Hedy: A Gradual Language for Programming Education

Felienne Hermans
f.f.j.hermans@liacs.leidenuniv.nl
Leiden Institute of Advanced Computer Science, Leiden University
Leiden, the Netherlands

ABSTRACT

One of the aspects of programming that learners often struggle with is the syntax of programming languages: remembering the right commands to use and combining those into a working program. Prior research demonstrated that students submit source code with syntax errors in 73% of cases and even the best students do so in 50% of cases. An analysis of 37 million compilations by 250.000 students found that the most common error was a syntax error, which occurred in almost 800.000 compilations. It was also found that Java and Perl are not easier to understand than a programming language with randomly generated keywords, stressing the difficulties that novices face in understanding syntax.

This paper presents Hedy: a new way of teaching the syntax of a programming language to novices, inspired by educational methods by which punctuation is taught to children. Hedy starts as a simple programming language without any syntactic elements such as brackets, colons or indentation. The rules slowly and gradually change until the novices are programming in Python. Hedy is evaluated on 9714 programs.

CCS CONCEPTS

• **Applied computing → Education**; • **Software and its engineering → Parsers**; *General programming languages.*

KEYWORDS

programming education, gradual programming, Hedy, Python

ACM Reference Format:
Felienne Hermans. 2020. Hedy: A Gradual Language for Programming Education. In *2020 International Computing Education Research Conference (ICER '20), August 10–12, 2020, Virtual Event, New Zealand.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3372782.3406262

1 INTRODUCTION

With computer technology being a tool of rapidly growing importance in nearly all aspects of life, the world also needs programmers. In fact, the world needs ever more programmers; the number of software developer jobs is expected to grow with 17% by 2024, much faster than the average rate among other professions [1]. In response to this rising demand, countries such as the Netherlands are now implementing programming and computer science (CS) curricula in high schools [7]. University CS programs around the world are growing rapidly as well. However, this increase does not address the shortage of programmers, since CS programs typically have attrition rates as high as 40% [9], which is higher than other programs that are traditionally considered as difficult such as physics. The Dutch bureau of statistics reports that about 60% of physics students but only 50% of CS students finish their degree in 6 years [62]. It has been hypothesized that, in universities, this is due to the current instructional techniques that are used [9] and expectations that are set too high [38], current introductory programming courses ask too much of novice CS students, while providing too little guidance.

One of the aspects of programming that learners struggle with is the syntax of programming languages. The precision that is needed while programming is often named as a factor which contributes to the difficulty in learning programming. This requires *"a level of attention to detail that does not come naturally to human beings"* [40]. Denny found that weak students submit source code with syntax errors in 73% of cases and even the best students do so in 50% of cases [16]. Altadmri and Brown analyzed 37 million compilations by 250.000 students and found that the most common error to be a syntax error: mismatched brackets, which occurred in almost 800.000 compilations [3]. Other researchers found that common programming languages Java and Perl are not easier to understand than a random language, stressing the difficulties that novices face in understanding syntax [56]. Interestingly enough, students assess learning syntax as more problematic than teachers [44], which might help us understand why little effort is given to the explicit explanation of syntax in many programming classrooms.

Various approaches have been tried to make learning to program easier for novices. Three main approaches in previous work can be distinguished: 1) mini or toy languages specifically aimed at teaching, such as Scratch [47] or LOGO [42], 2) sub-languages of full programming languages like MiniJava [48] and 3) incremental languages which start with a subset of a language and incrementally add new concepts, such as DrScheme [22].

This paper proposes a new form, a *gradual* language with an increasingly complex syntax, based on how punctuation is taught to novice readers in natural language education. Programming is often seen as part of STEM (Science, Technology, Engineering & Mathematics) [50, 64], however, learning a programming language also shares significant characteristics with learning a natural language, since learners also have to learn about both semantics and syntax. It has been argued that programming education might be improved by employing instructional strategies common in natural language teaching [27, 45].

This paper is an implementation of that line of thought, and seeks inspiration to learn syntax from how students learn to write their first natural language. When learning a first language, novices

ICER '20, August 10–12, 2020, Virtual Event, New Zealand
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-7092-9/20/08…$15.00
https://doi.org/10.1145/3372782.3406262

do not learn syntax, punctuation and capitalization at once. Initially, they only write letters in lowercase, as shown in Figure 1.

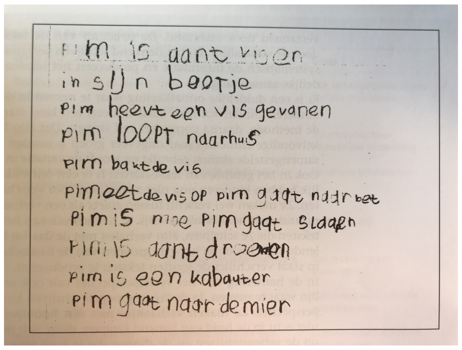

pim is aant vissen
in sijn bootje
pim heevt een vis gevanen
pim loopt naarhuis
pim bakt de vis
pim eet de vis op pim gaat naar bet
pim is moe pim gaat slapen
pim is aant droomen
pim is een kabauter
pim gaat naar de mier

Figure 1: Handwriting of a novice learner (Dutch), using only lowercase letters in initial writing [30]

Only in later stages, learners will learn to add uppercase, periods, comma's and semicolons. It has been argued that this gradual form of teaching works best at a young age, since beginning writers are young (around 6-7 years of age), and at that age still operate at the pre-operational stage [41]. This means they will be so occupied with writing, that they cannot concentrate yet on, for example, punctuation and story-lines. Many modern researchers believe that when novices are learning a new skill they will operate at one of the lower Piagetian stages, irrespective of their age. This notion has also gained popularity in Computer Science education [36]. Lister also argued that the cognitive load of students while programming might be overloaded by the simultaneous teaching of programming concepts, syntax and problem solving, and that students behave similar to younger children learning to write.

This paper presents a new way of teaching the syntax of a programming language to novices: with a small and very simple programming language of which the syntax rules slowly and gradually change until the novices are programming Python, similarly to how students acquire their first written language. Hedy supports different levels, each with new commands *and* increasingly complex syntax. For example, printing at Level 1 is done by simply using the command `print` followed by text:

```
print Hello!
print Welcome to your first programming lesson
```

In Level 2, variables are added as a concept, but quotes still are not needed:

```
print name is Jason
print Welcome to your first programming lesson name
```

In Level 3, quotation marks are introduced in the `print` statement, and the code to print is then:

```
print name is Jason
print 'Welcome to this programming lesson' name
```

Hedy is evaluated on 9714 programs, showing a relative low number of errors, and giving rise to several improvements of the language and its error messages.

The contributions of this paper are:

- The design of a new, *gradual*, programming language for novices (Section 3)
- A corresponding open source implementation (Section 4)
- A detailed examination of the implications of Hedy, and of gradual programming languages in general (Section 5)
- An exploratory evaluation on almost Hedy 10.000 programs (Section 6)

2 RELATED WORK

2.1 Learning Punctuation

Research has shown that learning correct punctuation is a long process [21] and that, while learning, novices often temporarily forget previously acquired knowledge [53]. Extensive practice can speed up learning to use punctuation in a correct way [35]. Research also shows repetition is important in learning language, for example: a word needs to be read seven times before it is stored in long-term memory [61].

2.2 Programming Languages for Novices

In previous research three different approaches in programming languages for novices can be distinguished [12]:

Mini-languages Mini-languages are languages that are small and especially designed to support learning to program. A well-known example of a mini-language is Papert's LOGO [42]. More modern examples of mini-languages are Scratch [47] and Karel the Robot [10]. Mini-languages are said to *"provide a solid foundation for learning a general purpose language"* [13], but learning a mini-language can also be a goal in itself, leading to the acquisition of algorithmic thinking.

