CPS721: Assignment 2

Due: October 7, 2025, 9pm
Total Marks: 110 (worth 4% of course mark)
In groups of no more than 3

Late Policy: Late submissions are penalized by $3^n\%$ when they are n days late.

Clarifications and Questions: Please use the discussion forum on the D2L site to ask questions. These will be monitored regularly. A Frequently Asked Questions (FAQ) page will also be created. Only ask questions by email if it is specific to your submission. Any questions asked with less than 12 hours left to the deadline are not likely to be answered.

PROLOG Instructions. Your code must run on ECLiPSe 7.1. If you cannot install this on your own machine, test it on the lab machines or the department moon servers.

You are NOT allowed to use ";" (disjunction), "!" (cut), "->" (if-then), `setof`, `bagof`, `findall`, or any other language features and library predicates that are not covered in class. Similarly, you should only use "=" and "is" for equality, and "not" for negation. Do NOT use "=:=", "\=", or other such commands. You will be penalized if you use these commands/features. You are only allowed to use ";" to get additional responses when interacting with PROLOG from the command line (this is equivalent to using the "More" button in the ECLiPSe GUI).

You ARE allowed to use the built-in `member` or `append` methods if they are useful to you.

Academic Integrity and Collaboration Policy: Any submission you make must be the work of your group and your group alone. You are also not allowed to post course materials (including assignments, slides, etc.) anywhere on the web, and you are also prohibited from using generative AI systems or online "help" websites for completing assignments.

Submission Instructions: You should submit ONE zip file called `assignment2.zip` on D2L. It should contain the following files:

```
assignment2_submission_info.txt
q1_list_equality.pdf              q2_increasing_power_sum.pl
q3_split_on_int.pl                q4a_bacon.pl
q4b_bacon.pl                      q5_zipper.pl
q6_highest_cost_path.pl
```

All these files have been given to you and you should fill them out using the format described. Ensure your names appear in all files — especially `assignment2_submission_info.txt` — and your answers appear in the file sections as requested. Do NOT edit, delete, or add any lines that begin with "`%%%%% SECTION:`". These are used by the auto-grader to locate the relevant code. You will lose marks if you edit such lines.

Your submission should not include any other files than those above. You will be penalized for using compression formats other than `.zip`. Submissions by email will not be accepted.

You may submit your solution multiple times. Doing so will simply overwrite your previous submission. We will therefore always mark the last submission your group makes.

1 List Equality [15 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, element by element, or why they can't be made identical. 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 explanations for why the lists do or do not match will lose marks.

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

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

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

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

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

Each of these pairs is worth 3 marks. You should submit your answers in a pdf file called `q1_list_equality.pdf`. The names, emails, and student IDs of your group members should appear at the top of this PDF file.

Note, you are permitted to use ECLiPSe to check the answers to these questions, but you need to provide the proof itself.

2 Increasing Power Sum [10 marks]

Write a program called `increasingPowerSum(List, Power, PowerInc, Sum)` where `List` is a list of integers, and `Power` and `PowerInc` are both non-negative integers. The value of `Sum` should be the sum of the integers in the list, each taken by a different power starting with `StartingPower` and incremented by `PowerInc`. This means that `Sum` will be given by the sum of the first element of the list to the exponent of `Power`, plus the second element of the list to the exponent of `Power + PowerInc`, plus the third element of the list to the power of `Power + 2 * PowerInc`, etc.

Below you can find examples of queries using this predicate.

```
?- increasingPowerSum([1, 2, 3, 4], 1, 2, 16636).
    Yes
```

This holds because $1^1 + 2^3 + 3^5 + 4^7 = 16636$

```
?- increasingPowerSum([5, 4, 3], 1, 0, X).
    Yes with X = 12
```

This holds because $5^1 + 4^1 + 3^1 = 12$

Note that `X^Y` computes `X` to the exponent `Y`. Your program should return 0 if it is given an empty list. We will not test with enormous numbers or negative powers to avoid overflow or floating point issues. You can assume that whenever your program is called, `List`, `Power`, and `PowerInc` will not be variables.

Submit your program, including any helper predicates, in the `q2_rules` section of the file `q2_increasing_power_sum.pl`.

3 Split on Integers [10 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, LargerList)` - 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 `LargerList` is the list of values in `List` larger than `Value`.

