# Investigating the Relationship Between Programming and Natural Languages Within the Primm Framework

Alex Parry
King's College London
London, UK
alex.parry@kcl.ac.uk

## ABSTRACT

This paper investigates the relationship between learning a programming language and learning a natural language, and how this may influence the way in which text-based programming is taught in schools. To do this, I used action research to conduct an experimental study in a school in England with 60 students aged 12-13 years old. A control group was taught with the exemplar materials developed for the initial study of PRIMM (Predict, Run, Investigate, Modify, Make). For the experimental group, the exemplar PRIMM materials were adapted to include additional activities used to support reading and writing in natural language education. The baseline test found no significant difference between the two groups prior to the intervention, whilst the experimental group achieved a significantly higher score than the control group in the posttest. This indicates that the inclusion of additional language-focused exercises made a positive impact on students' learning. A focus group was also carried out to explore students' views of learning a text-based programming language compared to a natural language. Analysis of these comments suggests that a positive view of foreign or native languages may benefit students' belief in their ability to learn a text-based programming language, whilst negative preconceptions can discourage students.

## CCS CONCEPTS

• Social and professional topics → Computing education

## KEYWORDS

Computer programming, natural language education, K-12 education, PRIMM, reading and writing code

**ACM Reference format:**

Alex Parry. 2020. Investigating the relationship between programming and natural languages within the PRIMM framework. In *Proceedings in the 15th Workshop in Primary and Secondary Computing Education (WiPSCE '20), October 28–30, 2020, Virtual Event, Germany.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3421590.3421592


WiPSCE '20, October 28–30, 2020, Virtual Event, Germany
 ACM ISBN 978-1-4503-8759-0/20/10…$15.00
https://doi.org/10.1145/3421590.3421592

## 1 INTRODUCTION

The role of language in the field of computer science education is seen by some researchers as a particularly important aspect that is often overlooked when designing and implementing introductory programming courses [38, 51, 52]. An area that has gained some traction recently is to consider an introductory programming course less as a pure problem solving domain and more as a language course [51, 52]. One approach to planning programming lessons and activities that builds on Vygotsky's work on language is PRIMM (Predict, Run, Investigate, Modify, Make) [57]. The PRIMM framework encourages the use of language and dialogue to facilitate learners' understanding of programming concepts.

This empirical study further explores how stratagems used in natural language can support novices with learning a text-based programming language. To investigate this, I adapted the exemplar materials from the initial large-scale study of PRIMM [56] to include additional learning tools found in natural language education. An experimental group was taught with the modified activities and the original exemplar materials was used to teach the control group. This meant that the impact on learning could be compared through a baseline test and posttest. Another aim of this study was to gain an insight into how students view learning a programming language compared to a natural language, and how these perceptions can influence their learning.

## 2 BACKGROUND

Learning a programming language has historically been likened to problem solving, and less as a language course [38, 51, 52]. However, there are quite a few areas of overlap between how first and second natural languages are taught in schools and introductory programming courses. For instance, one of the most widely supported teaching strategies in the field of natural languages is the read before you write approach [3] since it allows learners to focus on the meaning of words before applying their knowledge [25]. Similarly, Lister [23, 38, 65] and his colleagues have spent years advocating for students to first practice reading code and tracing the execution of a program before attempting to write code. The validity of this approach is highlighted in a number of empirical studies: some report that novice programmers find writing code more challenging than reading code [14, 53, 66] and others strongly indicate that students find writing code harder than tracing code [39, 60]. Moreover, most researchers acknowledge that it is not the syntax and semantics

that students struggle with the most but actually how these can be put together to build a program [54].

That begs the question: if some of the challenges are similar when learning a programming language and a natural language, can teaching tools be applied effectively between these domains? Research into learning first and second languages emphasises the importance of building a strong foundation of vocabulary in reading [26], speaking, [29] and writing [24, 34]. Learners acquisition of vocabulary is seen as the greatest problem that novices face when learning a language and should be a major focus of teaching [21]. In natural language education, instructors often use *receptive* tasks to develop vocabulary knowledge passively, such as learning the meaning of a word, though are less likely to incorporate *productive* tasks to actively apply this knowledge, such as writing tasks [70]. Most vocabulary is said to be learnt through receptive listening or reading exercises [27, 45], which allows students to recognise a word though they may find it difficult to use it.

Some types of receptive and productive activities that are often used in schools are matching exercises, colour coded words, and cloze exercises [48]. *Matching exercises* are a tool widely implemented by teachers when learning the meaning and form of vocabulary in native and foreign languages [19, 46]. The purpose of this method is to match a word with its definition, and this is supposed to be particularly good for learning and reviewing the lexical semantics of vocabulary [12]. Another receptive tool that is often used with children is *colour coding words* based on their grammatical class [55]. The use of colours and number of categories can be applied as the instructor chooses (e.g. nouns are blue, adjectives are red, prepositions are green). Using the same colour coding system throughout the teaching material is said to help learners make connections between words, identify patterns, pay more attention to the function of words, and ultimately remember grammatical rules [4].

A type of activity that focuses more on the productive application of vocabulary are *cloze exercises* [70], which are a type of fill in the gap activity where certain words are removed and then the learner has to work out what is missing [64]. There are systematic techniques that can be used to delete the words in a piece of text; one type is known as *rational deletion* and is proposed to be more useful than random or pseudo-random methods as it is designed to target certain concepts or keywords [1]. However, it is suggested that the use of cloze exercises as a test are limited as they are "not suitable tests of higher-order language skills, but can provide a measure of lower-order core proficiency" [1], and therefore cannot be seen as an accurate assessment of skill in a language. In the field of computer science education, some researchers have promoted the use of fill in the gap exercises as a way to overcome the difficulty novice programmers have with applying existing knowledge to new contexts [35]. The code provided is usually familiar to learners whilst the removed code is the concept they are practicing, thus helping bridge the gap between what they know and what they are learning [44]. An activity that shares traits with cloze exercises and has more recently been applied to programming is *fading worked examples*, which are supposed to support students from reading and tracing code to writing code [20]. This sequence of tasks starts by providing students with a fully complete program, then each subsequent task removes the number of worked out steps until the students are given problems with no worked steps at all.

