Universiteit Leiden
The Netherlands

Computer Science & Economics

Designing and implementing a debugger for the Hedy programming language

Felicia Redelaar

Supervisors:
Dr.ir. F.F.J. Hermans
Dr.ir. E. Aivaloglou

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

21/06/2022

Acknowledgements

I would like to say a special thanks to my supervisor Felienne Hermans. Her guidance, support, and insight in this field have made this work possible. I would also like to thank the PERL research group for the discussions and weekly support I received. Also, a big thank you to all of the participants who participated in the study's interviews and a thank you to the participants' programming teacher. Finally, I would like to thank my family and loved ones for the endless support, understanding, and encouragement to complete my academic journey.

Abstract

Hedy is a text-based programming language used by children for educative purposes. This thesis describes the design and implementation of a debugger for Hedy. It also involves interviewing six children to get feedback on the debugger's user-friendliness and usefulness. Quantitative data has been collected to discover whether the website's loading time is statistically significantly slower when the debugger is implemented. Our findings show that children value the debugger and could easily use it. Children also expressed more understanding of Hedy code and variables when using the debugger. Statistical testing showed no significance for slower webpage loading time.

Contents

1 Introduction

Many tools have been developed to teach children how to code. Some of these are relatively well known, such as Scratch [1]. Many of these tools are block-based. Block-based programming tools use draggable blocks in a work area where instructions are represented as the blocks. At some point, however, children have to switch to text-based programming languages like Java and Python. In text-based programming languages, coding is done by typing in a specific syntax. There are strict rules for each particular programming language. Researchers have found that children find the transitioning to be difficult [2, 3]. There are many problems that children could face. Powers et al. [2] found that children struggle with syntax errors.

Research has been conducted regarding the transition from block-based to text-based coding [4, 5]. One of the developed tools to smoothen the transition is Pencil Code [6]. Pencil Code allows students to toggle between text and block code back and forth. Hedy [7] is another tool that has been developed to teach children how to code. Hedy is a text-based programming language. The creators of Hedy are aware of the problems stated earlier. This is why Hedy is created and why Hedy's core concept is to teach children to code gradually. Gradual learning means that the user does not have to learn all the syntax and coding rules at once. Instead, the user learns this gradually throughout many levels. For example, the user starts at level 1 and progressively learns more rules at each higher level. Finally, at level 18, the user has learned enough rules to write a subset of syntactically valid Python.

Next to learning how to program, an essential concept in programming is learning how to debug. In most programming languages, there is an ability to debug code. The contributions of this research are finding the core characteristics of a debugger and then researching whether there is a way to design and implement a debugger for Hedy that is helpful and user-friendly for children. Six children were interviewed to get user feedback and results on the newly implemented features.

2 Background and Research Questions

There are many debugging techniques, and a debugger can come in many shapes and sizes. This section will discuss some debugging techniques and existing debuggers used by children learning to code and industry professionals. We start by looking at Hedy's user interface, what debugging is, and then specify the core characteristics of a debugger by looking at examples. This Chapter will function as a foundation for making design decisions for Hedy's debugger. At the end of the section, we will state the research questions.

2.1 Hedy user interface

Hedy is web-based, and users visit `www.hedycode.com` to use Hedy. Figure 1 shows Hedy's UI. In Figure 1, five features have been highlighted with a number. At number 1, multiple tabs can be seen. These are used to showcase different adventures or puzzles within a level. Number 2 represents the code editor for the users. Number 3 is the output of the code editor. Number 4 is the button that will run the user's code. Finally, number 5 is a collapsible list. This list contains the newest commands at the current level.

The design of the debugger will have to match the existing theme. More, the implementation will be around the code editor and the output. Lastly, the debugger will have to work with the existing technologies, such as Ace editor. These technologies are discussed in detail in Section 4.

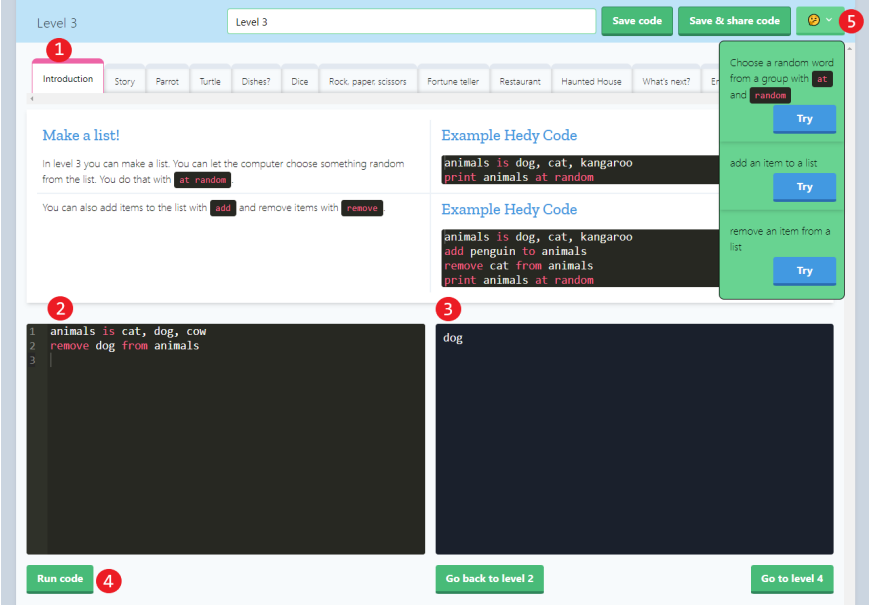


Figure 1: Hedy's user interface at level 3 with tabs (1), code editor (2), output (3), run button (4), and a list of commands (5).

2.2 Debugging techniques

Whenever programmers write code, they must ensure that their program execute as designed. Bugs and errors cause programs to malfunction or sometimes not work at all. The debugging procedure often exists from analyzing the code, googling, and reading the error messages. Besides this, many programmers use debuggers where it is available. For example, a debugger can help a programmer to examine the values of variables while a program is running [8].

Many techniques can be applied when debugging code. One of the techniques is Print debugging. Print debugging is watching print statements to get more insight into a program. This technique can be seen as unsophisticated but also as fast and easy. Another technique is called Post-mortem debugging. Here programmers debug the program after it has crashed. Programmers, for example, read the log files to reconstruct what happened [9].

2.2.1 Debugger characteristics

Besides debugging techniques, there also are debuggers. Debuggers are a tool for debugging. Debugging tools have existed since the early 1980s. One of the first debuggers is included in BASIC Programming [10]. BASIC Programming is a video game cartridge for the Atari 2600. Gamers used the cartridge to learn how to program video games in BASIC.

BASIC's programming display was divided into six regions. The first region was the program where the user typed the instructions. The second region showed the stack with temporary program results. The third region would show variables stored by the program and their values. This functionality is still used in many modern debuggers nowadays. The fourth region showed the output values that the program created. The fifth region showed the status with the remaining amount of available memory. Finally, the last region showed a display with the graphics [10]. See Figure 2 to see the main display of the Atari 2600.

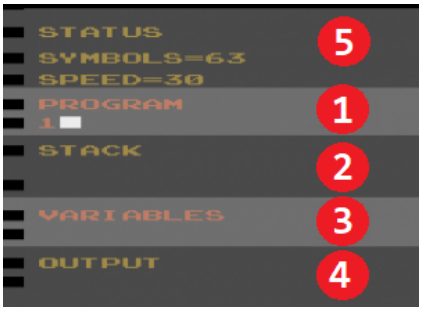


Figure 2: Main display of Atari 2600 BASIC programming with the editor (1), stack (2), variable values (3), output (4), and status (5). The region that shows the graphics are not visible in this Figure.

