Universiteit Leiden
The Netherlands

Opleiding Informatica

Parsons Problems

in Hedy

Charlotte Marosvölgyi

Supervisors:
Felienne Hermans

BACHELOR THESIS

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

22/06/2022

Abstract

Hedy is a gradual programming language with the goal to learn children the syntax of Python and to be able to write programs in Python. Hedy consists of levels each including adventures to practice syntax and coding. Every level can be completed with a quiz to test your knowledge of that level. In addition to the adventures and quizzes, another feature was implemented into Hedy called *Parsons Problems*. Parsons Problems are drag-drop puzzles where code lines need to be placed in the correct order. The feature was tested by a group of children between the ages of 11 and 12 years old. The aim of this research is to investigate the effectiveness of Parsons Problems in Hedy and to answer the question whether they could offer a higher quality of learning experience. My research shows that Parsons Problems are very well received by the children and it may be used to investigate where children have difficulties in learning new coding concepts. Parsons Problems may also offer a way to tackle these difficult coding concepts by offering more practice material.

Contents

1 Introduction **1**
- 1.1 Research question . . . 1
- 1.2 Contributions . . . 1
- 1.3 Thesis overview . . . 1

2 Background **2**
- 2.1 Design principles of Hedy . . . 2
  - 2.1.1 Concepts are offered at least three times in different forms . . . 2
  - 2.1.2 Concepts offered in Hedy . . . 2
- 2.2 Parsons Problems . . . 3
- 2.3 Related work . . . 3
  - 2.3.1 Fixing vs. Writing Code . . . 4
  - 2.3.2 Lowering Cognitive Load . . . 4
  - 2.3.3 Design of Parsons Problems . . . 4
- 2.4 Parsons Problems in Hedy . . . 5

3 Implementation **6**
- 3.1 Goal . . . 6
- 3.2 Structure . . . 6
  - 3.2.1 Designing the Parsons Problems . . . 6
  - 3.2.2 Assignment description of the Parsons Problems . . . 7
  - 3.2.3 Coding the Parsons Problems . . . 9
  - 3.2.4 Deploying Parsons Problems for testing . . . 9

4 Experiments **10**
- 4.1 Goal . . . 10
- 4.2 Structure . . . 10
  - 4.2.1 Data collection . . . 10
  - 4.2.2 Analyzing the participants' screen activity and observations . . . 12

5 Results **14**
5.1 Were the Parsons Problems self-explanatory? 14
5.2 Were the assignments clear for the participants? 14
5.3 How many mistakes did the participants make before solving the Parsons Problem correctly? 15
5.4 What kind of mistakes did the participants make during solving the Parsons Problems? 16
5.5 Is there a correlation between Parsons Problems and the scores of the end quiz? . . 17

6 Conclusions and Future Work **21**
6.1 Answering the research question 22

References **23**

1 Introduction

Learning how to program is becoming a more in demand skill everyday due to the spur development in technology. However, programming can be an overwhelming skill to learn. In this thesis we focus on a programming language called Hedy that aims to learn the novice programmer how to code. Hedy is designed to introduce children to programming with the focus on teaching the syntax of Python. Hedy is presented as a gradual programming language. This means that students learn to write simple commands first without having to worry about syntactical details. Then, not unlike learning a natural language, the various rules of its grammar (syntax) are introduced in a stepwise fashion. The eventual goal of Hedy being to teach children to master Python. This goal is achieved by carefully chosen design principles among other concepts [Her20]. One of the principles being to *offer concepts at least three times in different forms to ensure a concept is stored in our long-term memory*. At this moment, Hedy offers two different forms of code concepts per level. In this research I will be implementing a third form of the concept that is being taught, to meet the first design principle of Hedy. This third form will be presented in the form of *Parsons Problems*. A lot of research has been done on Parsons Problems that show positive effects in learning programming.

1.1 Research question

The goal of my bachelor thesis at the Leiden Institute of Advanced Computer Science, supervised by Felienne Hermans; will be to implement Parsons Problems in Hedy and further investigate their effectiveness by running a small experiment which will answer the overarching question:

"Could implementing Parsons Problems in Hedy be a promising tool to offer an even higher quality of learning experience?"

1.2 Contributions

The contributions this thesis makes are implementing Parsons Problems into the first eight levels of Hedy. On the sixth of June 2022 the Parsons Problems went live on the Hedy website and were already used 132 times in less than 24 hours. In a time span of two weeks the Parsons Problems were used more than 700 times.

1.3 Thesis overview

This chapter contains the introduction; Section 2 includes the background on Hedy and Parsons Problems; Section 2.3 discusses related work; Section 3 describes how the Parsons Problems were implemented in Hedy; Section 4 describes the setup of the experiment that was conducted with children on a Primary School; Section 5 shows results that were collected during the experiment. Section 6 discusses the results of the experiment and the conclusions that are drawn from them.

2 Background

As mentioned in the introduction the idea for my thesis came from the desire to complete the first design principle of Hedy. In this section I will elaborate on the meaning of this design principle, what concept I have chosen to implement in order to meet the design principle and why I chose this concept.

2.1 Design principles of Hedy

Hedy offers a Python like language with the overall goal being to teach a novice programmer Python itself. In order to achieve this goal, syntax that becomes gradually more complex is added to the Python like language until a programmer masters Python itself. Hedy follows six design principles to achieve this goal:

1. Concepts are offered at least three times in different forms.
2. The initial offering of a concept is the simplest form possible.
3. Only one aspect of a concept changes at a time.
4. Adding syntactic elements like brackets and colons is deferred to the latest moment possible.
5. Learning new forms is interleaved between concept as much as possible.
6. At every level it is possible to create simple but meaningful programs.

As Hermans describes in her paper many of these design principles are already met [Her20]. To keep in line with these design principles my research will focus on the first design principle that states that concepts are offered at least three times in different forms. As mentioned before, I will be introducing a third form of offering a concept.

2.1.1 Concepts are offered at least three times in different forms

The first design principle that a concept should be offered at least three times in different forms emerged from the following research. Writing education [FL89, Sim73] found that concepts need to be offered in different forms over a longer period of time. Additionally, it has been found in order for words to be stored in long-term memory it needs to be read at least 7 times [VV99].