Sub-languages In the sub-language approach, programming is taught to novices using only a set of commands from a bigger programming language, which typically is one that is used in practice such as Pascal or later, Java. Initially the idea of sub-languages was not to have them successively grow, but to simply select a subset to teach. Examples are Helium, a subset of Haskell for educational purposes [26], and MiniJava [48] and ProfessorJ [25] for Java.

Incremental approach The incremental approach first teaches a small subset of a programming language where each subset introduces new programming language constructs. This approach was first implemented for PL/1 by Holt [29] *et al.* and later also applied to Fortran [6] and Pascal [5]. Some other versions of incremental teaching used subsets that were explicitly not arranged as a hierarchy where "higher level" contained the "lower level" but instead divided the language into overlapping languages like chapters in a textbook would. ones [58]. More recently, DrScheme used a similar approach for Scheme [22].

2.3 Educational Effects of Different Languages

Researchers have attempted to understand the effect of different languages on study success of learners. Wainer and Xavier for example compared a course in Python with one in C and found students using Python were less likely to fail and scored higher on the exam [63] This gives credibility to our hypothesis that a simpler language with less syntactic elements—Python in this case–has benefits in teaching. One might wonder though, whether teaching a simpler language interferes with more complicated language learner later, but research does not suggest this. Manilla *et al.* studied eight students who learned programming in Python before moving on to Java, and found no disadvantages from having learned to program in a simpler language when later learning a more complex language [39]. Enbody also performed two studies showing that Python can also serve as good preparation for subsequent C++ courses [19, 20].

2.4 Spiral Approach

As early as 1977, Shneidermann described what he called a 'spiral approach' to learning programming, which shares a great number of design goals with Hedy. Shneidermann argues that, to accommodate the cognitive limits of learners, learning to program should start with a small amount of syntax (and accompanying simple semantics). He writes: *"A programming course might begin by teaching the semantics and syntax of free-format input and output statements, then progress to the simplest forms of the assignment statement and arithmetic expressions. At each step the new material should contain syntactic and semantic elements, should be a minimal addition to previous knowledge, should be related to previous knowledge, should be immediately shown in relevant, meaningful examples and should be utilized in the student's next assignment. This is the spiral approach".* Shneidermann also recommends extensive practice of simpler forms before advancing to new forms and argues this could help students who would otherwise be overwhelmed [51]. While this idea seems very promising, and is rooted in cognitive science, it was not followed up; the paper currently only has 15 citations. This might also be due to the fact that the paper did not have a corresponding implementation in a language or lesson series. Recently, papers have also described languages that extend over time, for example Cazzola and Olivares [14] describe a language which gradually builds up to JavaScript, in which students were provided with different JavaScript variants, where each variant focused on another language feature, e.g., loops, recursion, exception handling, object orientation. Vega *et al.* describe their Java-based system Cupi2, in which students solve increasingly more complicated problems, with partly generated programs [60]. DrScheme also consists of a set of languages, each a larger subset of Scheme [22]. Like Shneidermann argues, these languages start small, add one concept at a time, and limit the language to concepts which have been explained and practiced. However, while the above papers share philosophical principles with Hedy, as far as we know a language of which the syntax gradually changes, rather than being extended, has not been described or implemented before.

2.5 Extensible and Growable Languages

Languages that change over time have also been proposed in domains outside of programming education. Fortress for example [2] is a *growable* language that is designed to *"grow over time to accommodate the changing needs of its users"*. Lusth *et al.* proposed using reflection and overloading to grow and shrink languages, for example to comply with specific coding style guidelines [37].

3 HEDY DESIGN GOALS & PRINCIPLES

3.1 Design Goals

The overarching goal of Hedy is to gradually add syntactic complexity to a Python-like language, until novices have mastered Python itself. The target audience of the language are novices from the age of about 11, who are expected to exert a limited amount of cognitive load on reading natural language, and are old enough to be ready to reason at the formal operational stage. Hedy follows these design principles:

(1) **Concepts are offered at least three times in different forms** Research from writing education [21, 53] has shown that it is needed to offer concepts in different forms over a long period of time. Furthermore it has been shown that a word needs to be read 7 times before it is stored in long-term memory [61].
(2) **The initial offering of a concept is the simplest form possible** Previous research has shown that syntax can be confusing for novices [16, 56]. We therefore want to keep the initial syntax free of as many keywords and operators as possible to lower cognitive load.
(3) **Only one aspect of a concept changes at a time** In his paper on the Spiral approach, Shneidermann argued for minimal changes [51]. Recent work sheds a light on why small changes are effective, Tshukudu and Cutts show that learners are able to transfer knowledge from one programming language to another if the syntactic differences are not too big [59].
(4) **Adding syntactic elements like brackets and colons is deferred to the latest moment possible** Previous research in the computer science education domain has shown that operators such as == and : can be especially hard for novices, and prevent their effective *vocalization* of code [28], which is known to be an aid in remembering [57]. Research from natural language acquisition shows that parenthesis and the colon are among the latest element of punctuation that learners typically learn [23]. Given the choice between colons and parenthesis and other elements like indentation, the latter are introduced first.
(5) **Learning new forms is interleaved between concepts as much as possible** We know that *spaced repetition* [24] is a good way of memorizing; and that it takes time to learn punctuation. With that in mind, we want to students as much time as possible to work with concepts before the syntax changes.
(6) **At every level it is possible to create simple but meaningful programs** It is important for all learners to engage in meaningful activities [11]. Our experience in teaching high-school students (and even university CS students) is that learning syntax is not always seen as a useful activity. Students experience a large discrepancy between the computer being smart, for example by being able to multiply 1,910 and 5,671 within seconds, while simultaneously not being able to add a missing colon independently. We anticipate that when the initial syntax is simple, allowing novices to create a fun and meaningful program, they will later have more motivation to learn the details of the syntax.

3.2 Levels

In its current form, Hedy consists of 13 different levels. Table 1 shows an overview of commands available at different levels.

Table 1: Overview of commands available at different levels. 'x' indicates a command is available, 'new' means it is available in a new form compared to the previous level

Command Level	print	ask	echo	assign	assign list	if	else	repeat
1	x	x	x					
2	x	new		x	x			
3	new	x		x	x			
4	x	x		x	x	x	x	
5	x	x		x	x	x	x	x
6	x	x		x	x	x	x	x
7	x	x		x	x	new	new	new

3.2.1 Level 1: Printing and input. At the first level, students can print text with no other syntactic elements than the keyword `print` followed by arbitrary text. Level 1 code and the corresponding output can be seen in Figure 2. Furthermore students can ask for input of the user using the keyword `ask`. Here we decided to use the keyword `ask` rather than `input` because it is more aligned with what the role of the keyword is in the code than with what it *does.* Input of a user can be repeated with `echo`, so very simple programs can be created in which a user is asked for a name or a favorite animal, fulfilling Design Goal 6.

3.2.2 Level 2: Assignment using 'is': numbers and lists. At the second level, variables are added to the syntax. Defining a variable is done with the word `is` rather than the equals symbol fulfilling Design Goal 3 and Design Goal 4. We also add the option to create lists and retrieve elements, including random elements from lists with `at`. Adding lists and especially adding the option to select a random item from a list allows for the creation of more interesting programs such as a guessing game, a story with random elements or a customized dice.

3.2.3 Level 3: Quotation marks and types. In Level 3, the first syntactic element is introduced: the use of quotation marks to distinguish between variables and 'plain text'. In teaching novices we have seen that this distinction can be confusing for a long time, so offering it early might help to draw attention to the fact that computers need information about the types of variables. This level is thus an interesting combination of explaining syntax and explaining programming concepts, which underlines their interdependency.

3.2.4 Level 4: Selection with if and else flat. In Level 4, selection with the if statement is introduced, but the syntax is 'flat', i.e. placed on one line, resembling natural language more:

```
if name is Bert print 'Yellow'
```

Else statements are also included, and are also placed on one line, using the keyword `else`:

```
if name is Bert print 'Yellow' else print 'Orange'
```

3.2.5 Level 5: Repetition with repeat x times. In working with non-English native Python novices, we have previously found that the keyword `for` to be a confusing word for repetition, especially because it sounds like the word 'four' [28]. For our first simplest form, according to Design Goal 2, we opt to use `repeat x times`, as common in other educational programming languages also, including Quorum [55] and TigerJython [31].

