Issue 075: Teaching Programming

De Programmatica Ipsum

2024-12-02

Contents

Issue 075: Teaching Programming 1

Banning, Adopting, Reckoning 3

Jerry Cain 11

A Review Of Research Around Programming Education From The 1960s To Today 15

Issue 075: Teaching Programming

By Adrian Kosmaczewski, December 2nd, 2024

Welcome to the 75th issue of *De Programmatica Ipsum*, about *Teaching Programming*.

In this edition:

- We argue that we should embrace LLMs in the classroom[1], but only if we are aware of their shortcomings.
- In the Library section[2], we review decades of research[3] trying to answer the ques-

[1]https://deprogrammaticaipsum.com/banning-adopting-reckoning/
[2]https://deprogrammaticaipsum.com/category/library/
[3]https://deprogrammaticaipsum.com/a-review-of-research-around-programming-education-

tion: how can we best teach programming to younger generations?
- In our Vidéothèque section[4], we watch “Programming Paradigms” by Professor Jerry Cain[5] of Stanford University.

We would like to thank our patrons who generously contribute every month (or have contributed in the past) to our work and help us run this magazine. Thank you so much! In alphabetical order: Adam Guest, Adrian Tineo Cabello, Benjamin Sheldon, Christopher Nascone, Colin Powell, Franz Lucien Moersdorf, Guillermo Ramos Álvarez, Jean-Paul de Vooght, Dr. Juande Santander-Vela, Patryk Matuszewski, Paul Hudson, Quico Moya, Roger Turner, Szymon Licau, and countless more leaving anonymous tips every month.

Enjoy this issue! Please subscribe to our free newsletter[6] to stay updated about new releases, share the articles on social media, or contribute[7] if you would like to support our work with a donation via Liberapay[8].

Cover photo by Ivan Aleksic[9] on Unsplash[10].

from-the-1960s-to-today/

[4]https://deprogrammaticaipsum.com/category/videotheque/

[5]https://deprogrammaticaipsum.com/jerry-cain/

[6]https://deprogrammaticaipsum.com/newsletter/

[7]https://deprogrammaticaipsum.com/contribute/

[8]https://liberapay.com/

[9]https://unsplash.com/@ivalex?utm_content=creditCopyText&utm_medium=referral&utm_source=unsplash

[10]https://unsplash.com/photos/brown-wooden-table-and-chairs-PDRFeeDniCk?utm_content=creditCopyText&utm_medium=referral&utm_source=unsplash

Banning, Adopting, Reckoning

By Adrian Kosmaczewski, December 2nd, 2024

Last month, OpenAI, the company (in)famous for their ChatGPT product, released a course called "ChatGPT Foundations for K-12 Educators"[11], an event that has raised[12] more than a few eyebrows, and even some outrage[13]. We must have a serious conversa-

[11]https://www.commonsense.org/education/training/chatgpt-k12-foundations

[12]https://techcrunch.com/2024/11/20/openai-releases-a-teachers-guide-to-chatgpt-but-some-educators-are-skeptical/

[13]https://buttondown.com/maiht3k/archive/chatgpt-has-no-place-in-the-classroom/

tion about the value of a bullshit generator[14] in the context of teaching programming skills to new generations.

It will most certainly *not* come as a surprise to our regular readers the revelation that this magazine tends to have a strong tendency towards Luddism[15]. Yes, even when it comes to one of the most recent inventions made by man, the computer, we will continue to put our foot on the brake and call for reflection and evaluation, venture capital be damned. Of course, this does *not* happen, and we enter again and again cycles of hype[16] that do more harm than good.

These days, generative artificial intelligence is all the rage. All of a sudden, and seemingly out of nowhere, a statistical bullshit machine, very far-fetched from the artificial intelligence I dedicated an open letter[17] to a few years ago, is helping students craft 2000-words essays in seconds, and the same bullshit machine is used by overloaded and burnout teachers to summarize (and maybe even grade!) those same essays in an even shorter amount of time.

Oh brave new world, that has such people in it!

Can we ignore ChatGPT (or other LLMs for that matter) when teaching programming in the year 2024? Hardly. It is there, it is free to use, and it can be damn good at certain tasks. Damn good in the short term, unfortunately. In the long run, it might have a very detrimental effect, unless we adopt some measures right now.

Electronic Calculators

I belong to the generation of human beings who saw the sudden appearance of the electronic calculator in our pockets. During the 1980s, their price dropped as fast as their capacities grew; from being able to perform a square root calculation in seconds in 1980, we ended the decade in 1989 with programmable calculators featuring hyperbolic trigonometric functions and a myriad of other capabilities.

What was the reaction of math[18] teachers worldwide to this sudden invasion? First, as expected, they reacted with a *ban*. We could not use calculators of any kind, not in the classroom, and especially not (God forbid!) during tests. They were prohibited,

[14] https://link.springer.com/article/10.1007/s10676-024-09775-5
[15] https://en.wikipedia.org/wiki/Luddite
[16] https://deprogrammaticaipsum.com/mainstream-is-the-new-hype/
[17] https://deprogrammaticaipsum.com/open-letter-to-a-future-ai/
[18] https://deprogrammaticaipsum.com/in-praise-of-mathematics/

completely, absolutely, and entirely. We had to learn how to calculate square roots by hand (seriously, it is not *that* difficult) and to learn the algorithms of plenty of other integer and even real number operations.

Instead of calculators, we could sit at exams with certain books on our desks, filled with logarithm and trigonometry tables, featuring formulae for algebra, calculus, and geometry. French-speaking Swiss students might remember the ubiquitous "Formulaire et Tables"[19] of the "Commissions romandes de mathématiques, de physique et de chimie", published by the Éditions du Tricorne, that many generations of Swiss students used during their exams.

