Journal Pre-proof

Design, implementation and evaluation of the Hedy programming language

Marleen Gilsing, Jesús Pelay, Felienne Hermans

PII: S2590-1184(22)00055-7
DOI: https://doi.org/10.1016/j.cola.2022.101158
Reference: COLA 101158

To appear in: *Journal of Computer Languages*

Received date : 18 March 2022
Revised date : 29 June 2022
Accepted date : 12 September 2022

Please cite this article as: M. Gilsing, J. Pelay and F. Hermans, Design, implementation and evaluation of the Hedy programming language, *Journal of Computer Languages* (2022), doi: https://doi.org/10.1016/j.cola.2022.101158.

Highlights

Design, Implementation and Evaluation of the Hedy Programming Language

- Details the full implementation of Hedy; the first gradual language for programming education
- Introduces an EBNF extension used for merging partial grammars to enable gradual language implementation
- Describes the first user study on Hedy

Design, Implementation and Evaluation of the Hedy Programming Language

Abstract

Hedy is a programming language that implements the gradual programming approach in which the goal is to lower the syntax barrier by starting with a very simple language, and gradually adding both concepts and refining syntax. This paper describes the design and implementation of Hedy, as well as a first user study involving 39 children between the ages of 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 educational 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.

1. 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 due to too high teacher expectations [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 all 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. For example in mathematics, students first learn a model of subtraction based on taking away items, such as cookies or apples, such that 5 *take away* 3 equals 2, but 3 *take away* 5 equal 0 since we can't take more than 3 items. Later on, this model is refined such that 3 *minus* 5 equals 2. Note that this is a change both in semantics as in syntax (*take away* versus *minus*), similar to the changes Hedy makes.

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].

This paper firstly covers the design and implementation of Hedy in more depth than previous studies.

Secondly, it presents a lesson series executed 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 the following three research questions:

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

Our study 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. Im-

provements might lie in a less sensitive language that allows different syntax combinations, better error messages and the localization of keywords.

Finally, this paper presents a number of improvements that were made to the Hedy implementation, following the findings of the above user study.

1.1. Comparison to Gilsing and Hermans 2021 [7]

This paper is an extension of a paper that previously appeared at VL/HCC 2021 [7] and also described the user study reported on in this paper.

The current paper however describes not only the user study, but also Hedy itself in its full depth. The first Hedy paper [5] described a previous version containing just the first thirteen levels. Since then, Hedy has been extended to 18 levels, the grammar system has been considerably improved and several new features have been added. This paper describes Hedy in full, more specifically, this paper extends [7] in the following ways:

1. An updated overview of Hedy's full 18 levels leading to a subset of Python (Section 3).
2. A extensive description of Hedy's implementation, including an innovative EBNF syntax and corresponding system to merge grammars (Section 4).
3. A new section describing additional educational features of Hedy (Section 7).
4. A more extensive Discussion section, especially detailing the connection between the Hedy language and its corresponding learning trajectory.

2. Related Work

2.1. 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 [8]. 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 [9]. Other researchers found that two languages commonly used for teaching, Java and Perl, are not easier to understand than a random language, stressing the difficulties that novices face in understanding syntax [10]. Interestingly enough, students assess learning syntax as more problematic than teachers [11], which

might shed some light on why little effort is given to the explicit explanation of syntax in many programming classrooms.

2.2. Gradual learning in Natural Languages

When learning to write a first natural 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. It has been argued that this gradual form of teaching works best at a young age, since beginning writers are young, and at that age learners typically operate at a low level [12]. This means learners 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 [13, 14], meaning that they do not plan carefully, but instead operate using trial and error, making small changes and continuing based on the feedback they receive on these small steps.

This notion has also gained popularity in Computer Science education. Lister argued that the cognitive load of students while programming might be overloaded by the simultaneous teaching of programming concepts, syntax and problem solving [15]. Because of this overload, students might behave similar to younger children learning to write.

Since learning a programming language shares significant characteristics with learning a natural language, i.e. 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 [16, 17].

2.3. Related Approaches for Teaching Novices

In previous research three different approaches in programming languages for novices can be distinguished [18]:

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 [19]. More modern examples of mini-languages are Scratch [20] and Karel the Robot [21]. Mini-languages are said to *"provide a solid foundation for learning a general purpose language"* [22], 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 [23], and MiniJava [24] and ProfessorJ [25] for Java.

Incremental approach As early as 1977, Shneiderman described what he called a 'spiral approach' to learning programming. Shneiderman 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."* Shneiderman also recommends extensive practice of simpler forms before advancing to new forms and argues this could help students who would otherwise be overwhelmed [26]. While a very inspiring idea, the paper did not come with a corresponding implementation in a language or lesson series and the term *spiral approach* has not caught on. Brusilovsky does not use the term spiral, but classifies these languages as *incremental* [18]. He states that incremental approaches teaches set of small subsets of a programming language, where each subset introduces new programming language constructs. The first implementation of a spiral or incremental approach was first created for PL/1 by Holt *et al.* [27] and later also applied to Fortran [28] and Pascal [29].

Other versions of incremental teaching used subsets that were explicitly not arranged as a hierarchy where a "higher level" contains the "lower level", but instead divided the language into overlapping languages like chapters in a textbook would [30]. DrScheme (later DrRacket) follow a similar incremental approach for Scheme [31, 32], in which users can select different Scheme subsets that limit the options for learners by implementing syntactical checks, for example to warn users that accidentally use infix rather than prefix notation. DrRacket's early language

levels are stricter then higher levels, e.g. in early languages, procedures must use at least one argument and names cannot be redefined, to prevent common but confusing mistakes.

More recently other papers have also described languages that are extended over the duration of a course, for example Cazzola and Olivares [33] 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 [34].

Of the three approaches, incremental languages are most similar to Hedy, especially the different languages of DrRacket where one of the inspirations for the Hedy language. Hedy's approach however, in a sense, is the opposite of DrRacket's. Where DrRacket is initially strict, warning the user with error messages, Hedy's syntax is initially very loose, allowing as much programs as possible, and later refines the syntax and error messages to be more strict. This decision is based on the experience that many novices find error messages discouraging, even when are correct and phrased precisely.

A second notable difference is Hedy's goal to be as close as possible to learners prior knowledge, leaning on their knowledge of natural language and mathematics. Programming in Hedy explicitly aligns with existing knowledge of concepts and syntax. For example, we believe that the choice of prefix operations (`(+ 4 4)`, also in the early languages of DrRacket) over infix operations (`4 + 4` in Hedy), while obviously easier to parse, poses both a high and also an unnecessary cognitive load for the learner since it differs from addition notation in mathematics, which and can reinforce the idea that programming is not just hard but also unreasonable.

Finally, in Hedy syntax explicitly changes over the different language levels, not so much to protect learners from making confusing mistakes, but because we believe that syntax is a concept which also needs to be explicitly taught and practiced. As such, Hedy's trajectory includes language levels that focus solely on syntactic changes. For example, at Level 4 only quotation marks are introduced as mandatory, so `print 'hello world'` will need to be used from then one.

2.4. Prior Work on Hedy

Hedy was initially introduced in [5]. Gilsing and Hermans [7] presents the first user study which this paper also reports on, and Yeni and van der Meulen [35] used the Technology Acceptance Model (TAM) to determine children's willingness and interest in using Hedy. Yeni and van der Meulen results indicate that many of their participants have a positive attitude towards Hedy, and felt that Hedy was indeed *"not too difficult but real programming"* in line with the original design goals, however some other participants felt Hedy was too limiting and desired more expressive power and programming features.

3. Language Design

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. Hedy is aimed at children from the age of 10. A more thorough description of Hedy's design philosophy can be found in [5], however in this paper we briefly present an overview of Hedy to contextualize the findings that will be presented later in the paper.

