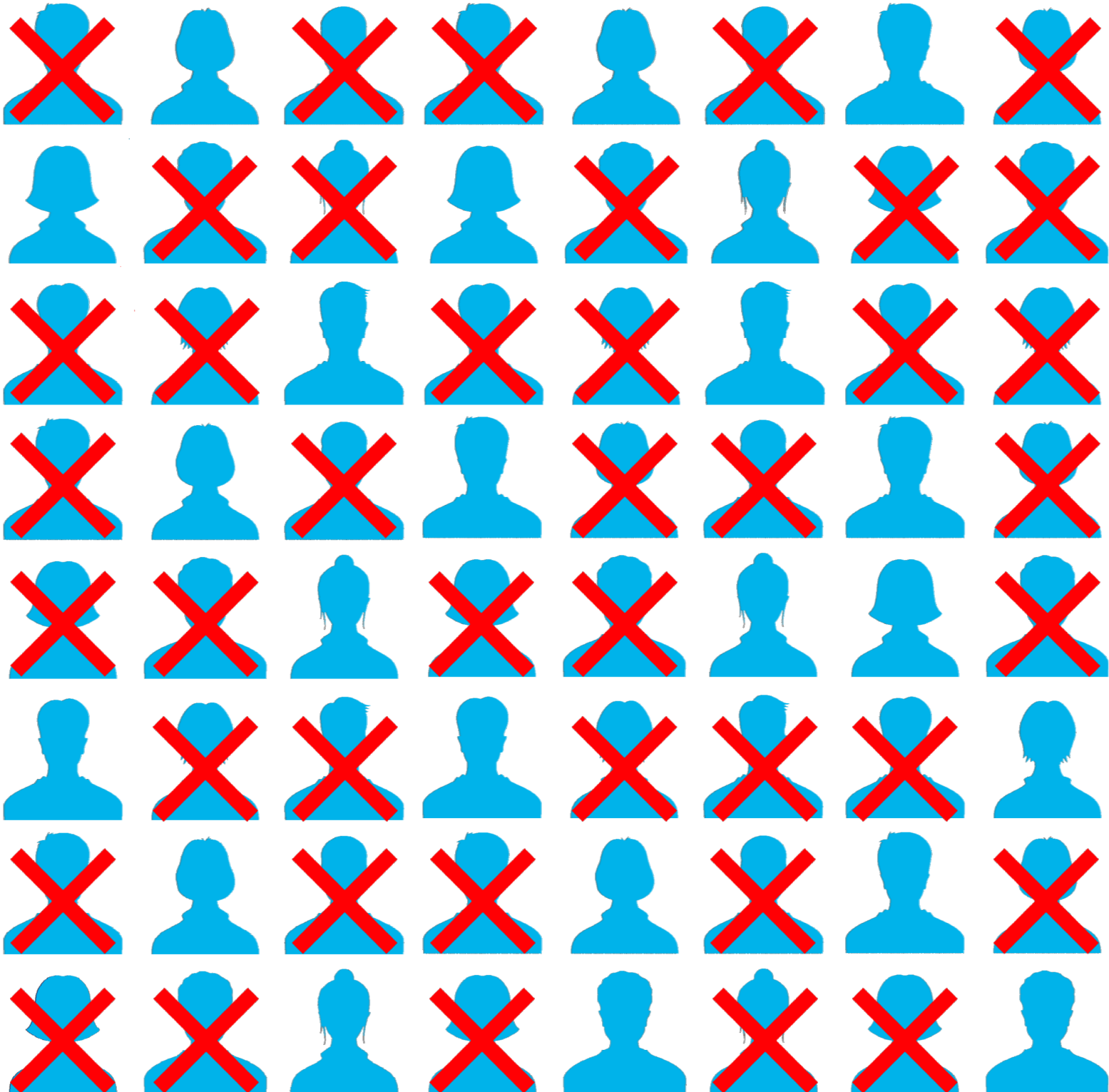



Generating Teacher Feedback through Parsons Problems

by

Lizzy Scholten



 = misconception #6 detected

Master Thesis

Generating Teacher Feedback through Parsons Problems

by

Lizzy Scholten

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on March 13, 2020 at 13.00

Student number: 4245008
Project duration: June 5, 2019 – March 13, 2020
Thesis committee: Prof. dr. A. van Deursen, TU Delft, supervisor
Dr. ir. W.P. Brinkman, TU Delft
Dr. ir. F.F.J. Hermans, Leiden University, supervisor

An electronic version of this thesis is available at <http://repository.tudelft.nl/>.

Abstract

As computer science lies at the heart of almost all technological progress, widespread computer science education, and particularly programming education, is of great importance. In order to reach a large group of students, secondary schools can play an important role. However, students have difficulty learning programming concepts. Programming is complicated and the education largely takes place on computers, which allows for little interaction with teachers. Consequently, teachers have difficulty gaining insights about the progress and misunderstandings of students and thus have little opportunity to intervene, which threatens to undermine the effectiveness of the learning process. Therefore, the purpose of this research is to improve on existing programming education by developing a feedback method for computer science teachers, which enables them to better understand and remedy programming misconceptions held by their students. Following a Grounded Theory approach, interviews were conducted with computer science teachers and programming education researchers in the Netherlands. Participants were asked to describe the problems they encountered in teaching programming concepts and to identify what in their views would help to improve their teaching. Furthermore, literature was reviewed on existing tools which offer teacher feedback. The majority of these tools appeared to provide little insight and/or require time-consuming analysis by the teacher in order to gain some insights. Teachers indicated that they have little time for complicated data analyses, but would highly value detailed feedback about their students' programming misconceptions. Furthermore, it was suggested that alternative testing methods, such as puzzles, might be useful. Based on these findings, Parsons Problems were proposed and tested as a solution for detecting programming misconceptions. The testing took place during two rounds, a small-scale trial experiment and a broader experiment in which 64 secondary school students participated. 'Primitive assignment works in opposite direction' was the number one misconception, held by 56% of the participants, and was the most occurring misconception across students. The 'invalid `else`-statement' misconception was the second most held misconception (53% of participating students), followed by 'primitive assignment works in both directions (swaps)' (45%), 'difficulties in understanding the sequentiality of statements' (39%), 'the natural-language semantics of variable names affects which value gets assigned to which variable' (20%), 'adjacent code executes within loop' (6%), and 'using `else` is optional' (5%). As 7 out of 8 misconceptions targeted in the experiment were successfully detected, this experiment demonstrates that employing Parsons Problems appears to be a viable method for misconception detection.

Preface

There are several people I would like to thank for making my master thesis period insightful and from whom I have learned many things along the way. Without their support, the result of this thesis would not be the same.

First of all, I would like to thank my supervisor Felienne Hermans, from Leiden University, for showing me a different aspect of computer science and providing me with many new insights. It was great to be a part of the many interesting brainstorm sessions about ways to tackle programming education difficulties. Felienne always made time to guide me through the thesis process. I am very thankful for her guidance and support.

I would also like to thank Arie van Deursen, my TU Delft supervisor, and Willem-Paul Brinkman, for being a part of my master thesis committee and taking the time to provide valuable feedback on the progress of this thesis.

Finally, I thank my friends and family for supporting me and always motivating me to do more, learn more and create the best possible result.

*Lizzy Scholten
Delft, February 2020*

Contents

Introduction	1
1 Literature Review	3
1.1 Teacher Feedback	3
1.1.1 Plug-Ins	3
1.1.2 Web-Based Applications	5
1.1.3 Stand-Alone Tools	6
1.1.4 Other Related Research	7
1.1.5 Summary	7
1.2 Parsons Problems	9
1.2.1 Design Choices	10
1.2.2 Current Application	10
1.3 Programming Misconceptions	12
1.3.1 Selection of Misconceptions	12
Part I	15
2 Interviews	17
2.1 Methodology	17
2.2 Interview Design	17
2.2.1 Introduction	18
2.2.2 Researcher	18
2.2.3 Teacher	18
2.2.4 Digitization/Automation	19
2.2.5 Other	19
3 Results Part I	21
3.1 Participants	21
3.2 The Interviews	21
3.3 Summary: working towards a solution	25
Part II	27
4 Pilot Experiment	29
4.1 Question Design	29
4.2 Questions	29
4.2.1 Variables	30
4.2.2 If/Else Statements	31
4.2.3 While Loops	32
4.2.4 For Loops	32
4.3 Setup and Participants Pilot Experiment	33
4.4 First Test Results	33
4.4.1 Participant 1	34
4.4.2 Participant 2	34
4.4.3 Participant 3	34
4.4.4 Participant 4	35
4.5 Conclusions Pilot Experiment	35
5 Experiment	37
5.1 Question Design	37

5.2	Questions	38
5.2.1	Variables	38
5.2.2	If/Else Statements	39
5.2.3	While Loops	39
5.2.4	For Loops	40
6	Results Part II	41
6.1	Setup and Participants	41
6.2	Findings	41
6.2.1	Targeted Misconceptions.	42
6.2.2	Targeted Misconceptions occurring in Non-Targeting Questions. . .	44
6.2.3	Other Misconceptions	44
6.2.4	New Misconceptions	44
6.2.5	Research Questions	45
7	Discussion	51
7.1	Part I	51
7.2	Part II	51
8	Conclusion	53
9	Future Work	55
9.1	Design of Parsons Problems.	55
9.2	Design of Experiments	55
9.3	Automation - Useful Functionalities	55
A	Experiment Questions and Answers	57
A.1	Variables	57
A.2	If/Else Statements	59
A.3	While Loops	60
A.4	For Loops	62
B	Experiment Google Forms	65
C	Experiment Collected Data	81
	Bibliography	83

Introduction

In a world that has quickly become increasingly dependent on technology, the future workforce needs to be adequately prepared to face the challenges that lie ahead. As computer science (CS) lies at the heart of almost all technological progress, effective and widespread education in this field is of the utmost importance [13] [34].

However, serious challenges can be discerned with regard to the scale and effectiveness of existing CS education. Most notably in that respect are a worldwide shortage of CS teachers [51] [58] [1], a relatively small number of secondary schools which offer CS classes [61] [51], and the fact that many students appear to have trouble learning programming concepts, which often manifests itself in the form of programming misconceptions. Misconceptions occur when students do not correctly understand the concepts of programming languages, how computers process the offered code, and therefore how code should be written in order to run properly and produce the desired result. It is important to teach programming concepts correctly so that students build correct models in their minds [27]. If a concept is remembered or construed incorrectly, it is important to intervene [29] and guide a student's train of thought back to the correct path. A further complicating factor is that programming education largely takes place on computers, which allows for very little interaction with a teacher. Consequently, it is very difficult for teachers to monitor the struggles and progress of individual students, which often undermines the effectiveness of the learning process.

As the role of the teacher is central in programming education, the main goal of this research is to design a method for teacher feedback, which enables teachers improve the effectiveness of programming education for their students. In this thesis, teacher feedback is defined as feedback which provides the teacher with insightful information about those things a student does not correctly understand, so that the teacher can intervene and adjust the teaching method if needed.

Main Contributions

Theory

The uniqueness of this research lies in the fact that it studies the combination of alternative programming exercises for the detection of programming misconceptions with the purpose of generating teacher feedback. In addition, this research uses a sample group of novice Python learners between the ages of 12 and 14, for whom a novel use of coding puzzles has been devised. With the previously mentioned misconceptions serving as reference, those puzzles have been designed with the specific purpose of detecting a selection of misconceptions. Furthermore, it appears that this research has brought to light new misconceptions, which had not been included in the extensive lists which resulted from previous research. Determining which misconceptions occur most often, and what goes wrong in a student's train of thought could be very helpful for the development and improvement of existing teaching methods. In order to further confirm or elaborate on the findings of this thesis, the exercises developed for the experiments in this research can be used for follow-up research, with or without adaptations.

Practice

The results of this thesis are also highly relevant for educational practice, i.e. for teachers, programming education researchers, educational institutions and their students. As the results of Part I of this research demonstrate, teachers and programming education researchers expressed the need for teacher feedback, since it is important that programming concepts are understood correctly so that students can build correct models in their minds. Should a concept be remembered or construed incorrectly, teachers need to – and want to - be able to intervene

and guide a student's train of thought back to the correct path. The fact that misconceptions have successfully been detected with this method is an important step towards generating insightful feedback for teachers. Moreover, many students partaking in the experiments stated that they enjoyed solving the Parsons Problems style exercises, even though many mistakes were made. This seems to indicate that teaching methods using Parsons Problems could not only generate useful teacher feedback but could also constitute an engaging learning method for the students. As such a teacher feedback tool would take over some difficult tasks of a teacher, it could also make it much easier for those teachers who are less proficient to provide quality education and perform at a uniform level.

Guidelines for Reading

This thesis is divided in two parts, which aim to provide answers to separate research questions.

Part I of this thesis is dedicated to analyzing the flaws in currently applied methods and processes through interviews with CS teachers and programming education researchers. Furthermore an inventory is made of their wishes, needs and suggestions with regard to the improvement of computer science education. Building on the results of Part I, Part II of this thesis proposes an alternative method for the detection of programming misconceptions with the purpose of generating insightful teacher feedback. The experiments conducted in order to prove the proposed method will be described and the results of the experiment will be presented. Consequently, the limitations of the findings in Part I and Part II will be discussed as well as the overall conclusion, followed by recommendations for future work.

Literature Review

In the following sections, this literature review addresses several key ingredients for the design of the teacher feedback method proposed in this thesis.

1.1. Teacher Feedback

As the main goal of this research is to design a method for generating insightful teacher feedback, this literature review aims to uncover which existing tools offer some kind of teacher feedback. The feedback offered by those tools will be further investigated in order to find leads for the development of an improved teacher feedback method.

The literature search effort produced a large number of researches on applications which predominantly provided student feedback. Therefore, a selection was made on the inclusion criterion that the researched applications provided teacher feedback for traditional text-based programming exercises. The choice for this type of tools was made on the assumption that free text-based coding could potentially provide a lot of detailed and varied information about students' programming behavior. Apart from the tools selected on this inclusion criterion, a few other tools were included in this review on the grounds that they collected data about students which could potentially be used to generate insightful teacher feedback, even if some of these tools did not provide any teacher feedback.

In order to create clarity about the type of tools to be discussed, they have been loosely categorized as follows:

- **Plug-Ins:** tools that are add-ons to existing applications, for example to Netbeans
- **Web-Based Applications:** tools that are web-based and function separately from other applications
- **Stand-Alone Tools:** tools that function separately from other applications, but that do require a certain input or future connection to an application in order to provide useful output

1.1.1. Plug-Ins

The following tools are plugins, which purposely collect data from coding environments in order to create some type of feedback.

Test My Code (TMC) can be used through its Netbeans plug-in. The Test My Code web interface gives students the option to register for a course, view the aggregated data that TMC has collected and submit exercises outside of the IDE. The TMC web interface gives instructors the opportunity to create courses, update exercises, view the students' submissions and review the students' code, find code review requests from students and view some statistics about their courses [59]. One of TMC's goals is to integrate scaffolding in its exercises. That way, students

can build their code piece by piece and receive feedback along the way, so that they learn to build their code correctly and validate the code each step along the way.

Retina is a tool which collects the student's name, date and time, and for each error, Retina collects the error type, file name, line number, and the associated error message [42]. Retina provides both a so-called Instructor View and a Student View. Instructors can either browse through individual student data or entire classes. Per student, they can select different assignments and view the previously mentioned data that Retina collects. Per class, the instructor can view how the class performed as a whole. In both scenarios, an overview is given of the most common error(s).

Karam et al. have developed an extension to BlueJ. BlueJ is an opensource IDE for Java, which was developed mainly for programming education, in particular object-oriented programming. The extension developed by Karam et al. consists of the following components: real-time code recorder, code repository, monitoring, typical error library, analysis, and feedback [36]. When students work on a project, code snapshots are collected periodically and upon compilation. They have compiled a list of sixteen compilation or run-time errors that novice programming students typically make. In case a typical error is found in the student's code, the line(s) in which the error occurs will be highlighted in yellow and the student can view hint(s) on how to solve the problem. Teachers receive a report on the common errors that were made by the students, including a list per error showing which students encountered the error and how long it took each one of them to find the solution. Students have the option to view snapshot of their code, which include errors or solutions to those errors. This lets them view their programming process and may help them learn how to solve or avoid making those errors in the future.

The goal of a Smart Lab is to provide teachers with insights about their students while they perform their assignments [5]. So-called 'Smart Labs' embody the strategy of equipping a lab with tools that improve instruction and learning [15]. Through the Code Transferer, the students' code is captured and sent to the central database. The Analyser/Visualiser makes it possible for the teaching staff to monitor students while they are working on assignments or at a later time. It provides an interface which tries to map students and their relative location in the classroom. Per student it shows which exercise the student is working on, the number of completed actions, the number of missing actions, the number of compilations and possible syntax errors. Furthermore, the system generates lists of Common Missing Actions and Common Syntax Errors and informs the teacher which student(s) need(s) the most or least attention, or if a student has been idle for more than two minutes. The teacher can also zoom in on a student. When zooming in, an overview is shown of the most current snapshot (it is possible to navigate through the other snapshots), the completed/missing actions, number of compilations and the history of compilation errors. It is also possible to view other exercises that the student has already attempted, other than the current exercise. Finally, it is also possible to view a final report/summary of a student's programming behavior.

CodeInsights is an IDE plugin tool that can autonomously capture real-time data about students through snapshots of their code [23]. Each time students run their code, a snapshot is automatically sent to CodeInsights, where it is then compiled and tested locally. The teacher's dashboard then shows some aggregated statistics about the students' behavior. In order to further support teachers who might not have time to continuously monitor the dashboard, a notification system has been set up. It will send a notification to the teacher in situations that might lead to problems, for example when students show a much higher than average amount of compilation errors or many more lines of code than average. The system will also keep track of how students are performing over time and how they perform relative to the rest of the group or class.

Marmoset is a system that supports automatic project snapshots, submission, and testing [56]. The system collects snapshots of the students' code, runs the teacher's tests and returns the results to the student. Teachers/instructors can view information regarding students such as whether students have passed test cases, the best submissions for each student and lists of students who have not yet made any submissions. The students' data gives instructors insights into which test cases are tough, from which teachers could deduce which topics should be dealt with more often in class. Using Marmoset means that the teacher has to do all the

work for the project before releasing it to students, as the system does not appear to be flexible for changes during the project timeline. The instructors that worked with Marmoset noticed that some students indeed started working on their projects earlier than they might have done without Marmoset, but that other stubborn students were not triggered by Marmoset to start their work.

The Blackbox project collects student data from BlueJ users. Students can choose to opt in and allow Blackbox to collect data concerning their programming behavior, such as source code, edit sequences, testing and execution interactions, and compilation results [11]. The goal is to provide researchers with an extensive collective database of student programming behavior data and to stimulate them to perform research that might otherwise not be possible. Brown et al.'s research shows that not all data collected by Blackbox has been useful. Also, in their survey, they received feedback that (potential) researchers were missing more contextual data. By 2018, eighteen researches have been performed by ten different groups, using the Blackbox data. In these researches, the most popular research topic was related to analyzing programming errors.

The ClockIt BlueJ Data Logger and Visualizer consists of two separate extensions. One extension registers event listeners for compile, package and invocation events [43]. The second extension visualizes the data that has been collected by the event listeners. Due to BlueJ being able to only record the first error during a compilation, only one error per compilation can be reported. The ClockIt web interface makes it possible for instructors to view data concerning their students and entire classes, and students can also view their own data. This data includes the types and amounts of compiler errors, number of lines of code and comments created over time, and an indication of the amount of time students spend on programs per day.

Jadud performed a research with first year university students at Kent University, studying their programming behavior while editing and compiling their programs. The students worked on their assignments in BlueJ. Every time students compile their code, some data is stored, such as a copy of the source code, the compiler output and other meta data [35]. Jadud developed 3 tools. First, a code browser which makes it possible for teachers to view the source code of successive compilations of a student's program. Secondly, a visualization of a student's programming behavior, which lists the types of errors that occurred in a session, including the time was spent until the next compilation, the number of characters changed between compilations (negative for deletes and positive for additions), and a visualization of where in the code an error was located and where the student decided to continue programming in response to that error. Thirdly, Jadud has used an algorithm to determine the Error Quotient (EQ) of a programming session. This EQ rates each programming session to determine how much or how little students struggle with syntax errors while programming [35]. An important insight for Jadud was that his research showed him which errors students encountered most often. It appeared that students spent most of their time on only a few different error types.

