**2.3.c. No.** There’s a cyclic definition for Adjs – which may result in an infinitely long sequence of adjectives, derived from adj. Therefore, we may end up trying sentences that have infinite adjectives before a noun, and may never reach a sentence that wasn’t produced before that infinite sequence. The main obstacle is again, the fact that Adjs may be derived into ‘Adj Adjs’, which creates a possible infinite cycle. If we wish to solve this issue, we may add a rule that allows only one appearance of a specific adjective before the same noun, effectively bounding the amount of possible adjective sequences before the same noun. We may fix the implementation to support such rule by checking if the adjective is part of the list of adjectives we generated, and to make sure we cut the proof tree to not end up in an infinite sequence anyways.

**3.1.** We can disprove it with a simple example: .

It’s easy to see that while . Therefore, is not always equal to .

**3.2.**

* A variable X occurring in A such that s(X) is defined and s’(X) is not defined:
  + **(A ◦ s) ◦ s’:**

X occurring in A and s(X) is defined so in (A ◦ s) 🡺 X=s(X).  
s’(X) is not defined so in (A ◦ s) ◦ s’ we get that X=s(X).

* + **A ◦ (s ◦ s’)**:  
    s(X) is defined and s’(X) is not defined so in (s ◦ s’) 🡺 X=s(X). X is defined in A so by applying the substitutions, in A ◦ (s ◦ s’) we get that X=s(X).
* A variable X occurring in A such that s(X) is not defined and s’(X) is defined:
  + **(A ◦ s) ◦ s’:**X occurring in A and s(X) is not defined so in (A ◦ s) 🡺 X=A(X). s’(X) is defined so in (A ◦ s) ◦ s’ we get that X=s’(X).
  + **A ◦ (s ◦ s’):**s(X) is **not** defined and s’(X) is defined so in (s ◦ s’) 🡺X=s’(X). X is defined in A so by applying the substitutions A ◦ (s ◦ s’) we get that X=s’(X).
* A variable X occurring in A such that both s(X) and s’(X) are defined:
  + **(A ◦ s) ◦ s’:**

X occurring in A and s(X) is defined so in (A ◦ s) 🡺 X=s(X).  
s’(X) is defined but X is already defined X=s(X) so the definition of X in s’ will be dropped as a result of the substitution, leaving us with X=s(X) in the result substitution.

* + **A ◦ (s ◦ s’):**  
    s(X) is defined and s’(X) is defined so in (s ◦ s’) 🡺 X=s(X) (the definition of X under s’ will be dropped as a result of applying the substitution). X is defined in A so by applying the substitutions A ◦ (s ◦ s’) we get that X=s(X).

**3.3.**

unify(A,B) =

let help(s, A’, B’) =

if then s

else let D = disagreement-set(, )