The second step was a slow *adoption* process for calculators in the classroom: we could use them, in particular to solve exercises outside of exams, but we could *not* use the programming kind thereof (which were quite expensive, anyway), and very often not at all during exams. Get yourself a cheap Casio or Texas Instrument calculator, like the ones featured next to a Walmart cash register, thank you so much. Again, no use of those during exams; but hey! We could do our exercises much faster, which meant we could do *more* exercises (and more complex ones at that) in the same amount of time.

Finally, the *reckoning* phase came, after all. By the 1990s I could use my all-powerful Hewlett-Packard 48GX[20] with 128 KB of RAM, with the RPL[21] programming language (inspired by Forth), featuring 3D graphics, matrix calculations, and even an equation editor, at any time, in any situation: classrooms, exercises, exams, you name it.

(Disclaimer: yes, the HP 48GX had a short-range infrared port that allowed devices to exchange data via Kermit[22]. Yes, we used that during exams to exchange information with fellow HP 48GX users. No, that was not really legal. I will not make any further comments.)

All of these possibilities came with a caveat: the problems to be solved during exercise and exam sessions were immensely more complex than those you would find on a term exam in 1981. Kermit or not, I failed too many of those exams. Turns out one needed much more than raw computing capability to get a good grade. Surprise!

And this is key to this story; we will have to learn how to deal with LLMs in the classroom in one way or another, just like we had to learn how to cope with calculators forty

[19]https://www.orellfuessli.ch/shop/home/artikeldetails/A1017488126
[20]https://en.wikipedia.org/wiki/HP_48_series
[21]https://en.wikipedia.org/wiki/RPL_(programming_language)
[22]https://en.wikipedia.org/wiki/Kermit_(protocol)

years ago. The genie is out of the bottle, and yes, it is a drunk genie that hallucinates quite a bit sometimes, unfortunately. But this is what we got today.

As I write these words, school systems everywhere are reacting to the sudden existence of this clumsy genie by either banning it (he? she? they?) or, in the best cases, by slowly adopting it as an integral part of the world those kids will inherit.

Should we then follow the ideas of OpenAI and attend the dreaded online course mentioned at the beginning of this article? Yes, but not blindly; be aware of who the creator of this course is, and what their agenda looks like.

As always, *caveat alumni*.

Large Bullshit Models

What is a teacher to do when they ask their students to program a system in some programming language, and a week later they submit some ChatGPT-generated nonsense that they cannot even understand, let alone run properly?

We have to come to the conclusion that the "Hello World[23] Era" is over. We cannot, and we must not teach computer programming ever again as we have in the past. The existence of LLMs must mark the debut of a new style of programming education, whether we like it or not. Sorry for all those educators who were looking forward to retiring while applying the same good old recipe semester after semester!

The same way we have left behind BASIC[24], Logo[25], Pascal[26], and to a certain degree, even Scratch[27]; the same way we have left behind the dubious practice of making students write code by hand on a piece of paper (a sad idea this author has witnessed first hand thirty years ago); we will have to leave behind the usual coding tests, the standard programming workshops, and we will have to accept the existence of LLMs in our world, whether we like them or not.

Embrace the hallucination, and teach your students to digest it, to work with it, to learn from it.

[23] https://deprogrammaticaipsum.com/memories-of-hello-world/
[24] https://deprogrammaticaipsum.com/dartmouth-college/
[25] https://en.wikipedia.org/wiki/Logo_(programming_language)
[26] https://deprogrammaticaipsum.com/niklaus-wirth/
[27] https://en.wikipedia.org/wiki/Scratch_(programming_language)

In terms of reckoning, this means effectively using LLMs in the classroom; ideally not ChatGPT or similar ones, owned by large corporate conglomerates, but smaller ones, hopefully trained *ad hoc* for teaching purposes, fed with open access or public domain material. We are almost at the dawn of 2025 and thankfully, there are a few open-source models to choose from that fulfill these requirements, and maybe a business or two will provide such LLMs in the near future, if they are not already doing that.

(The astute reader of the last paragraph will most probably have realized that I just gave away, for free, an idea for the next multi-billion AI-powered startup. Do not thank me, just send a few stock options my way, thank you so much.)

Once the bullshit machines generate their code, the role of the educator is then to drive the discussion around those inevitable big balls of mud[28]. Drive the reasoning process towards higher grounds: architecture, collaboration, large systems, maintenance, documentation, testing, quality. Use that code and make your students build large, very large systems even, with potentially hundreds of lines of code, and make them (the students and the systems) collaborate with one another. Take the output from those LLMs and then make students fix it (potentially with the help of an LLM!), document it, wire it, test it, maintain it.

This is something that the very creator of OOP had in mind:

> Contrary to general practice Kristen Nygaard argued that the teaching of object orientation should begin with sufficiently complex examples in order to expose the strength of analysing and describing complex situations in an object-oriented perspective.

(Jens Bennedsen and Michael E Caspersen, "Teaching Object-Oriented Programming", 2004.)

Teachers should drive their students to generate and analyze code in various programming languages. They should help them become polyglot software engineers, not just myopic single-language or single-paradigm zealots, able to deal with (at least) the top 20 languages of rankings such as TIOBE[29], RedMonk[30], and PYPL[31]. Teachers should drive their minds to become fluent in both object-oriented[32] and functional programming languages; in both academic and enterprise languages alike.

[28] https://deprogrammaticaipsum.com/goodness-gracious-great-balls-of-mud/
[29] https://www.tiobe.com/tiobe-index/
[30] https://redmonk.com/sogrady/2024/03/08/language-rankings-1-24/
[31] http://pypl.github.io/PYPL.html
[32] https://deprogrammaticaipsum.com/the-hype-cycle-of-oop/

Empower your students so that they can make the impossible dialogue[33] possible. We need this, more than ever.

Finally, make students sit for their final term programming exams in air-gapped[34] environments without network access to the broader internet. Provide them with virtual machines or containerized environments, with all the information they need (including man pages, HTML websites, books in PDF and EPUB formats, etc.) and all the tools they need (compilers, runtimes, etc.) and make them create systems from scratch. Evaluate those exams using automated testing (and, please, not with another LLM!) so that they can get immediate feedback of their final grade.

