Universiteit Leiden
The Netherlands

Opleiding Informatica

Continuing levels of the programming language Hedy

Laura Ottevanger

Supervisor:
Felienne Hermans

BACHELOR THESIS

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

22/06/2021

Abstract

The current way in which programming language are learned, is not optimal. Students are usually thrown into the deep end by being taught too many concepts at the same time. The programming language Hedy tries to solve this problem by gradually learning a student how to program. It works by making levels and adding new concepts in each level. At the start of this thesis, only a couple of concepts were introduced in Hedy. This paper shows which concepts have been added in each level, how to add levels to Hedy and the test results of the added levels.

Contents

1 Introduction

When learning how to program, one usually gets thrown into the deep end. When I started Computer Science without knowing a single thing about programming, it was quite overwhelming. In the first week we had to make a pyramid out of printed stars (*). For the students with previous knowledge about programming, this exercise is rather simple, they made the pyramid within 5 minutes, but I first had to learn how to print a single star (*), after that how a loop works and what that weird variable i is and eventually, after making quite some infinite loops, I made a nice pyramid out of stars. I had to close 10 infinite loops as they were running all at the same time as the concept of the compiler and how to run a program was new as well.

There is a possible solution to this problem and that is learning how to program gradually. A student can be given a tool to execute code and learn a couple of functions. They can test it out and go on to the next level once they are certain that they understand the level. This is what the programming language Hedy [Her20] does.

In this bachelor thesis for the Leiden Institute of Advanced Computer Science, supervised by Felienne Hermans; I will explain which concepts I have introduced to Hedy, how levels are being made and what resulted out of these levels.

1.1 Context

Hedy runs on `https://hedycode.com/`. It has several levels, in each level, another concept is added to the list of known concepts. When a student opens the first level, some example code is on the screen. It includes a small piece of text that informs the student what they can do in this level. The student can execute the example code and edit it to their liking. When a student has sufficient knowledge of the current level, they can continue to the next level.

At the start of the thesis, a couple of concepts were already introduced in Hedy:

- Basic in- and output
- Variables
- Syntax of a string
- If else statements
- Basic for loop
- Basic arithmetic calculations

When code is being run on the site, the code is being converted to Python, unbeknownst to the student. They only see the output that their code gives and possible errors.

1.2 Contributions

The contributions made were the addition of levels 8 to 22 to Hedy. The concepts introduced are visible in 1. The levels went live almost directly when they were finished. This resulted in a big data-set of user-programs on this levels. These were evaluated an possible changes and additions to these levels and Hedy are given in this thesis.

1.3 Thesis overview

This chapter contains the introduction; Section 2 includes explanation about Gradual Learning and about Hedy itself and the related work; Section 3 explains how a level can be implemented; Section 4 describes which levels have been implemented; Section 5 explains about the used evaluation methodology on the levels; Section 6 discusses the results of the evaluations; Section 7 talks about the difficulties in implementing the levels and testing them; Section 8 talks about conclusions from the tests and the conclusions made out of the total process of implementing the levels; Section 9 explains the possibilities that still can be implemented for in Hedy.

2 Background

2.1 Gradual Learning

Hedy teaches how to program with the use of gradual learning. Gradual learning is a learning process where a student learns something, usually a language, in multiple different steps, new concepts and grammar are introduced gradually.

Currently, programming languages are being taught by introducing a lot of concepts to the learner. They are being taught about grammar, syntax, control flow, coding and debugging and usually most of it is taught in the first week.

When one learns a normal language, for example Latin, they start with learning some basic words and eventually they learn a sentence like: “Jupiter is a God, Jupiter is a father”. Every week, they learn some more words and some more grammar to eventually learn an entire language. This is how one learns a language gradually.

Gradual learning is not widely implemented when teaching programming language. In 2015, a start has been made to teach students JavaScript using an assisted teaching model [CO15]. There are not many references to this work on the internet currently. The teaching tool that the writers of the paper mentioned has not been widely spread. There are a couple more teaching tools to gradual learn a single language. More about these tools will be explained in Section 2.2.

Hedy is (one of) the first programming language(s) that teaches a programming language on itself. It teaches the language Hedy. It is mainly based on Python and builds towards the Python language. It is also more easily available as the site is a free tool that is available to anyone.

2.2 Related work

Quite some teaching tools exist that have the intention to learn people how to program. In this Section we will discuss some existing teaching tools that learn users on how to program.

2.2.1 Block-based languages

One of these tools are the block-based programming languages. A lot of the tools for beginning programmers are actually Block-based languages as they are visually easier to understand than text-based languages.

These tools are very popular with children and teens. Some examples are Scratch [J. 04], Waterbear, Alice [UIG95] and Snap! Build Your Own Blocks [B. 14]. Block-based programming languages provide a lot of blocks to the user. Each block corresponds with a certain command. the user can drag blocks under each other to execute multiple commands, in this way, a lot of "functions" can be made.

There are multiple teaching tools that use the idea of blocks. There is the program Alice [UIG95], in which programs are created that make a small animation. The blocks provided are to move the characters or to make sound.

Scratch [J. 04] is more similar to actual programming. The commands that the blocks provide are more similar to programming languages like Python. An example of a Scratch program is shown in Figure 1. When the flag is pressed in this game, 10 times 10 steps are done, a variable called "steps" is set to 100. When space is pressed, the program will say "Hello!" if the flag has been pressed before space and will think "Hmm..." if the flag hasn't been pressed. The functions in these block-based programs are usually executed when specific keys or buttons have been pressed, this makes it a very convenient tool to make simple games. These block-based languages are very convenient to teach a user about execution order and functions. Unfortunately, when trying to make a program, no explanation is shown about how to make a program or what every single block does, learners still get thrown into the deep end but the step to start programming is easier as these tools are visually easier to understand than text based coding like Python. Scratch is a very popular tool but without any previous knowledge of programming, it's quite difficult to start and make a nice program.

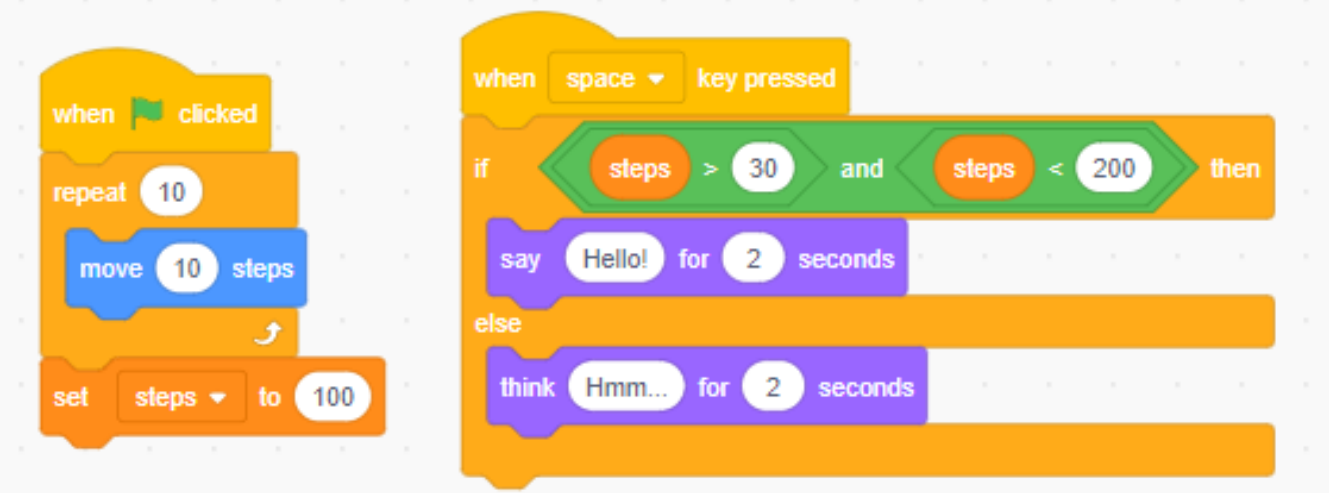

Figure 1: A small Scratch program

2.2.2 Games

There are also a lot of games to learn how to program. The problem with these games is usually that they are either too boring, or too difficult. These are the categories that these games fall into. The first one has for example: "Learn Programming: Python" this has a good introduction to Python but is more like an interactive Textbook. It learns some basic knowledge but does not have challenges in any way. A famous example that is too difficult is "Human Resource machine". The problem with this game is that it includes some logic puzzles for gamers who already have some previous knowledge about programming. If you start the game without any previous knowledge, you will have a very hard time and might need to search for the answers on the internet. Human Resource machine does introduce concepts gradually. However, it doesn't truly teach a language, it just teaches execution order and provides the gamer with some logic puzzles. Another game is TIS-100. This learns the basics of Assembly. Assembly is a bit too difficult to start with so the game in not truly accessible to beginning programmers.

In the paper on learning how to program with games and programming contests [Com16], a long list of aspects to make a programming game successful as an education tool has been made. They notice in this paper that it is difficult to include all of these concepts in one game as some concepts are better for certain types of games than others. This makes creating an educational programming game quite difficult. It has been noted in a paper on the effectiveness of games on learning how to program [Rug13], that cognitively complex concepts are easier to understand when learned with a game.A deeper analysis on the effectiveness of video games on the computational thinking skill that is necessary to learn how to program, can be found in the paper by Felienne Hermans, Giulio Barbero and Marcello A. Gómez-Maureira [Bar20].