Hedy follows these six design goals:

1. **Concepts are offered at least three times in different forms.** This ensures that learners can practice enough with concepts and are exposed to different forms to learn their essence.
2. **The initial offering of a concept is the simplest form possible.** Previous research has shown that syntax can be confusing for novices [8, 10].
3. **Only one aspect of a concept changes at a time.** We want to minimize changes between each level because prior work shows that learners are only able to transfer knowledge if the syntactic differences are not too big [36].
4. **Adding symbolic syntax elements like brackets and colons is deferred to the latest moment possible.** Previous research has shown that operators such as `==` and `:` can be especially hard for novices [37].
5. **Learning new forms is interleaved between concepts as much as possible.** We want to give 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 [38].

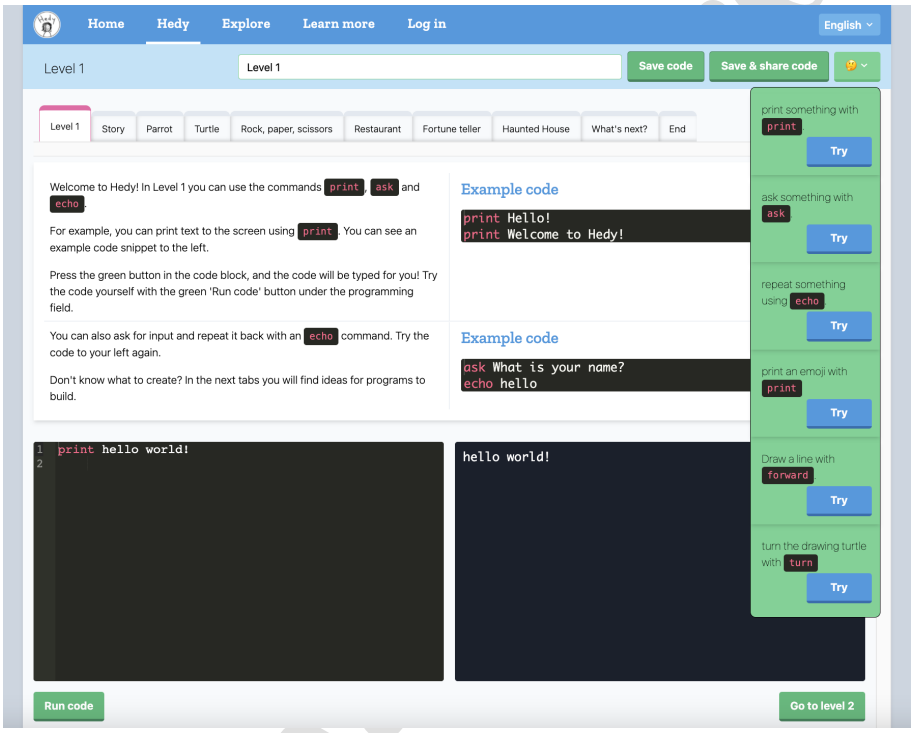

Figure 1: Level 1 of the Hedy user interface in English, including the green palette with commands on the right, and the built-in explanations above the coding field.

3.2. Levels

In this section, we describe all current levels of Hedy. The current version of Hedy consists of 18 levels, whereas earlier papers described only thirteen levels and studied only programs in the first seven levels [5, 7].

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 1. Furthermore students can ask for input of the user using the

keyword `ask`. Input of a user can be repeated with `echo`, optionally preceded by a textual prefix.

Initially, Hedy just supported text as output modality, however we found that just text as output is a bit limiting, so we added LOGO turtle commands too: `turn` and `forward`. More explanation on the turtle feature can be found in Section 7.2.

Level 2: Scalar variables using `is` At the second level, variables are added to the syntax but only scalar variables, holding one value. Defining a variable is done with the word `is` rather than the equals symbol fulfilling Design Goal 3 and Design Goal 4. Furthermore users do not have to add quotation marks to distinguish between strings and numbers: both `number is 12` and `name is Hedy` work.

Level 3: Assignment using `is` for lists It also adds the option to work with lists. More specifically, it lets users create lists with `is` syntax: `animals is car, dog, armadillo`.

Elements may also be indexed with `at`, for example `animals at 1`. List indexing is not restricted to numbers, random elements can also be accessed, with code like `animals at random`. Finally, Level 3 also allows learners to add and remove list elements: `add animal to animals`.

Level 4: Quotation marks and types In Level 4, the first non-textual 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. The variable syntax using `is` remains unchanged, meaning that learners can still use both `number is 12` and `name is Hedy`. Note that we do not yet require quotation marks in string assignment in Level 4, because we want to focus on the use of quotation marks in `print` statements, which is confusing enough. Quotes in string assignment become mandatory in Level 12.

Level 5: Selection with `if` and `else` flat In Level 5, 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. Learners are allowed but not required to use a newline before an `else` keyword.

Level 6: Calculations In Level 6, students learn to calculate: addition,

multiplication, subtraction and division are introduced. Therefore addition, multiplication, subtraction and division are introduced. While this might seem like a simple step, our experience taught us that the use of * for multiplication, rather than `x` as in mathematics is a steep learning curve and should be treated as a separate learning goal.

Level 7: Repetition with `repeat x times` In Level 7, repetition with a simple syntactic construction is introduced, as common in other educational programming languages [39, 40]: `repeat 5 times`. In this initial form, `repeat` is placed on one line, like `if` in Level 5:

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

It is allowed to use an `if` within a `repeat` but since that creates a very long line. we do not show that in the example code snippets.

Level 8: Code blocks with one level of nesting After Level 7, there is a clear need to 'move on', since the body of a loop (and also that of an conditional) can only consist of one line, which limits the possibilities of programs that users can create. This limitation could be a motivating factor for learners: rather than 'having to learn' the block structure of Python, learners are motivated by the prospect of building larger and more interesting programs (Design Goal 6). Since the end goal of Hedy is to teach Python, we choose to denote blocks using indentation rather that `begin` and `end` blocks. Using curly braces would not fit our Design Goal 4 of deferring symbolic syntax elements to the end of the learning trajectory. The syntax of the `repeat` itself remains otherwise unchanged as per Design Goal 3, so the new form is:

```
repeat 5 times
   print 'Hello'
   print 'I am repeated 5 times'
```

If Hedy users forget indentation, they receive a focused error message as shown in Figure 2. Nesting two levels deep will result in a message that that is not yet allowed in Level 8.

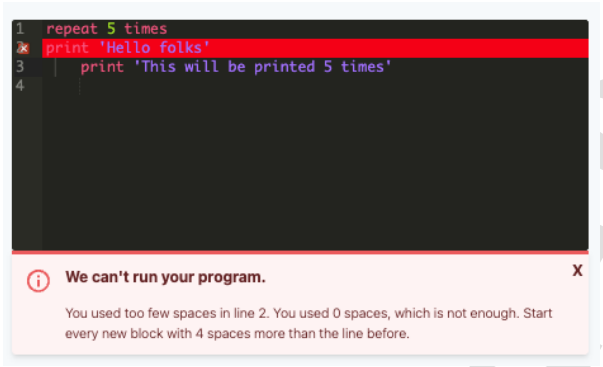

Figure 2: Error message given when indentation is missing in Level 8

Level 9: Code blocks with multiple levels of nesting To allow for enough interleaving of concepts (Design Goal 5), we defer the introduction of syntax concepts for now, and focus on more conceptual additions: the nesting of multiple blocks, for example a `repeat` command containing a condition, or the other way around, or multiple nested `repeat` blocks. We know indentation is a hard concept for students to learn, so nesting, using multiple levels of indentation, warrants its own level (Design Goal 3).

Level 10: For syntax looping over list In Level 10, learners the range syntax, looping over the values in a list, for example using with `for animal in animals`. This allows the customization of stories, drawing and songs and forms a gradual step towards the more complex numerical `for` construct using `range`.

Level 11: For syntax with in range 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 this allows for 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 introduced in Level 8 remains.

Level 12: Data types Learners are now allowed to use floats and need to place quotation marks around strings to distinguish them from numbers.

Level 13: Logical expressions In Level 13, learners learn about `and` and `or` in `if` statements to combine multiple conditions.

Level 14: Less than and greater than In Level 14, Learners learn

about < (=) and > (=) in preparation for `while` loops.

