CSE 505, Fall 2021 # HW1: Prolog Basics Due: Tue., Sep 21	Version 1.0.1 Sep 9, 2021

This is a Prolog programming assignment. You are expected to write a Prolog program that consists of predicates described below. You will place the definitions of all the predicates in a single file and submit it via Blackboard.

Warm-Up

1. **prefix:** Write a predicate **`prefix(L1,L2)`** that succeeds if and only if list **`L2`** is a *prefix* of list **`L1`**: i.e. all elements if **`L2`** occur, in the same order, at the beginning of **`L1`**. For instance:
   - `prefix([1,2,3], [])`, `prefix([1,2,3], [1])`, `prefix([1,2,3], [1,2])`, `prefix([1,2,3], [1,2,3])` all succeed.
   - `prefix([1,2,3],[2])`, `prefix([1,2,3], [1,4])`, `prefix([1,2], [1,2,3])` all fail.

   For this question, your predicate definition cannot use any helper predicates.
   For full credit, when **`L2`** is not given, your predicate should return the possible values of **`L2`** by backtracking. For instance, `prefix([1,2,3], L2)` should return `L2 = []`, and upon backtracking, return `L2 = [1]`, `L2 = [1,2]` and `L2 = [1,2,3]`.

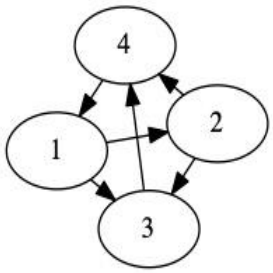
2. **increasing_subsequence:** Write a Prolog predicate **`incsub`** such that, given a list of integers **`L1`**, **`incsub(L1, L2)`** returns in **`L2`** an increasing subsequence of **`L1`**. (You may assume that **`L1`** contains distinct elements).
   L₂ is a subsequence of ***L₁*** if all elements of ***L₂*** also occur in ***L₁***, and if ***a*** occurs before ***b*** in ***L₂*** then ***a*** occurs before ***b*** in ***L₁***.
   L is an increasing sequence if the elements are in ascending order: i.e. if ***a*** occurs before ***b*** in ***L***, then ***a < b***.
   For instance, `incsub([2,4,1,7,3,8], L2)` should succeed with answers `L2 = []`, `L2 = [2]`, `L2 = [4]`, `L2 = [2,4]`, `L2 = [1]`, ... (upon backtracking; a total of 26 distinct answers, not necessarily in this order).

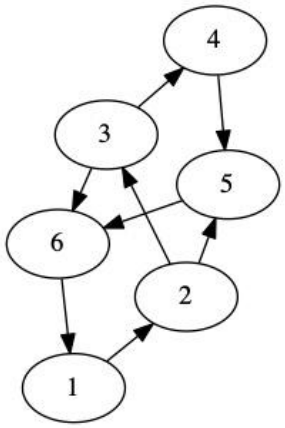
3. **rotate**: Write a Prolog predicate **`rotate(L1, L2)`** that succeeds if and only if list **`L2`** is a *prefix* of list **`L1`**. A rotation of list $[a_1,a_2, \ldots, a_n]$ is a list $[a_k, a_{k+1}, \ldots, a_n, a_1, a_2, \ldots, a_{k-1}]$ for some *k* in *[1,n]*. For example, [2,3,4,5,1] and [4,5,1,2,3] are rotations of [1,2,3,4,5]. For full credit, when **`L2`** is not given, your predicate should return the possible values of **`L2`** by backtracking. For instance, `rotate([1,2,3], L2)` should return `L2 = [1,2,3]`, and upon backtracking, return `L2 = [2,3,1]` and `L2 = [3,1,2]` (not necessarily in the same order).

Graph Problems:

The following set of problems are over graphs, and we will use the following encoding to represent ***directed*** graphs in Prolog.