2.1.2 Concepts offered in Hedy

As of now Hedy offers concepts in two different forms. The first form is offered to the programmer by writing code. The second form is offered by making a multiple choice quiz at the end of each level, to test the programmers' knowledge. The third form will be offered in the shape of Parsons Problems (discussed in Section 3.2). The idea for implementing Parsons Problems in Hedy arose from the conversation I had with my supervisor. It occurred to us that the knowledge of users is tested based on an end quiz. However, users are not yet tested based on code they have written. Implementing Parsons Problems is a great way to offer a more versatile way to test coding knowledge.

2.2 Parsons Problems

When learning a new programming language it is necessary to master basic syntax of the language and learn how to make logical constructs. When this basic knowledge is not present it is not possible to write programs [PH06]. Many research has been done on the best ways to teach Introductory Programming, which is also referred to as the programming paradigm [DH94]. Parsons & Haden try to look at this paradigm from another angle, by comparing `learning how to program` to `learning a new language` and `the early learning of mathematics`. Maths and foreign languages are taught by repeated exposure through practice and drill, also known as 'rote learning' techniques [NS96]. Since there is some resemblance in the three disciplines (learning a language, maths and programming), the idea originated that learning syntactic rules needed for programming might benefit from the same learning technique [PH06]. However, the downside of this technique is that students find it a boring and tedious task. To tackle this problem researchers Parsons and Haden developed a tool called 'Parson's Programming Puzzles' (hereafter: Parsons Problems) consisting of basic programming principles to help learn the required skills for programming. A Parsons Problem is an interactive puzzle where students need to rearrange code blocks until they end up with a program that is ready for execution. The design of the Parsons Problem was based upon five principles [PH06].

1. Maximize engagement: Research has shown that making a concept that needs to be learned into a game/puzzle is an effective way to promote engagement. Maximizing engagement is important to ensure repeated exposure that is required for rote learning.

2. Constrain the logic: Students need to be able to make syntactic judgements. Therefore it is important that Parsons Problems provide a good structure (i.e. semantics), so students are exposed to as little distractions as possible.

3. Permit common errors: Students have shown to make certain syntactic errors time after time. Allowing students to make errors by offering distractor options (code blocks with syntactically incorrect code), enables them to compare correct and incorrect syntax.

4. Model good code: Sometimes a student provides a solution which will generate correct output. However, the approach the student has taken to solve the problem might be too cumbersome. It is of importance the student will be exposed to the more obvious coding solution. This can be achieved by ensuring students have to continue solving the Parsons Problem until it is a 100% correct.

5. Provide immediate feedback: Students often struggle with finding out why a program does not work correctly or does not execute. By providing immediate feedback when an incorrect choice is made during solving a Parsons Problem this struggle can be made much easier.

Parsons Problems where originally designed to be a fun and effective tool to learn Turbo Pascal. However, the Parsons Problems can be used for any programming language.

2.3 Related work

Extensive research on Parsons Problems has been done after the first publication by Parsons & Haden [PH06]. The studies I deemed most relevant for my own research will be discussed next.

2.3.1 Fixing vs. Writing Code

Ericson et al. [EMR17] researched the effectiveness of Parsons Problems with regards to writing and fixing code. Students were divided into three groups. Each group was given a different configuration to solve the same problem set. The problem was then represented as either a debugging exercise, writing exercise or a Parsons Problem. The students were given a pretest and posttest to measure the learning effectiveness. They found no significant difference in scores between all three groups. What they did find was that Parsons Problems took less time to solve than writing or fixing code. Making Parsons Problems an efficient learning tool [Sha20]. Some other research on using Parsons Problems vs. writing code has found the same outcome [Sha20].

2.3.2 Lowering Cognitive Load

Sweller found in order for learning to occur the working memory has to process new information first. After processing, the information is added to knowledge representations (schemas) that exist in our long-term memory [Swe88]. The working memory has a limited capacity. When this capicity is used entirely there is no room for modifying or building schemas in our long term memory. Since novice programmers are expected to learn complex material, it is of importance to try to lower the cognitive load. Hermans and Ericson [Her20, EMR17] both stress the importance of limiting cognitive load of a learner several times. In Hedy several measures have been taken to lower the cognitive load by for example using the Spiral Approach [Shn77]. Ericson believes Parsons Problems to have a lower cognitive load than a problem that requires to write code from scratch since the problem space of a parsons problem is more constrained [EMR17].

2.3.3 Design of Parsons Problems

Not only the effectiveness of Parsons Problems has been researched, a great emphasis lies on the design of the problems as well. It is important to discuss the following three design principles [Sha20]. This way design decisions that align with the design principles of Hedy can be made to implement Parsons Problems in the best way possible.
The following designs will be discussed:

1. **Two-dimensional Parsons Problems** allow for indenting code blocks correctly in addition to placing code correctly. At present it is common practice to offer Parsons Problems in two-dimensions in Python [DLRD20]. Be that as it may, it is important to keep in mind that using two-dimensional Parson Problems does make the problem more difficult to solve. Which in turn could result in increasing the cognitive load.

2. **Distractors** are code blocks that are not part of the solution [Sha20]. They are placed between the correct code blocks to distract the learner. This way comparisons between correct and incorrect code are enforced. Research on the effectiveness of using distractors points to different outcomes. Ericson found them to be highly efficient whereas Harms et al. suggests that using distractors decrease learning efficiency [EMR17, HCK16].

3. **Feedback** given to students can take shape in two ways. The first one being *line-based* feedback and the second one being *execution-based* feedback. Line-based feedback entails immediate feedback after a code block is placed. This entails two downsides. In order for this

to be implemented a unique solution is required. Second, a trial-and-error based approach to solve the problem becomes obvious. On the upside, the problem might be easier to solve. Execution-based feedback relies on feedback after execution of the problem. This is similar to feedback programmers get after compiling code. Research shows this way might take a learner longer to solve the problem. On the other hand they ask for less feedback, since this approach is said to encourage a student to think more critically about the solution to the problem beforehand [Sha20].

2.4 Parsons Problems in Hedy

In short, the idea for creating Parsons Problems originates from recognizing that syntactical errors and semantics form an obstacle in learning how to code [PH06]. Hermans acknowledges these problems as well: "One of the aspects of programming that learners often struggle with is the syntax of programming languages: remembering the right commands to use and combining those into a working program" [Her20]. Since Hermans and Parsons both emphasize the importance of learning correct syntax, Parsons Problems and Hedy seem to go hand in hand.