Your students should program stuff during those exams in an isolated environment that contains the minimum required to do their work. In terms of technology choices, small LLMs like the Granite[35] family of large language models, web-based development environments such as Eclipse Che[36], or even (if all else fails) VirtualBox[37] virtual machines could certainly be used to bring such experience to life.

(Disclaimer: at the time of this writing the author of this article works for Red Hat, an IBM company, and yes, this employer sells some of the items enumerated above.)

We are firm believers in what Seymour Papert[38] once said:

> Being a mathematician is no more definable as "knowing" a set of mathematical facts than being a poet is definable as knowing a set of linguistic facts. Some modern math ed reformers will give this statement a too easy assent with the comment: "Yes, they must understand, not merely know." But this misses the capital point that being a mathematician, again like being a poet, or a composer or an engineer, means doing, rather than knowing or understanding.

(Seymour Papert, "Teaching Children to Be Mathematicians Versus Teaching About Mathematics", July 1971. Emphasis in the original.)

And in the age of the LLM, we believe that the "doing" part has to change substantially… once again.

[33] https://deprogrammaticaipsum.com/the-impossible-dialogue/
[34] https://en.wikipedia.org/wiki/Air_gap_(networking)
[35] https://www.ibm.com/granite
[36] https://en.wikipedia.org/wiki/Eclipse_Che
[37] https://en.wikipedia.org/wiki/VirtualBox
[38] https://en.wikipedia.org/wiki/Seymour_Papert

Conclusion

As the big caveat of this article, I feel I must quote a recent paper highlighting many of the risks brought by LLMs in society, including those related to the educational context:

> Educational uses of generative AI pose several other challenges. One is the perpetuation of biases and discrimination, potentially reinforcing racial or gender-based stereotypes during personalized learning, automated scoring, and admission processes.
> (...)
> The current debate about the role of generative AI, from primary schools to universities, revolves around whether generative AI should be banned, permitted under only some cases, or allowed as assistance for teachers and students. (...)
> We argue that these approaches are limited in vision. A more forward-thinking approach would involve a curricular revolution to redefine the skills and competencies necessary to effectively utilize generative AI.

(Valerio Capraro, Austin Lentsch, Daron Acemoglu, et al. "The Impact of Generative Artificial Intelligence on Socioeconomic Inequalities and Policy Making," 2024.)

Assuming LLMs are to be accepted in education, what should be the guiding principle? First, teach your students to think like humans. Even if it hurts (spoiler alert: it does). Second, rise the abstraction ladder, using the LLM as a tool to generate the minutiae of the larger systems at hand.

If anything, this author firmly believes that programming skills are second to those related to communication; most engineers coming out of colleges these days are unable to express themselves in public, to teach[39] their peers, to write[40] an essay or a blog post, to communicate their ideas to stakeholders, or to put together a simple documentation bundle without suffering a seizure in the process.

If we are going to have LLMs performing the grunt work of coding, we need students to become the architects of tomorrow, not just another coder selling their work at Fiverr for a living. Help those students build larger and more complex systems, while at the same time making them conscious of the process, and helping them develop those much required "soft skills".

[39] https://deprogrammaticaipsum.com/less-evangelization-more-honestization/

[40] https://deprogrammaticaipsum.com/on-the-aversion-to-writing/

At the end of our article about BASIC[41], a programming language that was quite literally created to teach students how to program, we lamented the current state of programming education with these words, which we reproduce here for the sake of memory and repetition:

> We used to teach kids how to think like a computer. These days, however, we are more interested in teaching computers how to think like kids, and we are doing a terrible job in both cases.

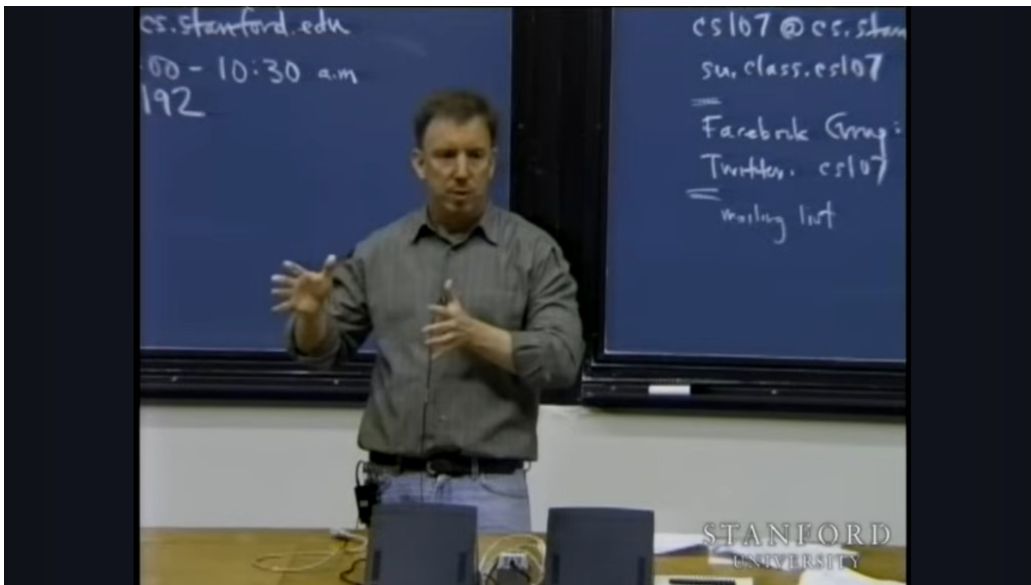
Cover photo by Aaron Lefler[42] on Unsplash[43].

