Informatics in Education, 2024, Vol. 23, No. 4, 791–822

© 2024 *Vilnius University*
DOI: https://doi.org/10.15388/infedu.2024.28

The Story of Building Hedy

A Programming Language with Cognitive Science in Mind

Felienne HERMANS
Vrije Universiteit Amsterdam, De Boelelaan 1105, 1081 HV Amsterdam, NL
e-mail: f.f.j.hermans@vu.nl

Abstract. This autoethnographic paper is part of a special issue trying to answer the question "*How to design or choose languages for programming novices?*" I will describe how my programming language Hedy was created, how the initial design goals were formed, how my perspectives on learning and teaching changed along the way, and how Hedy changed with it. The paper also discusses how the Hedy community came to be. Hedy was initially made for my own classroom and teaching, but quickly attracted a community, which I learned a lot from. This special issue has given me a unique opportunity, after 5 years of working on Hedy, to reflect on the process and to learn from it myself, and will hopefully also allow other programming language designers to learn from.

Key words: programming languages, education.

1. Introduction

Hedy is a programming language aimed to teach beginners, which grew from a small prototype to a large-scale open source project. In this paper I reflect on the creation and growth of Hedy with the goal to help people who are creating educational languages, and also others involved in designing software in a community-driven way. More specifically, I will describe the process the led to the creation of Hedy and the community of users surrounding it. The road to Hedy's current version is described somewhat chronologically, and with autobiographic elements, with much attention being paid to the main ideas, such as cognitive load theory, that shaped the software and its philosophy. Moreover, I address some pitfalls and mistakes that I made throughout the journey so that people with a similar idea in mind may possibly avoid them. The path that I took from an idea to the current version of Hedy was anything but linear. I will highlight a few aspects of Hedy which I believe have contributed to its success. The history of the language, and in parallel my own maturing as a researcher and teacher, is subdivided into 13 parts. In between I reflect on some key events (both academic and personal) and insights in intermezzos.

2. Part 1: Not Doing My Actual Job

In 2017, I was in the middle of my tenure track at TU Delft. I was not 100 % happy with my work as an academic. While doing my PhD, I had liked research, but upon reflection

later in my career, what had enabled me to do focused research was the clear pace direction that a thesis project gives: every 9 months or so I was expected to deliver a paper, living by the saying "*after the deadline is before the deadline*."

However, when I started as an assistant professor in early 2013, I lacked this direction, focus, and a larger goal in doing research. Soon, I was stuck and unhappy. Around the same time, I came in contact with a local high school that was looking for a programming teacher. I suggested that I could teach, because it seemed like a fun and valuable way to spend my time. In the academic year 2018/2019 I started to teach in this school, two parallel 7th grade classes from summer until the winter break, and one 7th grade class after that; each group of students only followed the program for half a year.

In the first semester, I co-taught with a qualified high school teacher, and our initial strategy was to teach these 12-year-olds using teaching materials that I had been using in the university, in a Python course for non-CS students. After all, these university students also had no prior experience with programming or Python, so the prior knowledge of both groups was similar.

We simply gave the high school students access to the online platform of the university, and had them do little programming puzzles building up in difficulty, based on Python Koans. Koans were originally created for Ruby[1] but have since been recreated for many other programming languages, including Python.[2] We allowed students to take notes during programming (thus creating a *cheat sheet*), which they could use in next week's exercises, thinking that would encourage them to study and take notes.

However, the first two classes that I taught in semester 1 were struggling. Students were not learning a lot; they were mainly battling the compiler, not really enjoying programming and wanting to drop out. At that point we were nearing the end of the semester, and a new 7th grade was going to start after the Christmas break (semester 2). Having seen this initial failure, I realized something needed to change.

3. Intermezzo: Cognitive Science

In the meantime, my interest in learning had increased a lot. Before teaching in the school system formally, I had already taught kids programming in an out-of-school Saturday program connected to the same school, and I had seen similar results. However, part of me blamed the failure of those groups of kids on the loose and not 'schooly' way of teaching. Of course, in a Saturday afternoon club I was not going to drill the kids heavily; if they wanted to play games there, that was fine by me.

However, now that I was teaching in the school system, it was serious business: The pupils I was teaching were in a high school stream that preps for university, so they were smart kids, capable of learning mathematics, and second and third languages. Many of them chose our high school specifically to follow our programming program (called *Codasium*). In short, I could only trace their failure and lack of interest back to my failure to teach them properly.

[1] https://rubykoans.com
[2] https://github.com/gregmalcolm/python_koans

Seeing the kids fail, I came to the conclusion that I knew nothing about learning! Looking back, I realized how weird it was to not know anything about teaching, and reflecting upon that, I began to find that weird. After all, in our field, we often tell students that they will have to learn many new programming languages during their career. For example, in introductory courses, we tell them that it does not matter so much which language they learn now, since they will work with a different one for sure later in life, but that the concepts in these languages will be very similar and that learning a second language will be easier than the first. If that is so, should we then not also tell the students *how* to learn those new languages, for example, teach them the concept of *learning transfer*, both negative and positive (Hermans, 2021)? No CS programs that I know of teach any course related to learning to prepare the students for the learning that they will need.

As such, I (a person with a BSc, MSc, and PhD degree in CS) had no knowledge whatsoever of learning. My failure in the school motivated me to obtain that knowledge. As said, these kids were smart and determined. So why were they not learning?

While teaching, my mind went to my own childhood, when I was teaching myself using my dad's Commodore 64, reading magazines and books with printed out BASIC listings that I got from the library, and hanging out with friends who also liked games and programming. Without explicitly realizing it at the time, I was mimicking that experience of learning, where I had never been explicitly taught anything until I was in late high school; all of my learning until that time had been self-driven, exploratory, and fun. However, as I had been doing initially, just giving the kids access to Python code in the form of the koans did not work at all.

Why? I had no idea! After all, this way of learning had worked for me, for all my friends from uni, for my spouse, basically for all people whom I knew in programming.

After a while I figured that researchers from other fields must have also thought about this question, and through the help of a colleague well versed in developmental psychology, I learned a lot, most notably about the idea of *cognitive load* (Sweller, 1988, 1994). Cognitive load models the cognitive work that a learner's working memory has to execute. When there is too much work to be done, often caused by a lot of newly learned information, the cognitive load gets too high. As a result, the long-term memory cannot remember the learned information, since the working memory is busy dealing with all the new input. In other words, there is something like 'too much of a good thing' when you show kids, say, the concept of a variable but also the complicated new syntax in a complex program. In that situation, they are not likely to remember anything. The reason the exploratory approach worked for me and my friends is probably that we were extremely motivated. (I had no internet access, and only a few games were circulating on diskettes at the time; so if I wanted to get cool new games, there was no other option than to create them. Learning was more of a byproduct.) And, in all honesty, probably our learning was not effective at all. I remember struggling with the concept of saving something to the hard drive for months, reading the one Pascal book I had over and over again, and it just did not click for a kid that had, at that point, spent most of her life on a computer without a hard drive. I would not really grasp this idea until upper high school when a math teacher explained it to me step by step.

A paper that influenced me a lot in those early days was called 'Why Minimal Guidance During Instruction Does Not Work' (Kirschner *et al.*, 2006), which I wholeheartedly recommend the programming community to engage with. The paper explains, in a way that aligned with my struggles with teaching kids and with my memories of hitting my head against the file saving wall, how letting kids figure out stuff alone just, simply, "*does not work,*" at least not if the goal is to get them to understand things, and to get them to a state where they can get stuff done. In Kirschner's paper on the failure of minimal guidance teaching, I recognized not only my own mistakes, but so much of programming education. And research in programming education specifically showed similar findings, Salac and Franklin (2020) have shown for example that, in Scratch, "*student performance on specific questions on the written assessments is only very weakly correlated to specific attributes of final projects typically used in artifact analysis.*" In other words, to paraphrase their paper's title: if they built it, they will not always understand it.

As I had now understood, kids were not deeply understanding the programming concepts in their artifacts, as learning Python was causing too much cognitive load. As probably predicted by anyone with knowledge of how brains work, no long-term learning would be achieved in most kids.

Before diving into my solution to lower cognitive load with Hedy and how I have and have not succeeded at that, let's talk about the issues with programming education a bit more. This part, I believe, applies as much to university teaching as it does to teaching at school.

When we teach programming, typically the main activity that learners do is... programming. We might explain a variable and a loop, but then we don't provide small, simple worksheets (as would be common when kids learn a spelling rule, or when kids learn math; from addition or multiplication to integration, practicing on really small problems on paper is often the way to go), rather the homework exercise after such an explanation is going to be something like: *and now print all prime numbers below 100* or *find the longest ordered substring in this text*. These types of puzzles are then in many situations graded by autograders, programs that verify the output of a student's submission against an oracle.

Computer science is known to have high dropout rates, as high as 40 % (Beaubouef and Mason, 2005). It is my impression that this form of teaching, where programming is the main activity, contributes to that. Kirschner's work helped me understand why in depth: by overloading the working memory of learners with both problem solving and dealing with errors of compilers or interpreters, the brain gets full and retains little. It is funny how so many teachers of programming (myself included) insist that introductory programming courses are about concepts and certainly 'not about syntax,' but then ask students from day 1 in lab sessions to wrestle the complex syntax, and the beast that is an error message. It is almost a form of gas lighting.

Working with the kids in school helped me see another reason why minimal explanation causes issues. The high schoolers kept asking "Why are we doing these silly puzzles?" With a lot more energy than university students that I had taught had done. I realized that these types of exercises do not, in any way, explain what programming is for, or why it

is a valuable thing to learn. The unspoken assumption is that the learners already want to learn programming, so it does not matter what exercises we pick. In my experience, in the 90s and 2000s that was generally true. I went to university just after the dot com bust, when programming was not seen as a great career. The people I studied with, with very few exceptions, were computer geeks that all had been programming since they were in their teens.

But the 7th graders I was teaching were not computer geeks, they did not want to learn programming per se, so they wanted to know why to reverse a string, or why to use a loop. I clearly remember one kids asking me "*Why would I type* `print("Hello world")`*? If I want Hello world on the screen, I can type that in Word.*" They wanted to know why, and explaining that became a core part of Hedy.

I was now at a crossroads. Would I teach the new group in the same way ploughing on, would I make a radical change, or would I give up? The thought of giving up had crossed my mind several times, especially since after the winter break I would be paired with a new teacher, who was not at all so excited to co-teach with a professor in the room. He was a relatively new teacher, and wanted the freedom to teach in his own way. Giving up was tempting, but I was so invested now in figuring out how to do this that I wanted to continue.

