CPS721: Assignment 2

Due: October 11, 2022, 9pm
Total Marks: 100 (worth 4% of course mark)
You MUST work in groups of 2 or 3

Late Policy: The penalty for submitting even one minute late is 10%. Assignments are not accepted more than 24 hours late.

Clarifications and Questions: Please use the discussion forum on the D2L site to ask questions as they come up. These will be monitored regularly. Clarifications will be made there as needed. A Frequently Asked Questions Page will also be created. You may also email your questions to your instructor, but please check the D2L forum and frequently asked questions first.

Collaboration Policy: You can only discuss this assignment with your group partners or with your CPS721 instructor. By submitting this assignment, you acknowledge that you have read and understood the course policy on collaboration as stated in the CPS721 course management form.

PROLOG Instructions: When you write your rules in PROLOG, you are not allowed to use ";" (disjunction), "!" (cut), and "->" (if-then). You are only allowed to use ";" to get additional responses when interacting with PROLOG from the command line. Note that this is equivalent to using the "More" button in the ECLiPSe GUI.

We will use ECLiPSE Prolog release 6 to mark the assignments. If you run any other version of PROLOG, it is your responsibility to check that it also runs in ECLiPSE Prolog release 6.

Submission Instructions: You should submit a zip file that contains 4 files: `question1_list_equality.pdf`, `question2_list_manipulation.pl`, `question3_highest_cost_path.pl`, and `question4_merge_sort.pl`. Your submission should not include any other files. If you submit a `.rar`, `.tar`, `.7zip`, or other compression format aside from `.zip`, you will lose marks. The name of the zip file should be `yourLoginName.zip` where `yourLoginName` is only the group member who made the submission. Note that only one student in the group should make submissions, as D2L can present confusing information if more than one student in the group submits.

All submissions should be made on D2L. Submissions by email will not be accepted.

You are allowed to make as many submissions as you want, and you do not have to inform anyone about this. The new copy will override the old one. The time stamp of the last submission will be used to determine the submission time. However, the same group member should make all the submissions for the reasons explained above.

If you write your code on a Windows machine, make sure the files are saved on plain text and are readable on Linux machines. Ensure your PROLOG code does not contain any extra binary symbols and that they can be compiled by ECLiPSE Prolog release 6.

1 List Equality [24 marks]

For each of the following pairs of lists, state which can be made identical and which cannot. You must also provide a short proof as to why. This means you should convert the lists to the same style (*ie.* the '`|`' based representation or the standard ',' based representation), and use that to explain how they can be made identical or not. For example, if given the pairs `[X,Y]` and `[a|[b]]`, you could say that these match with `X=a` and `Y=b` since the second list can be written as `[a,b]` (or equivalently the first list can be written as `[X|[Y]]`). Any answers that do not contain detailed explanations for why the lists do or do not match will lose marks.

You should submit your answers in a pdf file called `question1_list_equality.pdf`. The names, emails, and student IDs of your group members should appear at the top of this PDF file. Note that each part below is worth 3 marks.

a. `[X, Y | Z]` **and** `[p, q, r, s, t | [u, v, Y] ]`

b. `[a, [Y | [b,  c] ],  d ]` **and** `[a, [b, [b, c] ] | Z]`

c. `[yyz  |  [yow  |  [ yyc |  [yvr |  [ yul | [YEG] ] ] ] ] ]` **and** `[A1, A2, | A3]`.

d. `[apple, Z, bee | [Z, car, door] ]` **and** `[X | [bee, Y | [Q | R ] ] ]`

e. `[Z | [ Z | [ [ Z | [ [Z] ] ] ] ] ]` **and** `[b | Y]`

f. `[U | [W |  [U] ] ]` **and** `[the, quick, brown, fox, W]`

g. `[first | [U | [ [R] | U]]]` **and** `[Q,  [] ,  [Q]|  U]`

h. `[Did | [ [An,  X] | [ever, Win,  An, X] ] ]` **and**
`[Only,  [One, oscar] | [Did,  X, hammerstein, TheSecond] ]`

2 Recursive List Manipulation in Prolog [46 marks]

For this question, you are to implement several programs that manipulate lists in a variety of ways. You should add your programs to the file `question2_list_manipulation.pl` in the appropriate areas. If you do not properly put the programs in the correct place or do not include the correct information about your group members, you may lose marks.

a. [**5 marks**] Write a program that uses two equal length lists to implement a map, where the first list is the keys and the second list are the values. The *i*-th key should map to the *i*-th value. The predicate definition should be as follows:

`listMap(KeysList, ValuesMap, Key,Value)` - where `KeysList` is the list of keys, `ValuesMap` is the list of values, `Key` is an input argument that acts as a key, and `Value` is an output argument that returns the value corresponding for that key.

