Gradual Programming in Hedy: A First User Study

Marleen Gilsing, Felienne Hermans
Leiden Institute of Advanced Computer Science (LIACS)
Leiden, the Netherlands
{m.gilsing, f.f.j.hermans}@liacs.leidenuniv.nl

***Abstract*—Recently the 'gradual programming' approach was introduced, which proposes to lower the syntax barrier by starting with a very simple language, and gradually adding both concepts and refining syntax. Hedy is the first language to implement a gradual approach, and this paper presents the first user study on Hedy with 39 children between age 11 and age 14 who followed online lessons for six weeks. Based on lesson observations and a written survey filled out by the participants, we aim to understand the impact of using a gradual language. Our findings show that children appreciate the gradual nature of Hedy, find Hedy easy to learn and especially appreciate the power to control the difficulty of Hedy themselves. They also like and frequently use built-in education features like example code snippets. Challenges of a gradual approach are the fact that commands sometimes change or overlap, and remembering commands and specific syntax remain a challenge. According to the participants, improvements could be made by making Hedy less sensitive to syntax errors, by improving error messages and by localizing keywords to the native language of children.**

I. Introduction

Computer science programs suffer from high dropout rates, as high as 40% [1], 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 [2].

It has been hypothesized that the high dropout rate is due to the current instructional techniques that are used [1] and expectations that are set too high [3]. Current introductory programming courses might 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. For example, Mow states that the precision that is needed while programming requires *"a level of attention to detail that does not come naturally to human beings"* [4].

To specifically address issues with syntax when learning to program, we have recently coined the idea of a *gradual* programming language, a language that teaches syntax in steps, rather than at once [5]. As is common in both mathematics and natural language teaching, learners using a gradual language initially learn incomplete and partly incorrect models, which are refined step by step.

The programming language Hedy is an implementation of the idea of gradual programming. Hedy is an open-source programming language that runs in the browser and is available for free. In previous work, we have manually examined almost 10.000 Hedy programs to gain an understanding of how novices learn with Hedy [5].

In this paper we examine Hedy in two classes in the Netherlands. The lessons were recorded, and participants filled out an open text survey after the 12 lessons. Participants' answers and videos were coded using thematic analysis [6] in order to gain insights into the participant's experiences with Hedy and to answer our three research questions:

1) What are benefits of a gradual programming approach?
2) What are challenges of a gradual programming approach?
3) How can gradual programming approaches such as Hedy be improved?

Our study firstly shows that novices appreciate the fact that Hedy works in a gradual way and is easy to learn. They also like the fact that they have control over the difficulty of Hedy and that built-in explanations are available. There are some aspects of programming that Hedy helps with, but Hedy does not remove obstacles entirely; participants in our study still struggle with remembering the right commands and producing correct syntax. Improvements might lie in a less sensitive language that allows different syntax combinations, better error messages and the localization of keywords.

II. Background

A. Issues with learning syntax

Learning syntax is a well-known issue in learning to program. Denny *et al.* for example 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 [7]. Altadmri and Brown analyzed 37 million compilations by 250.000 students and found that the most common error is a syntax error: mismatched brackets, which occurred in almost 800.000 compilations [8]. 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 [9]. Interestingly enough, students assess learning syntax as more problematic than teachers [10], which might shed some light on why little effort is given to the explicit explanation of syntax in many programming classrooms.

B. *Gradual learning in natural languages*

When learning a first language, novices do not learn syntax, punctuation and capitalization at once. Initially, they only write letters in lowercase. Only in later stages, learners will learn to add uppercase, periods, commas and semicolons. This gradual form of teaching works well because young children are so occupied with writing letters, that they cannot concentrate yet on other aspects of writing, like punctuation and story-lines [11]. This means that the children operate at a low *Piagetian stage*, where they cannot oversee larger problems and can only focus on small steps.

Many modern researchers believe that when novices are learning a new skill they will operate at a low Piagetian stages, irrespective of their age [12], [13]. This notion has also gained popularity in Computer Science education; Lister argues that the cognitive load of students while programming might be too high when simultaneous learning of programming concepts, syntax and problem solving [14]. This might cause novice programmers to behave similarly to younger children learning to write.

Since learning a programming language also shares significant characteristics with learning a natural language—learners in both fields 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 [15], [16].

C. *Related approaches for teaching novices*

Various approaches have been tried to make learning to program easier for novices. Previous work has typically classified educational programming languages into three different categories: 1) mini or toy languages specifically aimed at teaching, such as Scratch [17] or LOGO [18], 2) sub-languages of full programming languages like MiniJava [19] and 3) incremental languages which start with a subset of a language and incrementally add new concepts, such as DrScheme [20].

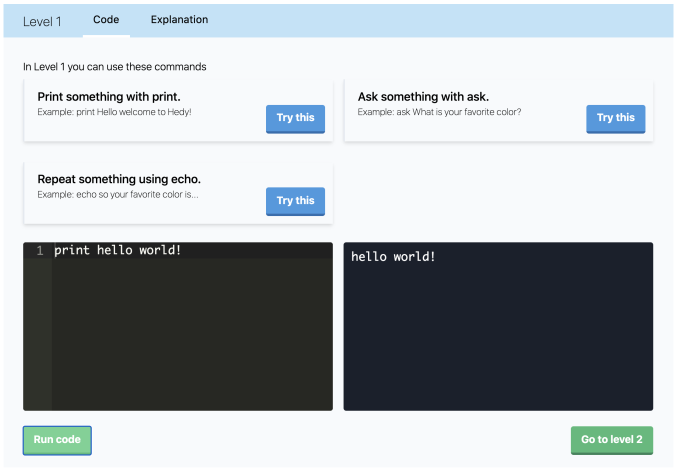
Of these three approaches, incremental languages are most similar to Hedy. The biggest difference between Hedy and incremental languages however is the fact that in Hedy, syntax is also gradual. That means that syntax changes with the levels and becomes more powerful and more complex, e.g. printing at level 1 is done with `print hello world` while at level 3 quotation marks become mandatory, so `print 'hello world'` will need to be used.