3 Implementation

3.1 Goal

The primary goal of this research is to implement a third form in which code is presented, with the aim to further improve the learning experience of Hedy. For this third concept code will be shown in the form of Parsons Problems. In this section I will discuss the implementation of the Parsons Problems in Hedy.

3.2 Structure

The implementation of Parsons Problems in Hedy consisted of the following steps:

1. Design the Parsons Problems
2. Content of the Parsons Problems
3. Implementing the Parsons Problems
4. Deploying the Parson Problems

3.2.1 Designing the Parsons Problems

The first draft of the design was made using copying and pasting screenshots of the existing page of Hedy. In the screenshots I added colored blocks and text using the original Hedy colors. Next when implementing the design into code, I found other design choices where more preferable (e.g. drag code from the top of the page to bottom of the page). Lastly, the final design with a few added details was completed. The different stages of the designs are shown in Figure 1.

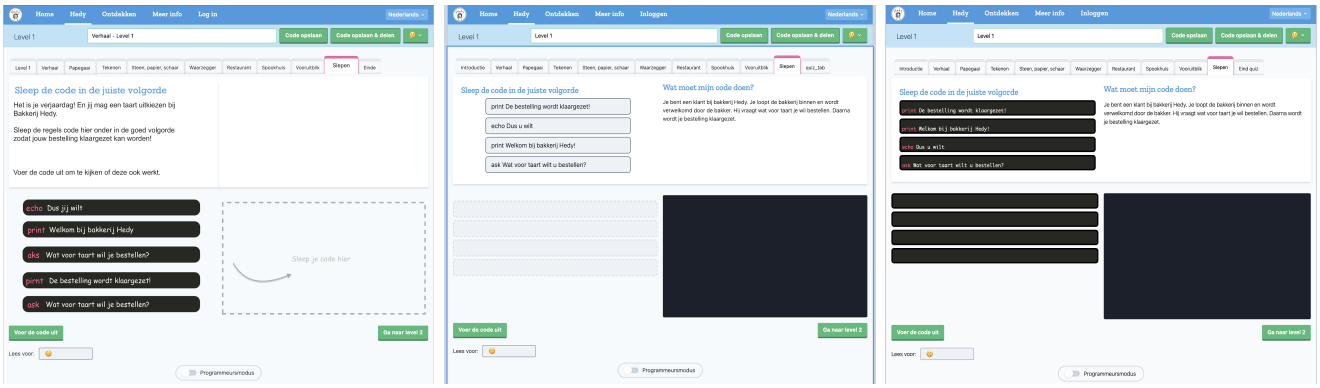

Figure 1: Stages of the designs (fLTR: initial design, second design and final design

As mentioned in the Background Section 2, Parsons Problems are represented as drag-drop puzzles in which code lines are randomly displayed in containers. It is the task of the user to drag these code lines into the correct order into a set of empty containers.

The design was based on the following demands:

- *The standard design of Parsons Problems.* The requirements that the standard design should meet are: (1) draggable code lines, (2) two sets of containers, the first set containers each consisting of a code line, the second set is displayed as empty containers, (3) a mechanism that checks whether the code lines are dragged to the empty containers and are then placed in the correct order.

  Taking the standard design principles the programmer can choose numeral ways to enhance and implement these designs. The following ideas had to be taken into consideration.

- *Checking mechanism.* Next, I had to consider a way of making clear to a user whether their code line(s) were placed into the correct order. In order to do so, I had two options. The first option was using line-based feedback. This would be done by coloring the container the code line was dragged in immediately. Making the color of the container green, red, if a code line was placed correctly or incorrectly, respectively. The second option was execution-based feedback. This would be done by colouring the containers after a user hit the execute button. I chose for the latter, based on research discussed in the related work Section 2.3.

- *Executing code.* In other Parsons Problems when code lines are placed in the empty containers, their order is checked. However, to make the Parsons Problems even more interactive, my demand was to also be able to execute this code. This is the first extra demand. This way the users is exposed to the output of the code lines they placed. The moment the users execute incorrect placed code lines, they will receive error messages and feedback on what went wrong. The user can endlessly replace the code blocks until the answer is 100% correct.

- *Distractors.* The second design choice I wanted to take into consideration was the use of distractors. Distractors are code lines that contain grammatically incorrect code on purpose. Initially the inventors of Parsons Problems included these distractors into the puzzles in order to make users aware of the importance of well-written code. However, later research was done on the use of distractors which showed a decrease in learning efficiency [HCK16]. Based on this research, I chose to not to use distractors in my design, to ensure the cognitive load stays low.

- *Syntax Highlighting.* The last demand was that in my design, the commands in code lines that were already present in Hedy were highlighted. By doing this, I wanted to get the users familiarized with the commands. Ensuring the design of Parsons Problems was in line with the existing design of Hedy.

3.2.2 Assignment description of the Parsons Problems

Each Parsons Problem consists of an assignment that can be found under the heading *What is my code supposed to do?*. A user must read this assignment before starting the problem. Since it is not always clear in what order code should be dragged in, the story should be used as guideline. This is necessary in terms of creating the correct and desired output. For example `print` statements can be put in any kind of order and will still execute correctly. However, it might not give the desired output.

The stories were created taking the following concepts into account:

- Interest of the children
- What does a user learn in this level

I have created Parsons Problems for the first eight levels of Hedy. Table 1 shows the concepts that are taught in each level. My assignments were set up in such way that they are interwoven with these concepts. Figure 2 and Figure 3 show examples of assignments that are based on concepts 1 and 2 respectively. The inspiration for the assignments is based on the common interests of today's children such as Harry Potter and movies like Sonic the Hedgehog.

Level	Concept
1	printing and input
2	assignment using is (defining variables)
3	quotation marks and types
4	selection with if and else flat
5	repetition with repeat x times
6	calculations
7	code blocks
8	for syntax

Table 1: Different concepts introduced in the levels of Hedy

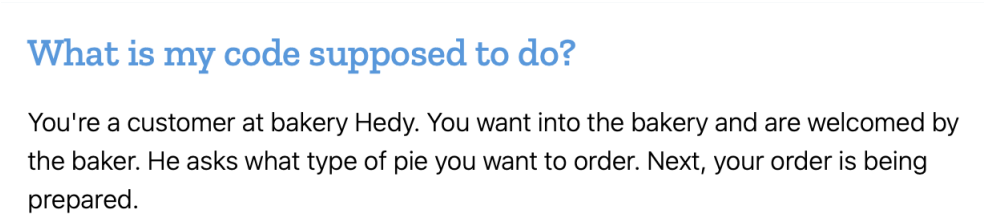

Figure 2: The assignment of the Parsons Problem of level 1 which is based on concept 1.

