Here is a detailed explanation of the circular dependency issue, written in English, which you can share with your users.

Understanding the Calculation Behavior: A Circular Dependency

What you've observed is a classic and fundamental behavior in calculation software like SMath Studio or Mathcad, known as a **circular dependency**.

The important thing to understand is that **your `find()` function is working perfectly correctly**. The issue is not with the solver itself, but with how the SMath Studio calculation engine manages dependencies when you use the same variable names for both initial guesses and final results.

Let's walk through the "domino effect" of SMath's calculation engine step-by-step.

The Initial State

You start with this setup:

```
// Initial guesses (low priority)
h ≈ 1 'cm
b ≈ 1 'cm

// The solver call
S : find(sys(h·b³... ≡ Ix, ...), sys(h, b, ...))
```

Everything works fine here. SMath uses the "guess values" (≈) for `h` and `b` to start the calculation. The `find()` function runs successfully, and the result vector is assigned to the variable `S`.

The First Re-assignment: `h : el(S, 1)`

When you ask SMath to compute this line, the following happens:

1. SMath evaluates the right side, `el(S, 1)`, which correctly returns the solved value, e.g., `3 'cm`.
2. It then performs the assignment `h := 3 'cm`.
3. **This is the crucial moment.** The variable `h` has now changed from a low-priority "guess" (≈) to a high-priority "hard definition" (`:=`).
4. The SMath engine immediately detects this change and triggers a recalculation of **every expression that depends on `h`**.
5. Which expression depends on `h`? The `S : find(...)` line!
6. SMath automatically re-runs the `find()` function in the background. At this point, everything is still okay, and the `S` variable is updated.

The Failure: `b : el(S, 2)`

This is where the unbreakable loop occurs, often described as a "chicken and egg" problem.

1. You ask SMath to compute `b : el(S, 2)`.
2. To do this, it must first evaluate the right side, `el(S, 2)`.
3. To get the value of `S`, it must re-evaluate the `find()` function.
4. To evaluate `find()`, it needs the values of its inputs, `h` and `b`.

5. Here is the paradox:
    - To define b, SMath needs the value of S.
    - To calculate S, SMath needs the value of b.
    - However, the variable b is currently **undefined**. Its old "guess" value is no longer valid because SMath is in the middle of giving b a new, hard definition.

The calculation engine gets stuck in a logical loop it cannot resolve. The error message you see (e.g., `"h is not defined"`) is a symptom of this dependency chain breaking down during the failed recalculation attempt.

The Solution: The Golden Rule of Solvers

To avoid this, you must **break the cycle** by using different variable names for your inputs (initial guesses) and your outputs (the final solutions).

The Golden Rule: Never use the same variable names for initial guesses and for the final results.

Here is the correct and robust way to structure your worksheet:

```
// 1. INPUTS: Use unique names for initial guesses
h_guess ≈ 1 'cm
b_guess ≈ 1 'cm

// 2. PROCESS: Call the solver using the guess variables
// Note that the 'find' block still solves for the symbolic 'h' and 'b'
Solution : find(sys(h·b³/3 ≡ Ix, h³·b/3 ≡ Iy, h ≈ h_guess, b ≈ b_guess), sys(h, b))

// 3. OUTPUTS: Assign the results to new, unique variables
h_sol : el(Solution, 1)
b_sol : el(Solution, 2)
```

By doing this, you create a clear, one-way flow of calculations: `h_guess` → `Solution` → `h_sol`

There is no loop, and the SMath engine can correctly calculate the entire sheet without any ambiguity.