Level 15: While loops In Level 15, learners are introduced to the `while` loop. With the previous knowledge < (=) and > (=), learners can make basic while loops.

Level 16: Adding rectangular brackets In this level, learners encounter a type of brackets for the first time, because Level 16 adds rectangular brackets for list access, which up to now was done with the keyword `at`, following Design Goal 2. This level also explains accessing lists with a numeric index, starting at 1. While numerical access is possible from Level 3 we don't explicitly explain it in the learning sequence since there is not much value in using it without a loop.

Level 17: Learning elif and the colon symbol 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 and practiced over several levels, we can teach learners to use a colon before every indentation. This level also introduces `elif` to allow for more exciting programs, since just adding a colon does not really allow for more meaningful programs (Design Goal 6).

Level 18: Adding round brackets Level 18 adds round brackets in `print`, `range` and `input`. As per Design Goal 4, these are added as late as possible.

3.3. User Interface

The current user interface 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. In addition to the gradual approach, Hedy's UI has other educational features. For example, when starting a level, the programming field contains 'start code' that demonstrates the concepts that will be introduced in this level, so learners can try it immediately.

Hedy also features a *palette* that many educational languages like Scratch, Snap! and App Inventor also have. A palette is an overview of all possible blocks that can be dragged into the programming field [41, 42, 43]. A palette is helpful, because prior work shows that looking up information in a separate place increases cognitive load [44]. Students have confirmed this in studies. For example, Weintrop and Wilensky found that students frequently cited the *browsability* of blocks-based environments as a feature that made it easy to use [45].

Figure 1 shows the palette on the right, where each of the new commands in a level is shown with a 'Try' button, which will place the corresponding 'demo code' in the editor. This demo code is not executed automatically, so the user can 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 Python or JavaScript could also offer a palette to their users with possible code snippets. Weintrop and Wilensky suggested this in prior work: "*Adding an easily browsed, well organized library of valid commands that lives inside the Java or Python programming environment is one example of how we can use what we learn from novices about what makes blocks-based tools easy to improve and better prepare them for the transition to the text based tools that await them in more advanced courses*" [45].

However a palette for textual languages 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.

In addition to the palette with demo code snippets, Hedy also contains built-in explanation and exercises for each level. 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 the possibility to show a few brief explanations per level.

4. Implementation

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 comes with a built-in Ace[2] editor that allows Hedy code to be edited in the browser, as common in modern web-based IDE's for teaching such as repl.it and Trinket. We execute the resulting Python code with the Skulpt library[3].

The use of Skulpt enable teachers to run Hedy without installing anything but a browser, and will thus likely increase adoption in schools where teachers often have limited or no possibilities to install software. It also means Hedy

[1] https://github.com/lark-parser/lark
[2] https://github.com/ajaxorg/ace
[3] https://github.com/skulpt/skulpt

can be used on mobile phones and tablets. Hedy's code base is open source, available on GitHub.[4].

The remainder of this section explains Hedy's grammar and grammar merging system that are at the core of Hedy's implementation.

4.1. Grammars

The most important part of Hedy are its grammars that enable the gradual nature of the programming experience. Listing 1 shows the core part of the grammar used in Level 1, which allows for five basic commands: `print`, `ask` and `echo` for text output, and `forward` and `turn` for turtle output.

```
program: _EOL* (command _EOL+)* command?
command: print | ask | echo | turtle | error_invalid |
   error_invalid_space

print: _PRINT print_argument
ask: _ASK print_argument
echo: _ECHO print_argument
print_argument: (_SPACE text)?

turtle: _FORWARD (_SPACE NUMBER)? | _TURN (_SPACE (_LEFT |
   _RIGHT))?
error_invalid_space: _SPACE text?
error_invalid: textwithoutspaces text?
```

Listing 1: Partial Grammar of Level 1 in Lark

4.1.1. Symbol Tokens

Tokens definitions are not shown here for brevity, but are defined using uppercase names, such as `_EOL` for end of line or `_SPACE` for one or more spaces, this is a Lark convention that filters out uppercase production rules in the abstract syntax tree.

4.1.2. Keyword Tokens

Tokens for keywords are also uppercase, since the name of the rule determines the name in the AST, so tokens need not to be kept. Examples of keyword tokens are for example `_PRINT` and `_ECHO`. These will be replaced by their concrete strings in a preprocessing step. Initially this was done to make the grammar more readable, but recently Hedy supports localized keywords,

[4]https://github.com/felienne/hedy

so they can also be replaced by keywords in other languages (see further Section 7.4).

4.1.3. Error Productions

All Hedy levels make extensive use of error production rules, prefixed in the grammar with `error_`.

For example, the rule `error_invalid` matches any word that is placed before text separated with a space. Because Lark supports prioritization in the production rules, this rule is only matched when no prior rule (such as print or ask) matches. As such, the Level 1 parser also accepts `show hello world` and turns that into a parse tree with an error node. This allows us to give specific error messages like *"prnt is not a command in Hedy. Did you mean print?"*., as shown in Figure 9. Similarly `error_space` allow us to catch lines of code that start with spaces to warn learners you can't begin lines with spaces, as a preparation for the space sensitivity of Python.

You might notice in Listing 1 that commands do not need an argument to parse, e.g. the argument of `print` in the production is `print_argument`, defined as `(_SPACE text)?`. This too is done to generate more informative error messages. Before the transpilation step, we check the AST to see if all commands have arguments, and when relevant, whether the arguments have the right type. Using that information we can also generate more informative error messages, such as the message shown in Figure 3.

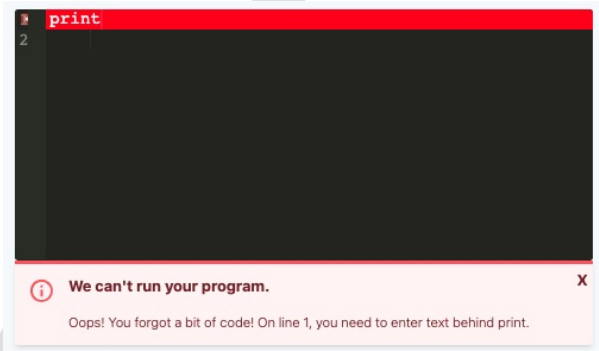

Figure 3: Error message given when a print command is used without an argument

You can also see in the grammar that rules `forward` and `turn` are gathered under one production rule, since the UI needs to adapt to the presence of

turtle code by adding a turtle *canvas*, as shown in Figure 4. To ease that analysis, both turtle commands are combined under one node so the parser can return a Boolean value indicating the presence of turtle commands.

Figure 4: When one of the turtle commands (`turn` or `forward` are used, the Hedy UI shows a smaller text output field and a large turtle canvas.

4.2. Grammar Merging

In our initial implementation, every Hedy level simply used its own grammar, as if each level was a truly separate language. This worked, but led to considerable duplication in the grammars, as each level redefined all rules of the previous level. This is not a problem when the rules change from one level to the next, but is needless duplication when the rules remain the same.

We have therefore introduced a grammar merging preprocessing step, that transforms partial grammars into full Lark compatible grammars. The basics of the grammar merging system are that rules may be redefined at new levels. When a rule is not defined at Level n, the definition from $n-1$ is copied. This already saves a lot of duplication.

However, sometimes we want to do operations that are a more complex, and therefore we add an abstraction on top of EBNF, which defines how rules will be merged in later levels. Our addition adds the following syntax to EBNF defining merges:

$+=$ To add new commands to an existing disjunction

$-=$ To remove commands from an existing disjunction

$>$ To indicate priorities between new options

The process of merging the partial grammars works pairwise; starting with the grammars of Level 1 and 2. These two grammars are merged together to create a full grammar for level 2. Subsequently, the grammar of Level 3 in merged into the grammar resulting from levels 1 and 2, and this process is repeated until all grammars are merged into a complete file, as shown in Figure 5 .

We call a grammar into which all prior levels are merged the *total grammar* for a level.