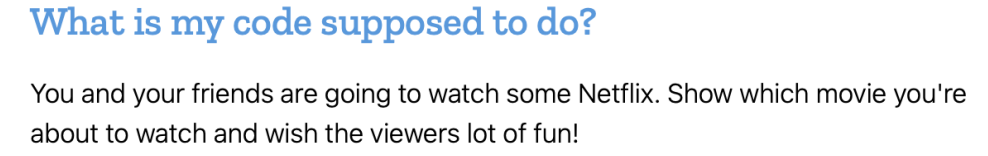

Figure 3: The assignment of the Parsons Problem of level 2 which is based on concept 2.

3.2.3 Coding the Parsons Problems

Working in a big collaboratory project, just like Hedy, was a new experience for me. I had not developed any code in majority of the languages that Hedy is written in or participated in a fullstack development project. Through online meetings with Timon Bakker, who is a coworker on the Hedy project, and lots of trial and error I managed to implement the design of Parsons Problems into Hedy. The design was implemented by coding in HTML, CSS and TypeScript. The back-end, which was needed for the checking mechanism of the assignments was coded in Python.

3.2.4 Deploying Parsons Problems for testing

After implementing Parsons Problems in the first eight levels of Hedy, the new feature was exploited on the alpha version of Hedy, see website. By exploiting the feature the design was ready for its first test run, which will be discussed in the Experiments Section 4.

4 Experiments

4.1 Goal

The goal of the experiment is to test the user experience of the Parsons Problems that I implemented in Hedy. In order to do so users will be observed during a pilot of the implemented feature. For this pilot a selected group of participants will be testing the Parsons Problems. The user experience will be measured by tracking the screen activity of the participants and the outcomes of the Parsons Problems they try to solve.

4.2 Structure

Running the experiment consisted of the following steps:

1. Data collection
2. Analyzing the participants' screen activity and observations

4.2.1 Data collection

The data collection consisted of four steps, namely the selection of participants, the system requirements, the planning of the pilot and retrieving the participants' screen activity and output. The pilot was divided into two days.

Selection of participants
Firstly, a selection of participants had to be made. Since Hedy is a platform suited primarily for children between the age of 11-14, this was also my target group for the experiment. A prior knowledge in coding was not required. The test group consisted of six children from primary school 'De Vos' in Voorschoten aged between 11-12 years old. For the selection I tried to make sure the group consisted of children with different levels of learning skills. The different levels of learning skills where chosen in a way that an even distribution in learning skills was taken into account as much as possible, see Table 4.2.1. In the Dutch school system there exist three different streams that represent different educational paths. These streams are: VMBO (preparatory secondary vocational education), HAVO (senior general secondary education), and VWO (university preperatory education). In other countries usually two streams of educational paths are offered: (1) the stream of vocational training (VMBO) and (2) the stream of university (VWO). The stream HAVO is not present in other countries, but is meant to prepare students for universities of applied sciences in The Netherlands.

System requirements
Secondly, multiple system requirements had to be met. These requirements include: (1) establishing a stable internet connection, and (2) setting up three laptops displaying the alpha-version source of Hedy which at the time included the Parsons Problems.

Pilot day 1
On the first day of the pilot the aim was to get the participants familiarized with Hedy. First, the

Participants	Learning skills
User 1	VMBO
User 2	VMBO/HAVO
User 3	VMBO/HAVO
User 4	HAVO
User 5	HAVO/VWO
User 6	VWO

Table 2: The 6 participants including their learning skills.

participants were given a laptop that displayed the alpha-version of Hedy. Second, I provided the participants with a short introduction about Hedy and the aim of that day. The participants were also encouraged to ask any questions at any time if something was not clear for them. I announced that each participant was given 1.5 hours to practice coding in Hedy. The coding in Hedy had the following restrictions. The participants were asked to make exactly five coding exercises, called 'adventures', per level ensuring that the programmers were exposed to the same amount of coding exercise. The set of adventures that were chosen always consisted of the introduction adventure followed by the next four contiguous adventures, and always excluding the drawing adventure (called 'Turtle'). The drawing adventure was excluded since I did not deem this exercise to be of as much importance in learning the basic commands exposed in each level. Figure 4 shows how the adventures in each level are displayed. Figure 5 shows multiple adventures (in the same level). At the end of the pilot of day one the level they reached in 1.5 hours was noted. The amount of levels the participants reached (and made) was their base for the second day of the pilot.

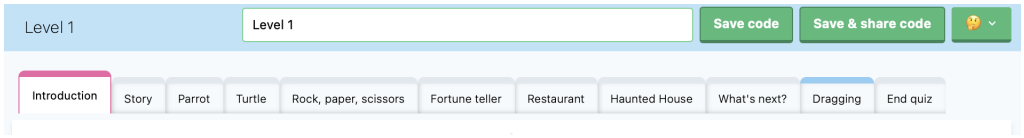

Figure 4: The display of the adventures of level 1

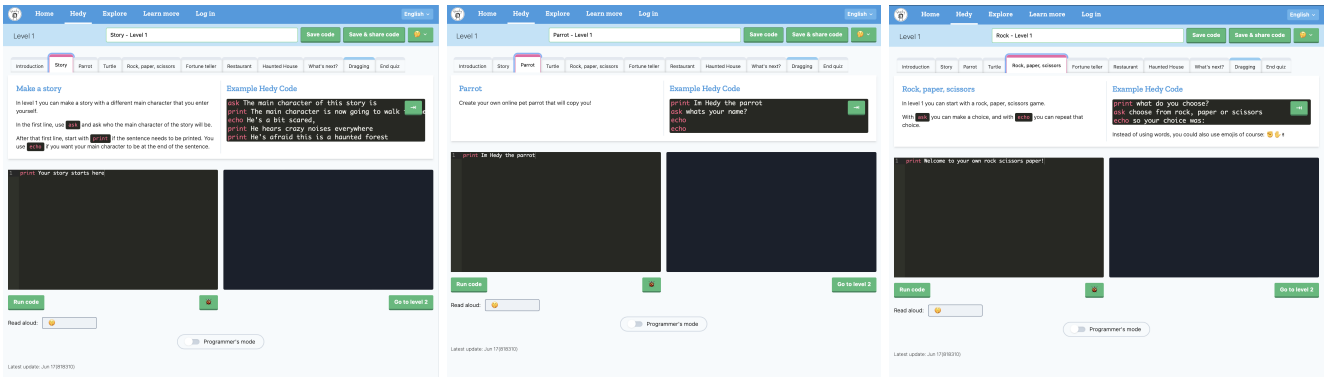

Figure 5: Multiple adventures of level 1 (fLTR: story, parrot and rock paper scissors)