1.1.2. Web-Based Applications

The following web-based applications are online learning environments, which in some form provide the opportunity for students to perform exercises and of which data is being collected that could be translated into teacher feedback.

CodingBat is a website that offers a series of exercises in Java and Python [2]. It is possible to create an account, so that all results will be saved. Also, users can share their results with a teacher by sending it to the teacher's CodingBat account. The teacher can then view how many exercises each student has completed, and by zooming in, the teacher can view which tests their code has passed. Another feature is the Progress Graph, which maps the (number of) compiling attempts over time, as well as the level of correctness of the piece at that moment in time.

CloudCoder, which is based on CodingBat, is an open-source web-based exercise system [44]. CloudCoder supports exercises in Java, Python, C/C++ and Ruby. When working on an exercise, the application shows the exercise description, the code editor, test results and possible

compiler errors. From the command line, it is possible to create courses and register students in a course. Consequently, a teacher can create exercises and add them to a course. Users can freely import exercises from the CloudCoder repository or share exercises through the CloudCoder repository. CloudCoder also collects a lot of data concerning the programming behavior of students. All code edits, submissions and test outcomes are saved with the aim of providing some extra insights for teachers and researchers about the exercises and the students.

The Hour of Code event takes place every year in the Computer Science Education Week, but an Hour of Code can also be hosted by someone separate from the event. It consists of many coding programs with a duration of one hour. It facilitates several programming projects and consequently collects data about the students' submissions of the projects. A research by Basawapatna and Repenning has used the data collected in the Hour of Code projects to try and determine at what point student retention levels drop significantly and what the reason for this drop could be [9]. For this research, the Hour of Code project XML files for 2014 and 2015 were searched for the Lines of Code (LOC), in order to find out how far students usually made it through an exercise. By tweaking the exercise experience after 2014, it was analyzed if the performance of students in that exercise would improve between 2014 and 2015.

WebToTeach is a web-based tool that automatically checks students' programming homework [6]. It was developed at Brooklyn College and used locally. There are three user groups of WebToTeach, i.e. guests, students and teachers. Guests can do the available exercises, but their track record or previous submissions will not be saved. Students can see which exercises they (still) have to do, can view their deadlines, and they can perform programming work within the WebToTeach environment. It is possible for teachers to provide hints, and as soon as a student is done with the exercise, they can submit their solution. They will receive immediate feedback as to whether their submission is correct and has been accepted or whether it was faulty and therefore not accepted. In case of acceptance, the student's roster is immediately updated and in case of failure, the student will be informed what was wrong and what should be worked on. Teachers can create exercises/assignment, provide hints, create tests, maintain roster information, view correct submission of students' work, broadcast messages to students and keep track of homework completion.

1.1.3. Stand-Alone Tools

The following applications are focused on providing student insights based on code snapshot data. However, since they are not actually integrated in existing learning environments, they require explicit data input by the user, in this case the teacher.

CodeBrowser is a web-based application that facilitates code snapshot analysis [28] and is structured so that data can be fetched across REST-APIs. A user can choose to either browse courses or students. When viewing student data, the user can see in which courses the student participated and the exercises on which the student has been working. When selecting an exercise, a Snapshot View appears, showing a timeline, file browser, snapshot(s) and a tagging option. The timeline shows when snapshots have been taken over time and how many changes were applied compared to the previous snapshot. Users have the option to view two consecutive snapshots and the relative differences. Also, users can place tags when going through the snapshots.

SnapViz is a web service that visualizes code snapshot data from tab-delimited data files. This way, multiple snapshot driven systems can use SnapViz as a common visualization system. Its online requirement is that the tab-delimited data file contains the following column headings: Primary attribute, Secondary attribute, Timestamp, Score (either % or 0/1), a Label or Tag [7]. In a graph, SnapViz visualizes when compilation events took place, at what time and on which day, showing a dot at that spot in the graph. It uses hues of the color red and green to indicate the correctness of the code.

Espresso is a pre-compiler tool that recognizes a certain pre-defined list of compiler errors and returns these in an easily explained manner to the student [30]. Hristova et al. compiled a list of 20 Java programming mistakes that occur often and are important to explain well. This list is

based on a survey of Computer Science professors, teaching assistants and students, combined with their own knowledge. Their goal for Espresso was to improve upon the existing compiler error messages and to also provide hints as to how to tackle the solutions. Espresso would be most useful in an introductory programming course.

1.1.4. Other Related Research

A number of other related researches have been studied with the purpose of gaining additional insights, which might be useful for the development of insightful teacher feedback. The following studies were included in this thesis, because they address topics such as which type of data could be collected and through which methods these data could be collected, amongst others in order to facilitate intervention.

Ihantola et al. have analyzed a wide range of applications which collect educational data. They found that approximately 45 of the analyzed researches (48%) studied Java and only 8 (11%) studied Python. Several levels of data granularity have been established: keystrokes, line-level edits, file saves, compilations, executions, and submissions. Their results show that there seems to be a trend towards collecting high-level data (e.g. submissions) versus low-level data (e.g. keystrokes), which has only seemed to have gained some traction during more recent years [33]. These differences in granularity can serve applications with different purposes, for example automated grading systems, IDE instrumentation, version control systems, and key logging. Some extra, and useful, functionalities that several tools included are visualizations or summaries of data, and the option to export data.

Hundhausen et al. suggest and promote the use of IDEs, not only to support computer programming, but to collect learning process data and enable intervention techniques [31]. The idea of using IDEs for these purposes is based on the argument that students already spend a lot of time doing actual programming in IDEs. Moreover, IDEs can quite easily be instrumented with data collection facilities and can automatically collect data while students perform their programming exercises. It can be useful to provide feedback or interventions dynamically while the student is working in the IDE. Several IDEs are instrumented to capture programming process data such as edits or compilation errors. Some ideas are offered on additional data that could possibly be collected, for example through offering survey or quiz questions or the use of eye tracking data. Hundhausen et al. discuss different types of data that can possibly be collected through IDEs, but a substantial part of their research concerns the design principles that could be used when creating programming process interventions. Their research emphasizes that up till now, little research has been performed concerning IDE interventions. A few existing examples of interventions are mentioned, such as dynamically tailored feedback, incentive mechanisms, enhancing syntax errors so that they are more understandable, dynamically generating hints based on patterns, awarding badges to students who reach certain time management or learning goals, and scaffolding the learning materials by splitting up exercises into smaller steps. They propose a cyclical process model for IDE based learning analytics, which consists of the phases 1) Collect data, 2) Analyze data, 3) Design intervention, and 4) Deliver intervention. This model is then used to explore the design space of IDE based learning analytics. Despite the advantages of using IDEs for learning analytics and interventions, its disadvantage is that implementation skills do not directly transfer between IDEs, as each plug-in architecture has its own standards and libraries. A possible solution could be a standard API for IDE plug-in implementation.

1.1.5. Summary

The purpose of this literature review was to gather information about existing tools that provide teacher feedback. Therefore, based on the information found in literature, we made a subjective attempt to categorize the selected tools on the criteria whether they appeared to provide 'insightful teacher feedback' and whether or not 'time-consuming' analysis appeared to be required. Feedback categorized as 'insightful' was deemed to provide information to teachers which would enable them to intervene in the learning process by offering additional explanation about a specific programming concept. 'Time-consuming' indicates whether or not a teacher appears to need a lot of time to analyze the provided data in order to extract usable information.

The following table shows which of the reviewed tools provided teacher feedback.

Tool	Type	Teacher Feedback		
		Teacher Feedback	Insightful	Time Consuming
Test My Code	Plug-In	Yes	Unknown	Yes
Retina	Plug-In	Yes	Yes	No
Karam et al.	Plug-In	Yes	Yes	No
Smart Labs	Plug-In	Yes	Yes	Yes
CodeInsights	Plug-In	Yes	No	Yes
Marmoset	Plug-In	Yes	Unknown	Yes
Blackbox	Plug-In	n.a.	n.a.	n.a.
ClockIt	Plug-In	Yes	Yes	Yes
Jahud	Plug-In	Yes	Yes	Yes
CodingBat	Web-Based	Yes	No	No
CloudCoder	Web-Based	Yes	Yes	Yes
Hour of Code	Web-Based	No	n.a.	n.a.
WebToTeach	Web-Based	No	n.a.	n.a.
CodeBrowser	Stand-Alone	Yes	Unknown	Yes
SnapViz	Stand-Alone	Yes	No	No
Expresso	Stand-Alone	No	n.a.	n.a.

Table 1.1: Existing tools reviewed on teacher feedback usefulness

Table 1.1 shows that the majority of tools selected and reviewed for this research do provide some sort of teacher feedback. After taking Blackbox (only data collection), Hour of Code, WebToTeach and Expresso (no teacher feedback) out of the equation, 50% of reviewed tools appear to provide insightful teacher feedback. The remaining 50% of tools provide teacher feedback which does not enable teachers to intervene in the learning process of students. However, with the exception of Retina and Karam et al., all tools which do provide potentially useful feedback require that teachers spend a lot of time on data analysis before they can potentially intervene. This raises questions about whether teachers will ultimately intervene, because they need to dedicate considerable time to analysis and need to possess the skill to analyze the provided data.

Some of these applications also provide students with feedback on their progress. However, this same information is often not provided to their teachers. The student feedback mainly consists of error messages during the coding process. When students have finished an assignment, some tools also offer the option to review snapshots which have been made upon compilation of the code. However, students need to take the initiative to analyze this data and have to draw their own conclusions, which may be too complicated or time-consuming for many students.

The researched tools are often very specific and operate between narrow boundaries, offering functionalities which serve only a distinct purpose. A lot of focus also appears to be on the topics of syntax and compilation errors, how they can be saved, responded to or relayed to teachers. However, it appears that the data collected is often not very suitable for the purpose of generating insightful feedback. An exception may be the data related to most common errors, as here an attempt has been made to make the gathered data and resulting feedback more specific and therefore more useful for intervention.

For the purpose of this research, three of the reviewed tools appeared particularly interesting, because they seem to provide insightful feedback to teachers or, in the case of Expresso, to students. These three tools have been highlighted in Table 1.1. The aspect these tools have in common is that they generate feedback, based on a pre-defined list of most common errors. Through Instructor View, Retina facilitates that teachers can access an overview of most common errors per student or class. Furthermore, Karam et al. also provide a list of most common errors. However, other information provided by these tools may once again require quite some analysis by the teacher. Expresso, which does not generate teacher feedback, may offer some inspiration for this research because it does generate student feedback including hints based on a pre-compiled list of most common errors.

1.2. Parsons Problems

As the main purpose of this thesis is to improve on currently existing programming education by developing a method which generates insightful teacher feedback, this section of the literature review explores the possibilities of Parsons Problems as an alternative testing method for the purpose of collecting data that can be used for such feedback.

Parsons Problems (PP) are code puzzles that consist of different puzzle pieces of one or several lines of code. Students receive these puzzle pieces in a randomized order and need to assemble them in the correct order to solve a question. An example of a Parsons Problem is illustrated in Figure 1.1. The code puzzle pieces on the left need to be dragged to their correct position on the right side of the screen in order to create the solution.

Your First Parsons Problem

Your task: Construct a Python program that prints strings "Hello", "Parsons", and "Problems" on their own lines. You can get feedback on your current solution with the feedback button. You should construct your program by dragging and dropping the lines to the solution area on the right.



Figure 1.1: Parsons Problem example [3]

There appear to be advantages to using Parsons Problems for computer science education in beginners' courses [45]. These advantages include: 1) maximized engagement due to gamification, 2) high level of planning, which encourages logical structure and modularization, 3) through the use of distractors, students are intentionally tempted to make common mistakes, 4) when students are asked to continue answering a question until the answer is correct, they will be exposed to the correct answer code, and 5) when automated, Parsons Problems can give immediate feedback on whether a choice for a puzzle piece is (in)correct [45].

In his Cognitive Load Theory, John Sweller makes a distinction between three types: intrinsic, germane and extraneous cognitive load. Each type of cognitive load has a different source. Intrinsic cognitive load is the load due to the complexity of a problem, extraneous cognitive load is the load due to the added complexity of the instructional material, and germane cognitive load is the load originating from the processing and creation of schemas in long-term memory [57]. A schema is used to describe a pattern of information organization and interpretation. It is important for instructional material to be set up in such a way that the extraneous load is as low as possible and working memory is freed up, in order to allow the germane cognitive load to be free enough for the correct construction of schemas. If the working memory is overloaded due to the instructional material, this can obstruct the correct learning process. The amount of cognitive load that a student experiences, is based on a student's prior knowledge. If the intrinsic load is too much, it could be interesting to split up problems into simpler problems. Parsons Problems take traditional programming exercises to a higher level of abstraction, which translates into a lower level of cognitive load for novices. Therefore, PP could be an attractive educational method for programming novices.

A research by Ericson et al. has found that students prefer Parsons Problems to other low cognitive load exercises such as Multiple Choice Questions, or high cognitive load exercises such as writing code [18]. Harms found that 80% of the students in his research enjoyed PP more than the tutorials [26], which would show a step by step textual instruction on how to execute that step. Other research shows that students that practise with new programming concepts through PP, perform 26% better on transfer tasks and take 23% less time completing exercises than students who learn the concepts through coding from scratch [25].

1.2.1. Design Choices

When creating a Parsons Problem, there are several design options to choose from to extend their functionality and their purpose. Each design choice or combination of design choices will determine the complexity and difficulty level of the Parsons Problem. The different alternatives, or additions to Parsons Problem design to be considered are:

- **(Paired) Distractor:** a distractor is a puzzle piece that is incorrect, in the sense that a student should not use it when composing the solution. It is possible to design the question so that every correct puzzle piece has a counterpart distractor [45], however, it is of course also possible to use no distractors or as many distractors as one wishes to use. Besides the number of distractors used, it is also a choice whether to randomly order all puzzle pieces, or to randomly order the correct puzzle pieces while pairing them with their incorrect counterparts. Solving PP with distractors should help students to build an understanding of the common mistakes that can occur [45] [24]. It has been found that solving PP with paired distractors is a more efficient method for practising coding than writing code or fixing code [19]. Although it is possible that distractors may increase the cognitive load for students [27], they are very useful and necessary tools to illustrate a specific mistakes
- **Two-Dimensional:** Ihanola and Karavirta introduced the concept of Parsons Problems in two dimensions. In which the vertical dimension represents the order of the puzzle pieces and the horizontal dimension represents the indentation of these puzzle pieces [32]. Such a concept is especially useful when creating PP for languages such as Python, in which indentation plays an important role in indicating which lines of code execute within which constructs
- **Partially Completed Code Pieces:** code puzzle pieces could be created with parts of the solution left open, for the student to fill out. This concept implements scaffolding on a very small scale [21]. This gives the opportunity to further test a student's understanding and logical thinking. It does, however, raise the complexity of the problem [29], raising the cognitive load level
- **Context:** it can be useful to already place some puzzle pieces in the correct location, around which the unsorted code puzzle pieces are to be placed by the student [22] in order to create more focus on a specific topic
- **Multiple Lines:** it can be useful to combine several lines of code into one puzzle piece, in order to, for example, avoid interchangeability when the same line appears more often in a solution [40]

1.2.2. Current Application

Parsons Problems are already being used in programming education as an alternative assessment tool instead of traditional coding exercises. Multiple researches have been done on the different ways to design Parsons Problems, and the possible situations in which PP can be used. Additionally, several tools have been created in which PP can be completed. These tools vary with regard to general functionalities, feedback options and user interface. Parsons Problems' creators Parsons and Haden used Hot Potatoes for the creation of their Parsons Problems. A teacher only needs to fill in the question and the corresponding code pieces, to create a Parsons Problem. At run time, the puzzle pieces will be randomized. A student needs to drag those randomized pieces from the right side to the left side and then place them in the right spot. Parsons and Haden received feedback from students, which indicated that they would prefer more exercises per problem type and that they would like to be provided with more detailed feedback on their errors [45].

One very well known web-based platform for PP is js-parsons [32]. Js-parsons is an open source JavaScript widget, under MIT licence, which makes it possible to embed Parsons Problems in HTML with drag-and-drop functionality. A few other advantages are that js-parsons offers two-dimensionality, toggle-able parts in the code, and that js-parsons can collect log data on how students construct their solutions and send it to a server. When checking an incorrect solution, students receive highlighted feedback on the first line that is incorrect. The feedback addressed

both the location and the indentation of a puzzle piece. Js-parsons facilitates easy integration of PP in the existing curriculum of learning environments such as Codio [14] or Ebooks such as Runestone [4].

The creators of js-parsons built upon their own js-parsons platform with the creation of MobileParsons. Since PP consist of code puzzle pieces which only have to be moved to the right location by the user, PP appear to be suitable for mobile devices. MobileParsons supports automatically assessed PP on IOS and Android mobile devices [37].

PyKinetic, a Python tutor that supports PP, is another smartphone based application. These PP focus on implementing scaffolding in the exercises through the use of partially completed code pieces. PyKinetic also provides automatic indentation. Especially low performing students seem to learn more from PP than high performing students, for whom a ceiling effect appears to apply [21].

ViLLE is a tool that was originally created with the purpose of visualizing program code execution. More recent versions contain the functionality of developing Parsons Problems [49]. Its functionalities allow for context puzzle pieces to be provided. Distractors, however, are not supported by ViLLE. In case a student's solution is correct, the feedback will include a score and the option to visualize the execution of the solution. In case the answer is wrong, an error message will appear.

Amruth Kumar has created a tool named Epplets, which facilitates adaptive Parsons Problems [40]. Lines of code are offered in the Problem Panel and a student has to drag each of them to the Solution Panel or the Trash Panel. Once a solution is submitted by the student, feedback is provided about which line is not put in the right position. The exercises a student receives depend on the ability a student has shown in previous PP exercises. Each PP exercise is designed with a certain learning objective in mind. If a student shows trouble with a particular learning objective, the student will receive additional puzzles until the student demonstrates to have fulfilled the learning objective. Barbara Ericson also proposes adaptive PP through the use of intra-problem and inter-problem adaptation. Intra-problem adaptation entails that when a student is not able to solve a problem, the problem itself will be simplified, and inter-problem adaptation means that the next problem will be based on the student's performance in relation to the previous problem. Ericson et al. have shown that solving PP through intra- and inter-problem adaptation is more efficient, but equally effective, as writing code [20].

Looking Glass is a programming environment for beginners with 3D animated exercises. which offer the possibility to create programming completion problems similar to Parsons Problems [27]. In order to help reduce extraneous cognitive load, Looking Glass offers scaffolding options for the user interface. However, Looking Glass does not support two-dimensionality, which limits the amount of topics that can be integrated in the PP exercises.

Vitel et al. have developed EvoParsons [60], which is a software implementation of PP, designed to facilitate ready-made puzzles and automatically generated puzzles. Each puzzle is created by taking a code program from the program library. Next, the puzzle will be split into code fragments and one or more transformations from the transforms library will be applied to the code fragments, in order to create 'bugged' or incorrect pieces of code (distractors). This transforms library contains semantic and syntactical misconceptions, such as capitalized keywords ('Main' instead of 'main') and removed semicolons [8]. Students have the option to request a hint, which means that one misplaced fragment will be highlighted. In EvoParsons, code fragments are automatically indented, which makes the exercise easier to solve.

Helminen et al. developed a tool which traces the full trails of student interactions when solving Parsons Problems. This tool records all changes in user input, all changes to the solution, and time stamps. The goal was to track students' programming process, analyze their problems and provide specific feedback. With the analyzed data from this research, Helminen et al. created graphs, which show patterns and anomalies concerning the student's problem solving process [29]. For example, these graphs showed that students often preferred to start the problem solving process by adding code pieces that defined control structures such as `for` loops

and if statements. Through these interactive graphs it is possible to observe the most common incorrect code states that students created for each assignment.

In conclusion, previous research has focused on 1) proving that PP are a viable alternative for traditional coding assignments, and 2) the development of environments in which PP can be created, solved, and sometimes analyzed.

1.3. Programming Misconceptions

In light of the main goal of this thesis, namely to improve on programming education by developing a method for generating insightful teacher feedback, this section of the literature review examines programming errors, i.e. programming misconceptions, of which the efficient detection would contribute to that purpose.

In order to develop effective programming education, it is very important to understand which things students tend to struggle with. While novices learn how to program, they can - and will - make mistakes. These mistakes are often related to so-called 'misconceptions'. These misconceptions can have varying causes. For example, it is possible that novices do not fully grasp a certain programming concept. Sometimes their understanding of a concept is the opposite of the reality, and sometimes their understanding is incomplete and only consists of half of the entire concept. Some mistakes or misunderstandings are simply vague and cannot be clearly defined, but nonetheless those are mistakes that novices make [55].