Nowadays, modern debuggers are more sophisticated. Some of Atari's debugging features are still present in modern-day. The main characteristics of a modern debugger are:

(a) *"Displaying program source code and permits browsing through the code"* [8].

(b) *"Setting breakpoints to suspend the execution of a program at breakpoints"* [8].

(c) *"Examining the stack and symbol table"* [8]. The call stack is a list of all functions that have been called in order to get to the current point of execution in the program [11].

(d) *"Examining the values of local and global variables"* [8]. This can be at breakpoints or by setting variable watches.

(e) *"Controlling the execution of a program. This can be done by slowly stepping through a program one line at a time to inspect and analyze it"* [12]. This is at the source code level statements, not low-level single-machine instruction.

(f) *"Reverse debugging is the ability of a debugger to stop after a failure in a program has been observed and go back into the history of the execution to uncover the reason for the failure"* [13]. Reverse debugging is a notable mention. It is a feature that some debuggers support.

2.2.2 Command-line based debugger

Now that we have discussed some debugging techniques and the common characteristics of debuggers, we will focus on some debugger examples. A popular command-line debugging tool is the GNU debugger (GDB) by the GNU Project. The initial release was in 1986 [14]. This debugger runs on Unix systems for many programming languages. It does not contain a graphical

user interface, although several front-ends have been built for the GNU debugger, such as gdbgui [15].

The GNU debugger features all characteristics *a* to *f* of a debugger. Commands such as `next` steps to the next line of code. `print <variable name>` prints the stored value of a specified variable. In Figure 3, an example of the GDB can be seen. Here, a part of the source code is displayed, and the variable values are shown.

```
(gdb) l
3       # include "../headers/liblist_single.h"
4
5       int main (void)
6       {
7               void* ce;
8               int a = 42;
9               int b = 1337;
10              int c = 314;
11
12              struct list_s* list = create_s('s'); // Our data is on the stack
(gdb) n
9               int b = 1337;
(gdb) n
10              int c = 314;
(gdb) n
12              struct list s* list = create s('s'); // Our data is on the stack
```

Figure 3: GDB controlling the execution of a program using breakpoints and the `next` keyword by typing `'n'` [16, 17].

2.2.3 Console log

The Console log is a more modern way of debugging than using the command line. However, the console log is not a debugger. Instead, it is a tool used for debugging. This tool is used heavily as a debugging tool in the web development industry.

The console log is a method for the earlier mentioned print debugging. In Figure 4, the console of a web browser is used as a tool for debugging. Whenever code is executed that contains `console.log()` statements, these will be visible in the web browser's console. In Figure 4, the console shows in which file and line the `console.log()` statements were executed. In this Figure, all three `console.log()` statements were in `app.js` at line 298, 312 and 895. Although the console log is not a debugger, it does have the debugger feature characteristics *a* and *d*.

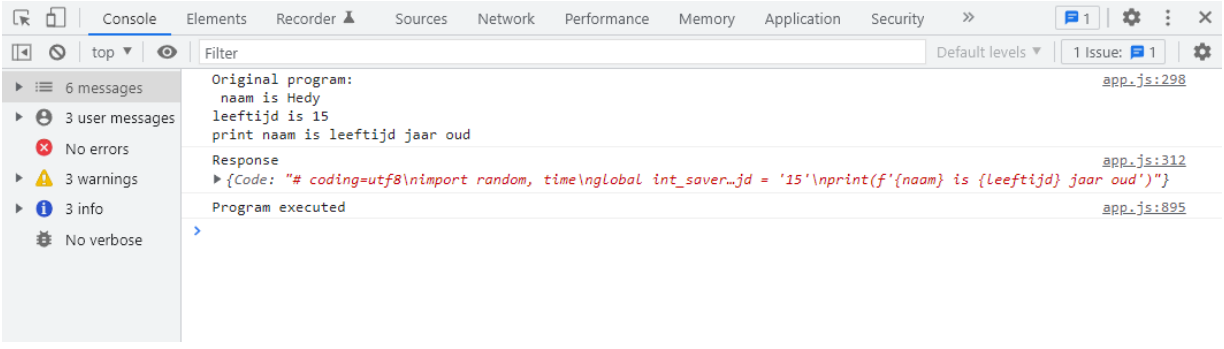


Figure 4: Google Chrome Console showing the output of a console.log() statements in `app.js`.

2.2.4 DevTools

Most popular web browsers such as Google Chrome [18] and Firefox [19] contain a debugger for developers. This debugger is often called DevTools. DevTools can be accessed using the keyboard shortcut Crtl + Shift + I. Another way is by right-clicking the web page and clicking on the "Inspect (Element)" menu item.

The DevTools debugger can set breakpoints in JavaScript files. When the page is refreshed, the execution is paused on the breakpoint. For example, in Figure 5, the breakpoint was set at line 869. This can be seen at the highlighted point 1 in Figure 5. The variables and their values can be seen at highlight point 3. The programmer can also control the program's execution through the arrow symbol shown at highlight point 2. DevTools features characteristics *a* to *e* of a debugger.

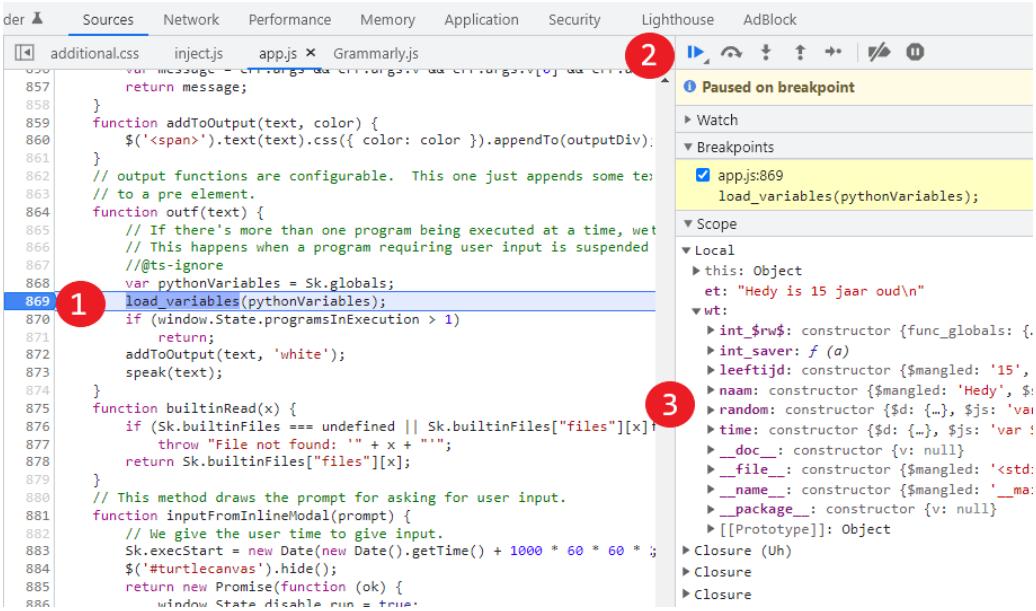


Figure 5: Chrome DevTools Debugger showing variables on breakpoint in `app.js`.

2.2.5 IDE debuggers

An integrated development environment (IDE) is a software application that provides facilities for software development. An IDE contains a source code editor, a build tool, and a debugger. There are many IDEs available such as Microsoft Visual Studios [20], Eclipse [21], and PyCharm [22]. PyCharm is used to write and execute Python code. Figure 6 is a screenshot of PyCharm's debugger. Highlight point one shows a breakpoint. Highlight point two shows the arrows that control the program's execution. Highlight point three shows the variables and their values. The main functionalities of Devtools' and PyCharm's debugger are the same. This debugger features characteristics *a* to *e* of a debugger.