Pilot day 2

On the second day of the pilot the aim was to get the participants to solve the Parsons Problems

and make the end quiz of the levels they reached on the pilot of day one. To clarify, the amount of Parsons Problems and end quizzes were equal to the amount of levels they reached on the first day of the pilot. In addition, the aim was to obtain the participants' screen activity and to note any uncertainties found by the participants during this session (e.g. questions on how to proceed if the given task was not clear for them). The participants were divided over two groups each consisting of three participants, again with an even distribution of learning skills. The first group was asked to first make the end quiz, followed by solving the Parsons Problems. The second group was asked to start with solving the Parsons Problems, followed by making the end quiz. Again the participants were provided with the laptops with the Parsons Problems or the end quiz showing on their laptop screens. I explained the two tasks of the day were to make multiple choice quizzes (end quiz) and to make the Parsons Problems that I had implemented. However I did not any context on what a Parsons Problem is or what the exercise consisted of. While the participants were making the tasks given, I quietly observed them by walking around and making sure I was always available for any questions. In order to obtain the screen activities of the participants, a screen recording was made of the entire session (Parsons Problems and end quizzes). A total of six recordings was stored on my computer in order to analyze this data. Since screen recordings only capture digital motion, handwritten notes were taken as well during these sessions in order to capture verbal communication. These notes were included for keeping track of the times that a participant did not understand the assignment or needed more guidance.

4.2.2 Analyzing the participants' screen activity and observations

In order to measure the effectiveness of the Parsons Problems that were implemented I took the following questions into consideration:

1. Were the Parsons Problems self-explanatory?
2. Were the story descriptions clear for the participants?
3. How many mistakes did participants make before solving the Parsons Problem correctly?
4. If any, what kind of mistakes did the participants make during solving the Parsons Problems?
5. Is there a correlation between Parsons Problems and the scores of the end quiz?

To be able to answer these questions the following data was extracted:

1. **Were the Parsons Problems self-explanatory?**
   Both the screen recordings of solving the Parsons Problems and the hand-written notes of day two of the pilot were analyzed. In the screen recordings immediate action was measured in terms of dragging the code lines. The notes were meant to keep track of any participant that asked for verbal confirmation before starting the Parsons Problem.
2. **Were the story descriptions clear for the participants?**
   Again, the hand-written notes of day two of the pilot were analyzed. The notes were used to keep track of any participants that asked for verbal confirmation to understand the story that belongs to each of the Parsons Problem.

3. **How many mistakes did participants make before solving the Parsons Problem correctly?**
   The screen recordings of solving the Parsons Problems of day two of the pilot were analyzed to count the number of mistakes the participants made. For each participant the number of mistakes they made per Parsons Problem was noted.

4. **What kind of mistakes did the participants make during solving the Parsons Problems?**
   Again, the screen recordings of solving the Parsons Problems of day two of the pilot were analyzed. This time it was noted what type of mistakes the participants made while trying to solve the Parsons Problem.

5. **Is there a correlation between Parsons Problems and the scores of the end quiz?**
   The screen recordings of solving the Parsons Problems and solving the end quiz of day two of the pilot were analyzed. In doing so, the number of mistakes the participants made while solving the Parsons Problems were counted and the scores of the end quiz were kept. These two measurements were compared with each other.

5 Results

In this section the results of the experiments will be discussed. The five questions from Section 4.2.2 were taken into consideration for the analysis of the data.

5.1 Were the Parsons Problems self-explanatory?

Table 3 shows whether the Parsons Problem was self-explanatory for the users. This was tested by asking the following two questions.

- Immediate dragging: Did the user immediately understand that the code blocks had to be dragged to the empty containers? This question can be answered with a simple yes or no.
- Verbal confirmation problem: Did the user ask for an explanation before starting the assignment? This question will again be answered with a simple yes or no.

It should be emphasized that the answer to these questions will always be yes/no or no/yes. When a user finds the Parsons Problem self-explanatory (starts immediate dragging) then there was no verbal confirmation about how to start. Vice versa when the user found the Parsons Problem non self-explanatory, verbal confirmation about how to start was asked.

Participants	Immediate dragging	Verbal confirmation for problem
1	yes	no
2	yes	no
3	no	yes
4	yes	no
5	no	yes
6	yes	no

Table 3: Parsons Problems: self-explanatory

5.2 Were the assignments clear for the participants?

Table 4 shows whether the assignment that was written to support the Parsons Problem was clear. This was done by asking the following question:

- Verbal confirmation for assignment: Did a user ask for verbal confirmation in the case they did not understand the assignment. The assignment can be found under the heading *What is my code supposed to do?* in every Parsons Problem on the Hedy website, see Figure 6.

Participant number 5 was under the impression that the draggable containers were supposed to be clicked on and asked why this was not working. I gave a small hint by pointing to the upper left corner of the screen where the text *Drag the code in the correct order*, see Figure 6, is displayed. Participant number 5 then proceeded with the task at hand without hesitation.