I think this interest was mainly spurred by a desire to redress my own previous failures, not only in the school but also in the university. Upon seeing the struggle with the 7th graders, who would relentlessly tell me how stupid the materials were, and upon reading more about learning, I realized that in my university teaching I must have also been causing cognitive load and corresponding frustration. I really wanted to do better.

So when the school offered me to solo teach the next round, I jumped at the opportunity. I would do it alone, and I would do it well this time. I would follow Kirschner's advice and use… *explicit direct instruction*.

4. Part 2: The Pendulum Swings the Other Way

In the new setting I went all out with explicit direct instruction. Every small aspect of Python would be explained at first, and then extensively practiced, on paper. The keyword `print` was first spelled, then brackets, quotes, and commas would be practiced; and pupils had to distinguish correct code snippets from erroneous ones. And not just syntax was practiced, so were concepts. I designed all sorts of paper worksheets with exercises to practice variable declaration and use, loops, conditions, and function calls; see Figure 1.

And it worked… to a certain extent. Surely kids were learning more, that was clear, and they were no longer struggling with frustration coming from misunderstanding. However they were now struggling from frustration due to finding programming boring and stupid. They came into class with the expectation of being quickly allowed *and* able to create cool things. And where groups 1 and 2 had found themselves *unable* to do so, group 3 felt themselves not allowed to do so – by me. Maybe my cure was worse than the ailment?

1) Wat printen deze codes uit? Schrijf het antwoord op in je schrift! Wees *heel* precies.

Let op: **Er zitten ook foute codes tussen!** Als een code fout is, schrijf dan op: **FOUT.**

```
1.print('Hallo', 'allemaal')
```

```
2.print('Hallo)
  print('Allemaal')
```

```
3.print('Hallo' , allemaal')
```

```
4.print('Hallo')
  print('allemaal')
```

```
5.print('Hallo Allemaal')
```

```
6.prit('Hallo')
  prit('Allemaal')
```

Fig. 1. Paper worksheet

5. Intermezzo: Summer of 2019

In the summer of 2019, at the end of my first full academic year teaching at high school, I went to *Dagstuhl Seminar 22302* on Educational Programming Languages and Systems. While attending, I was still processing the two distinct failures of the year before. Just letting kids explore does not get them to learn much (Salac and Franklin, 2020), but excessive instruction does not motivate kids a lot.

I referred back to my research diary on this and other topics, and I came across a note that I made when reading the *Handbook of Reading*. I wrote: “Using a language is changing it. Can we give learners freedom? Can we design a more permissive compiler?”

This idea stuck in my head as I was going to Dagstuhl, and the definitive aha moment happened when I saw a talk about Racket, also a programming language for education (Findler, 1996). Racket uses levels, and in lower levels some options are not available to prevent learners from confusing themselves. For example, in lower levels, functions are not allowed to have no arguments, since the “`()` syntax” is weird and often not what they want to do. By explicitly disallowing this, and mandating functions to have at least one argument, learning will be less frustrating. In higher levels of Racket, more is allowed. When I was seeing the presentation, happening at a poster on the 1st floor hallway of Dagstuhl, I felt the inspiration hit. (And I just looked it up, it is exactly 5 years ago, to the day today!)

What if we could make a language like that, but reverse? A language that starts permissive, and that, gradually, adds more rules and becomes more strict. The idea would not leave my mind, but I also had no concrete plan to move this idea forward.

In the months after the summer, I occasionally pitched the idea here and there, to varying success. I had a master student approach me that I proposed the idea to, but when she went to find a second supervisor in a PL group, the professor there dismissed the idea,

asking whether it was something like AppleScript, and arguing there was not enough novelty in it.

6. Part 3: Vaporware

But the idea kept popping up in my head, and in late 2019 I decided to start working on this idea, and I submitted a talk to the *Booster Conference*, a conference for programmers and people in related professions. I had been there before and it was a nice group of people that might be excited about a vague idea, more than I expected from an academic workshop. At that point there was no code, no concrete vision, basically nothing. But I thought, if the talk gets accepted, I can sketch the vision, get some feedback, and think about it a bit more.

Asides: sometimes I really think that CS academia should somehow allow for more of this. Conference proceedings in CS are basically journals, so you can't easily submit an abstract and get feedback, as common in other fields. Instead, you need to submit a full paper. Of course there are workshops, but they are often also paper-based (albeit shorter papers).

Anyway, I submitted a talk to Booster called *A gradual programming language*. Initially I called the talk *An incremental programming language*, but I really liked the notion of gradual typing, which I feel captures the same idea of freedom initially and more strictness when needed, so that I later changed the title. I learned the talk got accepted in December 2019. Oops, now I needed to really get going with something! The acceptance of the talk gave me a positive excitement about the idea (and I am so, so happy I did not try to create something *before*, for reasons I will explain later in this paper).

On December 9, I created the first version of Hedy, a shoe-stringed project running a very simple Flask web server, a simple grammar created with Lark,[3] and the Ace editor[4] on a non-styled HTML page. Hedy code was parsed with Lark, transpiled into Python. The corresponding Python was then ran in the browser with a tool called Skulpt,[5] a Python interpreter written in JavaScript. I addition to the editor, each level of Hedy came with a very simple explanation of what types of programs could be created with it; see Figure 2.

I put the code on GitHub, and Hedy was born![6] I did this without any thoughts of starting an open source community. I had never really cared for that community because of the raving sexism I had seen happen there, and in particular the behavior of some of its icons, peaking in things like the 2015 controversy about FOSDEM's Code of Conduct[7] or the plea from a leading open source figure for developers to kill themselves.[8] Plus, I never envisioned Hedy to be anything 'real' anyway.

[3] https://github.com/lark-parser/lark
[4] https://github.com/ajaxorg/ace
[5] https://github.com/skulpt/skulpt
[6] https://github.com/hedyorg/hedy/commit/79cd42ed7cf56bed9bcb493460a1f15b5aa139ff
[7] http://www.sarahmei.com/blog/2015/02/01/the-fosdem-conundrum
[8] https://forums.freebsd.org/threads/linus-to-opensuse-devs-kill-yourself-now.30414

Hedy: Level 1

In Level 1 kun je deze commando's gebruiken:

- Iets printen met print. Bijvoorbeeld: print Hallo welkom bij Hedy Probeer dit
- Iets vragen met ask. Bijvoorbeeld: ask Wat is je lievelingskleur Probeer dit
- Invoer herhalen met echo. Bijvoorbeeld: echo Jouw lievelingskleur is dus ... Probeer dit

```
1 print Hallo wereld
```

Voer de code uit

Fig. 2. Hedy's initial design

However, putting software on GitHub allowed for both free (at the time) and extremely easy deployment to Heroku. Hence, Hedy lived on GitHub from the first commit, without its open source character being an explicit decision at any point in time.

Between December 2019 and March 2020, when I was set to give my talk launching Hedy at the Booster Conference in Norway, I made a few changes to the code, but nothing all too spectacular. It is interesting now to go over my GitHub commits of the time and examine the changes I made in those three months. GitHub repositories are often used in our community to examine software engineering processes, to predict what bug report will be closed, or what line of code is likely to be refactored in the future (Mohamed *et al.*, 2018; Ortu *et al.*, 2019); but rarely we talk about GitHub as a tool that enables later *narratives* of software creation. Had I not developed Hedy on GitHub from day 1, which as noted before I only did because it made deployment easy, I would not have such a rich source of history for this paper.

7. Part 4: The Pandemic

As I was low key pitching Hedy in the weeks leading up to the Booster Conference, the pandemic was starting. Soon it became clear that I could not go to Norway to pitch Hedy at all, because the conference would be cancelled. However, as much as the conference has been a catalyst for me really fleshing out the Hedy idea and eventually building it, it was no longer really needed (as a true catalyst, it was not needed as an ingredient, just as a starter). I had seen how easy it was to create a simple gradual language, using existing grammar and web frameworks. This led me to believe that it was feasible to create something like that (although there are a few design decisions I have come to regret), and, having seen the language in action, I just felt I was on to something: a beginner friendly Python, with built-in explanations. This was going to help people.

When schools were closing as a response to the pandemic, Leiden University, where I worked at the time, wrote a lovely press release encouraging Dutch parents, stuck at home, to try a new and exciting programming platform for kids. This message, and my reach on Twitter, then about 17 000 followers, helped quickly spread the word about Hedy, and parents indeed started to try out the tool with their kids.

Fast forward to the end of 2020, and 100 000 programs would have been run on the site; but due to the ongoing pandemic, it would take until the summer of 2021, a full 2 years after the idea germinated in my head, until I would see kids program Hedy with my own eyes in real life.

Parents were not only trying out Hedy with their kids, they also started to send in comments, either by mail, or via GitHub, fixing typos and updating content. In the first 2 months, 30 pull requests (PRs) had been submitted. It was extremely exciting (and extremely scary) to have actual people use something I built at that scale. Nothing I had done in my career prepared me for that.

8. Part 5: Hedy and Academia

In early 2020, before the pandemic started, and just a bit after I had written the first Hedy code, I had also written a paper about Hedy and submitted it to ITiCSE (the ACM conference on Innovation and Technology in Computer Science Education). This paper contained the initial design goals of Hedy, nicely rooted in cognitive theory of not overloading kids while learning, and a description of the implementation. In my original field of software engineering, I had seen many such papers where an innovative idea paired with a prototype was enough to get published in high quality venues. However, in CSed, it does not work that way, or in any case it did not work for this paper, and it was rejected in early March 2020, for being a nice idea but with a lack of proper evidence. In the same week, a high profile grant proposal I was working on (comparable to an NSF career grant) was also rejected, as were three other proposals I was involved in a week later.

So at the start of the pandemic, my career prospects were not feeling great, and my mood too was, to say the least, not great. Working on Hedy was not only fun, it was a good distraction from ‘my actual job.’ I think if I am to ever reach the point where I get to reflect back on my career, the title of my retirement speech (which where I live has a formal character, and is called ‘uittreerede’) will be “*my research program is what happened to me when I was busy not doing my actual job.*”

Rather than looking at the shambles of my failed proposals, it was a preferred choice to grind on grammars and (with the help of my web savvy husband for which I am forever grateful) make the Hedy website look better than the plain HTML site (which I created initially, and which was the only thing that lay in the realm of my web development skills).

When, after the pandemic started, the schools closed, the press releases were written, and the ICER deadline was postponed, I figured I could try my hand at a Hedy paper again, and now include some data. I manually annotated the first 10 000 programs that had been created on the site in the first few weeks after launch, providing an insight into

how learners were using Hedy. Manually looking at all the programs was mind numbing work, but it beat thinking about my grant proposals; and also, I was not just seeing the errors that kids made, I was additionally seeing the sheer joy they were having, creating stories and quizzes.