One of the researchers who has done a lot of research on the topic of programming misconceptions is Juha Sorva. Through exploratory research, Sorva has compiled a list of 162 programming misconceptions [55]. The misconceptions in this list often occur for content which is typically covered in introductory programming courses (at higher education level). Tobias Kohn is another researcher who has performed in-depth research on programming misconceptions. Kohn has composed three lists with a total of 25 Python programming misconceptions, which he deemed most important. The first list contains mistakes based on programming misconceptions about syntax and semantics. The second list contains only syntax errors, such as missing brackets, while the third list consists of mistakes that do not violate the grammar, but which still pose problems for novices [39].

1.3.1. Selection of Misconceptions

As a starting point for this part of the thesis, it was decided to take the aforementioned two collections of programming misconceptions, and filter those lists on suitability for this research. First of all, it needed to be possible to translate the chosen misconceptions into PP. Parsons Problems would seem to be particularly suitable for exercises that are not too long or complex and do not consist of an exceedingly large number of code lines. When designing PP for the detection of misconceptions, it was therefore deemed important to limit the answer space in order to make the resulting data useful for analysis. This led to the second criterion, i.e. that the misconceptions should be applicable for novice learners, as the code complexity and concept difficulty applicable to novices is probably low enough for these PP to deliver distinctive conclusions. Python is the programming language of choice for this research, because it is often used in practise and in secondary school education and has so far been under-researched [33].

Therefore, the inclusion criteria for misconception selection were formulated as follows:

1. Seemingly applicable for translation into Parsons Problems
2. Applicable at novice level
3. Applicable for Python

In previous research by the author of this thesis, a selection of Sorva's programming misconceptions list [55] has been made [50]. This list was already filtered on applicability at novice level and applicability for Python. Therefore, this list only needed to be further filtered on suitability for translation into Parsons Problems. Furthermore, a selection of misconceptions was made from Kohn's list of Python misconceptions related to syntax and semantics [39]. This selection was

narrowed down by applying the conditions that the misconceptions should be suitable for novice learners and for translation into PP. This led to the following list of misconceptions considered for this research:

1. **Values are updated automatically according to logical context:** novices expect that values update automatically, depending on the context [48]
2. **Difficulties understanding the lifetime of values:** novices often do not comprehend the temporal scope of value assignment [17]
3. **Magical parallelism: several lines of a (simple non-concurrent) program can be simultaneously active or known:** this misconception entails that it is often believed that lines of code can run in parallel. For example, if a piece of code contains the `if` statement 'if $x > 4$:', and later on in the code x becomes larger than 4, it is believed that the previously occurring `if` statement then executes as a reaction to the `if` statement now becoming true [46]
4. **A variable can hold multiple values at a time / 'remembers' old value:** this misconception is one that is widely discussed in literature, since novices often have the misconception that a variable can hold multiple values, or remembers its old values when a new value is assigned [17] [54] [47] [53] [16]
5. **Variables always receive a particular default value upon creation:** students often forget to initialize their variables, for example when keeping a running total [17]
6. **Primitive assignment works in opposite direction:** this is a misconception that often stems from Mathematics. Novices think that variable value assignment works from left to right, instead of from right to left [17] [41]. Parsons Problems appear to offer good opportunities to detect this misconception through the use of a distractor that contains the wrong assignment direction. For example, when the correct solution contains the assignment " $d = e$ ", a distractor could be " $e = d$ ". In this example, students have to make a very conscious choice. If they choose the wrong answer, this is a strong indication that they truly do not understand the concept of primitive assignment direction
7. **Primitive assignment works both directions (swaps):** novices often have the misconception that variable value assignment works in both directions, meaning the $x = 3$ and $3 = x$ would both be valid [39], or sometimes even that $x = y$ entails that the values held by variables x and y are switched [53]
8. **The natural-language semantics of variable names affects which value gets assigned to which variable:** when using 'meaningful' variable names, some novices incorrectly believe that these variable names are also meaningful for the computer and that the computer will handle their values accordingly. For example, that a variable named 'even' will automatically hold the even number(s) [53]
9. **Difficulties in understanding the sequentiality of statements:** there are several often occurring misconceptions concerning this topic. For example, novices forget that every line of code has an effect on the environment in which the next line will be executed. Another misconception is that the computer will skip back and forth through the code in order to execute the lines in the desired order [52]. Parsons Problems clearly seem to be suitable to detect this, as this method relies on the cutting up of lines which need to be put in the right order by the student
10. **Code after `if` statement is not executed if the `then` clause is:** novices seem to have trouble understanding that when the condition in an `if` statement is true, the code within the `if` loop is executed. Sometimes novices have the opposite incorrect understanding that the content of an `if` statement is not executed when the `if` statement condition is true [17]
11. **`if` statement gets executed as soon as its condition becomes true:** this misconception is closely related to the magical parallelism misconception (number 3 in this list). Novices believe that if an `if` statement condition becomes true due to lines that execute later on in the code, the computer will reevaluate the `if` statement as true and execute it [46]

12. **Both then and else branches are executed:** novices think that if and else statements will both always be executed, no matter if the condition(s) are true or not [53]
13. **The then branch is always executed:** novices think that an if statement is always executed, whether the if statement condition is true or not [53]
14. **Using else is optional (the next statement is always the else branch):** novices sometimes think that the code that comes immediately after an if statement is part of an else statement, but that using the actual 'else:' statement is optional. This is incorrect, since code that comes after an if statement is not in any way related to the functionality of the if statement, and is executed in any case
15. **Adjacent code executes within loop:** novices believe that for example, a print statement positioned after a loop would print something on every iteration of the loop. However, all it does is print the final value of the thing it is supposed to print [53]
16. **Difficulty in understanding automated changes to for loop control variables:** it is difficult for students to understand that a for loop variable changes on every iteration of the for loop, and that as a consequence, if the variable is used within the loop, it will lead to different results on each iteration of the loop [17]
17. **while loops terminate as soon as condition changes to false:** novices hold the misconception that it is constantly monitored whether the while loop condition becomes true throughout the iteration of the loop [46], meaning that they also incorrectly believe that if the loop condition becomes untrue throughout the iteration, the while loop will terminate [17]
18. **for loop control variables do not have values inside the loop or their values can be arbitrarily changed:** it is believed that for loop control variables are only functional in the for loop condition, and that they do not and cannot have any other function within the loop [53]
19. **Confusion between an array and its cell:** novices often get confused about arrays, what is the array as a whole and what is a single cell within that array [17]
20. **Difficulties with dealing with 2D array subscripts and dimensions:** this misconception consists of two parts. Firstly, novices are easily confused whether an array subscript refers to a certain cell or to the value stored within that cell. Secondly, when dealing with two dimensional arrays, novices are often confused which subscript refers the columns and which subscript refers to the rows [17]
21. **Difficulties with arrays containing indices as data:** when an array cell is referenced in statements such as $a[3] = a[3] + 5$, confusion arises about whether the number 3 or the value in $a[3]$ is added to 5 [17]
22. **Invalid else-statements:** one misconception is that novices tend to use a condition in the else statement, similar to how one would use a condition in the if statement. For example, when code includes a statement such as 'if (x > 4):', many students think this should be followed by a similar condition, namely 'else (x < 4):' instead of just 'else:'. Another misconception is that novices believe that the else statement is part of the if statement, and should therefore be positioned inside the body of the if statement [39]
23. **Incorrect structure:** novices sometimes create incorrect structures, for example by placing entire programs within a loop [39]

Part I

The first step of this research is directed at gaining a better understanding of programming education in secondary schools with a focus on the difficulties that teachers encounter. Furthermore, an inventory will be made of the needs and wishes of teachers which have not been satisfied yet, but which would in their view make a difference with regard to the effectiveness of programming education.

Research Questions

Therefore, the research questions for the first phase of this research are:

1. Which difficulties do participants see with regard to the teaching process?
2. What avenues for improvement or further research do the participants see?
3. How could digitization or automation contribute to improving programming education?

Approach

In order to answer the Research Questions listed above, a number of interviews were conducted with Dutch secondary school CS teachers and researchers in the field of programming education.

2

Interviews

As the purpose of this research was to improve on existing programming education by developing a feedback method for computer science teachers, a series of interviews was conducted with secondary school CS teachers and programming education researchers. The goal of these interviews was to uncover the strengths and weaknesses in currently applied methods, form a more focused idea about how the effectiveness of programming education could be improved, and explore the possibilities of digitization and automation.

2.1. Methodology

For this research, a Grounded Theory approach was applied. That means that this research is not based on a hypothesis which is consequently proven to be true or false, but rather that the research focuses on an area of interest and that interesting thoughts or ideas are allowed to surface during the course of this exploration. The idea behind Grounded Theory is to find out what is out there, in order to create a theory or idea that is grounded in the data, instead of being based upon an already existing research [38]. This approach seems to fit well with the idea behind this thesis, namely to explore the current programming education landscape and to see where improvement can be made or further research can be done.

2.2. Interview Design

The interview questions were designed with the clear goal of maintaining the right balance between guiding the conversation enough towards the right area of conversation, while not steering the interviewee towards a certain answer or opinion. This was particularly important since the goal of the chosen research method was to explore adjacent ideas and facts related to the area of programming education in secondary schools. Also, it needs to be noted that, since the interviews concerned a certain area of interest, but were not very specific, it is possible that the interviews differ from each other with regard to conversational aspects and topics. Topics were also allowed to be refined along the way, between interviews, in order to work towards saturation.

The Research Questions were taken into careful consideration while designing the interview questions. The goal was to not ask these specific questions right out, but to create some open questions related to all the Research Questions. Some of the formulated questions were more suited to researchers and others more suited to teachers, so both groups were asked the questions most applicable for their specific profession. The questions can be divided into several subcategories, namely 1) Introduction, 2) Researcher, 3) Teacher, 4) Digitization/Automation, and 5) Other.

2.2.1. Introduction

This section contains a few basic questions which simply serve as an introduction to the conversation and simultaneously handle a few formalities.

- Do you mind if I record the interview?
- Could you please state your full name for me?
- Could you tell me about what it is that you do and how it relates to computer science education?

2.2.2. Researcher

The Researcher section contains questions that are more related to performing research and experiments and are therefore mainly suited to the programming education researchers who were willing to take part in this research.

- Could you describe the field in which you perform research?
- As an example, can you tell me something about your previous researches?
- Can you describe the data collection process you typically go through? Or went through in one of your researches?
- Can you give an example of the type of data you collect?
- How do you process that data?
- What type of analysis do you perform on that data? (E.g. visualizations, mutations, statistical analyses)
- What format is most ideal for different types of data?
- According to you, which part or step of the research process could be improved?
- Where in the research process could digitization or automation play an important role or contribute to improvement?
- According to you, in the field of computer science education, what could be interesting directions for future research?

2.2.3. Teacher

The questions in this section are mainly applicable for computer science teachers, since those questions are related to the educational process and the students.

- Have you ever given programming lessons?
- Could you please tell me something about those lessons.
- How would a typical lesson go?
- Which things do you notice that students typically understand well or pick up quickly?
- Why is that, do you think?
- Can you give an example of something that students struggle with? How did that go? Was the matter resolved?
- Which questions do students often ask?
- How do you evaluate if a student understands a (programming) exercise? Based on what information? How do you measure that?
- Can you give an example of how you test whether or not a student understands a concept?
- Can you give an example of a moment when you had to intervene while giving a lesson? In which way?

- How could digitization or automation help improve the computer science education process?
- Can you give an example of something you would like to understand better about students with regard to programming?
- How could you, or I, do something about that?
- Are there examples of information, which is already available or known about students (or their behavior), which could be useful, but with which nothing/little has been done yet?

2.2.4. Digitization/Automation

For this thesis, an interesting topic is to find out what the role of digitization/automation could be in programming education. This is why some extra focus is given to this topic through the following questions.

- What is it that an application would have to be able to do, in order to add value/be useful?
- Can you give an example of how such an application might have helped you in your past researches/lessons?

2.2.5. Other

At the end of an interview there is some space to ask a few final open questions.

- Is there anything else you would like to tell me?
- Would you possibly be open to participating in a follow up interview in a later phase of this research?

3

Results Part I

The interviews took place over a period of four months between July and November 2019. Each interview took roughly one to two hours, depending on the flow of the conversation and on the availability of the participant. New interviews were planned until saturation in information gathering had taken place and a focused direction for the continuation of this research had been developed. The term 'saturation' means that enough data has been gathered to satisfy the research questions and that no real new information came to light during subsequent interviews.

3.1. Participants

A total of seven participants were interviewed for this research: three CS teachers and four programming education researchers. The programming education researchers (two female and two male) are affiliated with Leiden University and Delft University of Technology. Two of them are also programming teachers at secondary schools. The three participating CS teachers (one female and two male) work at Dutch secondary schools. In Grounded Theory, there are several recommendations when it comes to the number of participants that should be interviewed. This number ranges from 5 to 30 [12], but the necessary number of participants is not strictly bound to this range.

The aforementioned participants were recruited through different channels. Programming education researchers were recruited through the Programming Education Research Lab at Leiden University. The secondary school CS teachers were recruited through an advertisement that was placed in the newsletter of the professional association I&I, which represents teachers and employees related to computer science. The advertisement simply stated a general message without any details about the research, in order to avoid guiding the conversation or thoughts before conducting the interviews.

All participants signed an Informed Consent Form, which explicitly states that they agree taking part in the interviews and that participants allow the interview to be recorded.

3.2. The Interviews

After each interview was conducted, the same analytical process was executed. Every interview was recorded and therefore had to be transcribed afterwards. After transcribing, two analysis steps were performed, namely identifying codes and open coding.

Identifying codes mainly entails generating descriptive words or phrases that correctly represent an interesting segment of data. This was done for the entire transcribed interview. While creating new codes, a constant process of comparison took place in order to determine whether codes were similar to or different from each other. Once all these codes had been created, a more general open coding step took place. This step entailed a deeper examination of the created codes, labelling them, determining how they compared with or differed from each other and

finally grouping them into larger concepts or categories.

This process was repeated for every interview. In some instances, the exact setup of the next interview was slightly altered, based on the results of the previously analyzed interview. New information from new interviews was added to the total picture. Therefore, the development of concepts and categories was part of an ongoing process.

Classification of the codes has thus led to the creation of a number of overarching concepts which represent those codes. In order to create a clear overview of the analyzed data, a visual representation was made of all codes and their overarching concepts, as specified in the following list:

1. Research topics (see Figure 3.1)
2. Difficulties encountered during the research process (see Figure 3.2)
3. Difficulties encountered during the teaching process (see Figure 3.3)
4. Practical recommendations (see Figure 3.4)
5. Possible functionalities for digital learning environments (see Figure 3.5)
6. Learning tools (see Figure 3.6)
7. Miscellaneous (see Figure 3.7)

In order to clarify how remarks during the interviews were translated into codes and then assigned to overarching concepts, this process will be briefly explained. For example, it was remarked that error messages are often confusing for students because they might refer to the line of code which follows the mistake and not the line containing the mistake itself, which leads to panic for the student. This remark was translated into the code 'error messages were not always clear'. This code was then categorized under the concept 'problems encountered during the teaching process', which can be found in Figure 3.3. Codes such as 'make assignments directed at specific misconceptions', 'repetition' and 'make sure that faulty answers correspond with misconceptions' were created to represent remarks pertaining to those subjects and placed in the diagram representing Practical Recommendations (see Figure 3.4), even though they may not necessarily have been meant as recommendations. However, during the grouping of codes into overarching concepts, these particular remarks were judged to possibly contribute to solutions and therefore placed in the recommendations diagram. In this manner, all relevant remarks and topics broached by the participants were coded and categorized (see Figures 3.1 to 3.7).

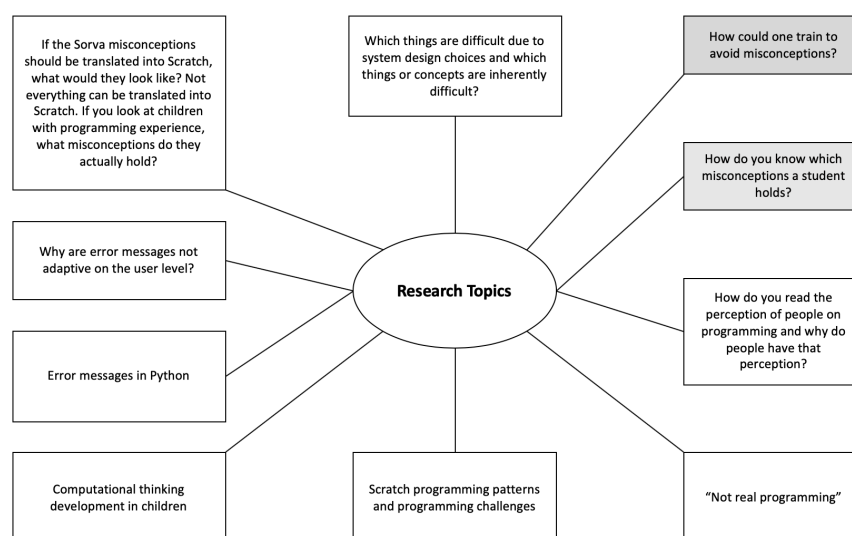


Figure 3.1: Research topics. The highlighted codes relate to the topics listed in the summary of this chapter

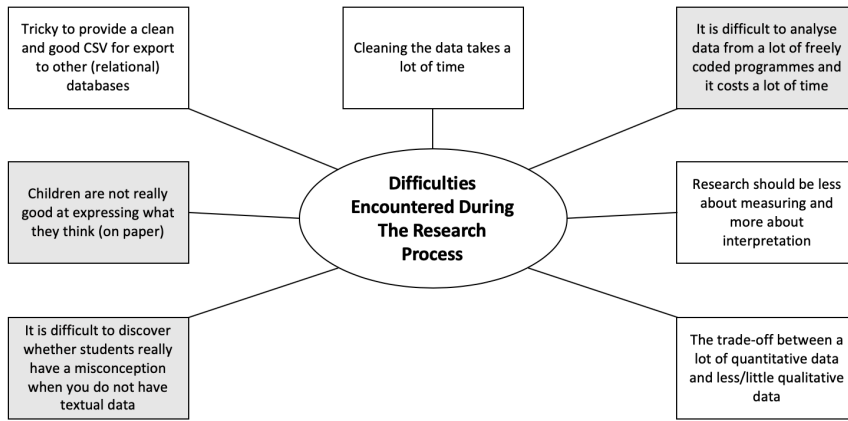


Figure 3.2: Difficulties encountered during the research process. The highlighted codes relate to the topics listed in the summary of this chapter

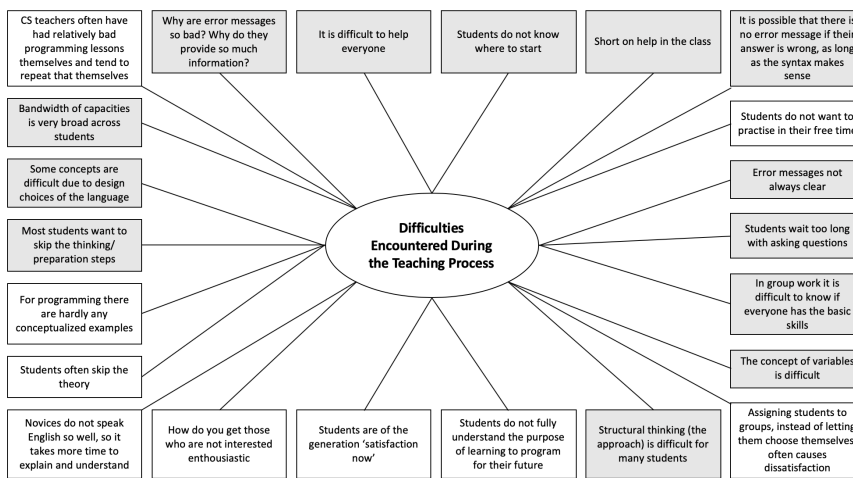


Figure 3.3: Difficulties encountered during the teaching process. The highlighted codes relate to the topics listed in the summary of this chapter