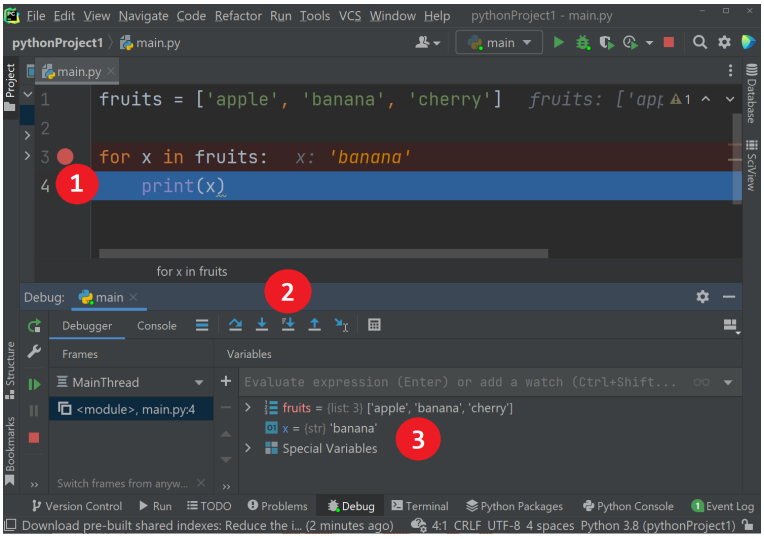


Figure 6: PyCharm's debugger showing variables and their values on a breakpoint. [22]

2.2.6 Scratch

Now that we have seen what modern debuggers look like in IDEs and web browsers, we will dive into debuggers for educational tools. The first tool that we will discuss is Scratch [1]. In Figure 7, a screen capture can be seen of a Scratch program. The user can click the cat in this program to make a "meow" sound. The cat makes a "woof" sound whenever the space bar is pressed. The meow and woof sounds are counted and displayed at highlight point one. At highlight point two, the user can select which variables to display. Scratch does not have a typical debugger. It does, however, have one core feature of a debugger. Scratch can show the used variables and their values. Scratch therefor features characteristics *a* and *d* of a debugger.

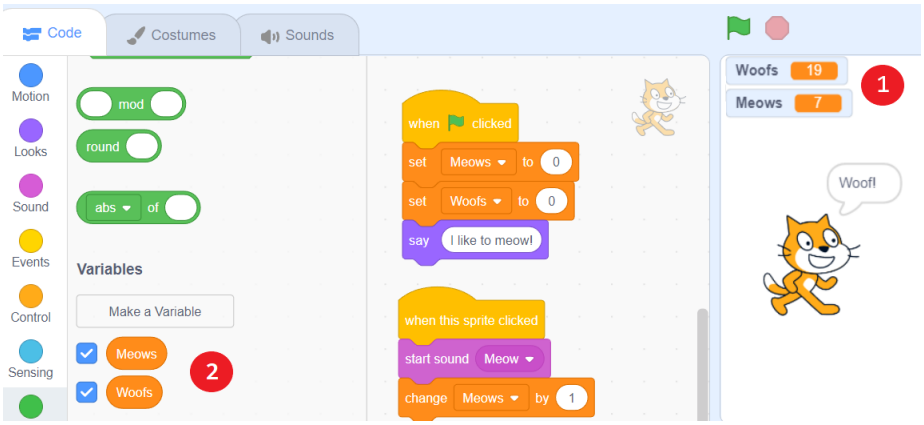


Figure 7: Scratch showing variable and its value during the execution. [1]

2.2.7 Code.org

The second educational tool that we will discuss is Code.org [23]. In Figure 8, a screen capture can be seen of a program. The user has to use code blocks in this program to move the

spelling bee to create a specific word. Code.org does not have a typical debugger. It does, however, have some core features of a debugger. These features are only available in specific levels of Code.org.

In Figure 8, there are two highlight points. Highlight point 1 shows the ability to control the execution of a program by using a step button. Highlight point 2 shows the variable and its current value. Code.org therefor features characteristics *a*, *d*, and *e* of a debugger. Other popular educational tools such as Code Combat and Khan Academy do not contain a debugger or common debugger features.

Figure 8: Code.org [23] Spelling Bee level showing a step-by-step button and a variable and its value during the execution.

2.3 Research questions

Debugging is often experienced as difficult, especially for novice programmers [24, 25]. According to Kernighan's law, debugging is twice as hard as writing a program in the first place [26]. Children who use Hedy can debug programs using error messages or analyzing their code. However, a built-in debugger could be a powerful tool for children to get more insight into their code.

An important aspect to keep in mind while developing the built-in debugger is that it should be intuitive to use for children. Making a confusing tool for children could damage Hedy's reputation and might decrease children's spirit when coding. Therefore, to effectively design and build a child-friendly debugger for the Hedy programming language, we answer the following research questions:

RQ1: Can we build a debugger for the Hedy programming language?

RQ2: How can we design and implement this view to be as user-friendly as possible?

RQ3: What is the impact of the debugger on Hedy's performance?

3 Designing the debugger

Now that we know a debugger its main characteristics and have seen examples, we will start designing Hedy's debugger. In this section, we begin by selecting the features for the debugger. Then we will present and discuss the design choices for each element.

3.1 Selecting the features

In Section 2.2.1, the main characteristics of a debugger were summed up. To summarize, the six characteristics are to browse the source code (a), set breakpoints (b), examine the stack (c), show values of variables (d), control the execution of a program (e), and to reverse debug (f). In order to design the debugger, we have selected three features out of all six. The features we want to include are characteristics a, d, and e.

Feature characteristic *a: displaying source code*, has been selected since it seems beneficial for the user to see their code while debugging. It also did not require extra work since the code editor and output already exist. Characteristic *d: examining values of variables*, could help children to learn the concept of variables better. This functionality also appears in Scratch. Characteristic *e: controlling the execution*, could be a powerful functionality for kids to get more insight into their code. Later in this research, the implemented debugger will be tested on its performance. The methodology and results are shown in Sections 6 and 7.

The features we are not including into the debugger are characteristics b, c, and f. Feature characteristics *b: setting breakpoints*, and *f: reverse debugging* are not selected in order to keep the scope of this research feasible and the result obtainable. These functionalities are complex to implement and would have been time-consuming. It could, however, be exciting for future work. For this reason, a design will be presented for characteristic b in Section 3.4. Despite that this functionality will not be implemented for this research.

Feature characteristic *c: examining the stack*, is not selected because Hedy does not allow users to define functions. A call stack is a list of functions that have been called to get to the current execution point [11]. If the Hedy compiler stopped the execution due to an error created by the user, no functions would be shown in the call stack. Implementing this feature would therefore add no value.

3.2 Examining variables

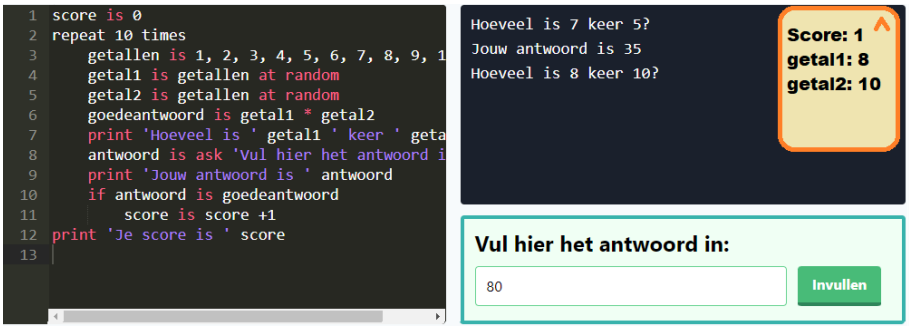
This section will present the initial and final design for displaying variables and their values in a Hedy program. All designs have been created using tools such as Photoshop.

