DAKSH

Error & Debugging Guide

Phase 0 — Web Interpreter Edition

Document 04 of the Daksh Documentation Set

A practical guide to understanding, locating, and fixing errors in Daksh programs.

Scope: This guide documents the error categories and behaviors established in the Phase 0 web interpreter. It does not invent unfinished error syntax.

1. What Is an Error?

An error means the Daksh program cannot be parsed, executed, or completed correctly. The current web interpreter reports errors in the output area and marks the run as an error.

The practical debugging process is:

```
1. Read the error message.
2. Identify whether it is a parser/syntax or runtime problem.
3. Find the statement responsible.
4. Check the value, type, index, name, or module involved.
5. Fix one problem at a time.
6. Run the program again.
```

The most useful debugging habit is to reduce a failing program to the smallest example that still produces the error.

2. Syntax Errors

A syntax error occurs when the parser cannot understand the structure of the Daksh source code.

Common causes include missing braces, incomplete statements, invalid expressions, or an incorrect keyword/construct.

Example — incomplete conditional:

```
if true {
    tell "Hello"
```

The closing **}** is missing. The parser cannot complete the block.

Correct version:

```
if true {
    tell "Hello"
}
```

Debugging rule: when a parser error appears, first check the surrounding brackets, keywords, operators, and statement structure before looking for runtime problems.

3. Undefined Variable

An undefined-variable error occurs when code tries to read a variable that does not exist in the current visible scope.

```
tell unknownValue
```

Fix by declaring the variable before using it:

```
set unknownValue = 100
tell unknownValue
```

Also check spelling and capitalization. Daksh variable names are case-sensitive.

4. Constant Modification

A value declared with **const** cannot be reassigned.

```
const pi = 3.14

tell pi

set pi = 3.14159
```

The interpreter reports a runtime error because the program attempts to modify a constant.

Correct approach:

```
set pi = 3.14
```

```
set pi = 3.14159
```

Use **set** when a value is expected to change.

5. Division by Zero

Division by zero is treated as a runtime error in the Daksh language/runtime behavior.

```
set x = 10
tell x / 0
```

Check the divisor before performing a calculation when its value may become zero.

```
set divisor = 5

if divisor != 0 {
    tell 100 / divisor
} else {
    tell "Cannot divide by zero"
}
```

6. Invalid List Index

List indexes are zero-based. An index must refer to an existing element.

```
set numbers = [10, 20, 30]

tell numbers[0]
tell numbers[2]
tell numbers[10]
```

The third access is invalid because the list contains indexes 0, 1, and 2 only.

Valid version:

```
set numbers = [10, 20, 30]

tell numbers[0]
tell numbers[1]
tell numbers[2]
```

For assignments, the same validation applies:

```
set numbers = [10, 20, 30]
set numbers[1] = 99
```

7. Invalid Assignment Target

An assignment must target a valid variable, object property, or list/object index.

Valid examples:

```
set x = 10
set user.name = "Daksh"
set numbers[1] = 99
```

When a nested assignment fails, inspect the object/list value first. A null or missing intermediate value cannot be safely assigned through.

8. Null Nested Assignment

A nested property/index assignment requires the intermediate value to exist.

```
set data = null

set data.name = "Daksh"
```

The runtime reports that a property cannot be assigned through a null value.

Correct approach:

```
set data = {
    name: "Daksh"
}

set data.name = "Manideep"
tell data.name
```

9. Import Errors

An import error occurs when the requested module is not available to the current runtime.

```
import unknownModule
```

Use one of the currently established standard-library modules:

```
import math
import string
import list
import random
import file
import time
import system
```

Do not assume a future or unimplemented module exists just because a name sounds reasonable.

10. Type and Value Problems

Daksh Phase 0 uses dynamic typing. Values can change type through reassignment.

```
set value = 10
set value = "Daksh"
set value = true
set value = [1, 2, 3]
```

When debugging a value-related problem, inspect what value the variable currently contains and, when useful, use **type(value)**.

```
set value = 10

tell value
tell type(value)
```

Exact type-coercion rules are not finalized in the Phase 0 language design. Do not rely on implicit conversions when a deliberate conversion such as **integer()** or **text()** is available.

11. Input Conversion Problems

User input is text and may need conversion before numeric calculations.

```
set age = input("Enter age: ")

tell age + 5
```

For numeric input, explicitly convert it:

```
set age = integer(
    input("Enter age: ")
)

tell age + 5
```

Explicit conversion makes the program's intention clear.

12. Function Errors

When a function does not behave as expected, inspect its parameters, local variables, return paths, and the values passed to it.

Example:

```
function divide(a, b) {
    return a / b
}

tell divide(10, 0)
```

The function itself is syntactically valid, but the call produces a division-by-zero runtime problem.

Debug by checking the arguments at the call site and the operations inside the function.