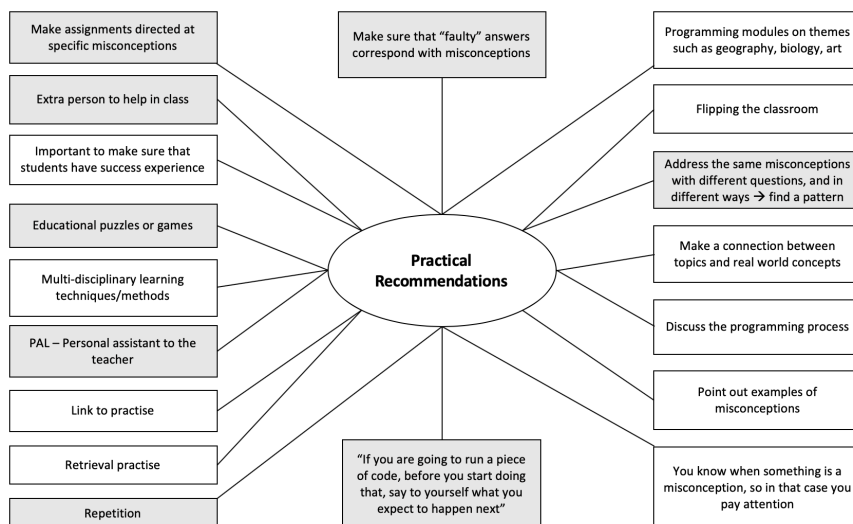


Figure 3.4: Practical recommendations. The highlighted codes relate to the topics listed in the summary of this chapter

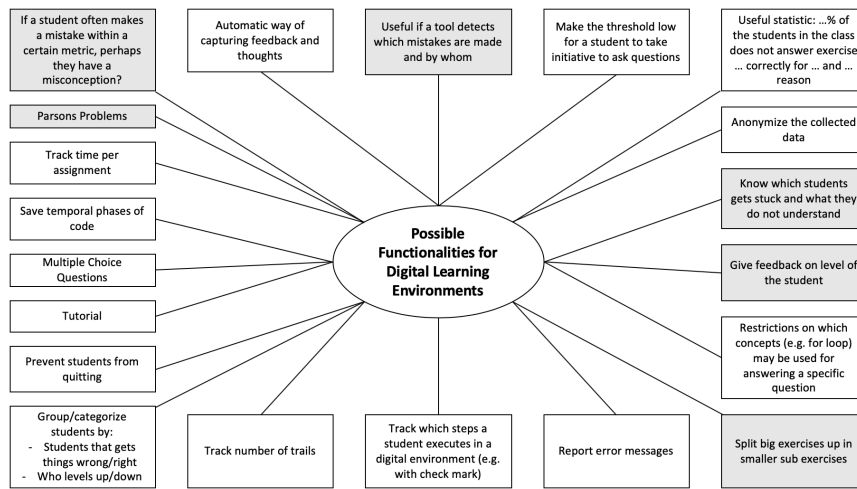


Figure 3.5: Possible functionalities for digital learning environments. The highlighted codes relate to the topics listed in the summary of this chapter

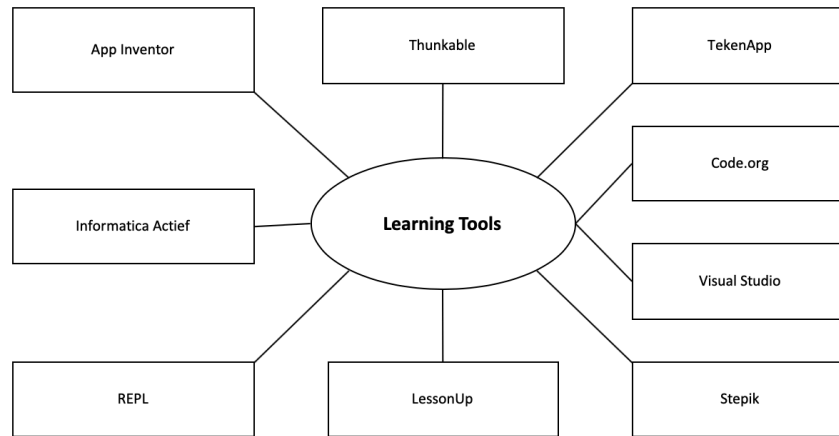


Figure 3.6: Learning tools. The highlighted codes relate to the topics listed in the summary of this chapter

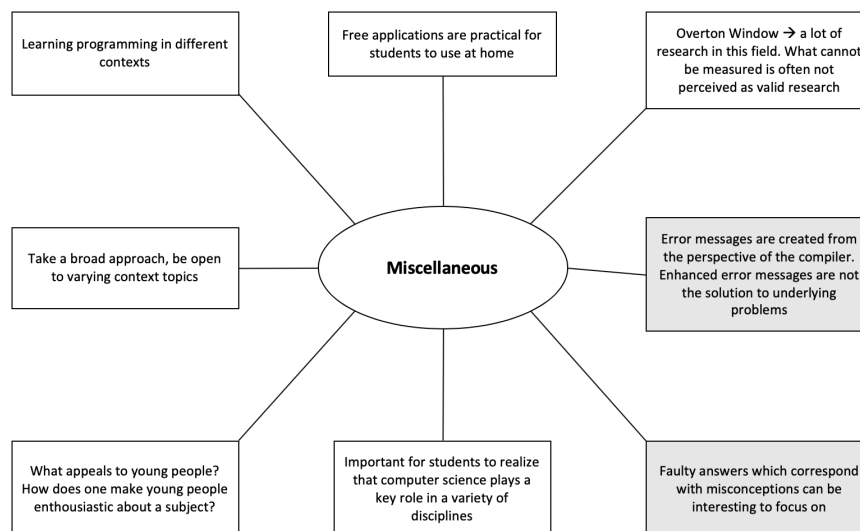


Figure 3.7: Miscellaneous. The highlighted codes relate to the topics listed in the summary of this chapter

Based on the interviews with the participants in this research, the research questions formulated in the introduction of Part I can be answered as follows.

RQ 1: Which difficulties do participants see with regard to the teaching process?

Teachers appeared to encounter a lot of challenges, which varied from annoyance about the difficulty of reading error messages, and the problem of cognitive differences between students, to the challenge of detecting which concepts an individual student does not grasp. Students appear to have difficulty understanding certain concepts and “most students want to skip the first steps of [logical thinking]”. According to teachers, useful tools should, amongst others, offer the functionalities of categorizing students based on their shown skill level and on whether or not students “answered correctly and [...] spend for example different time ranges on a specific question”. Furthermore, it was deemed useful to track the “number of trails” and to determine whether repeated mistakes could indicate misconceptions.

Programming education researchers mentioned the role of misconceptions relatively often. Researchers wanted to know “how one can train [to avoid misconceptions]” and “which misconceptions students really have”. Furthermore, the difficulties concerning error messages, particularly in Python, were mentioned. Researchers emphasized that it is difficult and extremely time-consuming to analyze data from an abundance of freely coded exercises, and that “there should be less measuring and more interpretation [of data]”.

RQ 2: What avenues for improvement or further research do the participants see?

Recommendations from the interviewees included “repetition”, “retrieval practise”, “let ‘faulty’ answers correspond with misconceptions”, “investigate the same misconception [...] but in different questions” in order to find a pattern and to present the problems in different ways, e.g. through tutorials, multiple choice questions or Parsons Problems. Parsons Problems were thought to be of use because students can make mistakes in a limited number of ways only, so participants thought it would be interesting to investigate whether those mistakes could be categorized in relation to misconceptions.

RQ 3: How could digitization or automation contribute to improving programming education?

Teachers do not have the time to analyze all freely coded submissions or code snapshots. Teachers would benefit from a tool which could alert them to which mistakes are being made by whom.

3.3. Summary: working towards a solution

Based on the input during the interviews, ten important topics were deduced. Codes related to these topics have been highlighted in the above diagrams.

With regard to teacher feedback, it appeared that teachers would find it very helpful to have more insight in the misconceptions of their students. As alternative testing methods, such as puzzles or Parsons Problems, and the need for recognizing misconceptions, were mentioned several times, the plan was conceived to explore the idea of using Parsons Problems for the detection of misconceptions.

In order to determine whether a method using Parsons Problems could be effective for such a purpose, the following section aims to determine whether a method involving Parsons Problems could be useful in addressing the ten main topics distilled from the interviews. This will be discussed below.

1. Students have difficulty learning programming

Parsons Problems take traditional programming exercises to a higher level of abstraction, reducing the cognitive load for students. By breaking problems up into smaller pieces in this way, working memory will be freed up, making it easier for novices to learn

2. Students have difficulty thinking in a structured way about how to approach a problem

Due to the high level of abstraction, Parsons Problems puzzles stimulate students to think about the general structure and logic, instead of all the details, when placing the code puzzle pieces in the correct order

3. **Students might benefit from alternative testing methods**
Parsons Problems constitute puzzles and would be suitable for gamified solutions
4. **Students have a lot of difficulty reading and understanding error messages**
Depending on the technique used for the digitization, Parsons Problems can give simple feedback such as which lines are in the wrong position or wrongly indented. The shape and content of this feedback is less complex than in traditional coding exercises
5. **Repetition is of great importance in programming education**
Although this recommendation was meant in the sense that students learn better when they have to repeat complicated exercises and topics, repetition will be an important factor in misconception detection as well. Misconceptions will have to be integrated multiple times in different questions during a testing session. In this case, the repetition of the exercise will provide increased certainty about the gravity of the misconception, as well as repeated exposure of topics and exercises to the students
6. **Teachers have difficulty determining what exactly their students have trouble with**
If PP could successfully detect misconceptions that students hold, this would greatly help teachers understand the difficulties that their students have
7. **Teachers would benefit from a tool which could alert them to which mistakes are being made by whom**
If Parsons Problems successfully detect misconceptions, this would be valuable input for a teacher feedback tool. In that case the analyzed results of the PP could provide insights on which students make which mistakes
8. **Students either do not notice when they do not understand an important concept, or they wait too long before asking for assistance**
If teachers would have access to an automated tool which successfully employs Parsons Problems for misconception detection, the resulting feedback would bring the students' misconceptions to the attention of the teacher which would enable the teacher to intervene
9. **Teachers do not have the time to analyze all freely coded submissions or code snapshots**
An automated tool would generate immediate and focused feedback, which would not require time-consuming additional analysis by the teacher
10. **Teachers have difficulty helping all students individually, in order to do that they would need extra help**
The described tool could inform the teacher about the number of students who hold a certain misconception. Teachers would then be able to give additional instructions directed at certain concepts for the whole class, or part of the class. Additionally, it would be possible to integrate and provide digital tutorials or extra explanations to students who seem to hold certain misconceptions, so that they can continue individually without necessarily needing the teacher. As a consequence, individual help to students will most likely not be needed as much as before

With regard to the above mentioned topics, it appears that the use of Parsons Problems could indeed be very interesting to address the main topics covering the remarks brought forward by the participants in the interviews. Therefore, it was decided to further explore the validity of using Parsons Problems for misconception detection.

Part II

Since the purpose of this thesis was to improve on existing programming education through the development of a feedback method for computer science teachers, the first part of the research was focused on conducting interviews with secondary school CS teachers and programming education researchers to uncover the strengths and weaknesses of currently applied methods. The results of Part I indicated that a method, which would provide specific and non-time-consuming teacher feedback, particularly about common mistakes and misconceptions, would contribute to improving the effectiveness of programming education. Analysis of the findings in Part I also indicated that Parsons Problems could be an interesting alternative method to test for those misconceptions. Therefore, the purpose of Part II of this research is to design a method for misconception detection through Parsons Problems, which generates specific and ready-to-use teacher feedback.

In order for such a method to generate clear and specific feedback, the data to be collected needs to fit certain requirements. First of all, the choice was made to limit the variety of possible mistakes made by students by focusing on a set of pre-defined misconceptions. As mentioned in the above and in line with suggestions in previous research about the use of survey or quiz questions in order to collect additional data [31], this research will test the use of Parsons Problems for detecting specific misconceptions. Moreover, it has been suggested that Parsons Problems can be used as an effective learning tool, particularly for novices. Parsons Problems offer specific design options, as discussed in the Parsons Problems chapter, which enhance their suitability for this particular purpose. As it is important to identify misconceptions in the early stages of programming education [29], it was decided to focus on misconceptions which are likely to occur for first or second year secondary school students, particularly for novices. As mentioned before, Python is the programming language of choice for this research, because it is often used in practise and in secondary school education and has so far been under-researched [33].

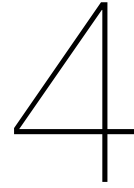
Research Questions

As Parsons Problems have so far not been researched or used in the context of misconception detection and feedback generation, the research questions for Part II of this thesis are:

1. Are Parsons Problems suitable for the detection of programming misconceptions?
2. What kind of insightful teacher feedback can be generated from the data collected through this method?

Approach

Building on the information gathered during this research, a set of Parsons Problems was designed for the purpose of detecting a select number of specific misconceptions. These exercises were tested during a pilot experiment followed by a more extensive experiment, of which the findings will be discussed in the results section of Part II.



Pilot Experiment

Before designing and executing an extensive experiment using Parsons Problems coding puzzles, a pilot experiment was run. The main goal of this pilot experiment was to get a rough idea whether Parsons Problems would indeed work for the purpose of identifying programming misconceptions. Moreover, it was important to learn which questions, setup and functions would work well for such an experiment, or which aspects could be improved upon before executing the more extensive experiment round.

In this chapter, the development of the first set of Parsons Problems for the pilot experiment will be described, followed by an analysis of lessons learned, which were integrated in the design of the more extensive follow-up experiment.

4.1. Question Design

It was decided to make the questions for the pilot experiment two-dimensional, since indentation is very important for Python. Furthermore, the choice was made to include a few distractors and to provide a bit of context. The questions in the pilot experiment served as a try-out in order to find out what does and what does not work.

Furthermore, the literature review already showed that particularly those existing methods which made use of previously compiled lists of most common errors (Retina, Karam et al. and Espresso) appeared to generate insightful feedback which did not require extensive further analysis. Therefore, it was decided to design the Parsons Problems in such a way that they would be able to detect errors related to misconceptions from the pre-defined list of most occurring misconceptions (see Literature Review, section Programming Misconceptions).

After selecting a number of misconceptions and after reviewing the potential Parsons Problems design options, ten Parsons Problems were created. In the following section, these ten questions and their solutions will be presented, including the misconception(s) they are targeting. Some questions contain distractors, which in those cases will be shown below the question. Some questions contain pieces of code which have already been placed in the right position, in order to provide context. These pieces of code will be highlighted in italics. For each of the questions an explanation will be given as to which misconception(s) might be detected if a student answers the question incorrectly.

4.2. Questions

The questions can be divided into four categories based on the main programming concepts selected for this research: Variables, If/Else Statements, While Loops and For Loops. In the following section, each of these categories and corresponding questions will be discussed.

4.2.1. Variables

All three PP which are related to variables attempt to uncover whether the students understand variables, assignment (directions) and whether they understand which, and how many, values are stored in which variables.

Question 1

Put the pieces of code in the right order, so that the variable `b` has the value 5.
! Beware ! Not all the pieces of code have to be used.

`a = 5`

`b = 10`

`c = a`

`b = c`

Distractors:

- `c = b`

Misconceptions targeted:

- Primitive assignment works in opposite direction
- Primitive assignment works both directions (swaps)

Question 2

Put the pieces of code in the right order, so that variable `b` has the value 23 and so that the number 25 is printed.

`a = 23`

`b = a`

`a = 25`

`print(a)`

Distractors: None.

Misconceptions targeted:

- Difficulties in understanding the sequentiality of statements
- A variable can hold multiple values at a time / 'remembers' old value

Question 3

Put the pieces of code in the right order, so that the value of the variable `x` and the value of the variable `y` are switched. So the variable `x` should have value 5 and variable `y` should have value 3.

! Beware ! Not all the pieces of code have to be used.

`x = 3`

`y = 5`

`temp = 0`

`temp = x`

`x = y`

`y = temp`

Distractors:

- `y = x`

Misconceptions targeted:

- Primitive assignment works in opposite direction
- Primitive assignment works both directions (swaps)

4.2.2. If/Else Statements

The following questions are mainly constructed to test whether students understand the general concept of if/else statements.

Question 1

Put the pieces of code in the right order so that it is printed whether x is a positive or a negative number.

! Beware ! Not all the pieces of code have to be used.

```
if x >= 0:
    print('x is a positive number')
else:
    print('x is not a positive number')
```

Distractors:

- if:
- else x < 0:

Misconceptions targeted:

- Invalid else-statement
- Using else is optional

Question 2

Put the pieces of code in the right order so that the number 10 is printed if variable a is smaller than 5, otherwise the number 20 should be printed.

```
if a < 5:
    a = 10
else:
    a = 20
```

`print(a)`

Distractors: None.

Misconceptions targeted:

- General understanding of if/else statement

Question 3

Put the pieces of code in the right order so that the size of the variables a and b are compared.

```
a = 180
b = 35
if b > a:
    print('b is larger than a')
else:
    print('b is not larger than a')
```

Distractors: None.

Misconceptions targeted:

- General understanding of if/else statement

4.2.3. While Loops

Through these two while loop PP, the goal is to find out whether students properly understand the concept of a while loop and the sequentiality of statements. In both questions, if lines 3 and 4 are reversed, it might indicate the sequentiality of statements misconception.

Question 1

Put the pieces of code in the right order so that the following number sequence is printed: 1 2 3 4 5 6 7 8 9 10

```

i = 0
while i < 10:
    i = i + 1
    print(i)

```

Distractors: None.

Misconceptions targeted:

- Difficulties in understanding the sequentiality of statements

Question 2

Put the pieces of code in the right order so that the following number sequence is printed: 4 6 8

```

i = 4
while i < 9:
    print(i)
    i = i + 2

```

Distractors: None.

Misconceptions targeted:

- Difficulties in understanding the sequentiality of statements

4.2.4. For Loops

The PP in this section aim to find out whether students understand how to construct two different loops within each other, which code is part of the body of a loop, which code executes outside a loop, and whether the students understands the sequentiality of statements.

Question 1

Put the pieces of code in the right order, so that variable p increases with 1 if p is smaller than 3 and otherwise it should be printed that p is not smaller than 3. Also, every value of p should be printed.

for p in range(7):

```

if p < 3:
    p = p + 1
else:

```

```
print('p is not smaller than 3')
```

```
print(p)
```

Distractors: None.

Misconceptions targeted:

- Adjacent code executes within loop

Question 2

Put the pieces of code in the right order so that the following number sequence is printed: 0 1 3 6

```
x = 0
```

```
for z in range(4):
```

```
    x = x + z
```

```
    print(x)
```

Distractors: None.

Misconceptions targeted:

- Difficulties in understanding the sequentiality of statements
- Difficulty in understanding automated changes to for loop control variables

4.3. Setup and Participants Pilot Experiment

The pilot experiment took place during a coding workshop evening at Metis, a secondary school in Amsterdam. Students from all ages in secondary school could sign up for this night. The pilot experiment was not a workshop in itself, but several students from different workshops volunteered to participate. The experiment was performed with one student at a time.

The exercises had been printed on a piece of paper. The puzzle pieces of code had also been printed and cut into pieces. First, the concept of Parsons Problems was explained to the student. The student had to put all the puzzle pieces in the right order. The student would show the result after completing the exercise, after which the answer produced by the student would be noted.

A total of five students started with the experiment. One of them stopped after trying one exercise. In the end four students fully participated in a session. They were second and third year secondary school students, aged 13 and 14.

4.4. First Test Results

The Parsons Problems were divided into four categories, as described in the Questions section. The four students who participated received a question category based on their prior knowledge of Python programming. After finishing one category, students could choose to either continue and complete another question category or to stop participating in the experiment. This approach was chosen since the experiment was on a voluntary basis and it was a 'side experiment' in the context of the workshop evening.

Due to this approach, not all four categories received the same number of answers. Three students answered the Variables questions, three answered the While Loops questions, one answered the For Loops questions and one answered the If/Else Statements questions. The level of the four participating students varied slightly since the group was composed of both second and third year students from different difficulty levels of secondary school. Therefore, some questions might have been too easy for some of these students. However, some interesting results were gained from this pilot experiment, which indicated that the use of Parsons Problems might hold promise for the detection of programming misconceptions.

Several misconceptions surfaced in answers to questions that targeted those misconceptions. Some of the mistakes made by these four students indicated misconceptions which were not anticipated to surface if a student would answer a question incorrectly. Some faulty answers were the result of misconceptions which had not been expected based on the literature researched for this thesis. Furthermore, students made other mistakes, which were not expected to be found. Each of the students and the mistakes they made will be discussed hereafter.