3.2.1 Initial design

In Figure 9, the initial design is presented. Figure 9 is a collection of two photoshopped designs that represent screen captures. The screen captures only show the text editor and output to save space. In Figure 9a, there is an orange rectangle in the upper right corner. When the user clicks on this drop-down button, the view will display the variables and their values. An example of this

can be seen in Figure 9b. A key element of the feature presented is to update the variable values automatically whenever their values change in the program.

(a) Screen capture 1: Minimized variable view.



(b) Screen capture 2: Expanded variable view.

Figure 9: Initial design for displaying variables and values. Figure 9a represents the first screen capture. Here, the variables view is minimized. Figure 9b represents the second screen capture. Here, the variable view is expanded. This allows the user to examine their Hedy program variables and their values.

3.2.2 Final design

The purpose of the initial design was to communicate the functionality of characteristic d to Hedy's core team. Once the team approved the initial design, we adjusted the design. The initial design would work functionally but could be distracting since it did not match the current theme. The final design has improved in its UI consistency and has been designed to be as straightforward as possible. In Figure 10, the final design is presented. Similar to Figure 9, Figure 10 exists out of two screen captures.

In Figure 10, the orange rectangle has been replaced by a small blue drop-down with a label icon. A box or label is sometimes used to showcase variables. In Hermans et al. [27] research, a box and label outperform each other in certain situations. However, we chose a label in consultation with a PERL group member for this design. When the drop-down is clicked, the view will display the variables and values used. This does not differ from the initial design. The design aspect, however, has changed. Blue matches the Hedy theme since this specific color is used frequently in the UI. The font family and color have also been adjusted. The reason behind this is that the editor uses a white font color. The variable view matches this to create consistency in the UI.

(a) Screen capture 1: Minimized variable view.

(b) Screen capture 2: Expanded variable view.

Figure 10: Final design for displaying variables and values. Figure 10a represents screen capture 1. Here, the variables view is minimized. Figure 10b represents screen capture 2. Here, the variable view is expanded.

3.3 Execution control

In this section, we will present the design for controlling the execution of a Hedy program. The design is presented in Figures 11, 12, and 13. According to the design, the user can control the execution by running a program step by step. In Figure 11, a new button is presented. This button shows an image of a bug representing a debugging button. The regular "run" button is next to the debugging button.

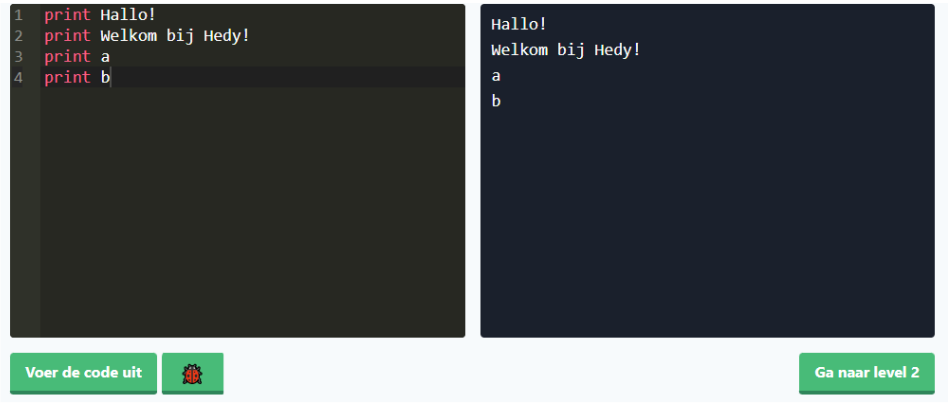


Figure 11: Design for adding execution control in Hedy by adding a debug button.

Whenever the debug button is clicked, the UI changes from Figure 11 to Figure 12. In Figure 12, the debugging button is replaced with three new buttons. These buttons are 'forward', 'refresh', and 'stop'. Normally, all Hedy code is executed at once when executing a Hedy program. In Figure 12, a blue highlight line shows which code is being executed. This highlight line moves one step each time the forward button is pressed. See Figures 12 and 13 to see an example of step-by-step execution. The lines before and on the blue highlighted line are executed in both Figures.

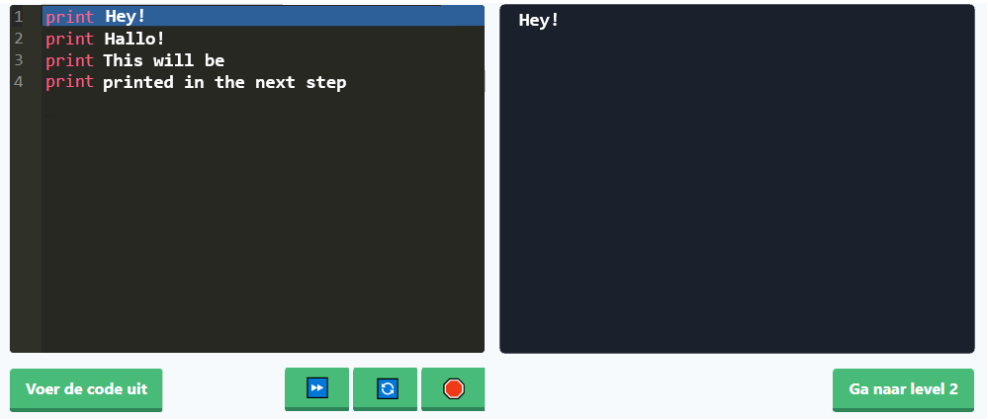


Figure 12: Design for execution control in Hedy, showing a step-by-step execution. The first line is executed.

The stop button will hide the forward, refresh and stop buttons. It will also deactivate the step-by-step execution. Pressing the stop button will result in Figure 11 in the UI. The refresh button makes sure that the step-by-step execution starts at the first line. This could be useful whenever a child wants to repeat the step-by-step execution from line 1. Pressing the refresh button will result in Figure 12 in the UI.

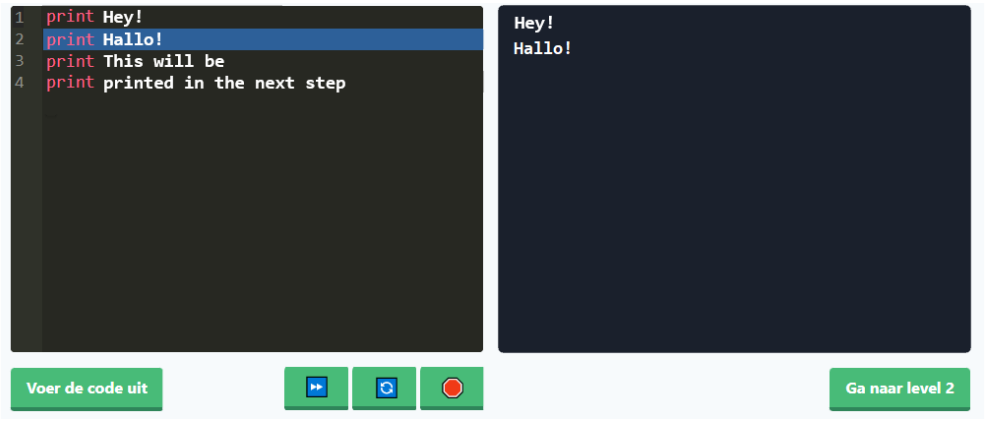


Figure 13: Design for execution control in Hedy, showing a step-by-step execution. The first and second line are executed.