This was what I had intended: to create a system in which kids could, without battling the unreasonable compiler, see how fun programming could be as a means of expression and communication.

I did think a bit about whether ICER would be the right place for this paper (a fact that would later inspire my thinking about the programming language community extensively). Was it sensible to submit a paper to a CSed conference that was more talking about innovation in PL design than it was about teaching? But I was too afraid to submit Hedy to a PL venue, I know I would be laughed at, so I went for ICER, and was not only accepted (Hermans, 2020), but also awarded the *John Henry Award* for most innovative work.[9]

The meta review also mentioned the ill fit and commented on the fact that this was more of a PL paper than a CSed paper, but they ended up accepting it because the target was education.

This was instrumental in my further exploration of Hedy. Somehow I had made this hobby project, which I had initially seen more as part of my teaching job, the one I took to get away from academia, into a part of my academic work. With an ICER paper, Hedy now became (at least partly) 'my actual job.'

9. Part 6: Hedy's Design Goals

In the rejected ITiCSE paper, and later in the accepted ICER paper, I formulated a number of design goals for Hedy. It is interesting to revisit them now. Let's go over them one by one, as in a close reading, reflecting on their background, implications, and my current thoughts on the topic.

In the original paper, I write that the "goal of the Hedy language is to start with a language that is as simple as possible, and to add more and more syntax as the levels progress."

I find it interesting to see the focus on syntax now. Not that it is no longer true, but when I explain Hedy nowadays, for example in a talk I did at the developer conference *StrangeLoop* in 2022, I always explain that Hedy levels grow in both syntax *and* concepts.

Much as in natural language learning, we change the rules as learners advance, first kids write only letters, than words, and then sentences starting with uppercase letters and ending in fulls stops. Those changes are not just syntax changes but are also adding expressive power to the language. Capital letters and full stops, for example, enable sentences that span multiple lines.

I don't know why I put most focus on syntax in the early paper, but I suspect it had something to do with distancing myself from languages growing in semantics like Racket,

[9] https://icer2020.acm.org/info/awards

and a desire to root the ideas of Hedy in one form of prior work, namely punctuation teaching. I think I was trying to say: look, if we want to teach syntax and semantics in programming courses in a solid way, we can learn from teaching writing and punctuation, which has been studied extensively.

The strategy of taking an idea and moving it to a different context had been successful in my career (e.g., bringing code smells from code to spreadsheet formulas (Hermans *et al.*, 2012a,b; Hermans, 2013; Hermans *et al.*, 2015a,b), why not try it again? Even though my thinking is now much more wide, I do think this strategy still makes sense given the way publishing works. Taking one idea from a distant field seems 'sellable' to computer scientists; taking more and more, and combining them in intricate ways is a lot more risky, because it requires more work from both the author of the paper and the reviewers.

With that, let's examine the initial six design goals of Hedy (Hermans, 2020):

1. Concepts are offered at least three times in different forms.
2. The initial offering of a concept is in the simplest form possible.
3. Only one aspect of a concept changes at a time.
4. Adding syntactic elements like brackets and colons is deferred to the latest moment possible.
5. Learning new forms is interleaved between concepts as much as possible.
6. At every level it is possible to create simple but meaningful programs.

Earlier in this paper I described two distinct issues with programming education: (1) causing too much cognitive load and (2) not explaining the *why* of programming. Looking back upon this list of 6, it is now clear to me that items 1 to 5 focus on the first problem of lowering cognitive load, much as the overarching goal focused on syntax. Only the final goal of creating meaningful programs addressed point 2. This too is a result of the workings of academia or maybe even of the PL community. I was focused on the creation of a language, not on that of a learning platform where the role of the exercises would be a lot bigger. Creating a programming language is seen as hard, which gives status, making puzzles and worksheets is easier; see also Hermans and Schlesinger (2024) for a description of this aspect of PL culture.

Still, I am really proud of the care we put into the cognitive design of Hedy, trying to create a language with the explicit goal of being easy to learn, driven by cognitive science. Our ideas of step-by-step learning are solid, rooted in scientific work, and helped contribute to the success of Hedy in a large way, even though we are now slowly venturing out to other aspects.

Looking back, some important design goals are missing. One that stands out now, if you look at our current website, is the lack of a target audience. When I was building Hedy, even though I was envisioning to use it in my school, I was thinking of the kids as main users, visualizing a kid sitting alone behind the computer, consuming the explanations, creating programs, and working through the levels. I was envisioning, in other words, my own experience of learning to program self-directed, but then times 30 within a classroom.

It seemed so obvious to me that learning to program was something one kid did alone behind a computer; I did not even have to write it down explicitly. It is funny how a question so obvious to any first-year business student (for whom is this thing for?) did not even occur

to me. It is also funny how my views on *for whom Hedy is for* have entirely changed over the last 5 years, but we will talk about that change later on.

And I am sure, much as in a large code base, the lines of code that have been changed a lot in the past are good predictors for lines of code that will change in the future; these aspects of Hedy will change and deepen more over the course of the next 5 years. I think that is ultimately what makes educational work so interesting: sometimes one experience you have with one struggling kid or coworker can totally change your views on things. Doing work with people not just changes the world, it changes you with it.

Over time design goal 6 has often been a bit at odds with the other 5 goals, for example, after a while we added drawing with a turtle, and more recently we added playing musical notes. Those features surely add to the expressive power of the language, not in the sense that it adds concepts, but in the sense that is makes room for more self expression. But for musical notes, we are not offering different forms, you can do ‘play C4’ from the first level. If we would limit music to just a few notes in early levels, it would not constitute meaningful programs. Over time, learnability in the very strict sense of the early days has been often traded in for a bit more ‘fun’ in program creation, and the line remains hard to walk. But I am getting ahead of myself… let’s return to the history of Hedy. What happened after I created the first prototype in early 2020?

10. Intermezzo: All Things Hedy Could Live Without

Recently, a programmer working on a similar tool for programming education asked me how their project could also reach users as well as Hedy did. Not that my $n = 1$ anecdote should be taken as the holy grail, but if asked what we did, I think an interesting way of looking at things is to describe the things we did not have for a long time. For example, we did not implement the option for users to log in and save programs until the end of 2020, and we only had 7 levels implemented until early 2021. The biggest omission I think is the fact that we did not have a syntax highlighter until the summer of 2021. By then I estimate around 250 000 Hedy programs had been run. Two of the three core features (built for teaching and localization) were not implemented until the end of 2021 (more on those two features later).

Of course this annoyed users (not so much kids that did not know what they had been missing, but surely programming parents reading along), but not having user accounts or syntax highlighting did not stop us from having users, and iterating on the product based on user reports. Because in all the months in between I was learning from users; responding to emails and issues on GitHub, analyzing common errors, and giving talks about Hedy (mostly online, because it was still during the pandemic) to interact with various communities, from teachers at the American Computer Science Teachers Association (CSTA) to PL people, for example at Strumenta.[10]

[10] https://d.strumenta.community/t/virtual-meetup-hedy-a-gradual-programming-language/1169

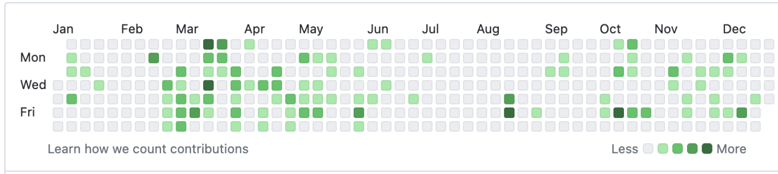

Fig. 3. My personal GitHub activity in 2020

11. Part 7: The Trough of Disillusion

The *Gartner Hype Cycle*[11] describes phases of technology adoption, but I feel it also adequatly captures personal excitement about technologies, at least about mine. And after the peak of inflated expectations, which I surely was feeling, caused by the accepted ICER paper, and the initial excitement of users, a personal trough of disillusion followed; I was losing my enthusiasm to work on Hedy, which is visible in my personal GitHub history;[12] see Figure 3. After the launch in March 2020, my commits grew ever sparser. Again it is interesting to note how much of this paper, and other reflections on Hedy, has been enabled through GitHub's history.

These feelings were in line with many of the projects I had undertaken in my career before. I love creating new things, but seeing them through is… not my strong suit. Nor, as I want to clearly point out here, is academia encouraging seeing through things. It is hard to get follow-up work on tools published as it is quickly seen as incremental, even if the initial work is very novel. Philip Guo explains this process in painstaking detail in his memoir (Guo, 2012), a work by the way I think every CS professor should read regularly. Whether I fit academia because I am not a natural 'follow through-er' or whether academia made me that way, I find hard to tell.

Anyway, there were, in addition to my natural tendency to be more excited about new things than about polishing old ones, two problems hampering my excitement in continuing to work on Hedy. I did want to program on Hedy more, because I still liked (and do to this day) the concrete activity of programming over the vague and abstract work of grant writing.

The first issue was with indentation. From a certain level on, we wanted to introduce Python-style indentation. Our grammar framework Lark supported that, but not together with the Earley parser we relied on, which was direly needed as it provided priorities in the grammar necessary to process our often ambiguous programs. After all, many of the syntax bells and whistles that we removed, like brackets or commas, are normally used to disambiguate programs. However, using Earley meant we could not use the built-in indentation, which annoyed me. I did not feel like hand-crafting indentation if the Lark framework, in theory, allowed me to get it for free. This lead me, for a few weeks if not more (the exact time is lost to history as my attempts were not committed and thus not

[11] https://en.wikipedia.org/wiki/Gartner_hype_cycle
[12] https://github.com/Felienne?tab=overview&from=2020-12-01&to=2020-12-31

preserved for all eternity), to keep bashing my head against combining Earley and the indentation implementation.

Ultimately I did end up building an indentation system myself, which is still haphazard and ugly (but it works, which at one point overtook any other concern); but it would take until March 2021, so almost a year and a half after the start of Hedy, for me to get that in order.

In the 16 months in between, the levels of Hedy just ran up to 7, and we did not have loops that needed indentation, only loops on one line:

```
repeat 5 times print hello
```

I remember being a bit ashamed of the simple nature of Hedy for a while, but in the end, it was actually fine. The small scope was enough to get sensible feedback, and some users and contributors.

