ARBEITSGRUPPE PROGRAMMIERSPRACHEN UND ÜBERSETZERKONSTRUKTION
INSTITUT FÜR INFORMATIK
CHRISTIAN-ALBRECHTS-UNIVERSITÄT ZU KIEL

USER MANUAL

Using the levels debugger in Atom

Thomas Ulrich

v1.0 - September 2016

1 User manual

This user manual is meant for anyone wishing to use the levels debugger within the Atom text editor. The goal is to get you started quickly and to introduce you to the features offered by the debugger. No prior knowledge about Atom or debugging in general are required, though programming experience will be helpful to write code that can be debugged.

To follow this guide, you will need the following:

- A computer running either Windows 7 or later, Ubuntu 16.04 LTS or later or OS X 10.10 Yosemite or later
- Access to the internet and a web browser to download software

The levels debugger was designed to be used with the levels framework and is therefore adaptable to different programming languages. For demonstration purposes, we will use Ruby as the programming language of choice. This requires the `levels` and `levels-language-ruby` Atom packages (more on that later) as well as a Ruby runtime. If you do not have a Ruby runtime installed on your computer, please install one before continuing with this guide. To get Ruby, visit `https://www.ruby-lang.org/en/downloads/`.

1.1 Installing Atom and a Java runtime environment

To get started, you'll first need to install the Atom text editor. It's free and doesn't require you to register to download. Visit `https://atom.io/` and select the installer for your platform (Windows, Ubuntu or Mac). Once downloaded, run the installer. A wizard will help you install the software. If you're unsure about what to select during the installation, keep the default settings which work well for our purposes. The wizard will install Atom and inform you upon completion at which point you'll have Atom installed on your computer.

Next, you will need a Java runtime environment (JRE) to run the debugger. You may already have one installed on your computer. If you do, make sure it supports Java 8. If you're unsure whether Java 8 is supported by the JRE currently installed on your computer, consider installing the most recent version. We're going to use Oracle's JRE for our purposes, however, any Java SE compliant JRE will work.

To download the Oracle JRE, visit `http://www.oracle.com/technetwork/java/javase/downloads/index.html`. Choose the JRE that is right for your platform (Windows, Ubuntu or Mac) and system type (32 bit or 64 bit). If you're unsure whether to use a 32 bit or 64 bit version, choose the 32 bit version. Proceed to download the JRE. Once downloaded, run the installer. Again, the default settings will work well and you do not need to customize

the installation. The wizard will inform you upon completion. To verify that Java was installed correctly, open a command prompt (Windows) or terminal (Ubuntu and Mac) and type `java -version`. You should see a message informing you about the Java version you just installed. With Atom and a JRE installed, you're now ready to install the debugger itself.

Note for Mac users: In some cases, running `java -version` might return a message informing you that a JDK (Java SE Development Kit) is required. In that case, you must download a JDK and install it before you can proceed. To download a JDK, go to the website mentioned above and choose to download the JDK instead of the JRE. Installing the JDK works the same way as installing the JRE. After installing the JDK, try to verify the Java installation again by opening a terminal and typing `java -version`.

1.2 Installing the debugger package in Atom

Open Atom and go to *File* → *Settings* (on Windows) or *Edit* → *Preferences* (on Ubuntu and Mac). The Atom settings will open, with a menu on the left. Choose *Install* to install a new package. We will choose the `levels-debugger-ruby` package. Type `levels-debugger-ruby` into the text box and click on „Packages“ to search for the package. Once found, you can install the package by clicking on „Install“ (see figure 1.1).

