

**COMP 410**  
**Summer 2025**  
**Midterm Practice Exam #2**

This is representative of the kinds of topics and kind of questions you may be asked on the midterm. This practice exam, along with assignments 3-6 and the in-class handouts from Unification and Recursion through Optimizations, are intended to be comprehensive of everything on the exam. That is, I will not ask anything that's not somehow covered by those sources.

You are permitted to bring two 8.5 x 11 sheets of paper into the exam with you, and these may have anything on them, **as long as it is handwritten**. Both sides of both sheets can be used.

**Unification with Lists**

Consider each of the following unification attempts involving lists. If the unification succeeds, record any values any variables take. If the unification fails, say so.

1.)  $[1, 2, \_] = [A, B, C|D]$

2.)  $A = [1, 2|B], B = [4]$

3.)  $[[A|B], C] = [[1, 2]|D]$

4.)  $X = [A|[2]]$

$$5.) [A, [B, [C|D]]] = [1, [2, [3, 4]]]$$

6.) Consider the following inductive list definition, which makes use of Prolog atoms and structures:

$$e \in ListElement$$

$$\ell \in List ::= cons(e, \ell) \mid nil$$

Now consider the following unifications, using Prolog lists. Rewrite these unifications using the above definition.

$$6.a.) X = [1, 2, 3]$$

$$6.b.) X = [Y|Z]$$

$$6.c.) X = [A|[2]]$$

$$6.d.) X = [1, [2, [3]]]$$

## Recursion

7.) Consider the following mathematical definition of a recursive function:

$$f_n = \begin{cases} 2 & \text{if } n = 0 \\ 3 & \text{if } n = 1 \\ (3 \times f_{n-1}) + (4 \times f_{n-2}) & \text{otherwise} \end{cases}$$

Write an equivalent definition in Prolog.

8.) Write a procedure named `evensBetween`, which will nondeterministically produce all the even numbers within an inclusive range. As a hint, a number `N` is even if and only if the clause `0 is mod(N, 2)` is true. An example query is below:

```
?- evensBetween(1, 4, Even).  
Even = 2 ;  
Even = 4.
```

9.) Write a procedure named `allEqual` which will succeed if all list elements are equal to each other according to unification (=). You may introduce any helpers you wish. Example calls are below:

```
?- allEqual([]).  
true.  
?- allEqual([1, 1, 1]).  
true.  
?- allEqual([1, 2, 3]).  
false.  
?- allEqual([1, X, 1]).  
X = 1.  
?- allEqual([A, B]).  
A = B.  
?- allEqual([X, 1, 2]).  
false.
```

10.) 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.
```

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

- `leaf`: An 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.
```

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

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

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

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



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

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

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

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

14.) Define a procedure named `isPrime` which will determine if a given input number is prime. You may introduce any helpers you wish. Example queries follow:

```
?- isPrime(2).  
true .  
?- isPrime(3).  
true .  
?- isPrime(4).  
false.
```

As a hint, the following Java-like code:

```
int x = y % z;
```

...is equivalent to the following Prolog code:

```
X is mod(Y, Z)
```

## Test Case Generation

15.) Consider the following grammar-based definition of simplistic SQL queries:

$$\begin{aligned} c &\in \textit{ColumnName} & t &\in \textit{TableName} \\ q &\in \textit{SQLQuery} ::= \textbf{select } c \textbf{ from } t; \end{aligned}$$

15.a) Assume the only possible columns are named `c1` and `c2`, and the only possible tables are named `t1` and `t2`. Write a generator of valid SQL query ASTs. An example of a valid AST is `select (c1, t1)`. Do not simply hardcode all possible ASTs.

15.b) Bounds or related mechanisms are not necessary for this problem, at least as described. Why?

15.c) Name a change to this problem which would necessitate adding a bound or a related mechanism, and explain why such a change would add this necessity.

16.) Consider the following grammar:

```
tree ::= `node(` tree `,` tree `)` | `leaf`
```

16.a.) Write a generator for `tree` below. It's ok if the generator gets "stuck" generating similar values over and over again.

16.b.) Write a modified version of your prior generator, which takes an additional depth bound, and will only generate values that are no deeper than this bound.