The second issue was of different nature. I already mentioned that my initial goal was, even if implicit and not even vocalized to myself, to build a system for self-directed learning. But the goal had also been to show what programming is for. I did this with a set of exercises, following up on each other, using an increasing set of programming concepts. As an example consider the exercise of making an interactive story, first by just printing sentences, then by using variables to create different stories, and then a condition to choose a good or bad ending. Or in a *rock, paper, scissors* game, first only having the computer choose an option using a list, and then deciding with a condition whether you have won the game (we later expanded the possibilities from just outputting text to also creating drawings with a turtle and musical notes).

To motivate learners, the idea was to then give the kids the option to choose from the different types of programs. However, this lead to the question of how to present the options to the learners. Would they choose one exercise, such as creating a story, and do all sub-tasks for that exercise? That would mean just one exercise per level, robbing them of much needed repetition and practice which were core to the system. Alternatively, they would have to do all exercises in one level, the story, and the rock, paper, scissors, etc. forcing them to work through unexciting exercises. Figure 4 shows the current user interface where all adventures in one level are shown, but not mandatory.

Independently of the choice of order, there was the issue of how to present the, inherently multi-dimensional, sea of possibilities to novices, who are prone to get overwhelmed? This issue, to which there is not really one good answer since all options have pros and cons, caused me to overthink and not do anything.

12. Part 8: From a Solo Project to a Community

In a different world, this might have been the end of Hedy, or at least of my sustained energy for it. But there were a few lucky strokes in those early days. The first was that we kept gaining more Hedy enthusiasts, both in terms of users and in terms of contributors on GitHub. In the first year, 100 000 programs had been run; Spanish, French, and Portuguese

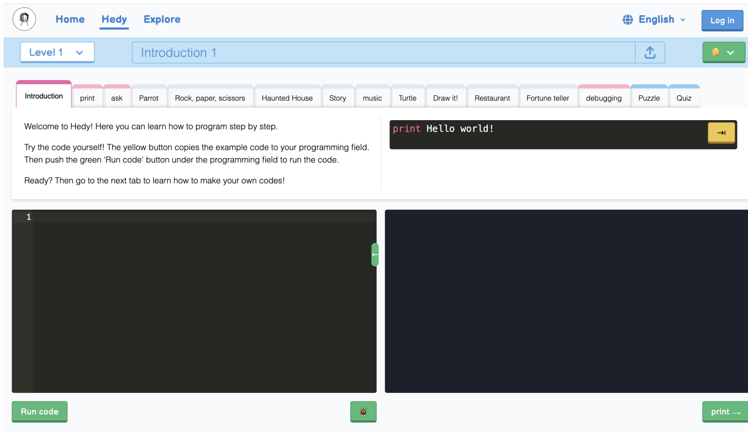

Fig. 4. Hedy level 1 with different adventures in different tabs

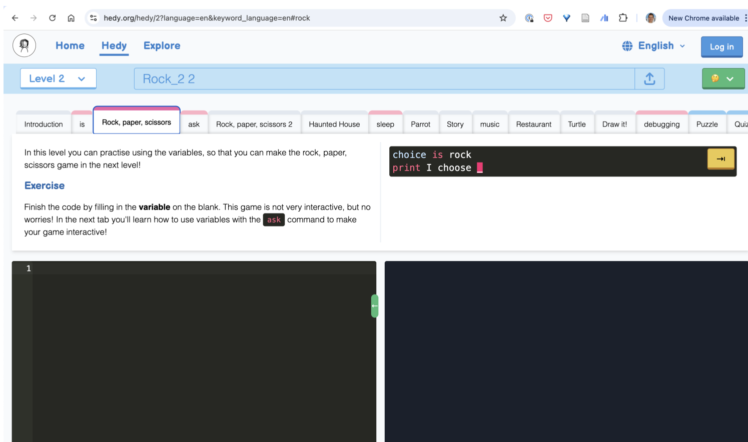

Fig. 5. Hedy level 2 where some adventures return, like *rock, paper, scissors*, to show new concepts

translations had been added and 70 PRs had been created. But users and contributors alone did not get my thinking head off.

Secondly, I got a little sum of money because I was selected as teaching fellow at Leiden, which allowed me to hire the first part time programmer on the project for a few months. This meant we could add some long-desired features, such as allowing users to log in and save their programs; but it also meant I had to make the shift from a small project on which I worked alone, occasionally accepting PRs from people, but mostly from people I already knew either in real life or because they emailed me beforehand, to a project with more serious process management.

Had I never hired the first programmer needing me to clean up code and practices, we might have never been ready for what was to come. Our first employed developer also helped in forcing a decision in the longstanding 'how to deal with exercises' problem,

because now that he joined the team, I needed to vocalize my thinking.

In his excellent book *How Music is Made* David Byrne, singer of *Talking Heads*, reflects on his time in the music scene in the Bowery in New York in the 70s (Byrne, 2013), and describes the aspects that lead to a music scene emerging, which are great guidelines for building any community. He notes for example how the owner of the bar they frequented, *CBGB*, allowed performers to hang out there for free if they were not performing, so that anyone on stage always had audience to get feedback from. The programmer that I hired was my first audience to vibe with, and even though the solutions for the exercises problem were not perfect and rewritten a few times, after he joined, someone was witnessing my lack of decision making, helping me to move along.

13. Intermezzo: Hedy as an Open Source Project

As I mentioned before, I had never planned Hedy to be an open source project. My understanding of how open source worked was mainly limited to the interactions I had seen in popular CS blogs and on social media: people, mostly men, who (until this day) defended that the way to interact with each other was to virtually shout at each other. If you can't handle that, just go away, or, as said, kill yourself. Despite having met amazing people in open source, I did not think this was the world I would enjoy working in.

A second issue I had with open source software, thinking back to my university days of endless googling and hassle to get something as simple as a mouse driver to work on even a mainstream version of Linux, was that I did not see this community making software with a deep love for users or usability. Most software I had enjoyed using has not been open source, whether those were games on proprietary consoles, macOS, or Visual Studio which in grad school I had preferred over Eclipse that never seemed to run well. In short, I did not think 'open source' as a concept cared about programmers or users, two groups that I did and do care about.

Luckily, my early experience with open sourcing Hedy was actually quite good. In the early days, people sent in issues, but by and large they were really nice and encouraging. Parents sent in ideas that they motivated with evidence like "my kids were confused by this and that," and people added code through PRs that was useful and valuable. 'Random people on the internet' helped me achieve my mission, pretty awesome!

Still, looking back there were a few things that I could have done better had I understood how much larger the community was going to be, so here are some words of advice for academics (or other people) that want to run a programming language in an open source fashion.

In her book *Working in Public* (Eghbal, 2020), which I surely should have read before I started Hedy, Nadia Eghbal describes that there are, nowadays, four different types of open source projects: toys, clubs, stadiums, and federations. Toys and clubs have a low number of users, while federations and stadiums have a high number of users. Federations and clubs have high numbers of contributors, while toys and stadiums have few contributors. It is a good idea, when starting a project, to decide what type of project you want it to be, because each requires a different type of organization and accompanying rules.

Many academic programming language projects, such as Scratch[13] or Snap![14] are organized as stadiums, in which only the core contributors create changes or PRs, or only theirs are accepted. Eghbal states that previously most open source projects were stadiums, with a few contributors, and many users. She attributes the new forms of projects to the popularity of GitHub. In the 'old days,' when open source projects communicated mainly via mailing lists, and patches were harder to create, the threshold for participation was higher, both because of social reasons (responding to a mailing list might feel scary if you don't know anyone) and for technical ones (mailing lists might require to first be admitted as a member, and a simple change could not, as it can now, be created from within the browser). As such, projects now suffer from 'drive-by' contributions, people that are not members of the community (formally or informally) and that might lack the deep understanding of the context and software of a project. Hedy too has such contributions, with as absolute low point a PR[15] which changed only the use of quotes in our license file. In retrospect I should have simply closed such silly PRs and removed people with the wrong intentions from the community. Every ounce of energy spent on managing silliness is a distraction from building great things.

One final dynamic that Eghbal described which I also experienced is that the main maintainer of a project is often quickly overwhelmed by issues and PRs, even by interesting ones adding value on first sight. Early in the project I got so excited about people helping me, that I did not always think deeply about whether a certain PR was 'good enough' in terms of code quality, but also in terms of 'is this where we want to take this project?' One feature I regret adding, for example, are achievements. Kids can get badges for some activities, like saving a program, or for using certain programming concepts. I see how achievements can be fun and motivating in many situations, but they don't fit with the calm and step-by-step way of learning that Hedy is meant to offer. A pop up saying 'you got an achievement' is adding cognitive load, which we are trying to lower. At the time I was still enchanted by the notion that people wanted to help, and excited by their excitement, but I should have balanced the needs and focus of the project with the enthusiasm of one contributor.

Having clear guidelines for what a project aims to do (and what it aims not to do) can help to keep a community (including the main maintainer) sharp, and outline which ideas might work. I should have thought more deeply about what I was building, and should have communicated this to the community earlier and better. In other cases, I got excited about a feature, but I did not deeply think of the technical debt (Brown *et al.*, 2010) it would create, or what future maintenance it would require. For example, a novice programmer connected Hedy to a translation system called *Weblate*,[16] which I loved as it substantially lowered the effort of translators to submit translated lesson plans, but he did this in a naive way that several contributors and I spent months of efforts untangling, and we are still not happy with it. Thinking about how to maintain features in the future should not be a frequent thought.

[13] https://github.com/orgs/scratchfoundation
[14] https://github.com/jmoenig/Snap
[15] https://github.com/hedyorg/hedy/pull/1909
[16] https://weblate.org/en-gb

Finally, I could have organized the community better from the beginning. Early on, and for quite a long time, all of the communication was between me and the other contributors, rarely between other contributors directly. This led to more of a stadium model in which I was the core maintainer. After a while, we started a *Discord* community, where it was easier for me to bring people together in themed channels; and even later, we started weekly online hangouts in which decisions were made in a more open and democratic way. There are many decisions where I just do not have a strong opinion and these meetings allowed me to say: “I don’t really know or care, what do you all think?”

If I can give open source maintainers just one word of advice, having a weekly meeting would be it. It seems like such a cheap investment in creating a community, but it is absolutely worth it. Even if people rarely join the meetings, having them demonstrates what type of show you are running.

14. Part 9: The Trouble We Were Expecting

As I was interacting with teachers and learners that were trying out Hedy online in the first year after its launch, I received a lot of feedback. One of the most commonly received comments, often from parents with a background in programming, was: “*Aren’t you frustrating kids by changing the rules on them?*” This was of course an expected frustration, but this aspect often gets addressed in questions about Hedy, so this paper is an excellent opportunity to refute it.

