

# StepRepl Guide

## Table of Contents

---

Introduction .....	2
Installation.....	2
Using Visual Studio Code .....	2
Starting StepRepl .....	3
The command box .....	3
Selecting a project .....	3
Running code.....	4
Running code from the command box .....	4
Command output .....	4
Stopping or pausing a running program .....	4
Editing and reloading code .....	4
Editor support .....	4
Warnings.....	5
Singleton variables .....	5
Uncalled tasks.....	5
Programmatic control of the UI.....	5
Making menus .....	5
Debugging tools .....	6
Stack traces .....	6
Reading a stack trace .....	6
Breakpointing and single-stepping.....	7
Call graph visualization.....	8
Visualizing your own graphs .....	9
.....	9
Logging.....	9

## Introduction

---

*StepRepl* is an interactive execution environment for *Step* code. It is a read-eval-print-loop (REPL), meaning you type commands, it runs them, and prints the results. It supports standard debugging tools such as breakpointing and single-stepping, as well as providing tools for profiling, sampling, and burn-in testing. This guide describes its capabilities and user-interface.

## Installation

---

To install *StepRepl*, download the appropriate binary for your operating system from the current github release.

If you are running on Windows, it will look like a folder, and you will run the `StepRepl.exe` file inside the folder. In either case, you can place them wherever you like; they do not need to be in any special location.

If you are running on **MacOS**, the binary will look like an application, and you can just run it. You may need to run the terminal command:

```
xattr -dr com.apple.quarantine path
```


To enable execution of it. Moreover, when you run it, it will likely ask you whether to allow it access to your documents. If you do not allow access, then MacOS will silently fail file operations and *Step* will be unable to access your projects.

## Using Visual Studio Code

Although it does not support full source-level debugging, *Step* has some level of integration with Visual Studio Code:

- Syntax highlighting for `.step` files
- Clicking on a warning or stack frame in *StepRepl* will bring you to the correct line and file in your code.

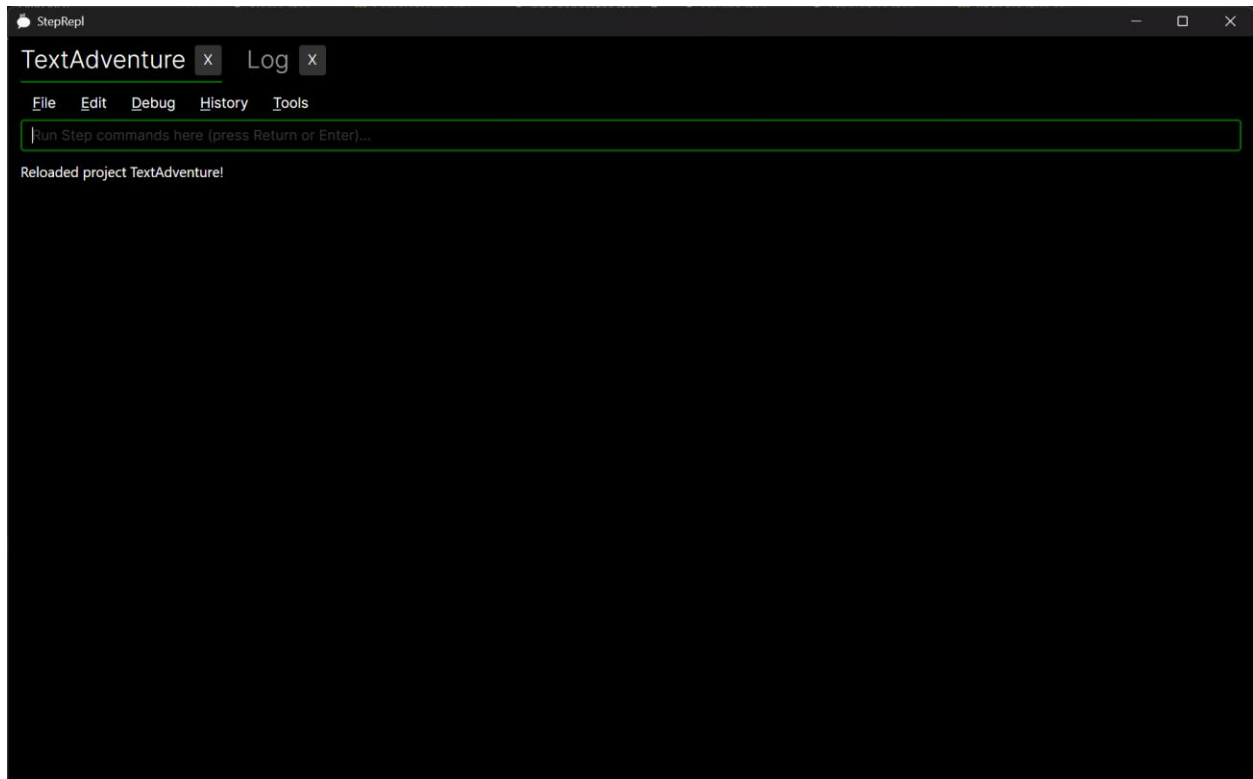
To take advantage of this, install visual studio code (do an internet search for “download visual studio code”), then open VS Code and perform the following steps:

- Click the  icon on left this brings you to the “marketplace” for extensions
- Type “step syntax highlighting” in the search box
- Select it and click Install.
- If you are on MacOS you must also do the following in VS Code:
  - Type Command-Shift-P. This opens the command palette
  - Type “Shell” in the command palette
  - Select “Shell command: Install Code in PATH”

You should only have to do this once, although some Mac Users find they must run the command palette command every time they update VS Code.

## Starting StepRepl

To launch StepRepl simply open the application (MacOS) or run `StepRepl.exe` (Windows). You can exit either by clicking the close box. You should see something like this:



Instead of “TextAdventure”, you will see the name of the project (folder) with your code, or just “Runner” if you have no project selected. The Log tab is used for debugging.

The green text box at the top is in the **command box**, where you can type code to run. Beneath it, you may see a **button bar** containing buttons to run code defined by the project you are running. Then at the bottom you will find:

- Any text output by the command you type
- Any warnings about your code, in yellow
- A stack trace of your code if it throws an exception, in orange

### The command box

The command box is the green text box at the top of the screen. You can type any command here and it will execute it.

### Selecting a project

You cannot use StepRepl to enter new tasks and methods, only to call existing tasks and methods. These are loaded from your current project folder, which you can select using the **File>Open project ...** option in the menu. Select the folder containing your code, and Step will load all `.step` and `.csv` files within that folder and its subfolders. If you modify the code, you can reload it by selecting **File>Reload project** or typing **control-R**.

## Running code

---

StepRepl is primarily an application to let you load and run Step code. It provides a number of ways to call tasks, and basic tools for debugging them.

### Running code from the command box

The most versatile way to run code is to type it in the green command box. You cannot type new methods here, but you can call any tasks defined in your project or built into *Step*. When typing a call to execute, the square brackets around it are optional. However when running multiple calls in one command, they must each be enclosed in brackets. Thus:

- `Story`  
Runs the `Story` task with no arguments, assuming you've written such a task.
- `[Story]`  
Does the same thing.
- `Write "Hi there"`  
Runs the `Write` task with the argument "Hi there."
- `[Write "Hi there"]`  
Does the same thing.
- `[Story] [Paragraph] The end.`  
Calls `Story`, then starts a new paragraph and prints "The end."

### Command output

When you run a task, the system displays its output in the left column. If the call you performed contained local variables (local variables begin with `?` characters), the system will also print the final values of those variables, in the form: *variable* = *value*. If the variable ended with a value (a number, string, tuple, etc.), then that's what will be displayed for *value*. If it either had no value or was unified with some other variable, then *value* will be that variable.

*Step* prints variable names with serial numbers appended so you can distinguish between different variables with the same name. Thus a variable named `?x` in the source might display as `?x3178`.

If the code throws an uncaught exception, then the program will stop and StepRepl will show the exception followed by a Stack trace in the right column.

### Stopping or pausing a running program

You can kill the running Step program by selecting **Debug>Abort** in the menu.