2.2.3 Mini-languages

Mini-languages are simple languages to provide insight in the basic idea of programming and algorithmic thinking. These mini-languages don't build towards a certain existing programming

language but are languages on itself. Each language has a specific (limited) amount of instructions based on the language. There is an example with a robot where it can walk, put stuff in a bag and place it back. This Robot is called Karel the Robot [Pat00]. Based on these instructions, a user can make a small program where it can walk to something, put it in his bag and walk back. The user is limited to the instructions: turn on, turn left and turn right, move, pick up a beeper, put down a beeper and turn off. The user learns the basics of typing a program and learns about execution order.

A big problem with these types of mini-languages, is that most of them don't correspond with a programming language. As mentioned before, the set of instructions is specific for the mini-language itself. They might use the action "Move()" but students don't know the code behind this action. Students also don't learn how the specific instructions work. When one uses the "Move(2)" action, they see the robot move two steps on the screen. It might not be clear that it the move action might be using a for loop to walk the two steps, for example. The programming knowledge they learn is very specific for that certain mini-language. Stepping to a "real" programming language will be really confusing as they will not see the commands they have learned in the mini-language, but complete new ones. Furthermore, students might not understand the programs they have build. In the paper by Jean Salac and Diana Franklin on the analysis between understanding programming concepts and using the concepts in programs [Sal20], the conclusion is made that students are more likely to understand a single concept when they used it in their program but the correlations aren't big enough to show that students actually understand a concept better when learned. These two points are major flaws when using mini-languages to learn how to program.

2.2.4 Step-based languages

The last kind of teaching tools are step-based teaching tools. These are tools that learn a specific language in steps. Hedy [Her20] is such a tool, more will be explained in Section 2.3. For Java there is a tool called ProfessorJ [GF03]. It offers a simplified interface to the Java Compiler. There are multiple levels: beginner, intermediate, intermediate+access, advanced, full, dynamic. Each level is a subset of Java, when one goes to a higher level, more parts of Java are added. When a learner arrives at the level full, they know almost all concepts of Java. ProfessorJ is considered gradual in some aspects as it introduces new concepts every level. However, the syntax doesn't change, it is Java syntax that is being taught to the learner from the start. This makes ProfessorJ not truely gradual.

2.2.5 Difference between Hedy and existing tools

There is a big need for gradual teaching tools or languages. This is because gradually teaching a programming language has not been widely implemented yet. The mini-languages do not teach a certain language but they do learn some execution order and how to program some basic code. This, however can't be compared to learning an entire language. The discussed step-based based learning tool, ProfessorJ isn't used a lot anymore. The paper on the step-based language ProfessorJ has only been cited 45 times. Furthermore, the paper on ProfessorJ was published in 2003 and the authors haven't published anything about ProfessorJ after their first publication. It is safe to assume that ProfessorJ isn't used a lot anymore. As there are very few gradual teaching tools or languages, and ProfessorJ is one of the biggest that existed before Hedy, we can assume that

gradual teaching isn't implemented a lot anymore. From the paper by Walter Cazzola and Diego Mathias Olivares [Caz16] on gradual learning, we learn that gradual learning has some promising aspects to teach a programming to students. They also notice that there is a lack of teaching tools to learn how to program gradually. The lack of gradual teaching tools is a big "gap in the market". This is what Hedy fills in.

There are a couple more features that make Hedy unique in the world of programming teaching tools. As mentioned before, Hedy can be considered as almost the only gradual language. It is more gradual than ProfessorJ as it also introduces the syntax gradually. ProfessorJ does not do this. Secondly, Hedy is not just a"textbook", it doesn't just introduce a concept and move on, it also has challenges and exercises for learners, this makes Hedy not just a teaching tool that teaches something new every level, but it ensures that students need to actively use the concepts that they just learned to take on the challenge. Some of the existing programming games also try to use this concept but these games do not teach a language gradually, they are more focused on interactively learning students computational thinking and are not concerned with teaching syntax. Finally, because Hedy can be executed in browser, no programs need to be installed, this makes Hedy more easily available to learners than programs that must be bought or installed, like the games, ProfessorJ and the mini-languages. We can conclude that Hedy is a unique in the current world of programming teaching tools and languages.

2.3 Hedy

As mentioned before, Hedy is a programming language that is publicly available. It is open-source and anyone is allowed to contribute. For example, one can send in a translation of the text so the tool can be used by as many people as possible. More information on Hedy can be found in the ICER paper on Hedy [Her20].

A level consists of a couple of parts. In Figure 2 a level is shown. First of all, there is the explanation about what changed each level (1). It explains what the new changes do and how to use it. Second, there is example code to demonstrate how the changes work (2). This example code can be executed. And the output will be shown on the right side of the screen (3). For every level, more explanation is put on the left side of the screen (4). When the try button is pressed (5), some example code for the extra explanations will replace the current code (4). The user can change the example code to practice with the new changes and even replace it entirely with their own program.

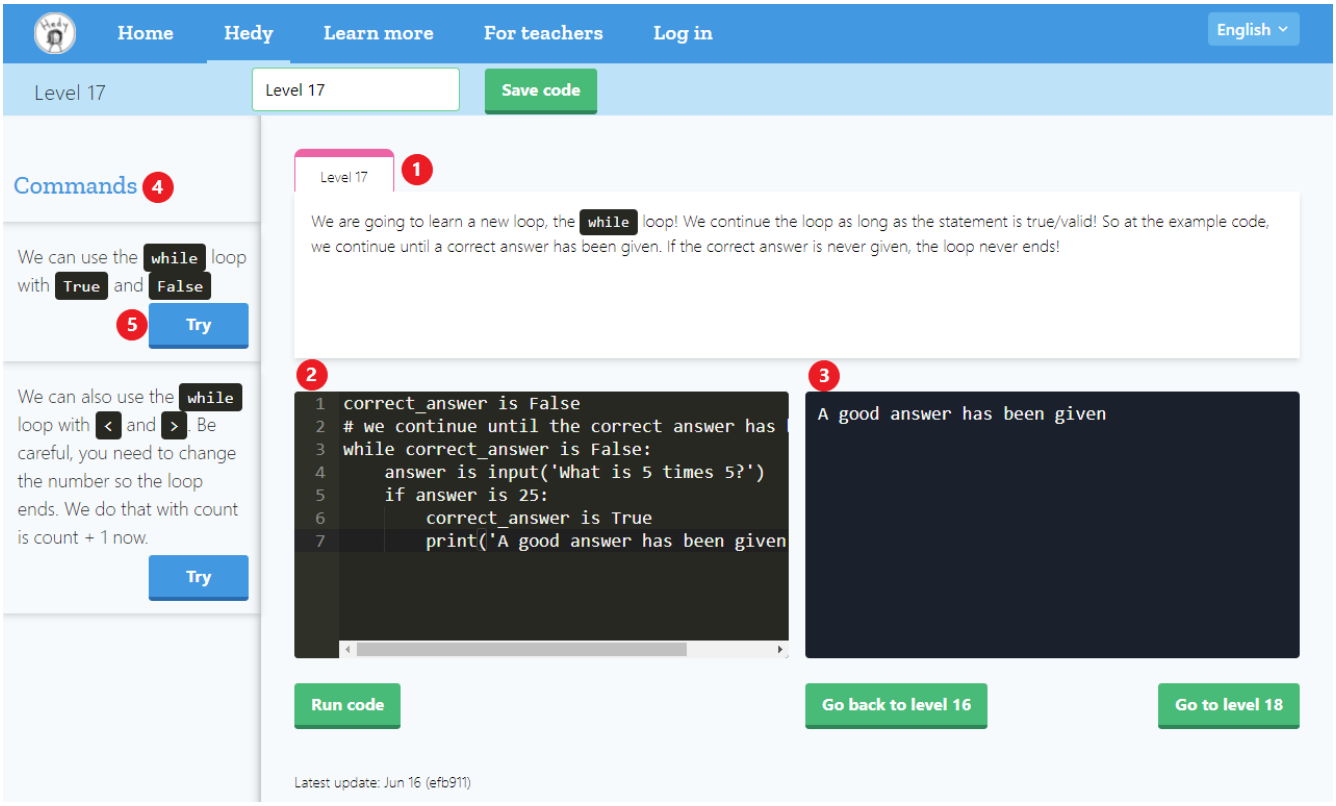

Figure 2: Level 17 of Hedy. 1: Explanation on the level, what changes and how to use it. 2: Editable example code. 3: Output of the program. 4: Extra example code and additional explanations. 5: Button to place additional example code in the edit window

When the user presses the execute button, the code will be parsed using a Lark grammar file. It checks if the code if valid grammar, then it gives the variables of that sentence to a Python function with the same name. For example, when the following equality check comes in: "hello is world" then the Lark file takes hello as arg[0], world as arg[1] and sends it to the equality_check function in Figure 3. This code returns a valid Python string. These strings are gathered and executed and the result will be shown in the window on the right. This happens in a matter of milliseconds so the user will just see their output directly.

```
def equality_check(self, args):
    return f"{args[0]} == {args[1]}"
```

Figure 3: The Python code to accept an equal statement

2.3.1 Existing levels

Before the start of this thesis, 7 levels of Hedy existed already. Each level either changed some syntax or introduced new commands. The following things were changed per level.

In level 1, the learner learns in- and output. They learn “print”, “ask” and “echo”. “Ask” corresponds with the term “input” in Python. The syntax for these three terms is for example:

```
print hello world
```

There are no quotation marks and no brackets necessary yet.

In level 2, the learner gets introduced to variables. A variable is assigned by the word “is”. For example:

```
    hello is world
```

They can assign a list and get a random item out of the list.