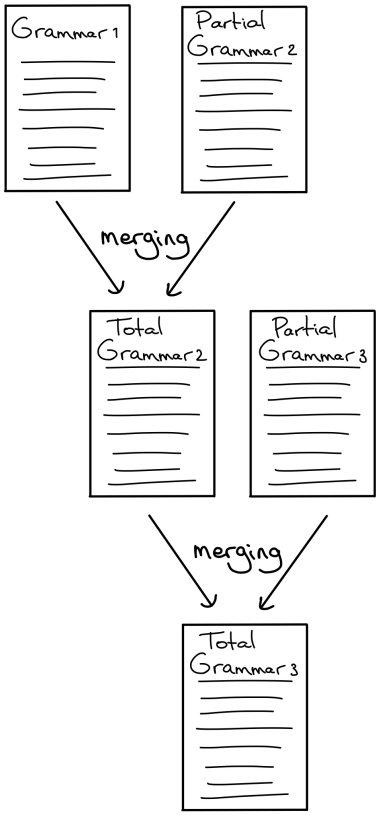

Figure 5: Process of merging partial grammar

4.2.1. Merging Example

Let's look at an example of grammar merging in action. The (simplified) grammar of Level 3 is shown in Listing 3. In Level 3, we add the option to define lists, and the option to access list elements. Note that `var` here in a built-in Lark construct that defines variable names starting with underscore or a letter, optionally followed by another underscore or letter or a number.

```
assign_list: var _SPACE _IS _SPACE text (_COMMA text_list)+
list_access: var _SPACE _AT _SPACE (INT | _RANDOM)

_print_argument: (list_access | text | punctuation)*
assign: var _SPACE _IS _SPACE (list_access | text)

add: _ADD_LIST text _SPACE _TO_LIST _SPACE var
remove: _REMOVE text _SPACE _FROM _SPACE var

command:+= assign_list | add | remove > error_invalid
```

Listing 2: Grammar of Level 1 in Lark

Unchanged Rules In the grammar of Level 3 we can firstly see that not all rules present in the grammars of Level 1 are redefined in Level 3. For example, `program` is not redefined, and as such, the existing definition of Level 1 will be used in the total grammar. Not that Level 2 also did not redefine this production.

New Rules Furthermore, we see that a number of new rules are introduced, such as creating a list with the rule `assign_list`. This rule allows for the definition of lists, for example: `animals is cat, dog, parrot`. Another addition is `list_access`, which defines accessing a list, using either a number `animals at 4` or with random access: `animals at random`. Because these rules were not present in the grammar of Level 2, they are added to the merged grammar. Finally, `add` and `remove` are added to grow and shrink lists.

Redefined Rules Some rules get a new definition, for example the argument of a `print`: `print_argument`. While in Level 1 `print` could only use `text` as an argument, we now also add the option to print access into a list, with `print animals at 4`

Partially Changed Rules Because `print_argument` is changed, there are also rules that are partially changed. The rule `print` itself is not defined again, and thus stays the same (namely `print: _PRINT print_argument`) However, because its arguments may now also include list access, conceptually the rule `print` also changed.

Expanded Disjunctions `command` was initially defined as this disjunction: `print | ask | echo | turtle | error_invalid_space | comment | error_invalid`. In Level 2 this rule was already expanded, since assignment was introduced with a new rule. Level 3's grammar expands the rule even more by adding the rules `assign_list`, `add` and `remove` as possible commands.

Forced Ordering Because we want the error production rule `error_invalid` to only match if none of the other rules match, this needs to be placed at the end of the disjunction to indicate the right priority. Our syntactic element > adds this rule at the end of the existing options.

The full syntax and the semantics of Hedy programs at all levels, as expressed in their corresponding Python programs can be found in our Github Repository [5]

4.3. Transpiling to Python

Once Hedy programs have been parsed, the resulting abstract syntax tree (AST) is then transformed into Python, which we run in the browser using Skulpt. In many cases, this transformation is a simple process, for example, `print hello` in Level 1 only requires us to surround the print argument hello with brackets and quotes.

In some cases however, we need to perform more elaborate processing of the AST. For example, in levels 2 and 3, where users may freely mix variables and plain text strings. Enabling this is done in two steps: in the first step we gather all variable names from the assignment nodes of the AST. In the second step we determine, for each print node, whether its children are contained in the list of variable names. If so, they are printed without quotes, if not, they are printed with quotes. Hence, this Hedy Level 2 program:

```
name is Hedy
print hello name
```

will be converted into:

```
name = 'Hedy'
```

[5] https://github.com/Felienne/hedy/blob/main/semantics/SEMANTICS.md

```
print('hello', name)
```

5. Research setup

Now that we have covered the design and implementation of Hedy, it is time to direct our attention to its first qualitative evaluation. The goal of this study 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.

5.1. Participants

In total, 39 children participated in our study, all students from two different classes within one school in the Netherlands, in a grade equivalent to American seventh grade.

5.1.1. Prior experience

From the 39 children in our study, 36 had prior experience with programming. Figure 6 shows an overview of the programming language(s) the participants were familiar with before the study. Most children (29 out of 36 with experience, or 80%) had experience only with Scratch, hence no experience with textual languages.

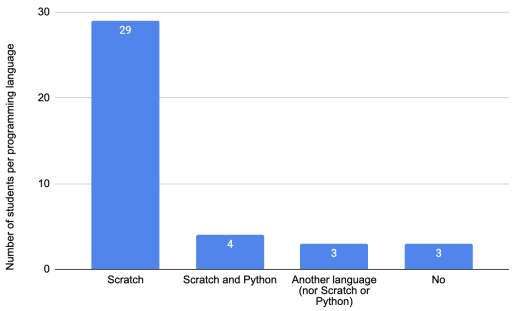

Figure 6: Languages the participants had experience with

The children acquired their previous experience at various different sources, as shown in Figure 7. 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 7 twice.

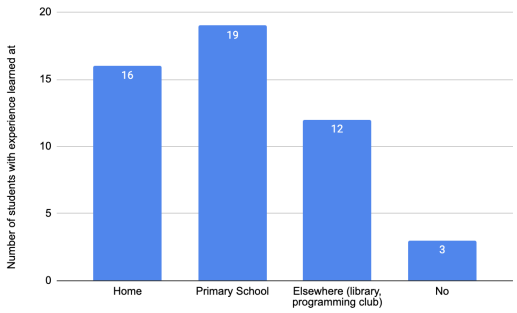

Figure 7: Places where the participants gained programming experience

5.1.2. 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 8.

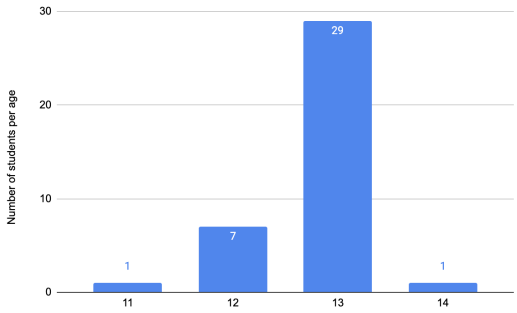

Figure 8: Ages of participants in the study

5.1.3. Gender

The 38 students that disclosed gender were 22 boys and 16 girls.

5.2. Lessons

All participants followed 12 online Hedy lessons during 6 consecutive weeks, each lesson of one hour. All lessons were conducted online by the first author. She is trained and employed as a teacher, but is not a regular teacher at the school in which the study was run, and she 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.

5.3. Survey

After the 12 lessons were completed, students were asked to fill out a written open text survey in Dutch, 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 5.1.1.

5.4. Research questions

The goal of this paper is to gain a deeper understanding of how gradual programming in general, and Hedy in particular, 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.

While it is hard to separate the Hedy language and the gradual programming approach from each other, 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 designers of both Hedy and future gradual programming approaches in deciding on trade-offs in their implementations.

As such, our research questions are:

1. What are the benefits of a gradual programming approach?
2. What are the challenges of a gradual programming approach?
3. How can gradual programming approaches like Hedy be improved?

5.5. Data Collection and Processing

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.