In its initial form code is placed on one line, similar to the `if` statement in Level 4:

```
repeat 5 times print 'Hello World'
```

3.2.6 Level 6: Calculations. In Level 6, students learn to calculate with variables, so addition, multiplication, subtraction and division are introduced. While this might seem like a simple step, the use of `*` for multiplication, rather than $\times$, and the use of a period rather than a comma as decimal separator for non-US students is a steep learning curve and we thus believe it should be treated as a separate learning goal, following Design Goal 3.

3.2.7 Level 7: Code blocks. After Level 6, there is a clear need to 'move on', as the body of a loop (and also that of an if) can only consist of one line, which limits the possibilities of programs that users can create. We assume this limitation will be a motivating factor for learners, rather than 'having to learn' the block structure of Python, they are motivated by the prospect of building larger and more interesting programs (Design Goal 6). The syntax of the loop remains otherwise unchanged as per Design Goal 3, so the new form is:

```
repeat 5 times
    print 'Hello'
    print 'World'
```

3.2.8 Level 8: For syntax. Once blocks are sufficiently automatized, learners will see a more Python-like form of the for loop, namely: `for i in range 0 to 5`. This allows for access to the loop variable `i` and that in turn enables the creation of more interesting programs, such as counting to 10. As per Design Goal 3, the change is made small, and to do so (following Design Goal 4), brackets and colons are deferred to a later level, but indentation which was learned in Level 7 remains.

3.2.9 Level 9: Learning the colon. To make the step to full Python, learners will need to use the colon to denote the beginning of a block, in both loops and conditionals. Because blocks are already known, we can teach learners to use a colon before every indentation, and have them practice that extensively.

3.2.10 Level 10: Repetition and selection nested. To allow for enough interleaving of concepts (Design Goal 5), we defer the introduction of round brackets and focus on more conceptual additions: the nesting of blocks. We know indentation is a hard concept for students to learn, so this warrants its own level (Design Goal 3).

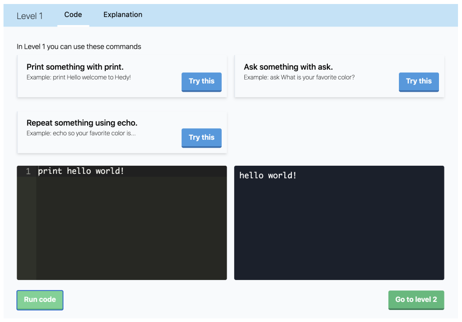
3.2.11 Level 11: Adding round brackets. Level 11 adds round brackets in `print`, `range` and `input`. As per Design Goal 4, these are added as late as possible.

3.2.12 Level 12: Adding rectangular brackets. In level 12, learners encounter different types of brackets for the first time, because it adds rectangular brackets for list access, which up to now was done with the keyword at, following Design Goal 2.

3.2.13 Level 13: is becomes = and ==. In the final level Hedy becomes a subset of Python by replacing the word `is` in assignment and equality checks by = and ==.

3.3 User Interface

The current implementation of Hedy is shown in Figure 2. The interface includes an editor in which to enter code on the left, and a field for output on the right. Each level also includes buttons to try out commands introduced in each level. For each level videos with explanation and written assignments are also accessible from within the user interface.



In Level 1 you can use these commands

Print something with print.
Example: print Hello welcome to Hedy!

Ask something with ask.
Example: ask What is your favorite color?

Repeat something using echo.
Example: echo so your favorite color is...

```
print hello world!
```

```
hello world!
```

Figure 2: Level 1 of the Hedy user interface in English

4 HEDY IMPLEMENTATION

Currently Hedy is implemented in Python, using the Lark parser. Code is parsed and subsequently *transpiled* into Python, for example by adding brackets where needed. The resulting Python code is then executed. Hedy can be downloaded and ran in an IDE, but also has a web version in which Hedy can be simply typed in the browser, as common in modern web-based IDE's for teaching such as repl.it and Trinket. This enables running Hedy without installing anything, and will thus likely increase adoption in schools where teachers often have limited or no posiibilities to install software. It also means Hedy can be used on mobile phones and tables. Hedy's code base is open source, available on GitHub[1].

4.1 Grammar and parsing

Listing 4.1 shows the grammar used in Level 1. As shown in the listing, the grammars of Hedy parse more than the language allows. For example, the grammar of Level 1 not only parses `print`, `ask` and `echo`, but any word that is placed before text separated with a space. As such, the Level 1 parser also accepts `show hello world`. This allows us to give specialized error messages like *"show is not a command in Hedy. Did you mean print?".*

```
start: program
program: command (newline + command)*
command: "print " text -> print
        | "ask " text -> ask
        | "echo " text -> echo
        | textwithoutspaces " " text -> invalid
PUNCTUATION : "!" | "?" | "."
newline: "\n"
text: (LETTER | DIGIT | PUNCTUATION | WS_INLINE )+
textwithoutspaces: (LETTER | DIGIT)+ -> text
%import common.LETTER
%import common.DIGIT
%import common.WS_INLINE
```

Listing 1: Grammar of Level 1 in Lark

[1]A link will be shared in the final version of the paper

5 IMPLICATIONS OF GRADUAL PROGRAMMING LANGUAGES

Hedy is a programming language which gradually adds more syntax, and is designed to lower the syntax barrier by giving novices more time to learn syntax. Here we hypothesize on a number of ways in which gradual programming languages can contribute to learning.

5.1 Reach Higher Levels of Abstraction Early

Programming is taught for various reasons, including teaching programming to train *computational thinking* [66]. We envision that because the syntax barrier is deferred to later in a novice's programming career, a learner can learn about higher order skills such as how multiple statements work together (Macro-Structure or Relations in Schulte's *Block Model* [49]).

5.2 Differentiation Between Learners

Programming education knows large differences between students within a classroom, which can lead to frustration both in quicker and slower students [4]. From Level 7 on, Hedy allows learners within a classroom to create the same programs with a varying level of 'real' Python syntax. One can imagine that all students work on the same assignment, where some use Hedy Level 7 while others have already learned full Python, while still producing similar end-results.

5.3 Run Code at Previous Levels

It is known that learners, when learning a new concept, often 'fall back' and forget about other concepts they previously acquired [36]. Because of the leveled structure of Hedy, it is possible to attempt to compile failing code with a previous level of Hedy, to detect exactly where the student made a mistake and give a concrete suggestion based on that knowledge, i.e. "Remember, you need to indent a loop".

5.4 Localized Keywords

Many programming languages for young novices use keywords which can be localized to various natural languages, including Scratch [47]. Unsurprisingly, research has shown that learners learn programming more easily in their own mother tongue [15]. Since Hedy code, especially at the lower levels, has a grammar simpler than full Python, this opens up the possibility of using keywords in different languages, and changing them to English and Python versions gradually too.

5.5 Gradual Error Messages

It is known that error messages are a source of frustration for novice programmers. Especially in Python error messages do not always aid in finding the error [17, 32]. One reason for this is that the learner is exposed to the full spectrum of error messages at once, and that a program as simple as `print('Hello')` can lead to various error messages if the closing quote or round bracket are omitted. An omitted closing quote leads to the error message: *SyntaxError: EOL while scanning string literal* which is unlikely to help a novice programmer locate their mistake. The design philosophy of Hedy can improve error messages in two ways. Firstly, the simple grammar of early levels will also allow for more precise error messages since the range of options is more limited. This is illustrated by the grammar of Level 1 in Listing 1. The three first options of the `command` rule are allowed Hedy commands, when none of those match, the fourth option will, allowing for the generation of a detailed error message. For example this Hedy code: `prnt Hello World` generates the error message: `prnt is not a Hedy level 1 command, did you mean print?` which is arguably more helpful than `NameError: name 'prnt' is not defined` which Python would produce. This code could even be fixed for the novice programmer (see further Section 5.6).