The effectiveness of the aforementioned types of activities have been reported in many fields, nonetheless a criticism of these tools is that they are a "superficial or passive use of the vocabulary, especially when compared to writing original sentences" [19]. Other researchers disagree, arguing that there is no conclusive evidence that learners are processing in much more depth when completing writing tasks compared to other types of activities [32]. Nevertheless, it is important that learners are comfortable with their basic vocabulary before applying it in new ways.

As beneficial as these teaching tools may be, it is imperative that the content is accessible to learners. A recent framework developed for structuring programming lessons is PRIMM, which stands for Predict-Run-Investigate-Modify-Make [58]. The PRIMM approach aims to counter the known problems that novices encounter as they attempt to write programs before being able to read them, which echoes the 'read before write' strategy often voiced in natural language education [3, 25]. PRIMM builds on several well-established strategies, most notably: Use-Modify-Create [33], the Abstraction Transition Taxonomy [13], and reading and tracing code before writing [38]. PRIMM takes a sociocultural perspective on learning, regarding language as a crucial tool for mediating knowledge from the social level to the cognitive level [16, 40]. Some of the stages of PRIMM are meant to promote discussions between students so that they can practice explaining programs in English, CS speak and code [13]. For instance, at the beginning of a lesson students can be provided with a starter program and are encouraged to work in pairs or groups to *predict* what they think the program will do when it is executed, then once they *run* the program they can *investigate* their predictions together. This is designed to allow students to "accommodate or assimilate their understanding with language and vocabulary becoming the oil to facilitate the transitions" [57].

The initial large-scale study of PRIMM in 2018 involved nearly 500 students aged 11-14 years old from schools in England and the findings were promising. Students in the experimental group who were taught using the exemplar PRIMM materials for ten lessons or more made significantly better progress than those in the control group when the posttest results were compared [57]. Furthermore, the teachers interviewed that delivered PRIMM to their students reported that it was a useful framework for structuring the content of the lessons as it "offers routine, accessibility for lower ability students, and an enriched understanding of core programming constructs" [57]. However, some teachers observed students spending nearly all their time in the early stages of *predict*, *run*, and *investigate* and less in the code writing stages *modify* and *make*. This aligns with the majority of other programming research which has found that students struggle the most with writing code [38, 39, 53, 66].

Though there have been some encouraging steps towards programming strategies that promote the use of language, as well as teaching tools that have been successfully adopted from natural language education, there is still quite a lot of scope to further investigate the connections between these domains [51, 52]. This study aims to provide an insight into how introductory programming courses could further benefit from the extensive research into natural language education, and some of the challenges that may be encountered.

## 3 RESEARCH DESIGN

This action research used an experimental design to investigate whether applying methods for improving natural languages can benefit students' learning of a programming language. To do this, I adapted the exemplar teaching materials created for the initial PRIMM study [57]. This involved incorporating activities often used for learning first and second languages, such as *colour coding words* and *cloze exercises*, as well as programming exercises, for instance *fading worked examples*. The study took place in a secondary school in England in 2019, where an experimental group was taught with the modified lesson materials and the control group was taught using the original exemplar materials. This allowed me to measure and compare the progress of students in both groups using a baseline test and a posttest [22]. Furthermore, I wanted to explore students' perceptions of learning a programming language compared to learning a natural language, along with their experiences of the modified materials. To conduct this part of the study, I carried out a focus group with students from the experimental group just after the intervention had finished. Specifically, I investigated the following two research questions:

- **RQ1**: What are the perceptions of students regarding the similarities between learning a programming language and a natural language?
- **RQ2**: How has the additional natural language activities influenced students' learning?

### 3.1 Research setting

The intervention took place during normal lesson time at an all-girls state secondary school located in London. The two groups of students used in the experimental and control groups were in Year 8, aged 12-13 years old, and set in mixed ability computing classes. Students had not studied a text-based language within the school curriculum but did have experience with developing algorithms using interactive flowcharts.

Both groups contained 30 students and the intervention was carried out over a sequence of six lessons, not including the baseline test nor the posttest. The baseline test was given to the students one lesson before the intervention began to complete during lesson time, as was the posttest one lesson after the intervention had ended. The focus group was conducted one week after the posttest at the end of the school day.

### 3.2 Teaching materials

The lessons and activities for the intervention was based on the exemplar PRIMM materials that are freely available to edit [56], whilst the unmodified exemplar materials were used to teach the control group. Both the experimental and the control group were taught six lessons of programming which covered the basics of sequence, selection, and iteration in Python. The main changes I made to the exemplar PRIMM materials were adding more supportive materials in the *investigate* stage to assist students with understanding new concepts, as well as adapting the *modify* exercises to further scaffold the code writing tasks. The reason for focusing on these two stages was to help students more so with bridging the gap between reading, tracing and writing code.

For each new programming construct that the students were introduced to, I included a colour coded program statement to the start of the *investigate* stage. This conveyed the building blocks and the grammatical structure that the command needed to follow [55]. The colours were based on the syntax highlighting colours used in the Python programming environment IDLE, which students used to run and edit programs in all of the lessons. An example of a colour coded `if-else` statement is shown in Figure 1. Furthermore, I included a correct and an incorrect example of the code in Python that again matched the colours to visually aid students with recognising the programming commands and grammatical rules [4]. The incorrect examples were based on common misconceptions as reported in the extensive work of Sorva [62], as shown in Figure 2.