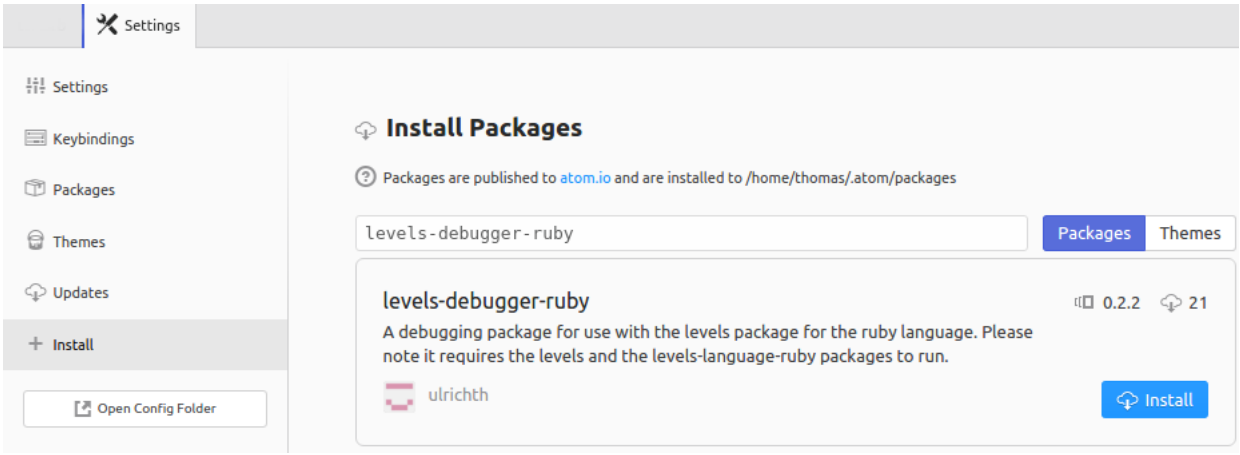

Figure 1.1: Installing the levels-debugger-ruby package

Please be patient while the package installs. If you do not have the `levels-language-ruby` and `levels` packages installed already, they will be downloaded and installed automatically. The `levels` package contains a *terminal* which can be used to input and output data. The `levels-language-ruby` package contains the Ruby language definition. Once the installation is complete, you can start a new text file and save it with the `.lrb` ending. Upon saving, the debugger controls will be visible to the right. The debugger is now installed and ready for use.

1.3 Using the debugger

The debugger controls are automatically displayed on the right once you start editing a `.lrb` file. On the top of the right pane you'll see a status message informing you about the current status of the debugger. Status messages are color coded in the following way:

- Red - The debugger cannot be used on this level or this language is not supported.
- Blue - The debugger is not running, but is available on this level.
- Yellow - The debugger is waiting for you to step to the next instruction.
- Green - The program being debugged is busy, e.g. because a long running calculation is in progress or because it is waiting for input.
- Purple - The debugger has exhausted all available replay information, thereby ending the replay.

Figure 1.2 shows the debugger controls during a debugging session. Below the status message (1) you'll find the control buttons (2), which will be covered in the following sections. Directly beneath the controls are the table of defined variables (3) and the call stack (4). You can resize the controls by placing the cursor on the left of the controls pane and dragging it left or right.

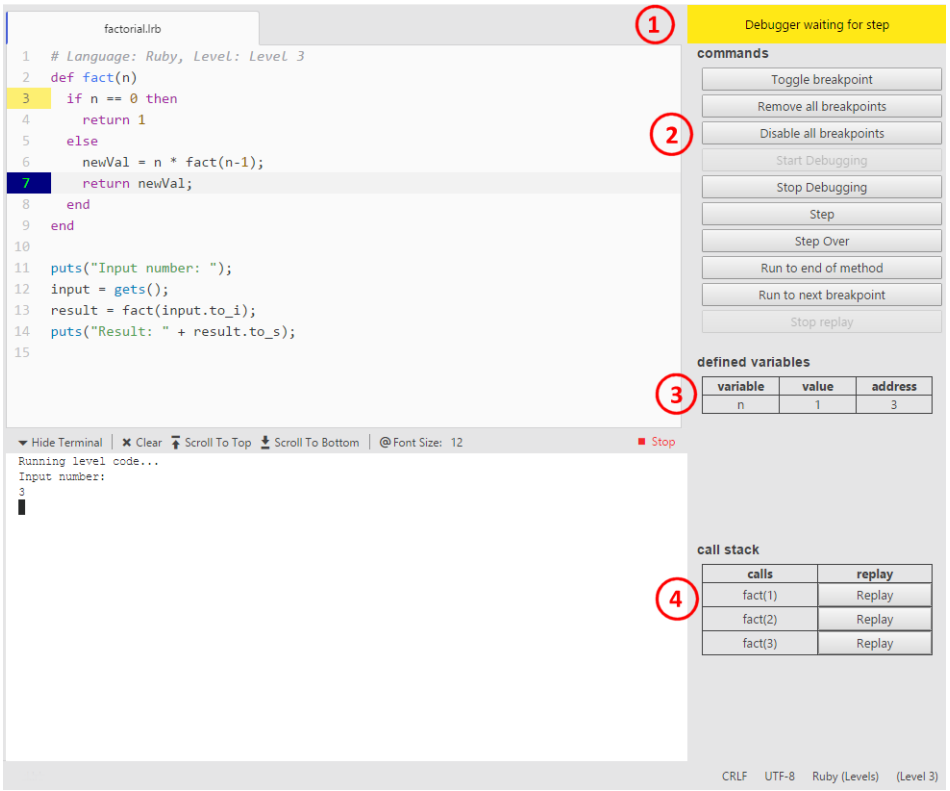

Figure 1.2: The debugger GUI elements shown during a debugging session

You can debug code written on levels 1 through 3 of the `levels-language-ruby` package. For testing purposes, you can use the program displayed in Listing 1.1, which recursively calculates the factorial of a number.

```
def fact(n)
  if n == 0 then
    return 1
  else
    newVal = n * fact(n-1);
    return newVal;
  end
end

puts("Input number: ");
input = gets();
result = fact(input.to_i);
puts("Result: " + result.to_s);
```