One answer to the criticism of changing rules is “Yes, this is frustrating.” However, this frustration is one I was, and remain, willing to exchange for the frustration of having to learn everything at once. Another answer is that this is actually a super common teaching strategy in many different things we teach to kids.

In writing, one of my inspirations for Hedy, we allow kids to write without capital letters at first, only later teaching the rule that sentences (in most Western, Latin-based languages) start with capital letters. We teach without punctuation at first, only later adding rules for ending sentences in full stops, and separating parts with commas. This is all common practice and might frustrate kids, but beats the alternative of requiring correct sentences from day one. And in math, a topic more close to programming for many CS teachers (although recent evidence shows that programming might be as similar to language as it is to math (Prat *et al.*, 2020)), we do similar things, first explaining minus as ‘take away’ and equating 5 take away 6 to 0, since if one had 5 cookies to give to 6 kids, 0 will be left on the table. Only later do kids learn about the option of having a cookie debt to the cookie bank. In division $8/3$ is first equal to 2, remainder 2, then to 2 and 2/3 and then to 2.6666666… All of these three options have consistent mental models given what kids have learned. In physics, a teacher is common to first explain electricity as something that ‘flows through the outlet’ as water flows in a river, allowing for an initial understanding close to what kids already know, before explaining where that mental model breaks, and then going into depth about electrons; and we could expand this list a whole lot.

In other words, isn’t it weird that programming has always insisted on teaching the whole language at the same time? Why could that be? Of course, I don’t know exactly what has constituted to this situation, but one can hypothesize.

15. Intermezzo: Why Do We Teach Programming in a Full Language?

Comparing and contrasting the ways of programming with the ways of other topics as above, this question becomes too important to not cover in depth. I think at the deepest level, a lot of the ways of teaching programming are caused by the fact that teaching programming originated in universities rather than in schools, while almost all other topics have been taught in both, with most didactic theories stemming from research on school teaching, like language, math, and physics.

This fact has a number of different implications that all contributed to the weird form of teaching. One result is that most teachers of programming are also, to a certain extent, programmers. They might use little scripts to analyze grades, program autograders (more on those later), or, if they are active researchers, use all sorts of programming in their work, from building prototypes to using proof assistants. Even if not actively programming, many CS researchers identify as programming persons, much more than a physics teacher will feel like a 'physicist.' As such, programming feels so familiar that the 'weird' syntax causing struggles in learners feels almost natural, and empathizing with these students becomes hard. I had a university student tell me they found it unfair that the syntax of a list comprehension in Python (such as `[x for x in letters if not x == 'x']`) does not require the use of a colon, while a regular `for`-loop does. Such confusion seems remote to someone well-versed in syntax.

Another reason is that by and large people teaching programming have no formal background in any teaching theories. While high school teacher qualification programs, in addition to proficiency in the course being taught, also include pedagogical and didactically courses, and sometimes also developmental psychology, if you are a CS researcher, you will be exposed to none of that. Even as teaching of programming in recent years has been moving from universities to high schools and even elementary schools, people from universities are often involved in shaping curricula and teaching methods. For example, the end terms in my country dictate that 17 and 18 year olds should be able to evaluate algorithms by performance and elegance, something that I think many educators will agree with me undergrads can hardly properly do. In my opinion, it shows a total disconnect from actual school teaching practice. (And those end terms are even required for high schoolers graduating after 11th grade, who follow a program not preparing for university-level education.)

These two factors (1) being knowledgeable about programming, and (2) lacking educational training lead to many CS teachers teaching at a level way too advanced for students to follow, as research also shows (Altadmri and Brown, 2015). Part of those struggles are due to the way of teaching (which we will cover later in the paper), but part of it also is due to the complex nature of programming languages, which, combining syntax, semantics, and concepts, will be hard to learn as quickly as CS people think it can. Compared to the massive amounts of practice spend on learning to read, write, or perform mathematical algorithms like tail division or integration, we might have unrealistic expectations of programming language learning.

One might wonder why there is not more outrage about dropout statistics for CS programs, but there is a second dynamic at play here (which I extensively cover in my up-

coming paper about PL design (Hermans and Schlesinger, 2024)), which is the discourse in the PL community to see doing hard things as a desirable state, rather than a problem to be solved. Programming *is* hard, and that is the way it should be. If you can't quickly learn programming, which is 'just syntax' after all, you are just not made for it.

In such a dynamic it is hard to create novel programming systems that ease the burden of syntax – I have over the years experienced a lot of push-back, for making programming 'too easy' or for wrecking kids brains with 'wrong syntax.' As work along those lines is hardly done, full industry programming languages remain the norm.

16. Part 10: The Trouble We Were Not Expecting

After the summer of 2020, in which I presented Hedy at ICER and won the John Henry award, I had realized Hedy had potential for academic work, and I wanted to expand my informal learning with a formal study; with the type of students which I had originally created Hedy for: 7th graders in the high school where I taught. In February 2021 our colleague Marleen, an experienced teacher, taught 6 online lessons to 39 learners (online because schools were still closed physically due to the COVID-19 pandemic). These students were in the same grade that I had taught Python to in prior years, so together with other prior teachers, we could do some comparisons. This group had no programming experience in school; since learners in our school only get CS for one semester out of two, this was their first encounter with programming lessons.

The user study was simple in setup, with three research questions: (1) what the kids liked about Hedy, (2) what they disliked, and (3) what one thing they would change about Hedy. The likes and dislikes were somewhat predictable. Which is not bad, I want to add; confirming what you think works is an important part of doing science and I hate to still so often see papers being rejected at conferences because their results are not 'surprising.' Kids liked the step-by-step approach, because it helped them figure out what to do; especially girls (more on programming and gender later). And kids, as we had expected, disliked the fact that sometimes grammar changed from one level to the other, as we discussed in Section 14.

The final question in the user study was how Hedy could be improved, and some of the answers there were also expected. Kids were struggling with error messages that were sometimes cryptic. Our paper contains an epic example in which a kid did not know how to understand the error message "`You typed ',' but that is not allowed`," because they interpreted the comma as an actual comma, making the message read "`You typed, but that is not allowed`," confusing the kid, because why on earth would *typing* not be allowed? These types of comments were pure gold to improve Hedy in small ways. Without a kid vocalizing that error message for me, it would have taken me forever to figure out that confusion; but the changes were not radical. One comment though, which kids were expressing in the lessons, and that one participant answered in our survey, was: "Why can't we program in Dutch?"

Since I has designed to use Hedy with my own middle schoolers in the Netherlands, Hedy has always been partly localized. The website (UI, explanations, error messages)

were first available exclusively in Dutch (as shown before in Figure 2). Only after a few weeks I added support for multiple languages, and just a week later again someone added the first translation of content, into Spanish.[17]

So learners in my class and in the experiment would program in code that mixed Dutch text and Dutch variable names with English keywords like this:

```
naam is ask 'hoe heet jij'?
if naam is Hedy print 'Hoera!' else print '...'
```

So it was maybe logical for learners to be puzzled why one little part of Hedy, the keywords, were not also in Dutch. It would be easy now to proclaim I was promptly swayed by this argument, but I was not. I thought firstly that localizing a language would be a lot of hassle, for too little gain. Couldn't the kids 'just' learn English? After all, proficiency of English is extremely high in the Netherlands, especially for a non-English-speaking country; the EU reports that 90 % of Dutch people speak English.[18]

The kids however had good arguments, of which maybe the best was simply "Why not?" Yes, why not? Things don't have to remain as they are now, and the kids made me realize that maybe I was too easily accepting a status quo. Plus, the direct access that they had to me, Hedy's creator, helped them to envision the possibility of change. One kid said to me "You build it like this, you can build it in Dutch, too!" And, being a school with a large immigrant population, a lot of kids spoke Dutch already as their second language, and were not at all as proficient in English as I would expect, being a middle-class white person. At home they would not watch English movies and cartoons, as I had once done, but watch TV in the language of their parents. These arguments planted a seed in my brain, but I did not act upon it immediately.

17. Part 11: Hedy in Non-English

My thinking about non-English programming did not start with this experiment in early 2021; in fact, I had thought about it before, somewhere in 2018 or 2019, because of Alaaeddin Swidan, my grad student at the time, with whom I had worked on analyzing variable name usage in Scratch (Swidan *et al.*, 2017). He is a native speaker of Arabic, and in those days he had expressed an interest in exploring whether we could see differences between kids programming Scratch (which supports localization of keywords) in Arabic and kids programming in Dutch or English. We had thought about it a bit but we did not pursue it further for a variety of reasons, from him needing to finish his thesis to the difficulty of analyzing the language used to program. Scratch allows the presentation of a program in any language, so it would be hard to know what language was used to create it originally.

But when the idea surfaced again, now in a new context we had more control over, I reached out to Alaaeddin and we extensively discussed the idea. He showed me other work

[17] https://github.com/hedyorg/hedy/pull/24

[18] https://op.europa.eu/en/publication-detail/-/publication/f551bd64-8615-4781-9be1-c592217dad83

on Arabic programming, including قلب (pronounced 'Elb' or 'Quelb' meaning heart)[19] and I got excited! My thought had been that making the keywords non-English would enable learning, because kids did not have to learn a new language. This would fit Hedy's philosophy of cognitive theory perfectly: not having to learn new words but use familiar ones would lower cognitive load.

However, Alaaeddin quickly convinced me that there were much more interesting reasons to consider this, because how many keywords are there, maybe two dozen at most? Of course kids learning to program would quickly acquire those words. One such notion that he pointed out, which I growing up with a Latin language had never really thought of, were the ergonomic implications. One ergonomic issue is that a computer keyboard can only work with one alphabet at a time. If you want to switch between the Arabic, Hebrew, or Cyrillic alphabet, you have to use a hotkey or click a button with the mouse. Imagine that you want to print a string in Arabic, which seems likely if that is your native language.

A simple program like

```
print مرحبا بالعالم
```

meaning `print hello world` requires a keyboard switch in the middle of the line, which is doable but very distracting. More complex scenarios in which you would use non-Latin variable names (supported by several programming languages other than Hedy, such as Python) need several keyboard switches back and forth.

Another aspect of localization which I had never considered, but which, much like the error message containing the comma, would ring true immediately upon hearing it, is that there are large groups of people on earth, for which using English is not an enjoyable activity. English speaking people had, not so long ago, ruled their countries with an iron fist or even in modern days continue to wage war on them. Being required to use English for an activity like programming underscored the fact, uncomfortable for many, that in order to succeed in the world, one has to participate in English language and culture. Being liberated from that made programming not easier, but freer, opening up room for their own culture.