6. Results

In this section we answer our three research questions based on participants' survey answers and our lesson observations. Note that all quotes that we present here were originally given in Dutch by participants. We have translated them as best as possible for presentation in this paper.

6.1. RQ1: 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.

6.1.1. 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!*" when looking at a program containing `echo`. In observing students working with Level 1, we saw relatively little struggle with syntax, especially compared to our experiences teaching Python to the same age group in the same setting, using similar assignments. This strengthens our belief that Hedy lowers the cognitive load associated with syntax.

6.1.2. 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". While this program might be very confusing to experienced programmers because of the lack of symbols indicating the different roles of program parts, this is a trade off, because 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 print 'round and round'
print 'the wheels on the bus go'
repeat 3 times print 'round and round'
print 'All through 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.

6.1.3. 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 this observation confirms our initial hypothesis.

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*".

6.1.4. Palette and Explanations

As outlined in Section 3.3, Hedy has features that go beyond gradual programming: a palette and built-in explanation and examples. 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.

6.2. RQ2: What are challenges of a gradual programming approach?

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

6.2.1. 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 [44, 46, 47]. 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).

6.2.2. 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.

6.2.3. 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*".

6.3. RQ3: How can gradual programming languages like Hedy be improved?

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

6.3.1. 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 is 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 from 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 4, 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 4 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.

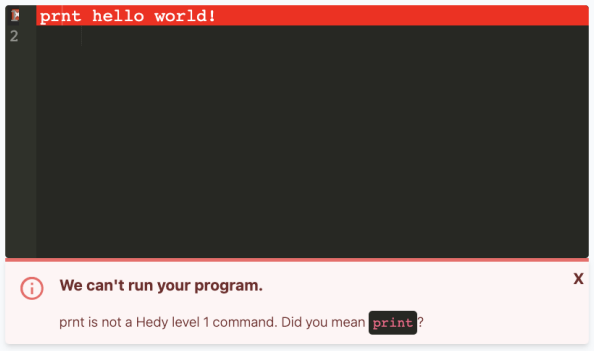

Figure 9: Misspelling a keyword in Level 1 produces an error message that is relatively easy to understand.

The sensitivity issues might also be related to error messages. Error messages regarding to the misspelling of keywords, as illustrated by Figure 9 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 10 we can only detect that there are mismatched quotation marks, and remind children to begin and end with a quotation mark.

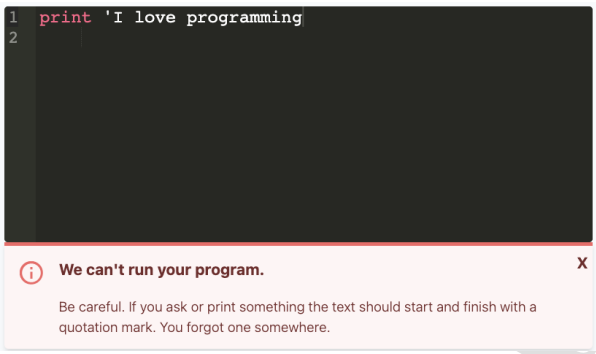

Figure 10: Mismatched quotation marks in Level 3 produces an error message when a quotation mark is not matched.

While that is a fine message for a program like the one in Figure 10, the same error is produced when a single quotation mark is used inside of a string, as demonstrated by Figure 11. 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*".

Figure 11: A string containing a single quotation mark in Level 4 produces an error message that can be confusing.

6.3.2. 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, which is in line with earlier work on error messages. For example, the work by Barik which showed that novice programmers read the error messages [48], and the work of Becker *et al.* who found that beginners struggle with error messages more than experts [49].

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 [37, 50] 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.

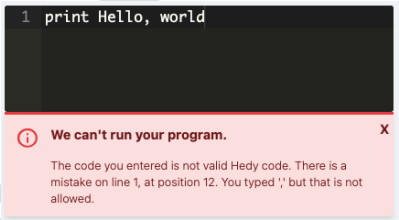

Figure 12: Error messages in Hedy refer to characters, such as the comma here, by symbol, and place them between quotation marks.

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 12. This is not an unusual choice, in fact,

this is how most languages present characters, see also Figure 13 that shows a similar error message in JavaScript as shown by Chrome.

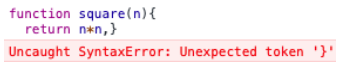

Figure 13: Error messages in JavaScript also refer to characters, such as the closing curly brace here, by symbol, and place them between quotation marks.

When asking children to read these messages aloud, we noticed they would skip these symbols entirely. For the error message shown in Figure 12, 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 all their lives to not verbalize 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 4. However, we saw this behaviour also after quotation marks were introduced, showing that this knowledge in programming syntax does not necessarily lead to the understanding that in error messages quoted text means a literal string value.

We suspect that error messages in other languages, such as the message in Figure 13 will confuse novices in similar ways, although that of course warrants further research.

6.3.3. 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, since the interface, lessons and some parts of the example programs (raw text, variable names) were also written in Dutch.

7. Recent Additions to Hedy

The above user study and other experiments have led to a number of new features and additions to Hedy which we will describe in this section. These features were not yet incorporated in the presented user study, and their effectiveness will be the topic of future studies.

7.1. Read Textual Output Aloud

We have not only improved the Hedy language and its interpreter, but also the Hedy interface. For example, we have added the option to have the textual output of a Hedy program read aloud. In other observations, we saw children copy-pasting the output of Hedy programs into Google Translate, which allows for it to be read aloud. Especially when the output is a song, this makes the output more authentic to children. Using the Google Speech API it was relatively easy to add this feature into the Hedy user interface directly. Figure 14 shows the UI element that learners can use to select a voice. When a voice is selected and code is run, textual output will be printed as before, and the output will also be read aloud.

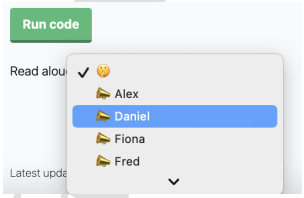

Figure 14: Drop down list to choose from different voices

This feature adds two different values: Firstly, it adds fun! Children enjoy hearing robot voices read stories and 'sing' songs, increasing their engagement. Secondly, it creates an inclusive environment in which both low vision users and sighted users can share the same experience and collaborate, rather than having a different output for low vision users.

7.2. Turtle Graphics

Initially, Hedy only supported textual output, however in the study described in [35], some children explicitly requested other modalities, like the drawing turtle, which the researchers showed them in Python. We therefore added the commands `turn` and `forward` so children can also create drawn

output and create geometric shapes and images, as shown in Figure 4 earlier in this paper. We doubted a bit whether the turtle commands should be added to Level 1 or should come later in the trajectory. On the one hand side, we want to keep Level 1 simple to not overwhelm learners, but on the other hand, we also want to demonstrate to children what Hedy's possibilities are so they understand the path that they will embark on. Ultimately we have decided for that reason to add turtle options to Level 1.

7.3. Improved Error Messages

After this user study, we have updated Hedy error messages that refer to characters (such as ','), showing the characters in words(such as comma), so they stand out more to users, as shown in Figure 15.

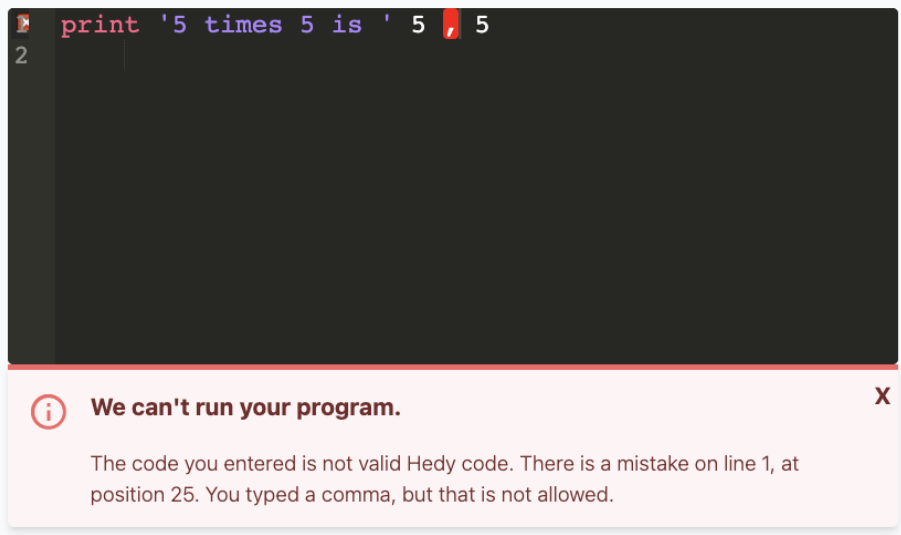

Figure 15: Error messages in Hedy refer to characters, such as the comma here, by symbol, and place them between quotation marks.