Listing 1.1 : Recursive factorial calculation in Ruby

1.3.1 Starting and stopping

To start the debugger, open a `.lrb` file, write a program and then click „Start debugging“. You may be prompted by your firewall about allowing incoming TCP connections on port 59599 and 59598. These ports are used by the debugger to communicate with the Atom GUI and the program to be debugged. These connections are used only locally, they do not require internet access. Allow these connections in order to use the debugger. Once the TCP connections have been established, the debugger's status will change from stopped (blue) to waiting (yellow).

To stop the debugger, click „Stop debugging“. This will terminate the program being debugged as well.

1.3.2 Stepping, stepping over and running to the end of a method

Immediately after the debugger has been started, execution of the debugged program is paused and a yellow marker is placed on the left of the line that the debugger is currently working on. The debugger then waits for you to choose how to navigate through the code. To move by one instruction at a time, choose „Step“. This will allow the program to move on to the next instruction in the code. Once reached, the program will pause again and move the yellow marker to that line.

Stepping through the program in this way allows you to inspect the effect of each instruction, thus facilitating a more thorough understanding of the code. This way of moving through the program is especially helpful when trying to find a coding error („bug“).

As helpful as stepping on a per instruction base is, it is also time-consuming. Sometimes you may have to step through a long method that you already know works just to get to the code that you suspect contains a bug. In such cases, you can choose to step over a method. Choose „Step

over“ right before a method call to skip ahead. Please note that this will not skip the execution of this method entirely. Instead, the code will run until the end of the method without you having to step through it manually. Once the method is finished, you'll be able to continue stepping through the code.

Other times you may have stepped into a method before realizing that you're not interested in stepping through it on a per instruction base. Choose „Run to end of method“ to allow the debugger to finish execution of this method call without forcing you to step through it manually. This is especially handy if you stepped into a method by mistake or if you find yourself inside a deep recursion.

1.3.3 Using breakpoints

Another way of navigating through the code is using *breakpoints*. Breakpoints allow you to mark lines within the source code at which you want to pause the program. You can add or remove a breakpoint at a specific line by placing the cursor inside that line and clicking „Toggle breakpoint“. This will add or remove a blue marker to the left of that line, respectively. By choosing „Run to next breakpoint“ you can then run the program without having to manually step until said breakpoint is encountered (or „hit“). Execution will pause at this point, allowing you to continue using manual stepping. Note that hitting a breakpoint takes precedence over stepping over a method or running to the end of a method, i.e. if a breakpoint is encountered execution always pauses.

You can always choose to run to the next breakpoint, even when you don't have any breakpoints defined. In that case, the program will run to completion. If you want to disable breakpoints without removing them completely, click „Disable all breakpoints“. Disabled breakpoints can no longer be hit, the debugger ignores them. If you wish to reactivate all breakpoints, choose „Enable all breakpoints“. You can remove all breakpoints at once by clicking „Remove all breakpoints“.

1.3.4 Viewing variables

When a variable definition is encountered during the execution of the program, said variable will be placed in the table of defined variables. This table displays the name, value and address of every currently visible variable. A row containing a newly added or changed variable will flash yellow, making it easier to detect changes.

By default, variables are sorted ascending by name. To switch to descending sorting, click the „variable“ column header.

Some names or values may be too long to display inside the table, especially large data structures such as arrays or lists. Such entries will be cut off after a set length and an ellipsis (...) will be displayed where they've been cut off.

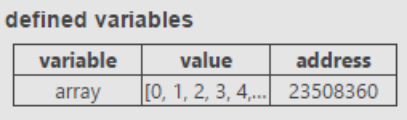

Figure 1.3: An entry in the variable table that has been cut off due to length restriction

To view such entries, double click them to open them in a separate window.