In level 3, the syntax of “print” changes a little. Everything that needs to be printed now needs a single quotation mark around it. For example:

```
print ‘hello world’
```

In level 4, the “if” statement is introduced. Equality checks also done using the “is” word. So now an equality check can look like this:

```
if hello is world
```

In level 5, a first version of a loop is being introduced; the “repeat” statement. The entire “repeat” statement is on one line. For example:

```
repeat 3 times print ‘Hello world’
```

In level 6, basic arithmetic is introduced, students can now use plus, minus and multiply in “print” statements. For example:

```
print ‘5 times 5 is ’ 5 * 5
```

In level 7, indentation now makes multiple lines in an “if” or “repeat” statement possible. This is done by introducing indentation and putting the statement that needs to be repeated underneath the “repeat” statement. For example:

```
repeat 5 times
    print ‘Hello World’
    print ‘print it 5 times’
```

3 Implementation of new levels

3.1 Required changes

In this section, each file that needs to be changed to implement a single level is described. The test files is not included in this chapter as these will be discussed in Section 3.2.

3.1.1 Level defaults yaml file

In the level defaults files, the text for each level is added. The text exists of the explanation and example code for each level. A user can use Hedy in several different languages. In Figure 2, the program is in English. To accommodate for this, there is a unique file for every translation. In these files the explanation and example code are added. Multiple example codes can be added for each level.

3.1.2 Course\Hedy yaml file

In the course\Hedy yaml file is defined which levels are introduced. These files are a list with level numbers. When a level is added, the number is added to the list for each language that the level has been implemented in.

3.1.3 Grammar file (Lark)

In the grammar file, every allowed command and the requested syntax for each command is visible. When a new level is added, the grammar file of the previous level is copied. After this, new commands can be added to the list or current ones can be changed in this new file. It is important to know how the Lark grammar works. The rules that are allowed to be used in the grammar file is defined in the Rules section on the following site: https://Lark-parser.readthedocs.io/en/latest/grammar.html.

3.1.4 Hedy.py

The Hedy.py file contains the functions to convert the grammar to correct Python. First of all the list of allowed commands is in this file. This list is called "commands_per_level". Adding allowed commands to this list doesn't change the operation of the level but it makes sure that when the learner makes a typing error while using the level, it shows what is supposed to be there.

In the "transpile_inner" function the call to the correct convert to Python function is defined. For each new level, a new "elif" statement with the correct function needs to be added.

A new class needs to be made for each level. The syntax for these classes is the following: "ConvertToPython_<levelnumber>(ConvertToPython_<previous function>)". If a new function is defined in the Lark or if a function changes, the function needs to be added to this class. The functions in these classes take the input from the Lark files. This input are the arguments that are given to the function. A string is returned by the function. This string is the corresponding Python code with the given arguments in the Lark file.

When a new function has been defined, it needs to be added to a couple of functions in these files. The following functions are relevant:

- "AllAssignmentCommands": This function returns only variable assignments and places where variables are accessed so these can be excluded when printing. All functions where a variable is being accessed, needs to be put in this list.

- "Filter", "IsValid", "IsComplete": These functions check if all arguments that need to given for a certain function exist. For example, if a statement has the following syntax:

  ```
  for i in range(0,)
  ```

  This code misses an ending variable to the loop. This function filters incomplete call and returns when there is an error. If a statement can have missing variables, like the print statement, it must be defined in "IsComplete".

3.1.5 DESIGN.md

In the design file, there is an explanation of what changes on each level. When a new level is added, it is convenient to put the information about what changes in the levels. This is to show the contributers of Hedy what concepts have already been introduced.

3.2 Testing levels in development

When a level is made, it needs to be tested. The first step to test a level is to make a testfile for it. The correct syntax for a testfile is the following: "tests_level_<level number>.py". When a new level is made, the test file of the previous level can be copied. The level numbers need to be changed to the correct level. The line that shows the level number is the following:

```
result = hedy.transpile(code, <level number>)
```

For each new level, new tests must be designed and entered into the new test file. When a change is made that changes the syntax of the previous levels, the existing tests must be changed to accommodate that change. There is also a file called "test_multiple_levels". The level number can be added to this file. If anything changes concerning lists, a change to the existing function must be made to implement the changes.

The tests can be run by entering the following command in the terminal:

```
python -m unittest discover -s tests
```

This needs to be done while being in the environment. More information about this is found in the CONTRIBUTING.md file on the Hedy Github.

4 Implemented levels

The levels that were added are level 8 to 22 are shown in Table 1, they will be explained in more detail in Section 4.1 and in Section 4.3.

Level number	Concept
8	“For” loop
9	Colon after “if”, “else”, “for”
10	Nesting
11	Round brackets after “if”, “elif”, “for”, “input”
12	Square brackets in lists
13	Booleans
14	“And” and “or”
15	Comments
16	“Less than” and “greater than” signs
17	“While” loop
18	Get an item from a list
19	Loop through a list and length of list
20	Change “is” to “=” and “==”
21	“not equal” sign
22	“less than or equal” and “greater or equal” signs

Table 1: The list of concepts that were introduced in levels 8 to 22

4.1 Concepts

The concepts that we wanted to add started with implementing loops. Until now, only “repeat‘” was introduced. For level 8 the “for” loop was introduced, it explains how a “for” loop works and how to use the variable that the “for” loop uses to loop. As “repeat” has already been introduced, the “for” loop wasn’t as big as a step as they just learned what a loop does.

After this, a small syntax change was introduced, this was the addition of the colon. This was a small syntax change. Some other syntax changes were also introduced to get closer to Python, for example the addition of brackets around “print” statements and lists.

The next step was introducing the “while” loop. To introduce this, some previous knowledge about comparison checks is necessary. The easiest way to introduce this was to introduce the “less than” and “greater than” signs and the “True” and “False” first. So the next additions were the introduction of “True” and “False” and “less than” (<) and “greater than” (>). While making this, the “and” and “or” statements were also introduced, users were able to use these in the “while” loop but this was not introduced in the examples of this level, because introducing it might be too difficult to learn it while learning what a “while” loop is.

In between these previous levels, it became apparent that that the example code got a bit long and commenting the code was not possible yet so the introduction of comments would be a nice addition to help explain long example codes.

While working on Hedy and testing on some users, it became apparent that the lists weren’t explained a lot so the next levels repeated some explanation about the lists. It also showed how to use the “for” loop with getting items from lists.

The final part was changing the syntax from the word is to "==" in equality checks and "=" to variable assignment. When these were introduced, the "not equal", "less than or equal" and "greater than or equal" were able to be introduced.

4.2 Order

The order of the introduction of concepts has been changed during production. The original plans were to introduce the "for" loop first, then learning the colon, nested statements, brackets and finally introducing the "=" and "==". After some discussion it became clear that the concept of changed the "is" to the single and double equal signs needed to be introduced later as it is quite a difficult change. Syntax changes like this should be introduced after all the other items have been introduced. At the point of the discussion some levels were already made so the levels needed to be reordered. The reordering caused some difficulties which will be explained in Section 7.1.

4.3 Changes

In this section, the changes on each level will be shown. It is important to know that every level is based on the code of the previous one. When a new concept is introduced, a statement is added to the Lark file and a Python function is made where the Lark statement can refer to. If an existing function has some changes, the old one is changed to accommodate the changes.

4.3.1 How to read the Lark files

The Lark files can be read the following way; if a part is in brackets, it is a literal part. If the word in brackets is "Hello ", it expects the word "Hello" with a capital H and a space after it. Capital words are certain items, for example NAME or NUMBER. These are what they say they are, a name or a number. Small words without brackets are rules, they are defined in the rest of the file. In the following code examples not all rules will be shown. If a part is in brackets with an "or" sign in between (—) it means either the left side of the "or", or the right side must occur. For example (NUMBER — var) the expected item is either a number or a variable. A part in brackets can have a plus (+), multiply (*) or question mark (?) behind it, If it has a plus (+) behind it, it means that the part in the brackets has to occur once but can occur more times. If it has a multiply (*) behind it, it means that it can occur zero or more times. If there is a question mark (?) behind it a statement in brackets, it means that the statement can occur zero or only one time. There is also the possibility to fix the amount of times an instance can occur but that will not be used in the following examples. The _EOL means that an end of line is expected, this means that the next expected code is on the new line.

4.3.2 Level 8

Level 8 introduces the "for" loop. The "repeat" statement has already been introduced at this point so the "for" loop was a small step. In the Lark file the expected grammar has been added. An correct example of the "for" loop is the following:

```
for i in range 1 to 5
```

After this it expects an end of line. Then multiple “commands” can be added, separated by end of lines. It ends with an end-block. the command rule is the list of all possible commands, it includes for example: print, ask, if. The “end-block” is to show the end of a indented block.

```
//new for level 8
for_loop: "for " (NAME | var) " in " "range " (NUMBER | var) " to " (NUMBER | var) _EOL (" "+ command) (_EOL " "+ command)* _EOL "end-block"
```

Figure 4: The changes in the Lark file for level 8

The Python program takes all the arguments that were given in the Lark file. The arguments consist of every part that is not in quotation marks in the Lark file. The Python program turns these arguments into correct Python. It adds brackets and adds casts to integers so the given numbers will always be cast to int to prevent problems when the numbers are being added as strings.