III. Hedy

Hedy is the first implementation of the idea of gradual programming. Currently Hedy is implemented in Python, using the Lark parser.[1] 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 run locally, but also has a web version in which code can be simply typed in the browser, as common in modern web-based IDEs for teaching such as repl.it and Trinket. Hedy's code base is open source and available on GitHub.[2]

A. *User Interface*

The current implementation of Hedy is shown in Figure 1. 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!
```

Fig. 1. Level 1 of the Hedy user interface in English

B. *Educational features*

In addition to the gradual approach, Hedy has features that go beyond gradual programming, but also aim at making learning easier. These features are not necessarily part of the gradual programming language paradigm, but are enabled by a gradual language.

1) A palette of available commands: Block-based languages like Scratch, Snap! and App Inventor support a feature called the *palette* which is an overview of all possible blocks, which can be dragged into the programming field [21], [22], [23]. The palette has been frequently names by students as "*as a feature that made it easy to use*" [24]. While not a block-based language, Hedy also aims to make its possibilities discoverable by users inside of the interface. This saves the users the cognitive effort of looking up commands. We know from prior work that looking up information in a separate place increases cognitive load [25].

The Hedy user interface thus contains an overview of the commands available in a level, integrated into the editor, as shown in Figure 1. Each of the commands has a 'Try this' button, which will place a code snippet containing that command in the editor. The demo code is not executed automatically; the user can still adapt it before running the code.

The palette is a feature that is commonly associated with block-based languages, but in principle, textual languages like

[1]https://github.com/lark-parser/lark

[2]https:www.github.com/felienne/Hedy

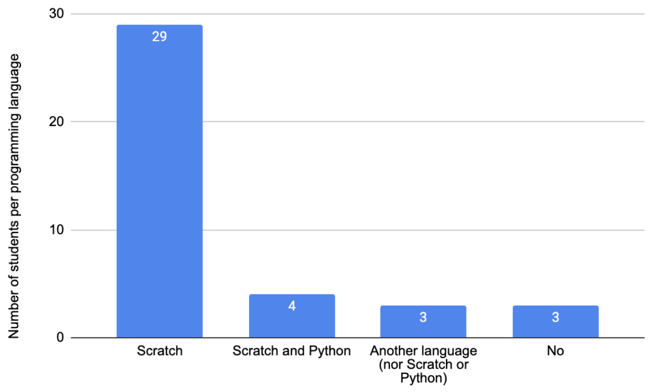


Fig. 2. Languages the participants had experience with

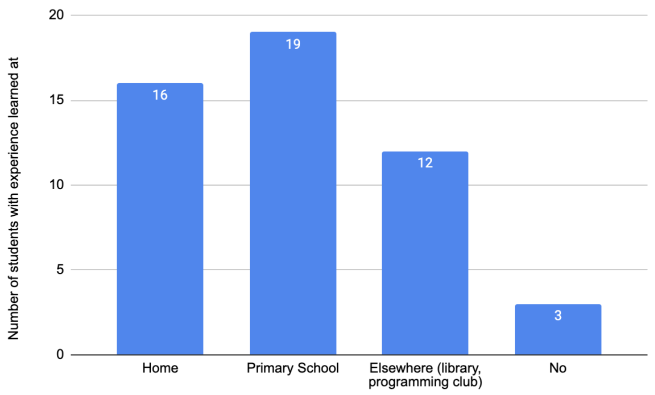


Fig. 3. Places where the participants gained programming experience

Python of JavaScript could also offer a palette to their users with possible code snippets. However, this is complicated by the vast number of options that would be possible. What language features would be shown, and in what exact form? Because the Hedy language is initially small, creating a palette is more straight-forward.

2) Built-in explanations: In addition to the palette of potential code snippets, Hedy also contains built-in videos and exercises for each level, which the user can open and close within the user interface. Again, an integrated tutorial is not a feature that is necessarily part of the gradual language paradigm, yet is enabled by the small size of each level, which leads to brief explanations in a clear order.

IV. Research setup

The goal of the paper is to understand the benefits and challenges of the gradual programming language approach, and explore how it can be further improved. To that end, 39 seventh-graders followed 12 hours of Hedy lessons covering the first 6 levels of Hedy. After these lessons, the participants filled out a written survey on their experiences.

A. Participants

In total, 39 children participated in our study, all seventh graders, from two different classes in one school in the Netherlands.

1) Prior experience: From the 39 children in our study, 36 had prior experience with programming. Figure 2 shows an overview of the programming language the participants were familiar with before the study. Most children (29 out of 36 with experience, or 80%) had experience with Scratch, hence no experience with textual languages.

The children acquired their previous experience at various different sources, as shown in Figure 3. Note that the total in this graph is more than 39, since some children acquired programming experience in, e.g. elementary school and at home and are thus represented in Figure 3 twice.

2) Age: Out of the 39 participants, 38 disclosed their age and gender. The average age of these 38 participants is 12.8, and the age distribution can be seen in Figure 4.

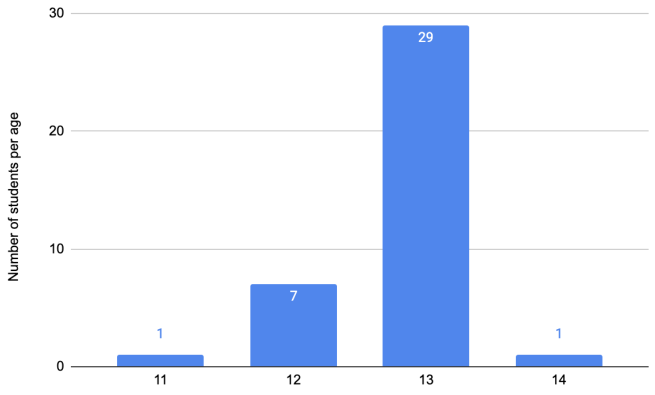


Fig. 4. Ages of participants in the study

3) Gender: The 38 students that disclosed gender were 22 boys and 16 girls, as shown in Figure 5.

B. Lessons

Participants all followed 12 online Hedy lessons during 6 consecutive weeks, each lesson of one hour. All lessons were conducted online and given by a Hedy guest teacher who is

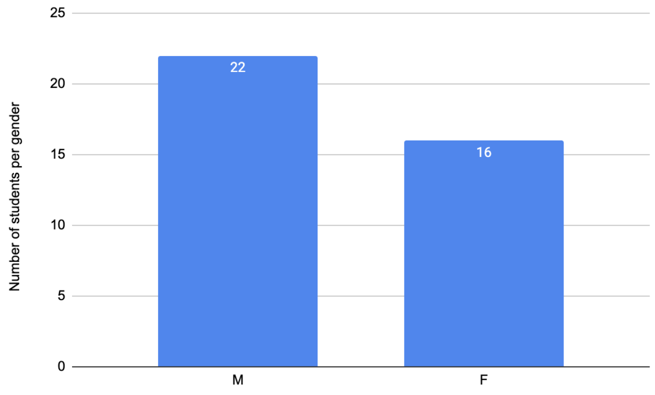


Fig. 5. Gender of students in the study

trained and employed as a teacher, but is not a regular teacher at the school. The teacher was not previously acquainted with the children in the study.