In addition to more precise error messages, Hedy could also gradually increase the complexity of error messages, for example by initially using 'End of Line' rather than EOL complemented with an explanation that Python gets confused because it does not know where the code ends. This could later be replaced by EOL, so learners are prepared to make the switch to the full Python language including Python error messages.

5.6 Program Repair

The fact that Hedy code at low levels is simpler than a full programming language also creates an opportunity to apply program repair techniques. While program repair techniques have improved significantly over the past years [33, 34], the general problem of repairing buggy programs remains hard. Recently, some initial experiments were attempted to use program repair techniques to generate hints for novices, however only limited success was achieved: about 30% of programs could be repaired and novices did not benefit from the generated hints [67]. Hedy might ease this. For example, at Level 1, Hedy consists of three keywords only: `print`, `ask` and `echo`, detecting typos in keywords and subsequently suggesting the correct keyword is almost trivial.

6 EVALUATION

In order to evaluate the Hedy programming language and the corresponding web-based user interface, we ran an exploratory user study on Level 1 to Level 7. Hedy was released to the public on Monday March 16th 2020, after which data was collected for 3 weeks, until April 6th 2020. Announcements about Hedy were shared through Twitter, through a press release of the university where the first author is employed, and were subsequently picked up by some Dutch news sites. Over the course of the evaluation, 11133 programs were gathered. Out of these, 791 programs were *demo programs* provided within the user interface, the code snippets which are generated for the Hedy user when pressing one of the 'Try this' buttons shown in Figure 2. The fact that 7.1% of programs consist of demo programs shows that these buttons are used, and as such that the interface is probably helpful in getting started. These demo programs were however written by us, and are therefore discarded from the evaluation which follows. We also discard *start programs*, which are the programs present in the user interface when a new level is opened by a Hedy programmer. In total, 628 (5.6%) of these start programs were excluded from further analysis. As such, the evaluation that follows will be based on 9714 submitted Hedy programs.

6.1 Dataset Overview

6.1.1 Level Usage. Not all Hedy levels were used equally over the course of our experiment. We assume this is mainly caused by the fact that not all levels were released at the same time, so we have been collecting data for the earlier levels for a longer period of time. Level 1 was released on March 19th, Levels 2 to 4 on March 23rd and 5 to 7 on March 31st. Figure 3 shows an overview of the number of programs created per level.

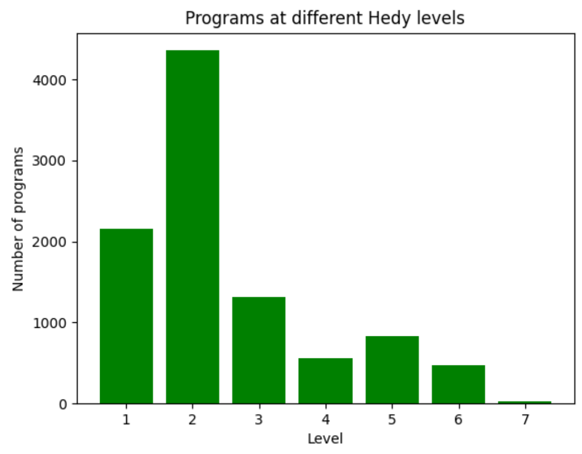


Figure 3: Histogram of Hedy programs per level

6.1.2 Program Lengths. Most Hedy programs that were created are small, with an average of 2.9 lines of code and a median of 2.0. However larger programs were also present in the dataset, with the longest one being 100. Figure 4 shows an overview of program lengths. As expected, programs increase in length for higher levels. Since more complicated command are included at higher levels, this allows for more interesting programs. Figure 5 shows the average program length per level.

6.2 Commands Used

As explained in Section 3, different levels of Hedy support different commands. An overview of these commands is presented in Table 1. Figure 6 then presents the commands that programmers used in the evaluation in correct programs. From this figure we can learn a number of things about Hedy. Firstly, the specially designed `echo` command seems to be popular in Level 1. This strengthens our belief

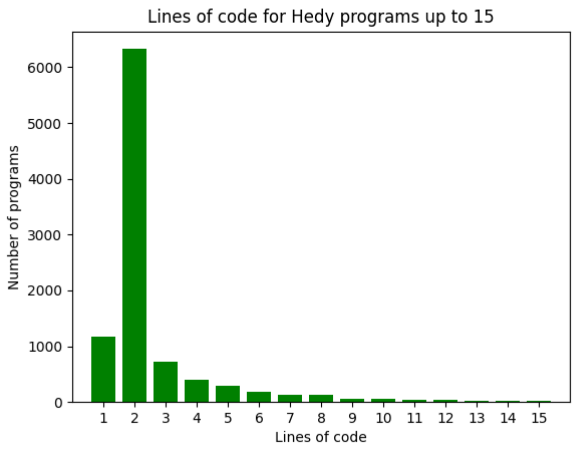


Figure 4: Histogram of program lengths

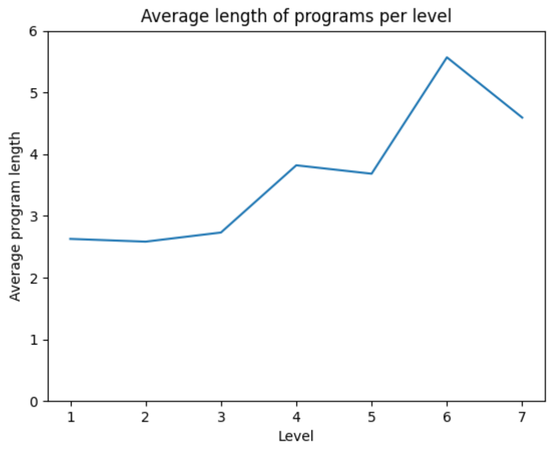


Figure 5: Overview of average program lengths per level

that giving novice programmers an easy way to interact with the interface is helpful. In Level 2, assigning to a list is a very popular language feature, accounting for more than half of the commands used in Level 2. Here too we present users with a language feature of which we think it would be out of their reach syntactically: correctly using rectangular brackets and command and quotation marks for strings is an error prone operation for novice programmers. Our assumption is that early exposure to complex features will motivate learners later on to learn about the same concepts with more syntax rules. We do see the usage of `assign list` drop starkly in Level 3, which might mean that novices focus mainly on *new* features and only practice them in moderation in later levels.

Figure 6 also points to potential issues. For example, it indicates that the use of `ask` decreases from Level 1 to Level 2. This might be caused by the fact that the syntax changes from Level 1 to Level 2, and students might be confused or discouraged by the new form.

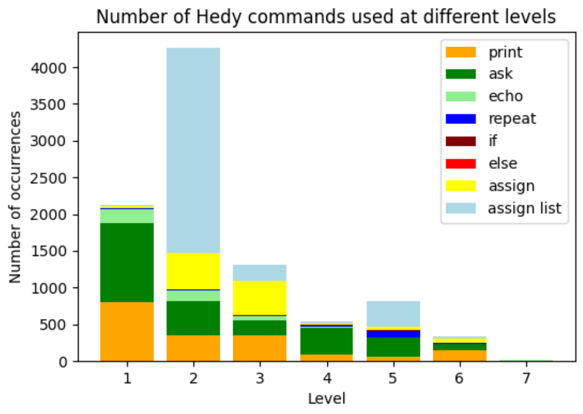


Figure 6: Use of commands over different levels

To study that in more detail, we analyze erroneous programs and examined which commands were used in the wrong way. This analysis needed to be done manually, since erroneous programs do not parse, and as such we could not extract the commands for the generated parse tree. The result of that analysis can be found in Figure 7. This figure indeed confirms that the new form of `ask` and the removal of `echo` are sources of errors. Although it is important to note that in absolute numbers, the amount of errors is quite low, for example there are less than 100 errors involving `ask` in Level 2, in which over 4000 programs were created in total.