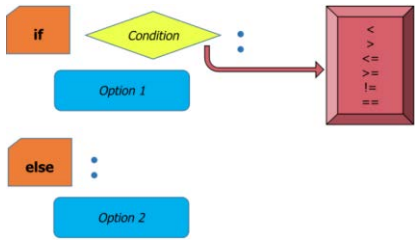


**Figure 1: A colour coded if-else statement.**

```
Correct
age = int(input("How old are you? "))
if age < 18:
    print("You can't vote yet")
else:
    print("You are able to vote")

Incorrect
if < 18:
    print("You can't vote yet")
else >= 18:
    print("You are able to vote")
age = int(input("How old are you? "))
```

**Figure 2: A correct and incorrect example of an if-else statement.**

As previously highlighted in this paper, cloze exercises and fading worked examples share some characteristics as they both aim to target concepts or keywords by removing certain parts of

an artefact [63, 64]. I decided to combine the two tools in what I refer to as *fading cloze exercises*. This was used as a scaffolding method in the *modify* stage to support students with writing code. The first task was provided with a program statement showing a full worked example and each subsequent task removed more of the code so that students could focus on what was essential [59], an example of which can be seen in Figure 3. As the scaffolding was reduced gradually, students would hopefully gain more confidence in writing code before they moved on to the final *make* stage where there was no scaffolding provided to promote independent learning [2].

| Task | Expression |
|---|---|
| Change the program so that it asks the user (the person using your program) what their name is at the beginning of the program. | `name = input("What is your name?")` |
| Change the program so that it lets you type in your answer on the same line as the question. | `answer = _____("Do you like programming" + name + "?")` |
| Ask the user what they had for breakfast and output a suitable response. | `breakfast = _____("What did you have for breakfast?")`<br><br>`print("I really like " _____ + "too!")` |
| Ask the user what their favourite colour is and output a suitable response. | `colour _____(_____)`<br><br>`_____(_____ " is great!")` |

**Figure 3: Fading cloze exercises to support code writing.**

The use of matching exercises was another type of exercise included in the *investigate* stage with the goal of getting students to think about the definition of a key term and match it with the piece of code it corresponded to [19]. Students were told that each definition matches with one program statement and vice versa. Some of the definitions I included had a nuanced meaning to try and get students to consider the subtle differences between certain program statements. For example, in Figure 4 two of the definitions were "Gets the user input from the keyboard" and "Displays the message that is between the speech marks and gets the user input from the keyboard". The first definition could be applied to both programming statements which contained the `input` function, yet the second definition only described the `input` statement that asked the user what type of pet they have. Furthermore, I also included the type of command the definition was related to (e.g. function or operator) to try and improve students' programming vocabulary and facilitate CS speak [13].

| Meaning | Example |
|---|---|
| Function<br>Displays the message that is between the speech marks on the screen. | print() |
| Function<br>Creates a new blank line on the screen. | "My age is " + age |
| Operator<br>Joins two pieces of text together or text with a text variable (string). | choice = input() |
| Variable<br>Something you can assign a value to and then change it at other times in the program. | print("Hi there!") |
| Function<br>Gets the user input from the keyboard. | pet = input("What type of pet do you have?") |
| Function<br>Displays the message that is between the speech marks and gets the user input from the keyboard. | year = 2020 |

**Figure 4: A matching exercise based on variables, print statements and inputs.**

## 3.3 Experimental design

The use of an experimental design can be especially beneficial when evaluating how effective a programme was and the impact it had on the participants. This is due to the emphasis on comparative data for interpreting the findings of a study which can "increase our confidence that observed outcomes are the result of a given program or innovation instead of a function of extraneous variables or events" [22]. For this study, I taught the experimental group using the adapted exemplar PRIMM materials and the control group with the original materials before comparing the results of the two groups.

Due to the content being delivered during normal lesson time, I employed a quasi-experimental design as it was impractical to randomise the students between the control and experimental groups. This design is said to be appropriate for "groups [that] constitute naturally assembled collectives such as classrooms, as similar as availability permits" [8]. It is also one of the most common types of experimental design used in education research when it is unfeasible to randomly assign participants to each of the groups [22]. However, there are important considerations to be made regarding the reliability and validity of implementing such an approach. Even though both groups were mixed ability classes, the existing programming knowledge individuals had would vary and this would likely result in an average class level that was not the same. To counteract these differences, the students in both groups were given a baseline test to complete before the study began to measure their initial programming ability. Then at the end of the study, students took a posttest which was compared with the baseline test to more accurately gauge the progress of both groups. This is a specific type of quasi-experimental design called the nonequivalent group, pretest-posttest design [71].

## 3.4 Baseline test and posttest

The baseline test and posttest were two separate tests that aimed to test students' comprehension of programming concepts and how to apply them. The questions were designed around the

receptive and productive test types proposed by Webb [70] for assessing vocabulary. However, it is argued that some questions can involve both receptive and productive skills [50], for example tracing a program and then writing the output. It should therefore be noted that these question types may not be strictly assessing just receptive or productive knowledge but are more inclined to be testing one process over another. Moreover, I wanted to test both bottom-up and top-down program comprehension. To promote top-down comprehension in some of the questions, I used identifiers for variables that students may have been familiar with as *beacons* [61], such as the name `sum` for a variable that stores the sum of values. In other questions, I obfuscated the identifier names so that it forced students to employ bottom-up comprehension in order to work out how the program would function [11], such as using the variable `result` which could store all sorts of information.

Based on the test characteristics recommended by the extensive review of empirical computing studies from Margulieux [41], I formatted the baseline test as follows: six multiple choice questions, one question on tracing code, and one question explaining what the code does. Each question was worth one point and the total number of points available was eight. Furthermore, the types of knowledge these questions were assessing was based on the receptive and productive tests recommended by Webb [70]: the multiple choice questions tested receptive knowledge of grammatical functions, the tracing question tested productive knowledge of meaning and form, and the code explain question tested productive knowledge of grammatical functions. As the students had not yet been taught the programming language Python in school, it was more relevant for the baseline test to assess their lexical understanding of general programming concepts rather than syntactic. Similarly, receptive tests were more appropriate to use than productive tests as the students had not coded in a text-based programming language before the study. The reason one question was included on tracing and one on writing was to account for any students who had programmed in Python or another text-based language before the study had begun. An example of a baseline test question assessing receptive knowledge of meaning and form is shown in Figure 5.