in if D = {X, t}

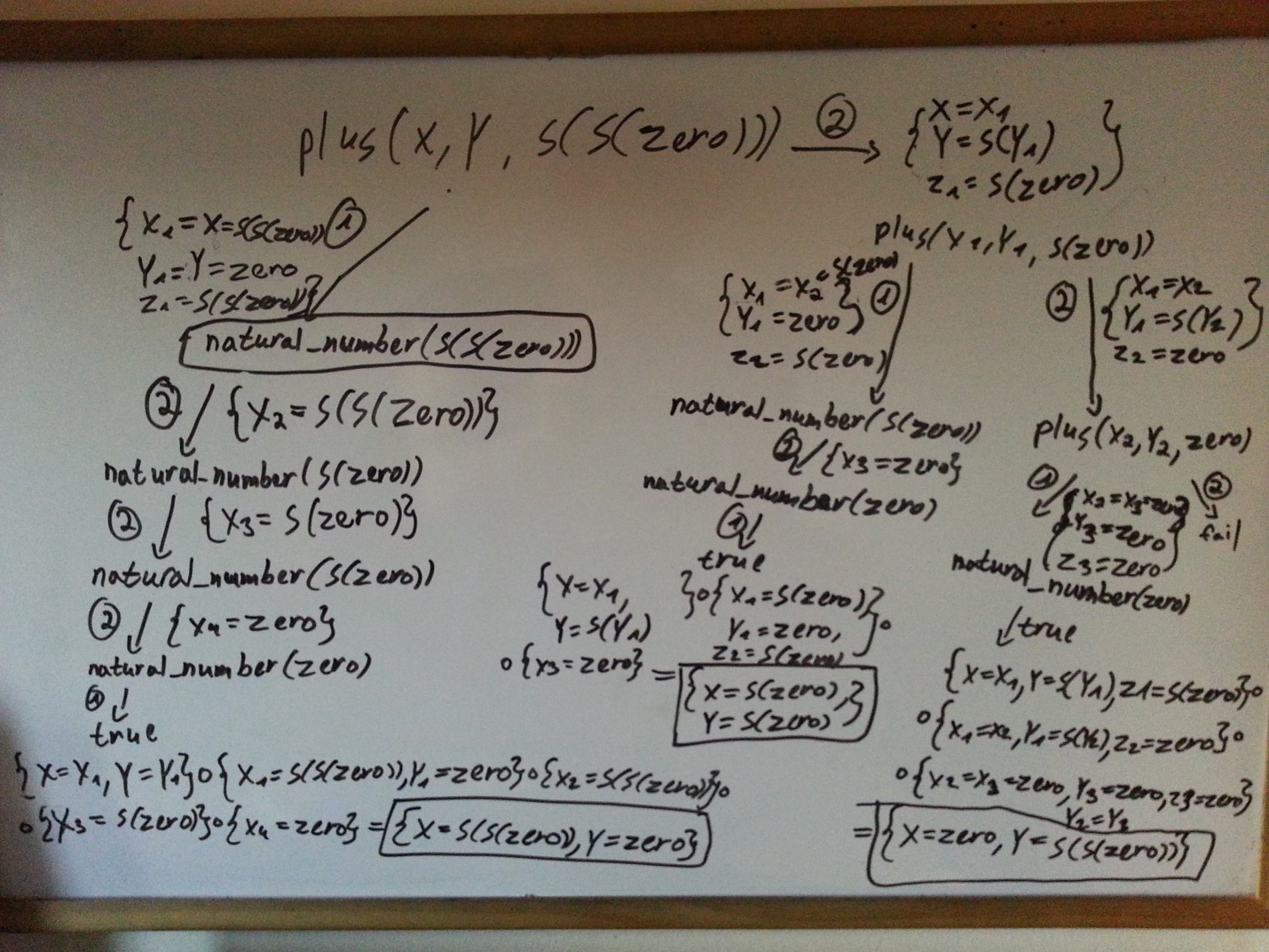
then help()

else FAIL

end

in help({}, A, B)

end

**4.1.** 

**4.2.**

% Signature: natural\_number(N)/1

% Purpose: N is a natural number.

rule (natural\_number(zero), []). %1

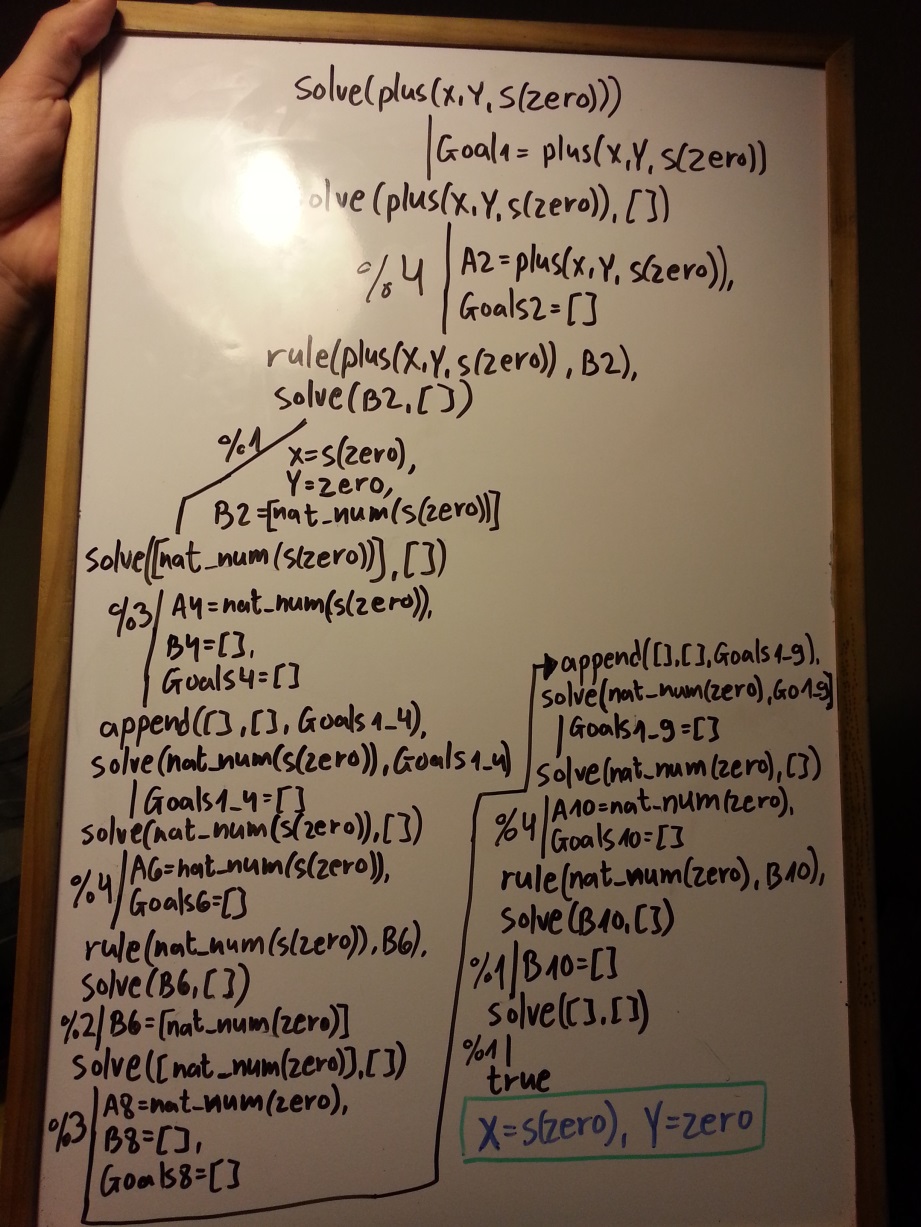
rule (natural\_number(s(X)), [natural\_number(X)]). %2