The Python programs adds one to the last given argument, this is the number where the loop ends. This is, because usually in Python, “range 1 to 3” means 1 and 2. It is not really clear that it loops to one number less than the last argument. To prevent this confusion, the plus one has bee added so 1,2,3 are included if “range 1 to 3” is given.

```
class ConvertToPython_8(ConvertToPython_7):
    def for_loop(self, args):
        args = [a for a in args if a != ""]  # filter out in|dedent tokens
        all_lines = [indent(x) for x in args[3:]]
        return "for " + args[0] + " in range(" + "int(" + args[1] + ")" + ", " + "int(" + args[2] + ")+1" + "):\n"+"\n".join(all_lines)
```

Figure 5: The changes in the Python file for level 8

4.3.3 Level 9

Level 9 introduces the colon (:). The colon is now required to be added after the if, else and for loop. As this change is a small one, the “elif” (else if) statement is also introduced. The “elif” can be added after an if and can be added as many times as possible.

```
// addition to level 9, the :
elses : _EOL (" ")* "else"":" _EOL (" "+ command) (_EOL " "+ command)* _EOL "end-block"
ifs: "if " condition ":" _EOL (" "+ command) (_EOL " "+ command)* _EOL "end-block"//'if' cannot be used in Python, hence the name of the rule is 'ifs'
elifs: _EOL (" ")* "elif " condition ":"_EOL (" "+ command) (_EOL " "+ command)* _EOL "end-block"

for_loop: "for " (NAME | var) " in " "range " (NUMBER | var) " to " (NUMBER | var) ":" _EOL (" "+ command) (_EOL " "+ command)* _EOL "end-block"
```

Figure 6: The changes in the Lark file for level 9

```python
class ConvertToPython_9_10(ConvertToPython_8):
    def elifs(self, args):
        args = [a for a in args if a != ""]  # filter out in|dedent tokens
        all_lines = [indent(x) for x in args[1:]]
        return "\nelif " + args[0] + ":\n" + "\n".join(all_lines)
```

Figure 7: The changes in the Python file for level 9

4.3.4 Level 10

Level 10 allows the user to put loops and "if" statements in each other. It is also called nesting. As this was already possible in the previous levels, no code changes were made. This level was solely to explain how nesting works.

4.3.5 Level 11

Level 11 changes ask to input and introduces round brackets, it also changes the syntax of the for loop. The term "ask" changes to "input", this is because it is called "input" in Python. It also adds brackets to the "input" statement. The "input" statement is now the exact same as Python. The "print" statement also changes so it is the same as the "print" statement in Python. Finally the syntax of the "for" loop changes. The correct syntax for the "for" loop was:

```
for i in range 1 to 3:
```

And it changes to:

```
for i in range(1, 3):
```

The" to " part has been removed and brackets have been added. The "for" loop now changed to the exact same syntax as the "for" loop in Python. The only difference with Python is that it still loops to one number higher than a Python loop does, as is explained in level 8.

```
//for level 11 brackets were added around print and for and ask changed into input
print : "print(" (quoted_text | list_access | var | sum) (" " (quoted_text | list_access | var | sum))* ")"
input : var " is input("  (quoted_text | list_access | var | sum) (" " (quoted_text | list_access | var | sum))*  ")"
for_loop: "for " (NAME | var) " in " "range(" (NUMBER | var) (", "|",") (NUMBER | var) "):" _EOL (" "+ command) (_EOL " "+ command)* _EOL "end-block"
```

Figure 8: The changes in the Lark file for level 11

The changes in the code are only the definition of the "input" statement. It is the same as how the "ask" statement is defined in previous levels but the name of the definition changes. The "ask" statement is removed so it can't be used again.

```python
class ConvertToPython_11(ConvertToPython_9_10):
    def input(self, args):
        args_new = []
        var = args[0]
        for a in args[1:]:
            if type(a) is Tree:
                args_new.append(f'str({a.children})')
            elif "'" not in a:
                args_new.append(f'str({a})')
            else:
                args_new.append(a)

        return f'{var} = input(' + '+'.join(args_new) + ")"
```

Figure 9: The changes in the Python file for level 11

4.3.6 Level 12

Level 12 changes the syntax of a list, it adds the square brackets. Quotation marks need to be put around items in lists. In the Lark file everything that has something to do with lists changes to include the brackets and quotes. The new command is "change_list_item". While making this level, it became clear that there was no function yet to change an item in an list. This is what the "change_list_item" function does. The function has not been explained in this level as the syntax changes are quite difficult on itself.

```
?command: print
        | ifs (elifs)* elses?
        | input
        | for_loop
        | assign_list
        | list_access_var
        | change_list_item
        | assign

assign : var " is " sum | var " is " textwithoutspaces | list_access " is " sum | list_access " is " textwithoutspaces
assign_list: var " is [" quoted_text ((", "|",") quoted_text)+ "]"
list_access : var "[" (index | random | var) "]"
list_access_var : var " is " var "[" (index | random | var) "]"
equality_check: (textwithoutspaces | list_access) " is " (textwithoutspaces | list_access)

//new for level 12, assignment of list[i]
change_list_item : var "[" (index | var) "] is " (var | textwithoutspaces)
```

Figure 10: The changes in the Lark file for level 12

The most important part of the Python code are the -1's. The minus one is done because the user

does not know that arrays start at zero. It was decided to make arrays start at one as it makes more sense for children to start counting at one instead of at zero.

```
class ConvertToPython_12(ConvertToPython_11):
    def assign_list(self, args):
        parameter = args[0]
        values = [a for a in args[1:]]
        return parameter + " = [" + ", ".join(values) + "]"

    def list_access_var(self, args):
        var = args[0]
        if not isinstance(args[2], str):
            if args[2].data == 'random':
                return var + '=random.choice(' + args[1] + ')'
        else:
            return var + '=' + args[1] + '[' + args[2] + '-1]'

    def list_access(self, args):
        if args[1] == 'random':
            return 'random.choice(' + args[0] + ')'
        else:
            return args[0] + '[' + args[1] + '-1]'

    def change_list_item(self, args):
        return args[0] + '[' + args[1] + '-1] = ' + args[2]
# Custom transformer that can both be used bottom-up or top-down
```

Figure 11: The changes in the Python file for level 12

4.3.7 Level 13

Level 13 introduces “True” and ”False” statements. There are no Lark changes in the file as the Lark grammar doesn’t care if the statement that is given in assigns and equality checks is a Boolean, a variable or string. This distinction will be checked in the Python function. This function checks the values of the parameters. If the value is “True” or “true” it will change the line to “ = True” and if it’s “False” or ”false” the it changes to “ = False”. The learner is also allowed to use “True” and “False” without the capital letters, it is not shown in the example code and explanation but it is allowed because it is a common mistake and it prevents confusing error messages.

```
class ConvertToPython_13(ConvertToPython_12):
    def assign(self, args):
        if len(args) == 2:
            parameter = args[0]
            value = args[1]
            if type(value) is Tree:
                return parameter + " = " + value.children
            else:
                if "'" in value or 'random.choice' in value:
                    return parameter + " = " + value
                else:
                    if value == 'true' or value == 'True':
                        return parameter + " = True"
                    elif value == 'false' or value == 'False':
                        return parameter + " = False"
                    else:
                        return parameter + " = '" + value + "'"
        else:
            parameter = args[0]
            values = args[1:]
            return parameter + " = [" + ", ".join(values) + "]"

    def equality_check(self, args):
        arg0 = wrap_non_var_in_quotes(args[0], self.lookup)
        arg1 = wrap_non_var_in_quotes(args[1], self.lookup)
        if arg1 == '\'True\'' or arg1 == '\'true\'':
            return f"{arg0} == True"
        elif arg1 == '\'False\'' or arg1 == '\'false\'':
            return f"{arg0} == False"
        else:
            return f"str({arg0}) == str({arg1})" #no and statements
```

Figure 12: The changes in the Python file for level 13

4.3.8 Level 14

Level 14 introduces “and” and “or”. “And” and “or” can be used in “if” statements, they are not allowed to be used in any other statements yet. The “and” and “or” statements allow a learner to reduce their amount of “if” statements. This can make the code more readable.

```
ifs: "if " (condition|andcondition|orcondition) ":" _EOL (" "+ command) (_EOL " "+ command)* _EOL "end-block" //'if' cannot be used in Python, hence the name of the rule is 'ifs'
elifs: _EOL (" ")* "elif " (condition|andcondition|orcondition) ":"_EOL (" "+ command) (_EOL " "+ command)* _EOL "end-block"

//new for level 14
andcondition: (equality_check|in_list_check) (" and " condition)*
orcondition: (equality_check|in_list_check) (" or " condition)*
```

Figure 13: The changes in the Lark file for level 14

The original ”and” and ”or” are being filtered out by the Lark file to see whether the ”and” or ”or” condition is being given. This causes that the Python functions needs to add the word ”and” or ”or” between the given arguments.

```python
class ConvertToPython_14(ConvertToPython_13):
    def andcondition(self, args):
        return ' and '.join(args)
    def orcondition(self, args):
        return ' or '.join(args)
```

Figure 14: The changes in the Python file for level 14

4.3.9 Level 15

Level 15 introduces comments. It is now allowed to comment a line. The expected grammar is a hashtag (#) and any text with spaces can be added after the hashtag.

```
//new for level 15
comment: "#" (textwithspaces)*
```

Figure 15: The changes in the Lark file for level 15

```python
class ConvertToPython_15(ConvertToPython_14):
    def comment(self, args):
        return f"# {args}"
```

Figure 16: The changes in the Python file for level 15

4.3.10 Level 16

Level 16 introduces the “less than” (<) and “greater than” (>) signs. It is now allowed to use them in conditions. Before this introduction, only equality checks and in list checks were allowed to be done in an “if” statement.

```
condition: (equality_check|in_list_check|smaller|bigger)

//new for level 16, smaller and bigger
smaller: (textwithoutspaces | list_access) " < " (textwithoutspaces | list_access)
bigger: (textwithoutspaces | list_access) " > " (textwithoutspaces | list_access)
```

Figure 17: The changes in the Lark file for level 16

```
class ConvertToPython_16(ConvertToPython_15):
    def smaller(self, args):
        arg0 = wrap_non_var_in_quotes(args[0], self.lookup)
        arg1 = wrap_non_var_in_quotes(args[1], self.lookup)
        if len(args) == 2:
            return f"str({arg0}) < str({arg1})"  # no and statements
        else:
            return f"str({arg0}) < str({arg1}) and {args[2]}"

    def bigger(self, args):
        arg0 = wrap_non_var_in_quotes(args[0], self.lookup)
        arg1 = wrap_non_var_in_quotes(args[1], self.lookup)
        if len(args) == 2:
            return f"str({arg0}) > str({arg1})"  # no and statements
        else:
            return f"str({arg0}) > str({arg1}) and {args[2]}"
```