7.4. Localized Keywords

One of the more surprising outcomes of this study was the desire of users for localized keywords. Since the Netherlands has the highest command of

English outside of English speaking countries[6], we expected teenagers to be happy using English keywords, especially given the small set of keywords in the earlier Hedy levels.

However, prior research shows that children learn programming constructs quicker when they program in their own native language [51], and other educational languages do allow for keywords to be presented in the language of the learner, including Scratch [20] and Snap! [43].

The wishes of our users combined with the findings of Dasgupta led us to implement localized keywords, so, for example, Spanish users can use `imprimir hola!` in addition to `print hola`. We allow both English keywords and the native language to cater for bilingual learners, and for learners that might already know some but not all English keywords.

We achieve this by substituting the keyword tokens, such as `_PRINT` or `_ASK` by a disjunction with their localized counterparts in the grammar merging step: `"print" | "imprimir"`.

Figure 16 shows the UI, code and its execution and the palette in Spanish. Because we rely on community efforts for our translation, not all parts have been translated yet.

[6]https://www.ef.nl/epi/regions/europe/netherlands/

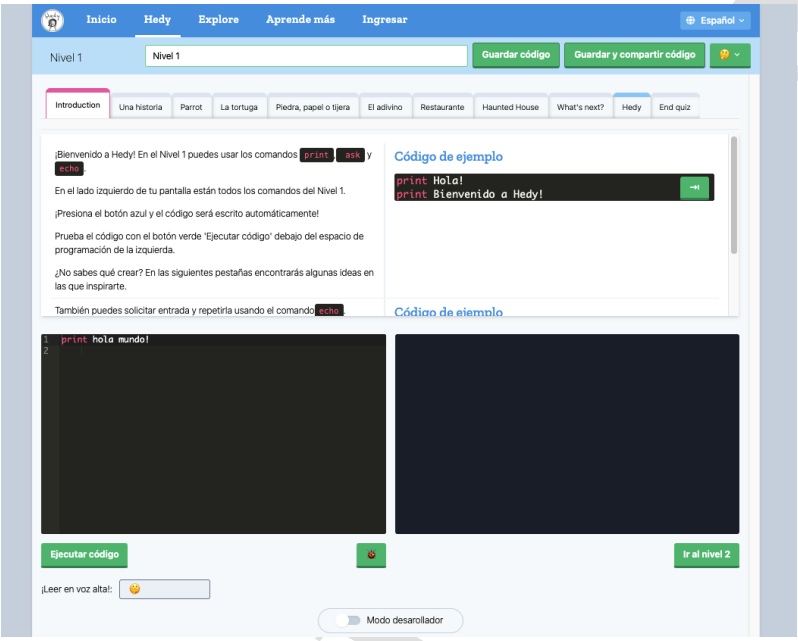

Figure 16: Hedy User Interface shown (largely) in Spanish.

7.5. Program Repair

The suggestion by users in our study that programs should be able to auto-correct themselves lead to some interesting new challenges. While the problem of *program repair* is hard in the general case, a lot of progress has been made in recent years [52, 53, 54].

Since the user study, we have implemented some simple repair techniques. For example, programs that start with a space but are otherwise correct can be transpiled to Python by removing the space. We made the choice to indeed repair the program and run the repaired program. For the user this results in both a warning message and execution of the code, as shown in Figure 17. As an extra help, students can click the Hedy light bulb, which will show the correct code.

In some cases, such as the program shown in 9, where the keyword `print` is misspelled as `prnt`, we *can* repair the program, since we can reliably guess what the right keyword should be.

However, we do not directly repair and run the code, so learners will have

to type the code manually, and learn the right syntax. In this situation, we also show the Hedy light bulb that can be used to show the correct code in a modal popup.

We decided to show the code in such a modal popup, rather than in the coding field, so learners deliberately copy the code and not just click the repair icon and then run the code immediately.

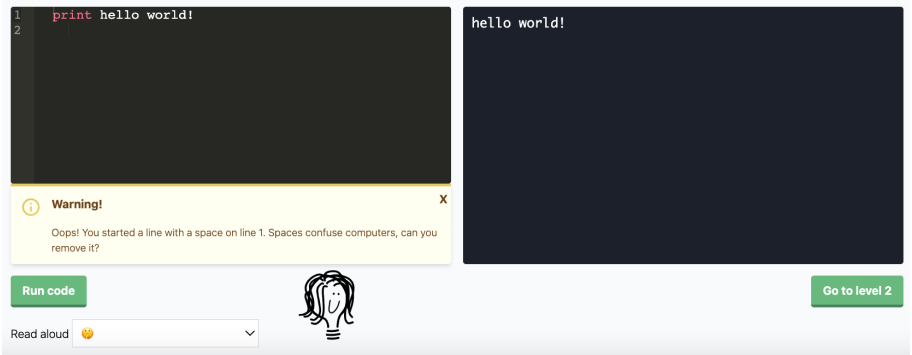

Figure 17: A program on the left that starts with a space will execute, but also give a warning.

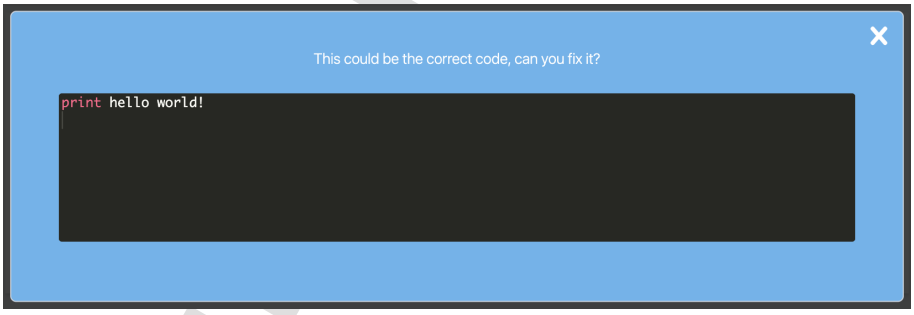

Figure 18: When users click the light Bulb Icon, they are presented with the correct code. They will have to enter the correct code themselves.

7.6. Pause in Repetitions

Once we started to teach repetitions in Level 7, we realized that it is not informative to output the results of a repetition at once. The execution of

Python (and other programming languages) is typically so quick that code that repeatedly prints something looks like it is executed instantaneously. For example, the following code snippet:

```
for i in range(5):
   print(i)
```

This will print 0 to 4 on the screen, seemingly in one go. We found that for novice learners, this does not stress the fact that this code repeatedly executes the second line. We have therefore added a little pause in the execution of each iteration of the loop. While this is a small change, it is an interesting one, since it is enabled by the fact that Hedy is a language and platform for learners. In a language meant for professionals, obviously one would never want to intentionally slow down code execution.

8. Discussion

8.1. Study Limitations

This study suffers from a number of limitations. Firstly, the sample contains just two classes within one school in one country. In the future, we plan to run larger and more diverse studies. Secondly, due to Covid-19, the lessons were conducted online. This has of course impacted the ability of the authors to explain Hedy, to help individual learners and to observe programming behaviour. While we recorded videos of learner screens and used written surveys to collect opinions, a physical lesson series might have led to less confusion and richer observation data. A more recent paper on Hedy was able to collect student opinions in real life and confirmed some of our findings on Hedy's ease of use [35].