I would see such a dynamic with my own eyes much later, when we had implemented multi-lingual support. Firstly when we were rolling out Hedy in a national programming initiative in Botswana (Tshukudu *et al.*, 2024) and teachers remarked upon the value they derived from being able to teach kids programming in Setswana, showing their pupils that their language too is the language of technology. Secondly, closer to home, when I allowed kids in my own school who do not speak Dutch at home to use Hedy in their own native language, in the case of my classroom Turkish or Arabic. The light in the eyes of one boy when I told him he was allowed to use Arabic in school (which the school usually disallows so that all kids can interact with each other, a policy that I can understand and normally always enforce when concerning spoken language) was just magic. I could not help myself and told him I had personally put a lot of effort into making Arabic work, and he could hardly believe it, asking me "Do you speak Arabic then, Miss?" I told him I didn't, but I just wanted kids like him to be able to enjoy programming and he was baffled.

[19] https://github.com/nasser/---

Non-Dutch speaking kids in the Netherlands learn at a young age that their language makes white people uncomfortable and that they should thus adapt. Don't speak Arabic with your friends on the tram or people will frown. That a white person had put in months of work so that he could do this was just unimaginable.

Another kid in my class, a Turkish speaking girl, told me that she was excited, too; not for her she quickly added, of course she could do it in Dutch, but so that she could show her mom her schoolwork. Since her mom hardly spoke any Dutch and read even less of it, she had since a young age not been able to show her schoolwork at home. This was another dynamic I would have never considered: if your kids go to school in a language you do not speak yourself, you are excluded from helping them, and even more so, cannot sincerely appreciate their school work.

As Hedy's localized implementation grew, I realized that one more aspect of language matters, namely varieties of text that English does not have. Many languages for example have gender, and kids might want to use gendered words for assignment, congruent with the gender of a variable. Being forced into one gender with one keyword feels unnatural and removes cultural context. Some languages have different plurals for two, three, or more, and being forced into one keyword (`times` for repetition) feels wrong. In many cases supporting different languages meant supporting multiple alternative keywords, something that is uncommon in English-based languages since it is not needed.

I am forever grateful to all people involved, Ramsey for his pioneering work, Alaaeddin for our months of intense conversation, the kids in my school, the teachers in Botswana, Ethel Tshukudu, and all people reaching out to me over the last 2 years with so many more ideas and perspectives, because this line of work means, in the most literal sense, the world to me. It was a consistent exercise in changing my mind, in listening and learning rather than teaching and doing myself: "*Oh, does Arabic have multiple plurals? I had no idea.*" or "*Does Hindi use different numerals? I did not even know there were numerals other than mine.*"

Regularly community members would be in awe of my absolute lack of knowledge of, well… anything non-Western. The equal footing that I had created with people working on Hedy paid off absolutely in this sub project. Without the atmosphere of me (and other people only used to the Latin alphabet) being open to learning and being wrong, non-English Hedy would have never came to be.

In summary, reasons that I ultimately became convinced that Hedy needed to become a localized programming language are:

- It makes programming easier, not only because the keywords are more understandable, but also because of the ergonomic benefit of being allowed to remain within one keyboard setting.
- It allows people to program in their own cultural context, helping them to envision a world in which their culture lives in the digital world.
- It allows people that do not know English to participate in the broader culture of programming.

Alaaeddin and I ended up writing and talking extensively about this (Swidan and Hermans, 2023). It is interesting, as with Hedy in a larger context also, that a focus on making

something easier, from a cognitive perspective, ended up opening a whole new world of directions which were less directly rooted in cognitive theory, less about decreasing something (cognitive load), and started to be much more about increasing something: the enjoyment, belonging, and community of programming.

Implementing the multi-lingual support for Hedy was quite a technical endeavor that a lot of community members spent considerable time on. When I had announced among people working on Hedy that this was a direction I wanted to go, the first PR just copy-pasted all 13 grammars for the 13 levels we then supported, and swapped out English words for French words.[20] As I alluded to before, initially I just accepted change requests that looked cool without a lot of thoughts given to future maintenance, but by the time this PR came in (in March 2021) I had somewhat learned from my mistakes and decided to not merge the PR in that form (and my fears of killing peoples' excitement were a bit justified, this contributor never came back).

I then enlisted a group of students at Leiden university that built a first version of the grammar system that we still use today, which separates the keywords from the rest of the grammar and merges them to create a full file.[21] As of summer 2024, Hedy supports 57 different languages, including over a dozen non-Latin languages. Almost all of them have been added by volunteer translators: teachers, programmers, students, and general translation enthusiasts (apart from English and Dutch which Marleen and I mostly maintain). With more translators, the Hedy community also grew larger. Firstly, translators often wanted to chat with each other, within and across languages, to decide on the perfect translation for a keyword. Sometimes, some language communities made their own decisions; for example, Spanish uses 'imprimir' (infinitive) for `print`, rather than imperative. In English (and Dutch) those are similar, so one does not have to decide, but in Spanish you have to choose. Other educational programming languages in Spanish have also used forms like 'imprimir' (Zegiestowsky, 2017) or 'escribir.'[22] To align with those, the Spanish translators decided to go with 'imprimir.'

In addition to active communication about translations, translators very often also became local champions of Hedy. After all they did a lot of work translating, so they wanted other people to also use it. And often the translators were well connected in their local communities and knew how to reach teachers and learners we could not, in their native language. Without us planning it, our translators became Hedy ambassadors.

18. Intermezzo: Girls and Programming

In the above, I already mentioned that girls especially like the step-by-step approach that the levels of Hedy offer. Even though I am a relatively rare woman in programming (when I was a student, there was only one other girls in my year with over a hundred students), I never saw Hedy as a system 'for girls,' it was a system to make programming easier for

[20] https://github.com/hedyorg/hedy/pull/340

[21] It turns out that there are a lot more aspects of programming languages that can be localized: numbers, accents, symbols; for all those details, see Swidan and Hermans (2023).

[22] https://www.lenguajelatino.org

everyone. Seeing however how girls responded to the levels and the steps of Hedy made me reflect on their experiences in more depth.

Why would girls, more than boys, appreciate the steps of our programming trajectory, and the hints and tips for what to build? One answer is that girls, through the way that current society operates in the Western world, come into programming classes with a lot less belief in their own programming ability than boys do, something social scientists call *self-efficacy* (Bandura, 1986). Research in fields including programming has shown that high self-efficacy affects course performance (Ramalingam *et al.*, 2004), and that girls typically have lower self-efficacy (Pajares and Johnson, 1996).

As such, girls start in a programming class doubting their abilities, and often need more positive reinforcement. Imagine being already a bit shy about your abilities in programming, having just been told by a classmate or adult that 'programming is super hard and not for girls,' and the first thing you see after your first program is "`syntax error unexpected EOF`" or "`indentation error unexpected indent`," words that you have never seen before, in red. Such a message surely drives the message home that programming is not for girls. That effect of disengagement due to an error (message) is much smaller if your confidence is higher to start with. This is why we put a lot of care in clear error messages: we make extensive use of error productions (Fischer and Mauney, 1980), and aim to minimize parse errors, which are almost always confusing to learners. In addition to error messages, Hedy also supports 'success messages' (a concept that sounds even nicer in my native language of Dutch, as error messages are called 'wrong messages' so we complement them with 'right messages'). These positive messages of course do not work on girls only. We saw a similar effect on kids from families without a computer, who would often have less computer self-efficacy.

Self-efficacy is not just affecting learners' experiences with error messages. Often girls would ask me in those early days: "Have I done it correctly?" to which I would initially say: "Well, do you like the story or game that you have made? If so, you did great!" However, they were unconvinced. Girls wanted to have a clear goal that they could achieve, and having achieved that goal increased their self-efficacy.

I realized that other systems, such as Scratch, did not provide any goals, and this is part of their design philosophy of allowing kids to express themselves. In theory, I think that is an awesome way of teaching, and it has certainly worked for me as a kid. I set my own goals of making games and achieving those goals filled me with joy and pride. However... What if you don't know what you want to build? I saw several kids over the years who were not able to come up with ideas right away. What do you do with those kids? Earlier, while teaching, I had provided elaborate recipes for them to recreate, but that did not create pride in them, they often then expressed that they were just 'following the steps.' After a while, I realized that the kids that I saw as 'not creative' were very often the kids without prior knowledge of programming concepts and without extensive computer and gaming experience. Lacking those, it is very hard to know what you can even do.

My best comparison that works for programmers is to imagine that you have to make a sweater, but you don't know how to knit or sew. Can you come up with creative ideas right away? It is really hard if you have no conception of what is possible or what a beginner can realistically learn in a few weeks.

Hedy's step-by-step approach shows kids what is possible, and helps them to learn and then become creative over time rather than requiring creativity and self-motivation as a prerequisite. Seeing this, and seeing the changes in engagement that came when we added more and more structure, I changed my views on how to learn and teach programming. The most profound change I think was to see with my own eyes that what worked for me (and what worked for a lot of programmers employed in the workforce and in academia today) does not necessarily work for a lot of kids, many of them girls.

It made me think of a phrase that I have often heard in CSed explaining why LOGO (Solomon *et al.*, 2020) never took off in schools, which says that "the problem of Logo is that you can't put Seymour in a box with it" (I have tried tracking down the origin of this quote but I was not able to pinpoint it to a person or written source). My experience with Hedy is actually that yes, you can package a 'teacher' in with a programming system, and we have been doing exactly that to great success. However, and that is maybe what this phrase alludes to, you cannot do it for free.

As successful teachers and educational researchers know, every intervention you do in a classroom that improves the situation for some kids, makes it worse for others; an intervention might benefit the advanced kids at the detriment of kids with less prior knowledge; or the other way around, an intervention might punish or reward risk taking or patience of which some kids have more and others have less.

Adding steps and explanations helps kids that have no idea what programming is, what it is for, and why they should like it, but it might dampen the creativity of other kids. The expressive power of the lower Hedy levels is a lot weaker than in Scratch, which allows for the use of all concepts from the start.

As time had progressed, I have become more and more vocal of making explicit choices with respect to what type of kids you are trying to service with your tools. Hedy is targeting kids without programming experience and programming confidence. If this means that some kids that are already super knowledgeable and excited about programming are less served, then that is something I can live with. Those kids will very likely get there on their own.

One more important point is that the intersection user experience and inclusion is talked about way too less.

When we usually talk about diversity in tech, we talk about women in tech or girls in tech groups, about role models, or about explicit acts of discrimination and oppression, but we don't talk about user experience and programming language design. However, girls and women, because of their lived experiences of discouragement in technology participation,[23] are much more likely to be insecure about their technical abilities, and therefore react differently to error messages. So they might require a different design, and to make matters worse, women are hardly represented in the creation of programming languages and systems, so it is hard to change the status quo from within.