[41] https://deprogrammaticaipsum.com/programming-the-liberal-arts/

[42] https://unsplash.com/@alefler?utm_content=creditCopyText&utm_medium=referral&utm_source=unsplash

[43] https://unsplash.com/photos/a-calculator-sitting-on-top-of-a-piece-of-paper-ySZdYkPGEbs?utm_content=creditCopyText&utm_medium=referral&utm_source=unsplash

Jerry Cain



By Adrian Kosmaczewski, December 2nd, 2024

The late 2000s were an interesting time for online education. The wider availability of faster and more reliable bandwidth led to an explosion of online video. This, in turn, led to the emergence of an ever-expanding number of providers of online learning services, and then to a wave of "Massive Open Online Courses"[44] or MOOCs, many of which were offered by large universities and high schools all over the world. This month's Vidéothèque movie is a full playlist featuring one of the earliest (and, in the opinion of this author, one of the most useful) examples of an online programming course.

It was at the end of the 2000s when services like the following broke into the main-

[44] https://en.wikipedia.org/wiki/Massive_open_online_course

stream, offering unprecedented access to online education resources that were until that moment restricted to a few lucky (and wealthy) attendees: Coursera[45], Udemy[46], edX[47], Udacity[48], MasterClass[49], Pluralsight[50], Platzi[51], Khan Academy[52], Brilliant[53], LinkedIn Learning[54] (originally known as Lynda.com), StackSkills[55], DeepLearning.AI[56], Frontend Masters[57], Project Euler[58], and so many more that it would be impossible to list them all.

But the Internet is an inherently democratizing medium. Many smaller institutions thus chose to deploy a Moodle[59] instance and host their educational content themselves. To avoid the burden of maintenance required by a self-hosting solution, some allocated budgets and decided to hire some third-party providers to serve their content, ranging from bigger ones like Instruqt[60], to smaller ones like nu.education[61].

Most importantly, many individuals took a cue from the bigger names and expanded their online presence with some dedicated training resources. Some of these are Gary Bernhardt[62] and his Execute Program[63], Dylan Beattie[64] and his Ursatile[65], Amy Hoy's 30x500 Academy[66], Jessica Brody's Writing Mastery Academy[67], Jason

---

[45]https://www.coursera.org/
[46]https://www.udemy.com/
[47]https://www.edx.org/
[48]https://www.udacity.com/
[49]https://www.masterclass.com/
[50]https://www.pluralsight.com/
[51]https://platzi.com/
[52]https://www.khanacademy.org/
[53]https://brilliant.org/
[54]https://www.linkedin.com/learning/
[55]https://stackskills.com/
[56]https://www.deeplearning.ai/
[57]https://frontendmasters.com/
[58]https://projecteuler.net/
[59]https://moodle.org/
[60]https://instruqt.com/
[61]https://nu.education/
[62]https://deprogrammaticaipsum.com/gary-bernhardt/
[63]https://www.executeprogram.com/
[64]https://deprogrammaticaipsum.com/dylan-beattie/
[65]https://ursatile.com/
[66]https://30x500.com/academy/
[67]https://www.writingmastery.com/

Calacanis' Founder University[68], Scott Galloway's Section School[69], Anna Bey's School of Affluence[70], or Rachel Thomas and Jeremy Howard's fast.ai[71] service.

For self-taught programmers, like the one writing the lines you are reading at this moment, there is no shortage of interesting resources online. Suffice to mention MIT's recent Missing Semester of CS Education[72]; also from MIT, the now legendary 6.001 Structure and Interpretation[73] featuring video lectures by Hal Abelson and Gerald Jay Sussman given in July 1986 for Hewlett-Packard employees; and last but not least, the Spring 2015 Computer Architecture Lectures[74], taught by ETHZ Professor Onur Mutlu[75] in Carnegie Mellon University.

This month's Vidéothèque entry is a full playlist titled Programming Paradigms[76] published on YouTube in July 2008 by the Stanford Center for Professional Development of Stanford University, and taught by Professor Jerry Cain.

In this course, consisting of 27 lectures, Professor Cain dives into a magnificent multi-paradigm, multi-programming-language course, explaining various concepts of computer science on a medium as analog and mechanic as chalk and blackboard can be.

Maybe one of the greatest advantages of this course is that Professor Cain jumps immediately into one of the most puzzling aspects of programming for people new to the subject: the correspondence between lines of code and the internal representation of data in memory. Noteworthy are the seemingly endless sequences of "byte patterns" that represent various types of data in memory, sometimes featuring entire regions filled with a large zero: integers, IEEE 754 floating-point numbers[77], arrays, null-terminated strings, etc.

Professor Cain uses various programming languages to explain these concepts: he starts with C (roughly from the second lecture to the 8th), then Assembly (lectures 9 to 11), C++ (lectures 12 to 18), Scheme (lectures 19 to 23), and Python[78] (from 24 to 26).

[68] https://www.founder.university/
[69] https://www.sectionschool.com/
[70] https://annabey.com/about-school-of-affluence/
[71] https://www.fast.ai/
[72] https://www.youtube.com/playlist?list=PLyzOVJj3bHQuloKGG59rS43e29ro7I57J
[73] https://www.youtube.com/playlist?list=PLE18841CABEA24090
[74] https://www.youtube.com/playlist?list=PL5PHm2jkkXmi5CxxI7b3JCL1TWybTDtKq
[75] https://people.inf.ethz.ch/omutlu/
[76] https://www.youtube.com/playlist?list=PL9D558D49CA734A02
[77] https://deprogrammaticaipsum.com/the-smartest-concept-in-computer-science/
[78] https://deprogrammaticaipsum.com/the-state-of-python-in-2021/

The last lecture (27) dives into some other functional programming languages like ML, Miranda, and even Haskell, as well as some advanced type design concepts, to round up your general programming knowledge.

Let us remind ourselves of what Steve McConnell[79] said about such a multi-language approach, a quote we shared in a previous article[80] of this magazine:

> The examples are in multiple languages because mastering more than one language is often a watershed in the career of a professional programmer.