Figure 18: The changes in the Python file for level 16

4.3.11 Level 17

Level 17 introduces the “while” loop. In previous levels, “True” and “False” have been introduced, so were the “less than” (<) and “greater than” (>) checks. These are used in the “while” loop. “And” and “or” conditions are also allowed in the “while” loop but this concept is not introduced in the explanation of the level.

```
//new for level 17, while loop
while_loop: "while " (condition|andcondition|orcondition) ":" _EOL (" "+ command) (_EOL " "+ command)* _EOL "end-block"
```

Figure 19: The changes in the Lark file for level 17

As the Lark file expects a condition after the ”while”, the condition of the ”while” will be put in

args[0]. All arguments will be unindented first to ensure the correct indent is being done for this while loop. The Python file places this condition after the "while" word. All the other arguments are the body of the while loop. The body gets indented once to ensure that the return value is correctly indented.

```python
class ConvertToPython_17(ConvertToPython_16):
    def while_loop(self, args):
        args = [a for a in args if a != ""]  # filter out in|dedent tokens
        all_lines = [indent(x) for x in args[1:]]
        return "while " + args[0] + ":\n"+"\n".join(all_lines)
```

Figure 20: The changes in the Python file for level 17

4.3.12 Level 18

Level 18 introduces no new functions and doesn't change any syntax. This level is to show the user again how to get specific item from a list. This has already been introduced in level 12 but it was a secondary goal in that level. This level repeats the explanation and shows some more examples on how to get a variable from the list.

4.3.13 Level 19

Level 19 initially only introduced how to get a variable from a list with the use of a "for" loop. At this moment no new code was necessary.

When testing the level, it became clear that when the user changed the amount of items in a list, they needed to change the last variable in the "for" loop every time. A solution for this problem was to introduce the "length" statement. In Python it is also known as "len". This "length" statement is allowed to be used in "print" statements, "input" statements, assignment, "for" loops, "less than" checks, "greater than" checks and list access statements. The statement gets the length from the list that it is called on.

```
print : "print(" (quoted_text | list_access | var | sum | length) (" " (quoted_text | list_access | var | sum | length))* ")"
input : var " is input("  (quoted_text | list_access | var | sum | length) (" " (quoted_text | list_access | var | sum | length))*  ")"
assign : var " is " length | var " is " sum | var " is " textwithoutspaces | list_access " is " sum | list_access " is " textwithoutspaces
for_loop: "for " (NAME | var) " in " "range(" (NUMBER | var | length)(", "|",") (NUMBER | var | length) "):" _EOL (" "+ command) (_EOL " "+ command)* _EOL "end-block"
smaller: (textwithoutspaces | list_access | length) " < " (textwithoutspaces | list_access | length)
bigger: (textwithoutspaces | list_access | length) " > " (textwithoutspaces | list_access | length)
list_access : var "[" (index | random | var | length) "]"

//new for level 19
length: "length(" var ")"
```

Figure 21: The changes in the Lark file for level 19

```
class ConvertToPython_18_19(ConvertToPython_17):
    def length(self, args):
        arg0 = args[0]
        return f"len({arg0})"

    def assign(self, args):
        if len(args) == 2:
            parameter = args[0]
            value = args[1]
            if type(value) is Tree:
                return parameter + " = " + value.children
            else:
                if "'" in value or 'random.choice' in value:
                    return parameter + " = " + value
                elif "len(" in value:
                    return parameter + " = " + value
                else:
                    if value == 'true' or value == 'True':
                        return parameter + " = True"
                    elif value == 'false' or value == 'False':
                        return parameter + " = False"
                    else:
                        return parameter + " = '" + value + "'"
        else:
            parameter = args[0]
            values = args[1:]
            return parameter + " = [" + ", ".join(values) + "]"
```

Figure 22: The changes in the Python file for level 19

4.3.14 Level 20

Level 20 changes the syntax of equality checks and assign statements. Before this level, an assignment was done by using the word "is". For example, a valid assignment was:

```
variable is True
```

We change the "is" to the equal sign (=). So a valid assignment in this level is now:

```
variable = True
```

Equality checks were also done with the word "is". A valid equality check was:

```
if variable is True:
```

This is changed by the double equality sign (==). A valid equality check is now:

```
if variable == True:
```

```
input : var " = input("  (quoted_text | list_access | var | sum | length) (" " (quoted_text | list_access | var | sum | length))*  ")"
assign_list: var " = [" quoted_text ((", "|",") quoted_text)+ "]"
assign : var " = " length | var " = " sum | var " = " textwithoutspaces | list_access " = " sum | list_access " = " textwithoutspaces
list_access_var : var " = " var "[" (index | random | var) "]"
equality_check: (length | textwithoutspaces | list_access | sum) " == " (length | textwithoutspaces | list_access | sum)
```

Figure 23: The changes in the Lark file for level 20

```
class ConvertToPython_20(ConvertToPython_18_19):
    def equality_check(self, args):
        if type(args[0]) is Tree:
            return args[0].children + " == int(" + args[1] + ")"
        if type(args[1]) is Tree:
            return "int(" + args[0] + ") == " + args[1].children
        arg0 = wrap_non_var_in_quotes(args[0], self.lookup)
        arg1 = wrap_non_var_in_quotes(args[1], self.lookup)
        if arg1 == '\'True\'' or arg1 == '\'true\'':
            return f"{arg0} == True"
        elif arg1 == '\'False\'' or arg1 == '\'false\'':
            return f"{arg0} == False"
        else:
            return f"str({arg0}) == str({arg1})"  # no and statements
```

Figure 24: The changes in the Python file for level 20

4.3.15 Level 21

In level 20 the equal sign has been introduced. This allows to introduce more symbols that use the equal sign. In level 21 the "not equal" sign (!=) is introduced. This is allowed to be used in condition checks.

```
condition: (equality_check|in_list_check|smaller|bigger|not_equal)

//new for level 21
not_equal: (length | textwithoutspaces | list_access) " != " (length | textwithoutspaces | list_access)
```

Figure 25: The changes in the Lark file for level 21

```python
class ConvertToPython_21(ConvertToPython_20):
    def not_equal(self, args):
        arg0 = wrap_non_var_in_quotes(args[0], self.lookup)
        arg1 = wrap_non_var_in_quotes(args[1], self.lookup)
        if len(args) == 2:
            return f"str({arg0}) != str({arg1})"  # no and statements
        else:
            return f"str({arg0}) != str({arg1}) and {args[2]}"
```

Figure 26: The changes in the Python file for level 21

4.3.16 Level 22

Level 22 introduces the "less than or equal to" (<=) and "greater than or equal to" (>=) signs. These can be used in condition checks. The "less than" (<) and "greater than" (>) signs have already been introduced in a previous level. This made the introduction of this level easier. These statements are introduced this late and not directly after the "less than" and "greater than" signs because the "equal" sign (=) has been introduced in level 20.

```
condition: (equality_check|in_list_check|smaller|bigger|not_equal|smaller_equal|bigger_equal)

//new for level 22
smaller_equal: (textwithoutspaces | list_access | length) " <= " (textwithoutspaces | list_access | length)
bigger_equal: (textwithoutspaces | list_access | length) " >= " (textwithoutspaces | list_access | length)
```

Figure 27: The changes in the Lark file for level 22

```
class ConvertToPython_22(ConvertToPython_21):
    def smaller_equal(self, args):
        arg0 = wrap_non_var_in_quotes(args[0], self.lookup)
        arg1 = wrap_non_var_in_quotes(args[1], self.lookup)
        if len(args) == 2:
            return f"str({arg0}) <= str({arg1})"  # no and statements
        else:
            return f"str({arg0}) <= str({arg1}) and {args[2]}"

    def bigger_equal(self, args):
        arg0 = wrap_non_var_in_quotes(args[0], self.lookup)
        arg1 = wrap_non_var_in_quotes(args[1], self.lookup)
        if len(args) == 2:
            return f"str({arg0}) >= str({arg1})"  # no and statements
        else:
            return f"str({arg0}) >= str({arg1}) and {args[2]}"
```

Figure 28: The changes in the Python file for level 22

5 Evaluation Methodology

In this Section, the Evaluation Methodology is explained. The testing plan is explained and the expected results are being explained.

5.1 Testing plan

The implemented levels went live on the Hedy website in batches. Whenever levels went live, users almost immediately made programs on them. This resulted in quite some collected data for each level.

The collected data consisted of some information about the user (if they were logged in); The username and email. It also collected the language in which they are running Hedy. Some information about the program they made like the actual code of the program, the amount of lines the program consisted of, on which level they made it, when they made the code, the error they received if they made a mistake and if it was test or demo code has also been recorded.

This resulted in a very big set of programs. Using Python pandas, I was able to retrieve the data from the file.

The first part of the testing plan was to remove the instances where no mistakes were made, as there should be no mistakes in the example and start code, these would also be filtered out. This was done by removing all the instances where '-' was received as an server error. This means that

there was no error given to the user.

This resulted in a set of all the programs that had a mistake in them and generated an error message for the users. This set could be separated so only the programs for a certain level were visible. The sets per level were small enough to evaluate manually. The programs were printed to the terminal and a manual read through them made common mistakes clear.

Whenever an interesting mistake was spotted, I adjusted the Python code that sorted the programs to count (and print) all examples of this occurrence. These results will be evaluated in Section 6.