3.4 Breakpoints

In Section 3.4.1, we will present the design of an implemented feature where users can make the compiler ignore certain lines of code. In Section 3.4.2, we will present a design for adding breakpoints in Hedy even though this research will not implement this. The design could be helpful in future work.

Figure 14: Design for breakpoints in Hedy.

3.4.1 Deactivating lines of code

We will implement a feature to mark lines in Hedy programs that the compiler will not execute. Some programmers find it helpful to be able to "switch off" some code while debugging code. This, however, can be seen as bad programming practice since programmers should use comments for explanations. This feature has been developed to allow children to switch off some code while avoiding teaching them bad coding habits.

In Figure 15, children can switch off code by clicking on a specific line number in the gutter. The editor will strike the code on this line whenever the number next to it is clicked. A sleeping emoji will also replace the number. The line with struck-through text will not be executed whenever the code is run. The user can reactivate the line by clicking on the emoji.

3.4.2 Traditional breakpoints

In Figure 14, the design for a traditional breakpoint in Hedy is presented. Breakpoints are often characterized as a red dot in modern IDEs. This can also be seen in Figure 6. The red dot will appear when the user clicks on a specific number in the gutter. Once the debugging button is pressed, the compiler will execute the code till the breakpoint. From there on, the user can analyse the program using the step-by-step execution control presented in Figures 11, 12, and 13. The values of variables can also be analysed using the feature presented in Figures 9 and 10.

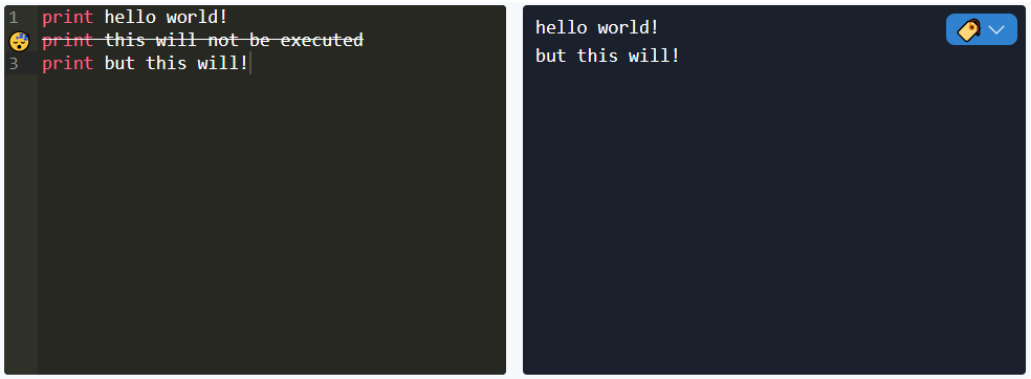


Figure 15: Design for "turning off" code in Hedy. The second line is not executed in the program.

4 Technology

Previously we have seen which features we want to implement for Hedy's debugger. We have also seen what they should look like. In the next section, we will discuss what technologies Hedy uses. The following technologies will have to be used to implement the debugger.

4.1 ACE Editor

Ace is an embeddable code editor written in JavaScript [28]. Ace is an open-source project that is used by different kinds of applications. Some of these applications are created for educational purposes such as the earlier mentioned Khan Academy [29], Codecademy [30] and Code Combat [31]. See Figure 16 for an example of how Khan Academy is using Ace editor. Hedy also uses Ace as a code editor. Ace Editor allows for code editing. When the edited code is executed, Ace has an optional output window. In addition, there are many features available such as (custom) syntax-highlighting.

Some of ACE's built-in functions have been used for this research. A subset of the functions that have been used are `setBreakpoint()`, `getBreakpoints()`, `clearBreakpoints()`, `getValue()`, and `getFirstVisibleRow()`. Custom functions have also been written for the implementation. Whenever custom functions for the front-end are written, it is necessary to be aware of the multiple Ace layers. The editor exists out of the following layers: `ace_content`, `editor`, `ace_text-layer`, and `ace_line`. Knowing these layers allows one to edit the front-end on all Ace layers.

4.2 HTML, CSS and TypeScript

The three main languages used to build websites are HTML, CSS, and JavaScript. This also holds for Hedy's website. HTML is used as the skeleton of a website, and it contains all content of a website. CSS is used to improve the appearance of a website. For example, web developers can set the colors of specified web elements through CSS. Finally, JavaScript is used to make a website interactive.

Figure 16: Ace Editor being used for Khan Academy [29].

Hedy uses HTML by using Flask. Flask is a micro web framework that programmers use to build web applications using Python. HTML pages are created in Hedy by creating HTML templates. For this research project, the HTML file `incl-editor-and-output.html` has been edited.

Hedy uses both JavaScript and TypeScript. The way this happens is through programmers writing their code in TypeScript. Then, the TypeScript code gets compiled into JavaScript code. Finally, in the web browser, the compiled JavaScript code is executed. TypeScript is an object-oriented programming language and is a syntactical superset of JavaScript. It extends JavaScript by adding types to the language [32].

One of the most powerful TypeScript functionalities is the ability to catch errors/bugs while writing code. JavaScript provides primitive values such as `boolean, string` and `number`. JavaScript, however, does not check whether these are assigned while writing the code. TypeScript does check this and notifies programmers by highlighting lines with unexpected behavior. By validating the code ahead of time with static type checking, the chance of writing bugs becomes lower. For this research project the TypeScript files `app.ts` and `tabs.ts` have been edited.

5 Implementation

This section will describe the implementation of the earlier presented designs. First, we will discuss the implementation of the variable view. Secondly, we will discuss the implementation of the execution control.

5.1 Variable View

The variable view exists in two parts. The first part loads the variables that users have created in Hedy programs. The second part is regarding the color-coding of variable types.

5.1.1 Loading variables

Whenever users create Hedy programs with variables, the variables are stored by Skulpt. Skulpt is an implementation of Python for web browsers [33]. Skulpt stores the variables that users have created under the variable `Sk.globals`. This variable has been used to built the variable view. We will discuss three functions since they contain the largest part of the implementation. These functions are `load_variables()`, `clean_variables()`, and `show_variables()`.

The function `load_variables()` receives the Skulpt variables as parameter. Then, it calls the function `clean_variables()`. A snippet of this function can be seen in Listing 1. This function removes unwanted variables that Hedy uses for calculations behind the scenes. Variables such as `random`, `time` and `turtle` are used to execute certain Hedy keywords such as `at random`, `sleep` and `turn right`. The user does not need to see these variables in the variable view, since it could cause confusion. Finally, the function creates a tuple with the variable names and values.

```
1  function clean_variables(variables: any) {
2    const new_variables = [];
3    const unwanted_variables = ["random", "time", "int_saver", "turtle"];
4    for (const variable in variables) {
5      if (!variable.includes('__') && !unwanted_variables.includes(variable)) {
6      let newTuple = [variable, variables[variable].v, extraStyle];
7      let extraStyle = special_style_for_variable(variables[variable]);
8      new_variables.push(newTuple);
9      }
10   }
11   return new_variables;
12 }
```

Listing 1: TypeScript snippet from function `clean_variables()` in `static/js/app.ts`. This snippet filters unwanted variables and creates a TypeScript tuple.

After the variables are filtered, an HTML list is created with the remaining variables and their values. This is done by creating a for-loop that creates HTML list items for each variable in the created tuple. Listing 2 shows the responsible code. The HTML tag `<li>` is an element that defines a list item. The view will show the variable's name on the left side and the value on the right side. Lastly, the function `show_variables()` is used to turn the variable view on or off.

```
1 for (const i in variables) {
2    variableList.append('<li>${variables[i][0]}: ${variables[i][1]}</li>');
3 }
```

Listing 2: TypeScript snippet from function function `load_variables()` in `static/js/app.ts`. The snippet creates a HTML list with variables and their values.