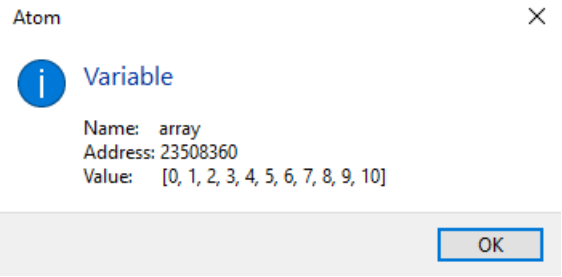

Figure 1.4: The same entry, shown in full in a modal window

Over time, the list of known variables may grow beyond the original size of the table. In that case scroll bars will be displayed to the right of the controls, allowing you to scroll to a specific entry in the table.

1.3.5 Viewing the call stack

Upon entering a method, an entry for this method will be placed on the *call stack*. This entry contains the name of the method and the values of any arguments passed to the method. The call stack contains all calls to methods that have not run to completion. Since it is a stack, the most recent call is always placed on top.

The call stack allows you to trace which method called which, making the program flow more tangible. As with the variable table, calls that are too long to display in the table are cut off. Double click on a call to display it in full.

Next to each call is a „Replay“ button. Clicking this button will activate the replay feature, which will be covered next.

1.3.6 Replaying a call

The replay feature allows you to view a recording of everything that’s happened from a specific method call on forward. Sometimes you may encounter an error or unexpected behavior that was caused by an earlier instruction. Using the replay feature, you can return to that instruction to inspect it more closely. Please note that the replay feature only uses recorded data, i.e. using it does not revert the program to an earlier state. In fact, using the replay feature has no impact at

all on the program you're currently debugging. The program is not aware that a replay is taking place, it simply waits for you to return from the replay.

A replay can only be started from a method call on forward. To start a replay, click the „Replay“ button next to a method call on the call stack. This will suspend the execution of the program being debugged and switch to the replay mode. The status will include a „(Replay)“ prefix and a striped background to emphasize that you're working on replay data. Other than that, the debugger will behave just as it does when debugging a live program. All the features discussed before are available and behave as described.

You can stop the replay at any time by choosing „Stop replay“. Alternatively, you can step through the program until there is no more replay information. This will be signalled by an „End of tape“ message (purple status). Choose „Stop replay“ to return to live debugging.

You can nest replays as you wish, i.e. you can start a replay within a replay. Please note that „Stop replay“ will always return you to live debugging.

1.4 Troubleshooting

Should you encounter an error or unexpected behavior while using the debugger, you may be able to resolve the problem by following the advice below.

Ensure level can be debugged: If the status message reads „Level not debuggable“ (red background) then the debugger cannot be used on the currently selected language level. For instance, the fourth level defined in the `levels-language-ruby` package cannot be debugged due to technical restrictions. Please switch to a level that can be debugged to use the debugger.

Restart Atom and/or your computer: If you encounter an error, please consider restarting Atom and/or your computer and trying again. If the error persists, read on.

Ensure a Java 8 compliant JRE is installed: As described in 1.1, a Java 8 compliant Java runtime environment (JRE) is required to run the debugger. Please ensure that you have a working Java 8 compliant JRE installed by opening a command prompt (on Windows) or a terminal (on Ubuntu or Mac) and typing `java -version`. Note that Java 8 may sometimes be referred to as Java 1.8, i.e. `java version 1.8.0_102` denotes Java 8, patch level 102. If you're using a Mac and running `java -version` returns a message informing you that a JDK is required, please install a JDK as described in 1.1.

Ensure TCP connections are allowed: The debugger uses the TCP protocol to communicate with the Atom GUI and the (Ruby) runtime. If you have a firewall installed, such as the Windows firewall, ensure that it allows TCP connections from your computer (`localhost`) to your computer on TCP ports 59599 and 59598.

Stop debugger process: If you have successfully used the debugger in the past but it doesn't start anymore, then the debugger process may have gone into an undefined state. To stop the debugger process, you must stop the Java Virtual Machine running the debugger. This will be a process named `java`. To stop it, use the Task Manager (on Windows), System Monitor (Ubuntu)

or Activity Monitor (Mac). Select the `java` process and choose „end task“, „kill“ or „force quit“ to end the process. Then try launching the debugger again. If you're unsure about how to stop a process on your computer, you can also reboot your computer to stop the process.

Uninstall debugger package: If your error persists, please uninstall the debugger package using the Atom package manager, restart Atom and reinstall the package (see 1.2).

Open an issue on GitHub: If none of these tips help and you feel that you have encountered a bug, feel free to open an issue on GitHub. Go to `https://github.com/ulrichth/levels-debugger-ruby/issues` and choose „New Issue“. Please describe your problem as accurately as you can and include steps to reproduce the error if possible. Note that a free GitHub account is required to create a new issue.