Each week of lessons has a similar setup, in which the students followed instruction in the first lesson of the week. In the instruction, the guest teacher explained the basic concepts of a level, and gave students an assignment to work on. After this instruction part of the lesson, lasting about 15 minutes, the students would be allowed to work on Hedy and the assignment independently. If students ran into issues, they could ask questions with voice or using the chat, and share their screen where needed. In the second lesson of the week, children were allowed to continue working on their assignments and could again ask questions where needed.

Videos of the lessons were recorded, including the screens of the teacher when they were teaching, and the screens of students when they were screen sharing, but without the video of the learners, for privacy reasons.

C. Survey

After the 12 lessons were completed, students were asked to fill out a written open text survey, consisting of these questions:

1) Q1 The greatest thing I created with Hedy was...
2) Q2 What I liked most about Hedy was...
3) Q3 The hardest thing about Hedy was...
4) Q4 If I could change one thing about Hedy, it would be...
5) Q5 The thing I like most about programming is...

In addition to these questions, demographic information (gender and age) was collected and the information about prior experience with programming as presented in Section IV-A1.

D. Research questions

The goal of this paper is to gain a deeper understanding of how gradual programming and Hedy can support children in learning to program, and how both the idea of gradual programming, and the implementation of this idea into Hedy can be improved.

We make the deliberate choice here to separately evaluate the concept of gradual programming itself from the specific implementation into Hedy. We do this so we can both gain a deeper understanding of the benefits and downsides of a gradual language, but also inform the creators of Hedy and of future gradual programming approaches in deciding on trade-offs in their implementations.

As such, our research questions are:

1) What are benefits of a gradual programming approach?
2) What are downsides of a gradual programming approach?
3) How can gradual programming approaches like Hedy be improved?

E. Approach

The answers of the learners to the open questions, and the videos of the lessons were coded by the authors using thematic analysis [6] to answer the three research questions. Firstly, both the written survey data and the video observations were processed and quotations were coded. Following the initial coding process, the codes were grouped into themes for each research question.

V. Results

The goal of this paper is to understand what the benefits and challenges of gradual programming in Hedy are. In this section we answer our three research questions based on participants' survey answers and our lesson observations.

A. What are benefits of a gradual programming approach?

In the participants' answers to the survey, we distinguish four different benefits of the gradual programming approach.

1) Gradual learning: Asked about what they liked best about Hedy (Q2), two children specifically mention the gradual nature of Hedy as a benefit. L31 states that the levels "*get increasingly hard and become a real challenge*", while L37 says that the levels "*get harder and form a step by step guide*". We also saw in the lesson observations that children in the earlier levels were very focused on building programs and getting to know the mechanics of programming. For example, L12 at one point remarked in an early lesson "*ah, the computer can remember my answers!*". The fact that there was no struggle with syntax yet, meant that there was cognitive load left to fully focus on the workings of computers.

2) Easy to use: Five children mention that what they liked best (Q2) was that Hedy is easy to use. L9 states Hedy enables them to create exciting programs, while specifically expressing they aren't good at "*the programming world*". We see this as a testament to the ease of use of Hedy.

In the lesson observations, we saw that learners could create programs that engaged them in programming at early levels. For example L12 built a program in Level 2 that simulates a soccer themed fortune teller as follows (text translated from Dutch):

```
print I am Hedy the fortune teller
question is ask Who will win the Soccer
Cup?
print you think it will be: question
answers is win, not win
print they will answers at random
```

This program will print "you think it will be" followed by the answer of the users, and then either "they will win" or "they will not win". Building a similar program in Python would entail creating a list, importing the random module and dealing with a dozen quotation marks and brackets.

L39 created a song with repetition in Level 5 as follows:

```
print 'the wheels on the bus go'
repeat 3 times 'round and round'
print 'the wheels on the bus go'
repeat 3 times 'round and round'
```