The HTML written for the variable view can be seen in Listing 3. A container has been built that contains two parts. The first part is a button to minimize or expand the variable view, using the function `showVariableView()`. The second part is the variable list and a header. The header in line 5 can be translated to other languages.

```
<div id="variables_container">
    <button id="variable_button" onclick="hedyApp.showVariableView()">
    </button>
    <div class="hidden" id="variables">
        <div id="variables_header">{{_('variables')}}</div>
        <ul id="variable-list"></ul>
    </div>
</div>
```

Listing 3: HTML snippet from `templates/incl-editor-and-output.html`. The snippet shows the HTML for the variable view.

5.1.2 Color-coding variables

The variables in the view can be color-coded. Currently, this function is not used since the syntax highlighting in the editor needs to be adjusted first. The function `special_style_for_variable()` is being called in the `clean_variables()` function. This function can assign any color to number, string, boolean, and list variables. Currently, this function colors each variable type as white. The function recognizes the variable type by using JavaScript's `typeof()` function. This function is used to find the data type of a variable. The colors are passed through the variable view by adjusting Listing 2 to Listing 4. The function `clean_variables()` has also been adjusted. The tuple now includes a variable color as well. This can be seen in Listing 1, lines 6-7.

```
for (const i in variables) {
    variableList.append('<li style=color:${variables[i][2]}>${variables[i][0]}:
    ${variables[i][1]}</li>');
}
```

Listing 4: TypeScript snippet from function `load_variables()` in `static/js/app.ts`. This snippet assigns colors to variables in the frontend.

5.2 Execution Control

In the following section, we will discuss the implementation of the execution control in two parts. The first part will discuss the control buttons and the flow of the debugger. The second part explains how the debugger only executes certain lines.

5.2.1 Control buttons

Four buttons have been created for the debugger in HTML. These buttons can start the debugging execution mode, increment the line, stop the debugging execution mode and reset the debugger to the first line. In Listing 5, the HTML code can be seen for these buttons. Whenever the debugging button is clicked, the function `startDebug()` is called. This function hides the regular run button and shows the increment, stop and reset buttons. Finally, it sets the debugger to the first line and executes this line. This causes the user to see the first line of their Hedy program being executed.

The user is able to increment the debugging line using `incrementDebugLine()`. This function keeps

track of which line has to be executed, and it will increment this by one line each time. Finally, it will run the program by calling the `debugRun()` function. The `debugRun()` function executes the Hedy program to the line that is being kept track by the `incrementDebugLine()` function. The `incrementDebugLine()` function calls the `setdebugline()` function. The `setdebugline()` function highlights the line that is being kept track. This highlighter function also gets called when the debugger is stopped or reset.

When the debugging mode is stopped, the debugging buttons are hidden, and the traditional run button is shown. The function `clearDebugVariables()` is called. This makes sure that possible given user answers by the `ask` keyword are cleared. When a user reruns their program, the functionality prevents it from being answered by old user inputs. Figure 17 shows a visual summary and a complete overview of the debugger's workflow through a flow chart.

```
1 <button id="debug_button"    onclick="hedyApp.startDebug()">          </button>
2 <button id="debug_continue" onclick="hedyApp.incrementDebugLine()"> </button>
3 <button id="debug_restart"  onclick="hedyApp.resetDebug()">          </button>
4 <button id="debug_stop"     onclick="hedyApp.stopDebug()">           </button>
```

Listing 5: HTML snippet from `templates/incl-editor-and-output.html`. The snippet shows the HTML for the execution control buttons.

5.2.2 Executing to the debug line

In this section, we will go into more detail on how the debugger only executes certain lines of code. The function `get_trimmed_code()` is responsible for this. The function already existed, but we have altered it. The function gets called each time `debugRun()` is called. The function `get_trimmed_code()` reads the Hedy program and the current debugging line. It splits the Hedy program per line. Finally, it defines the code to be executed by slicing the Hedy code from line 1 to the debugging line. This can be seen in Listing 6, line 7.

```
1  let code = theGlobalEditor.getValue();
2  var storage = window.localStorage;
3  var debugLines = storage.getItem('debugLine');
4  if (code) {
5  let lines = code.split('\n');
6  if (debugLines != null) {
7    lines = lines.slice(0, parseInt(debugLines) + 1);
8  }
9  code = lines.join('\n');
10 }
```

Listing 6: TypeScript snippet from function `get_trimmed_code()` in `static/js/app.ts`. This snippet slices code from line 1 to the debug line. This ensures that the debugger executes a program til a certain line.

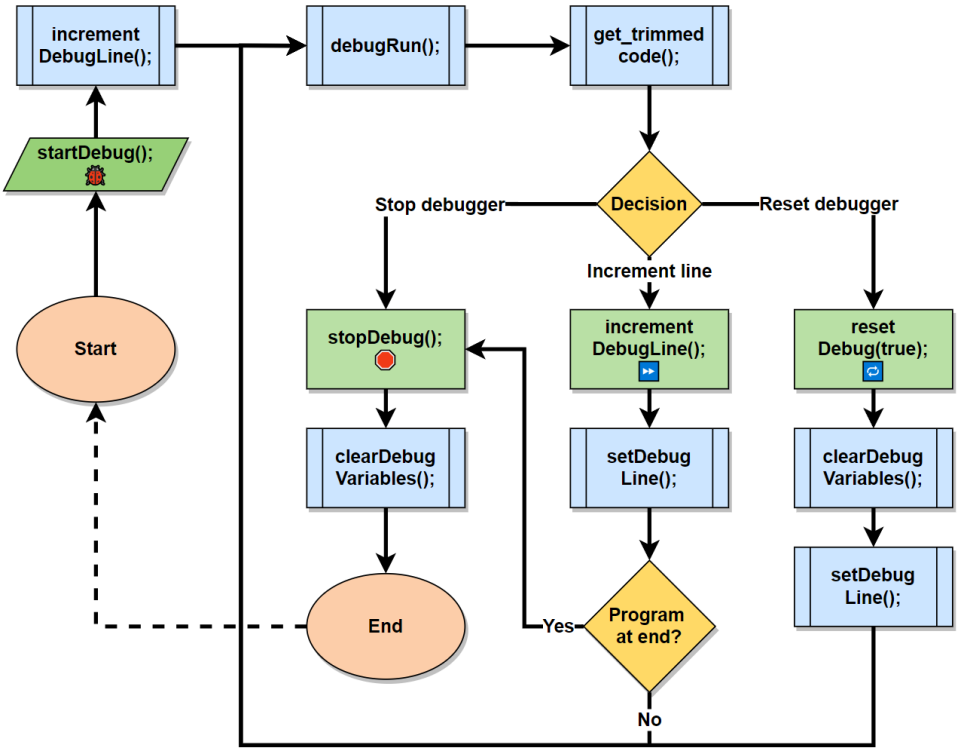


Figure 17: Flow chart for the execution control. The green squares represent calling functions connected to front-end buttons. The blue squares represent called functions.

6 Methodology

Now that we have covered the design and implementation of the debugger, we will focus on the evaluation. This research aims to design and implement a debugger and evaluate its user-friendliness and performance. To answer RQ2, we have used qualitative methods. To answer R3, we have used quantitative methods.

6.1 Qualitative data

Qualitative data gives the advantage of obtaining the participants' full opinions and thought processes. In order to collect qualitative data, six seventh-graders were shown Hedy's debugger and were interviewed. The six interviews took place with one child at a time for 30 minutes in Dutch on April 13th, 2022.