## Editing and reloading code

---

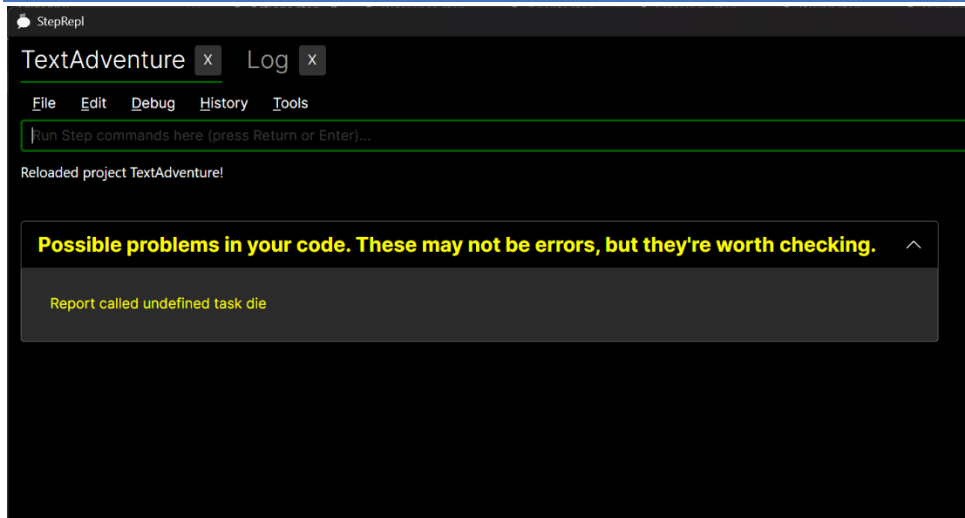
StepRepl does not include a code editor. So if you need to change the code, edit it externally, then save the files and type **control-R** (or type **reload** in the command box). If there are errors or warnings, they will appear in the right (debug) column.

### Editor support

The preferred editing environment for Step is [Visual Studio Code](#). See the section on installation for instructions for installing VS Code and configuring it for Step.

## Warnings

---



After loading your code, *Step* runs a simple static analysis on it and prints warnings related to common errors. If you have installed and configured VS Code, then clicking on the message will bring you to the relevant line of your code when applicable.

### Singleton variables

Since local variables aren't explicitly declared in *Step*, a typo in a variable name will generally cause *Step* to treat it as a valid, but completely different variable. The calling card of such mistakes is a variable that is used only once, and so *Step* reports warnings for these so-called singleton variables.

As with other logic programming languages, *Step* uses a naming convention to override the warning. To override the warning, either change the name of the variable to either `?_` or place an underscore after the `?` of the existing name, that is rename `?unused` to `?_unused`.

### Uncalled tasks

Tasks that are defined but never called occur when:

- The task is an **entry point** to be called from outside the step code (e.g. by the user in *StepRepl* or by the game engine)
- A **typo** in the task name in the method in any calls to it
- The task is **dead code** that has become obsolete

The first of these is not an error, but the other two are. So *Step* issues a warning for uncalled tasks. You can override it by annotating the task with `[main]` in the source code.

## Programmatic control of the UI

---

*StepRepl* allows you to add menu items to the UI declaratively by including methods for the `MenuItem` predicate.

### Making menus

To add a menu item, add a declaration of the form:

`MenuItem "menu" "item" code.`

This will define Menu>Item in the menu bar and configure it to run *code* when it is selected. For example, the declaration:

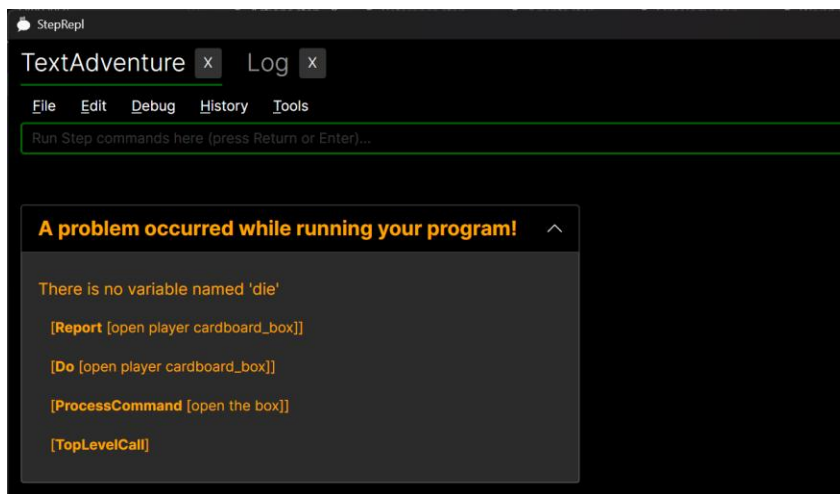
```
MenuItem "Story generator" "Make a story" [Story].
```

will add **Story generator>Make a story** to the menu bar. When the user selects it, it will run [Story].