13. Recursion Problems

Recursive functions need a condition that eventually stops recursion.

```
function factorial(n) {
    if n <= 1 {
        return 1
    }

    return n * factorial(n - 1)
}

tell factorial(5)
```

The base case **n <= 1** prevents the recursion from continuing forever.

When debugging recursion, verify three things: base case, progress toward the base case, and the returned value.

14. Scope Errors

A variable declared inside a nested block is not available outside that block.

```
set name = "Manideep"

if true {
    set message = "Hello"
    tell message
}

tell name
tell message
```

The final **tell message** is invalid because **message** belongs to the inner scope.

Fix by declaring the value in an outer scope if it needs to survive outside the block.

```
set message = ""

if true {
    set message = "Hello"
}

tell message
```

15. Module/API Debugging

If a module imports successfully but a function call fails, check the module name, function name, arguments, and expected value type.

```
import math

tell math.sqrt(144)
```

Then compare your call against the Standard Library Reference. Avoid guessing API names.

16. File and Browser Errors

The Daksh web interpreter runs inside a browser. Browser security prevents unrestricted access to arbitrary local filesystem paths.

Use the supported browser-safe file operations:

```
import file

set content = file.read()
tell content

file.write("Hello from Daksh!")
file.download("hello.txt", "Hello from Daksh!")
```

The current **file.exists()** operation intentionally cannot behave like unrestricted native filesystem inspection.

17. Time/System API Availability

Some system values depend on browser/device support. For example, battery information may not be available in every environment.

```
import system

tell system.isOnline()
tell system.batteryLevel()
tell system.memory()
```

A browser runtime should be treated as an environment with capabilities and restrictions, not as unrestricted native OS access.

18. A Systematic Debugging Method

```
Step 1  Read the complete error message.
Step 2  Identify parser vs runtime failure.
Step 3  Find the smallest failing statement.
Step 4  Inspect names and spelling.
Step 5  Inspect values and types.
Step 6  Check indexes and object paths.
Step 7  Check function arguments.
Step 8  Check imports and API names.
Step 9  Fix one issue.
Step 10 Run the program again.
```

Do not change ten unrelated lines at once. That makes it harder to know which change fixed or introduced the problem.

19. Debugging With Small Tests

If a large program fails, isolate the feature.

Instead of debugging a complete application immediately, test the smallest behavior first:

```
set numbers = [10, 20, 30]
tell numbers[1]
```

Then test mutation:

```
set numbers = [10, 20, 30]
```

```
set numbers[1] = 99
tell numbers[1]
```

Then add the surrounding function/loop/module logic. This creates a clear progression from isolated feature to integrated program.

20. Debugging a Full Program

When a combined program fails, break it into layers:

```
Layer 1: Check imports.
Layer 2: Check input/data creation.
Layer 3: Check function calls.
Layer 4: Check loops.
Layer 5: Check conditions.
Layer 6: Check returned object/list.
Layer 7: Check final output.
Layer 8: Check mutations.
```

Example debugging order for a student-processing program:

```
import string
import list

set marks = [85, 90, 95]
tell list.length(marks)

set name = string.upper("manideep")
tell name
```

Once each dependency works independently, combine them into the full program.

21. Common Mistakes Checklist

```
[ ] Did I spell the variable name correctly?
[ ] Did I use the correct capitalization?
[ ] Did I close every { } block?
[ ] Did I close every [ ] list/index?
[ ] Did I use the correct operator?
[ ] Is the variable declared before use?
[ ] Am I modifying a const?
[ ] Is a list index inside the valid range?
[ ] Is an intermediate nested object/list actually present?
[ ] Could a divisor be zero?
[ ] Did I import the required module?
[ ] Does the module function name exist?
[ ] Are function arguments correct?
[ ] Does recursion have a base case?
[ ] Is the value in the scope where I am using it?
[ ] Does this operation depend on browser capabilities?
```

22. Error Categories — Quick Reference

```
Syntax Error
  Parser cannot understand the source structure.

Undefined Variable
  A variable is not available in the current scope.

Division by Zero
  A numeric division uses zero as divisor.

Constant Modification
  A const value is reassigned.

Invalid List Index
  A list index is invalid or outside the allowed range.
```

```
Invalid Assignment Target
  Assignment does not point to a valid target.

Null Nested Assignment
  A property/index is assigned through null.

Import Error
  Requested module is not available.

Runtime Error
  Program structure is valid but execution cannot continue.

Browser Capability Limitation
  A requested system/file operation is restricted by the browser.
```

23. Phase 0 Debugging Boundary

Phase 0 establishes core error reporting and runtime validation. Custom user-defined error syntax is not finalized. Native compiler/runtime diagnostics, richer stack traces, source maps, and advanced debugger tooling are later-stage work.

Do not treat an unfinished future feature as a Phase 0 syntax rule.