8.2. Gradual Languages and Learning Trajectories

While Hedy is an implementation of a gradual language, the order in which the Hedy platform presents concepts is also a form of a learning trajectory, and the levels are (deliberately) combined with a series of levels, and assignments of increasing difficulty.

It is interesting to consider whether Hedy could exist without accompanying assignments. Would it be valuable to have a 'bare-bones' version of Hedy in which the teacher could insert all lessons themselves? What guides

the level progression if there are no exercises or advised structure, and thus would the gradual approach also add enough value in such a case?

Furthermore it would be interesting to reflect on the order of the concepts. Can the gradual approach be adapted such that teachers, or maybe even learners, make their own trajectories? While it might be hard to merge the grammars in a different order on the fly that should not be a reason to not explore the potential value.

8.3. 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?*"

8.4. Use of Multiple Correct Syntactic Forms

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 characters 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 levels, where its grammars 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 novices can get used to the syntax and learn it, without being frustrated by errors.

8.5. Better Alignment with Operators in Mathematics education

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 × or x were not used for multiplication. From their perspective of course, this is a reasonable request, if × 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 :[7] or ÷ for division.

Clearly there are sensible reasons that these symbols are not used in traditional programming languages; × 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 × and x for a few levels, as proposed in Section 8.4, before mandating only * and /.

8.6. Focus on Memorization

Some participants in our study specifically name the memorization of commands as 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 precise way to create certain commands. Future gradual programming languages and 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.

[7]: is commonly used in Dutch schools for division

8.7. Localized Productions and Gradual Delocalization

Some of the newly introduced features also lead to new discussion points. For example, while we support localized keywords, this does not entirely address the problem of a localized language.

Two large open problems remain. Firstly, truly localizing a language entails more than just keywords. For example, in some languages, productions need to be overridden to change the order of words in programming constructs. For example, in Dutch, word order in condition is different; instead of subject verb object "if number is 5") Dutch uses subject object verb "[if] number 5 is". This does not fit the traditional form of conditions in programming languages. In other cases, we do not need to reorder, but radically change productions, for example, Arabic does not have a verb for "are" and hence needs a different grammatical structure for assignment since "x is 5" as Hedy Level 2 introduces, does not have a clear translation. Finally, there are various internationalized versions of characters like quotes, French traditionally uses <<, while Arabic has its own version of the comma and the double quote.

A second issue is the gradual 'delocalization' of the grammars. If we allow learners to use keywords in their own language, and our goal is also to have them use a subset of a professional language like Python at the end of the level trajectory, we need to disallow the use of localized keywords at a certain level. This brings interesting didactic challenges: do we delocalize all keywords at once, at the end of the trajectory, in Hedy's case in or after Level 18? Or do we teach a keyword localized first, i.e. `imprimir` for `print` in Spanish, and then in English, before we move on to a new keyword?

8.8. Differences in Levels in the Classroom

As explained in Section 6.1.3, some children moved through the levels quicker than others. It was not always clear to us what drove those differences. Was that a genuine difference in ability? Or were some children more confident and thus willing to continue to a higher level, while others might have lower self-efficacy and thus felt the desire to practice more. It might be useful to include an adaptive leveling system in Hedy that could both motivate some children to continue to a higher level while also putting a brake on children that skip several levels and then get confronted with too many error messages.

8.9. Program Repair

While we offer a crude form of program repair that can remove spaces or suggest a corrected keyword, there are a lot more avenues to explore for program repair. For example, random textual edits like inserting quotes or replacing words by legal keywords might reach valid programs quickly. These edits could be gathered from the history of edits that learners have made in the past. It could also 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. Program repair in Hedy stands at an interesting intersection between technical and instructional challenges. Because suppose a buggy program could be correctly repaired, should we present that to the user directly? Or should we show only the edit? Should we maybe reveal the correct program gradually (first the edit, and if that does not work, the full program). How certain should we be that the program is indeed correct before we show it to the user at all, and how can we be sure? These are all interesting open questions to explore.

9. Concluding remarks

Gradual programming is an innovative approach for teaching programming, which starts with a simple syntax and gradually adds both concepts and more complex syntax.

The goal of this paper is twofold. Firstly, it describes the gradual programming language Hedy in depth, including its design and implementation using gradual grammars. Furthermore, in order to understand the benefits and challenges of the gradual programming language approach, and explore how it can be further improved, we taught 39 children aged 11 to 14 for six weeks using Hedy. This study helped us to answer these research questions:

RQ1 What are benefits of a gradual programming approach?
RQ2 What are challenges of a gradual programming approach?
RQ3 How can gradual programming approaches such as Hedy be improved?

Our findings show that children appreciate the gradual nature of Hedy, especially the control they have 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 with the lessons we learned from the current study. Secondly, the current version of Hedy consists of 18 levels, while this study only studies the first levels. Hence, a replication of this user study 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.

Secondly, not all Python features are currently present in the learning trajectory, most notable, functions are not explained. Designing a more advanced learner path which also incorporates functions, and maybe even classes and methods, might be a valuable path for further Hedy development, enabling the use of Hedy not just in middle school but also in high-schools or universities.

References

[1] T. Beaubouef, J. Mason, Why the High Attrition Rate for Computer Science Students: Some Thoughts and Observations, SIGCSE Bull. 37 (2) (2005) 103–106. doi:10.1145/1083431.1083474.
URL http://doi.acm.org/10.1145/1083431.1083474

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

[3] A. Luxton-Reilly, 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, 2016, pp. 284–289, event-place: Arequipa, Peru. doi:10.1145/2899415.2899432.
URL 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, ICER '20, Association for Computing Machinery, New York, NY, USA, 2020, pp. 259–270, event-place: Virtual Event, New Zealand. doi:10.1145/3372782.3406262.
URL https://doi.org/10.1145/3372782.3406262

[6] V. Braun, V. Clarke, Reflecting on reflexive thematic analysis, Qualitative Research in Sport, Exercise and Health 11 (4) (2019) 589–597, publisher: Routledge _eprint: https://doi.org/10.1080/2159676X.2019.1628806. doi:10.1080/2159676X.2019.1628806.
URL https://doi.org/10.1080/2159676X.2019.1628806

[7] M. Gilsing, F. Hermans, Gradual Programming in Hedy: A First User Study, in: 2021 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), 2021, pp. 1–9, iSSN: 1943-6106. doi:10.1109/VL/HCC51201.2021.9576236.

[8] P. Denny, A. Luxton-Reilly, E. Tempero, J. Hendrickx, Understanding the syntax barrier for novices, ACM, 2011, pp. 208–212. doi:10.1145/1999747.1999807.
URL http://dl.acm.org/citation.cfm?id=1999747.1999807

[9] A. Altadmri, 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 (2015) 522–527doi:10.1145/2676723.2677258.

[10] A. Stefik, S. Siebert, An Empirical Investigation into Programming Language Syntax, Trans. Comput. Educ. 13 (4) (2013) 19:1–19:40. doi:10.1145/2534973.
URL http://doi.acm.org/10.1145/2534973

[11] M. Piteira, 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, ISDOC '13, Association for Computing Machinery, New York, NY, USA, 2013, pp. 75–80, event-place: Lisboa, Portugal.

doi:10.1145/2503859.2503871.
URL https://doi.org/10.1145/2503859.2503871

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

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

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

[15] R. Lister, 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, 2016, pp. 5–16, event-place: Münster, Germany. doi:10.1145/2978249.2978251.
URL https://doi.org/10.1145/2978249.2978251

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

[17] F. Hermans, 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.

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

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

[20] M. Resnick, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman, Y. Kafai, Scratch: Programming for All, Commun. ACM 52 (11) (2009) 60–67.

doi:10.1145/1592761.1592779.
URL http://doi.acm.org/10.1145/1592761.1592779

[21] B. W. Becker, Teaching CS1 with karel the robot in Java, in: SIGCSE '01, 2001. doi:10.1145/364447.364536.