***Question 1***

Select the correct meaning of the programming term "iteration"

a) A decision based on a condition
b) A named block of code
c) A series of instructions that are performed one after the other
d) A block of code that repeats a set number of times

**Figure 5: A baseline question assessing receptive knowledge of meaning and form.**

The posttest was based on the concepts covered during the intervention period: sequencing, selection and iteration. For this test I included two multiple choice questions, two code tracing questions, two questions writing the term for a definition, and two questions writing a program statement using the term provided. Again, each question counted as one point and students could score a maximum number of eight points. As the intervention was supposed to support students so they could spend more time in the final code writing stage, the posttest was more focussed on assessing the productive skills of students. This is why six out of eight questions were testing productive skills as opposed to two out of eight in the baseline test. Furthermore, the posttest included two questions where the students had to read the definition of a programming construct and write the term associated with it, which was not a question type included in the baseline test. The reason for this inclusion was to assess how well they understood the purpose of the constructs represented. An example of a posttest question assessing productive knowledge of meaning and form is shown in Figure 6 and one assessing productive knowledge of grammatical functions is shown in Figure 7. The validity of the baseline test and the posttest was ensured by asking other computer science teachers and researchers to review the questions and confirm that the knowledge being measured was appropriate for the proposed concepts and course level [43].

***Question 3***

If the value of the choice variable is 13,
what would the outcome of the program be?

**Answer:** _____

```
1 choice = int(input("Enter value: "))
2
3 if choice < 2:
4     print("Free")
5 elif choice < 13:
6     print("£5.00")
7 else:
8     print("£11.50")
```

**Figure 6: A posttest question assessing productive knowledge of meaning and form.**

***Question 7***

Write a one-line program statement using the term "input"

**Answer:** _____

**Figure 7: A posttest question assessing productive knowledge of grammatical functions.**

## 3.5 Focus group

Focus groups can be beneficial as a group interview technique as they encourage interaction with other participants, helping to stimulate discussions that are in-depth whilst also relieving the anxiety of being interviewed which may be more likely with younger participants [9]. It also allows a researcher to produce a descriptive account of the views of participants and any themes that may emerge, though it is important to realise that this data may not fully reflect the actual events that occurred accurately or completely [73]. Therefore, it was important to check any issues identified and to clarify or confirm any comments made during the focus group in real time [15].

The purpose of the focus group was to understand how students perceived learning a programming language compared with a natural language, and to gather feedback on the intervention materials. As such, I chose students from the experimental group to participate in the focus group one week after the intervention had finished. When choosing the students most suitable for the focus group, it was important to understand the attitudes of a broad range of students. To achieve this and help mitigate against potential bias, I applied a "maximum variation sample" [42] by selecting students with a mixture of abilities based on their current grade in computing. The five students who took part in the focus group were a broad sample from the intervention; two higher ability, one medium, and two lower ability.

## 3.6 Ethics

The participants of the study were all students from my computing classes so it was imperative to mitigate against pressure they might have felt to participate in the data collection aspects of my research. Moreover, as all the students were of a vulnerable age between 12-13 years old it was important to ensure students were comfortable with providing consent and that consent was maintained throughout the study [47]. In my role as moderator during the focus group, I took a reflexive approach by ensuring that I did not direct students' comments in terms of correctness to promote agency and choice in their answers [49]. I also reaffirmed that there were no right answers, and this was an open discussion that they could join in with as much as they like, whilst being alert to emotional responses for students and any potential signs of distress [69].

## 3.7 Analysis

To find out whether the intervention materials had an impact on students' learning in comparison to the unmodified exemplar PRIMM materials, I analysed both groups' results from the baseline test and the posttest. Furthermore, I examined the qualitative data from the transcription of the focus group by coding the comments and then categorising these into emerging themes, which was an iterative process until new or modified themes ceased to transpire.

# 4 RESULTS

The experimental group contained 30 students and the control group also consisted of 30 students. All students completed both the baseline test and the posttest. The Mann-Whitney U test was utilised for the analysis of the scores since the data was being compared across two independent groups and it was not normally distributed. For the focus group, 5 students from the experimental group were chosen with a range of abilities based on their computing grade.

## 4.1 Baseline test and posttest

The descriptive statistics of the baseline test can be viewed in Table 1, which shows that the control group had a mean and median score of 3.037 and 3.0 respectively, whilst the experimental group scored 3.6552 and 4.0. To examine whether the results showed any significant difference between the control group and the experimental group before the intervention began, the Mann-Whitney U test was used on the data with the null hypothesis: "there is no significant difference between the scores of the experimental group and the control group in the baseline test". Table 2 displays the results, revealing that the p value (0.191) is greater than the common alpha risk value of 0.05. Therefore, the performance of students on the baseline test was not significantly different between the two groups so the null hypothesis could not be rejected (U = 313.5, Z = -1.307, p = 0.191).

**Table 1: Descriptive statistics of the control and experimental groups.**

| | Control | | Experimental | |
|---|---|---|---|---|
| | Baseline test | Posttest | Baseline test | Posttest |
| Mean | 3.0370 | 3.7037 | 3.6552 | 5.1379 |
| Median | 3.0000 | 4.0000 | 4.0000 | 5.0000 |
| Variance | 2.037 | 2.986 | 2.448 | 1.623 |
| Std. Deviation | 1.42725 | 1.72793 | 1.56470 | 1.27403 |
| Skewness | -.156 (se = .448) | .110 (se = .448) | .321 (se = .434) | -.387 (se = .434) |
| Kurtosis | -.365 (se = .872) | -.340 (se = .872) | -.818 (se = .845) | .553 (se = .845) |

**Table 2: Mann-Whitney U test comparing the baseline test and posttest of the two groups.**

| | Baseline test | Posttest |
|---|---|---|
| Mann-Whitney U | 313.500 | 198.000 |
| Wilcoxon W | 691.500 | 576.000 |
| Z | -1.307 | -3.237 |
| Asymp. Sig. (2-tailed) | .191 | .001 |

The data from the posttests was used to analyse if there were any differences between students' learning in the intervention lessons compared to the control group once all of the lessons had ended. The statistics from Table 1 shows that the control group had a mean and median score of 3.7037 and 4.0 respectively, whilst the experimental group scored 5.1379 and 5.0. Table 2 shows the results of the Mann-Whitney U test against the null hypothesis: "there is no significant difference between the scores of the experimental group and the control group in the posttest". Considering that the p value (0.001) is below the alpha risk value of 0.05, there is a statistically significant difference between the two groups and the null hypothesis can be rejected. Judging by the results of the Mann-Whitney U test, the experimental group achieved a significantly higher score on the posttest than the control group (U = 198, Z = -3.237, p = 0.001). Furthermore, the effect size (r) can be calculated using the formula $r = Z / \sqrt{N}$ where N is the total number of students from both groups. Using this formula gives the value of r as $r = 3.237 / \sqrt{60} =$

0.41789. This suggests that 42% of the variance between the experimental and the control groups was due to the intervention, which represents a 'medium' effect size.