Each graph is given an identifier (name or number), and we enumerate its edges as triples in a 3-ary **`edge`** relation. Each triple of the form **`edge`(*i*, *v1*, *v2*)** represents an edge in the graph ***i*** from vertex ***v1*** to vertex ***v2***. Example graphs and their Prolog encoding are shown below:

a		`edge(a, 1, 2).`
		`edge(a, 1, 3).`
		`edge(a, 2, 3).`
		`edge(a, 2, 4).`
		`edge(a, 3, 4).`
		`edge(a, 4, 1).`

b		`edge(b, 1, 2).`
		`edge(b, 2, 3).`
		`edge(b, 3, 4).`
		`edge(b, 4, 5).`
		`edge(b, 5, 6).`
		`edge(b, 6, 1).`
		`edge(b, 2, 5).`
		`edge(b, 3, 6).`

4. **`exists_path:`** Write a binary predicate **`exists_path`** such that, given a graph ***I*** and a list of vertices ***P***, **`exists_path`(*I*, *P*)** succeeds if and only if vertices in ***P*** forms a path in the graph.

   For the example graphs above, `exists_path(a, [1,3,4])`, `exists_path(a, [2,4,1,3,4,1,2])`, `exists_path(b, [1,2,3,4,5,6])` all succeed; `exists_path(a, [1,2,3])` and `exists_path(b, [1,2,3,5])` fail.

5. **`exists_simple_path:`** Write a binary predicate **`exists_simple_path`** such that, given a graph ***I*** and a list of vertices ***P***, **`exists_simple_path`(*I*, *P*)** succeeds if and only if vertices in ***P*** forms a **simple (i.e cycle-free) path** in the graph.

   For the example graphs above, `exists_simple_path(a, [1,3,4])` and `exists_simple_path(b, [1,2,3,4,5,6])` succeed; `exists_path(a, [1,2,3]`, `exists_path(a, [2,4,1,3,4,1,2])`, and `exists_path(b, [1,2,5,6,1,2,3])` fail.

6. **`simple_path`**: Write a 4-ary predicate **`simple_path`** such that, for queries of the form **`simple_path`(*i*, *v1*, *v2*, *P*)** where ***i*** is a graph id, ***v1*** and ***v2*** are vertices in graph ***i***, succeeds by binding ***P*** to a list of vertices that lie on a simple path (i.e. cycle-free) path from ***v1*** to ***v2***.
   For example, `simple_path(a, 1, 4, Y)` should succeed with `Y=[1,2,4]`, and upon backtracking, `Y=[1,3,4]` and `Y=[1,2,3,4]`. The order of answers does not matter.

Analysis of Program Traces

For this problem, we consider *traces* of a procedural program's execution. We log four events in these traces:

- `malloc`(*p*): Allocate a block of memory and assign its address to pointer *p*.
- `read`(*p*): Read the value at the address in pointer *p*
- `write`(*p*): Write a value to address in pointer *p*
- `free`(*p*): Free (i.e. de-allocate) memory located at the address in pointer *p*.

A trace is a list of such events. (**Note:** this problem assumes no aliases: no two pointers refer to the same address in memory).

7. Write a predicate **`invalid_access`**, that, given a trace, succeeds if and only if the trace has a read/write event after that pointer has been freed, or before that pointer is malloced.
8. Write a predicate **`memory_safe`**, that, given a trace, succeeds if and only if all the memory operations are safe:
    - reads and writes are to previously malloced (and not yet freed) memory.
    - free refers to a previously malloced (but not yet freed) memory.
9. Write a predicate **`leak`**, that, given a trace, succeeds if and only there is a memory leak: a pointer is re-assigned with a malloc without freeing the earlier block.
10. Write a predicate **`useful_writes`**, that, given a trace, succeeds if and only if every value that is written to an address is read before the pointer is freed or re-allocated.

Your program should be in a **single** file (see submission instructions below).

Grading:

All problems are worth 3 points each. The grading rubric will be:

3	A correct solution or a solution with very few flaws
2	A solution that is substantially correct; some flaws but mostly correct
1	A solution that is substantially incorrect; many flaws with few correct parts
0	Incorrect or missing solution

Code documentation is expected unless the intent of the solution is obvious from its form. Partial credit may not be possible without clear documentation.

Submission:

Your submission will consist of a **single** Prolog program file (name of the file and the file extension does not matter for the submission). Submit the file as an attachment to the HW1 form on Blackboard.

Important: Work on the warm-up problems individually. You can work on Graph and Trace problems with 1 other team member. In that case, both have to submit complete solutions separately. Make sure that your submission has a clear marking of who your team mate is.

Errata:

- Sep 9, 9:30am: Added note in the “Submission” section on teams (in purple)