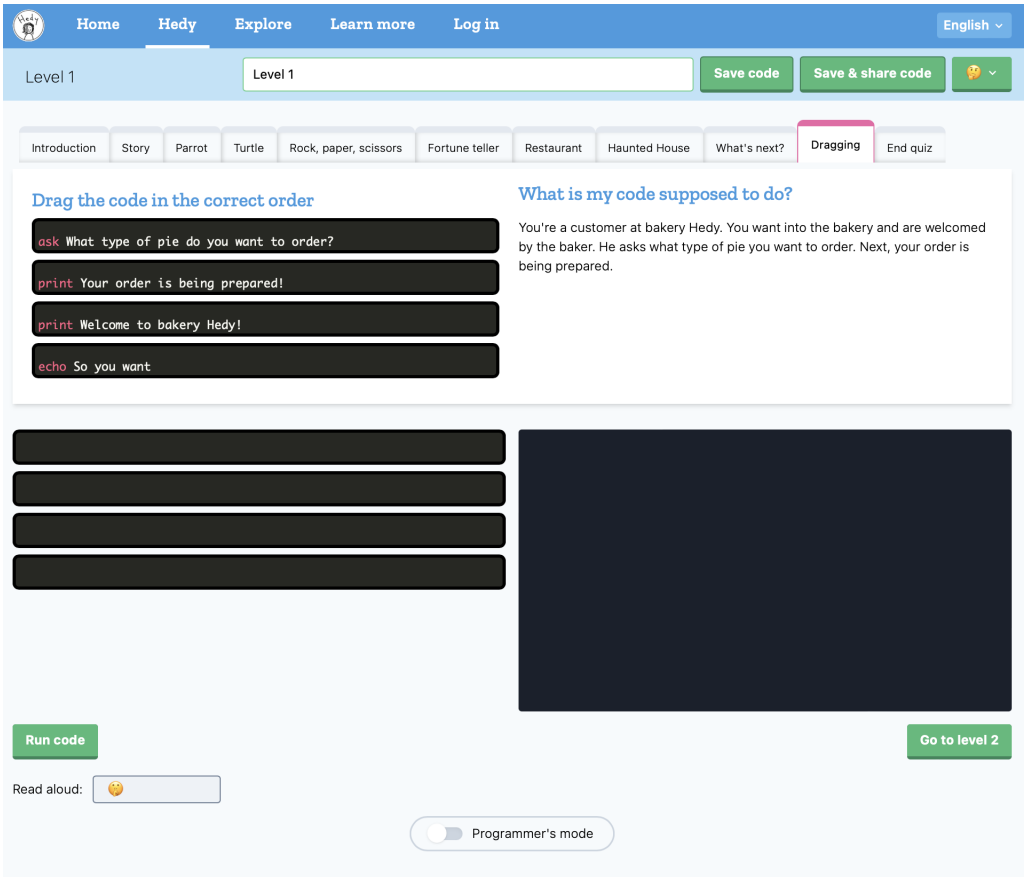

Figure 6: Example of a Parsons Problem

Participants	Verbal confirmation for assignment
1	no
2	no
3	no
4	no
5	yes
6	no

Table 4: Parsons Problems: clarification of assignments

5.3 How many mistakes did the participants make before solving the Parsons Problem correctly?

Table 5 shows how many mistakes where made during solving the Parsons Problem. This was measured by counting the times a user needed to replace the code blocks in order to get a correctly executable program. To clarify, the table shows that user 1 made 0 mistakes in solving the Parsons Problem in level 1 and made 1 mistake until solving the Parsons Problem in level 2. Additionally, a '-' is shown when a user did not finish this level on the previous day or did not want to continue

any further.

	Number of mistakes		
Participants	level 1	level 2	level 3
1	0	1	-
2	0	0	-
3	0	0	-
4	0	1	-
5	0	1	0
6	0	1	-

Table 5: Parsons Problems: the number of times before the Parsons Problem was solved

As Table 5 shows, when a participant made a mistake while solving the Parsons Problem it only took them one time to correct the mistake.

5.4 What kind of mistakes did the participants make during solving the Parsons Problems?

Table 6 shows what type of mistake a user made during solving the Parsons Problem. The type of mistake that was made was the same for each of the users, namely the wrong order of variable declaration. Other types of mistakes where not observed. Here a '-' means that the user did not make any type of mistake.

	Types of mistakes		
Participants	level 1	level 2	level 3
1	-	variable declaration	-
2	-	-	-
3	-	-	-
4	-	variable declaration	-
5	-	variable declaration	0
6	-	variable declaration	-

Table 6: Parsons Problems: types of mistakes

Figure 7 shows the exact way the participants made the mistake. Every participant placed the code-line `print We're going to film` above the code-line `film is Sonic the Hedgehog 2`. Remarkably, the type of mistake that was made by the participants was the same. All the participants seemed to struggle with the notion that a variable name, such as `film` should be declared, before actually being able to use it. Therefore I identified this mistake as a mistake in the declaration of a variable.

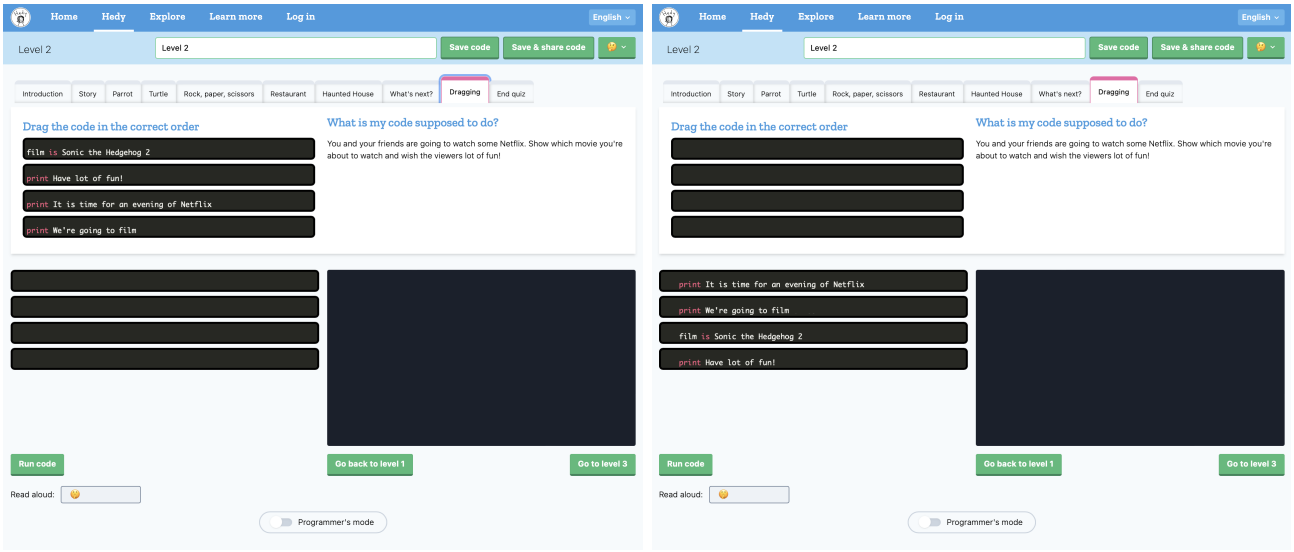

Figure 7: A mistake in variable declaration (level 2)

5.5 Is there a correlation between Parsons Problems and the scores of the end quiz?

Table 7 shows the users that made a Parsons Problem followed by the end quiz of the corresponding level. Table 8 shows the users that made the end quiz followed by the Parsons Problems of the corresponding level. Both tables show if the user made any mistakes in the Parsons Problem and the end quiz. If a user made a mistake in the end quiz, the number of the question that was answered incorrect is noted. If a user made a mistake in the Parsons Problem the number of mistakes was saved. Again, a '-' is shown when a user did not finish this level on the previous day or did not want to continue any further.

