Cultures of Programming

Understanding the History of Programming through Controversies and Technical Artifacts

Tomas Petricek

November 7, 2022

Preface

The first spark that eventually led to this book was a conversation between two programming experts that I witnessed at the NDC conference in Oslo in 2014. They were advocates of two different programming languages, Erlang and Haskell, respectively, talking about how a language can help you write good software. The two experts, both knowledgable and open-minded speakers at the conference, were simply not able to have a reasonable conversation. The Erlang advocate explained how robust error recovery in Erlang makes software reliable, whereas the Haskell advocate was explaining how type systems can be used to eliminate programming errors. One could not understand why bother with types when you will have failures in your software anyway, whereas the other could not understand how restarting a process can make software correct. The two developers were talking about similar kind of programming problems, but each of them was looking at the problem from a different perspective and hence saw very different issues.

Despite the contemporary motivation, this book is about the history of programming. Once you start looking for the kind of disagreement I witnessed in Oslo, you find them in many places throughout the history of programming, ranging from the 1968 conference that popularized the term "Software Engineering" to the 1980s debate about whether formal verification of software is logically possible and the 2000s rise of Agile development methods. Fortunately, the interactions between different ways of thinking about programming do not lead just to arguments. Once you start looking, you also find that many great advances in programming happened when different ways of thinking contributed to a single concept, starting with the birth of the very idea of a *programming language*.

What is surprising about those interactions is that the different basic approaches to programming that have been shaping the discipline since the 1950s remain surprisingly stable over its 70-year history. In the 1950s, the different approaches, which I refer to as *cultures of programming*, originated from the individual disciplines that contributed to programming including logic, electrical engineering, business and psychology. Cultures of programming are often linked to specific communities, but they can also reappear independently in different contexts. In the early days, this diversity may have been a sign of an immature field. Seventy years later, the existence of the same cultures of programming is instead a sign that programming is, and will remain, an inherently pluralistic discipline.

This book retells the history of programming through the perspective of interactions between five cultures of programming: the hacker culture, the engineering culture, the mathematical culture, the managerial culture and the humanistic culture. Many of those cultures are, in some way, a part of the computing folklore, but this book uncovers their more basic nature. I follow a number of historical strands from the 1950s to present day, looking how the different cultures clash over the nature of programming, but also how they contribute to programming concepts such as programming languages, structured programming, types, objects and tests.

Thinking about the different cultures of programming sheds new light on the technical history of programming, but it also provides a new way of thinking about contemporary programming issues. The answer to what is a correct program and how to create one differs dramatically when we see programs as mathematical entities, engineered socio-technical systems or media for assisting human thought. Similarly, the answer to how to best teach programming differs when you see programming as applied mathematics, new form of literacy or a craft that can only be mastered through practice.

This book also relates to broader socio-technical issues that are often the subject of history and philosophy of computing and programming. The history of software engineering cannot be told without mentioning the struggle for managerial control and attempts to develop a more masculine notion of a computer professional. Similarly, the history of interactive programming cannot be told without a reference to the 1960s counterculture and the political ideology of free software. Although I foucs on how different cultures shape the way we program, the individual cultures can also be associated with particular social and political views. This way, the presented book makes it possible to find connections between technical and social history. In other words, this book is more technical than most history books and more historical than most computer science books.

To emphasize the importance of interactions between the cultures, each chapter starts with a dialogue between a teacher and students that represent positions of the individual cultures. Although I sometimes adapt quotes from actual historical debates, the dialogues are largely a work of fiction. In reality, individuals are less explicit about their beliefs and rarely align with a single culture of programming. In fact, the most interesting historical actors often contribute to two or even more cultures over time. The purpose of the dialogues is then to illustrate the clashes between the cultures and the ways in which they can contribute to a single programming concept despite having different basic idea about the nature of programming.

Contents

Chapter 1

Introduction

Teacher: The author of this book seems to think that we all have different ways of thinking about programming. I wonder if that is really the case. Maybe we can find out whether we can agree on what programming is in the first place…

Gamma: I could already raise an objection, because your method is too analytical and may obscure or prevent interesting thoughts, but let's proceed with this exercise. I am curious to hear the answers.

Omega: I do not see how we could disagree on this. Programming is the process of developing a software system that solves some business problem. It has various stages such as requirements gathering, design, development, testing and operation.

Epsilon: You are going in the right direction, but are confusing a few things. We do not develop software in stages like these nowadays. If you are using the Agile methodology, programming, testing and operation are much more closely integrated. More importantly, programming is only a part of the process and is accompanied with tasks like testing and debugging. But I agree programming is fundamentally about building software to solve some business problem.

Gamma: Not solve, but understand! Programming is a way of thinking about problems that forces you to think in a certain, very clear, way that you otherwise do not experience. That is why it is more useful to see programming as a kind of literacy. If you are creating software, you understand the problem domain through programming. At the same time, you are shaping the world through your programming.

Tau: I'm all for treating programming as a kind of literacy, but only because that is how we can teach mathematical thinking. Programming is a form of applied mathematics. It is a process of constructing a textual program in a formal and well-defined language. And you can understand a problem equally well by typing this on a computer or by writing programs formally on a whiteboard.