6.1.1 Participants

The six seventh-graders that participated in our study were all students from one school in the Netherlands. All participants had around 16 lessons of prior experience with programming using Hedy. The participants' ages were between 12 and 14 years old. Three of the participants were girls. Two of the participants said that programming was their favorite subject. Another two participants said that programming is their least favorite subject. The rest did not have a preference. Children could only participate if their parents signed an online consent form.

6.1.2 Interview protocol

The interview consists of three parts. These parts are divided into the following paragraphs. It started by briefly explaining the variable view to the participant. Participants were then asked to answer two questions while debugging a couple of Hedy programs. The Hedy programs contained variables such as lists. Using the `add` and `remove` keywords, the values in a list can change. Participants read the prepared programs by the experiment, and got questions asked such as: "Is there a variable in this program, and if so, where?" and "What happens with the value of a specific variable through the program?". Afterward, the participants were asked their general opinions on the variable view.

The interview continued by showing the participants the second feature of the debugger, execution control. Furthermore, participants were encouraged to create their own Hedy programs using the step-by-step execution. Finally, after a couple of minutes, the participants were asked their general opinions on the execution control.

In the end, the participant answered questions regarding the user-friendliness and the swiftness of the debugger.

6.2 Quantitative data

We will measure the performance of the debugger in terms of speed, specifically, the loading speed of the website. For example, when `www.hedycode.com` is loaded, this takes less than a second on average. An independent samples t-test will test the difference between two means to test whether the website's loading time has increased significantly. The website's loading time with and without the debugging features will be measured using the network tab of Chrome Devtools. Instead of using the live website and the live alpha version, a local website will be running. This is to reduce variables that could influence the loading time. During the data collection, the samples will be randomly sampled. The sample size for each group is 30 loading times.

7 Results

This section shows the results of the qualitative and quantitative data. The results of the qualitative data are based on the participants' responses. For the quantitative data, we look at the results of the performed t-test for Hedy's loading time.

7.1 Qualitative data

In this subsection, we answer our research question 2 "*How can we design and implement this view to be as user-friendly as possible?*" based on participants' answers.

7.1.1 Easy to use

A design can be called intuitive and easy to use whenever a user can use a design immediately without having to get lost, puzzled, or reading a manual book. All children mentioned that they

understood the tool within a couple of seconds. Some children mentioned that they liked the simple design. They were all able to use the tool successfully on multiple levels.

7.1.2 Useful and valuable

When asked whether they would use the debugger, all children mentioned using it for every programming class. When asked what children value in the debugger, four children mentioned that they would find it especially useful when learning new keywords or creating extended programs. Three participants said they often forget which variables they had initialized at the beginning of a program. Participant 3 said "*The variable view is useful since it can help me remember which variables I have initialized in the beginning. I often forget this when I am at line 13 or so*". The debugger, therefore, could be a valuable tool for keeping track of created variables. Two other participants mentioned that they struggled with the concept of variables and often got confused. They mentioned that the debugger could give more insight. Participant 1 said "*The variable view is a good addition since I always get confused when using variables. This tool allows you to understand it instantly*". Participant 2 stated "*The variable view is very convenient because my brain sometimes gets cluttered and chaotic. This tool will help me check whether my logic is correct!*".

7.2 Quantitative data

To answer the quantitative aspect of the research question "*What is the impact of the debugger on Hedy's performance?*", we will present the test results of the t-test. We will determine whether the loading speed is significantly slower when the debugger is added. Table 1 contains the descriptive statistics of the loading time. We start by stating the null hypothesis (H_0) and alternative hypothesis (H_1). Table 2 contains the results of the t-test.

H_0: There is no significant difference in the loading speed when the debugger feature is added.

H_1: The loading speed is significantly slower when the debugger feature is added.

	N	Mean	Std. Deviation	Std. Error Mean
Without debugger	30	750.33	90.1	16.45
With debugger	30	776.2	51.1	9.32

Table 1: Average loading time per group in milliseconds

With $p > 0.05$, we may assume that the variants are the same in both groups. H_0 is not rejected ($p = 0.088$). No significant difference was found in the average loading time of these two groups.

				t-test difference of two means		*95% CI*	
	t	**df**	**Sig. (1-tailed)**	**Mean Difference**	**Std. Error Difference**	**Lower**	**Upper**
Equal variances assumed	-1.368	58	0.088	-25.867	18.905	-63.708	11.975
Equal variances not assumed	-1.368	45.896	0.089	-25.867	18.905	-63.922	12.189

Table 2: T-test difference of two means (independent samples) for loading time

8 Conclusions

This thesis has two goals. The first one was to research a debugger's common features and design a debugger for Hedy. The second goal was to implement the debugger and test its performance and value for children. We interviewed six children aged 11 to 14. This study helps us to answer the following research questions:

RQ1: Can we build a debugger for the Hedy programming language?

Our findings show that it is possible to build a debugger for Hedy. By analyzing which features a debugger consists of, we selected the best fitting features for Hedy's debugger. The debugger is designed to control the execution of a program using line-by-line execution and examine variables and their values. The debugger is built using TypeScript for the logic of the debugger, Tailwind, CSS for the styling, and by editing existing HTML to include buttons and a view.

RQ2: How can we design and implement this view to be as user-friendly as possible?

The debugger has been designed to be consistent with Hedy's current theme. Besides this, the simplicity and specific label choices on buttons created more intuitiveness for users. According to the participants, the debugger is easy to use.

RQ3: What is the impact of the debugger on Hedy's performance?

Statistical testing showed no significant slower webpage loading time when implementing the debugger. Besides, participants mentioned that they found the debugger valuable and would use it in every programming class. Therefore, combining the speed element of the debugger and the participants' opinions, we can conclude that the performance of the debugger is satisfactory.

9 Future Work

Future work should be conducted to improve Hedy's debugger. There are three areas for improvement of the debugger. We will discuss these in the following sections. Afterward, a user study regarding the debugger is suggested.

9.1 Execution control

The first improvement that could be made is to make the debugger available after level 7. The problem, however, is that level 8 introduces code looping using the `repeat` keyword. The debugger has not been optimized for looped code. The improvement would be to make the debugger able to iterate lines it has already executed. The debugger would have to recognize the looping keywords and react accordingly.

Another improvement that could be made regarding the execution control of the debugger is to implement reverse debugging. This feature was mentioned in Section 2.2.1. The traditional way for execution control in debuggers is to execute lines chronologically. The downside of this method is that when the user has stepped over a specific line they want to analyze, the debugging process needs to be restarted. This is due to the debugger not being able to execute lines in reverse order. In reverse debugging, this problem is solved by being able to execute lines reversely.

9.2 Selecting watches and variables in view

The second area for improvement is regarding the variable view. It could be beneficial to create a feature where users can select which variables they want to see in the view. This way, the user can focus on the specified variables. Besides selecting variables, it could be beneficial to set watch expressions in the variable view. A watch expression would allow a user to specify mathematical or Hedy expressions that will be recalculated and displayed every time the debugger is paused. As the user steps through the code, the variable view will "watch the expression" and return the results. A user-friendly design would first have to be created to implement this feature.

9.3 Color coding variable types

Variable types such as strings, integers, booleans, and lists are often colored differently in many coding languages. Currently, open-source Hedy contributors are working on this specific syntax highlighting implementation. The color coding in the variable view could be changed when the implementation has finished in the code editor. We have written a function that can color code variable types automatically. Currently, the function displays all variable types as white. The colors per variable type can be easily adjusted in function `special_style_for_variable()` in the file `app.ts`.

9.4 Researching the didactic value

Participants expressed that they gained more insight into their programs using the debugger. It could be beneficial to research which programming concepts could be taught better by utilizing the debugger. This future research could help teachers understand which concepts to explain using the debugger.