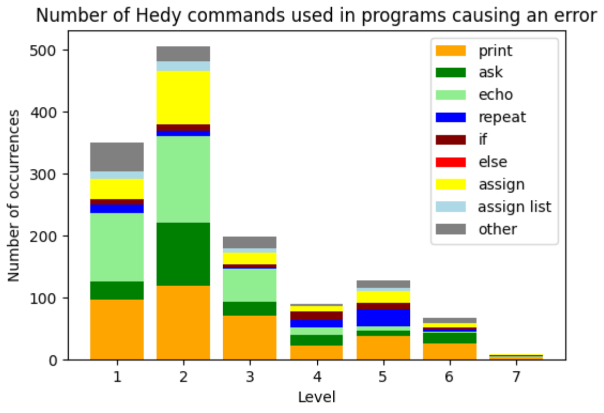


Figure 7: Commands causing errors over different levels

7 ERROR CLASSIFICATION

The main goal of Hedy is to reduce the initial complexity of programming languages sometimes called the *syntax barrier* for novices, and as such, we aim at an error rate that is lower than that of traditional textual languages. From our first exploratory study, this seems to be the case. Out of the 9714 programs we evaluated, 1363(12.2%%) resulted in an error, either a syntax error or a conceptual mistake. While this obviously is a broad strokes comparison, this is lower than percentages reported in traditional textual languages. For example, Denny *et al.* revealed that weak students submit source code with syntax errors in 73% of cases and even the best students do so in 50% of cases [16]. As a first rough measure of success, the error rate of Hedy is quite a lot lower than these numbers in textual languages. Of course this might not only be due to simpler syntax bit also due to a lower number of programming concepts present in levels 1 to 7.

To gain a deeper understanding of the underlying issues which caused the errors, we manually inspected all 1363 programs containing an error. The following subsections describe the 11 categories into which we coded the answers.

Table 2: Errors classified into 11 different categories

Category	# programs	% of errors
Incomplete code	253	18.6%
Code ran at the wrong level	251	18.4%
Misspelled or misplaced keyword	233	17.1%
Invalid character	219	16.1%
Use of invalid spaces	171	12.5%
Quotes	111	8.1%
Multiline command	72	5.3%
Python	26	1.9%
Line numbers	13	1.0%
Nested code	10	0.7%
Misconception	4	0.3%
Total	1363	100

7.1 Incomplete code

In many cases (253) novices forgot to use a keyword or an argument, or other mandatory syntax elements such as commas in a list. There are a few directions of improvement for the parser and transpiler possible that would improve the reporting of these issues. For example adapting the grammar to allow for keywords without arguments, and subsequently presenting an error message to the user. However even with better error messages, it is likely that issues with incomplete code will common. Incomplete code is known to be of the common issues with textual languages: the most common error found by Altadmri and Brown was a case of incomplete code too (mismatched brackets [3], so this seems to be an inherently hard concept to learn for novices.

7.2 Code Ran at Wrong Level

As expected, the most common mistake, occurring in 251 programs (18.4% of programs with an error) is the mistake where learners use commands from a wrong level. We expected this to happen when learners run code they learned at a previous level at a higher one. This was the case in 207 cases. Somewhat surprising to us, the reverse also occurred, although not very often: only in 46 cases students used keywords or commands not yet introduced. For example a student aimed to run this code at Level 4: `repeat 100 times print '.'` while repeat is only introduced at Level 5.

An obvious improvement to Hedy, also discussed in Section 5.3, would be to try to run the code at different levels, and give feedback to the learner that they are using either a command no longer in use, or a command not yet unlocked. This is however more complicated than simply calling the transpiler again at a higher level, since in many cases students did not simply write perfect code from a lower level, but mixed code from different levels. A more elaborate technique is needed to slice the code and treat different lines differently.

7.3 Misspelled or Misplaced Keywords

With 233 errors, misspelled keywords are a relatively common error. Examples are writing keyword in uppercase or omitting one letter. We also saw cases of programs using keywords in Dutch or Spanish rather than in English, or changing keywords to fit natural language better, for example using `are` instead of `is`:

```
choices are m, n, b, v, c
print choices at random
```

Most likely this problem is caused by the fact that because a list is assigned and the word 'are' is the most fitting form of the verb.

While misspelling a keyword is a simple mistake to make, generating meaningful error messages is not a trivial endeavor. In Levels 1 and 2 the error message is quite meaningful, for example the code `prin Hello'` results in the error message: `prin is not a keyword in Level 1, did you mean print?`, which is possible because of the small grammar of these levels. At higher level though, errors gets increasingly harder to pinpoint, as is common in many textual programming languages.

7.4 Issues with Quotation Marks

Out of all erroneous programs, 111 are related to issues with quotes. In the current version of Hedy, only singular quotes are supported, but some students tried double quoted instead of attempted to naively print string literals containing a single quotation mark, such as: `print 'what's your name?'`. Forgotten closing quotation marks also occurred, as well as the omission of quotation marks altogether, although that could also be seen as running code at a lower level, since Level 1 does not require quotation marks in printing.

An especially interesting case occurred in two Level 2 programs in the same session, which reversed the meaning of quotation marks entirely: `print we do not know that my name is 'name'`. This might be a general misconception about the meaning of quotation marks, but might also be some residue of Level 1 understanding, where the addition of the fact that we can print variables and text together is interpreted in the exact opposite direction: variables need quotation marks, but text does not (as in Level 1).

A subcategory of quotation issues is the use of quotation marks in commands other than `print`. In an attempt to keep the language simple, and to make small changes as per Design Goal 3, quotation marks are firstly only added to `print`. The `ask` keyword and assignment can still be used without quotation marks in Level 3. So these are both valid Level 3 programs:

```
naam is Hank
ask What is your name?
```

In 35 programs however, students assumed that quotation marks would also be needed in other commands involving string literals. While that assumption is sensible, the limited amount of 35 mistakes with this (versus 2818 successful programs involving `ask`, `print` or assignment in Level 3 and up) strengthen our belief that this was the right design choice.

7.5 Issues involving Spaces

145 errors were related to the incorrect use of spaces, for example starting a line of code with a space or forgetting a space between arguments, like: `print 'hey 'name`
Some of these issues might be addressed with a stringer parser—the example above could be parsed with relative ease—but for others the question of how to address them is more related to pedagogy. For example, starting a line with spaces currently is not allowed, even in lower levels, because that might cause issues later when indentation is introduced. Error messages could be improved for these kinds of errors. These programs now cause parse errors since the grammar does not support starting with a space, hence the error message is: `No terminal defined for ' ' at line 1 col 1`, which is not friendly. The parser could be improved to generate more meaningful errors here.

7.6 Issues involving Other Invalid Characters

Apart from quotation marks, other characters can also cause issues, which they did in 219 programs. For example, some students attempted to print HTML code, maybe to make the output of the code look nicer:
`print <p> cheese <p/>`, or used comma's in print statements:
`print He is thinking, Pete must be here`. Other invalid characters included letters with accents, or errors due to encoding issues. Improving the parser and the reporting of such errors is certainly an improvement to be made, however these issues are not directly related to the design philosophy of Hedy.

7.7 Multiline Commands

72 errors were due to the fact that Hedy programmers assumed that commands could span multiple lines, for example:

```
k is ask choose a number
if k is 1 print 'yes'
else print 'wrong'
```

This, however, could also be a case of negative transfer from other programming languages in which `if` and `else` are placed on different lines.

7.8 Python Commands

Almost 2% of errors (26 programs) seem to be caused by students with some experience in programming in Python, or other programming languages, using commands that they are used to in those languages. For example a program submitted at Level 3 read:
`print 'Hello world', ' d'.`
This program uses the comma as a separator between arguments, and likely stems from exposure to other languages. Due to the similarity Hedy shares with Python, some transfer (negative or positive) is expected. It is certainly an interesting endeavour to try to understand this transfer better, for example by running a study in which we measure the Python knowledge and programming knowledge of Hedy users. It is an open question how to communicate the overlap in programming languages to users; should we stress the fact that Hedy is like Python, or should we present it as its own language and downplay the overlap?