```
print 'All trough the town'
```

In Python, using basic repetition without an iterator variable requires the use of `for i in range(3):` and correct indentation, which are hurdles for novices. In Hedy, learners can focus on the use of the concept of repetition and its application.

3) Control over difficulty: One aspect that participants explicitly stated that they liked was the fact that they had control over the difficulty of the exercises. In the later lessons, we allowed children to explore higher levels that we had not yet explained, while others remained at a lower level than the current lesson. The original Hedy paper [5] envisioned the fact that different children within a classroom could work on similar assignments at different levels, and it is great to see that hypothesis in action.

Five children responded with answers along those lines. For example L25 answered on Q2 that they liked "*being allowed to work at my own pace and level*", while L20 answered on Q5, on what is best about programming, that they could "*go to a different level and challenge yourself*".

4) Palette and explanations: As outlined in Section III-B, Hedy has features that go beyond gradual programming. These features were seen as helpful; two learners pointed at these features when asked for the best thing about Hedy (Q2). L13 said that they liked the 'try this' buttons best, and L26 mentioned the explanations as being the best aspect of Hedy.

B. What are challenges of a gradual programming approach?

From the participants survey answers, we gather three challenges of gradual programming.

1) Remembering commands: One of the design goals of gradual languages is to make the learning of both syntax and concepts easier. A gradual approach lowers cognitive load and thus should allow learners to retain more information in their long-term memory [25], [26], [27]. Despite this goal, learners still find it challenging to remember the different commands and their correct application. Six participants in the study state they struggled with "*remembering codes*"(L7, L13 L25, L37 and L38), one stated they struggled with "*remembering code and quotation marks*" and one participant found it hard to remember "*the right order of codes*" (L24).

2) Syntax issues: Despite the easier design of the syntax of Hedy as compared to other languages used by novices such as Python, learners still found syntax in Hedy hard to use. Five participants named syntax issues as a struggle. Two named syntax as the hardest thing in working with Hedy (Q3), saying that the hardest thing is "*when you forget a single quote and it stops working*"(L16) while L18 named the use of quotes in general as the hardest thing. Being asked what can be improved (Q4), L15 and L32 suggest to get rid of the punctuation altogether, while L18 suggests that we change the syntax of Hedy, removing quotation marks.

One additional learner (L27) suggests as biggest improvement (Q4) to add an auto-correct feature to Hedy which fixes programs with errors. While this learner does not directly mention syntax as an issue, the underlying sentiment is that Hedy programs are hard to write correctly.

3) Changes to commands: A downside of a gradual language is the fact that commands can change over the course of levels. This was confirmed by the participants in the current study, five of which indicated that they found the changes difficult. L12 suggested to get rid of the levels altogether as the biggest improvement possible to Hedy (Q4), while L11 said it would be best to limit the changes. Asked what was the hardest about working with Hedy (Q3), L6 said that the hardest thing was that things kept changing, and L15 specifically said level 3 was a big leap where "*everything was different all of a sudden*". L39 stated that commands kept changing over the levels, resulting in "*a need to make the switch all the time*".

C. How can gradual programming languages like Hedy be improved?

We distinguish three different directions in which gradual programming approaches like Hedy can be improved.

1) Less sensitive to errors: Four learners indicate that they want Hedy to be less sensitive. L26 and L29 indicate that the hardest thing about using Hedy (Q3) was that it is so sensitive and you have to be so precise. Hedy's sensitivity was also named as the number one thing to improve (Q4), saying we should change Hedy so that "*it is less sensitive*"(L17) and "*it is less strict*" (L20).

The focus on sensitivity in an interesting one. We are not aware of other work on programming education in which learners specifically comment on the *sensitivity* of languages. We assume these observations are connected to the core idea of gradual programming. In traditional languages, like Python or C++, syntax is introduced form the beginning, so learners have to accept the fact that languages are sensitive and precise from the beginning.

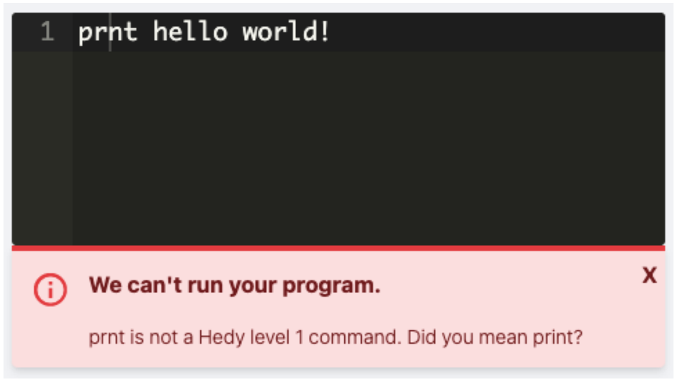
In Hedy however, the preciseness of programming is deferred to later in the learning sequence. At level 1, Hedy has no syntactic elements apart from the three keywords: ask, print and echo that may be followed by any string, including strings with quotation marks. While Hedy is also 'sensitive' at level 1, e.g. a user cannot misspell a keyword, in earlier levels we did not observe children expressing their frustration with sensitivity. It was only at later levels, especially at level 3, where quotation marks are introduced, that children start to have issues with Hedy being seen as 'sensitive' or 'picky' about syntax. One learner (L31) specifically indicates level 3 as the level where things started to get difficult.

These difficulties might indicate that the steps between the levels might be too big, and children are not ready for more complex syntax. One could for example imagine that the introduction of an intermediate level 2.5 in which quotation marks are allowed, but not necessary, so learners can get used to the syntax without being forced to use it correctly.

In addition to the quotation marks being hard to get right, we also noticed that children did not always see their value. They sometimes expressed a desire to go back to the way things were in level 2, because that was a lot easier. Our lesson plan stresses that the value of quotations lies in the fact that it allows variable names to also be printed as strings; in level 2 one cannot print the sentence "your name is Hedy" if a variable name is declared beforehand. In the lesson plan, we show that the following code works, but prints "your Hedy is Hedy":

```
name is Hedy
print your name is name
```

While we explained that the value of quotation marks is to distinguish between variable names and plain text, this was not convincing enough for some learners to warrant the extra effort of putting in quotation marks. Some learners came up with the solution of renaming a variable in case of a conflict with a string, which is obviously a reasonable solution from their perspective. Our additional argument that other textual programming languages also need quotations seemed not to convince learners also. Hence the solution might be in changing the explanation around the use of syntax.