## Debugging tools

*StepRepl* provides the standard kinds of debugging tools you would expect in a development environment.

### Stack traces



When a program pauses fails, the system displays the currently running task, followed by the task that called it, its caller, and so on. This is known as a stack trace. Each task is shown with its arguments. Again, if you have installed and configured VS Code, then clicking on one of the tasks will bring you to the relevant line of code.

### Reading a stack trace

Each task is printed together with its arguments. Arguments are often local variables. As mentioned previously, when local variables are printed in the trace, a serial number is appended so that you can tell whether two different variables named ?x are the same ?x or different ones. If an argument is a local variable that has been unified with a value, that value is printed instead of the name of the variable. If you call [MyTask ?x], it will initially display in a trace as something like:

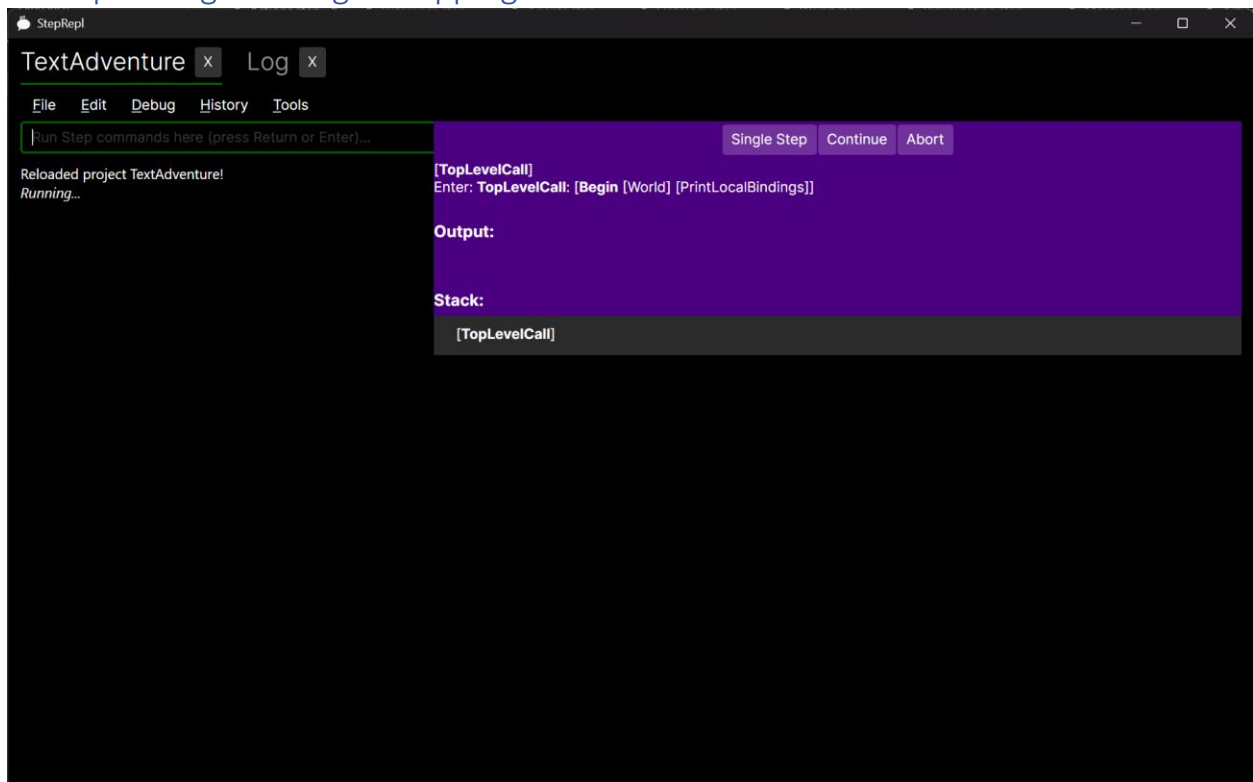
```
MyTask ?x259
```

Then, if ?x is unified with some other value, its appearance in the stack can change. If it is unified with another variable that doesn't have a value, then the name of either variable can be displayed in the trace. If ?x is unified with a constant, or if some other variable unified with ?x is unified with a constant, then that constant will appear in the stack trace rather than the name ?x. Thus, the call my go from appearing in the trace as shown above to simply saying:

```
MyTask 7
```

If ?x had been unified with 7.