7.9 Line Numbers

To help learners read the example programs provided in the accompanying lesson materials, we included line numbers in the example programs. A small number of programs accidentally included these line numbers, causing errors. We observed in subsequent programs within the same session that these error were relatively easy to solve for learners. This phenomenon, although not frequent, does raise the question of whether using line numbers in example code is helpful, or whether we should make it clearer that they are not part of the code snippets themselves.

7.10 Nested Code

Levels 1 to 6 do not support the nesting of more than 2 commands; you can include one `if` in a `repeat` or the other way around, but not more, because there are no code blocks yet. However, some learners (0.7%) attempted to nest multiple `if` commands. In a way, this is a special case of running code at a level too low, but not exactly, since these programs tried to nest the `if` commands on one line. While this category is very small, it does point to interesting issues in the Hedy design, since there will always be a small part of the learners that will try to stretch the limits of a level. How to handle that is an interesting open question.

7.11 Misconceptions

Four of the errors seem to be not related to syntax errors, but are due to misconceptions about how computers operate. A common misconception for example is that computers are smart and can process natural language [43].

Firstly, one programmer kindly asked the computer to select a number at Level 2 rather than using the random syntax:

```
l is 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 , 10
ask choose a number above 0 and below 10
```

Hedy shares with other programming languages the issue that learners do not see the difference between a machine that can fully comprehend natural language and a machine that simply executes a programming language, even though Hedy syntax is simpler. It would be valuable to compare Hedy to a more traditional programming language to investigate whether Hedy induces this misconception to a lesser extent, because it supports less commands initially and thus looks less like a programming language, or to a higher extent, because it is more natural language like.

The second program showing a misconception is this one:

```
print 'the animal is a' animal
animal is cat
```

The misconception here seems to be the assumption that the order of a program does not matter, and potentially also that assignment to a variable creates a connection which can be resolved later. These too are both common misconceptions [18, 46, 52, 54]

8 DISCUSSION

This paper describes Hedy: a new programming language for novices that gradually adds and modifies syntax to ease the learning of

Python. This section describes some open issues not yet addressed in this paper.

8.1 Trade-offs

One of the overarching aspects of discussion around Hedy is the trade-off between starting simple and then later learning a new syntactic form, which takes mental effort, and learning the right syntax from the start. While research shows that learning a simpler language does not interfere with more complicated ones later-on [39], we are not sure of the effects within a single language. One could argue that different levels of Hedy are different languages, but most likely in the mind of the user they will be seen as one. In computer science the vision seems to have been to learn the right thing from the start which presents cognitive overhead of remembering symbols and also might give the impression to learners that they need to learn arbitrary things. Learning to write on the other hand has mainly taken the opposite stance of allowing many different forms and adding small steps. Our goal is to explore a form of middle ground between these two forms of teaching.

8.2 Starting List Indices at 1

One of such trade-offs, addressed in the previous subsection, is where to start counting. In the current implementation, we abide by the convention adopted by Python and most other programming language's of starting at 0. We can very much envision a future version of Hedy in which counting initially starts at 1 to align with what students know from math, and only changing this to 0 in a later level of the language, since we have found that learning to count from 0 takes considerable effort and practice, and does not necessarily add value for learners.

8.3 Multiplication and Division

A second trade-off is the choice of operators for basic calculation. For Level 4, in which calculations are first introduced, we considered using × initially for multiplication and : for division, rather than * and \. That would be more in line with Design Goal 2 because it would align with what learners know in early high-school. We decided against that for practical reasons, because it would either require students to type the × character with a special key combination which is not something used in the remainder of the lessons, or require us to parse the letter `x` as multiplication which presented parsing issues. Also, most students will be familiar with a calculations on the computer, for example in a spreadsheet, where the `*` is also commonly used for multiplication and \ for division.

8.4 Access to the Type System

In Level 3, where types are introduced, we considered also adding a simple syntactic form of `type()` to allow novices to interact with the type system and ask Hedy what type a variable is. Ultimately we decided against that, because we want to train novices to remember and/or understand the types of variables themselves. However adding this could be an interesting experimental feature.

8.5 Remaining at Level 7

When learners have reached Level 7, a quite reasonable subset of Python is covered which includes variables, loops and conditions. One could imagine learners simply using this level of Hedy without advancing to full Python, using it as a mini-language. One could also imagine remaining at this level while teaching a range of computer science concepts such as sorting and filtering and search algorithms before adding more syntax.

8.6 Comments

The current version of Hedy does not include characters to be used to add comments. We made that decision to keep the language simple. However, while examining the error data set, we saw learners removing lines to understand what lines were erroneous. That means we might reconsider the inclusion of a comments character for future versions.

8.7 Access to Generated Python

Currently, we do not give learners in Hedy access to the Python code that their program generates, but there is no technical reason not to do so. Some block-based languages, including PencilCode [8] allow a user to translate their blocks into JavaScript, inspect and even edit the generated code. Research on switching between the block and text modality of Pencil showed that students often 'fell back' to using blocks when trying new constructs [65], but we are unaware of studies investigating the educational benefits of inspecting or adapting generated code.

9 CONCLUDING REMARKS

This paper introduces Hedy: a programming language for educational purposes of which the syntax grows with the level of the students. Hedy is designed based on 6 concrete design goals rooted in prior research. The paper presents an exploratory evaluation on 9714 programs in Level 1 to 7 which indicates a relatively low error rate of 12.2%. While confusion between code of different levels is the second most common source of errors which accounts for about 20% of errors, these mistakes only occurred in 251 of programs (2.6% of programs), which increases our confidence that the benefits of syntax that starts small outweighs potential confusion it causes in later levels. The evaluation furthermore presents a number of actionable improvements for Hedy, including more improved feedback for incomplete programs and program that misspell keywords, and better analysis of programs that combine commands from different levels.

The current research gives rise to a number of avenues for future research. Firstly of course—when times will allow for this again—we plan to perform observation studies on students programming Hedy, for example using a think aloud protocol. In that setting we can also measure the prior understanding of students of programming in general and Python in particular to gain a deeper understanding of interaction effects. Secondly, Levels 8 and up could be released and tested, improved with the lessons we learned from the current study. It would be interesting to see whether the error rate remains low when complexity of programs increases.

Additional studies, whether done online or in classrooms, would also allow for us to explore many of the trade-offs outlined in Section 8, such as starting counting at 1 or giving students access to the generated Python code.

REFERENCES

[1] 2019. Software Developers : Occupational Outlook. https://www.bls.gov/ooh/computer-and-information-technology/software-developers.htm#tab-6

[2] Eric E. Allen, Ryan Culpepper, Jon Rafkind, and Sukyoung Ryu. 2008. Growing a Syntax.

[3] Amjad Altadmri and Neil Brown. 2015. 37 Million Compilations: Investigating Novice Programming Mistakes in Large-Scale Student Data. *SIGCSE 2015 - Proceedings of the 46th ACM Technical Symposium on Computer Science Education* (2015), 522–527. https://doi.org/10.1145/2676723.2677258

[4] Sherry Andrews, Cara McFeggan, and Cynthia Patterson. 1998. Problems Students Encounter during Math Instruction in Mixed-Ability Classrooms. (1998).