	Number of mistakes in Parsons Problems			Mistakes in end quizzes		
Participant	level 1	level 2	level 3	level 1	level 2	level 3
5	none	1	none	3	2, 4, 10	7, 8, 9, 10
2	none	none	none	none	1, 2, 4, 9, 10	-
3	none	none	none	none	1, 2, 4, 10	-

Table 7: Measurements of group 1: Parsons Problems followed by end quiz

	Mistakes in end quizzes			Number of mistakes in Parsons Problems		
Participant	level 1	level 2	level 3	level 1	level 2	level 3
4	5	2, 9	-	none	1	-
1	3, 8	8, 9, 10	-	none	1	-
6	none	1, 2, 4, 6, 10	-	none	1	-

Table 8: Measurements of group 2: End quiz followed by Parsons Problems

Figures 8 and 9 correspond with Tables 7 and 8. Figures 8 and 9 show the exact questions that were answered incorrect.

Level 1
In level 1 the questions of the end quiz test the knowledge on the commands `print`, `echo` and `ask`. The most commonly made mistake in level 1 was question 3. Questions 5 and 8 will be omitted since these are outliers (infrequently made mistakes) in the set of incorrect answered questions.

- Question 3 was answered incorrect by two participants. We will refer to question 3 as an `ask` problem, see Figure 8. The command `ask` is used to obtain user input. Participant number one answered the question with 4 and 1 before giving the correct answer, which is 3. Participant number five answered with 2 before giving the correct answer.

Since the Parsons Problem in level 1 was made correct and the incorrect answers of the end quiz of level 1 among the groups were sporadic, I do not see any causal relationship between the Parsons problem and the end quiz in level 1.

Level 2
In level 2 the questions of the end quiz test the knowledge on the commands `is ask`, `is`, `print` and `sleep`. The most commonly made mistakes among the groups in level 2 were the questions 2, 4, 9, and 10. Questions 1, 6 and 8 will be omitted since these are outliers in the set of incorrect answered questions.

- Question 2 was answered incorrect by five participants. We will refer to question 2 as an `is ask` problem, see Figure 9. This is a form of variable declaration where the emphasis lays on declaring the variable `name` with user input. Participants number three, four, five and six answered 2 before giving the correct answer, which is 1. Participant two answered 3 followed by 4 before giving the correct answer.

- Question 4 was answered incorrect by four participants. We will refer to question 4 as an `is` problem, see Figure 9. This is a form of variable declaration where the emphasis lays on declaring the variable `name` and then printing it to the output console. Participants number two and five answered 2 and 3 before giving the correct answer, which is 4. Participants number three and six answered 2 before giving the correct answer.

- Question 9 was answered incorrect by three participants. We will refer to question 9 as an `is` problem, see Figure 9. This is a form of variable declaration where the emphasis lays on declaring the variable `animal` and then printing it to the output console. Participants number one and four answered 1 before giving the correct solution, which is 2. Participant number two answered 4 before giving the correct answer.

- Question 10 was answered incorrect by five participants. This is a form of variable declaration where the emphasis lays on declaring the variable `flavor` with user input, see Figure 9. Participants number one and three answered 2 before giving the correct answer, which is 4. Participants number two, five and six answered 1 and 2 before giving the correct answer.

The Parsons Problem in level 2 shows that the mistake that was made was always in variable declaration. As can be seen from the questions that were answered incorrect in the end quiz of level

2, these were also all related to variable declaration. The reason for these mistakes could be due to a lack of coding practice or the concept of declaring a variable may be too difficult to understand. It is interesting to see that most of the participants experience difficulties in both the Parsons Problem and the end quiz exercises that deal with variable declaration.

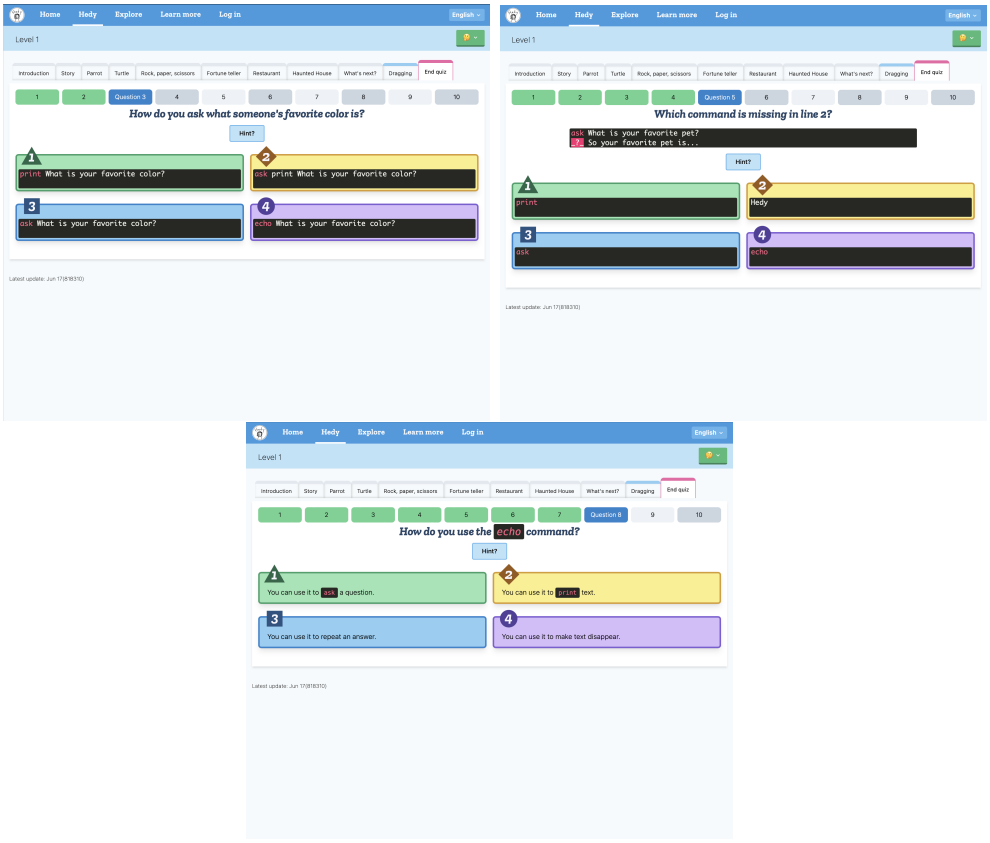

Figure 8: Incorrect answered questions in the end quiz of level 1 (fLTR: questions 3, 5 and 8)