4.4.1. Participant 1

Participant 1 only completed the questions concerning Variables. The answers for questions 2 and 3 showed similar mistakes. In question 2, the lines `'a = 23'` and `'b = a'` were not correctly indented, while the order of the lines was correct. In question 3 the lines `'temp = x'` and `'y = temp'` were incorrectly indented, while also the order of `'x = y'` and `'temp = x'` proved to be incorrect. These mistakes together seem to imply that this student does not fully understand the implication of variable value assignment and the concept of assigning the value of one variable to another variable.

4.4.2. Participant 2

Participant 2 answered all questions related to Variables correctly. However, when answering the While Loop related questions, a mistake occurred in question 2. The lines `'i = i + 2'` and `'print(i)'` were put in the wrong order. This indicates that this student might hold a misconception concerning the sequentiality of code. Perhaps the student does not understand that each line of code has an effect on the environment in which the following line of code will be executed. Maybe this student thought that all lines would be executed at the same time or that the computer would know what to do in terms of which line needs to be executed when, in order to have a certain result. This was the purpose of the exercises with while loops. The two questions are very similar, except for the fact that the line changing the value of `i` and the print line should be in a different order.

4.4.3. Participant 3

Participant 3 answered all questions related to Variables and While Loops correctly. However, this student did make some interesting mistakes when answering question 2 of the For Loop questions. Firstly, the lines `'print(x)'` and `'x = x + z'` were put in the wrong order. This implies that the student might not have understood sequentiality as well as it seemed during the other questions concerning variables and while loops. It might also have to do with a misunderstanding concerning the concept of the control variable `z` and its mechanisms. Secondly, the line `'x = x + z'` was positioned upside down, namely `'z + x = x'`. This mistake was only possible because this exercise was performed with pieces of paper instead of on a computer. However, it may actually highlight a misunderstanding of the variable assignment direction. Apparently this student believed that assignment from left to right is possible. An explanation as to why this mistake was made here and not with the other questions containing a similar piece of code, may be that the actual letters used to represent variables (`x` and `z`) look the same when put upside down. If for example the line `'i = i + 2'` would be put upside down, one would more quickly determine that this cannot be right because the letters and numbers do not make sense upside down. This is an interesting discovery, which may be useful when designing future questions, since it appears that using this mechanism shows a student's actual level of understanding concerning assignment direction.

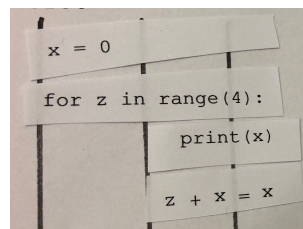


Figure 4.1: Answer to question 2 about For Loops by Participant 3

4.4.4. Participant 4

Participant 4 only made one mistake when answering the questions concerning If/Else Statements. This question included some distractors focused on the syntax of the if and else conditions. The 'else x < 0:' line was used instead of 'else:', which could indicate that this student does not fully understand the syntax concerning if statements. Namely, that an else statement does not contain a condition, only in the case of an elif statement, there would be a condition. Since this type of distractor was only used in this question and not in the other two If/Else Statement questions, it is difficult to determine whether this was a one-off or whether the student really holds this misconception. Based on the answers to the While Loop questions, it can be concluded that this student did not fully grasp how a variable functions in a while loop. In both questions the variable i is declared within the while loop, which is incorrect. For example, for question 1, the lines 'i = 0' and 'while i < 10:' were switched. The same mistake was made with the answer to question 2. The fact that this very specific mistake is repeated in both answers, shows that it is very likely that this student does not understand that the variable used in a while loop expression, which is altered within the loop, should be initialized before the while loop. Also, for question 2, the lines 'i = i + 2' and 'print(i)' were switched, which may be a sign of the sequentiality misconception.

4.5. Conclusions Pilot Experiment

Several things became clear as a result of this pilot experiment, both in relation to the execution of the experiment itself and with regard to the direction in which this research needed to be continued.

Concerning the Parsons Problems used in the pilot experiment the following became clear. Some questions turned out to be more useful than others and some question descriptions could be formulated more precisely. For example, the first two questions seemed too easy and would therefore not generate interesting results. The development of Parsons Problems related to variables could thus use some more focus. Question 1 of the For Loop Parsons Problems could have been formulated more explicitly with regard to the printing of every value of p. This question was answered correctly, but upon further consideration it seemed that this part might be interpreted differently than intended and could lead to an alternative, though technically correct, answer. A better way to put it then would seem to be: the value of p should be printed within every loop, no matter what the scenario turns out to be. However, in that case, the 'print(p)' could also be 'correctly' positioned before the if statement. Therefore it would be better to reconsider the setup or description of this specific question altogether. The while loops seemed to show some possibly useful insights regarding misconceptions. Especially the repetitiveness and similarity between the questions helps to determine whether a misconception is simply a one-off, or whether the misconception occurs more often and should be addressed. The mistake made by one of the participants regarding the syntax of an else statement also asks for more repetition and similarity between questions. In order to determine whether a mistake is accidental, or occasional, or whether it is a consistent misconception, more repetition should be applied and more questions should be created per category or across categories that show similar insights.

In general, it is not always easy to translate misconceptions into Parsons Problems which will detect these misconceptions. Therefore, it was decided to include more distractors for certain questions.

Although this pilot experiment did not include many questions per category and the participants were not asked all of the questions, the results do already show some promise. As discussed in the results section above, the pilot experiment did uncover several recurring mistakes, which most likely indicates misconceptions. Some mistakes seem to indicate that a student holds a certain misconception, but, in order for the analysis to be more precise, more questions and answers per individual student would be necessary. However, this pilot experiment has shown, that if a student holds these misconceptions, they do come to light if students are tested with questions translated into Parsons Problems. This means that so far, these preliminary test results show promise regarding the use of Parsons Problems in the detection of programming

misconceptions.

The next part of this research consists of constructing more and better Parsons Problems which will be used in the more extensive follow-up experiment.

5

Experiment

The results of the pilot experiment indicated that PP could indeed be useful for misconception detection in novices. In this chapter, a much more extensive experiment will be described, consisting of more PP per student. Moreover, many more participants were recruited and tested than in the pilot experiment.

5.1. Question Design

Concerning the question design, several lessons were learned from the pilot experiment.

- **(Non-paired) distractors:** distractors are very useful for embodying misconceptions and are therefore very useful for the purpose of this research. However, in order to minimize the answer space and preserve the clarity of the results, it was decided not to use paired distractors for all code pieces of the solutions. Questions will not contain distractors if it is thought that distractors will not add enough value in the detection of targeted misconceptions
- **Two-dimensional:** since indentation is of great significance in Python, it is very important to use two-dimensionality in these PP to gain an as complete as possible insight into a student's understanding of constructs and concepts
- **Context:** context is another method for limiting the answer space and emphasizing specific misconceptions. This concept is therefore used on a few occasions
- **Multiple lines:** this concept is especially useful for limiting the answer space and emphasizing misconceptions regarding variable value assignment. This method is only used on one occasion

Partially completed code pieces were not included for several reasons. Although partially completed code pieces provide an opportunity for free and differing input per student, doing this would unnecessarily complicate additional analyses, but more importantly, using this method requires a different kind of thinking and analysis level from students. However, as this experiment was designed for students at novice level, it was decided to not unnecessarily complicate the questions in order for students to be able to focus on the more important basics.

One of the important things learned from the pilot experiment is that enough questions need to be created that target the same misconceptions. In case students holds a misconception, they will have the opportunity to make the same or a similar mistake more often, which will give more certainty about whether a mistake is a misconception or just a random mistake or a one-off.

In order to be able to draw conclusions based on the experiment to be performed, the following selection of misconceptions [55] [39] was made:

1. Primitive assignment works in opposite direction
2. Primitive assignment works both directions (swaps)
3. Invalid `else`-statement
4. Using `else` is optional
5. The natural-language semantics of variable names affects which value gets assigned to which variable
6. Difficulties in understanding the sequentiality of statements
7. Adjacent code executes within loop
8. For loop control variables do not have values inside the loop or their values can be arbitrarily changed

5.2. Questions

This experiment consists of fifteen Parsons Problems, which have been divided over the same four categories as the questions which were designed for the pilot experiment: Variables, If/Else Statements, While Loops and For Loops. Table 5.1 shows which PP questions target which misconceptions from the above mentioned list of eight targeted misconceptions.

Question	Category	Targeted Misconception
1	Variables	1, 2
2	Variables	1, 2
3	Variables	1, 2
4	Variables	1, 2
5	If/Else Statements	3, 4
6	If/Else Statements	3, 4
7	If/Else Statements	3, 4, 5
8	While Loops	6
9	While Loops	6
10	While Loops	6
11	While Loops	3, 7
12	While Loops	3, 7
13	For Loops	6, 8
14	For Loops	3, 7
15	For Loops	No specific misconception targeted

Table 5.1: Targeted misconceptions per experiment question. Misconception numbers correlate with targeted misconception list

All fifteen PP can be found in Appendix A. The four different categories and a few question examples will be explained in more detail in the following sections.

5.2.1. Variables

The questions related to variables have been designed to detect whether students have misconceptions concerning the direction in which values should be assigned to variables. These four questions can be found in Section A.1 in Appendix A. The second and third question were inspired by a questionnaire developed by Bornat and Dehnadi [10], which focused on misconceptions concerning variables in the context of multiple choice questions.

An example of one of the PP belonging to this category is Question 3 below. This question was designed by using two lines of context (indicated in italics), namely '*j = 5*' and '*k = 10*', which are already placed in the correct position. Furthermore, this Parsons Problem consists of four puzzle pieces, two of which are distractors. The goal of this question is to detect whether a student shows signs of a misconception concerning the direction in which values should be assignment

to variables. The actual format of this question, as it was presented in the experiment, can be found in Figure 5.1.

Question 3

Put the pieces of code in the right order, so that variable *k* has the value 5.
! Beware ! Not all the pieces of code have to be used.

```
j = 5
k = 10
```

m = j

k = m

Distractors:

- j = m
- m = k

Misconceptions targeted:

- Primitive assignment works in opposite direction
- Primitive assignment works both directions (swaps)

Vraag 3 - Variabelen ✕ ⋮

Zet de code stukjes in de goede volgorde zodat variabele *k* de waarde 5 heeft.
! Let op ! Je hoeft niet alle stukjes te gebruiken.

1. *k* = *m*
2. *m* = *j*
3. *m* = *k*
4. *j* = *m*

Deze dik gedrukte code staat al op de goede plaats.

```
j = 5
k = 10
```

a b c

Zet hier de nummers van de stukjes code die je wilt gebruiken op de goede volgorde. Vergeet *
niet achter elk stukje te zetten op welk lijntje het hoort te staan (a, b of c).

Long-answer text

Figure 5.1: Question 3 in official experiment

5.2.2. If/Else Statements

The Parsons Problems related to *if/else* statement mainly focus on whether students will create valid *else* statements. These questions can be found in Section A.2 in Appendix A.

5.2.3. While Loops

The Parsons Problems related to *while* loops aim to detect whether students have difficulties understanding the sequentiality of statements and/or whether students understand that code

located adjacent to a loop or if statement does not execute within the body of that loop or if statement. These five questions can be found in Section A.3 in Appendix A.

An example of one of the `while` loop questions is illustrated below. This question does not contain any distractors or context, it simply makes use of two-dimensionality. The question shown below, targets the sequentiality of statements misconception. The way in which students answer this question should indicate whether they correctly understand that within the body of the `while` loop, first '`i = i + 1`' should be executed, before the value of `i` is printed, in order to obtain the correct number sequence. The actual format of this question, as it was presented in the experiment, can be found in Figure 5.2.

Question 8

Put the pieces of code in the right order so that the following number sequence is printed: 1 2 3 4 5 6 7 8 9 10

```

i = 0
while i < 10:
    i = i + 1
    print(i)

```

Distractors: None.

Misconceptions targeted:

- Difficulties in understanding the sequentiality of statements

Vraag 8 - While Loops

Zet de code stukjes op de goede volgorde zodat de volgende getallen reeks geprint wordt: 1 2 3 4 5 6 7 8 9 10

1. `print(i)`
2. `i = 0`
3. `i = i + 1`
4. `while i < 10:`

Image title

a b c

Zet hier de nummers van de stukjes code op de goede volgorde. Vergeet niet achter elk stukje te zetten op welk lijntje het hoort te staan (a, b of c). *

Long-answer text

Figure 5.2: Question 8 in official experiment

5.2.4. For Loops

The Parsons Problems related to for loops simply aim to test whether students understand the concept of for loops. The three questions dedicated to this concept can be found in Section A.4 in Appendix A.

6

Results Part II

The extensive experiment, which followed the pilot experiment, took place in January 2020 in Rotterdam during several days.

6.1. Setup and Participants

An initial total of 65 students from the first and second year classes of a secondary school in Rotterdam partook in this experiment. One student did not submit answers, so a final number of 64 students was counted as participants in the experiment, the majority ranging from 12 to 14 years old. The students were from varying educational levels, namely Havo and VWO. The experiment was conducted in classrooms where each individual student completed the test behind a computer. The students were allotted one hour to complete the experiment.

First, the concept of PP was explained in short, as well as the setup of the questions. The fifteen PP consisted of a brief question description and the puzzle pieces. Moreover, a partial impression of the answer sheet consisting of three lines which represented indentation options was shown (labeled a, b and c) as well as the fixed puzzle pieces (if applicable). The entire experiment, as presented to the students, can be found in Appendix B.

Due to time limitations, the experiment was not created in a web-based puzzle-like environment, but as a Google Forms document. Google Forms was deemed a good alternative, as the main focus of this experiment was to create the correct type of question in order to find out whether PP could be useful for misconception detection.

6.2. Findings

The raw data collected during this experiment can be found in Appendix C. All answers were checked on correctness. Table 6.1 below shows how many students answered the fifteen individual PP wrongly or correctly. In case an answer was correct, no further analysis was conducted. For all the incorrect answers, the corresponding code was composed and analysed in order to establish whether or not the mistake was the result of a misconception.

	Parsons Problems														
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Wrong (% of students)	76.6	56.3	81.3	82.8	79.7	82.8	93.8	85.9	92.2	82.8	98.4	95.3	93.8	96.9	82.8
Wrong (# of students)	49	36	52	53	51	53	60	55	59	53	63	61	60	62	53
Correct (% of students)	23.4	43.8	18.8	17.2	20.3	17.2	6.3	14.1	7.8	17.2	1.6	4.7	6.3	3.1	17.2
Correct (# of students)	15	28	12	11	13	11	4	9	5	11	1	3	4	2	11

Table 6.1: Number and percentage of students that answered the Parsons Problems in the experiment correctly and wrongly

After the analysis, one misconception, i.e. 'for loop control variables do not have values inside the loop or their values can be arbitrarily changed', was discarded. No useful results were

generated from the answers to the related question (question 13), which may be an indication that this particular misconception is not suitable for detection through Parsons Problems.

Although each question in the experiment had been tailored for the detection of a specific misconception, data analysis indicated that other misconceptions occurred in the answers as well. Therefore, all misconceptions found during the experiment have been divided in four categories of misconceptions:

- **Targeted Misconceptions:** misconceptions that were specifically targeted by the questions in the experiment
- **Targeted Misconceptions occurring in Non-Targeted Questions:** misconceptions that were part of the set of targeted misconceptions, but that occurred in questions in which they were not targeted
- **Other Misconceptions:** misconceptions which were not specifically targeted by the questions in which they were encountered, but which have been listed in previous research (see chapter Misconceptions)
- **New Misconceptions:** mistakes that were made repeatedly and which could be defined as misconceptions, but have so far not been found in literature

As the ultimate goal of this experiment is to generate insightful teacher feedback, the focus of this analysis is on:

- Providing insight for the teacher about which students do not understand which programming concepts
- Uncovering which mistakes occur most often across students

6.2.1. Targeted Misconceptions

As mentioned before, each of the Parsons Problems was designed with the purpose of targeting a specific misconception. Therefore, the analysis started by establishing whether the misconceptions occurred for questions in which they were actually targeted. The first analysis was made per student. For each student, each wrong answer was evaluated, in order to establish whether or not the mistake did indeed occur as a result of the targeted misconception. Mistakes which were not the result of the targeted misconception have been left out of this analysis, but will be discussed in the following analysis sections. Table 6.5 shows how many times each student showed a misconception when answering a question in which that particular misconception was targeted. Such a mistake was indicated with an 'x' for each time the same misconception occurred in a different question. Therefore, if four x-es are displayed for a particular student regarding a certain misconception, this means that this student showed to hold the same misconception in four different questions. In Table 6.7, further information is provided concerning the specific questions in which the detected misconceptions occurred, indicated by the numbers assigned to those questions (see Appendix A).

As an example, Table 6.5 indicates that participant 46 showed the misconception 'primitive assignment works in opposite direction' in four different questions. Table 6.7 further elaborates on this by showing that the four occurrences of this misconception arose in all four of the Variable questions. Figures 6.1 and 6.2 illustrate the answers given by participant 46 to these four Variable questions. The fact that this participant repeated the same mistake four times in four questions, is a strong indication that the student does indeed hold the targeted misconception.

In Table 6.2, the misconceptions targeted in the experiment are ranked based on the number of students that appear to hold a certain misconceptions (in descending order from most to least students). Some misconceptions have been tested through several questions and therefore allow repeated occurrence of a faulty answer linked to that misconception, but for this analysis, students were deemed to hold a certain misconception if that misconception occurred at least once. As this table exemplifies, most students appeared to encounter difficulties with primitive assignment direction and `else`-statements, followed by the sequentiality of statements.

<code>x = 3</code>	<code>x = 3</code>	<code>j = 5</code>	<code>j = 5</code>
<code>y = 5</code>	<code>y = 5</code>	<code>k = 3</code>	<code>k = 3</code>
<code>temp = 0</code>	<code>temp = 0</code>	<code>m = 7</code>	<code>m = 7</code>
<code>temp = x</code>	<code>y = temp</code>	<code>j = m</code>	<code>m = j</code>
<code>x = y</code>	<code>x = y</code>	<code>k = j</code>	<code>j = k</code>
<code>y = temp</code>	<code>temp = x</code>		

Figure 6.1: On the left: Question 1 correct answer and Question 1 participant wrong answer. On the right: Question 2 correct answer and Question 2 participant wrong answer

<code>j = 5</code>	<code>j = 5</code>	<code>j = 23</code>	<code>j = 23</code>
<code>k = 10</code>	<code>k = 10</code>	<code>k = j</code>	<code>j = k</code>
<code>m = j</code>	<code>j = m</code>	<code>j = 25</code>	<code>j = 25</code>
<code>k = m</code>	<code>m = k</code>	<code>print(j)</code>	<code>print(j)</code>

Figure 6.2: On the left: Question 3 correct answer and Question 3 participant wrong answer. On the right: Question 4 correct answer and Question 4 participant wrong answer

Rank	Misconception	Students (#)	Students (%)
1	Primitive assignment works in opposite direction	36	56%
2	Invalid <code>else</code> -statement	34	53%
3	Primitive assignment works in both directions (swaps)	29	45%
4	Difficulties in understanding the sequentiality of statements	25	39%
5	The natural-language semantics of variable names affects which value gets assigned to which variable	13	20%
6	Adjacent code executes within loop	4	6%
7	Using <code>else</code> is optional	3	5%

Table 6.2: Number of students and percentage of participant base that hold a misconception

However, some misconceptions were formally targeted in more questions than others. For example, ‘primitive assignment works in opposite direction’ was tested in four questions, while ‘the natural-language semantics of variable names affect which value gets assigned to which variable’ was targeted in only one question. The reason for this is that certain questions did target a specific misconception, but turned out to address another misconception as well. For example the loop questions sometimes also incorporated an `if` statement. Therefore, the weighted average of misconceptions versus the number of times the misconception could have occurred across questions was calculated, resulting in a slightly different ranking, as can be seen in Table 6.3. Primitive assignment direction remains in first place, indicating that this is the most often occurring misconception when measured per student and when measured in absolute and weighted occurrence. Second and third, however, are the natural-language semantics of variable names and the sequentiality of statements.

Rank	Misconception	Students (#)	Occurrences (#)	Questions (#)	Relative Occurrences (#)
1	Primitive assignment works in opposite direction	36	56	4	14
2	The natural-language semantics of variable names affects which value gets assigned to which variable	13	13	1	13
3	Difficulties in understanding the sequentiality of statements	25	48	4	12
4	Invalid <code>else</code> -statement	34	67	6	11
5	Primitive assignment works in both directions (swaps)	29	36	4	9
6	Adjacent code executes within loop	4	6	3	2
7	Using <code>else</code> is optional	3	3	3	1

Table 6.3: Absolute and weighted number of misconceptions detected

The analysis for Table 6.2 and Table 6.3 provide valuable information for a teacher as these tables show which individual student or which percentage of a class does not grasp an important

concept. This could be a clear indication that intervention is needed, i.e. that a particular concept should be explained in more detail to an individual student or for the class as a whole.