[5] J. W. Atwood and E. Regener. 1981. Teaching Subsets of Pascal. *SIGCSE Bull.* 13, 1 (Feb. 1981), 96–103. https://doi.org/10.1145/953049.800969

[6] T. Balman. 1981. Computer assisted teaching of FORTRAN. *Computers & Education* 5, 2 (Jan. 1981), 111–123. https://doi.org/10.1016/0360-1315(81)90020-8

[7] Erik Barendsen, Nataša Grgurina, and Jos Tolboom. 2016. A New Informatics Curriculum for Secondary Education in The Netherlands. In *9th International Conference on Informatics in Schools: Situation, Evolution, and Perspectives*. Springer International Publishing, 105–117. https://doi.org/10.1007/978-3-319-46747-4_9

[8] David Bau, D. Anthony Bau, Mathew Dawson, and C. Sydney Pickens. 2015. Pencil code: block code for a text world. In *Proceedings of the 14th International Conference on Interaction Design and Children (IDC '15)*. Association for Computing Machinery, Boston, Massachusetts, 445–448. https://doi.org/10.1145/2771839.2771875

[9] Theresa Beaubouef and John Mason. 2005. Why the High Attrition Rate for Computer Science Students: Some Thoughts and Observations. *SIGCSE Bull.* 37, 2 (June 2005), 103–106. https://doi.org/10.1145/1083431.1083474

[10] Byron Weber Becker. 2001. Teaching CS1 with karel the robot in Java. In *SIGCSE '01*. https://doi.org/10.1145/364447.364536

[11] John Seely Brown, Allan Collins, and Paul Duguid. 1989. Situated Cognition and the Culture of Learning. *Educational Researcher* 18, 1 (Jan. 1989), 32–42. https://doi.org/10.3102/0013189X018001032

[12] P. Brusilovsky, , and Others. 1994. Teaching Programming to Novices: A Review of Approaches and Tools. (1994). https://eric.ed.gov/?id=ED388228

[13] Peter Brusilovsky, Eduardo Calabrese, Jozef Hvorecký, Anatoly Kouchnirenko, and Philip Miller. 1997. Mini-languages: A way to learn programming principles. *Education and Information Technologies* 2 (March 1997), 65–83. https://doi.org/10.1023/A:1018636507883

[14] Walter Cazzola and Diego Mathias Olivares. 2016. Gradually Learning Programming Supported by a Growable Programming Language. *IEEE Transactions on Emerging Topics in Computing* 4, 3 (July 2016), 404–415. https://doi.org/10.1109/TETC.2015.2446192

[15] Sayamindu Dasgupta, William Hale, Andrés Monroy-Hernández, and Benjamin Mako Hill. 2016. Remixing As a Pathway to Computational Thinking. In *Proceedings of the 19th ACM Conference on Computer-Supported Cooperative Work & Social Computing (CSCW '16)*. ACM, New York, NY, USA, 1438–1449. https://doi.org/10.1145/2818048.2819984 event-place: San Francisco, California, USA.

[16] Paul Denny, Andrew Luxton-Reilly, Ewan Tempero, and Jacob Hendrickx. 2011. Understanding the syntax barrier for novices. ACM, 208–212. https://doi.org/10.1145/1999747.1999807

[17] Rachel D'souza, Mahima Bhayana, Marzieh Ahmadzadeh, and Brian Harrington. 2019. A Mixed-Methods Study of Novice Programmer Interaction with Python Error Messages. In *Proceedings of the Western Canadian Conference on Computing Education (WCCCE '19)*. ACM, New York, NY, USA, 15:1–15:2. https://doi.org/10.1145/3314994.3325090 event-place: Calgary, AB, Canada.

[18] Benedict Du Boulay. 1986. Some Difficulties of Learning to Program. *Journal of Educational Computing Research* 2 (Jan. 1986), 57–73.

[19] Richard J. Enbody and William F. Punch. 2010. Performance of Python CS1 Students in Mid-Level Non-Python CS Courses. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education (SIGCSE '10)*. Association for Computing Machinery, New York, NY, USA, 520–523. https://doi.org/10.1145/1734263.1734437 event-place: Milwaukee, Wisconsin, USA.

[20] Richard J. Enbody, William F. Punch, and Mark McCullen. 2009. Python CS1 as Preparation for C++ CS2. In *Proceedings of the 40th ACM Technical Symposium on Computer Science Education (SIGCSE '09)*. Association for Computing Machinery, New York, NY, USA, 116–120. https://doi.org/10.1145/1508865.1508907 event-place: Chattanooga, TN, USA.

[21] Michel FAYOL and Patrick LEMAIRE. 1989. Une étude expérimentale du fonctionnement distinctif de la virgule dans des phrases: perspective génétique. *Études de Linguistique Appliquée; Paris* 73 (Jan. 1989), 71–80. https://search.proquest.com/docview/1307660874/citation/AD75E7B1D3194174PQ/1

[22] Matthias Felleisen, Robert Findler, Matthew Flatt, and Shriram Krishnamurthi. 2004. The TeachScheme! Project: Computing and Programming for Every Student. *Computer Science Education* 14 (2004), 55–77. https://doi.org/10.1076/csed.14.1.55.23499

[23] Emilia Ferreiro and Clotilde Pontecorvo. 1999. Managing the written text: the beginning of punctuation in children's writing. *Learning and Instruction* 9, 6 (Dec. 1999), 543–564. https://doi.org/10.1016/S0959-4752(99)00006-7

[24] Susan T. Fiske and Sean H. K. Kang. 2016. Spaced Repetition Promotes Efficient and Effective Learning: Policy Implications for Instruction. *Policy Insights from the Behavioral and Brain Sciences* 3, 1 (March 2016), 12–19. https://doi.org/10.1177/2372732215624708

[25] Kathryn E. Gray and Matthew Flatt. 2003. ProfessorJ: A Gradual Introduction to Java through Language Levels. In *Companion of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '03)*. Association for Computing Machinery, New York, NY, USA, 170–177. https://doi.org/10.1145/949344.949394 event-place: Anaheim, CA, USA.

[26] Bastiaan Heeren, Daan Leijen, and Arjan van IJzendoorn. 2003. Helium, for Learning Haskell. In *Proceedings of the 2003 ACM SIGPLAN Workshop on Haskell (Haskell '03)*. Association for Computing Machinery, New York, NY, USA, 62–71. https://doi.org/10.1145/871895.871902 event-place: Uppsala, Sweden.

[27] Felienne Hermans and Marlies Aldewereld. 2017. Programming is writing is programming. In *Companion to the first International Conference on the Art, Science and Engineering of Programming*. 1–8.

[28] Felienne Hermans, Alaaeddin Swidan, and Efthimia Aivaloglou. 2018. Code phonology: an exploration into the vocalization of code. In *Proceedings of the 26th Conference on Program Comprehension*. 308–311.

[29] Richard C. Holt, David B. Wortman, David T. Barnard, and James R. Cordy. 1977. SP/k: a system for teaching computer programming. *Commun. ACM* 20 (1977), 301–309. https://doi.org/10.1145/359581.359586

[30] Henk Huizinga. 2004. *Taal & Didactiek - Stellem* (wolters-noordhoff ed.).

[31] Tobias Kohn. 2017. *Teaching Python Programming to Novices: Addressing Misconceptions and Creating a Development Environment*. PhD Thesis. ETH Zurich.

[32] Tobias Kohn. 2019. The Error Behind The Message: Finding the Cause of Error Messages in Python. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education (SIGCSE '19)*. ACM, New York, NY, USA, 524–530. https://doi.org/10.1145/3287324.3287381 event-place: Minneapolis, MN, USA.

[33] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. 2012. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *2012 34th International Conference on Software Engineering (ICSE)*. 3–13. https://doi.org/10.1109/ICSE.2012.6227211 ISSN: 0270-5257.

[34] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. *IEEE Transactions on Software Engineering* 38, 1 (Jan. 2012), 54–72. https://doi.org/10.1109/TSE.2011.104

[35] J. Paul Leonard. 1930. The Use of Practice Exercises in Teaching Capitalization and Punctuation. *The Journal of Educational Research* 21, 3 (March 1930), 186–190. https://doi.org/10.1080/00220671.1930.10880030

[36] Raymond Lister. 2016. Toward a Developmental Epistemology of Computer Programming. In *Proceedings of the 11th Workshop in Primary and Secondary Computing Education (WiPSCE '16)*. Association for Computing Machinery, New York, NY, USA, 5–16. https://doi.org/10.1145/2978249.2978251 event-place: Münster, Germany.