Figure 9: Incorrect answered questions in the end quiz of level 2 (fLTR: questions 1, 2, 4, 6, 8, 9 and 10)

6 Conclusions and Future Work

Recalling the research question from Section 1.1, in this section an answer will be provided in agreement with the five sub-questions that the results where based on.

Before answering the research question the following should be kept in mind at all times:

- The study was conducted with a small test group (n=6). It is however a great way to start testing the Parsons Problems that were implemented for the first trial. In addition, it could be an indication to whether this will be a useful tool to implement in the future.
- The participants were nervous and excited. The participants knew they were being observed and had to take tests which showed immediate feedback both of which can be quite daunting.
- Whether participants really 'read' the assignments can not be guaranteed. Due to nerves, performance pressure and the average age of the participants it is not unlikely that a participant failed to grasped the understanding of the assignment at hand.
- The participants had only practiced with Hedy once prior to the experiment (day one of trial). Therefore, perfect knowledge can not be expected. A mistake in the quiz that was made by almost everyone was not knowing which commands were omitted in a level. This is a mistake that can be winked at.

Overall the Parsons Problems were well received among the participants. The results that were obtained from the experiment show that 1) the majority of the participants immediately started dragging the code blocks into the empty containers and 2) the majority of the participants understood the assignments that were at hand. These two results combined show a positive effect on the self-explanatory nature of the Parsons Problems that were implemented.

Second, an effect in first letting participants complete the Parsons Problem followed by the corresponding end quiz of that level and vice versa has not been observed. It might be interesting to mention that the group of participants that first solved the Parsons Problems followed by the end quiz (group 1) made 11 mistakes in the end quiz of level 2. While the other group of participants (group 2) made 10 mistakes in that same end quiz. The number of mistakes the groups made in the end quiz of level 2 is about the same. In addition, it can be noticed that group 2 made three mistakes in solving the Parsons Problem of level 2, while group 1 made only one mistake in that same problem. We can observe a kind of trend where making the end quiz before solving the Parsons Problems tends to result in more mistakes.

Lastly, it can be observed that the mistakes that were made in level 2 of the end quiz were the same in nature as the mistakes that were made in level 2 of the Parsons Problem. The Parsons Problem in level 2 is focused on the declaration of variables. In the same way, the most commonly made mistakes in the end quiz of level 2 were questions {2, 4, 9, 10}, which also focus on variable declaration. It is interesting to see that a problem arises when the declaration of variable names comes into play.

6.1 Answering the research question

Could implementing Parsons Problems in Hedy be a promising tool to offer an even higher quality of learning experience?

It is clear that Parsons Problems in Hedy are a fun way to offer a concept for the third time. A user spends less time on reading and is immediately drawn to the dragging of the code blocks. Parsons Problems might not only be an effective tool in learning how to program, but also in emphasizing topics students struggle with. This implementation shows that combining the end quiz of Hedy with Parsons Problems could be a promising tool to extract types of mistakes that are made by users. This way certain topics that are difficult to grasp for novice programmers, like variable declaration, were brought to our attention by Parsons Problems. Developing more Parsons Problems to practice with these difficult topics could tackle the problem at hand, offering a higher quality in learning experience for children.

References

[DH94] Rick Decker and Stuart Hirshfield. The top 10 reasons why object-oriented programming can't be taught in cs 1. *ACM SIGCSE Bulletin*, 26(1):51–55, 1994.

[DLRD20] Yuemeng Du, Andrew Luxton-Reilly, and Paul Denny. A review of research on parsons problems. In *Proceedings of the Twenty-Second Australasian Computing Education Conference*, pages 195–202, 2020.

[EMR17] Barbara J Ericson, Lauren E Margulieux, and Jochen Rick. Solving parsons problems versus fixing and writing code. In *Proceedings of the 17th Koli Calling International Conference on Computing Education Research*, pages 20–29, 2017.

[FL89] Michel FAYOL and Patrick LEMAIRE. Une étude expérimentale du fonctionnement distinctif de la virgule dans des phrases: perspective génétique. *Études de linguistique appliquée*, 73:71, 1989.

[HCK16] Kyle James Harms, Jason Chen, and Caitlin L Kelleher. Distractors in parsons problems decrease learning efficiency for young novice programmers. In *Proceedings of the 2016 ACM Conference on International Computing Education Research*, pages 241–250, 2016.

[Her20] Felienne Hermans. Hedy: a gradual language for programming education. In *Proceedings of the 2020 ACM conference on international computing education research*, pages 259–270, 2020.

[NS96] Donald A Norman and James C Spohrer. Learner-centered education. *Communications of the ACM*, 39(4):24–27, 1996.

[PH06] Dale Parsons and Patricia Haden. Parson's programming puzzles: a fun and effective learning tool for first programming courses. In *Proceedings of the 8th Australasian Conference on Computing Education-Volume 52*, pages 157–163, 2006.

[Sha20] Mansi Shah. Exploring the use of parsons problems for learning a new programming language. Technical report, Technical Report EECS-2020–88. Electrical Engineering and Computer Sciences . . . , 2020.

[Shn77] Ben Shneiderman. Teaching programming: A spiral approach to syntax and semantics. *Computers & Education*, 1(4):193–197, 1977.

[Sim73] Jean Simon. *La langue écrite de l'enfant*. Presses universitaires de France, 1973.

[Swe88] John Sweller. Cognitive load during problem solving: Effects on learning. *Cognitive science*, 12(2):257–285, 1988.

[VV99] Marianne Verhallen and Simon Verhallen. *Woorden leren, woorden onderwijzen: handreiking voor leraren in het basis-en voortgezet onderwijs*. CPS, 1999.