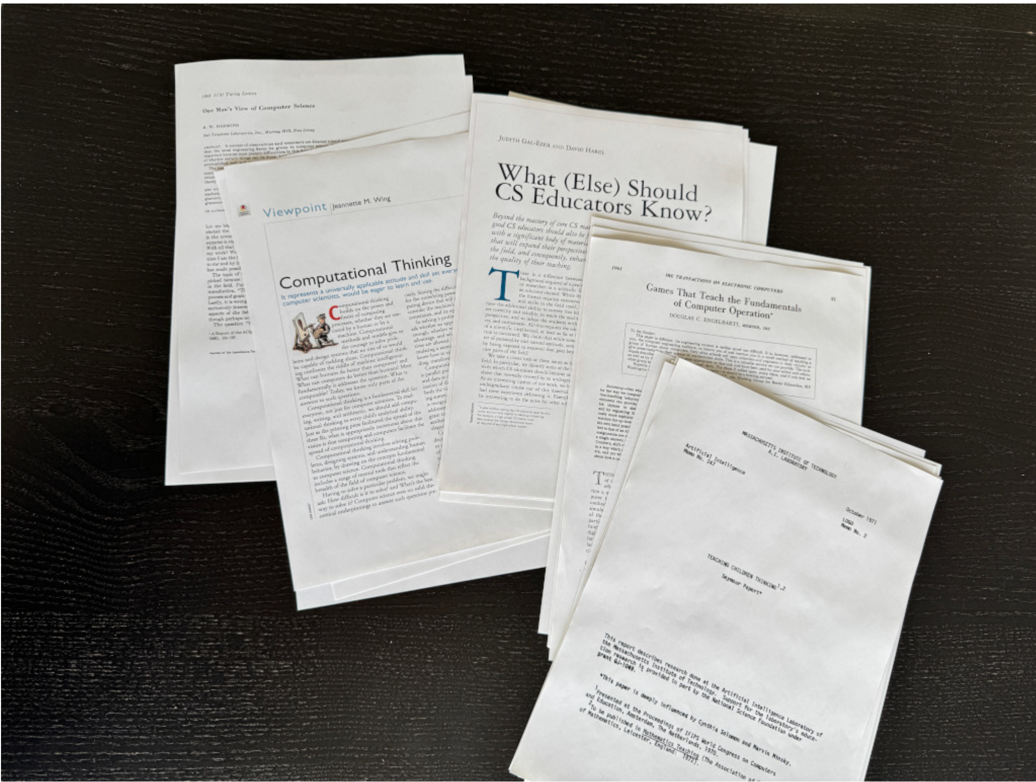
I remember watching Professor Cain's course as I was giving the final touches to my Master's Degree thesis, and being completely in awe. I have recommended this series, again and again to quite a few generations of software engineers who asked me for guidance at the start of their careers. And now I am doing it again, albeit in the pages of this magazine, as a tribute to a fantastic resource that should be more widely known.

Watch this month's Vidéothèque playlist, Programming Paradigms[81], by Professor Jerry Cain of Stanford University, on YouTube, and I am sure this will make sense to people. (Watch the series, and you will get this last joke. I promise.)

Cover photo chosen by the author from the first lecture[82] of the series.

[79] https://deprogrammaticaipsum.com/steve-mcconnell/
[80] https://deprogrammaticaipsum.com/how-to-choose-a-programming-language-for-your-book/
[81] https://www.youtube.com/playlist?list=PL9D558D49CA734A02
[82] https://www.youtube.com/watch?v=Ps8jOj7diA0&list=PL9D558D49CA734A02&index=1

A Review Of Research Around Programming Education From The 1960s To Today



By Adrian Kosmaczewski, December 2nd, 2024

The problem of teaching programming skills to new generations of software engineers is as old as the computers themselves. Each generation has tried to do it in a slightly different way, with various degrees of success. There is a lot of literature available online about the subject, and in this article we will point out papers and books that we found to be the most noteworthy. By no means this is an exhaustive list, but it features some interesting entries that might serve as a starting point for your own research.

The Beginnings

Teaching people about computers has never been as complicated as it was in the early 1960s. To begin with, computers were relatively hard to come by according to our standards (and this is an understatement). However, many scientists (rightly) understood that the computer was a breakthrough machine in the history of mankind, and that knowledge about this invention had to be spread as much as possible.

Douglas Engelbart[83], of "Mother of All Demos"[84] fame, came up with an interesting gamified approach to teach the inner workings of binary computers to "laymen".

> The novel feature of the teaching method is that it makes use of human participants to simulate the function of logical elements that are typical of those used in digital computers. A group of such participants can be "wired" into a network that will function in a manner very similar to that of an actual digital network.

(Douglas C. Engelbart, "Games That Teach the Fundamentals of Computer Operation", IRE Transactions on Electronic Computers EC-10, no. 1 (March 1961): 31–41. https://doi.org/10.1109/TEC.1961.5219149.)

Of course this was not precisely teaching programming, but merely teaching about what computers are and how they work internally. You gotta start somewhere.

We have already remarked in this magazine[85] that 1968 was an *annus mirabilis* for computer science, similarly to how 1905 was one for Physics. It was the year of Engelbart's "Mother of All Demos", the year of Edsger Dijkstra's "Go To Statement Considered Harmful" article, the year of the NATO Software Engineering Conference and its definition of "software crisis", and the year of ALGOL W starting to become Pascal[86] inside

[83] https://en.wikipedia.org/wiki/Douglas_Engelbart
[84] https://en.wikipedia.org/wiki/The_Mother_of_All_Demos
[85] https://deprogrammaticaipsum.com/edward-nash-yourdon/
[86] https://deprogrammaticaipsum.com/lazarus-come-forth/

Wirth[87]'s brilliant mind (a language that, just 15 years later, would become a staple of programming curricula all over the world.)