5.2 Expected results

The expected results of the testing are that more mistakes are made when a syntax change has been introduced. These changes are more difficult than new introduced rules.

6 Evaluation Results

In this Section, we evaluate the programs that were made by users of Hedy. The Evaluation Methodology is found in in Section 5. We talk about the errors that were made in each level and how many times they occur. We also take a look at the commands that were used in each level. After evaluating the errors, some possible improvements to the levels will be given. Now on to the Evaluation.

6.0.1 Amount of programs analyzed

As mentioned before, the programs made by the users were collected in a very big data file. We will first separate these on programs made per level. In Table 2, the amount of programs that were made per level are shown. This table also shows how many programs caused an error and how many percent of the total amount of created programs caused an error.

Level number	Amount of programs with errors	Total amount of programs	Percentage wrong
8	4066	11076	36.71%
9	1736	6164	28.16%
10	2234	5009	44.60%
11	2284	3990	57.24%
12	1255	4894	25.64%
13	422	1254	33.65%
14	176	682	25.81%
15	98	480	20.42%
16	213	758	28.10%
17	447	1128	39.63%
18	236	624	37.82%
19	316	891	35.47%
20	62	143	43.36%
21	13	71	18.31%
22	43	117	36.75%

Table 2: Amount of all programs that were made per level and the amount of mistakes made per level

This data can be turned into a graph to show where the most mistakes are being made. The line in Figure 29 represents the percentage of programs with mistakes per level.

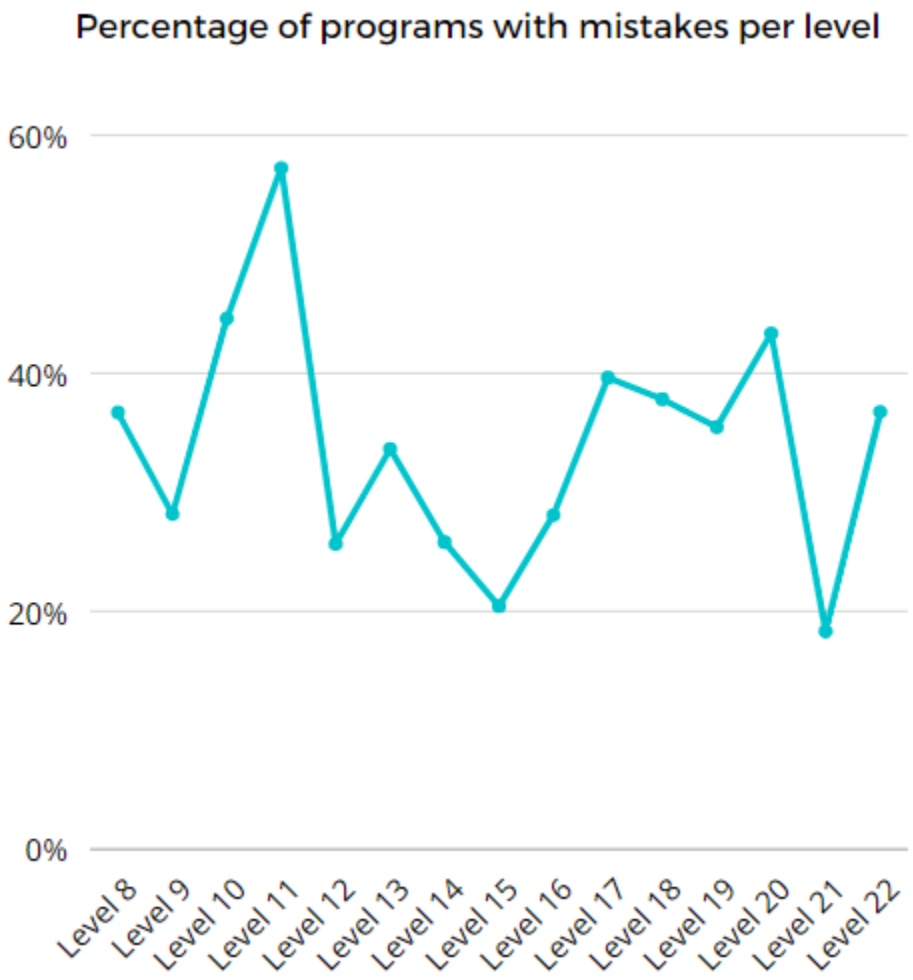

Figure 29: The graph of the percentage of programs with mistakes per level

In the following sections, there are amounts and percentages given for when a certain error is being made. A program that is executed with an error can be run multiple times by an user. It also occurs often that a user changes their program a little bit. It is important to note that the amount of errors given does not imply that these are unique programs or made by unique users.

6.0.2 Level 8

Out of the 11076 programs that were made for level 8, there were 5543 programs that included the word "for". Out of these, 1209 had an error in it. T

A lot of errors that were made with the actual "for" loop were indent mistakes. When they called the "for" loop, the next line was not indented and that resulted in an error. 827 programs (or 67% of all erroneous programs) had indentation errors.

There were also some syntax errors with the new "for" loop. As a reminder, the correct syn-

tax for the “for” loop in level 8 is

```
for i in range 1 to 10
    print i
```

The mistakes that appeared were the following

1. Already using brackets: for i in range(5,10)
2. Giving only one number in the range: for i in range 5
3. Switching the variable and the in: for in i range 1 to 10

Sometimes the word “in” was forgotten or there was only one number given for the range.

The “repeat” statement has also been used 374 times even though it has been removed in this level.

Another error that appeared at least in 11 different programs was that the Dutch users tried to use the Dutch word “tot” instead of “to”. A very small percentage of the users actually made this mistake but it is an interesting error as it might be not clear to user that the required word is “to” instead of “tot”.

The last error that is of note is that user try to use a “for” loop to loop through a list. Two examples that recurred in the user programs were the following:

```
    for i in listvariable
```

and

```
    for i in rock, paper, scissors
        print i
```

These errors might have occurred because it has not been shown how to loop through a list yet. This will be explained in level 19.

6.0.3 Level 9

There were 80 errors made that had correct code for level 8 but not for level 9. This means that these programs didn’t have the colon but had no further mistakes. There were more user programs that had the colon mistake but that had a different error in the code as well. In total 110 programs were made that used an if, else or for statement but didn’t use the colon.

The second most common error are indent errors. Either no indent was done or there were too much spaces as indent.

6.0.4 Level 10

Again, a lot of indent errors were made. This time it became apparent that there was a mistake in the start code. This resulted in 682 out of the 2234 errors. This is 30.53 %. Almost all the other errors that were relevant for this level were indent errors that were not caused by the start code. 1100 programs were made where the second loop missed the indent:

```
for i in range 1 to 3:
    for j in range 1 to 5:
    print 'i'
```

6.0.5 Level 11

A very common mistake on level 11 was the addition of a space between the command and the first bracket. For example:

```
print (‘Hello World’)
```

This results in an error message that there is a mistake on the next line with the index of the error in the line being very high (usually above 40), while the line itself is shorter than the index.

Another very common error was the following line:

```
answer = input(‘What is your name?’)
```

The is-sign (=) has not been introduced yet but due to the reordering, this error slipped through in the start code. Before the reordering this was allowed in this level but after the reordering, the introduction of the is sign was put in a later level. The amount of error messages that this caused was 1003. There were 2284 error messages total. This error caused 43.91% of the errors in the user programs.

There were also 95 programs that tried to use the input statement without the variable assignment. These users tried to use:

```
input('What is you name')
```

instead of the correct:

```
name is input('What is you name')
```

I suspect that these errors were made as the start-code resulted in an error caused by the equal sign, as most of these errors were made directly after the start-code error has been received.

6.0.6 Level 12

The call to get a random item from a list changed but it was only shown in the secondary explanation and not in the main explanation. This resulted in the use of of the old random call: “at random”. 71 programs were made with “at random”. This is 5.65% of all errors. It is a small amount but it’s a very relevant error made on this level.

Another relevant error that occurred was the use of the old syntax of lists. In total, there were 118 programs that received the error message: "That was correct Hedy code, but not at the right level. You wrote code for level 12 at level 11." and there were 112 programs that received the message: "The code you entered is not valid Hedy code. There is a mistake on line <linenumber>, at position <positionnumber>. You typed a comma, but that is not allowed.". This error occurred in many cases where the user forgot to put quotation marks around the items in a list. This does not mean that every time this error occurred, it was because of this failure. After a manual check 110 out of the 112 errors were because of missing single quotation marks. In total 228 out of 1255 errors were made with this syntax. This is 18.17%.

6.0.7 Level 13

Level 13 introduces the "True" and "False" statements. Only 74 out of the 422 errors that were made in level 13 are relevant to this level because they used "True" or "False" (or "true" and "false") in any way. Out of these, 10 mistakes were due to indent errors, all the others had mistakes in different statements.

In total, 50 out of the 1254 programs used "True" and "False" without the capital letter (so "true" and "false". We allowed this occurrence so it does not result in an automatic error. It is interesting that out of these 50 programs, 36 caused an error due to some other mistake, usually also the use of the wrong syntax. This is 72%.

6.0.8 Level 14

The "and" and "or" statements were introduced in this level. There were 135 programs that had an occurrence of "and" or "or" in it. Out of these programs, 33 had an occurrence in a print or input statement so these programs were not relevant. Most mistakes had nothing to do with the "and" and "or" statement. There were 18 programs that made a similar mistake; they used the "and" or "or" statements but they only put the variable name in once. For example:

```
if hello is world or world2 or world3:
```

Because there were 18 of these mistakes, out of a total of 102 programs that had an and or or occurrence, 17.65% of all the mistakes were of this kind.

6.0.9 Level 15

There are 15 occurrences in programs that have mistakes that include the hashtag. No mistakes were made in these programs that are relevant to this level as no mistakes can be made with the use of comments.