[23] In a recent job interview I, a person holding a PhD in software engineering from a leading engineering school, who led the creation of a programming language and platform, was asked whether I was 'technical enough' for a job in CS academia.

19. Part 12: Tools for Learning ≠ Tools for Teaching

As we come to the end of this paper, there is one dynamic that must be described, and that is my pivot away from seeing Hedy as a tool for individual learners to a tool for classroom teaching. As mentioned above, when I first made Hedy, the image in my mind was that of a kid sitting behind the computer, working alone. It was the way I grew up, and also the way I had been teaching Python initially. It is interesting that this image stuck in my brain so heavily, even after I have been seeing how kids in the second semester I taught before making Hedy were learning so much from *not* being on the computer. I had used 'unplugged' programming before in other experiments (Hermans and Aivaloglou, 2017), and I have concluded myself that prepping kids for programming worked well without the computer.

But in the early days of Hedy, I still thought that getting the kids behind the computer as soon as possible was important, and with this to get them to do actual programming. And it were not just my own memories, I had also seen that programming, even while being so frustrating, did create enthusiasm when it worked. If I could make the struggle smaller, the enthusiasm would be enough to get the students through. I did not explicitly weigh these options, a programming system was just what I gravitated towards. Partly maybe also motivated by a community in which I knew that building something, a language and platform, would be valued a lot more than designing some exercise sheets and PowerPoint slides for teachers.

Over time, I changed my mind on that, and I now believe that programming education should include some, but not too many, open programming exercises. I came to that conclusion, not because of my own teaching, but as I observed other teachers using Hedy.

19.1. *Slides*

Looking in other teachers' classrooms, I noticed that I did a lot of heavy lifting that I did not document within the platform. I would often start without laptops and explain a concept on the blackboard. After explaining the concept, I would often open a Hedy editor on my digital blackboard, create some demo programs, and ask kids to predict what the result would be, or to find the error in a faulty program. After that, I would have discussed with the students suggestions for what they could create, a lot more than were present in the learner facing materials at the time. If kids got stuck I could relatively easily figure out where their issues lay, having made the language. Other teachers did not readily know how to explain concepts, how to test knowledge, and how to get kids to start programming.

I think this might be one of the reasons that teaching programming in schools is hard for teachers in ways that sometimes frustrate programmers: for a teacher, even one with some programming knowledge, the image of 30+ kids on computers programming and then getting stuck in complex ways.[24] Such an image is scary enough for many teachers to consider whether they should even start. Of course, being professionals, many of them

[24] And then we are not even discussing the practicalities of getting all student laptops connected to the internet, and connecting all computers to outlets.

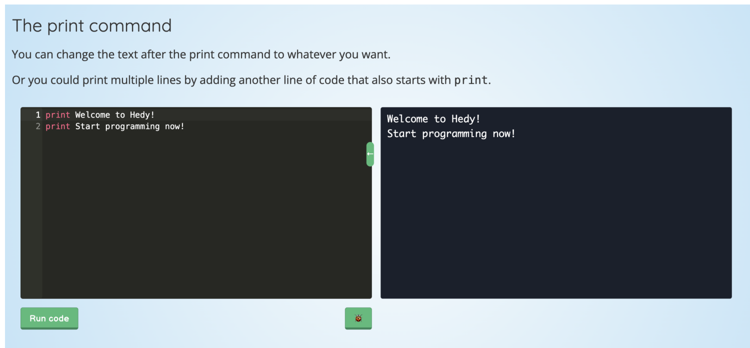

Fig. 6. Slides with an embedded Hedy editor

would figure out how to manage this, but that costs time and energy, and often teachers would stop the lessons after a while, and it prevented some others from really starting. "*Hedy looks cool and very understandable,*" one teacher said, "*but how do I use it in the classroom?*"

I then realized that if I wanted Hedy to really grow large and have an impact on kids, I needed to start seeing teachers, not students, as my target audience. Teachers bring in a whole class, including learners that have no interest in, or opportunity to learning programming at home. A system for teachers needs features that are very different from systems for learners. Obvious features are to be able to present information not as a lesson but as slides to be shown to a whole classroom; which we now support including an editor in which you can run code live, see `www.hedy.org/slides` and Figure 6.

There are features that are more subtle though and that took me longer to consider.

19.2. *Control What Content Viewable to Students*

For example, a teacher that is managing a large classroom might want to prevent the quicker kids from advancing through levels. After all, teaching a class in which some learners work in level 1 and some in level 13 is going to be very challenging. Teachers will have to switch between levels themselves, and will also have to decide whom to support first when stuck.

Teaching becomes a lot easier when you can simply tell kids at the end of level 2: "You are done with today's classwork, you may go read a book or do homework" (as is very common in other courses, too, at least in the culture in which I teach). The reverse by the way also happens, sometimes kids get stuck in lower levels and are afraid to move on, for instance, because of low self-efficacy. Closing lower levels mandates kids to move on even if they are still a bit insecure. A tool that is aimed at supporting self-directed learning (rather than classroom teaching) would not need such brakes or encouragements, hence: tools for teaching are really different from tools for self-directed learning.

19.3. *Creating a Community*

Finally, as we had done with the translators, we wanted to create a community of teachers that could learn from each other. So in our current version, teachers can upload and edit their own lessons, and mix our lessons with theirs or with public lessons from other teachers. In this way we are also allowing teachers to become digital creators, much in line with Hedy's goal of teaching kids to create something meaningful in the digital world.

With our focus shifting from learners to teachers, the community again grew, because teachers, having created lesson materials and interacted with other teachers, felt like they also were part of a community, again helping to foster adoption. Today, 80 % of teachers that create an account on the website do so per recommendation of another teacher.

20. Part 13: What Comes Next?

What is next for Hedy? What are my goals and aspirations still? One answer to that is to hand over the keys to the castle, or at least share them.

As Hedy is moving to a more professional organization, registered a few months ago as an official Dutch non profit, I am no longer making decisions alone. We have a board of 3 people, of which I am just one. And also on the programming side, I am delegating more organizational work to core contributors; a few months ago, at the same time that we founded the non profit, but unrelated, I gave one of our contributors the rights to update production, something that had always been my sole responsibility, which was annoying if there were issues that needed deployment while I was away from the computer.

Things that I want to personally focus on in the next few months or years are twofold. I want to separate out the code that makes Hedy's multi-lingual operation possible, so that it can in theory be used more easily by other programming languages or systems. Even though the multi-lingual nature of Hedy was not my initial idea, I have become convinced that the lower cognitive load it brings and the community that we have created are core to our initial philosophy and deserve an audience wider and broader.

Secondly, I want to focus on the non-coding parts of Hedy, developing more paper materials for kids to use around the platform, which I will try out in the next academic year with a fresh 7th grade in my school. Even if the results will be 'simple paper worksheets,' this will be hard work; partly because I am still trapped in thinking of programming education mainly as a thing a kid does alone on the computer, and partly because there is just not so much research to draw inspiration from, at least not in the programming space. One great example of code reading exercises that can be done away from the computer is the result of a 2019 ITiCSE working group (Izu *et al.*, 2019).

There is unplugged work by Battal *et al.* (2021), but that does not explicitly prepare for computer work. As such it will also be hard to convince teachers and students that working on paper is in fact programming and has educational value, but it has been my experience that this is more the case for teachers with more programming knowledge and experience. For many 7th grade teachers, explaining and demonstrating followed by worksheets is a very common way to teach anything, so I am expecting most uptake there.

If I could wave a magic wand on programming education, in schools or in universities, it would be to convince programming teachers that there is a world of methods to teach with beyond just having students write programs and having them graded by auto graders. There was a small study done in the 90s that established that having learners *read* code, accompanied by explanations from experienced programmers, rather that writing code, is as effective, but it had hardly any impact on teaching practice (Linn and Clancy, 1992).

Working with multiple choice questions, fill the blank exercises, or paper worksheets is so effective, and I hope as a field we will attempt to integrate those into our own teaching, as some work has been doing, such as that by Xie *et al.* (2019) and Goletti *et al.* (2022).

Maybe, if we are lucky, the prevalence of AI tools will be a reason that teachers will be more interested in trying out these techniques; after all, those can still be guaranteed AI-free. Never waste a good crisis as they say.

21. Concluding Remarks

As I expressed in the introduction, writing this paper has forced me to document the unlikely story from a burned-out academic whose career was in the mud, to a successful academic once more, but also a transformation into a programming language creator and open source maintainer. I enjoyed the freedom in writing down my personal history unconstrained by traditional research forms, allowing me to document, and reflect on, the myriad of decisions one has to make when creating a software project of unusual size for an academic, and to share how often I have changed my mind on how to do things, rather than being forced into a single, coherent narrative centered around a research question.

Recently a friend whom I was talking to about Hedy remarked that I was only talking about the multi-lingual aspect, and had that not originally be missing from Hedy? I said: "Yes, I did not think of it, some kids did, and than hundreds of people in the internet made it happen in 57 languages." It is all too easy to portray yourself as a visionary, or be portrayed as such by a community in which programming languages are usually attributed to one or a few creators; see for example, the book *Masterminds of Programming* (Biancuzzi and Warden, 2009). It is also easy to attribute the success of Hedy to the initial innovation behind it, the gradual nature of the levels; but in reality, as this paper has explained, it is much more likely due to the community we created around it, which was largely created due to the localization aspects, and the teacher facing and teacher created content. Sometimes I joke with an early Hedy contributor that we might have created a localized Python with built-in teacher content and that might have been just as big of a hit, and maybe that is true. Maybe the gradual nature only created a fertile ground in which the real success factors were likely to bloom. A smaller language does make lesson plans and translation easier. Reflecting upon what helped and what hindered in this paper has been a valuable experience.

One inspiration for this paper came from a recent book by the German-Korean philosopher Byung-Chul Han, *The crisis of Narration* (Han, 2024), describing how as a society we are losing our ability to reflect and to connect the dots of lives and experiences. While

Han's book is not about academia in particular, it is nonetheless true that we do a lot of research, but not a lot of deep thinking, as also pointed out by a recent editorial in Nature (Nature, 2024). Narratives about a certain research program, as commonly told in keynotes and sometimes in grant proposals, would make for excellent vehicles for that type of deep thinking.

Outside of HOPL,[25] which is only published every decade or so, these histories are rare, I only know of Guo's work on *PythonTutor* (Guo, 2021) and Komm's reflections on *TigerJython* and *WebTigerPython* (Komm, 2024). I hope this paper, in addition to inspiring people to reflect on their own decision making, will inspire academics to write similar histories, and encourage journals and editors to solicit narratives more explicitly.