That same year (seriously!) the report "Curriculum 68: Recommendations for Academic Programs in Computer Science"[88] appeared in the pages of Volume 11 of the Communications of the ACM. This was the first attempt at a formal definition of a study program for future generations of computer scientists and programmers.

The impact of this document on academia is hard to ignore. Richard Hamming[89] mentioned this landmark event during his 1968 (again!) Turing Award lecture:

> The topic of my Turing lecture, "One Man's View of Computer Science," was picked because "What is computer science?" is argued endlessly among people in the field. Furthermore, as the excellent Curriculum 68 report remarks in its introduction, "The Committee believes strongly that a continuing dialogue on the process and goals of education in computer science will be vital in the years to come."
> (...)
> For example, let me make an arbitrary distinction between science and engineering by saying that science is concerned with *what* is possible while engineering is concerned with choosing, from among the many possible ways, one that meets a number of often poorly stated economic and practical objectives. We call the field "computer science" but I believe that it would be more accurately labeled "computer engineering" were not this too likely to be misunderstood.

(Richard R. Hamming, "One Man's View of Computer Science", 1968 ACM Turing Award Lecture.)

In the decades that followed, the ACM Curriculum has been (understandably) updated many times, and this author has found online copies of the 1978[90], 1991[91], 2001[92],

[87] https://deprogrammaticaipsum.com/niklaus-wirth/
[88] https://dl.acm.org/doi/10.1145/362929.362976
[89] https://en.wikipedia.org/wiki/Richard_Hamming
[90] https://dl.acm.org/doi/10.1145/359080.359083
[91] https://dl.acm.org/doi/book/10.1145/2594148
[92] https://www.acm.org/binaries/content/assets/education/curricula-recommendations/cc2001.pdf

2005[93], 2016[94], and 2023[95] editions. These are massive documents, describing in detail the subjects, topics, and activities to be conducted in order to guide new students through the maze of computer technology.

The Pioneer

Around 1970, Seymour Papert[96] applied the Constructivist theory[97] of Swiss child psychologist Jean Piaget[98] in order to come up with Logo[99], a language geared towards teaching programming to young kids.

Logo was (still is, actually) quite a controversial topic in teaching circles. For some, the mere idea of moving a turtle on a screen (which was exactly what Logo allowed you to do) was too much of a simplification; for others, it was an opinionated approach that had no place in a classroom. In retrospect, the language did not survive to its hype.

The final report of the Didapro 7 / DidaSTIC[100] 2018 conference in Lausanne called "De 0 à 1 ou l'heure de l'informatique à l'école"[101] was edited by my friend Gabriel Parriaux and many others of his colleagues. In the abstract of the opening keynote by Professor Pierre Dillenbourg[102] of the École Polytechnique Fédérale de Lausanne[103], we can read:

> In retrospect, Logo's pedagogical potential fell victim to the level of expectation created by Papert's speech, which was certainly brilliant and charismatic, but promised effects that no pedagogical approach could achieve. If you're promised 1 million, you'll be disappointed to receive half that.

(Gabriel Parriaux, Jean-Philippe Pellet, Georges-Louis Baron, Éric Bruillard, et Vassilis Komis (Eds.), 2018, "De 0 à 1 ou l'heure de l'informatique à l'école", actes du colloque Didapro 7 – DidaSTIC. Berne, Suisse: Peter Lang. http://hdl.handle.net/20.500.12162/1438.

---

[93] https://dl.acm.org/doi/10.1145/1121341.1121482
[94] https://ieeecs-media.computer.org/assets/pdf/ce2016-final-report.pdf
[95] https://dl.acm.org/doi/book/10.1145/3664191
[96] https://en.wikipedia.org/wiki/Seymour_Papert
[97] https://en.wikipedia.org/wiki/Constructivism_(philosophy_of_education)
[98] https://en.wikipedia.org/wiki/Jean_Piaget
[99] https://en.wikipedia.org/wiki/Logo_(programming_language)
[100] https://www.didapro.org/7/
[101] https://orfee.hepl.ch/handle/20.500.12162/1438
[102] https://people.epfl.ch/pierre.dillenbourg?lang=en
[103] https://www.epfl.ch/en/

Translated from French to English by Adrian Kosmaczewski.)

Logo started a long tradition of education research at MIT, which eventually yielded the programming languages Scheme[104] and Scratch[105], the latter a staple in programming labs during the first two decades of the 21st century, with a whole field of investigation to go with it. (We will come back to Scheme in a bit.)

The amount and impact of Seymour Papert's research in the field of programming education is too big to summarize in a single section of a single article like this. Let me just point the interested reader to his 1980 book "Mindstorms: Children, Computers, and Powerful Ideas"[106], widely considered to be the hallmark in the field, and whose name was co-opted by Lego (hopefully with Papert's blessing) for its famous line of construction sets[107].

Let us close this section with a quote from 2001 that describes the towering impact of Piaget's and Papert's research, still felt more than half a century later:

> Psychologists and pedagogues like Piaget, Papert but also Dewey, Freynet, Freire and others from the open school movement can give us insights into: 1. How to rethink education, 2-imagine new environments, and 3-put new tools, media, and technologies at the service of the growing child. They remind us that learning, especially today, is much less about acquiring information or submitting to other people's ideas or values, than it is about putting one's own words to the world, or finding one's own voice, and exchanging our ideas with others.

(Edith Ackermann, "Piaget's Constructivism, Papert's Constructionism: What's the Difference?," January 2001.)

The Personal Computer Era

The spread of the personal computer in the 1980s fundamentally changed the scenario for programming teachers. All of a sudden, schools of all levels and types could access, if they had the required monetary means, to a level of computing power that merely 10 years prior would have been unthinkable.