```
prnt hello world!
```

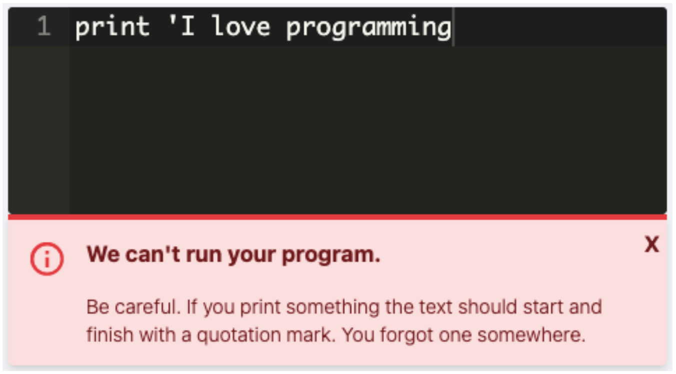
We can't run your program.

prnt is not a Hedy level 1 command. Did you mean print?

Fig. 6. Misspelling a keyword in level 1 produces an error message that is relatively easy to understand.

The sensitivity issues might also be related to error messages. Error messages regarding to the misspelling of keywords, as illustrated by Figure 6 are more precise because it is easier to generate both the cause and a solution.

In errors related to a missing quotation mark, it is harder to pinpoint the code issue and suggest a concrete fix, e.g. in a program like the one shown in Figure 7 we can only detect that there are mismatched quotation marks, and remind children to being and end with a quotation mark.



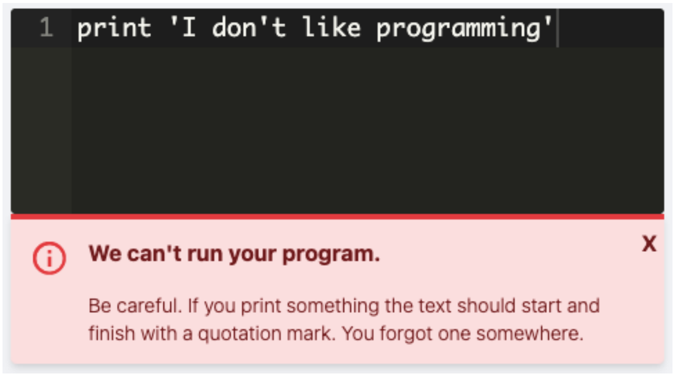
```
print 'I love programming
```

We can't run your program.

Be careful. If you print something the text should start and finish with a quotation mark. You forgot one somewhere.

Fig. 7. Mismatched quotation marks in level 3 produces an error message when a quotation mark is not matched.

While that is a fine message for a program like the one in Figure 7, the same error is produced when a single quotation mark is used inside of a string, as demonstrated by Figure 8. We have seen children get frustrated by this error message, because the string here **does** begin and end with a quotation mark. This is the type of situation in which children would express the opinion that Hedy is "*so sensitive*".



```
print 'I don't like programming'
```

We can't run your program.

Be careful. If you print something the text should start and finish with a quotation mark. You forgot one somewhere.

Fig. 8. A string containing a single quotation mark in level 3 produces an error message that can be confusing.

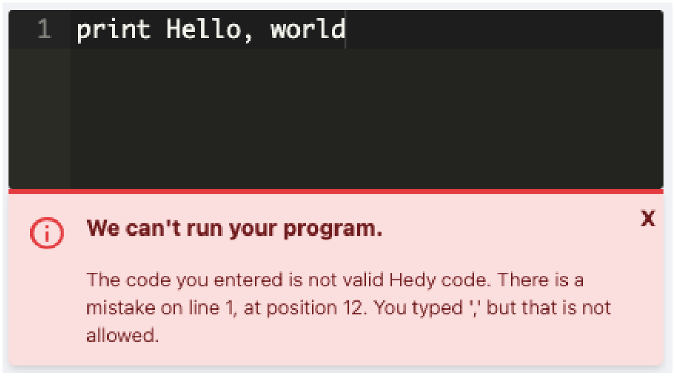
2) Better error messages: Four children mention error messages as a potential area that could be improved. In particular children indicate that they want to have "*more specific error messages*" (L24), "*error messages that are clearer*" (L38) and error messages that "*give a better indication of what you did wrong*" (L25) or "*help kids in fixing their problems*" (L31).

While observing the lessons, we saw something interesting regarding how young novices read error messages. Firstly, we noticed that children often look at error messages, but find them hard to read. This confirms earlier work on error messages, e.g. the work by Barik which showed that novice programmers read the error messages[28], but struggle with them more than experts [29].

Our experiment sheds new light on error messages also. In particular, we noticed that even when encouraged by the teacher to read error messages, often children did not understand them fully. Upon further investigation, we noticed a potential cause of the lack of understanding of error messages.

To gain a deeper insight into the understanding that participants had of Hedy error messages, we asked them to read the messages aloud to us. In prior work, we asked children of the same age group to read Python code aloud [30], [31] and found that the practice of reading aloud can reveal interesting patterns and misconceptions.

When we asked children to read error messages aloud, we noticed they would often skip punctuation characters.



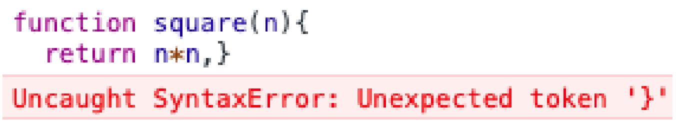
```
print Hello, world
```

We can't run your program.

The code you entered is not valid Hedy code. There is a mistake on line 1, at position 12. You typed ',' but that is not allowed.

Fig. 9. Error messages in Hedy refer to characters, such as the comma here, by symbol, and place them between quotation marks.

Error messages in Hedy refer to characters such as comma, period and colon with its symbol, rather than with its name. An error message referring to a comma will use the symbol , not the word *comma*. To indicate that the symbol comma is meant, the symbol is placed between quotation marks. An example of this is shown in Figure 9. This is not an unusual choice, in fact, this is how most languages present characters, see also Figure 10 that shows a similar error message in JavaScript as shown by Chrome.