Below you can find examples of queries using this predicate.

```
The following should succeed
?- splitOnInt([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).
```

You can assume that `List` and `Value` are always given as inputs (*ie.* we will never test with them set as variables).

Submit your program, including any helper predicates, in the `q3_rules` section of the file `q3_split_on_int.pl`.

4 Lists and "Six Degrees of Kevin Bacon" [40 marks]

For this question, you will explore writing a program for playing "Six Degrees of Kevin Bacon" just like Assignment 1. The difference is that you will now be using lists to avoid exploring paths that have loops in them.

a. **[20 marks]** In the first version of this problem you will write a program for the following predicate:

```
canReach(A1, A2, M, ActPath, MoviePath)
```

Here, `A1` and `A2` are actors, `M` is the maximum path length to consider, `ActPath` is the actual path of actors found from `A1` to `A2`, and `MoviePath` is the actual path of movies found from `A1` to `A2`. For example, consider a KB that contains Kevin Bacon, John Lithgow, and Mike Myers, where Kevin Bacon and John Lithgow acted together in "Footloose", and John Lithgow and Mike Myers acted together in "Shrek". Then the following should hold:

- `canReach(kevin_bacon, kevin_bacon, 5, [kevin_bacon], [])` holds since an actor can reach themselves in 0 or more steps provided they have acted in at least one movie in the KB ("Footloose" in this case).
- `canReach(kevin_bacon, john_lithgow, 1, [kevin_bacon, john_lithgow], [footloose])` holds since both Kevin Bacon and John Lithgow appeared in "Footloose". It would also hold for any `M` larger than 1.
- `canReach(kevin_bacon, mike_myers, 2, [kevin_bacon, john_lithgow, mike_myers], [footloose, shrek])` holds.

For other requirements, see the definition of `canReach` from Assignment 1. One significant difference with Assignment 1 is that no actor should appear more than once in `ActPath` and no movie should appear more than once in `MoviePath`. The actors and movies in `ActPath` and `MoviePath`, respectively, should appear in the order they appear in the path from `A1` to `A2`. Most marks will be given for handling queries where only `ActPath`, `MoviePath` are variables (or are given as input). However, for full marks, your program should handle having any combination of `A1`, `A2`, `ActPath`, and `MoviePath` being provided as variables. As in Assignment 1, you can assume that `M` is a non-negative integer that is given as input.

Submit your program, including any helper predicates, in the `q4a_rules` section of the file `q4a_bacon.pl`. YOUR TEST KB SHOULD NOT APPEAR IN THIS FILE. A section for importing your KB from another file has been given. You do NOT need to submit the KB you use for testing.

a. **[20 marks]** Next, you will consider the following generalization of `canReachThroughMovie` and `canReachThrough2Movies` from Assignment 1.

```
canReachThroughMovies(A1, A2, M, TargetMovies, ActPath, MoviePath)
```

`A1`, `A2`, `M`, `ActPath`, and `MoviePath` are defined as in part (a). `TargetMovies` are a list of movies that must be included at some point in the path (*ie.* all movies in `TargetMovies` must appear in `MoviePath`). The list of `TargetMovies` can appear in any order (*ie.* the movies can be "used" in any order in `MoviePath`). Thus, the following hold:

- `canReachThroughMovies(kevin_bacon, mike_myers, 2, [footloose], [kevin_bacon, john_lithgow, mike_myers], [footloose, shrek]).`
- `canReachThroughMovies(kevin_bacon, mike_myers, 72, [shrek], [kevin_bacon, john_lithgow, mike_myers], [footloose, shrek]).`
- `canReachThroughMovies(kevin_bacon, mike_myers, 72, [shrek, footloose], [kevin_bacon, john_lithgow, mike_myers], [footloose, shrek]).`

- `canReachThroughMovies(kevin_bacon, mike_myers, 72, [footloose, shrek], [kevin_bacon, john_lithgow, mike_myers], [footloose, shrek])`.