[104] https://en.wikipedia.org/wiki/Scheme_(programming_language)
[105] https://en.wikipedia.org/wiki/Scratch_(programming_language)
[106] https://en.wikipedia.org/wiki/Mindstorms_(book)
[107] https://en.wikipedia.org/wiki/Lego_Mindstorms

Even primary and high schools started experimenting with offering programming classes to their students, a fact that triggered a lot of controversy and research. Sadly, the conversation shifted around the most banal and useless of debates: which language is the best for teaching programming? (Insert rolling eye emoji here.)

> This perspective suggests that rather than arguing, as many currently are, over global questions such as which computer language is "best" for children, we would do better in asking: how can we organize learning experiences so that in the course of learning to program students are confronted with new ideas and have opportunities to build them into their own understanding of the computer system and computational concepts?

(Roy D. Pea and D. Midian Kurland, "On the Cognitive Effects of Learning Computer Programming", New Ideas in Psychology 2, no. 2, January 1984, 137–68. https://doi.org/10.1016/0732-118X(84)90018-7.)

Donald Norman[108], of "Design of Everyday Things"[109] fame, tried to shift back the conversation to the most important aspects: first, the usability of computers at the time (or rather, their lack thereof); second, to a much-needed distinction between levels of computer literacy.

> Let me quickly come to my main point: the difficulties that we mortals have with computers are unnecessary. If computers are not understandable, the few will dominate the masses, because the secret language of computation leaves out the uninitiated; those who understand make it hard for those who do not.
> (...)
> It is important to distinguish among four levels of computer literacy: 1. Understanding general principles of computation. 2. Understanding how to use computers. 3. Understanding how to program computers. 4. Understanding the science of computation.

(Donald A. Norman, "Worsening the Knowledge Gap*: The Mystique of Computation Builds Unnecessary Barriers", Annals of the New York Academy of Sciences 426, no. 1, November 1984, 220–33. https://doi.org/10.1111/j.1749-6632.1984.tb16522.x)

This paper by Donald Norman states as well the societal risk of increasing inequality

[108] https://en.wikipedia.org/wiki/Don_Norman
[109] https://en.wikipedia.org/wiki/The_Design_of_Everyday_Things

between those countries that can afford computer equipment for their classrooms, and those that cannot. (As an anecdote, this was a fact witnessed by the author of these lines, who experienced a no-computing-whatsoever learning environment in Buenos Aires in 1990, and then a fully-equipped computer room filled with Apple Macintoshes in Geneva in 1991. The contrast could not have been harsher.)

The Internet Era

By the mid-1990s, the rise of networking and the World Wide Web once again drove an acceleration of programming education. The future was online, and schools had to seize the day as quickly as possible. And to be able to face the new technological reality of the world, universities had to increase their professionalism; or at least, that is what Gal-Ezer and Harel thought:

> In fact, there is no clear agreement even on the name of the field. In European universities, the titles of many of the relevant departments revolve around the word "informatics," whereas in the U.S. most departments are "computer science." To avoid using the name of the machine in the title (a problem that prompted Dijkstra to quip that doing so is like referring to surgery as knife science), some use the word "computing" instead.
> (…)
> One of the main lessons we learned from teaching the material was that students must have an appropriate CS background. We cannot stress this statement enough. For example, one student in class was from electrical engineering, another's sole connection to computing was via her use of computers in general education, and a third's CS knowledge was 25 years old. These students simply did not fit in.

(Judith Gal-Ezer and David Harel, "What (Else) Should CS Educators Know?" Communications of the ACM 41, no. 9, 1998.)

(Note: that last paragraph is infuriating and ageist. There, I said it.)

The publication of "Structure and Interpretation of Computer Programs"[110] (SICP) by Harold Abelson, Gerald Jay Sussman, and Julie Sussman in 1984 made Scheme[111] the go-to programming language for education for the following decade.

[110] https://en.wikipedia.org/wiki/Structure_and_Interpretation_of_Computer_Programs
[111] https://en.wikipedia.org/wiki/Scheme_(programming_language)

Schools and colleges all over the world adopted Scheme in the 1990s, but by the end of the decade the industry wanted more Java[112] developers, not Scheme developers, so Java schools[113] started producing Blub programmers[114], functional programming be damned.

After all, even Philip Wadler[115] (of all people!) criticized Abelson and the Sussmans for their choice of programming language:

> This paper contrasts teaching in Scheme to teaching in KRC and Miranda, particularly with reference to Abelson and Sussman's text.
> (...)
> Some people may wish to dismiss many of the issues raised in this paper as being "just syntax". It is true that much debate over syntax is of little value. But it is also true that a good choice of notation can greatly aid learning and thought, and a poor choice can hinder it.

(Philip Wadler, "A Critique of Abelson and Sussman or Why Calculating Is Better than Scheming", ACM SIGPLAN Notices 22, no. 3, March 1, 1987, 83–94.)

Let us speak a bit more about programming paradigms. The peak of the hype curve of Object-Oriented programming[116] happened right in the middle of the 1990s, and it had a strong impact in programming curricula. This debate burned quite a bit of research funding:

> It is a prevailing opinion that learning a programming language equals learning to program. In the call for papers for this workshop it is stated that "Switching to object-orientation is not just a matter of programming language". We suggest rephrasing and strengthening this statement: Learning to program is not just a matter of learning a programming language.

(Jens Bennedsen and Michael E Caspersen, "Teaching Object-Oriented Programming", 2004.)

The research around teaching object-oriented programming to younger generations

---

[112]https://deprogrammaticaipsum.com/write-anywhere-run-once/
[113]https://www.joelonsoftware.com/2005/12/29/the-perils-of-javaschools-2/
[114]https://paulgraham.com/avg.html
[115]https://en.wikipedia.org/wiki/Philip_Wadler
[116]https://deprogrammaticaipsum.com/the-hype-cycle-of-oop/

produced, among other highlights, the BlueJ System[117]:

> An environment for an object-oriented language does not make an object-oriented environment. The environment itself should reflect the paradigm of the language. In particular, the abstractions students work with should be classes and objects.
> (...)
> BlueJ is an integrated Java development environment specifically designed for introductory teaching. BlueJ is a full Java 2 environment: it is built on top of a standard Java SDK and thus uses a standard compiler and virtual machine. It presents, however, a unique front-end that offers a different interaction style than other environments.