Below you can find examples of queries using this predicate.

```
The following should succeed
?- listMap([a, b, c, d], [1, 2, 3, 4], c, 3).
?- listMap([hello,yes], [goodbye, no], yes, X).
    Yes with X = no

The following should fail
?- listMap([a, b, c, d], [1, 2, 3, 4], c, 4).
?- listMap([hello,yes], [goodbye, no],  a, X).
```

b. [**5 marks**] Write a program that encodes a given list, by replacing the items in the list with corresponding code values. The codes values are stored using a list map. Thus, you will need to use your program from part **a**. The predicate definition should be as follows:

`encodeList(List, MapKeysList, MapValuesLIst, EncodedList)` - where `List` is the list to encode, `MapKeysList` is the list of keys for the map, `MapValuesList` is the list of values for the map, and `EncodedList` is the encoded version of the list.

Below you can find examples of queries using this predicate.

```
The following should succeed
?- encodeList([a, a, c, b], [a, b, c, d], [1, 2, 3, 4], [1, 1, 3, 2]).
?- encodeList([c,a,g,e], [a, b, c, d,e, f, g], [g,f, e, d,c, b, a],  X).
    Yes with X = [e, g, a, b]

The following should fail
?- encodeList([a, a, c, b], [a, b, c, d], [1, 2, 3, 4], [1, 2, 3, 2]).
?- encodeList([r,a,g,e], [a, b, c, d,e, f, g], [g,f, e, d,c, b, a], X).
```

c. [**12 marks**] Write a program that counts the number of times that consecutive elements in the list are equal. The predicate definition should be as follows:

`consecutiveCount(List, Count)` - where `List` is the given list and `Count` is the number of times that consecutive elements are equal.

Below you can find examples of queries using this predicate.

```
The following should succeed
?- consecutiveCount([a, a, c, b, b, e, g], 2).
?- consecutiveCount([a,  b, c, d, e, f, g], 0).
?- consecutiveCount([a, a, c, big, big, big, ear, gold, gold], X).
    Yes with X = 4

The following should fail
?- consecutiveCount([a, a, c, b, b, e, g], 1).
?- consecutiveCount([a, a, c, big, big, big, ear, gold, gold], 2).
```

d. [12 marks] Write a program that takes in a list of positive integers and a single positive integer, and generates two new lists. The first consists of values no larger than the given integer, and the second consists of values that are larger than the given integer. The predicate definition should be as follows:

`splitOnInt(List, Value, NoLargerList, NoSmallerList)` - where `List` is the given list, `Value` is the integer to split on, `NoLargerList` is the list of values in `List` that are no larger than `Value`, and `NoSmallerList` is the list of values in `List` larger than `Value`.

Below you can find examples of queries using this predicate.

```
The following should succeed
?- splitOnIntt([45, 67, 23, 100, 0, 1578], 99, X, Y).
    Yes with X = [45, 67,23, 0] and Y =[100, 1578]
?- splitOnInt([24, 67, 45], 67, [24, 67, 45], []).

The following should fail
?- splitOnInt([45, 67, 23, 100, 0, 1578], 99, [45, 67,0], [100, 1578, 23]).
?- splitOnInt([45, 67, 23, 100, 0, 1578], 99, [45, 67,0], X).
```

e. [12 marks] Suppose you wanted to keep a running average of some incoming data, like the number of daily subway riders that use a particular stop or the hourly number of Google searches made worldwide. The standard way to calculate that average would be to keep a running sum and the number of data points so far, and dividing that sum to get the current average.

However, this way of calculating the average is susceptible to overflow when there are a lot of items to take the average over. An alternative way to calculate the average is to do so incrementally. In this approach, we store the current average and number of data points. Where d is the number of data points so far, A_d is the average computed after d data points (A_0 is set to 0), and e_d is the d-the data point, this incremental calculation [1] is done as follows:

$$A_{d+1} = A_d + \frac{e_d - A_d}{d+1}$$

For this question, you should implement this incremental averaging approach on a given list. The predicate definition should be as follows:

`incAverage(List, Average, NumElements)` - where `List` is the given list, `Average` is average, and `NumElements` is the number of elements in the list.

Below you can find examples of queries using this predicate.