6.0.10 Level 16

89 out of the 213 programs that had mistakes in them had the "less than" (<) and "greater than" (>) signs. This is 41.78%. Out of these, most mistakes were caused by something not correlated with the "less than" or "greater than" signs.

8 out of these 89 programs had the same mistake: There was no space after a "greater than" or "less than" signs.

```
if 3 <5:
```

This is 8.99% of the programs with a "less than" or "greater than" sign in it. It might be interesting to allow this syntax in this level as most programming languages do allow this.

Another error that occurred was that users already tried to use the "equal" sign. 35 programs tried to use this sign in this level.

6.0.11 Level 17

Only 58 out of the 447 programs with errors had the word "while" in it. This is a very small amount, it is only 12.98%. There were only a small amount of mistakes that had something to do with the "while" loop. The errors that occurred were:

1. A user tried use a list as argument in the "while" loop.
2. A user tried to run the loop as

   ```
   while True:
   ```

3. The indent was forgotten after the call of the "while" loop.

The first two mistakes on this list are probably done by users who already know how to code. The mistakes that they made were valid Python code but not valid Hedy code. The other most common error was the forgotten indent. This is a very consistent error as the error occurs in almost all of the levels.

6.0.12 Level 18

55 errors were due to the single quotation marks missing in the lists. This is out of 236 errors. That results in a percentage of 22.47%. This is 4 percent more than in level 12.

Again, there are 43 programs that included the "at random" call again. This is 18.07%. This is almost 13% more than in level 12.

To make a better comparison to level 12, the percentage should not be amount of errors for a single mistake per total errors but per total programs in total. For level 12 there were 118 programs that forgot to use quotation marks of the 4894 programs. This is 2.41%. In level 18 it is 55 out of the 891 total programs. This is 6.17%.

We can do the same for the "at random" call. In level 12 it is 71 out of 4894 programs. This is 1.45%. In level 18, 43 out of the 624 total executions had this mistake, this is 6.89%.

The conclusion from these numbers is that in a later level, more mistakes were made. The explanation of the later level should be made more clear.

6.0.13 Level 19

Out of all the programs that include a mistake, 45 programs tried to loop through a list with the use of a “for” or “while” loop.

Only 16 programs used the word length.

No significant mistakes were made with looping through a list or with the use of length in this level. There was a user that tried to use the

```
len(list)
```

instead of the

```
length(list)
```

call that was introduced in this level. Once the user received the error code, they tried to replace the len with the word int. It shows that either the error message received when trying to use length or the explanation about length is not clear yet.

6.0.14 Level 20

The amount of data for this level was quite small. Only 4 programs that made a mistake, had the equal-sign included. The only mistake that has been made was that there was an argument missing on the right side of the equal statement.

There was one user who forgot the equal sign entirely. There were no users who still tried to use the word is.

6.0.15 Level 21

The amount of data for this level was also quite small. Only 3 users used the “not equal” sign. There were no mistakes made with the “not equal” sign. One user did try to use the single “equal” sign in an equality check instead of the “double equal” sign as it is supposed to be in an equality check..

6.0.16 Level 22

The amount of data for this level was also quite small. Two users still tried to use the word “is”. No programs were made with mistakes that included the “less than or equal” (<=) and “greater than or equal” (>=) signs.

6.0.17 Common mistakes

There are some common mistakes that were either not relevant to these levels or occurred in almost all the levels.

A common mistake in the user programs that is being done in all of the levels but not relevant to level 8-22, is the use of the single quotation mark (') in a sentence. It is used mostly when someone is trying to type the word "I'm". This results in an error as it closes the sentence but there is still text after the closing single quotation mark.

Another common error is the use of capital letters in statements. Some users start all their lines with capital letters as they might be used to doing while typing regular text and not code. This results in error messages.

A lot of users tried to use calculations in the "if" statements. This is not allowed. It might be interesting to include in a new level or to allow arithmetic in "if" statements in an earlier level..

Quite a lot of users made the mistake that they tried to make a variable consisting of different words (so there was a space in between two words). This is not allowed but it isn't specified that a variable is supposed to be one word in an earlier level.

There are many users who still try to use the word repeat or ask even though they are not allowed to use these anymore.

Almost all users have difficulties in using the indents. Maybe the level explaining the indents could be separated into multiple levels. This might cause less indent errors eventually.

6.0.18 Commands used per level

In this section, we evaluate the programs and the commands they used. Figure 30 shows us all the Hedy commands that were used in different levels. These programs do not include the programs with mistakes and the demo and start code.

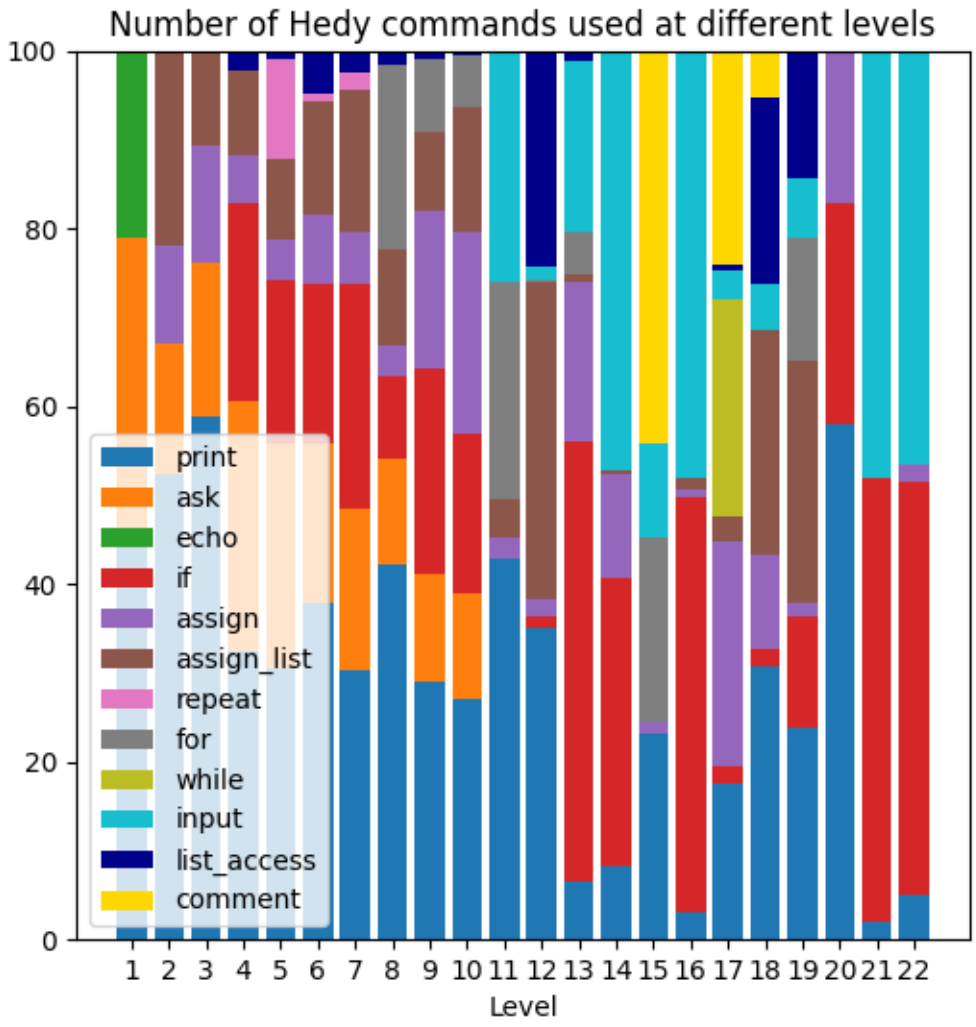

Figure 30: The commands used in each level

From this figure we can make a conclusion. If we take a look at the comments, it is obvious that a comment is only used in level 15, 17 and 18. These are also the only levels where the start or demo code includes a comment. The same does occur for the while loop, it is only used in start and demo code in level 17 so the users only use it in level 17. From this fact, we can make the assumption that users mostly use the commands that are provided by the demo and start code in a level.

6.1 Possible additions to levels

It is quite clear that some mistakes are being made are due to a missing or not clear explanation. The following levels had some common mistakes that could be fixed by a small implementation.

1. Level 8: Users still try to use the “repeat” statement even though it is no longer allowed to

use it. It might be nice to change the error message as it is not really clear now that "repeat" is not allowed when they try to use it. Another possibility is to make the explanation more obvious that the repeat is no longer allowed.

2. Level 10: A lot of users have problems with the use of the indents. Another level or more explanations about indents would be nice to include in this level. The error message when the indent is forgotten is not clear as well, this could also be improved.

3. Level 11: It is not clear to users that a space in-between the print, input and for statement and the first bracket is not allowed. It can be solved by either allowing it in the Lark file or adding extra explanation that they need to be careful not to add a space.

4. Level 12: The call to get a random item from a list changes in this level. It currently is a secondary explanation in the level. It should be added to the main explanation to make it more clear to users that it has been changed.

5. Level 14: To some users it is not clear yet that they should separate different equality checks with an "and" or "or" statement. It can be solved by adding a more clear explanation that only equality checks can be separated by "and" and "or".

6. Level 16: Some mistakes were made in this level by trying to use the "less than" and "greater than" signs with a number but the user didn't at a space in between these two. Currently this is not allowed. However, most programming language do allow this syntax. A small change in the Lark file could solve this error.

7. Level 18: In this level, a lot of users make the same mistakes as in level 12, they forgot to add single quotation marks around items in lists and they try to use the wrong syntax of getting a random item from a list. Another example should be added to get a random item from a list and both explanations should be repeated.