```
function square(n){
  return n*n,}
Uncaught SyntaxError: Unexpected token '}'
```

Fig. 10. Error messages in JavaScript also refer to characters, such as the clsing curly brace here, by symbol, and place them between quotation marks.

When asking children to read these messages aloud, we noticed they would skip these symbols entirely. For the error message shown in Figure 9, for example, some participants would read “You typed —pause— but that is not allowed” interpreting the comma as a comma in the message. Other participants read “You *typed* but that is not allowed”, interpreting the message as if typing code itself was not allowed. They would then express confusion about typing what exactly was not allowed. Other participants assumed that because the error message was malformed, e.g. does not look like a proper sentence, the error in the program caused the error message to also ‘glitch’.

In hindsight, this is not unreasonable behaviour. Children (and adults) are trained their lives to not explicitly read punctuation, so why change that now? Novices at this level have not really learned that quotation marks around a text mean that you should interpret that text as the string and not its meaning. This is especially true in Hedy where quotation marks are not introduced until level 3. However, we saw this behaviour also after quotation marks were introduced, showing that this knowledge in programming syntax does not necessarily transfer to understanding that in error messages too a quote means a literal string value.

We suspect that error messages in other languages, such as the message in Figure 10 will confuse novices in similar ways, although that of course warrants further research.

3) Localized keywords: In the final evaluation, one learner (L2) indicated as answer to Q4 that Hedy could be improved with the adoption of keywords in the native language of the users. This is a sentiment we heard in the lessons themselves also. Some learners asked in the lessons why the keywords were not in Dutch, as the interface, lessons and some parts of the example programs (such as sentences to print and variable names) were written in Dutch.

VI. Discussion

A. Learning to program is hard

Despite Hedy’s small steps, simple syntax and extensive explanation, 12 participants in our study still express that they find learning to program hard. In addition to the participants indicating issues with syntax and with remembering code, there was one more participant who suggested to make Hedy ‘easier’ as the biggest potential improvement, without specifying more detailed issues.

From the lesson observations, we know that some of the issues that these participants describe could be improved by addressing some of the challenges described earlier in the paper, such as more intermediate levels, explicit error messages and localized keywords. Some issues however are of a more conceptual nature. Learning the principles of programming is hard, and some confusion and frustration cannot be avoided. For example, some children in our study showed well-known programming misconceptions, such as assuming that a computer is ‘smart’ and can thus fix syntax problems, or can guess the intention of a programmer, saying things like “*why can’t Hedy guess that I meant to include a print code there?*”

B. Localized keywords and gradual de-localization

Some other educational languages do allow for keywords to be presented in the language of the learner, including Scratch [17] and Snap! [23]. However, those languages are block-based and as such it is a bit easier to localize keywords since there is no parsing involved. It can be quite cumbersome to localize keywords in textual languages because that would entail localized versions of the grammar, but for a gradual language this will be easier because of the small set of keywords. Also, the keywords could start out in the local language of a learner and gradually switch over to English to enable a smoother transition into Python or other mainstream languages.

C. Use of multiple correct syntaxes

The fact that many learners in our study found Hedy too strict points in an interesting direction for future development. In most programming languages, there is one correct way to format a command, while other ways are not allowed. There are some exceptions, for example Python allows both single and double quotes around strings. Python however allows that with the aim of make escaping a bit less cumbersome, not with the goal of being a lenient language.

A gradual language however could allow multiple versions of syntax, especially at lower lever where its grammar will have a small set of production rules. Hedy for example, could allow for strings with and without quotes at level 3, which is the level where quotes are first introduced. Gradually, this could be replaced by allowing strings without quotes but giving a warning to not allowing it at higher levels. This way kids can get used to the syntax and learn it, without being frustrated by errors.

D. Better alignment with operators in math class

In level 6, Hedy introduces calculations, e.g. this code is valid Hedy level 6 code: `print '5 times 5 is '5 * 5`. The creators of Hedy decided to use the standard programming syntax: * for multiplication and / for division. In these lessons, we realized that might be too far removed from what children aged 11 to 14 are used to. When we introduced the syntax for calculations, several children expressed surprise that $\times$ or x were not used for multiplication. From their perspective of course, this is a reasonable request, if $\times$ is used for multiplication in mathematics class, and even on calculators, why wouldn't programming languages use it, or use the letter x, clearly the most similar thing on the keyboard? Similarly, the participants were expecting : or $\div$ for division.