Most of the requirements also hold over from the definitions of `canReachThroughMovie` and `canReachThrough2Movies` from Assignment 1. However, please note the following:

- Unlike Assignment 1, you can assume that `TargetMovies` is given as an input, and we will never test it when set as a variable.
- You can assume that no movie appears in `TargetMovies` more than once. This differs from `canReachThrough2Movies` in Assignment 1, which allowed both movies to be the same.
- If `TargetMovies` is empty, then `canReachThroughMovies` should work identically to `canReach`. Thus, the predicate should succeed if `A1 = A2` for any `M >= 0`, as long as the actor in question appears in at least one movie in the KB.
- As in Assignment 1, self-loops are not allowed to satisfy the target movies requirement, meaning that if `TargetMovies` consists of a single movie, the actor cannot reach themselves in 0 steps even if they appeared in the target movie. Notice that this fact, along with the requirement that `ActPath` cannot contain the same actor more than once, means that `canReachThroughMovies` can never be true if `A1 = A2`, regardless of the value of `M`, unless `TargetMovies` is empty.
- Note that the fact that each actor can only appear once may actually make some paths impossible that were previously possible in Assignment 1.
- Again, most marks will be given for handling queries where only `ActPath`, `MoviePath` are variables (or are given as input). However, for full marks, your program should handle having any combination of `A1`, `A2`, `ActPath`, and `MoviePath` being provided as variables.

Submit your program, including any helper predicates, in the `q4b_rules` section of the file `q4b_bacon.pl`. YOUR TEST KB SHOULD NOT APPEAR IN THIS FILE. A section for importing your KB from another file has been given. You do NOT need to submit the KB you use for testing.

5 List Zipper [15 marks]

For this question, you will use the term `next(Head, Tail)` to represent a Prolog list `[Head | Tail]`. Here, we will use `nil` to represent the empty list `[]`. For example,
`next(7, next(1, next(5, next(0, next(9, nil)))))` represents the list `[7,1,5,0,9]`.

Write a program, `zipper(List1, List2, Result)` which takes in two lists, `List1` and `List2`, specified using the notation given above, and creates a new list called `Result` given by alternating the elements in the first two lists. The alternation should start with an element from `List1` (assuming one exists). If one list runs out of elements before the other, then the remainder of the list with elements should be added to the end of `Result`.

The following are examples of expected behaviour:

```
?- zipper(next(1, next(2, next(a, next(b, nil)))),
          next(f, next(g, next(c, next(d, nil)))),
          next(1, next(f, next(2, next(g, next(a, next(c, next(b, next(d, nil)))))))) ).
Yes

?- zipper(next(a, next(b, next(c, next(d, nil)))), next(1, next(2, nil)), Z).
Yes with Z = next(a, next(1, next(b, next(c, next(d, nil)))))
```

Most of the marks will be for handling cases as shown above. For full marks, your program should also be handle the following cases:

- `List1` is a variable, while `List2` and `Result` are not. For example, your program should handle `zipper(X, next(a, next(b, nil)), next(1, next(a, next(2, next(b, nil))))).`
- `List2` is a variable, while `List1` and `Result` are not. This is similar to the first case.
- Both `List1` and `List2` are variables and `Result` is not, For example, your program should handle `zipper(X, Y, next(1, next(a, next(2, next(b, nil))))).`. Using more should find all possible combinations of `X` and `Y` that will zip together to get the given list.

You do NOT need to handle the case where `Result` is a variable, and so either `List1` or `List2`.

Submit your program, including any helper predicates, in the `5_rules` section of the file `q5_zipper.pl`.

6 Highest Cost Path [20 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)`

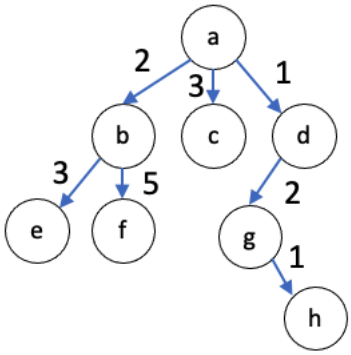
The arguments are defined as follows:

- `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 `tree3` 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]`.



Your can always assume that your program will be called with the tree being input. That is, `Tree` will never be tested as an unbound variable.

If there are multiple highest cost paths, we should be able to find all of them using "More", but it should never return a false answer. You will likely get A LOT of repetitive answers. As in the rest of the class (unless explicitly stated otherwise), that is entirely expected and fine. DO NOT use operators like '!' to try to decrease these repetitions. That is also true of all assignments, as specified on the first page of this PDF.

You should write your program in the `q6_rules` section of the file `q6_highest_cost_path.pl`. You may add helper predicates as you see fit. The tree above has already been included in that file for testing purposes. Instructions for how to use it are given. You may want to try additional trees to ensure your program works in all cases, but we will not mark them.