3, 7], [2], [-1, 9, 9]]
   ```

   Note: you can find out how to test if a number is even or odd from the Prolog Dictionary

---

5. In this question we consider binary trees which are represented as either `empty` or `tree(L, Num, R)`, where `L` and `R` are the left and right subtrees and `Num` is a number.

   A binary tree of numbers is called a *heap* (or, it is said to satisfy the *heap property*) if, for every non-leaf node in the tree, the number stored at that node is less than or equal to the number stored at each of its children. For example, the following tree satisfies the heap property, because $3 \leq 5$, $5 \leq 8$ and $5 \leq 7$.

   ```
   tree(empty,3,tree(tree(empty,8,empty),5,tree(empty,7,empty)))
   ```

   On the other hand, the following tree does not satisfy the heap property, because 6 is not less than or equal to 5.

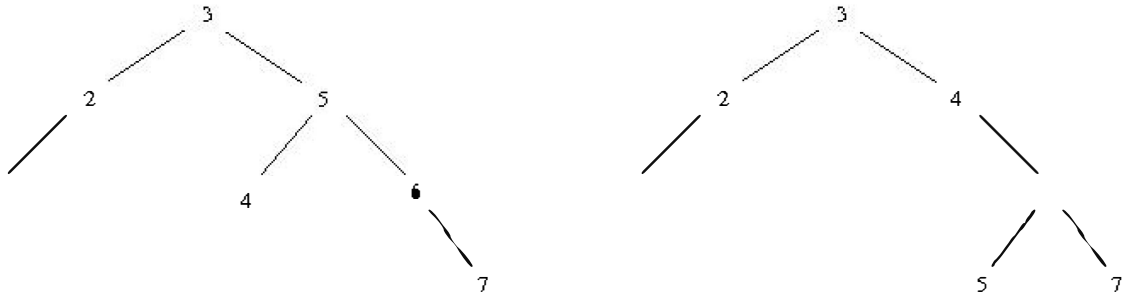
   ```
   tree(tree(tree(empty,4,empty),
          3,tree(empty,5,empty)),6,tree(tree(empty,9,empty),7,empty))
   ```

   Write a predicate `is_heap(Tree)` which returns `true` if `Tree` satisfies the heap property, and `false` otherwise. For example:

   ```
   ?- is_heap(tree(tree(tree(empty,4,empty),
            3,tree(empty,5,empty)),6,tree(tree(empty,9,empty),7,empty))).
   false.

   ?- is_heap(tree(empty,3,tree(tree(empty,8,empty),5,tree(empty,7,empty)))).
   true
   ```

COMP9814 only: The *height* of a binary tree is defined to be the number of nodes in the longest path from the root to a leaf. A binary tree is called *balanced* if, for every node in the tree, the height of its left and right subtree differ by no more than 1. For example, the tree on the left is balanced, with height 4, but the tree on the right is not balanced, because the left and right subtree of node 4 have heights 0 and 2, respectively.



COMP9814 only: Write a predicate `height_if_balanced(Tree, HiB)`, which takes a binary tree `Tree` and binds its second argument `HiB` to the height of the Tree, if it is balanced, or -1 if it is *not* balanced.

Trees are represented as either `empty` or `tree(L, Data, R)`, where `L` and `R` are the left and right subtrees. The `Data` in each node of the tree is irrelevant to this programming exercise.

You are free to copy the predicate `max(First, Second, Max)` from the lecture notes and use it in your program.

`Tree` must be instantiated at the time of the call. These examples use the trees shown above:

```
?- height_if_balanced(tree(tree(tree(empty,1,empty),2,empty),
                        3, tree(tree(empty,4,empty),
                             5, tree(empty,6,tree(empty,7,empty)))),HiB).
HiB = 4 ;
false.

?- height_if_balanced(tree(tree(tree(empty,1,empty),2,empty),
                         3, tree(empty,4,tree(tree(empty,5,empty),
                                              6, tree(empty,7,empty)))),HiB).
HiB = -1
false.
```

Testing

Your assignment will be tested by an automated testing system, and also read by a human marker. Marks will be allocated for test results, and for layout, comments, and comprehensibility.

Your code must work under the version of SWI Prolog used on the Linux machines in the UNSW School of Computer Science and Engineering. If you develop your code on any other platform, it is your responsibility to re-test and if necessary correct your code when you transfer it to a CSE Linux machine prior to submission.

Submitting your assignment