To summarise, the scores of the two groups when using the Mann-Whitney U test found no statistically significant difference between the baseline test results before the intervention began. However, the Mann-Whitney U test demonstrated that there was a significant difference between the posttest results in favour of the experimental group once the lessons had finished. This indicates that the intervention made a positive impact on students' learning.

## 4.2 Focus group

To interpret the data gathered from the focus group discussion, I used thematic analysis to find themes or patterns within the data that could illuminate the research questions [6]. To discover prevalent themes I used open coding, one of the main tools of the grounded theory approach for analysing data [7]. This involves two phases of processing data: initial and focused coding. During the initial coding phase, I assigned a label to each section of data that categorised, summarised, and took into account each piece of data [10]. These initial codes helped me to identify useful processes, actions and ideas. Following that, I applied focused coding to allow the more frequent and significant codes to surface from the data by deciding "which initial codes make the most analytic sense to categorize the data incisively and completely" [10]. From this categorisation process, I extracted four themes: programming and natural language similarities; range and scope of vocabulary; usefulness of the materials; and problems with syntax. Examples of each are shown in Table 3.

**Table 3: Themes and comments from the focus group.**

| Theme | Example comment |
|---|---|
| Programming and natural language similarities | "I think that's why I like coding because I like learning languages." |
| Range and scope of vocabulary | "...the reason why I find programming so interesting is that I find that so many different types of commands are so versatile and just really convenient because you don't have to learn a bunch of things rather than in other languages." |
| Usefulness of the materials | "Something that was useful and helped us understand... is that fact that we have the structure of the sentences... it showed how you could go and change some of the words, so it made it easier to code and easier to understand." |
| Problems with syntax | "I feel like what part of it is confusing for me it's having all these different symbols…so you have just random letters in the middle of symbols and it's a bit weird. It's confusing." |

# 5 DISCUSSION

## 5.1 Perceptions of languages

Comments from many of the students indicated that, for the most part, they associated learning the programming language Python with learning a natural language. This led to both positive and negative connotations in relation to this perceived similarity. Some students declared a love of learning natural languages, whether foreign or native, and also spoke about how much they enjoyed coding. The professed relationship between both types of languages was also expressed as a hindrance by one student who spoke about her struggles with learning programming and natural languages. These views conform with Dijkstra's claim that "an exceptionally good mastery of one's native tongue is the most vital asset of a competent programmer" [17].

Though most students did make some type of connection between learning programming and learning foreign or native languages, these were not always classified as having the same level of difficulty. One area of frustration raised by several students was the role of syntax in programming. Although this is also an important aspect of natural languages, faulty syntax can have a more terminal outcome when writing code since an incorrectly structured statement may stop the whole program from working. This relates to the claim by Vygotsky [68] that written language is more difficult than spoken due to the formal rules that need to be followed, and that the gap widens when encountering a challenging task such as programming [28]. Moreover, this may explain why the students who reported syntax as a problematic area often struggled to write error free code, and yet this was not an opinion voiced by the higher ability students. Similar traits were found in the empirical studies reported on by Lister [37], where a large range in the quality of written code was observed and the ability for students to write functioning programs was linked to their capacity to abstract [18]. Therefore, syntax may be less of an issue for students if they are more able to abstract what is necessary and what is not when solving a programming problem.

A significant finding from my study was around students' perceptions on the amount and range of vocabulary between programming and natural languages. Some of the students preferred learning a programming language due to it having a limited vocabulary with commands that can be applied in a huge range of ways (e.g. `if`, `while`, `for`) as opposed to a natural language which has a much wider, yet less adaptable vocabulary. This is counter to some of the research into novice programmers, which argues that many students only have superficial knowledge of programming that is often context specific [31]. However, the students who made these comments in the focus group may have

had a more complete notional machine [5], thereby being better able to apply concepts to different problems and scenarios.

## 5.2 Impact on learning

Data analysed from the experimental group and the control group found a significant difference between the posttest results, which revealed that students who were taught with the modified intervention materials performed better in the final assessment. This suggests that the inclusion of language teaching tools was beneficial to students' learning, which was also expressed in more detail by some of the students in the focus group.

One of the instructional tools that was mentioned by students as being particularly helpful was the inclusion of colour coded program statements to understand the grammatical structure of a new concept [55, 72]. The inclusion of a correct and incorrect example alongside the colour coded statements was said by students to support with recognising the order of the commands and symbols, as well as highlighting the parts of the code that could be changed and what must stay the same. This indicates that the consistent use of colour codes related to the functional role of the program commands assisted students with identifying patterns and remembering the grammatical rules [4].

A tool that students cited as being beneficial with writing code was the combination of cloze exercises [64] and fading worked examples [63]. This blended tool was used to incrementally reduce the amount of code provided in the *modify* stages, which many students reported as helping them to initially understand how to structure a program when writing their own code. These scaffolded exercises can therefore be seen to bridge the gap between what students knew and what they were learning [44]. Furthermore, gradually removing the scaffolding of the tasks was said by one student to support her with moving on from the *modify* stage to the final *make* stage, which counteracts one of the limitations raised by teachers during the initial PRIMM study [57].