(Michael Kölling, Bruce Quig, Andrew Patterson, and John Rosenberg, "The BlueJ System and Its Pedagogy", Computer Science Education 13, no. 4, December 2003, 249–68. https://doi.org/10.1076/csed.13.4.249.17496.)

What about on the other side of the fence? After teachers dropped Scheme because the industry wanted Java developers, should functional languages come back to the classroom? Surprisingly enough, the answer for some was a resounding no. (To be fair, Chakravarty and Keller propose to literally drop all programming paradigms, and instead to focus on, you know, teaching *thinking skills* instead of just teaching programming.)

> Let us start with a controversial thesis: We should not teach purely functional programming in freshman courses! In fact, we should not teach procedural, object-oriented or logic programming either. Instead, we should concentrate on teaching the elementary techniques of programming and the essential concepts of computing as a scientific discipline as well as foster analytic thinking and problem solving skills.
> (...)
> The central thesis of this article is that purely functional languages are ideally suited for introductory computing classes, but only if the focus is on general concepts rather than the specifics of functional programming.

(Manuel M. T. Chakravarty and Gabriele Keller, "The Risks and Benefits of Teaching Purely Functional Programming in First Year". Journal of Functional Programming 14, no. 1 (January 2004): 113–23. https://doi.org/10.1017/S0956796803004805.)

[117] https://en.wikipedia.org/wiki/BlueJ

Sign of the times, the 2022 edition of SICP switched from Scheme to… JavaScript. I guess Douglas Crockford[118] must be proud.

Following the steps of Chakravarty and Keller, other researchers wanted out of the whole "what is the best programming language for education?" debate, and proposed a novel idea: how about teaching "computational thinking" to our younger generations? Particularly given the fact that most researchers knew the time of LLMs was just beyond the horizon; maybe we should focus on teaching kids how to think, in the good old ways of Papert and Piaget:

> Computational thinking confronts the riddle of machine intelligence: What can humans do better than computers? and What can computers do better than humans?

(Jeannette M. Wing, "Computational Thinking", 2006.)

Computational Thinking has had a major impact in programming education research during the past two decades. Noteworthy are the 2018 book "Hello World: How to be Human in the Age of the Machine"[119] by Hannah Fry[120], which explores the subjects of computational thinking and its possible impact in society, and the 2019 book "Computational Thinking Education"[121] edited by Siu-Cheung Kong and Harold Abelson, and freely available in Open Access.

Learn To Code!

The 2000s saw the rise of Jupyter notebooks[122], Julia[123], and R[124], as implementations of Donald Knuth[125]'s concept of Literate Programming[126].

These implementations of Literate Programming were not the first; commercial software packages like Mathematica[127] and Maple[128] had already explored the same idea

[118] https://deprogrammaticaipsum.com/douglas-crockford/
[119] https://www.waterstones.com/book/hello-world/hannah-fry/9781784163068
[120] https://en.wikipedia.org/wiki/Hannah_Fry
[121] https://link.springer.com/book/10.1007/978-981-13-6528-7
[122] https://en.wikipedia.org/wiki/Project_Jupyter#
[123] https://en.wikipedia.org/wiki/Julia_(programming_language)
[124] https://en.wikipedia.org/wiki/R_(programming_language)
[125] https://deprogrammaticaipsum.com/the-art-of-the-art-of-computer-programming/
[126] https://en.wikipedia.org/wiki/Literate_programming
[127] https://en.wikipedia.org/wiki/Wolfram_Mathematica
[128] https://en.wikipedia.org/wiki/Maple_(software)

starting in the 1980s. But the availability of such packages as Open-Source and Free Software implementations changed the game completely, and greatly contributed to their spread.

Needless to say, this distribution model had a non-negligible impact in the current developments around Machine Learning and Large Language Models. It also fueled a whole industry of "Learn to Code!" workshops, training courses, and YouTube videos, still popular today despite the hiring freezes experienced by the software industry since 2022.

Conclusion

This review is purposely short and left out plenty of important books, milestones, and papers that have marked the research of programming education. We cannot name them all, but here go some honorable mentions we must make:

- Regarding the rise of Python in the field of programming education:
  - "Python for Everybody"[129] by Charles Severance.
  - "Python Programming: An Introduction to Computer Science"[130] by John M. Zelle.
- "Teaching and Learning with Jupyter"[131] by Lorena Barba et al.
- "Lifelong Kindergarten: Cultivating Creativity through Projects, Passion, Peers, and Play"[132] by Mitchel Resnick, the creator of Scratch.
- "Inventive Minds: Marvin Minsky on Education"[133] by Cynthia Solomon and Xiao Xiao.
- Last but definitely not least, we cannot forget the work of Alan Kay and Adele Goldberg[134], and in particular their outstanding contribution, namely Smalltalk, whose very name conveys the idea of teaching programming to younger generations in a playful yet sophisticated way.

We will end this short and opinionated summary with a quote by Turing Award winner John Hopcroft, one which we can only agree with:

---

[129] https://www.py4e.com/html3/

[130] https://mcsp.wartburg.edu/zelle/python/

[131] https://jupyter4edu.github.io/jupyter-edu-book/

[132] https://direct.mit.edu/books/book/3134/Lifelong-KindergartenCultivating-Creativity

[133] https://mitpress.mit.edu/9780262039093/inventive-minds/

[134] https://deprogrammaticaipsum.com/adele-goldberg/

> The potential of computer science, if fully explored and developed, will take us to a higher plane of knowledge about the world.

(John E. Hopcroft, “Computer Science: The Emergence of a Discipline”, Turing Award Lecture, 1986.)

Cover photo by the author.