6.2.2. Targeted Misconceptions occurring in Non-Targeting Questions

Two targeted misconceptions also occurred in answers to other questions than the ones that actually targeted those misconceptions, as shown in Table 6.6. Table 6.8 further elaborates on this by showing in which non-targeting questions these targeted misconceptions occurred.

Sequentiality of statements, which had been included in the set of misconceptions targeted in the experiment, also occurred for questions which did not specifically target this misconception. However, this can be explained by the fact that code is all about sequentiality.

'Adjacent code executes within loop' also occurs a number of times for questions that did not target this misconception, which can be explained by the fact that several questions contained loops.

The number additional students who showed the sequentiality of statements misconception and the adjacent code misconception (occurring in answers to questions which did not target those misconceptions) were added to the original numbers displayed in Table 6.2. The results of the new total numbers are listed in Table 6.4.

Rank	Misconception	Students (#)	Students (%)
1	Difficulties in understanding the sequentiality of statements	41	64%
2	Primitive assignment works in opposite direction	36	56%
3	Invalid <code>else</code> -statement	34	53%
4	Primitive assignment works in both directions (swaps)	29	45%
5	The natural-language semantics of variable names affects which value gets assigned to which variable	13	20%
6	Adjacent code executes within loop	5	8%
7	Using <code>else</code> is optional	3	5%

Table 6.4: Total number of students and percentage of participant base that hold a misconception targeted in this research

Most notably, sequentiality of code moves to first place, meaning that this appears to be the concept misunderstood by the largest number of students in the sample group. As sequentiality of code is one of the basic concepts of programming, it seems that it would be very important for teachers to know that their students encounter great difficulty with this concept.

6.2.3. Other Misconceptions

Apart from the misconceptions targeted in the questions designed for the experiment, two other known misconceptions (see Literature Review, section Programming Misconceptions) have been detected. 'A variable can hold multiple values at a time/'remembers' old value' and 'difficulties understanding the lifetime of values' emerge multiple times. As these two misconceptions address related concepts, and because mistakes could potentially be attributed to either of those concepts, it was decided to combine these two concepts for the purpose of this analysis. Table 6.6 exemplifies that these misconceptions occurred twelve times across 8 students (13% of total participants), and Table 6.8 shows for which questions the misconceptions occurred.

6.2.4. New Misconceptions

Further analysis of the data also brought to light several mistakes which were made repeatedly across multiple students. A selection was made of the two misconceptions that showed a significant number of occurrences.

As Table 6.6 shows, the following mistakes occurred exceptionally often, and therefore could potentially be labeled as new misconceptions:

- **Indentation of variable (re-)assignment:** students often seem to misunderstand how variable value (re-)assignment works. In this research, students seemed to believe that different values assigned to the same variable can exist in parallel, as long these value assignments are occurring at different indentation levels. This misconception occurred 68 times for 28 students (44% of total participants)
- **Indentation of loop condition or no indentation of loop content:** Students often mistakenly applied indentation to an if statement or a for/while loop condition, or they mistakenly did not apply indentation to the content of such statements/loops. This misconception occurred 74 times for 19 students (30% of total participants)

The original misconceptions list compiled by Tobias Kohn does refer to a mistake concerning indentation, described by him as 'invalid indentation'. However, Kohn labeled this type of mistake as a 'minor syntax error', which according to him could occur by accident when editing code. For example, when students remove a loop condition but leave the content indented [39].

However, the large number of occurrences found by the present research strongly indicate that these types of mistakes do constitute misconceptions, as defined by Sorva [55]. As the problem solving process of Parsons Problems poses strong limitations with regard to the free editing situation described by Kohn, it seems most likely that these mistakes were not made by accident but that they result from a deeper misunderstanding of the aforementioned concepts.

6.2.5. Research Questions

Based on the above analyses, the research questions formulated in the introduction of Part II can be answered as follows.

RQ 1: Are Parsons Problems suitable for the detection of programming misconceptions?

As the Parsons Problems used in the experiment of this research appear to have detected the targeted misconception, as well as non-targeted misconception, other misconceptions and new misconceptions, the answer to RQ 1 is yes.

RQ 2: What kind of insightful teacher feedback can be generated from the data collected through this method?

Several kinds of useful teacher feedback have been generated based on the data generated in the experiment:

- How many students (and which percentage of total participants) answered each question wrongly or correctly?
- How many students hold a certain misconception?
- Which percentage (and absolute number) of the student base holds which misconception?
- Which individual student holds which misconception
- To what extent does a student appear to have difficulty with a programming concept (how often does the student repeat a certain mistake)?
- In which question(s) did a certain misconception occur for individual students?
- Ranking of misconceptions on the number of students that showed the misconception
- Ranking of misconceptions on the weighted number of occurrences

Participant	Targeted Misconceptions						
	Primitive assignment works in opposite direction	Primitive assignment works both directions (swaps)	Invalid else-statement	Using else is optional	The natural-language semantics of variable names affects which value gets assigned to which variable	Difficulties in understanding the sequentiality of statements	Adjacent code executes within loop
1	XX	X	X			XX	
2	X	X					
3							
4	X	X					
5							
6	X						
7			XX		X	X	
8	X						
9			XXX		X	XXXX	
10	X	X	XX		X		
11	X	X	XX				
12		X					
13		X					
14	XX	X	X				
15	XX	X	X		X		
16							
17	X						
18	XX						
19	X	X					
20	X	X					
21		X					
22	X		X				
23	X		X				
24	X						
25	XX		X				
27			X			XXX	
28			X				
29			X			X	
30							
31							
32							
33		X					
34							
35	XXX		XX				
36	X		XXX			XX	
37	X	X	X			XX	
38			X			XX	
39	X		XX				
40	XX	XX	XXXX	X		XXXX	
41	XX	XX	XXXX	X		XXXX	
42							
43		X					
44		X					
45		X					
46	XXXX					X	
47	XX		XXX			X	
48						X	
49			XXX			X	X
50	XX	X	XXX		X		
51	XX	X				XX	
52	X	X	XXXX				
53	X	XXX	XX		X		
54	XX	X	XX		X	XXX	
55	X	X	X			X	XX
56		X	XX	X	X		
57		XX					
58	XX		X		X	X	
59	XXX		XXX		X	X	
60					X	X	X
61			XX		X	XX	
62	X	XXX	XX			XX	XX
63	XX		XX			XXXX	
64	X	X	XX		X	X	
65						X	

Table 6.5: Targeted misconceptions detected per student

Participant	Targeted Misconceptions					Adjacent code executes within loop	Other	New	
	Primitive assignment works in opposite direction	Primitive assignment works both directions (swaps)	Invalid else-statement	Using else is optional	The natural language semantics of variable names affects which value gets assigned to which variable			Difficulties in understanding the sequentiality of statements	A variable can hold multiple values at a time / remembers' old value & Difficulties understanding the lifetime of values
1					X			XXX	XXXX
2									
3									
4					X			X	
5								X	
6							X	X	
7					XX			XXX	XX
8					X			XX	
9					XXX		X		XXXXXXXX
10							X		
11					X				X
12					X				
13					X				
14									X
15					X			X	
16									
17					X			X	
18					X		X	XX	
19					X				
20					XX			X	
21					XX			X	
22					X			XX	
23					X			XX	
24					X			XX	X
25					X			X	XXX
27									
28									
29									
30									
31									
32									
33									
34									
35								XXX	XX
36								XX	XX
37					XX				
38									XXX
39									XXXX
40					XX			XXXXX	XXXX
41					XX			XXXXX	XXXX
42									
43									
44									
45					X		X		
46									
47									
48									
49									
50									
51					X		XX		
52								XXXXX	XXXXXXXX
53								XXXX	XXXXXX
54					X			XXXXXXXX	XXXXXXXX
55								XXXX	
56								XX	XXXXXXXXXX
57								XX	
58								XX	XX
59							XXXX		
60						X			
61									
62									
63					X	XXX		X	XXX
64					XX			XXX	
65					X		X		

Table 6.6: Other misconceptions detected per student

Participant	Primitive assignment works in opposite direction	Primitive assignment works both directions (swaps)	Invalid else-statement	Using else is optional	The natural-language semantics of variable names affects which value gets assigned to which variable	Difficulties in understanding the sequentiality of statements	Adjacent code executes within loop
	Targeted Misconceptions						
1	2 4	3	5			9 13	
2	2	3					
3							
4	3	4					
5							
6	2						
7			6 11		7	8	
8	3						
9			5 6 12		7	8 9 10 13	
10	4	3	6 14		7		
11	4	3	6 11				
12		3					
13		3					
14	2 4	3	6				
15	2 4	3	6		7		
16							
17	2						
18	2 4						
19	2	3					
20	2	3					
21		3					
22	3		5				
23	3		5				
24	3						
25	2 3		6				
27			11			8 9 10	
28			11				
29			11			13	
30							
31							
32							
33		3					
34							
35	1 3 4		6 7				
36	4		5 6 7			9 10	
37	2	1	5			8 9	
38			6			9 10	
39	4		6 7				
40	2 4	1 3	5 6 11 14	7		8 9 10 13	
41	2 4	1 3	5 6 11 14	7		8 9 10 13	
42							
43		3					
44		3					
45		1					
46	1 2 3 4					13	
47	1 2		6 7 11			9	
48						13	
49			11 12 14			13	12
50	2 3	1	5 6 12		7		
51	2 4	1				9 10	
52	4	1	6 11 12 14				
53	3	1 2 4	6 14		7		
54	2 4	1	5 6		7	8 9 10	
55	2	1	14			9	12 14
56		3	5 14	5	7		
57		2 3					
58	2 4		5		7	8	
59	2 3 4		5 6 7		7	9	
60					7	9	
61			11 14		7	8 13	
62	2	1 3 4	11 12			9 13	11 12
63	2 3		6 12			8 9 10 13	
64	4	1	5 6		7	9	
65						9	

Table 6.7: Targeted misconceptions detected per student. The numbers refer to the questions in which the misconceptions appeared, see Appendix A

Participant	Targeted Misconceptions					Other	New		
	Primitive assignment works in opposite direction	Primitive assignment works both directions (swaps)	Invalid else-statement	Using else is optional	The natural-language semantics of variable names affects which value gets assigned to which variable		Difficulties in understanding the sequentiality of statements	Adjacent code executes within loop	A variable can hold multiple values at a time / remembers 'old' value & Difficulties understanding the lifetime of values
1					11			4 8 13	6 10 11 15
2									
3									
4					4			4	
5								3	
6							4	4	
7					3 4			4 7 8	9 10
8					4			3 4	
9					3 12 14		4		5 6 7 8 10 11 13 14
10							4		
11					4				13
12					6				
13					6				
14									6
15					7			8	
16									
17					3			3	
18					3		4	3 4	
19					3				
20					3 4			4	
21					3 4			4	
22					4			3 4	
23					4			3 4	
24					4			3 4	5
25					4			4	5 6 7
27									
28									
29									
30									
31									
32									
33									
34									
35								1 3 4	8 9
36								1 4	6 7
37					3 4				
38									8 9 10
39									6 7 9 10
40					6 11			1 3 4 11 12	6 7 8 10
41					6 11			1 3 4 11 12	6 7 8 10
42									
43									
44									
45					3		4		
46									
47									
48									
49									
50									
51					3		8 9		
52								1 2 3 4 7	5 6 8 9 10 13 15
53								1 2 4 7	8 9 10 13 14 15
54					3			1 2 3 4 8 9	5 6 7 8 9 10 11
55								1 2 3 4	
56								1 4	5 6 8 9 10 11 12 13 14 15
57								2 3	
58								1 3	6 8
59							8 9 10 13		
60						13			
61									
62									
63					3	6 13 15		4	10 13 15
64					3 15			1 3 7	
65					11		8		

Table 6.8: Other misconceptions detected per student. The numbers refer to the questions in which the misconceptions appeared, see Appendix A

7

Discussion

7.1. Part I

With regard to the number of interviews conducted for this research, saturation was the criterion, so no further interviews were scheduled when no useful new information could be gathered. This does not exclude the fact that it is possible that more or other input could have been gathered, should more interviews have been held.

7.2. Part II

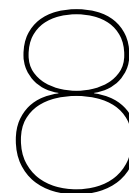
As no previous research guidelines were available with regard to misconception detection through Parsons Problems, suitable research and analysis methods had to be devised from scratch. Mistakes were marked as misconceptions if they matched the expected misconception or if they indicated a non-expected misconception. They were not marked as misconceptions if the mistakes appeared random.

Moreover, students were categorized as holding a certain misconception if the misconception occurred at least once. This choice was made because only a limited number of questions could be included in this experimental research. However, it needs to be noted that if a student made a mistake only once, the chance that the student holds a misconception is smaller than if the student repeated the same mistake in four exercises addressing the same misconception.

Some students did answer only a limited number of questions or gave answers that were nonsensical. Many of the mistakes made by those students were not considered misconceptions as their wrong answers were too random to be classified as such. In this research, mistakes were only marked as misconceptions if they truly showed the characteristics of misconceptions. Therefore, it is important to note that some rows (see Table 6.5 and 6.6), which show few 'x' imply that few mistakes or misconceptions were found for the students. However, even if some students have few 'x' to their name, it could be that they did not answer the question at all or that their answer made no sense at all. Consequently, the actual number of students who gave wrong answers and thus did not understand a concept was higher than the number displayed in Table 6.2 and Table 6.4. Along the same lines, it needs to be noted, that also the number of total mistakes in those questions was higher in actual fact than the numbers displayed in Table 6.3.

Students who made many mistakes may have had trouble applying their knowledge in a new testing method and environment using Parsons Problems. This may have required a level of skill which they may not have mastered yet.

The analysis of answers by relatively accomplished students is thought to have produced more reliable results, because the answers given by those students were generally close to the correct answer, which made it easier to identify misconceptions and minimized the chance of accidental or random mistakes.



Conclusion

This research has shown that teachers and programming education researchers believe that insightful teacher feedback would contribute to improving programming education. They have to teach a difficult subject to many students, and it is almost impossible for them to find out which student needs assistance on which topic. They regret that feedback is usually only provided to the students. It would be very useful for them to receive feedback which does not require time consuming analysis. Such information about individual students or a class as a whole would make it easier for them to provide effective education. If they can intervene in time, teachers feel they could avoid loss of motivation in their students.

Furthermore, this research has found evidence that purposely designed Parsons Problems can detect programming misconceptions. Seven out of eight targeted misconceptions have successfully been detected concerning a considerable number of students. Moreover, other non-targeted misconceptions as well as new misconceptions have been detected. Several analyses have been carried out which appear to provide insightful teacher feedback about individual students and about a group of students as a whole. Based on the data collected in the experiment, teachers can be provided with valuable feedback about which individual student(s) and which percentage of a class holds a certain misconception, which misconception occurs most often, how students perform on particular questions, and other information which might help teachers to better adjust their lessons and testing to the level of understanding of their students. These findings show a lot of promise for the development of insightful teacher feedback based on misconception detection through Parsons Problems.

9

Future Work

Results from this research show promise for the future development of automated feedback tools based on misconception detection through Parsons Problems. As this research was a first attempt to design a method for generating insightful teacher feedback, further research is needed, focusing on three development aspects to be discussed in this chapter.

9.1. Design of Parsons Problems

In order to simplify the data analysis, it would be useful to limit the answer space for wrong answers through the use of 'context' and 'multiple lines' in the Parsons Problems' design. The questions designed for this research contained two to nine puzzle pieces that could be chosen (and were not yet fixed in the right location), which generated a wide variety of other non-targeted misconceptions and mistakes in the students' answers. However, for the purpose of detecting misconceptions, a smaller number of choices would have several advantages: 1) the Parsons Problems design is easier, 2) due to a more limited answer space, results would more likely only include the targeted misconception, and 3) would make the method more suitable for automation. Furthermore, in order to avoid other random mistakes, it is important to ensure that the Parsons Problems are well attuned to the exact knowledge level of the students.

9.2. Design of Experiments

Follow-up experiments might benefit from posing (many) more questions about a targeted misconception. It might also contribute to clarity and conclusiveness of the data if the focus during a future test or experiment should be on less misconceptions at a time. It would be useful to follow-up on the results with instruction directly targeted at the misconceptions found, and consequently carry out a repeat experiment which tests to what extent the misconceptions have been eliminated after intervention.

9.3. Automation - Useful Functionalities

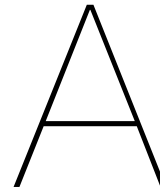
Misconception detection through Parsons Problems appears to be very suitable for automation, particularly if the above recommendations are integrated. A very interesting next step for research would be to create a digital environment in which Parsons Problems can be created by teachers and solved by students, and which outputs analyzed data in the form of insightful and non time-consuming teacher feedback.

In order to further improve the effectiveness of programming education, future research could aim to further extend such an environment with functionalities such as:

- **Automatic student feedback:** based on the detected misconceptions in completed questions, students could be presented with relevant follow-up instructions/tutorials that target those detected misconceptions. This could be a very useful method, since it is very im-

portant to intervene and provide corrective feedback when students show signs of misconceptions [27]

- **Adaptive PP:** it would be interesting to integrate adaptive aspects to the PP, as introduced by Ericson [20], in order to improve the student's capabilities where it is most needed and to keep students motivated
- **Extract patterns:** instead of only analyzing the final solution submitted by a student, it is also possible to track all movements made by a student with the purpose of extracting insightful patterns [29]
- **Gamification:** Parsons Problems offer the added advantage that they can be presented in a gamified setting, which has been proven to work well for young students and/or novice learners. It could be interesting to develop functionalities that make the digital Parsons Problems solving experience feel similar to how one would approach solving an actual puzzle



Experiment Questions and Answers

A.1. Variables

The questions related to variables have been designed to detect if students have misconceptions concerning the direction in which value should be assigned to variables. The second and third question were inspired by a questionnaire developed by Bornat and Dehnadi [10], which focused on misconceptions concerning variables in the context of multiple choice questions.

Question 1

Put the pieces of code in the right order, so that the value of the variable x and the value of the variable y are switched.

! Beware ! Not all the pieces of code have to be used.

$x = 3$
 $y = 5$
 $temp = 0$

<code>temp = x</code>
<code>x = y</code>
<code>y = temp</code>

Distractors:

- `x = temp`
- `temp = y`

Misconceptions targeted:

- Primitive assignment works in opposite direction
- Primitive assignment works both directions (swaps)

Question 2

Put the pieces of code in the right order, so that all variables have the value 7.

! Beware ! Not all the pieces of code have to be used.

$j = 5$
 $k = 3$
 $m = 7$

<code>j = m</code> <code>k = j</code>
--

Distractors:

- | |
|---------|
| $m = j$ |
| $j = k$ |

Misconceptions targeted:

- Primitive assignment works in opposite direction
- Primitive assignment works both directions (swaps)

Question 3

Put the pieces of code in the right order, so that variable k has the value 5.
! Beware ! Not all the pieces of code have to be used.

$j = 5$
 $k = 10$

$m = j$
$k = m$

Distractors:

- | |
|---------|
| $j = m$ |
|---------|
- | |
|---------|
| $m = k$ |
|---------|

Misconceptions targeted:

- Primitive assignment works in opposite direction
- Primitive assignment works both directions (swaps)

Question 4

Put the pieces of code in the right order so that:

1. The number 25 is printed
2. Variable b has the value 23

! Beware ! Not all the pieces of code have to be used.

$j = 23$
$k = j$
$j = 25$

`print(j)`

Distractors:

- | |
|---------|
| $j = k$ |
|---------|

Misconceptions targeted:

- Primitive assignment works in opposite direction
- Primitive assignment works both directions (swaps)

A.2. If/Else Statements

The if/else statement related PP mainly focus on whether students will create valid else statements.

Question 5

Put the pieces of code in the right order so that the number 10 is printed if variable j is smaller than 5, otherwise the number 20 should be printed.

! Beware ! Not all the pieces of code have to be used.

```

j = 3
if j < 5:
    j = 10
else:
    j = 20

```

Distractors:

- else $j \geq 5$:
- if $j \leq 5$:

Misconceptions targeted:

- Invalid else-statement
- Using else is optional

Question 6

Put the pieces of code in the right order so that it is printed that x is a positive number if x is larger than 0, and otherwise it should be printed that it is not a positive number.

! Beware ! Not all the pieces of code have to be used.

```

if x >= 0:
    print('x is a positive number')
else:
    print('x is not a positive number')

```

Distractors:

- else $x < 0$:

Misconceptions targeted:

- Invalid else-statement
- Using else is optional

Question 7

Put the pieces of code in the right order so that the variable `largestNumber` always holds the largest value and `smallestNumber` always holds the smallest value.

! Beware ! Not all the pieces of code have to be used.

```

largestNumber = 0
smallestNumber = 0

```

```

j = 1
k = 20

if j >= k:
    largestNumber = j
    smallestNumber = k
else:
    largestNumber = k
    smallestNumber = j

```

Distractors:

- else j < k:

Misconceptions targeted:

- Invalid else-statement
- Using else is optional
- The natural-language semantics of variable names affects which value gets assigned to which variable

A.3. While Loops

These PP typically revolve around detecting whether students have difficulties understanding the sequentiality of statements, as well as detecting whether students understand that code located adjacent to a loop or if statement does not execute within the body of that loop or if statement.

Question 8

Put the pieces of code in the right order so that the following number sequence is printed: 1 2 3 4 5 6 7 8 9 10

```

i = 0
while i < 10:
    i = i + 1
    print(i)

```

Distractors: None.

Misconceptions targeted:

- Difficulties in understanding the sequentiality of statements

Question 9

Put the pieces of code in the right order so that the following number sequence is printed: 4 6 8

```

k = 4
while k < 9:
    print(k)
    k = k + 2

```

Distractors: None.

Misconceptions targeted:

- Difficulties in understanding the sequentiality of statements

Question 10

Put the pieces of code in the right order so that an uneven number is printed.

```

j = 2
while j < 3:
    j = j + 1
    print(j)

```

Distractors: None.

Misconceptions targeted:

- Difficulties in understanding the sequentiality of statements

Question 11

Put the pieces of code in the right order so that:

1. The even numbers are printed, and
2. After that, variable m always increases with 1

! Beware ! Not all the pieces of code have to be used.

```

m = 1
while m < 8:
    if (m%2) == 0:
        print(m)
    m = m + 1

```

Distractors:

- else:
- else (m%2) != 0:

Misconceptions targeted:

- Adjacent code executes within loop
- Invalid else-statement

Question 12

Put the pieces of code in the right order so that as long as p is not 0, the following happens:

1. Only if p is positive, the value of q should increase
2. With every iteration, a new input should be requested and q should be printed

! Beware ! Not all the pieces of code have to be used.

```

q = 0
p = int(input("Vul een getal in:"))

```

```

while p != 0:
    if p > 0:

```

```

q = q + 1
p = int(input('Vul een getal in:'))
print(q)

```

Distractors:

- else:
- else p < 0:

Misconceptions targeted:

- Adjacent code executes within loop
- Invalid else-statement

A.4. For Loops

These PP simply try to test whether students understand the concept of for loops.

Question 13

Put the pieces of code in the right order so that the following number sequence is printed: 0 1 3 6

```

x = 0
for z in range(4):
    x = x + z
print(x)

```

Distractors: None.

Misconceptions targeted:

- Difficulties in understanding the sequentiality of statements
- For loop control variables do not have values inside the loop or their values can be arbitrarily changed

Question 14

Put the pieces of code in the right order so that:

1. Variable `n` is printed if `n` is an even number
2. Every iteration the value of `x` increases

! Beware ! Not all the pieces of code have to be used.

```

x = 5
for n in range(7):
    if (n%2) == 0:
        print(n)
x = x + 1

```

Distractors:

- else:
- else (n%2) != 0:

Misconceptions targeted:

- Adjacent code executes within loop
- Invalid else-statement

Question 15

Put the pieces of code in the right order so that a square is drawn.
! Beware ! Not all the pieces of code have to be used.

for i in range(4):

pen.forward(100)

pen.left(90)

Distractors:

- for i in range(3):

Misconceptions targeted: No specific misconception was targeted in this question. The main purpose was to check whether students understood the concept of a for loop and its control variable in a relatively simple question.

B

Experiment Google Forms

This appendix includes the entire Parsons Problems experiment as it was presented to the participants through Google Forms.

Section 1 of 16

Parsons Problems



Parsons Problems zijn puzzels die bestaan uit stukjes code die door elkaar staan.

Bij elke puzzel moet je uitvinden op welke volgorde de code stukjes moeten staan, zodat de code doet wat de vraag zegt.

Je krijgt bij elke vraag steeds:

1. Wat de code straks moet kunnen doen
2. De puzzel stukjes die je mag gebruiken (deze stukjes hebben een nummer ervoor staan)
3. Een afbeelding met drie strepen, waaronder a, b en c staat. Als je code schrijft moet je soms een regel code "indenteren", bijvoorbeeld in een loop. Daarom moet je ook aangeven op welke "lijn" je elk stukje code wilt hebben.

VOORBEELD:

Als je als antwoord wilt zeggen dat de code stukjes 2, 4 en 5 gebruikt moeten worden op volgorde 4, 2 en 5, dan antwoord je die onder elkaar op deze manier:

```
4
2
5
```

Maar om aan te geven op welk lijntje ze moeten staan, zet je achter elk nummer een letter (a, b of c), bijvoorbeeld:

```
4 a
2 a
5 b
```

Als je iets niet snapt hoe het werkt of wat de bedoeling is, steek dan je hand op, dan wordt je geholpen.

Vul hier graag het nummer in dat op jouw uitgedeelde papier staat: *

Short-answer text

After section 1 Continue to next section 

Section 2 of 16

Vraag 1 - Variabelen



Zet de stukjes code op de goede volgorde zodat de waarden van x en y gewisseld worden.
! Let op ! Je hoeft niet alle stukjes te gebruiken.

1. `x = y`
2. `temp = x`
3. `y = temp`
4. `temp = y`
5. `x = temp`

Deze dik gedrukte code staat al op de goede plaats.

```
x = 3  
y = 5  
temp = 0
```

|
|
|
a b c

Zet hier de nummers van de stukjes code die je wilt gebruiken op de goede volgorde. Vergeet *
niet achter elk stukje te zetten op welk lijntje het hoort te staan (a, b of c).

Long-answer text

After section 2 Continue to next section



Section 3 of 16

Vraag 2 - Variabelen



Zet de code stukjes in de goede volgorde zodat alle variabele de waarde 7 hebben.
! Let op ! Je hoeft niet alle stukjes te gebruiken.

! Let op! Deze puzzelstukjes bestaan dus uit 2 regels code.

1.

<code>j = m</code>
<code>k = j</code>

2.

<code>m = j</code>
<code>j = k</code>

Deze dik gedrukte code staat al op de goede plaats.

j = 5
k = 3
m = 7

|

a

|

b

|

c

Zet hier de nummers van de stukjes code die je wilt gebruiken op de goede volgorde. Vergeet * niet achter elk stukje te zetten op welk lijntje het hoort te staan (a, b of c).

Long-answer text

After section 3 Continue to next section



Section 4 of 16

Vraag 3 - Variabelen



Zet de code stukjes in de goede volgorde zodat variabele k de waarde 5 heeft.
! Let op ! Je hoeft niet alle stukjes te gebruiken.

1. $k = m$
2. $m = j$
3. $m = k$
4. $j = m$

Deze dik gedrukte code staat al op de goede plaats.

```
j = 5  
k = 10
```

```
|  
|  
|  
a  b  c
```

Zet hier de nummers van de stukjes code die je wilt gebruiken op de goede volgorde. Vergeet * niet achter elk stukje te zetten op welk lijntje het hoort te staan (a, b of c).

Long-answer text

After section 4 Continue to next section



Section 5 of 16

Vraag 4 - Variabelen



Zet de code stukjes in de goede volgorde zodat:

1. Het getal 25 geprint wordt, en
2. Variabele k de waarde 23 heeft

! Let op ! Je hoeft niet alle stukjes te gebruiken.

1. `j = 25`
2. `j = 23`
3. `j = k`
4. `k = j`

Deze dik gedrukte code staat al op de goede plaats.

|
a

|
b

|
c

`print(j)`

Zet hier de nummers van de stukjes code die je wilt gebruiken op de goede volgorde. Vergeet ^{*} niet achter elk stukje te zetten op welk lijntje het hoort te staan (a, b of c).

Long-answer text

After section 5 Continue to next section



Section 6 of 16

Vraag 5 - If Statements

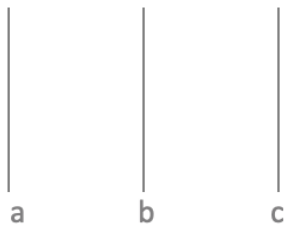


Zet de code stukjes in de goede volgorde zodat j de waarde 10 krijgt als variabele j kleiner dan 5 is, anders krijgt j de waarde 20.

! Let op ! Je hoeft niet alle stukjes te gebruiken.

1. if j < 5:
2. else j >= 5:
3. j = 10
4. j = 3
5. if j <= 5:
6. else:
7. j = 20

Image title



Zet hier de nummers van de stukjes code die je wilt gebruiken op de goede volgorde. Vergeet ^{*} niet achter elk stukje te zetten op welk lijntje het hoort te staan (a, b of c).

Long-answer text

After section 6 Continue to next section



Section 7 of 16

Vraag 6 - If Statements

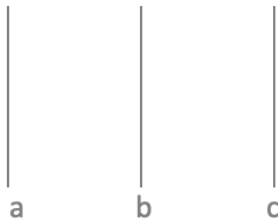


Zet de code stukjes op de goede volgorde zodat er geprint wordt dat het een positief getal is als x groter is dan 0, en anders dat er geprint wordt dat het niet een positief getal is.

! Let op ! Je hoeft niet alle stukjes te gebruiken.

1. `else x < 0:`
2. `print('Dit is een positief getal')`
3. `print('Dit is niet een positief getal')`
4. `if x >= 0:`
5. `else:`

Image title



Zet hier de nummers van de stukjes code die je wilt gebruiken op de goede volgorde. Vergeet niet achter elk stukje te zetten op welk lijntje het hoort te staan (a, b of c). *

Long-answer text

After section 7 Continue to next section



Section 8 of 16

Vraag 7 - If Statements



Zet de code stukjes op de goede volgorde zodat de variabele `grootsteGetal` altijd het grootste getal bevat en zodat `kleinsteGetal` altijd het kleinste getal bevat.

! Let op ! Je hoeft niet alle stukjes te gebruiken.

1. `if j >= k:`
2. `grootsteGetal = j`
3. `k = 20`
4. `grootsteGetal = k`
5. `else j < k:`
6. `kleinsteGetal = k`
7. `j = 1`
8. `kleinsteGetal = j`
9. `else:`

Deze dik gedrukte code staat al op de goede plaats.

```
grootsteGetal = 0  
kleinsteGetal = 0
```

a b c

Zet hier de nummers van de stukjes code die je wilt gebruiken op de goede volgorde. Vergeet ^{*} niet achter elk stukje te zetten op welk lijntje het hoort te staan (a, b of c).

Long-answer text

After section 8 Continue to next section



Section 9 of 16

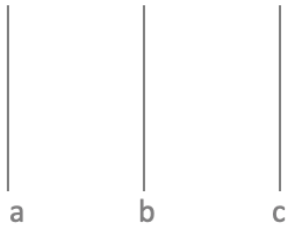
Vraag 8 - While Loops



Zet de code stukjes op de goede volgorde zodat de volgende getallen reeks geprint wordt: 1 2 3 4 5 6 7 8 9 10

1. `print(i)`
2. `i = 0`
3. `i = i + 1`
4. `while i < 10:`

Image title



Zet hier de nummers van de stukjes code op de goede volgorde. Vergeet niet achter elk stukje te zetten op welk lijntje het hoort te staan (a, b of c). *

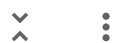
Long-answer text

After section 9 Continue to next section



Section 10 of 16

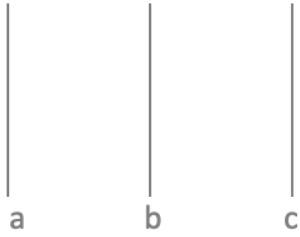
Vraag 9 - While Loops



Zet de code stukjes op de goede volgorde zodat de volgende getallen reeks geprint wordt: 4 6 8

1. `print(k)`
2. `while k < 9:`
3. `k = 4`
4. `k = k + 2`

Image title



Zet hier de nummers van de stukjes code op de goede volgorde. Vergeet niet achter elk stukje te zetten op welk lijntje het hoort te staan (a, b of c).

Long-answer text

After section 10 Continue to next section



Section 11 of 16

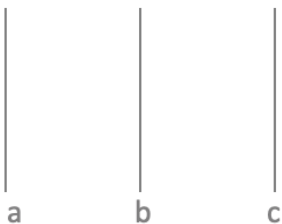
Vraag 10 - While Loops



Zet de code stukjes op de goede volgorde zodat een oneven getal geprint wordt.

1. `while j < 3:`
2. `j = 2`
3. `print(j)`
4. `j = j + 1`

Image title



Zet hier de nummers van de stukjes code op de goede volgorde. Vergeet niet achter elk stukje te zetten op welk lijntje het hoort te staan (a, b of c). *

Long-answer text

After section 11 Continue to next section ▼

Section 12 of 16

Vraag 11 - While Loops

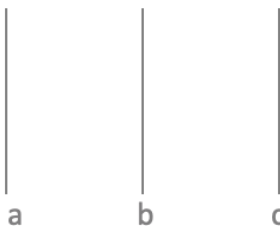


Zet de code stukjes op de goede volgorde zodat:

1. De even getallen geprint worden, en
 2. Variabele m daarna altijd toeneemt met 1
- ! Let op ! Je hoeft niet alle stukjes te gebruiken.

1. while m < 8:
2. else (m%2) != 0:
3. if (m%2) == 0:
4. m = m + 1
5. m = 1
6. else:
7. print(m)

Image title



Zet hier de nummers van de stukjes code die je wilt gebruiken op de goede volgorde. Vergeet niet achter elk stukje te zetten op welk lijntje het hoort te staan (a, b of c). *

Long-answer text

After section 12 Continue to next section ▼

Section 13 of 16

Vraag 12 - While Loops



Zet de code stukjes op de goede volgorde zodat zolang p geen 0 is het volgende gebeurt:

1. Alleen als p positief is, dat de waarde van q toeneemt
 2. Verder dat er bij elke iteratie een nieuwe input wordt gevraagd en dat q geprint wordt
- ! Let op ! Je hoeft niet alle stukjes te gebruiken.

1. if p > 0:
2. else:
3. q = q + 1
4. while p != 0:
5. p = int(input('Vul een getal in:'))
6. print(q)
7. else p < 0:

Deze dik gedrukte code staat al op de goede plaats.

```
q = 0  
p = int(input('Vul een getal in:'))
```

a b c

Zet hier de nummers van de stukjes code die je wilt gebruiken op de goede volgorde. Vergeet niet achter elk stukje te zetten op welk lijntje het hoort te staan (a, b of c). *

Long-answer text

After section 13 Continue to next section ▼

Section 14 of 16

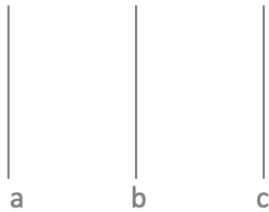
Vraag 13 - For Loops



Zet de code stukjes in de goede volgorde zodat de volgende reeks geprint wordt: 0 1 3 6

1. `for z in range(4):`
2. `x = 0`
3. `print(x)`
4. `x = x + z`

Image title



Zet hier de nummers van de stukjes code op de goede volgorde. Vergeet niet achter elk stukje te zetten op welk lijntje het hoort te staan (a, b of c). *

Long-answer text

After section 14 Continue to next section



Section 15 of 16

Vraag 14 - For Loops



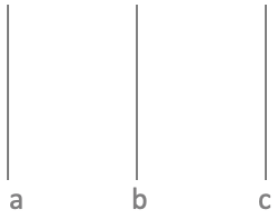
Zet de code stukjes op de goede volgorde zodat:

1. Variabele n geprint wordt als n een even getal is, en
2. Dat tijdens elke iteratie de waarde van x toeneemt

! Let op ! Je hoeft niet alle stukjes te gebruiken.

1. `if (n%2) == 0:`
2. `x = 5`
3. `for n in range(7):`
4. `else (n%2) != 0:`
5. `print(n)`
6. `else:`
7. `x = x + 1`

Image title



Zet hier de nummers van de stukjes code die je wilt gebruiken op de goede volgorde. Vergeet * niet achter elk stukje te zetten op welk lijntje het hoort te staan (a, b of c).

Long-answer text

After section 15 Continue to next section ▼

Section 16 of 16

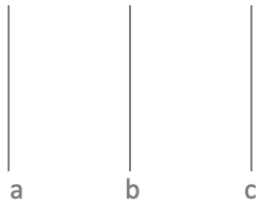
Vraag 15 - For Loops



Zet de code stukjes op de goede volgorde zodat een vierkant getekend wordt.
! Let op ! Je hoeft niet alle stukjes te gebruiken.

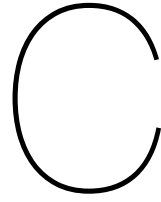
1. `pen.left(90)`
2. `for i in range(4):`
3. `for i in range(3):`
4. `pen.forward(100)`

Image title



Zet hier de nummers van de stukjes code die je wilt gebruiken op de goede volgorde. Vergeet * niet achter elk stukje te zetten op welk lijntje het hoort te staan (a, b of c).

Long-answer text



Experiment Collected Data

In this appendix, all the collected raw data from the Parsons Problems experiment is presented for all 64 participants.

