COMP 410
Fall 2025

Final Practice Exam

The final exam is cumulative, so **all** handouts, assignments, practice exams, and prior exams are relevant. You may bring three 8.5 x 11 inch sheets of paper into the exam, with some combination of printed and handwritten notes. This practice exam only covers material since exam 2.

Recursion in Prolog

The first two problems have been copy/pasted from practice exam #2. These have been copied because the final is guaranteed to have related problems covering recursion, especially recursion over lists.

1.) Write a procedure named `zip`, which takes two lists of the same length, an output list of the same length. The output list is a list of `pair` structures, where each `pair` holds an element from each list, preserving order. If the lists are not the same length, `zip` should fail, though you shouldn't need to explicitly check the length. Example calls are below:

```
?- zip([], [], Output).
Output = [].
?- zip([hello], [goodbye], Output).
Output = [pair(hello, goodbye)].
?- zip([1, 2, 3], [a, b, c], Output).
Output = [pair(1, a), pair(2, b), pair(3, c)].
?- zip([A, B], [C, D], Output).
Output = [pair(A, C), pair(B, D)].
?- zip([foo], [bar, baz], Output).
false.
?- zip([foo, bar], [baz], Output).
false.
```

2.) We can represent a binary tree in Prolog using the following:

- `leaf`: At atom representing a leaf node
- `internal(tree, value, tree)`: A structure recursively containing two trees and some integer value

Write a procedure named sumTree that will compute the sum of all the elements in a given binary tree. Leaf nodes are defined to have a sum of 0. Example queries are shown below:

```
?- sumTree(leaf, Sum).
Sum = 0.
?- sumTree(internal(leaf, 5, leaf), Sum).
Sum = 5.
?- sumTree(internal(internal(leaf, 1, leaf),
                    2,
                    internal(leaf, 3, leaf)),
           Sum).
Sum = 6.
```

3.) Consider the following code, computing the length of a list:

```
len([], 0).
len([_|T], Len) :-
     len(T, TLen),
     Len is TLen + 1.
```

3.a) This procedure is not very efficient when it comes to memory. Why is it inefficient?

3.b) Rewrite this procedure to be more efficient with memory. You may introduce a helper procedure if desired.

4.) Consider the following code which appends two lists together:

```
append([], List, List).
append([H|T], List, Result) :-
  append(T, List, Rest),
  Result = [H|Rest].
```

4.a) This procedure is not very efficient when it comes to memory. Why is it inefficient?

4.b) Rewrite this procedure to be more efficient with memory. You may introduce a helper procedure if desired.

5.) For this problem, you will implement quick sort, which is done as multiple parts. The actual exam will **not** have a single recursive problem which requires multiple steps in this manner. This problem is provided as more practice.

5.a.) Define a procedure named `partition`, which takes the following in this order:

1. An input list of integers
2. An input "pivot" value, an integer
3. An output list of all values less than or equal to the pivot
4. An output list of all values greater than the pivot

`partition` should split up the input list into two output lists: one containing all values <= the pivot, and another output list containing all values > the pivot. Example queries to `partition` are on the next page.

```
?- partition([], 5, LTE, GT).
LTE = GT, GT = [].

?- partition([3], 5, LTE, GT).
LTE = [3],
GT = [] .

?- partition([7], 5, LTE, GT).
LTE = [],
GT = [7].

?- partition([3, 2, 8, 9, 3, 4, 7], 4, LTE, GT).
LTE = [3, 2, 3, 4],
GT = [8, 9, 7] .

?- partition([3, 2, 8, 9, 3, 4, 7], 5, LTE, GT).
LTE = [3, 2, 3, 4],
GT = [8, 9, 7] .
```

Implement `partition` below.

5.b.) Using your `partition` from the prior part, implement a procedure named `quicksort` which will sort a given input list of values. As a reminder, `quicksort` works as follows:

- Empty lists are already sorted
- For a non-empty list, select an element from the list as the pivot. (For simplicity, the first element is good for this purpose). From there, `partition` the list into two lists: one containing only elements <= the pivot, and one containing only elements > the pivot.
- Recursively `quicksort` the two lists, yielding two sorted lists. It is guaranteed that all elements in the first list will be less than all elements in the second list, because of the partitioning around pivot.
- Combine the two sorted lists using `append` to form the final output list.

Some example calls to `quicksort` are below:

```
?- quicksort([], List).
List = [].

?- quicksort([3, 2, 7, 4], List).
List = [2, 3, 4, 7] .
```

Implement `quicksort` below.

6.) For this problem, you will implement merge sort, which is done as multiple parts. The actual exam will **not** have a single recursive problem which requires multiple steps in this manner. This problem is provided as more practice.

6.a.) Define a procedure named `merge`, which will merge together two sorted lists into a single sorted list. merge should run in `O(n)`. You may assume that the input lists are sorted. Example queries are below.

```
?- merge([], [6, 8], List).
List = [6, 8] .

?- merge([2, 7, 8], [], List).
List = [2, 7, 8] .

?- merge([2, 4, 8, 9], [1, 2, 3, 7, 9, 10], List).
List = [1, 2, 2, 3, 4, 7, 8, 9, 9, 10].
```

Implement `merge` below.

6.b.) Define a procedure named `splitAt`, which takes the following in this order:

1. An input list of elements.
2. Some integer `N`, where `N` is assumed to be in the range `0 <= N < length(inputList).`
3. An output list of all elements at indices `< N`. The list is assumed to be 0-indexed (starts at index 0).
4. An output list of all elements at indices `>= N`.

Example calls to `splitAt` are shown below.

```
?- splitAt([foo, bar, baz], 0, Before, After).
Before = [],
After = [foo, bar, baz] .

?- splitAt([foo, bar, baz], 1, Before, After).
Before = [foo],
After = [bar, baz] .

?- splitAt([foo, bar, baz], 2, Before, After).
Before = [foo, bar],
After = [baz] .

?- splitAt([foo, bar, baz], 3, Before, After).
Before = [foo, bar, baz],
After = [].
```

Define `splitAt` below.

6.c.) Define a procedure named `mergeSort`, which will use the `merge` and `splitAt` procedures you previously defined in order to apply merge sort to a given list. As a reminder, merge sort works as follows:

- Empty lists are already sorted
- Lists containing only one element are already sorted
- For lists containing more than one element:
    - Determine the length of the list using `length`
    - Split the list into two halves of roughly equal size, using the length computed in the prior step and `splitAt`
    - Recursively apply merge sort to each of these two smaller lists
    - Recombine the sorted sublists using `merge`, forming a single sorted result list

Some example calls to `mergeSort` are below:

```
?- mergeSort([], List).
List = [].

?- mergeSort([3], List).
List = [3] .

?- mergeSort([3, 2], List).
List = [2, 3] .

?- mergeSort([3, 2, 5, 7], List).
List = [2, 3, 5, 7] .
```

Implement `mergeSort` below. The next page is blank in case you need it.