7 Difficulties

Implementing new levels to a big project was not without its difficulties. During the implementation of the levels, some difficulties were encountered, these will be discussed in Section 7.1 to Section 7.3.2. The testing part of this thesis also didn't go to plan. This will be discussed in Section 7.4.

7.1 Adjusting levels

The first difficulty was the adjustment of the levels. Initially there were plans for 13 levels. These were made pretty quickly and then the next levels with new concepts were added. After level 13 to 17 had been added, level 13 needed to be removed and re-added in level 20.

Because the levels build on each-other, it's difficult to reorder the levels. The way that was used to reorder is to save the changes for each level in the grammar file and Hedy.py in a different file. Rename or remove the files with levels that need to be put later. Then the files from the previous level (so the last correct level) need to be copied and renamed for the new level. New

changes to the level can be added now. Make sure you also change the tests for the reordered level. Take note that levels build on each-other so if a certain level has been changed, the changes need to be implemented in the Lark file and tests of all the higher levels as well.

7.2 Example code

Another encountered difficulty in the creation of new levels was the creation of example code. As Hedy is mainly focused on children, the example code needs to be clear but also not too boring. It is quite difficult to combine these two concepts. Either example code is fun but it is too difficult or it is clear but it is too boring. Finding the balance between being exciting and clear is very difficult. Something that can help to create example code is to look at easy games. When the example code corresponds with a simple game like "Rock, Paper, Scissors", it is immediately more interesting to children and it can be coded with a couple of basic statements so the code isn't too difficult.

7.3 Collaboration in an active project

Hedy is an active project with a lot of contributors. When working in such a big project, a few common problems can occur.

7.3.1 Merge conflicts

A first example of a problem is that a pull request was made but another pull request was accepted in between the start of the first request and the time when another collaborator had time to look at the pull request. This caused a couple of merge conflicts. Luckily, these kind of merge conflicts were easy to fix as the other collaborators were usually working in different parts of the files.

7.3.2 Linux vs Windows

Another error that occurred was that other collaborators were working on a Linux system but some were working in PyCharm on Windows. This caused a couple of errors that prevented the execution of the code. There were two specific errors that occurred. The first error was an "UnicodeDecodeError". The solution for this problem was to force the UTF-8 encoding on the part that gave the error. Another was an error caused by a "Charmap". This error was also fixed by forcing the UTF-8 encoding on a different part of the code.

7.4 Testing Difficulties

The testing plan initially was attend some classes of children within the age range of 11 to 14 years old. Due to COVID-19, the schools were closed at the start of this thesis. It was not certain yet whether or not the schools would reopen in time to test the levels.

There were possibilities to test the levels on 5th and 6th graders (Groep 7 and Groep 8 in Dutch) and 7th graders (Brugklassers). Unfortunately, these schools were closed for a very long time. When schools finally reopened, it became apparent that it would not be possible to test the levels in class. The 5th and 6th graders would start with Hedy shortly after the final levels were done but they only went through 1 level each week. As the first level that needed testing was the

8th level, it would take too long before they arrived at this level. The 7th graders would probably go through the levels faster but they wouldn't work on Hedy in the near future. This resulted in a change of plans for the testing of the levels. More information on the actual plan is explained Section 5.1.

8 Conclusions

Creating a programming teaching tool or creating your own programming language includes a lot more than just programming the tool or language itself. First of all, one needs to make clear explanations for the concepts that they introduce. Second, they need to make example code that is obvious and clear but isn't too boring so the learners understand the examples and it challenges them to learn and change the code. The order of the concepts that are being introduced is also quite important, some concepts are more difficult than others to understand and to learn. The most important part of learning is repetition. It became apparent in the testing that when a new concept is introduced in every level, the learners forget what they were taught two levels ago.

It is very obvious that syntax changes can be quite difficult for users to understand and to remember. In the levels where syntax changed, the most mistakes were made. I've learned that when the syntax changes in a level, it must be repeated in the following levels so the users don't forget what they have learned and how the syntax works.

9 Further Research

9.1 Gradual programming/teaching

There can be more advancements in teaching Gradual programming. For example, a same kind of program can be made to teach different languages. Maybe an addendum can be made, once a student is done with the Python part of Hedy, they might be able to switch to C++ or Java for example. When they already know a single programming language, the steps to another, more difficult language or some concepts that might not exist in Python, like variable types, is a smaller step.

9.2 Hedy

After the testing phase, a lot of possible changes and additions to the levels became known. These were discussed in Section 6.1. When these are implemented, a lot of mistakes by the users can be prevented.

There are quite some more levels that can be created for Hedy. These will be explained in the following sections.

9.2.1 Arrays start at zero

Currently, arrays in Hedy start at 1. A new level can be to learn that they start at zero as this is common in almost every other programming language. It can be implemented by changing the "for" loop Python function. The +1 needs to be removed in the return statement.

9.2.2 Loops end correctly

Currently loops end one number higher than they do in Python. For example when one runs the following code:

```
for i in range(1, 3):
    print(i)
```

The text printed in Hedy will be:

```
1
2
3
```

As you might know, in Python the same code only prints

```
1
2
```

This change can also be implemented in a level as it is important for users to know that a loop in Python does stop at one number lower. It can be implemented by removing the -1 in the return statements of the "list_access_var", "list_access" and "change_list_item" Python statements.

9.2.3 Loop through a list

In Python, it is allowed to loop through a list without the user of numbers. For example:

```
  fruits = ["apple", "banana", "cherry"]
for x in fruits:
  print(x)
```

is correct Python code. This is currently not implemented in Hedy yet but some users tried to do this already but it did give an error message in Hedy. This feature can be implemented by adding Lark grammar for this occurence and changing the Python "for" loop function.

9.2.4 Change item in a list

In level 12, the "change_list_item" function has been introduced in the code. This function can adjust an item that is in a list. For example:

```
    fruit is [‘apple’, ‘banana’, ‘cherry’]
    fruit[1] is ‘pineapple’
    print(fruit)
```

Results in the following list:

```
['pineapple', 'banana', 'cherry']
```

The functionality does exist but it is not explained yet in the explanation. This is because there were too many changes in level 12 to introduce this. A level can be added to introduce this concept to the user.

9.2.5 "And" and "or" in "while" loops

Currently, "and" and "or" statements are allowed in "while" loops but there has not been an explanation yet that this is allowed in the "while" loops. Another level might be to explain how "and" and "or" can be used in a while loop.

9.2.6 Augmented Assignment Operators

In the last level "less than or equal" (<=) and "greater than or equal" (>=) have been introduced. The Augmented Assignment Operators (+=, -= and *=) have not been introduced. The Augmented Assignment Operators are small items that can improve the readability of the code but it is not a very necessary topic to introduce. It can be implemented by adding Lark grammar to allow their occurrence and adding a new Python function to convert the Hedy code to Python.

9.2.7 Switch statements

Switch statements have not been introduced yet, currently it is done by if, elif and else statements. A switch statement might be nice to learn as it might improve readability of the code. The switch statements can also be implemented by adding Lark grammar and by adding a new Python function to convert the Hedy code to Python.

9.2.8 Function calls

Function calls can be introduced to teach the learner about functions. Currently, the code is one big function. When functions are introduced, a lot more possibilities will open for the user like branching and recursion.

9.2.9 Branching

When function calls are introduced, a next level that can be introduced is branching. When function calls are allowed, no new code is necessary for branching, just some explanation is needed.

9.2.10 Recursion

The most difficult subject is teaching recursion to learners. When function calls are possible, recursion is also possible to be taught.

References

[B. 14] B. Harvey, D. D. Garcia, T. Barnes, N. Titterton, O. Miller, D. Armendariz, J. McKinsey, Z. Machardy, E. Lemon, S. Morris. Snap!(build your own blocks). *Proceedings of the 45th ACM technical symposium on Computer science education*, pages 749–749, 2014.

[Bar20] Barbero, Giulio and Gómez-Maureira, Marcello A. and Hermans, Felienne F.J. Computational thinking through design patterns in video games. *FDG '20: International Conference on the Foundations of Digital Games*, pages 1–4, 2020.

[Caz16] Cazzola, Walter and Olivares, Diego. Gradually learning programming supported by a growable programming language. *IEEE Transactions on Emerging Topics in Computing*, 4, 2016.

[CO15] Walter Cazzola and Diego Mathias Olivares. Gradually learning programming supported by a growable programming language. *IEEE Transactions on Emerging Topics in Computing*, 4:404–415, 2015.

[Com16] Combéfis, Sébastien and Beresnevičius, Gytautas and Dagiene, Valentina. Learning programming through games and contests: Overview, characterisation and discussion. 10:39–60, 2016.

[GF03] Kathryn E. Gray and Matthew Flatt. Professorj: a gradual introduction to java through language levels. *OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 170—177, 2003.

[Her20] Felienne Hermans. Hedy: A gradual language for programming education. *ICER '20: Proceedings of the 2020 ACM Conference on International Computing Education Research*, pages 259–270, 2020.

[J. 04] J. Maloney, L. Burd, Y. Kafai, N. Rusk, B. Silverman, and M. Resnick. Scratch: A sneak preview. *Second International Conference on Creating, Connecting, and Collaborating through Computing*, pages 104–109, 2004.

[Pat00] Pattis. Karel the robot. 2000.

[Rug13] Rugelj, Jože and Zapušek, Matej. Learning programming with serious games. *Transactions on Game Based Learning*, 2013.

[Sal20] Salac, Jean and Franklin, Diana. If They Build It, Will They Understand It? Exploring the Relationship between Student Code and Performance. pages 473—479, 2020.

[UIG95] UVa User Interface Group. Alice: Rapid prototyping for virtual reality. *IEEE Computer Graphics and Applications*, 15:8—11, 1995.