A part of the intervention materials that were reported not to provide much support was when students were constructing larger programs, especially in the later lessons. One student said that the materials did not show her how to structure a program when using multiple functions, which meant she was unsure where to define a new function and where the functions should be executed. This is backed up by much of the research into novice programmers that generally agrees it is not the syntax and semantics that students struggle with the most but how to combine these to build a program [54]. The problem with where to place functions in more complex programs may also be linked to an inability to accurately trace the code, as it is suggested that one of the constructs novices find particularly challenging is understanding the order that code is executed when encountering functions [67].

Another improvement to the materials that was suggested by a student was to include comments in the code to help understand what each line of code did within the programs given to students at the start of the lesson. This could have further supported students with reading and tracing code before they proceeded to the writing activities since it would have allowed them to focus more on the meaning of commands before applying their knowledge [25]. It would also further promote the read before write approach which is recommended by many researchers in programming education [36, 39, 60] and natural language education [3, 21, 25].

# 6 CONCLUSIONS

The overall purpose of this action research was to explore the relationship between learning a programming language and a first or second natural language. One aim was to investigate how incorporating additional natural language activities in the exemplar PRIMM materials would impact students' learning. Analysis of the posttest results showed that the experimental group who were taught with the modified exercises made significantly better progress than the students taught with the original exemplar materials after six lessons. Some of the students commented on these additional activities in the focus group. In particular, students found the *colour coded program statements* helpful for understanding the grammatical structure of a new concept in the *investigate* stage. They also referenced the benefits of the *fading cloze exercises* in the *modify* stage, finding this a supportive tool for writing code before moving to the final *make* stage. This gradual removal of scaffolding was seen to promote independent learning as the students gained confidence with writing code [2].

The other aim of the study was to explore how students perceive learning a programming language compared to a natural language. The analysis of the focus group indicates that a positive perspective of foreign or native languages may benefit students' belief in their ability to learn a programming language. Conversely, if students consider learning natural languages to be challenging then they could be discouraged when they are introduced to a text-based programming language. These findings correspond with the view that to be good at programming you must be highly skilled in your native language [17]. Another insight from the qualitative data was that most students recognise a connection with programming and natural languages, though they were not always considered to be similarly demanding. An area of programming that some students found particularly problematic was using the correct syntax when writing code, which is a common area of difficulty for novice programmers [30]. Conversely, some students expressed that a preference of programming over natural languages due to the relatively small vocabulary and versatility of commands, which is not something I had found reported in other computer science research. It is also counter towards some claims that novices often only have context specific knowledge [31].

# 7 LIMITATIONS AND FUTURE RESEARCH

There were some limitations from this study that should be noted as well as recommendations for future research. The method used was action research as I was also the teacher in both groups. Given the only medium effect size, it is difficult to know whether

some of this effect was due to unconscious bias, such as increased enthusiasm from myself during the experimental group lessons. It was also only a single study based in one all-girls school. To improve the validity of this research, the lessons could be led by other teachers in multiple schools with both mixed gender and single-sex classes.

Another limitation was that the experimental group were provided with a broader variety of tasks, which increased the chances of learning, and therefore a positive outcome. In a follow up study, the control group could be provided with another type of exercise that is not related to natural language education (e.g. Parson's problems) to counter this effect. Moreover, the colour coding was based on the syntax highlighting colours of the IDE. This most probably had a beneficial effect due to enhancing the recognition of these constructs later in the IDE, and is not based on natural language acquisition per se.

A further area of research that could gain a deeper insight into how to support novice programmers when teaching programming more as a language course is the effect that a small, yet versatile vocabulary has on the way students understand and apply their knowledge when compared to a natural language.

## REFERENCES


[1] J.C. Alderson. 1979. The Cloze Procedure and Proficiency in English as a Foreign Language. *TESOL Quarterly*. 13 (2), 219–227.

[2] N. Anderson and T. Gegg-Harrison. 2013. Learning Computer Science in the "Comfort Zone of Proximal Development". In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*. SIGCSE '13. 2013 Denver, Colorado, USA: ACM. pp. 495–500.

[3] N. Atwell. 1987. *In the Middle: Writing, Reading, and Learning with Adolescents*. Portsmouth, New Hampshire: Heinemann Educational Books.

[4] A. Benjamin, C. Jago, M. Kolln, H.R. Noden, N. Patterson, J. Penha, M.W. Smith, J.D. Wilhelm, C. Weaver, and R.S. Wheeler. 2006. Teacher to Teacher: What Is Your Most Compelling Reason for Teaching Grammar? *The English Journal*. 95 (5), 18–21.

[5] B. du Boulay. 1986. Some Difficulties of Learning to Program. *Journal of Educational Computing Research*. 2 (1), 57–73.

[6] V. Braun and V. Clarke. 2006. Using thematic analysis in psychology. *Qualitative Research in Psychology*. 3 (2), 77–101.

[7] A. Bryman. 2012. *Social research methods*. Oxford, UK: Oxford University Press.

[8] D.T. Campbell and J.C. Stanley. 1963. *Experimental and quasi-experimental designs for research*. Boston: Houghton Mifflin Company.

[9] A. Carey. 1994. The group effect in focus groups: planning, implementing, and interpreting focus group research. *Open Journal of Nursing*. 3 (6), 225–241.

[10] K. Charmaz. 2006. *Constructing grounded theory*. Thousand Oaks, CA, USA: Sage Publications, Inc.

[11] S. Chow, Y. Gu, H. Johnson, and V. Zakharov. 2001. 'An Approach to the Obfuscation of Control-Flow of Sequential Computer Programs', in George I. Davida & Yair Frankel (eds.) *Information Security*. Lecture Notes in Computer Science. 2001 Springer Berlin Heidelberg. pp. 144–155.

[12] R.C. Culyer. 1978. Guidelines for Skill Development: Vocabulary. *The Reading Teacher*. 32 (3), 316–322.