[22] P. Brusilovsky, E. Calabrese, J. Hvorecký, A. Kouchnirenko, P. Miller, Mini-languages: A way to learn programming principles, Education and Information Technologies 2 (1997) 65–83. doi:10.1023/A:1018636507883.

[23] B. Heeren, D. Leijen, A. van IJzendoorn, Helium, for Learning Haskell, in: Proceedings of the 2003 ACM SIGPLAN Workshop on Haskell, Haskell '03, Association for Computing Machinery, New York, NY, USA, 2003, pp. 62–71, event-place: Uppsala, Sweden. doi:10.1145/871895.871902.
URL https://doi.org/10.1145/871895.871902

[24] E. Roberts, 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, 2001, pp. 1–5, event-place: Charlotte, North Carolina, USA. doi:10.1145/364447.364525.
URL https://doi.org/10.1145/364447.364525

[25] K. E. Gray, M. Flatt, 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, 2003, pp. 170–177, event-place: Anaheim, CA, USA. doi:10.1145/949344.949394.
URL https://doi.org/10.1145/949344.949394

[26] B. Shneiderman, Teaching programming: A spiral approach to syntax and semantics, Computers & Education 1 (4) (1977) 193–197. doi:10.1016/0360-1315(77)90008-2.
URL http://www.sciencedirect.com/science/article/pii/0360131577900082

[27] R. C. Holt, D. B. Wortman, D. T. Barnard, J. R. Cordy, SP/k: a system for teaching computer programming, Commun. ACM 20 (1977) 301–309. doi:10.1145/359581.359586.

[28] T. Balman, Computer assisted teaching of FORTRAN, Computers & Education 5 (2) (1981) 111–123. doi:10.1016/0360-1315(81)90020-8.
URL http://www.sciencedirect.com/science/article/pii/0360131581900208

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

[30] I. Tomek, T. Muldner, S. Khan, PMS—A program to make learning Pascal easier, Computers & Education 9 (4) (1985) 205–211. doi:10.1016/0360-1315(85)90009-0.

[31] M. Felleisen, R. B. Findler, M. Flatt, S. Krishnamurthi, The DrScheme project: an overview, ACM SIGPLAN Notices 33 (6) (1998) 17–23. doi:10.1145/284563.284566.
URL https://doi.org/10.1145/284563.284566

[32] R. Findler, DrRacket (1996).
URL https://plt.cs.northwestern.edu/snapshots/current/pdf-doc/drracket.pdf

[33] W. Cazzola, D. M. Olivares, Gradually Learning Programming Supported by a Growable Programming Language, IEEE Transactions on Emerging Topics in Computing 4 (3) (2016) 404–415. doi:10.1109/TETC.2015.2446192.

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

[35] Yeni, Sabiha, van der Meulen, Anna, Students' Behavioral Intention to Use Gradual Programming Language Hedy, in: Proceedings of the 27th ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '22), Proceedings of ITiCSE, 2022, pp. –, to Appear.

[36] E. Tshukudu, Q. Cutts, 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, 2020, pp. 307–313. doi:10.1145/3341525.3387406.
URL https://doi.org/10.1145/3341525.3387406

[37] F. Hermans, A. Swidan, 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. doi:10.1145/3196321.3196355.
URL https://research.tudelft.nl/en/publications/code-phonology-an-exploration

[38] J. S. Brown, A. Collins, P. Duguid, Situated Cognition and the Culture of Learning, Educational Researcher 18 (1) (1989) 32–42. doi:10.3102/0013189X018001032.
URL https://doi.org/10.3102/0013189X018001032

[39] A. Stefik, R. Ladner, 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, 2017, pp. 641–641, event-place: Seattle, Washington, USA. doi:10.1145/3017680.3022377.
URL http://doi.acm.org/10.1145/3017680.3022377

[40] T. Kohn, Teaching Python Programming to Novices: Addressing Misconceptions and Creating a Development Environment, PhD Thesis, ETH Zurich (2017).

[41] D. Wolber, H. Abelson, M. Friedman, Democratizing Computing with App Inventor, GetMobile: Mobile Computing and Communications 18 (4) (2015) 53–58. doi:10.1145/2721914.2721935.
URL https://doi.org/10.1145/2721914.2721935

[42] S. C. Pokress, J. J. D. Veiga, MIT App Inventor: Enabling Personal Mobile Computing, arXiv:1310.2830 [cs]ArXiv: 1310.2830 (Oct. 2013).
URL http://arxiv.org/abs/1310.2830

[43] M. Ball, J. Mönig, B. Romagosa, 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, SIGCSE '19, Association for Computing Machinery, New York, NY, USA, 2019, p. 1279. doi:10.1145/3287324.3293863.
URL https://doi.org/10.1145/3287324.3293863

[44] F. Paas, 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 29 (4) (2020) 394–398, publisher: SAGE Publications Inc. doi:10.1177/0963721420922183.
URL https://doi.org/10.1177/0963721420922183

[45] D. Weintrop, 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, IDC '15, Association for Computing Machinery, New York, NY, USA, 2015, pp. 199–208. doi:10.1145/2771839.2771860.
URL https://doi.org/10.1145/2771839.2771860

[46] A. V. Robins, L. E. Margulieux, B. B. Morrison, Cognitive Sciences for Computing Education, in: A. V. Robins, S. A. Fincher (Eds.), The Cambridge Handbook of Computing Education Research, Cambridge Handbooks in Psychology, Cambridge University Press, Cambridge, 2019, pp. 231–275. doi:10.1017/9781108654555.010.
URL https://www.cambridge.org/core/books/cambridge-handbook-of-computing-educ

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

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

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

[50] A. Swidan, 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), 2019, pp. 178–184. doi:10.1145/3300115.3309504.
URL https://research.tudelft.nl/en/publications/the-effect-of-reading-code-al

[51] S. Dasgupta, B. M. Hill, Learning to Code in Localized Programming Languages, in: Proceedings of the Fourth (2017) ACM Conference on Learning @ Scale, L@S '17, Association for Computing Machinery, New York, NY, USA, 2017, pp. 33–39. doi:10.1145/3051457.3051464.
URL https://doi.org/10.1145/3051457.3051464

[52] C. Le Goues, T. Nguyen, S. Forrest, W. Weimer, GenProg: A Generic Method for Automatic Software Repair, IEEE Transactions on Software Engineering 38 (1) (2012) 54–72. doi:10.1109/TSE.2011.104.

[53] C. Le Goues, M. Dewey-Vogt, S. Forrest, 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), 2012, pp. 3–13, iSSN: 0270-5257. doi:10.1109/ICSE.2012.6227211.

[54] C. L. Goues, M. Pradel, A. Roychoudhury, Automated program repair, Communications of the ACM 62 (12) (2019) 56–65. doi:10.1145/3318162.
URL https://doi.org/10.1145/3318162

Title Page (with author details)

Title
Design, Implementation and Evaluation of the Hedy Programming Language

Authors
Marleen Gilsing[a], Jesús Pelay[b], Felienne Hermans[a,c]

a. Leiden University, Leiden, The Netherlands
b. Universidad de Carabobo, Carabobo, Venezuela
c. Vrije Universiteit, Amsterdam, The Netherlands

Corresponding author
Felienne Hermans – LIACS office 104
Niels Bohrweg 1
2333 CS Leiden, the Netherlands

Financial Disclosure:
None reported

Conflict of Interest:
None reported

Title
Design, Implementation and Evaluation of the Hedy Programming Language

Authors
Marleen Gilsing[a], Jesús Pelay[b], Felienne Hermans[a,c]

a. Leiden University, Leiden, The Netherlands
b. Universidad de Carabobo, Carabobo, Venezuela
c. Vrije Universiteit, Amsterdam, The Netherlands

CRediT author statement

Marleen Gilsing: Investigation, Data Curation, Writing - Review & Editing
Jesús Pelay: Software, Writing - Review & Editing
Felienne Hermans: Conceptualization, Methodology, Software, Resources, Writing - Original Draft, Writing - Review & Editing, Funding acquisition

Declaration of interests

☒ The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

☐ The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: