# Prolog Supervision 1

## Prolog Basics

### 1.1

An atom unifies with another atom if and only if they are identical.

A variable unifies with anything (although implementations vary when the variable occurs within a compound statement with which it is trying to unify)

A compound statement unifies with another compound statemnt if and only if their functors and arities match.

### 1.2

```
tree(tree(tree(1,2),A,B),tree(C,tree(E,F,G)))
tree(C,tree(Z,C))


tree(tree(tree(1,2),A,B),tree(tree(tree(1,2),A,B),tree(E,F,G)))
tree(tree(tree(1,2),A,B),tree(Z,tree(tree(1,2),A,B)))
// (Variable C)

tree(tree(tree(1,2),A,B),tree(tree(tree(1,2),A,B),tree(E,F,G)))
tree(tree(tree(1,2),A,B),tree(tree(tree(1,2),A,B),tree(tree(1,2),A,B)))
// (Variable Z)

tree(tree(tree(1,2),A,B),tree(tree(tree(1,2),A,B),tree(tree(1,2),F,G)))
tree(tree(tree(1,2),A,B),tree(tree(tree(1,2),A,B),tree(tree(1,2),A,B)))
// (Variable E)

tree(tree(tree(1,2),A,B),tree(tree(tree(1,2),A,B),tree(tree(1,2),A,G)))
tree(tree(tree(1,2),A,B),tree(tree(tree(1,2),A,B),tree(tree(1,2),A,B)))
// (Variable F)

tree(tree(tree(1,2),A,B),tree(tree(tree(1,2),A,B),tree(tree(1,2),A,B)))
tree(tree(tree(1,2),A,B),tree(tree(tree(1,2),A,B),tree(tree(1,2),A,B)))
// (Variable G)
```

One possibility is that the unification will fail, because the variable `A` occurs in the compound statement `a(A)`, and unification would result in an infinite loop. "Occurs" checks can negatively impact performance, and are rarely necessary in practice. As such, some Prolog implementations will return such an infinite loop.

### 1.2

```
// TODO
```

# Zebra Puzzle

## 2.1

1.2. This is because the given outcome can be reached with the fewest unifications, and so is returned earliest.

2.1. This puzzle is states as a series of facts about relationships between entities, and then asks questions which require deducing other facts about those relationships between those entities.

```
// TODO: part 2
```

## 2.2

"*The Spanish person has a dog*" is expressed as "*There exists a house whose nationality is spanish and whose pet is dog*"

# Rules

## 3.1

```
// TODO
```

## 3.2

$\forall X ((\textrm{material}(X) \lor \exists Y (\textrm{production}(X,Y) \land \textrm{valuable}(Y))) \implies \textrm{valuable}(X))$

## 3.3

```
// TODO
```

# Lists

## 4.1

- `[]`: The empty list
- `[A,B,C]`: The list containing `A`, `B`, and `C`
- `[A|T]`: The list with head `A` and tail `T`

## 4.2

```
last([1,2],A).

    // [H] = [1,2]
    // H = A
    // ❌

    // [_|T2] = [1,2]
    // H2 = A
    last([_|T2],H2) :- last(T2,H2).
```

```
        // [H3] = [2],
        // H3=A
        last([H3],H3).

            // A = 2 ☑
```

## 4.3

`append(T,A,R)` should be used to mean that `R` is the result of appending the list `A` to the list `T`.

For example:

```
append([1,2],[3,4],R).

    // [] = [1,2]
    // A = [3,4]
    // A = R
    // ✗

    // [H2|T2] = [1,2]
    // A2 = [3,4]
    // [H2|R2] = R
    append([H2|T2],A2,[H2|R2]) :- append(T2,A2,R2).

        // [] = 2
        // A3 = [3,4]
        // A3 = R2
        // ✗

        // [H4|T4] = 2
        // A4 = [3,4]
        // [H4|R4] = R2
        append([H4|T4],A4,[H4|R4]) :- append(T4,A4,R4).

            // [] = []
            // A5 = [3,4]
            // A5 = R4
            append([],A5,A5). // ☑

        // append([2],[3,4],[2,3,4]).

    // append([1,2],[3,4],[1,H4|R4]).

        // [] = [1,2]
        // A6 = [3,4]
        // A6 = [1,H4|R4]
        // ✗

        // [H7|T7] = [1,2]
        // A7 = [3,4]
        // [H7|R7] = [1,H4|R4]
```

```
append([H7|T7],A7,[H7|R7]) :- append(T7,A7,R7).

    // [] = [2]
    // A8 = [3,4]
    // A8 = [H4|R4]
    // ❌

    // [H9|T9] = [2]
    // A9 = [3,4]
    // [H9|R9] = [H4|R4]
    append([H9|T9],A9,[H9|R9]) :- append(T9,A9,R9).

        // [] = []
        // A10 = [3,4]
        // A10 = [H4|R4]
        append([],A10,A10). // ☑

    // append([2], [3,4], [2,3,4]).

// append([1,2],[3,4],[1,2,3,4])
```

## 4.4

1. `// TODO`
2. No. Aside from making the code less readable, Prolog is a declarative language, not an imperative one. This means that inlining as a concept doesn't really apply, as in either case the programmer has no way of specifying *how* the interpreter should perform these queries.
3. I think that the second implementation is preferable, as syntactically it looks a bit like `_X_`, which to me gives more of an intuitive sense of what the relationship actually means — anything, followed by `X`, followed by anything.
4. `// TODO`

## 4.5

`a(X)` means that `X` is a sorted list (asc.). `a(X,Y)` means that `X` would be a sorted list (asc.) if `Y` was to be prepended to it.

## 4.6

`B(X,Y)` means that `Y` is the sorted version of `X`.

It works by first stating that every sorted list is the sorted version of itself. It then states that `B(X,Y)` if there exist two consecutive values `H1` and `H2` in `X`which are in the wrong order, and were they to be switched, `Y` would also be the sorted version of the resulting list.

During the recursion, eventually all incorrectly ordered pairs would be switched, and the tree would reach a case like `B(Y,Y)` which, if `Y` is sorted, is stated to be true.

This is similar in essence to bubble sort.

# Arithmetic

## 5.1

`A = 1+2` will unify the variable `A` with the compound expression `1+2` without evaluating the latter.

`A is 1+2` will first evaluate the `1+2` term to `3`, and then unify `A` with `3`.

## 5.2

Last Call Optimisation occurs when the recursive checking of a compound term is the last term in the clause.

This results in a reduction of required stack usage, as once all of the other terms have been resolved and the value of their variables propagated to the recursive term, they can all be discarded, leaving only the recursive term.

As such, the required stack usage is $\mathcal{O}(1)$ in the number of recursive checks, rather than $\mathcal{O}(n)$.

## 5.4

```
s(z).
s(s(A)) :- s(A).

prim(0,z).
prim(A,s(B)) :- prim(X,B), A is X+1. % I don't know how to avoid that "is"

plus(A,z,A).
plus(z,A,A).
plus(A,s(B),s(C)) :- plus(A,B,C).
plus(s(A),B,s(C)) :- plus(A,B,C).

mult(_,z,z).
mult(z,_,z).
mult(A,s(B),C) :- mult(A,B,X), plus(X,A,C).
mult(s(A),B,C) :- mult(A,B,X), plus(X,B,C).
```

## 5.5

```
// TODO
```

## 5.6

```
// TODO
```

## 7.8

1. `false`. `1+1` is a composite term but `2` is a number.
2. `false`. `1+1` is a composite term but `2` is a number with no arithmetic to perform on it.
3. `true`. `1+1` is a composite term and so is not equal to the number `2`.

4. `true`. The variable `Two` is unified to the value `One + One`.
5. `true`. The variable `One` is unified to the value `s(z)` and the variable `Two` is unified to the value `s(s(z))`.

### 5.8

For example,

```
?- prim(3,Three), prim(2,Two), prim(A,A_), plus(A_,Two,Three).
Three = s(s(s(z))),
Two = s(s(z)),
A = 1,
A_ = s(z)
```

The difference is that the `is` operator does not simply perform unification over the arithmetic operators in order to find a solution, but instead mechanically evaluates the right-hand-side of the expression to a number.

By implementing `plus/3` and `mult/3` as composite terms, unification can be used to perform algebra with them.

### 5.9

```
% No LCO
sum([],0).
sum([A|B],X) :- sum(B,Y), X is Y + A.


% LCO
sum2([],Acc,Acc).
sum2([A|B],Acc,X) :- Y is Acc + A, sum2(B,Y,X).
sum2(X, Y) :- sum2(X, 0, Y).



biglist(1,[1]).
biglist(N,[1|T]) :- M is N-1, biglist(M, T).

% Fails --- out of stack
?-  biglist(10000000, A), sum(A, B), print(B).

% Succeeds
?-  biglist(10000000, A), sum2(A, B), print(B).
```

## Backtracking

### 6.1

A list of that length is returned. If another solution is requested, the stack overflows.

### 6.2

All possible such lists are returned.

### 6.3

All possible pairs of list which, when appended, result in the given list are returned.

## Generate and Test

```
// TODO
```