[1] This method is also the foundation for updates done by Reinforcement Learning agents which you may learn about in other courses

```
The following should succeed
?- incAverage([1, 2, 3, 4, 5],Num,Av).
    Yes with Num = 5 and Av = 3.

The following should fail
?- incAverage([1, 2, 3, 4, 5],5,4).
?- incAverage([1, 2, 3, 4, 5],2,4).
```

3 Longest Path [15 marks]

A ternary tree is a tree where each node has at most 3 children (much like how a binary tree has at most two children at every node). To represent a ternary tree, we will use the following term

```
tree3(Name,  LeftCost, Left,  MiddleCost, Middle, RightCost, Right)
```

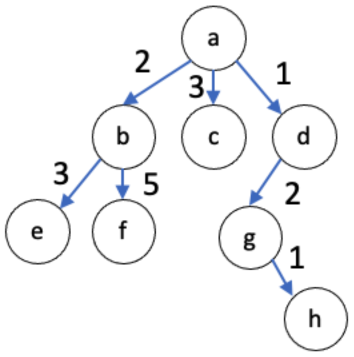
The arguments are defined as follows:

- `Name` is the name of the node
- `Left`, `Middle`, and `Right` represent the three branches of the tree. The values of each these will be terms: either a `ltree3` term or `none` indicating there is no branch below
- `LeftCost`, `MiddleCost`, and `RightCost` are the cost of the edges to each of the corresponding child nodes

Create a program that calculates the highest cost path from the root and returns a list of the names along that longest path. The predicate definition should be as follows:

`highestCostPath(Tree, PathCost, PathList)` - where `Tree` is the given tree, `PathCost` is the cost of the highest cost path that starts from the root of this tree, and `PathList` is a list of the nodes along this highest cost path starting with the name of the root and ending with the last non-`none` node.

For example, if your program is called as `highestCostPath(Tree, X, Y)` given the node a from the tree below, then your program should succeed with `X=7` and `Y = [a, b, f]`.


$$\begin{tikzcd}
 & & a \arrow[dl, "2"'] \arrow[d, "3"] \arrow[dr, "1"] & \\
 & b \arrow[dl, "3"'] \arrow[d, "5"] & c & d \arrow[dl, "2"] \\
e & f & g \arrow[dr, "1"] & \\
 & & & h
\end{tikzcd}$$

You should write your program in the file `question3_longest_path.pl`. You may add helper predicates as you see fit. The tree above has already been included in that file for testing purposes, though you may want to try additional trees to ensure your program works.

4 The Merge Sort Algorithm [15 marks]

For this question, you will implement the merge sort algorithm on a list implemented as a term. That is, we will use the term `listTerm(H, T)` to represent the Prolog list `[H | T]`, where `empty_list` represents the empty list. For example, the Prolog list `[4, 5, 2, 8]` will be represented by the term:

```
listTerm(4, listTerm(5, listTerm(2, listTerm(8, empty_list) ) ) )
```

You program should take in a `listTerm` and sort the list using merge sort. You should implement the following helper programs:

- Create a program called `listLength(ListTerm, Length)` which finds the length, i.e., the number of elements, of a given term `ListTerm` and outputs the number `Length`.
- Create a program called `divideList(ListTerm, Num, FirstHalf, SecondHalf)` which takes in a list term `ListTerm` and creates two new list terms such that `FirstHalf` contains the first `Num` elements in `ListTerm` and `SecondHalf` contains the remaining elements.
- Create a program called `mergeSortedLists(SortedListTerm1, SortedListTerm2, MergedList)` which takes in two sorted list terms, and merges them into a list `MergedList` that is also sorted.

Note that none of these programs should convert a given list term to a Prolog list before performing their operation. They should all work directly on the terms.

Once you have created these helper predicates, use them to implement `mergeSort(ListTerm, SortedListTerm)` which sorts the input list `ListTerm` to produce the output list `SortedListTerm` using merge sort. The following are some example runs

```
?- mergeSort(listTerm(4, listTerm(5, listTerm(2, listTerm(8, empty_list) ) ) ), X)
    Yes with X= listTerm(2, listTerm(4, listTerm(5, listTerm(8, empty_list) ) ) ).
?- mergeSort(empty_list, X).
    Yes with X = empty_list
?- Input = listTerm(25, listTerm(6, listTerm(99, listTerm(101,  listTerm(105,
             listTerm (7,  listTerm(8, empty_list) ) ) ) ) ) ), mergeSort(Input, X).
    Yes with X = listTerm(6, listTerm(7, listTerm(8, listTerm(25,  listTerm(99,
                     listTerm (101,  listTerm(105, empty_list) ) ) ) ) ) )
```

You should write your program in the file `question4_merge_sort.pl`. Ensure that your main program and helper programs are all in the correct sections of the file. You may lose marks if you do not follow that requirement. Additional space has also been provided in case you want to add additional helper programs, but this file should not include any atomic statements.