Clearly there are sensible reasons that these symbols are not used in traditional programming languages; $\times$ is not present on most keyboards and using a letter like x as an operator can lead to parsing issues. However, for an educational language like Hedy, it could be reasonable to use these less confusing characters. Later levels could introduce the proper Python operators, and we could even consider to allow both $\times$ and x for a few levels, as proposed in Section VI-C, before mandating only * and /.

E. Program repair

The suggestion of auto-correct leads to some interesting new challenges. While the problem of automatically repairing programs is hard in the general case, a lot of progress has been made in recent years [32], [33], [34]

For a small language like Hedy, there are a number of directions to explore. Firstly, random edits like inserting quotes or replacing words by legal keywords might reach valid programs quickly. Secondly, it could be feasible to create a small set of canonical programs at each level, and rather than suggesting an edit to an erroneous program, suggest a working program with similar features for the learner to adapt to their own use case.

F. More focus on memorization might be needed

Some participants specifically name the memorization of commands to be hard. This is especially interesting since the Hedy user interface contains a palette in which the new commands of the level are shown. While we saw learners use the buttons initially, when engaged in an exercise the learners often stopped using the buttons and struggled to recall the way to create certain commands. Future gradual programming languages, and also lessons plans using gradual languages, could consider mixing programming exercises with specific exercises aimed at strengthening memory, such as retrieval practice, before moving on to new concepts.

VII. Concluding remarks

Gradual programming is a teaching approach that starts with a simple syntax and gradually adds both concepts and more complex syntax. The goal of this paper is to understand the benefits and challenges of the gradual programming language approach, and explore how it can be further improved. We have therefore taught 39 children aged 11 to 14 for six weeks using Hedy, to answer our research questions:

1) What are benefits of a gradual programming approach?
2) What are challenges of a gradual programming approach?
3) How can gradual programming approaches such as Hedy be improved?

Our findings show that children appreciate the gradual nature of Hedy, find Hedy easy to learn and especially appreciate that they have control over the difficulty of Hedy. They also like and use the built-in education features like example code snippets (RQ1). Challenges of a gradual approach are the fact that commands sometimes change. We also find that despite the small steps of Hedy, remembering commands and specific syntax remain a challenge for learners (RQ2). According to the participants, improvements could be made by making Hedy less sensitive to syntax errors, by improving error messages and by localizing keywords to the native language of children (RQ3).

The current research gives rise to a number of future directions. Firstly, Hedy can be improved improved with the lessons we learned from the current study. Secondly, the current version of Hedy consists of 15 levels, so a replication of the current work on a larger number of levels can give us valuable insights into the working of Hedy over a longer period of time, covering more concepts. Once Hedy is more properly tested and evaluated, a controlled experiment comparing Hedy to Python for introductory programming could be run.

References

[1] T. Beaubouef and J. Mason, "Why the High Attrition Rate for Computer Science Students: Some Thoughts and Observations," *SIGCSE Bull.*, vol. 37, no. 2, pp. 103–106, Jun. 2005. [Online]. Available: http://doi.acm.org/10.1145/1083431.1083474

[2] C. B. voor de Statistiek Nederland, "StatLine - WO voltijd; rendement en uitval, 1995 - 2005." [Online]. Available: https://opendata.cbs.nl/statline//CBS/nl/dataset/71063ned/table?fromstatweb

[3] A. Luxton-Reilly, "Learning to Program is Easy," in *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*, ser. ITiCSE '16. New York, NY, USA: ACM, 2016, pp. 284–289, event-place: Arequipa, Peru. [Online]. Available: http://doi.acm.org/10.1145/2899415.2899432

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

[5] F. Hermans, "Hedy: A Gradual Language for Programming Education," in *Proceedings of the 2020 ACM Conference on International Computing Education Research*, ser. ICER '20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 259–270, event-place: Virtual Event, New Zealand. [Online]. Available: https://doi.org/10.1145/3372782.3406262

[6] V. Braun and V. Clarke, "Reflecting on reflexive thematic analysis," *Qualitative Research in Sport, Exercise and Health*, vol. 11, no. 4, pp. 589–597, Aug. 2019, publisher: Routledge _eprint: https://doi.org/10.1080/2159676X.2019.1628806. [Online]. Available: https://doi.org/10.1080/2159676X.2019.1628806

[7] P. Denny, A. Luxton-Reilly, E. Tempero, and J. Hendrickx, "Understanding the syntax barrier for novices." ACM, Jun. 2011, pp. 208–212. [Online]. Available: http://dl.acm.org/citation.cfm?id=1999747.1999807

[8] A. Altadmri and N. Brown, "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*, pp. 522–527, 2015.

[9] A. Stefik and S. Siebert, "An Empirical Investigation into Programming Language Syntax," *Trans. Comput. Educ.*, vol. 13, no. 4, pp. 19:1–19:40, Nov. 2013. [Online]. Available: http://doi.acm.org/10.1145/2534973

[10] M. Piteira and C. Costa, "Learning Computer Programming: Study of Difficulties in Learning Programming," in *Proceedings of the 2013 International Conference on Information Systems and Design of Communication*, ser. ISDOC '13. New York, NY, USA: Association for Computing Machinery, 2013, pp. 75–80, event-place: Lisboa, Portugal. [Online]. Available: https://doi.org/10.1145/2503859.2503871