Times Lamp	ID	Verbleiden	If Statements	While Loops	For Statements
1/15/2020 9:16:49	7	2a 1a, 3a	1a 2b 3a	6c 7a	10c 12a
1/15/2020 9:17:00	7	2a 1a 3a	1a 2b 3a	6c 7a	10c 12a
1/15/2020 9:17:54	24	2a 1a 3a	1a 2b 3a	6c 7a	10c 12a
1/15/2020 9:18:16	25	2a 1a 3a	1a 2b 3a	6c 7a	10c 12a
1/15/2020 9:18:32	22	2a 1a 3a	1a 2b 3a	6c 7a	10c 12a
1/15/2020 9:20:20	20	2a 1a 3a	1a 2b 3a	6c 7a	10c 12a
1/15/2020 9:20:20	11	2a 1a 3a	1a 2b 3a	6c 7a	10c 12a
1/15/2020 9:24:47	2	2A 1A,3A	m=j=k	x	x

1/15/2020 9:24:55	1	2a 1a 3a	m=j=k	20	2a 1a 3a
1/15/2020 9:25:27	22	2A 1A,3A	m=j=k	20	2a 1a 3a
1/15/2020 9:25:53	14	2A 1A,3A	m=j=k	20	2a 1a 3a
1/15/2020 9:26:01	21	2a 1a 3a	m=j=k	20	2a 1a 3a
1/15/2020 9:26:02	12	2a 1a 3a	m=j=k	20	2a 1a 3a
1/15/2020 9:26:06	15	2a 1a 3a	m=j=k	20	2a 1a 3a
1/15/2020 9:26:08	19	2a 1a 3a	m=j=k	20	2a 1a 3a
1/15/2020 9:26:10	3	2a 1a, 3a	m=j=k	20	2a 1a 3a
1/15/2020 9:26:11	18	2a 1a, 3a	m=j=k	20	2a 1a 3a
1/15/2020 9:26:12	16	2a 1a 3a	m=j=k	20	2a 1a 3a

1/15/2020 9:26:26	17	2a 1a 3a	m=j=k	20	2a 1a 3a
1/15/2020 9:26:43	5	5a 3a, 1b	m=j=k	20	2a 1a 3a
1/15/2020 9:26:48	28	2a 1a 3a	m=j=k	20	2a 1a 3a
1/15/2020 9:26:55	27	2a 1a 3a	m=j=k	20	2a 1a 3a
1/15/2020 9:26:56	39	5a 4a 3a	m=j=k	20	2a 1a 3a
1/15/2020 9:27:05	37	5a 4a 3a	m=j=k	20	2a 1a 3a
1/15/2020 9:27:17	49	1a 2a 3a	m=j=k	20	2a 1a 3a
1/15/2020 9:27:19	44	2a 1a 3a	m=j=k	20	2a 1a 3a
1/15/2020 9:27:31	32	2a 1a 3a	m=j=k	20	2a 1a 3a
1/15/2020 9:27:56	31	2a 1a 3a	m=j=k	20	2a 1a 3a
1/15/2020 9:28:47	30	2a 1a 3a	m=j=k	20	2a 1a 3a
1/15/2020 9:28:47	33	2a 1a 3a	m=j=k	20	2a 1a 3a
1/15/2020 9:28:59	48	2a 1a 3a	m=j=k	20	2a 1a 3a
1/15/2020 9:29:17	65	2a 1a 3a	m=j=k	20	2a 1a 3a
1/15/2020 9:29:18	58	2b 4b 1c	m=j=k	20	2a 1a 3a
1/15/2020 9:29:57	59	2b 4b 1c	m=j=k	20	2a 1a 3a
1/15/2020 9:29:56	57	2b 4b 1c	m=j=k	20	2a 1a 3a
1/15/2020 9:29:57	51	2b 4b 1c	m=j=k	20	2a 1a 3a

1/15/2020 9:30:38	59	x	a=2	a=2	a=2
1/15/2020 9:30:43	52	2a 1a 5b	a=4	a=4	a=4
1/15/2020 9:31:07	60	2a 1a 3a	a=4	a=4	a=4
1/15/2020 9:31:28	65	2a 1a 3a	a=4	a=4	a=4
1/15/2020 9:34:36	46	3a 1a 2a	a=4	a=4	a=4

1/15/2020 9:36:35	53	2a 3a 1b	a=4	a=4	a=4
1/15/2020 9:36:43	52	2a 3a 1b	a=4	a=4	a=4
1/15/2020 9:38:31	47	2a 3a 1b	a=4	a=4	a=4
1/15/2020 9:38:46	64	x	a=4	a=4	a=4
1/15/2020 9:41:37	64	x	a=4	a=4	a=4
1/15/2020 9:44:24	56	2b 1b 3b	a=4	a=4	a=4

1/15/2020 9:48:49	7	2a 1a, 3a	1a 2b 3a	6c 7a	10c 12a
1/15/2020 9:49:00	7	2a 1a 3a	1a 2b 3a	6c 7a	10c 12a
1/15/2020 9:49:54	24	2a 1a 3a	1a 2b 3a	6c 7a	10c 12a
1/15/2020 9:51:16	25	2a 1a 3a	1a 2b 3a	6c 7a	10c 12a
1/15/2020 9:51:32	22	2a 1a 3a	1a 2b 3a	6c 7a	10c 12a
1/15/2020 9:52:20	20	2a 1a 3a	1a 2b 3a	6c 7a	10c 12a
1/15/2020 9:52:20	11	2a 1a 3a	1a 2b 3a	6c 7a	10c 12a

1/15/2020 9:24:47	2	2A 1A,3A	m=j=k	x	x
1/15/2020 9:24:55	1	2a 1a 3a	m=j=k	20	2a 1a 3a
1/15/2020 9:25:27	22	2A 1A,3A	m=j=k	20	2a 1a 3a
1/15/2020 9:25:53	14	2A 1A,3A	m=j=k	20	2a 1a 3a
1/15/2020 9:26:01	21	2a 1a 3a	m=j=k	20	2a 1a 3a
1/15/2020 9:26:02	12	2a 1a 3a	m=j=k	20	2a 1a 3a
1/15/2020 9:26:06	15	2a 1a 3a	m=j=k	20	2a 1a 3a
1/15/2020 9:26:08	19	2a 1a 3a	m=j=k	20	2a 1a 3a
1/15/2020 9:26:10	3	2a 1a, 3a	m=j=k	20	2a 1a 3a
1/15/2020 9:26:11	18	2a 1a, 3a	m=j=k	20	2a 1a 3a
1/15/2020 9:26:12	16	2a 1a 3a	m=j=k	20	2a 1a 3a

1/15/2020 9:26:26	17	2a 1a 3a	m=j=k	20	2a 1a 3a
1/15/2020 9:26:43	5	5a 3a, 1b	m=j=k	20	2a 1a 3a
1/15/2020 9:26:48	28	2a 1a 3a	m=j=k	20	2a 1a 3a
1/15/2020 9:26:55	27	2a 1a 3a	m=j=k	20	2a 1a 3a
1/15/2020 9:26:56	39	5a 4a 3a	m=j=k	20	2a 1a 3a
1/15/2020 9:27:05	37	5a 4a 3a	m=j=k	20	2a 1a 3a
1/15/2020 9:27:17	49	1a 2a 3a	m=j=k	20	2a 1a 3a
1/15/2020 9:27:19	44	2a 1a 3a	m=j=k	20	2a 1a 3a
1/15/2020 9:27:31	32	2a 1a 3a	m=j=k	20	2a 1a 3a
1/15/2020 9:27:56	31	2a 1a 3a	m=j=k	20	2a 1a 3a
1/15/2020 9:28:47	30	2a 1a 3a	m=j=k	20	2a 1a 3a
1/15/2020 9:28:47	33	2a 1a 3a	m=j=k	20	2a 1a 3a
1/15/2020 9:28:59	48	2a 1a 3a	m=j=k	20	2a 1a 3a
1/15/2020 9:29:17	65	2a 1a 3a	m=j=k	20	2a 1a 3a
1/15/2020 9:29:18	58	2b 4b 1c	m=j=k	20	2a 1a 3a
1/15/2020 9:29:57	59	2b 4b 1c	m=j=k	20	2a 1a 3a
1/15/2020 9:29:56	57	2b 4b 1c	m=j=k	20	2a 1a 3a
1/15/2020 9:29:57	51	2b 4b 1c	m=j=k	20	2a 1a 3a

1/15/2020 9:30:38	59	x	a=2	a=2	a=2
1/15/2020 9:30:43	52	2a 1a 5b	a=4	a=4	a=4
1/15/2020 9:31:07	60	2a 1a 3a	a=4	a=4	a=4
1/15/2020 9:31:28	65	2a 1a 3a	a=4	a=4	a=4
1/15/2020 9:34:36	46	3a 1a 2a	a=4	a=4	a=4
1/15/2020 9:36:35	53	2a 3a 1b	a=4	a=4	a=4
1/15/2020 9:36:43	52	2a 3a 1b	a=4	a=4	a=4
1/15/2020 9:38:31	47	2a 3a 1b	a=4	a=4	a=4
1/15/2020 9:38:46	64	x	a=4	a=4	a=4
1/15/2020 9:41:37	64	x	a=4	a=4	a=4
1/15/2020 9:44:24	56	2b 1b 3b	a=4	a=4	a=4

Bibliography

- [1] Monitor impuls leraren tekortvakken. <https://www.cpb.nl/sites/default/files/omnidownload/CPB-Notitie-16jan2019-Monitor-impuls-leraren-tekortvakken.pdf>. Accessed: 2020-01-18.
- [2] <https://codingbat.com/about.html>. Accessed: 2020-01-18.
- [3] js-parsons - a javascript library for parsons problems. <https://js-parsons.github.io>. Accessed: 2020-02-22.
- [4] Parsons problems. <https://runestone.academy/runestone/static/authorguide/directives/parsons.html>. Accessed: 2020-02-01.
- [5] A. Alammery, A. Carbone, and J. Sheard. Implementation of a smart lab for teachers of novice programmers. *ACE '12 Proceedings of the Fourteenth Australasian Computing Education Conference*, pages 121–130, 2012.
- [6] D.M. Arnow and O. Barshay. Webtoteach: An interactive focused programming exercise system. *FIE'99 Frontiers in Education. 29th Annual Frontiers in Education Conference. Designing the Future of Science and Engineering Education*, 1999. URL <https://doi.org/10.1109/FIE.1999.839303>.
- [7] E. Balzuweit and J. Spacco. Snapviz: Visualizing programming assignment snapshots. *Proceedings of the 18th ACM conference on Innovation and Technology in Computer Science Education*, pages 350–350, 2013. URL <https://doi.org/10.1145/2462476.2465615>.
- [8] G. Bari, A. Gaspar, P. Wiegand, D. Vitel, K. Cheng Tan, and S.J. Kozakoff. On the potential of evolved parsons puzzles to contribute to concept inventories in computer programming. *Proceedings of ASEE'S 126th Annual Conference*, 2019.
- [9] A. Basawapatna and A. Repenning. Employing retention of flow to improve online tutorials. *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, page 63–68, 2017. URL <https://doi.org/10.1145/3017680.3017799>.
- [10] R. Bornat and S. Dehnadi. Generating, administering and marking dehnadi-style tests. 2016.
- [11] N.C.C. Brown, A. Altadmri, S. Sentance, and M. Kölling. Blackbox, five years on: An evaluation of a large-scale programming data collection project. *Proceedings of the 2018 ACM Conference on International Computing Education Research, ICER '18*, pages 196–204, 2018. URL <https://doi.org/10.1145/3230977.3230991>.
- [12] J. W. Creswell. *Qualitative inquiry and research design: Choosing among five traditions*. Sage Publications, Thousand Oaks, California, 1998.
- [13] D. Crow. Why every child should learn to code. <https://www.theguardian.com/technology/2014/feb/07/year-of-code-dan-crow-songkick>. Accessed: 2020-01-18.
- [14] E. Deitrick. Codio introduces a new type of assessment: Parson's problems. <https://www.codio.com/blog/parsons-problems>. Accessed: 2020-02-01.
- [15] C. Di, Z. Gang, and X. Juhong. An introduction to the technology of blending-reality smart classroom. *2008 International Symposium on Knowledge Acquisition and Modeling*, pages 516–519, 2008. URL <https://doi.org/10.1109/KAM.2008.172>.

- [16] D. Doukakis, M. Grigoriadou, and G. Tsaganou. Understanding the programming variable concept with animated interactive analogies. *Proceedings of the 8th Hellenic European Research on Computer Mathematics and its Applications Conference, HERCMA '07*, 2007.
- [17] B. Du Boulay. Some difficulties of learning to program. *Journal of Educational Computing Research*, pages 57–73, 1986. URL <https://doi.org/10.2190/3LFX-9RRF-67T8-UVK9>.
- [18] B.J. Ericson, M.J. Guzdial, and B.B. Morrison. Analysis of interactive features designed to enhance learning in an ebook. *Proceedings of the 11th annual International Conference on International Computing Education Research*, pages 169–178, 2015. URL <https://doi.org/10.1145/2787622.2787731>.
- [19] B.J. Ericson, L.E. Margulieux, and J. Rick. Solving parsons problems versus fixing and writing code. *Proceedings of the 17th Koli Calling International Conference on Computing Education Research*, page 20–29, 2017. URL <https://doi.org/10.1145/3141880.3141895>.
- [20] B.J. Ericson, J.D. Foley, and J. Rick. Evaluating the efficiency and effectiveness of adaptive parsons problems. *Proceedings of the 2018 ACM Conference on International Computing Education Research*, page 60–68, 2018. URL <https://doi.org/10.1145/3230977.3231000>.
- [21] G.V.F. Fabric, A. Mitrovic, and K. Neshatian. Evaluation of parsons problems with menu-based self-explanation prompts in a mobile python tutor. *International Journal of Artificial Intelligence in Education*, page 507–535, 2019. URL <https://doi.org/10.1007/s40593-019-00184-0>.
- [22] S. Garner. An exploration of how a technology-facilitated part-complete solution method supports the learning of computer programming. *Journal of Issues in Informing Science and Information Technology*, 2007. URL <https://doi.org/10.28945/966>.
- [23] N. Gil Fonseca, L.F.K. Macedo, and A. José Mendes. Supporting differentiated instruction in programming courses through permanent progress monitoring. *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, pages 209–214, 2018. URL <https://doi.org/10.1145/3159450.3159578>.
- [24] J.M. Griffin. Learning by taking apart: Deconstructing code by reading, tracing, and debugging. *Proceedings of the 17th Annual Conference on Information Technology Education*, page 148–153, 2016. URL <https://doi.org/10.1145/2978192.2978231>.
- [25] K.J. Harms, N. Rowlett, and C. Kelleher. Enabling independent learning of programming concepts through programming completion puzzles. *2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 271–279, 2015. URL <https://doi.org/10.1109/VLHCC.2015.7357226>.
- [26] K.J. Harms, E. Balzuweit, J. Chen, and C. Kelleher. Learning programming from tutorials and code puzzles: Children’s perceptions of value. *2016 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 59–67, 2016. URL <https://doi.org/10.1109/VLHCC.2016.7739665>.
- [27] K.J. Harms, J. Chen, and C.L. Kelleher. Distractors in parsons problems decrease learning efficiency for young novice programmers. *Proceedings of the 2016 ACM Conference on International Computing Education Research*, page 241–250, 2016. URL <https://doi.org/10.1145/2960310.2960314>.
- [28] K. Heinonen, K. Hirvikoski, M. Luukkainen, and A. Vihavainen. Using codebrowser to seek differences between novice programmers. *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*, pages 229–234, 2014. URL <https://doi.org/10.1145/2538862.2538981>.
- [29] J. Helminen, P. Ihanntola, V. Karavirta, and L. Malmi. How do students solve parsons programming problems? - an analysis of interaction traces. *Proceedings of the Ninth Annual International Conference on Computing Education Research*, page 119–126, 2012. URL <https://doi.org/10.1145/2361276.2361300>.

- [30] M. Hristova, A. Misra, M. Rutter, and R. Mercuri. Identifying and correcting java programming errors for introductory computer science students. *Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education*, pages 153–156, 2003. URL <https://doi.org/10.1145/611892.611956>.
- [31] C.D. Hundhausen, D.M. Olivares, and A.S. Carter. Ide-based learning analytics for computing education: A process model, critical review, and research agenda. *ACM Transactions on Computing Education 17, TOCE 17*, pages 1–26, 2017.
- [32] P. Ihanntola and V. Karavirta. Two-dimensional parson’s puzzles: The concept, tools, and first observations. *Journal of Information Technology Education: Innovations in Practice*, pages 1–14, 2011. URL <https://doi.org/10.28945/1394>.
- [33] P. Ihanntola, A. Vihavainen, A. Ahadi, M. Butler, J. Börstler, S.H. Edwards, E. Isohanni, A. Korhonen, A. Petersen, K. Rivers, M.A. Rubio, J. Sheard, B. Skupas, J. Spacco, C. Szabo, and D. Toll. Educational data mining and learning analytics in programming: Literature review and case studies. *Proceedings of the 2015 ITICSE on Working Group Reports, ITICSE-WGR '15*, pages 41–63, 2015. URL <https://doi.org/10.1145/2858796.2858798>.
- [34] H. Isenstein. Why i taught myself to code – and why you should too. <https://www.theguardian.com/education/2017/feb/09/why-i-taught-myself-to-code-and-why-you-should-too>. Accessed: 2020-01-18.
- [35] M.C. Jadud. Methods and tools for exploring novice compilation behaviour. *Proceedings of the Second International Workshop on Computing Education Research, ICER '06*, pages 73–84, 2006. URL <https://doi.org/10.1145/1151588.1151600>.
- [36] M. Karam, M. Awa, A. Carbone, and J. Dargham. Assisting students with typical programming errors during a coding session. *Seventh International Conference on Information Technology*, pages 42–47, 2010.
- [37] V. Karavirta, J. Helminen, and P. Ihanntola. A mobile learning application for parsons problems with automatic feedback. *Proceedings of the 12th Koli Calling International Conference on Computing Education Research*, page 11–18, 2012. URL <https://doi.org/10.1145/2401796.2401798>.
- [38] M. Kenny and R. Fourie. Tracing the history of grounded theory methodology: From formation to fragmentation. *The Qualitative Report*, 19(52):1–9, 2014.
- [39] T. Kohn. Teaching python programming to novices: Addressing misconceptions and creating a development environment. 2017. URL <https://doi.org/10.3929/ethz-a-010871088>.
- [40] A.N. Kumar. Epplets: A tool for solving parsons puzzles. *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, page 527–532, 2018. URL <https://doi.org/10.1145/3159450.3159576>.
- [41] L. Ma. Investigating and improving novice programmers’ mental models of programming concepts. 2007.
- [42] C. Murphy, G.E. Kaiser, K. Loveland, and S. Hasan. Retina: Helping students and instructors based on observed programming activities. *Proceedings of the 40th ACM Technical Symposium on Computer Science Education*, pages 178–182, 2009. URL <https://doi.org/10.1145/1539024.1508929>.
- [43] C. Norris, F. Barry, J.B. Fenwick Jr, K. Reid, and J. Rountree. Clockit: Collecting quantitative data on how beginning software developers really work. *Proceedings of the 13th annual conference on Innovation and Technology in Computer Science Education*, pages 37–41, 2008. URL <https://doi.org/10.1145/1384271.1384284>.

- [44] A. Papancea, J. Spacco, and D. Hovemeyer. An open platform for managing short programming exercises. *Proceedings of the 9th annual international ACM conference on International computing education research*, pages 47–52, 2013. URL <https://doi.org/10.1145/2493394.2493401>.
- [45] D. Parsons and P. Haden. Parson’s programming puzzles: A fun and effective learning tool for first programming courses. *Proceedings of the 8th Australasian Conference on Computing Education*, page 157–163, 2006.
- [46] R.D. Pea. Language-independent conceptual ‘bugs’ in novice programming. *Journal of Educational Computing Research*, pages 25–36, 1986. URL <https://doi.org/10.2190/689T-1R2A-X4W4-29J2>.
- [47] R.T. Putnam, D. Sleeman, J.A. Baxter, and L.K. Kuspa. A summary of misconceptions of high school basic programmers. *Journal of Educational Computing Research*, pages 459–472, 1986. URL <https://doi.org/10.2190/FGN9-DJ2F-86V8-3FAU>.
- [48] N. Ragonis and M. Ben-Ari. A long-term investigation of the comprehension of oop concepts by novices. *Computer Science Education*, pages 203–221, 2005. URL <https://doi.org/10.1080/08993400500224310>.
- [49] T. Rajala, M.J. Laakso, E. Kaila, and T. Salakoski. Ville – a language-independent program visualization tool. *Proceedings of the Seventh Baltic Sea Conference on Computing Education Research*, page 151–159, 2007.
- [50] L. Scholten. Programming misconceptions in novice python learners; challenges and proposed solutions. 2019.
- [51] E. Shein. The cs teacher shortage. <https://cacm.acm.org/magazines/2019/10/239667-the-cs-teacher-shortage/fulltext>. Accessed: 2020-01-18.
- [52] Simon. Assignment and sequence: Why some students can’t recognize a simple swap. *Proceedings of the 11th Koli Calling International Conference on Computing Education Research, Koli Calling ’11*, page 10–15, 2011. URL <https://doi.org/10.1145/2094131.2094134>.
- [53] D. Sleeman, R.T. Putnam, J. Baxter, and L. Kuspa. Pascal and high school students: A study of errors. *Journal of Educational Computing Research*, pages 5–23, 1986. URL <https://doi.org/10.2190/2XPP-LTYH-98NQ-BU77>.
- [54] E. Soloway, K. Ehrlich, J. Bonar, and J. Greenspan. What do novices know about programming? *Directions in Human-Computer Interaction*, 1982.
- [55] J. Sorva. Visual program simulation in introductory programming education. 2012.
- [56] J. Spacco, D. Hovemeyer, W. Pugh, F. Emad, J.K. Hollingsworth, and N. Padua-Perez. Experiences with marmoset: Designing and using an advanced submission and testing system for programming courses. *Proceedings of the 11th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education, ITICSE 2006*, pages 13–17, 2006. URL <https://doi.org/10.1145/1140124.1140131>.
- [57] J. Sweller. Cognitive load during problem solving: Effects on learning. *Cognitive Science*, page 257–285, 1988. URL https://doi.org/10.1207/s15516709cog1202_4.
- [58] P. Van Der Beek. 1,4 miljoen voor aanpak lerarentekort informatica. <https://www.computable.nl/artikel/nieuws/onderwijs/5720597/250449/14-miljoen-voor-aanpak-lerarentekort-informatica.html>. Accessed: 2020-01-18.
- [59] A. Vihavainen, T. Vikberg, M. Luukkainen, and M. Pärtel. Scaffolding students’ learning using test my code. *Proceedings of the 18th ACM conference on Innovation and Technology in Computer Science Education*, pages 117–122, 2013. URL <https://doi.org/10.1145/2462476.2462501>.

-
- [60] D. Vitel, B.A.T.M. Golam, and A. Gaspar. Lessons learned from available parsons puzzles software. *Proceedings of ASEE'S 126th Annual Conference*, 2019.
- [61] I. Zeemeijer. Leraren slaan alarm: scholen stoppen met informatica in de bovenbouw. <https://fd.nl/economie-politiek/1287621/leraren-slaan-alarm-scholen-stoppen-met-informatica-in-de-bovenbouw>. Accessed: 2020-01-18.