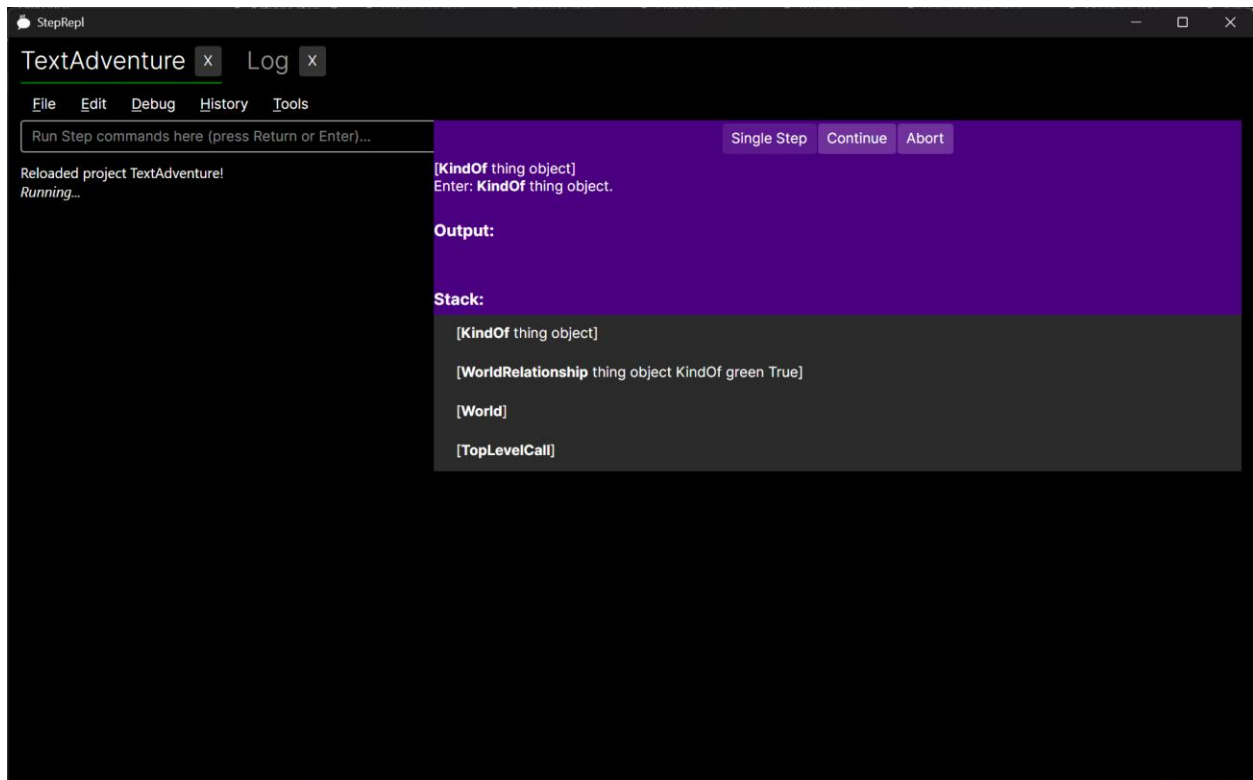
## Breakpointing and single-stepping



Although *StepRepl* doesn't support source-level debugging, it does support the standard single-stepping operations found in most debuggers.

To run a command in debug mode, simply type it in the command box and then type **control-return** rather than just return. It will pop up a debugger pane in purple showing the current stack and buttons for single-stepping, continuing (running to end or a breakpoint), and aborting execution:

Pressing single-step will run to the next user-defined task and pause again updating the stack:

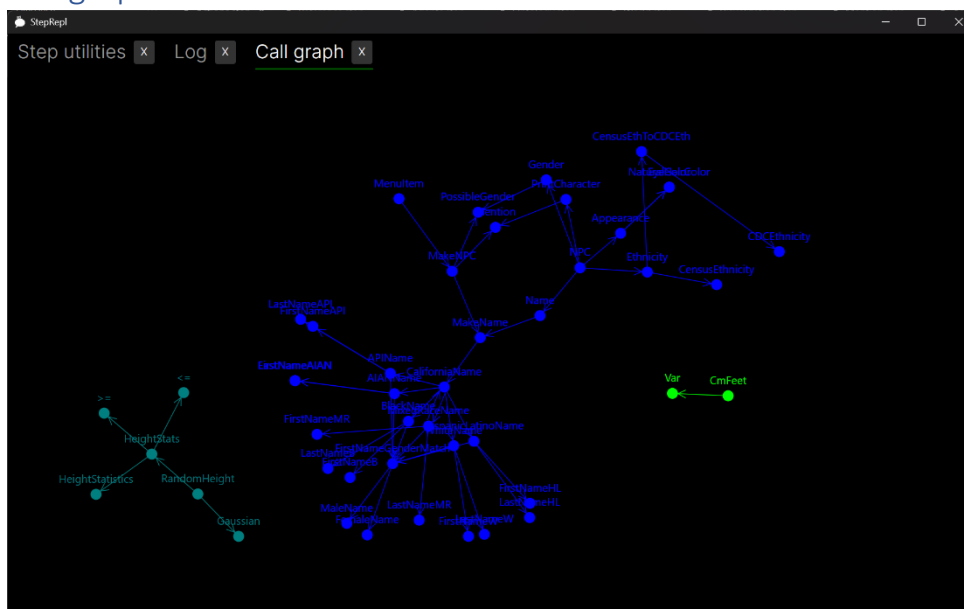


Breakpoints can be specified by adding calls to the `Break` primitive task:

- [Break]  
Pause execution at this point in the code. Display the stack trace and allow the programmer to either single-step or continue execution.

And reloading the project.

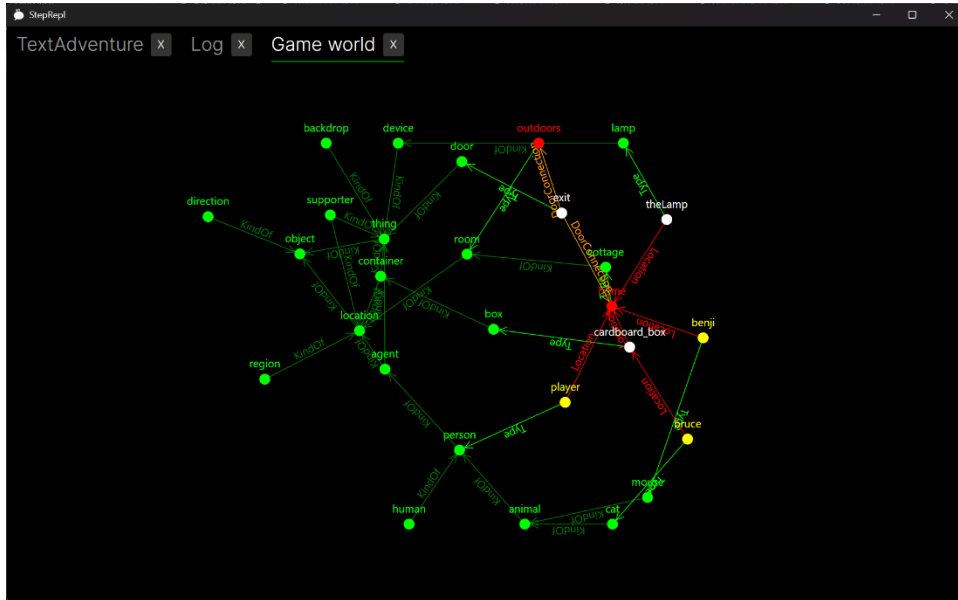
## Call graph visualization



StepRepl also includes a graph visualizer. Use the **Tools>Graph view** menu option to show the graph of which tasks call which other tasks. The visualizer will color-code different connected components, making it easy to find tasks that nobody calls (e.g. because of a type-o in a method name).



## Visualizing your own graphs



You can also call the graph visualizer from Step code to make your own visualizations. See the documentation in the **Step Task Reference**.

## Logging

You can also display text to the Log tab. Unlike normal output, log entries are not removed when the system backtracks. So you can use the log to understand how your program is exploring its search space.

- `[Log args ...]`  
Prints args as an entry in the Log tab. This is done when the call is first executed.
- `[LogBack args ...]`  
Prints args as an entry in the Log tab when the call to LogBack is backtracked.

The two tasks each succeed exactly once per call. The difference between them is that `Log` prints when it is called, `LogBack` waits until the program tries to backtrack over the call to `LogBack`. If the program never backtracks over the call, `LogBack` never prints anything.