[11] U. Müller, J. I. M. Carpendale, and L. Smith, *The Cambridge Companion to Piaget*. Cambridge University Press, Aug. 2009, google-Books-ID: IGggAwAAQBAJ.

[12] A. Demetriou, M. Shayer, and A. Efklides, *Neo-Piagetian Theories of Cognitive Development: Implications and Applications for Education*. Routledge, Jul. 2016, google-Books-ID: IZSkDAAAQBAJ.

[13] R. Case, "Neo-Piagetian theories of child development," in *Intellectual development*. New York, NY, US: Cambridge University Press, 1992, pp. 161–196.

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

[15] S. R. Portnoff, "The introductory computer programming course is first and foremost a language course," *ACM Inroads*, vol. 9, no. 2, pp. 34–52, Apr. 2018. [Online]. Available: https://doi.org/10.1145/3152433

[16] F. Hermans and M. Aldewereld, "Programming is writing is programming," in *Companion to the first International Conference on the Art, Science and Engineering of Programming*, 2017, pp. 1–8.

[17] M. Resnick, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman, and Y. Kafai, "Scratch: Programming for All," *Commun. ACM*, vol. 52, no. 11, pp. 60–67, Nov. 2009. [Online]. Available: http://doi.acm.org/10.1145/1592761.1592779

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

[19] E. Roberts, "An Overview of MiniJava," in *Proceedings of the Thirty-Second SIGCSE Technical Symposium on Computer Science Education*, ser. SIGCSE '01. New York, NY, USA: Association for Computing Machinery, 2001, pp. 1–5, event-place: Charlotte, North Carolina, USA. [Online]. Available: https://doi.org/10.1145/364447.364525

[20] M. Felleisen, R. Findler, M. Flatt, and S. Krishnamurthi, "The TeachScheme! Project: Computing and Programming for Every Student," *Computer Science Education*, vol. 14, pp. 55–77, 2004.

[21] D. Wolber, H. Abelson, and M. Friedman, "Democratizing Computing with App Inventor," *GetMobile: Mobile Computing and Communications*, vol. 18, no. 4, pp. 53–58, Jan. 2015. [Online]. Available: https://doi.org/10.1145/2721914.2721935

[22] S. C. Pokress and J. J. D. Veiga, "MIT App Inventor: Enabling Personal Mobile Computing," *arXiv:1310.2830 [cs]*, Oct. 2013, arXiv: 1310.2830. [Online]. Available: http://arxiv.org/abs/1310.2830

[23] M. Ball, J. Mönig, B. Romagosa, and B. Harvey, "Snap! A Look at 5 Years, 250,000 Users and 2 Million Projects," in *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, ser. SIGCSE '19. New York, NY, USA: Association for Computing Machinery, Feb. 2019, p. 1279. [Online]. Available: https://doi.org/10.1145/3287324.3293863

[24] D. Weintrop and U. Wilensky, "To block or not to block, that is the question: students' perceptions of blocks-based programming," in *Proceedings of the 14th International Conference on Interaction Design and Children*, ser. IDC '15. New York, NY, USA: Association for Computing Machinery, Jun. 2015, pp. 199–208. [Online]. Available: https://doi.org/10.1145/2771839.2771860

[25] F. Paas and J. J. G. van Merriënboer, "Cognitive-Load Theory: Methods to Manage Working Memory Load in the Learning of Complex Tasks," *Current Directions in Psychological Science*, vol. 29, no. 4, pp. 394–398, Aug. 2020, publisher: SAGE Publications Inc. [Online]. Available: https://doi.org/10.1177/0963721420922183

[26] A. V. Robins, L. E. Margulieux, and B. B. Morrison, "Cognitive Sciences for Computing Education," in *The Cambridge Handbook of Computing Education Research*, ser. Cambridge Handbooks in Psychology, A. V. Robins and S. A. Fincher, Eds. Cambridge: Cambridge University Press, 2019, pp. 231–275. [Online]. Available: https://www.cambridge.org/core/books/cambridge-handbook-of-computing-education-research/cognitive-sciences-for-computing-education/319D706EF1A2E8D6A6B8EA7697CE5BE2

[27] F. Hermans, *The Programmer's Brain: What every programmer needs to know about cognition*. S.l.: Manning Publications, Sep. 2021.

[28] T. Barik, J. Smith, K. Lubick, E. Holmes, J. Feng, E. Murphy-Hill, and C. Parnin, "Do Developers Read Compiler Error Messages?" in *Proceedings of the 39th International Conference on Software Engineering*, ser. ICSE '17. IEEE Press, 2017, pp. 575–585, event-place: Buenos Aires, Argentina. [Online]. Available: https://doi.org/10.1109/ICSE.2017.59

[29] B. A. Becker, P. Denny, J. Prather, R. Pettit, R. Nix, and C. Mooney, "Towards Assessing the Readability of Programming Error Messages," in *Australasian Computing Education Conference*. New York, NY, USA: Association for Computing Machinery, 2021, pp. 181–188. [Online]. Available: https://doi.org/10.1145/3441636.3442320

[30] F. Hermans, A. Swidan, and E. Aivaloglou, "Code Phonology: An exploration into the vocalization of code," in *Proceedings of the 26th Conference on Program Comprehension, ICPC 2018*. Association for Computing Machinery (ACM), 2018, pp. 308–311. [Online]. Available: https://research.tudelft.nl/en/publications/code-phonology-an-exploration-into-the-vocalization-of-code

[31] A. Swidan and F. Hermans, "The Effect of Reading Code Aloud on Comprehension: An Empirical Study with School Students," in *CompEd'19 : Proceedings of the ACM Conference on Global Computing Education*. Association for Computing Machinery (ACM), May 2019, pp. 178–184. [Online]. Available: https://research.tudelft.nl/en/publications/the-effect-of-reading-code-aloud-on-comprehension-an-empirical-st

[32] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "GenProg: A Generic Method for Automatic Software Repair," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 54–72, Jan. 2012.

[33] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, "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)*, Jun. 2012, pp. 3–13, iSSN: 0270-5257.

[34] C. L. Goues, M. Pradel, and A. Roychoudhury, "Automated program repair," *Communications of the ACM*, vol. 62, no. 12, pp. 56–65, Nov. 2019. [Online]. Available: https://doi.org/10.1145/3318162