References

[1] M. Resnick, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman, Y. Kafai, Scratch: Programming for all, Commun. ACM 52 (11) (2009) 60–67. doi:10.1145/1592761.1592779.

[2] K. Powers, S. Ecott, L. M. Hirshfield, Through the looking glass: Teaching cs0 with alice, in: Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education, SIGCSE '07, Association for Computing Machinery, New York, NY, USA, 2007, p. 213–217. doi:10.1145/1227310.1227386.

[3] D. Parsons, P. Haden, Programming osmosis: Knowledge transfer from imperative to visual programming environments (Jan 2007).

[4] D. Weintrop, C. Bain, U. Wilensky, U. S. Education, Blocking progress? transitioning from block-based to text-based programming (2017).

[5] L. Moors, A. Luxton-Reilly, P. Denny, Transitioning from block-based to text-based programming languages, 2018, pp. 57–64. doi:10.1109/LaTICE.2018.000-5.

[6] D. Bau, Pencil code: Block code for a text world, in: Proceedings of the 14th International Conference on Interaction Design and Children, IDC '15, Association for Computing Machinery, New York, NY, USA, 2015, p. 445–448. doi:10.1145/2771839.2771875.

[7] F. Hermans, Hedy: A gradual language for programming education, 2020, pp. 259–270. doi:10.1145/3372782.3406262.

[8] Y. N. Srikant, P. Shankar, S. Aggarwal, M. Kumar, Debuggers for Programming Languages, CRC Press, 2008, p. 295–298.

[9] D. Fuller, Virtual workshop (2014).
URL https://cvw.cac.cornell.edu/Profiling/debugging_postmortem

[10] Atari, Atari 2600 manual: Basic programming (1979).
URL https://archive.org/details/Basic_Programming_1979_Atari_US/mode/2up

[11] Alex (Apr 2022). [link].
URL https://www.learncpp.com/cpp-tutorial/using-an-integrated-debugger-the-call-stack/

[12] D. Fuller, Virtual workshop (2014).
URL https://cvw.cac.cornell.edu/Profiling/debugging_runtime_symbolic

[13] J. Engblom, A review of reverse debugging, in: Proceedings of the 2012 System, Software, SoC and Silicon Debug Conference, IEEE, 2012, pp. 1–6.

[14] M. Afaneh, Debugging firmware with gdb (May 2019).
URL https://interrupt.memfault.com/blog/gdb-for-firmware-1

[15] C. Smith, gdbgui (2016).
URL https://www.gdbgui.com/

[16] GDB, Gdb: The gnu project debugger (1986).
URL https://www.sourceware.org/gdb/

[17] S. Avenwedde, A hands-on tutorial for using the gnu project debugger (Jan 2021).
URL https://opensource.com/article/21/1/gnu-project-debugger

[18] Google, Chrome devtools (2016).
URL https://developer.chrome.com/docs/devtools/overview/

[19] Mozilla, Firefox developer edition (2022).
URL https://www.mozilla.org/en-US/firefox/developer/

[20] Microsoft, Ide and code editor for software developers and teams (May 2022).
URL https://visualstudio.microsoft.com/

[21] E. Foundation, The community for open innovation and collaboration: The eclipse foundation (May 2022).
URL https://www.eclipse.org/

[22] Jetbrains, Pycharm: The python ide for professional developers by jetbrains (2010).
URL https://www.jetbrains.com/pycharm/

[23] Code.org, Spelling bee 6: Course 1.
URL https://studio.code.org/s/course1/lessons/11/levels/6

[24] R. Chmiel, M. C. Loui, Debugging: From novice to expert, in: Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education, SIGCSE '04, Association for Computing Machinery, New York, NY, USA, 2004, p. 17–21. doi:10.1145/971300.971310.
URL https://doi.org/10.1145/971300.971310

[25] M. Ahmadzadeh, D. Elliman, C. Higgins, An analysis of patterns of debugging among novice computer science students, in: Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education, ITiCSE '05, Association for Computing Machinery, New York, NY, USA, 2005, p. 84–88. doi:10.1145/1067445.1067472.
URL https://doi.org/10.1145/1067445.1067472

[26] B. W. Kernighan, P. J. Plauger, Chapter 2 Expression, McGraw-Hill, 1978.

[27] F. Hermans, A. Swidan, E. Aivaloglou, M. Smit, Thinking out of the box: Comparing metaphors for variables in programming education (2018). doi:10.1145/3265757.3265765.

[28] Ace, The high performance code editor for the web (2022).
URL https://ace.c9.io/

[29] J. Resig, Redefining the introduction to computer science (Aug 2012).
URL https://johnresig.com/blog/introducing-khan-cs/

[30] Codecademy, Codecademy: Learning python (2020).
URL https://www.codecademy.com/courses/learn-python-3/lessons/python-hello-world/exercises/comments

[31] CodeCombat, Coding games to learn python and javascript (2013).
URL https://codecombat.com/play/level/dungeons-of-kithgard

[32] Microsoft, Javascript with syntax for types. (Mar 2022).
URL https://www.typescriptlang.org/

[33] S. Graham, Skulpt.org (2015).
URL https://skulpt.org/

Appendix

In this section, a list of discussions and merged pull requests for Hedy can be found during the research period from January 1st 2022 to June 30th 2022. The list exists in two parts. The first part is regarding the code of this research project. The second part of this list is all code written for Hedy during the research but not directly relevant to this research. The items in the second list are characterized using an asterisk as a bullet.

Discussion contributions:

- **Description**: Announced the functionality of the debugger.
  `https://github.com/Felienne/hedy/discussions/2704`
- **Description**: Showed initial design of debugger to receive feedback.
  `https://github.com/Felienne/hedy/discussions/2483`

* **Description**: Presented two design ideas to return the palette.
`https://github.com/Felienne/hedy/discussions/1835`

Pull request contributions:

- **Description**: First version implementation of variable view.
  `https://github.com/Felienne/hedy/pull/2122`
- **Description**: Front-end improvements for variable view and added color-coding for variable types.
  `https://github.com/Felienne/hedy/pull/2241`
- **Description**: Hide variable view when there are no variables in a program
  `https://github.com/Felienne/hedy/pull/2762`
- **Description**: Disable variable view in level 1
  `https://github.com/Felienne/hedy/pull/2736`
- **Description**: Bug fix where the variable header would show up unwanted when running Turtle code.
  `https://github.com/Felienne/hedy/pull/2419`
- **Description**: Bug fix where the user was not able to log in anymore.
  `https://github.com/Felienne/hedy/pull/2184`
- **Description**: Implementation of the step-by-step debugger, also turning feature flag off for the variable view. Lastly, it is possible to ignore certain lines using a breakpoint.
  `https://github.com/Felienne/hedy/pull/2498`
- **Description**: Adding margin between debugger button and edit button.
  `https://github.com/Felienne/hedy/pull/2789`
- **Description**: Debugger would not work when sharing code.
  `https://github.com/Felienne/hedy/pull/2788`

- **Description**: Switching order of execution control buttons.
  https://github.com/Felienne/hedy/pull/2733

* **Description**: Re-implemented the palette containing the list of commands per level by adding a new small button.
  https://github.com/Felienne/hedy/pull/1928

* **Description**: Adding 11 colors to use in Turtle.
  https://github.com/Felienne/hedy/pull/2456

* **Description**: Fixing indentation in text for example code that failed to run in levels 3, 14 and 15.
  https://github.com/Felienne/hedy/pull/2421
  https://github.com/Felienne/hedy/pull/2426

* **Description**: Fixing spelling mistakes in the text.
  https://github.com/Felienne/hedy/pull/1811
  https://github.com/Felienne/hedy/pull/1809
  https://github.com/Felienne/hedy/pull/1751
  https://github.com/Felienne/hedy/pull/1749