[37] J. C. Lusth, N. A. Kraft, and J. Tacey. 2009. Language subsetting via reflection and overloading. In *2009 39th IEEE Frontiers in Education Conference*. 1–6. https://doi.org/10.1109/FIE.2009.5350866

[38] Andrew Luxton-Reilly. 2016. Learning to Program is Easy. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '16)*. ACM, New York, NY, USA, 284–289. https://doi.org/10.1145/2899415.2899432 event-place: Arequipa, Peru.

[39] Linda Mannila, Mia Peltomäki, and Tapio Salakoski. 2006. What about a simple language? Analyzing the difficulties in learning to program. *Computer Science Education* 16, 3 (2006), 211–227. https://doi.org/10.1080/08993400600912384

[40] I. T. Chan Mow. 2008. Issues and Difficulties in Teaching Novice Computer Programming. In *Innovative Techniques in Instruction Technology, E-learning, E-assessment, and Education*.

[41] Ulrich Müller, Jeremy I. M. Carpendale, and Leslie Smith. 2009. *The Cambridge Companion to Piaget*. Cambridge University Press. Google-Books-ID: IGggAwAAQBAJ.

[42] Seymour Papert. 1980. *Mindstorms: Children, Computers, and Powerful Ideas*. Basic Books, Inc., New York, NY, USA.

[43] Roy D. Pea. 1986. Language-Independent Conceptual "Bugs" in Novice Programming. *Journal of Educational Computing Research* 2, 1 (Feb. 1986), 25–36. https://doi.org/10.2190/689T-1R2A-X4W4-29J2

[44] Martinha Piteira and Carlos Costa. 2013. Learning Computer Programming: Study of Difficulties in Learning Programming. In *Proceedings of the 2013 International Conference on Information Systems and Design of Communication (ISDOC '13)*. Association for Computing Machinery, New York, NY, USA, 75–80. https://doi.org/10.1145/2503859.2503871 event-place: Lisboa, Portugal.

[45] Scott R. Portnoff. 2018. The introductory computer programming course is first and foremost a language course. *ACM Inroads* 9, 2 (April 2018), 34–52. https://doi.org/10.1145/3152433

[46] Ralph T. Putnam, D. Sleeman, Juliet A. Baxter, and Laiani K. Kuspa. 1986. A Summary of Misconceptions of High School Basic Programmers. *Journal of*

Educational Computing Research 2, 4 (Nov. 1986), 459–472. https://doi.org/10.2190/FGN9-DJ2F-86V8-3FAU

[47] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and Yasmin Kafai. 2009. Scratch: Programming for All. *Commun. ACM* 52, 11 (Nov. 2009), 60–67. https://doi.org/10.1145/1592761.1592779

[48] Eric Roberts. 2001. An Overview of MiniJava. In *Proceedings of the Thirty-Second SIGCSE Technical Symposium on Computer Science Education (SIGCSE '01)*. Association for Computing Machinery, New York, NY, USA, 1–5. https://doi.org/10.1145/364447.364525 event-place: Charlotte, North Carolina, USA.

[49] Carsten Schulte. 2008. Block Model: An Educational Model of Program Comprehension As a Tool for a Scholarly Approach to Teaching. In *Proceedings of the Fourth International Workshop on Computing Education Research (ICER '08)*. ACM, New York, NY, USA, 149–160. https://doi.org/10.1145/1404520.1404535 event-place: Sydney, Australia.

[50] Pratim Sengupta, John S. Kinnebrew, Satabdi Basu, Gautam Biswas, and Douglas Clark. 2013. Integrating computational thinking with K-12 science education using agent-based computation: A theoretical framework. *Education and Information Technologies* 18, 2 (June 2013), 351–380. https://doi.org/10.1007/s10639-012-9240-x

[51] Ben Shneiderman. 1977. Teaching programming: A spiral approach to syntax and semantics. *Computers & Education* 1, 4 (Jan. 1977), 193–197. https://doi.org/10.1016/0360-1315(77)90008-2

[52] Simon. 2011. Assignment and sequence: why some students can't recognise a simple swap. In *Proceedings of the 11th Koli Calling International Conference on Computing Education Research (Koli Calling '11)*. Association for Computing Machinery, Koli, Finland, 10–15. https://doi.org/10.1145/2094131.2094134

[53] Jean Simon. 1973. *La Langue écrite de l'enfant.* Presses universitaires de France. Google-Books-ID: gJRLAAAAYAAJ.

[54] Juha Sorva. 2008. The same but different students' understandings of primitive and object variables | Proceedings of the 8th International Conference on Computing Education Research. In *in Proceedings of the 8th Koli Calling International Conference on Computing Education*. 5–15. https://dl.acm.org/doi/10.1145/1595356.1595360

[55] Andreas Stefik and Richard Ladner. 2017. The Quorum Programming Language (Abstract Only). In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE '17)*. ACM, New York, NY, USA, 641–641. https://doi.org/10.1145/3017680.3022377 event-place: Seattle, Washington, USA.

[56] Andreas Stefik and Susanna Siebert. 2013. An Empirical Investigation into Programming Language Syntax. *Trans. Comput. Educ.* 13, 4 (Nov. 2013), 19:1–19:40. https://doi.org/10.1145/2534973

[57] Alaaeddin Swidan and Felienne Hermans. 2019. The Effect of Reading Code Aloud on Comprehension: An Empirical Study with School Students. In *Proceedings of the ACM Conference on Global Computing Education*. 178–184.

[58] Ivan Tomek, Tomasz Muldner, and Saleem Khan. 1985. PMS—A program to make learning Pascal easier. *Computers & Education* 9, 4 (1985), 205–211. https://doi.org/10.1016/0360-1315(85)90009-0

[59] Ethel Tshukudu and Quintin Cutts. 2020. Semantic Transfer in Programming Languages: Exploratory Study of Relative Novices. In *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '20)*. Association for Computing Machinery, Trondheim, Norway, 307–313. https://doi.org/10.1145/3341525.3387406

[60] Carlos Vega, Camilo Jiménez, and Jorge Villalobos. 2013. A scalable and incremental project-based learning approach for CS1/CS2 courses. *Education and Information Technologies* 18, 2 (June 2013), 309–329. https://doi.org/10.1007/s10639-012-9242-8

[61] Marianne Verhallen and Simon Verhallen. 1994. *Woorden leren, woorden onderwijzen.* CPS.

[62] Centraal Bureau voor de Statistiek Nederland. [n.d.]. StatLine - WO voltijd; rendement en uitval, 1995 - 2005. https://opendata.cbs.nl/statline/#/CBS/nl/dataset/71063ned/table?fromstatweb

[63] WainerJacques and XavierEduardo C. 2018. A Controlled Experiment on Python vs C for an Introductory Programming Course. *ACM Transactions on Computing Education (TOCE)* (Aug. 2018). https://dl.acm.org/doi/abs/10.1145/3152894

[64] David Weintrop, Elham Beheshti, Michael Horn, Kai Orton, Kemi Jona, Laura Trouille, and Uri Wilensky. 2015. Defining Computational Thinking for Mathematics and Science Classrooms. *Journal of Science Education and Technology* 25 (Oct. 2015). https://doi.org/10.1007/s10956-015-9581-5

[65] David Weintrop and Nathan Holbert. 2017. From Blocks to Text and Back: Programming Patterns in a Dual-Modality Environment. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE '17)*. Association for Computing Machinery, Seattle, Washington, USA, 633–638. https://doi.org/10.1145/3017680.3017707

[66] Jeannette M. Wing. 2006. Computational Thinking. *Commun. ACM* 49, 3 (March 2006), 33–35. https://doi.org/10.1145/1118178.1118215

[67] Jooyong Yi, Umair Z. Ahmed, Amey Karkare, Shin Hwei Tan, and Abhik Roychoudhury. 2017. A Feasibility Study of Using Automated Program Repair for Introductory Programming Assignments. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*. ACM, New York, NY, USA, 740–751. https://doi.org/10.1145/3106237.3106262 event-place: Paderborn, Germany.