[13] Q. Cutts, S. Esper, M. Fecho, S.R. Foster, and B. Simon. 2012. The Abstraction Transition Taxonomy: Developing Desired Learning Outcomes Through the Lens of Situated Cognition. In *Proceedings of the Ninth Annual International Conference on International Computing Education Research*. ICER '12. 2012 Auckland, New Zealand: ACM. pp. 63–70.

[14] P. Denny, A. Luxton-Reilly, and B. Simon. 2008. Evaluating a New Exam Question: Parsons Problems. In *Proceedings of the Fourth International Workshop on Computing Education Research*. ICER '08. 2008 New York, NY, USA: ACM. pp. 113–124.

[15] N.K. Denzin and Y.S. Lincoln. 1994. *Handbook of qualitative research*. Thousand Oaks, CA, USA: Sage Publications, Inc.

[16] I. Diethelm and J. Goschler. 2015. Questions on Spoken Language and Terminology for Teaching Computer Science. In *Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education*. ITiCSE '15. 2015 Vilnius, Lithuania: ACM. pp. 21–26.

[17] E.W. Dijkstra. 1975. How do we tell truths that might hurt? *ACM SIGPLAN Notices*. 17 (5), 13–15.

[18] E.W. Dijkstra. 1972. The Humble Programmer. *Communications of the ACM*. 15 (10), 859–866.

[19] K.S. Folse. 2006. The Effect of Type of Written Exercise on L2 Vocabulary Retention. *TESOL Quarterly*. 40 (2), 273–293.

[20] S. Gray, C. St. Clair, and R. James. 2007. Suggestions for Graduated Exposure to Programming Concepts Using Fading Worked Examples. In *Proceedings of the Third International Workshop on Computing Education Research*. ICER '07. 2007 Atlanta, Georgia, USA: ACM. pp. 99–110.

[21] D. Green and P. Meara. 1995. CALL and vocabulary teaching. *Computer Assisted Language Learning*. 8 (2), 97–101.

[22] B. Gribbons and J. Herman. 1997. True and Quasi-Experimental Designs. *Practical Assessment, Research & Evaluation*. 5 (14).

[23] J. Hidalgo-Céspedes, G. Marín-Raventós, and V. Lara-Villagrán. 2016. *Understanding Notional Machines through Traditional Teaching with Conceptual Contraposition and Program Memory Tracing*. CLEI Electronic Journal. 19 (2), 3–3.

[24] E. Hinkel. 2001. Giving Personal Examples and Telling Stories in Academic Essays. *Issues in Applied Linguistics*. 12 (2), 149–170.

[25] A. Hirvela. 2004. *Connecting Reading & Writing in Second Language Writing Instruction*. University of Michigan Press.

[26] T. Huckin and J. Bloch. 1993. 'Strategies for inferring word meaning in context: A cognitive model', in *Second language reading and vocabulary acquisition*. Norwood, NJ, USA: Ablex Publishing. pp. 153–178.

[27] J.R. Jenkins, M.L. Stein, and K. Wysocki. 1984. Learning Vocabulary Through Reading. *American Educational Research Journal*. 21 (4), 767–787.

[28] T. Jenkins. 2002. On the difficulty of learning to program. In *Proceedings of the 3rd Annual Conference of the LTSN Centre for Information and Computer Sciences*. 2002 Loughborough University, UK. pp. 53–58.

[29] A. Joe. 1998. What Effects Do Text-based Tasks Promoting Generation Have on Incidental Vocabulary Acquisition? *Applied Linguistics*. 19 (3), 357–377.

[30] S.K. Kummerfeld and J. Kay. 2003. The neglected battle fields of syntax errors. In *ACE '03*. 2003 Adelaide, Australia: Australian Computer Society, Inc. pp. 105–111.

[31] E. Lahtinen, K. Ala-Mutka, and H Järvinen. 2005. A study of the difficulties of novice programmers. In *Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*. 14–18.

[32] B. Laufer and J.H. Hulstijn. 2001. Incidental vocabulary acquisition in a second language: the construct of Task-Induced Involvement. *Applied Linguistics*. 22 (1), 1–26.

[33] I. Lee, F. Martin, J. Denner, B. Coulter, W. Allan, J. Erickson, J. Malyn-Smith, and L. Werner. 2011. Computational Thinking for Youth in Practice. *ACM Inroads*. 2 (1), 32–37.

[34] S.H. Lee. 2003. ESL learners' vocabulary use in writing and the effects of explicit vocabulary instruction. *System*. 31 (4), 537–561.

[35] H. Lieberman. 1986. An example based environment for beginning programmers. *Instructional Science*. 14 (3), 277–292.

[36] R. Lister, E.S. Adams, and S. Fitzgerald. 2004. A Multi-national Study of Reading and Tracing Skills in Novice Programmers. In *Working Group Reports from ITiCSE on Innovation and Technology in Computer Science Education*. ITiCSE-WGR '04. 2004 New York, NY, USA: ACM. pp. 119–150.

[37] R. Lister. 2011. Concrete and Other neo-Piagetian Forms of Reasoning in the Novice Programmer. In *Proceedings of the Thirteenth Australasian Computing Education Conference - Volume 114*. ACE '11. 2011 Australian Computer Society, Inc. pp. 9–18.

[38] R. Lister, C. Fidge, and D. Teague. 2009. Further evidence of a relationship between explaining, tracing and writing skills in introductory programming. In *Proceedings of the 14th Annual ACM SIGCSE Conference on Innovation and Technology in Computer Science Education*. ITiCSE '09. 2009 New York, NY, USA: ACM. pp. 161–165.

[39] M. Lopez, J. Whalley, and P. Robbins. 2008. Relationships Between Reading, Tracing and Writing Skills in Introductory Programming. In *Proceedings of the Fourth International Workshop on Computing Education Research*. ICER '08. 2008 Sydney, Australia: ACM. pp. 101–112.

[40] P. Machanick. 2007. A social construction approach to computer science education. *Computer Science Education*. 17 (1), 1–20.

[41] L. Margulieux, T.A. Ketenchi, and A. Decker. 2019. Review of measurements used in computing education research and suggestions for increasing standardization. *Computer Science Education*. 29 (1), 1–30.