References

Altadmri, A., Brown, N. (2015). 37 Million Compilations: Investigating Novice Programming Mistakes in Large-Scale Student Data. In: *Proceedings of the 46th ACM Technical Symposium on Computer Science Education (SIGCSE 2015)*. Association for Computing Machinery, New York, NY, USA, pp. 522–527. https://doi.org/10.1145/2676723.2677258.

Bandura, A. (1986). *Social Foundations of Thought and Action*. Prentice-Hall Inc.

Battal, A., Adanır, G.A., Gülbahar, Y. (2021). Computer Science Unplugged: A Systematic Literature Review. *Journal of Educational Technology Systems*, 50(1), 24–47. https://doi.org/10.1177/00472395211018801.

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

Biancuzzi, F., Warden, S. (2009). *Masterminds of Programming: Conversations with the Creators of Major Programming Languages*. Theory in Practice (O'Reilly).

Brown, N., Cai, Y., Guo, Y., Kazman, R., Kim, M., Kruchten, P., Lim, E., MacCormack, A., Nord, R., Ozkaya, I., Sangwan, R., Seaman, C., Sullivan, K., Zazworka, N. (2010). Managing Technical Debt in Software-Reliant Systems. In: *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research (FoSER '10)*. Association for Computing Machinery, New York, NY, USA, pp. 47–52. 978-1-4503-0427-6. https://doi.org/10.1145/1882362.1882373.

Byrne, D. (2013). *How Music Works*. Canongate Books.

Eghbal, N. (2020). *Working in Public: The Making and Maintenance of Open Source Software*. Stripe Press, San Francisco. 978-0-578-67586-2.

Findler, R. (1996). DrRacket: The Racket Programming Environment. https://plt.cs.northwestern.edu/snapshots/current/pdf-doc/drracket.pdf.

Fischer, C.N., Mauney, J. (1980). On the Role Of Error Productions in Syntactic Error Correction. *Computer Languages*, 5(3), 131–139. https://doi.org/10.1016/0096-0551(80)90006-5.

Goletti, O., Mens, K., Hermans, F. (2022). An Analysis of Tutors' Adoption of Explicit Instructional Strategies in an Introductory Programming Course. In: *Proceedings of the 22nd Koli Calling International Conference on Computing Education Research (Koli Calling 2022)*. Association for Computing Machinery, New York, NY, USA, pp. 1–12. 978-1-4503-9616-5. https://doi.org/10.1145/3564721.3565951.

Guo, P. (2012). *The Ph.D. Grind: A Ph.D. Student Memoir*.

Guo, P. (2021). Ten Million Users and Ten Years Later: Python Tutor's Design Guidelines for Building Scalable and Sustainable Research Software in Academia. In: *Proceedings of the 34th Annual ACM Symposium on User Interface Software and Technology (UIST 2021)*. Association for Computing Machinery, New York, NY, USA, pp. 1235–1251. 978-1-4503-8635-7. https://doi.org/10.1145/3472749.3474819.

Han, B.-C. (2024). *The Crisis of Narration*. 978-1-5095-6043-1.

Hermans, F. (2013). *Analyzing and Visualizing Spreadsheets*. Phd thesis, Delft University of Technology.

Hermans, F. (2020). Hedy: A Gradual Language for Programming Education. In: *Proceedings of the 2020 ACM Conference on International Computing Education Research (ICER 2020)*. Association for Computing Machinery, New York, NY, USA, pp. 259–270. 978-1-4503-7092-9. https://doi.org/10.1145/3372782.3406262.

Hermans, F. (2021). *The Programmer's Brain: What Every Programmer Needs to Know About Cognition*. Manning Publications, S.l. 978-1-61729-867-7.

Hermans, F., Aivaloglou, E. (2017). To Scratch or not to Scratch? A Controlled Experiment Comparing Plugged First and Unplugged First Programming Lessons. In: *Proceedings of the 12th Workshop on Primary and Secondary Computing Education (WiPSCE 2017)*. Association for Computing Machinery, New York, NY, USA, pp. 49–56.

Hermans, F., Schlesinger, A. (2024). A Case for Feminism in Programming Language Design. In: *Proceedings of the 2024 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2024)*. Association for Computing Machinery, New York, NY, USA. To appear.

Hermans, F., Aivaloglou, E., Jansen, B. (2015a). Detecting Problematic Lookup Functions in Spreadsheets. In: *2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE Computer Society, pp. 153–157.

[25] https://dl.acm.org/toc/pacmpl/2020/4/HOPL

Hermans, F., Pinzger, M., Deursen, A.v. (2012a). Detecting and Visualizing Inter-Worksheet Smells in Spreadsheets. In: *Proceedings of the 34th International Conference on Software Engineering (ICSE 2012)*. IEEE Press, pp. 441–451.

Hermans, F., Pinzger, M., Deursen, A.v. (2012b). Detecting Code Smells in Spreadsheet Formulas. In: *Proceedings of the International Conference on Software Maintenance (ICSM 2012)*. IEEE Computer Society.

Hermans, F., Pinzger, M., van Deursen, A. (2015b). Detecting and Refactoring Code Smells in Spreadsheet Formulas. *Empirical Software Engineering*, 20(2), 549–575.

Izu, C., Schulte, C., Aggarwal, A., Cutts, Q.I., Duran, R., Gutica, M., Heinemann, B., Kraemer, E.T., Lonati, V., Mirolo, C., Weeda, R. (2019). Fostering Program Comprehension in Novice Programmers – Learning Activities and Learning Trajectories. In: Scharlau, B., McDermott, R., Pears, A., Sabin, M. (Eds.), *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education (ITiCSE-WGR 2019)*. Association for Computing Machinery, pp. 27–52. https://doi.org/10.1145/3344429.3372501.

Kirschner, P., Sweller, J., Clark, R. (2006). Why Minimal Guidance During Instruction Does Not Work: An Analysis of the Failure of Constructivist, Discovery, Problem-Based, Experiential, and Inquiry-Based Teaching. *Educational Psychologist*, 41. https://doi.org/10.1207/s15326985ep4102_1.

Komm, D. (2024). Programming Education Made in Switzerland – The Story of WebTigerPython. SVIA / SSIE / SSII Interface. https://svia-ssie-ssii.ch/interface/programming-education-made-in-switzerland-the-story-of-webtigerpython/.

Linn, M.C., Clancy, M.J. (1992). The Case For Case Studies of Programming Problems. *Communications of the ACM*, 35(3), 121–132. https://doi.org/10.1145/131295.131301.

Mohamed, A., Zhang, L., Jiang, J., Ktob, A. (2018). Predicting Which Pull Requests Will Get Reopened in GitHub. In: *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*, pp. 375–385. ISSN: 2640-0715. https://doi.org/10.1109/APSEC.2018.00052.

Nature (2024). Scientists Need More Time to Think. *Nature*, 631, 709–709.

Ortu, M., Marchesi, M., Tonelli, R. (2019). Empirical Analysis of Affect of Merged Issues on GitHub. In: *Proceedings of the 4th IEEE/ACM 4th International Workshop on Emotion Awareness in Software Engineering (SEmotion)*, pp. 46–48. https://doi.org/10.1109/SEmotion.2019.00017.

Pajares, F., Johnson, M.J. (1996). Self-Efficacy Beliefs And the Writing Performance of Entering High School Students. *Psychology in the Schools*, 33(2), 163–175. https://doi.org/10.1002/(SICI)1520-6807(199604)33:2<163::AID-PITS10>3.0.CO;2-C.

Prat, C.S., Madhyastha, T.M., Mottarella, M.J., Kuo, C.-H. (2020). Relating Natural Language Aptitude to Individual Differences in Learning Programming Languages. *Scientific Reports*, 10(1), 3817. https://doi.org/10.1038/s41598-020-60661-8.

Ramalingam, V., LaBelle, D., Wiedenbeck, S. (2004). Self-Efficacy and Mental Models in Learning to Program. In: *Proceedings of the 9th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE 2004)*. Association for Computing Machinery, New York, NY, USA, pp. 171–175. 978-1-58113-836-8. https://doi.org/10.1145/1007996.1008042.

Salac, J., Franklin, D. (2020). If They Build It, Will They Understand It? Exploring the Relationship Between Student Code and Performance. In: *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE 2020)*. Association for Computing Machinery, New York, NY, USA, pp. 473–479. event-place: Trondheim, Norway. 978-1-4503-6874-2. https://doi.org/10.1145/3341525.3387379.

Solomon, C., Harvey, B., Kahn, K., Lieberman, H., Miller, M.L., Minsky, M., Papert, A., Silverman, B. (2020). History of Logo. *Proceedings of the ACM on Programming Languages*, 4(HOPL), 79–17966. https://doi.org/10.1145/3386329.

Sweller, J. (1988). Cognitive Load During Problem Solving: Effects on Learning. *Cognitive Science*, 12(2), 257–285. https://doi.org/10.1016/0364-0213(88)90023-7.

Sweller, J. (1994). Cognitive Load Theory, Learning Difficulty, and Instructional Design. *Learning and Instruction*, 4(4), 295–312. https://doi.org/10.1016/0959-4752(94)90003-5.

Swidan, A., Hermans, F. (2023). A Framework for the Localization of Programming Languages. In: *Proceedings of the 2023 ACM SIGPLAN International Symposium on SPLASH-E (SPLASH-E 2023)*. Association for Computing Machinery, New York, NY, USA, pp. 13–25. https://doi.org/10.1145/3622780.3623645.

Swidan, A., Serebrenik, A., Hermans, F. (2017). How do Scratch Programmers Name Variables and Procedures? In: *2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE Computer Society, pp. 51–60.

Tshukudu, E., Dodoo, E.R., Hermans, F., Mudongo, M. (2024). Bilingual Programming: A Study of Student Attitudes and Experiences in the African Contex. In: *Proceedings of the 24th Koli Calling International Conference on Computing Education Research (Koli Calling 2024)*. To appear.

Xie, B., Loksa, D., Nelson, G.L., Davidson, M.J., Dong, D., Kwik, H., Tan, A.H., Hwa, L., Li, M., Ko, A.J. (2019). A Theory of Instruction for Introductory Programming Skills. *Computer Science Education*, 29(2-3), 205–253. https://doi.org/10.1080/08993408.2019.1565235.

Zegiestowsky, A. (2017). Tango: A Spanish-Based Programming Language. *Butler Journal of Undergraduate Research*, *3*(1).