% Signature: plus(X, Y, Z)/3

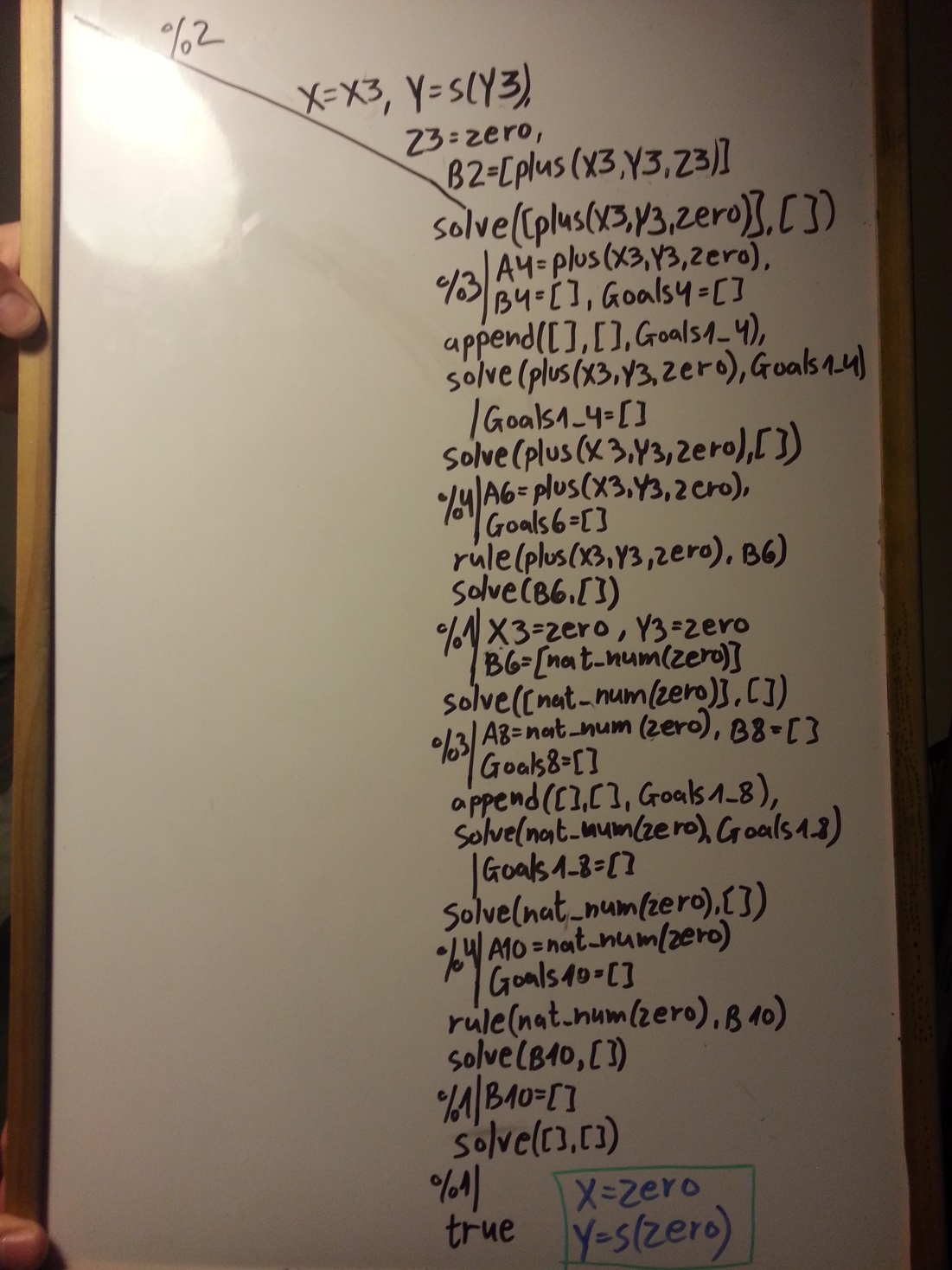
% Purpose: Z is the sum of X and Y.

rule (plus(X, zero, X), [natural\_number(X)]). %1

rule (plus(X, s(Y), s(Z)), [plus(X, Y, Z)]). %2



**Second branch of question 4.2:**



**Question 4.2 proof-tree results:**

**X=s(zero), Y=zero;**

**X=zero, Y=s(zero).**

**4.3.** Instead of going over the goals by order , after the change we will advance on it in a queue like method – process **one** rule from every goal, and push the rest of the rules to the end of the queue, and continue until we are done. If we imagine a graph tree, where previously we would advance to the leaves first (sort of like DFS), now we advance by tree levels (sort of like BFS).

**5.1.** They differ in the fact that ML forbids two of the same parameter, while in the unify algorithm it is allowed. Also, in logic programming we may provide any kind of values to the same parameter, while in ML we are restricted to only one type of data per parameter.

They are similar in the way of handling list parameters, allowing to get the first element and the rest of the list from the pattern, and specifying end cases is also similar.

**5.2.** Predicate symbols represent a predicate that is defined somewhere in our program, and will be evaluated to true or false somewhere down the road. They may be named with either lowercase first letter or uppercase. Constant symbols are what we use to represent actual data, and must be named with the first letter as a lowercase letter. They may be regarded as simply strings representing their value. Functor symbols are a way to tag data, and may be composed of several functors. Functors allow us to differentiate values that otherwise would be regarded as equal, and virtually allow us to create complex data types within logic programming.

**5.3.** **In version 1,** the interpreter receives an expression, which may be constructed with functors, and then tries to evaluate it using simply the Prolog interpreter. Predicates are given as functors and when passed to the Prolog interpreter they will be regarded as predicates, as needed.

**In version 3,** during the pre-processing stage, all predicates are transformed into functors and wrapped in a rule predicate. This allows us to construct a program which is based on facts only.

**5.4. No.** Calculating the transitive closure of a binary relation may result in an infinite tree path, as it’s very much like the parent/ancestor relation, which we saw already that has infinite tree path.

**5.5.** Since the answer-query algorithm iterates on the proof tree in a DFS like style, proving the new query it constructs at every step, we may change it to operate in a BFS like style. Now we are iterating on the proof tree by levels, and since the length from the root to any answer in the tree is finite, we will reach it at some point. We will stop our execution anyways after trying levels in the proof tree, as after that point we are guaranteed to repeat the same query in one of the tree paths to a leaf, therefore resulting in an infinite loop.

**5.6.** They are all similar in the way they provide us with a way to tag data, which allows us to differentiate between similar types of data. For example, if we decide to create a new type of numbers but still represent it with normal numbers, we won’t be able to differentiate between a normal number and our newly created number type number. So we tag it with a list of symbols in Scheme, value constructor in ML and a functor in FLP. They all allow ‘tagging’ any type of data. The differences between them reside on the fact that they are not native in Scheme, while they are in ML and FLP, and in Scheme we may pull the value of them easily while in ML and FLP we have to use the pattern matching/unify way.