[42] M.N. Marshall. 1996. Sampling for qualitative research. *Family practice*. 13 (6), 522–526.

[43] M.E. Matore and A. Khairani. 2015. Assessing the Content Validity of IKBAR using Content Validity Ratio. *International Journal of Basic & Applied Sciences*. 9 (7), 255–257.

[44] J.J. van Merriënboer and F. Paas. 1990. Automation and Schema Acquisition in Learning Elementary Computer Programming: Implications for the Design of Practice. *Computers in Human Behavior*. 6 (3), 273–289.

[45] W.E. Nagy, R.C. Anderson, and P.A. Herman. 1987. Learning Word Meanings from Context during Normal Reading. *American Educational Research Journal*. 24 (2), 237–270.
[46] T. Odlin. 1994. *Perspectives on Pedagogical Grammar*. Cambridge, UK: Cambridge University Press.
[47] S. Olitsky and J. Weathers. 2005. Working with Students as Researchers: Ethical Issues of a Participatory Process. *Forum Qualitative Social Research*. 6 (1), 1–26.
[48] I. Ozverir, J. Herrington, and U. Osam. 2016. Design principles for authentic learning of English as a foreign language: Authentic learning of English as a foreign language. *British Journal of Educational Technology*. 47 (3), 484–493.
[49] S. Pendlebury and P. Enslin. 2001. Representation, identification and trust: towards an ethics of educational research. *Journal of the Philosophy of Education*. 35 (3), 361–370.
[50] V. Pignot-Shahov. 2012. Measuring L2 Receptive and Productive Vocabulary Knowledge. *Language Studies Working Papers*. 4 (1), 37–45.
[51] S.R. Portnoff. 2018. The introductory computer programming course is first and foremost a language course. *ACM Inroads*. 9 (2), 34–52.
[52] C.S. Prat, T.M. Madhyastha, M.J. Mottarella, and C. Kuo. 2020. Relating Natural Language Aptitude to Individual Differences in Learning Programming Languages. *Sci Rep*. 10, 3817.
[53] Y. Qian and J. Lehman. 2017. Students' Misconceptions and Other Difficulties in Introductory Programming: A Literature Review. *ACM Transactions on Computing Education (TOCE)*. 18 (1), 1–24.
[54] A. Robins, J. Rountree, and N. Rountree. 2003. Learning and Teaching Programming: A Review and Discussion. *Computer Science Education*. 13 (2), 137–172.
[55] R.I. Scott. 1968. Teaching Elementary English Grammar with Color-Coded Word-Blocks. *Elementary English*. 45 (7), 972–981.
[56] S. Sentance. 2018. PRIMM materials 2018. (August 2018). Retrieved February 25, 2019 from: https://primming.wordpress.com/2018/08/23/primm-materials-2018/
[57] S. Sentance, J. Waite, and M. Kallia. 2019. Teachers' Experiences of using PRIMM to Teach Programming in School. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*. SIGCSE '19. 2019 New York, NY, USA: ACM. pp. 476–482.
[58] S. Sentance, and J. Waite. 2017. PRIMM: Exploring Pedagogical Approaches for Teaching Text-based Programming in School. In *Proceedings of the 12th Workshop on Primary and Secondary Computing Education*. WiPSCE '17. 2017 New York, NY, USA: ACM. pp. 113–114.
[59] K. Shabani. 2016. Applications of Vygotsky's sociocultural approach for teachers' professional development. *Cogent Education*. 3 (1), 1–10.
[60] J. Sheard, B. Simon, A. Carbone, E. Thompson, R. Lister, and J. Whalley. 2008. Going SOLO to assess novice programmers. In *Proceedings of the Conference on Integrating Technology into Computer Science Education, ITiCSE*. 209–213.
[61] E. Soloway and K. Ehrlich. 1986. 'Empirical studies of programming knowledge', in Charles Rich & Richard C. Waters (eds.) *Readings in artificial intelligence and software engineering*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc. pp. 507–521.
[62] J. Sorva. 2018. Misconceptions and the Beginner Programmer. In *Computer science education: Perspectives on teaching and learning in school*. London: Bloomsbury Academic. pp. 171–186.
[63] J. Sweller. 1988. Cognitive Load During Problem Solving: Effects on Learning. *Cognitive Science*. 12257–285.
[64] W.L. Taylor. 1953. Cloze Procedure: A New Tool for Measuring Readability. *Journalism Quarterly*. 30 (4), 415–433.
[65] D. Teague and R. Lister. 2014. *Blinded by their Plight: Tracing and the Preoperational Programmer*. 53–64.
[66] D. Teague and R. Lister. 2014. Programming: Reading, Writing and Reversing. In *Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education*. ITiCSE '14. 2014 Uppsala, Sweden: ACM. pp. 285–290.
[67] V. Vainio and J. Sajaniemi. 2007. Factors in Novice Programmers' Poor Tracing Skills. In *Proceedings of the 12th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*. ITiCSE '07. 2007 New York, NY, USA: ACM. pp. 236–240.
[68] L.S. Vygotsky. 2004. Imagination and Creativity in Childhood. *Journal of Russian & East European Psychology*. 42 (1), 7–97.
[69] J. Warin. 2011. Ethical Mindfulness and Reflexivity: Managing a Research Relationship With Children and Young People in a 14-Year Qualitative Longitudinal Research (QLR) Study. *Qualitative Inquiry*. 17 (9), 805–814.
[70] S. Webb. 2005. Receptive and Productive Vocabulary Learning: The Effects of Reading and Writing on Word Knowledge. *Studies in Second Language Acquisition*. 27 (1), 33–52.
[71] C.H. Weiss. 1972. *Evaluation research: methods for assessing program effectiveness*. Englewood Cliffs, NJ: Prentice-Hall.
[72] H. Wilson. 2005. Testing the Covert Method of Grammar Teaching: A Pilot Study. In *Proceedings of the CATESOL State Conference*. 2005.
[73] R.K. Yin. 2009. *Case study research: design and methods*. Thousand Oaks, CA, USA: Sage Publications, Inc.