Alpha: Excuse me, but you keep talking about requirements, thinking and mathematics, but continue to completely ignore actual computers! In the end, there are always bits to be shuffled around and programming is about controlling those. It is about getting the actual machine to do what you want and, at its best, exploring the possibilities of what can be done.

Teacher: We certainly seem to be disagreeing about the nature of programming, but I'm still not convinced that this necessarily changes how you actually program. Let's try to find some example that we can look at in more detail. What is your favorite program or, more generally, an exemplary programming achievement?

Epsilon: I do not have a single specific achievement, but I think we need to acknowledge that very large computer systems are everywhere around us, ranging from our phones to medical devices, and that they generally work reliably. There is definitely something right about the way we are learning to master programming.

Tau: I disagree! I think your standards are very low. Most software we use today has bugs and you often have to learn tricks to work around those. In 2012, a program bug cost Knight Capital $440 million in just 45 minutes and the bug in the Therac-25 radiation machine actually cost human lives. I think we are lucky that the high-profile failures are not even more common...

Gamma: What I find more worrying is that we often do not understand the end effects of programs that we create. Take for example Microsoft's infamous Tay chatbot, which soon learned inflammatory and racist language from users interacting with it.[1]

Teacher: We will get to issues with programming soon enough, but I wanted to start with important achievements and positive examples first. Can we please get back to those?

Epsilon: As I said, I do not think there is a single specific achievement. We are constantly trying to improve the way programming is done and I think we are getting better at it.

Gamma: That is a very general claim that even I would reluctantly agree with, but I think that what you mean by getting better at programming is quite different from what I mean. A concrete example would be helpful...

Epsilon: What I mean by "getting better at it" is that we are able to build software that brings value to companies. A good example of recent improvement is the Agile Software Development movement, which takes as central the idea that business people and developers need to work more closely together.

Gamma: Phrases like "we value individuals and interactions over processes and tools" from the Agile manifesto sound nice. But the Agile movement also subordinates programming into a support role for commercial enterprises. Like earlier software development methodologies, it is still a mechanism for control. Not to mention the fact that all 17 authors of the Agile manifesto are men, often in leading consulting roles, so the perspective they can offer is inevitably very narrow.[2]

Omega: I do not understand the issue you have with control. If you want to build anything interesting, you need a large team of programmers and then you obviously also need some form of team structure or "control" if you wish.

Alpha: I will join *Gamma* in disagreeing with this perspective, but first I'm curious to hear what *Omega* finds to be an example of important programming achievement.

Omega: An example? The development of the Apollo guidance computer (Figure 1.1), which helped to land a man on the moon! The development had its difficulties, but those were resolved thanks to rigorous definition of requirements, followed by careful coding and rigorous testing.

Figure 1.1: Computer scientist Margaret Hamilton poses with the Apollo guidance software she and her team developed at MIT (Source: MIT Museum).

Tau: You are playing it safe! Nobody can disagree that landing a man on the moon is an impressive achievement, but even the Apollo guidance software had bugs.

Omega: That is correct, but those bugs were well-documented and the crew knew how to operate the computer to avoid their effects. Flying with the bugs simply had a lower risk than attempting to fix them at the last minute.[3]

Tau: This is why I find the present state of computer programming unsatisfactory. A program is a formal entity and you can use formal mathematical methods to show that a piece of software is correct. We should build software that is provably without bugs rather than coming up with post-hoc workarounds!

Alpha: Has anybody actually done this in practice? What is your example?

Tau: Formal verification is challenging, but there are some good examples such as the formally verified microkernel sel4. It has been used as a basis for systems that are robust against cyber attacks, including one that controlled an unmanned flight of the AH-6 helicopter.[4] But my example of the most impressive achievement would actually be the ALGOL programming language, which pioneered the idea of treating programs as mathematical entities that can be formally analyzed and made all the follow-up work thinkable.

Teacher: Our examples so far include the Agile movement, the ALGOL language and the Apollo guidance system proposed by *Epsilon*, *Tau* and *Omega*, respectively. *Alpha* and *Gamma*, do you find any of those convincing?

Alpha: They may be convincing, but I do not find any of them very relevant. Most programs that we need are not of the same kind as the Apollo guidance computer software...

Gamma: And it should not be! The amazing thing about computers is that they give you an unprecedented creative freedom. What you can do with a computer is more restricted by your imagination than by technical limitations!

Alpha: The creative freedom of computers is something that has been very much present in the early days when the members of the Tech Model Railroad Club at MIT discovered the PDP-1 computer, started exploring its possibilities and started calling themselves 'hackers'.[5] The hacker culture was behind ARPAnet, a predecessor of the internet and early computer games like Spacewar! If I was to choose one exemplar that is the most relevant to programming today, it would be the UNIX operating system and the C programming language that emerged from those circles.

Omega: I thought that the C language remains widely used only for historical reasons. Aren't most people trying to find safer and more expressive alternatives these days?

Alpha: This is a common myth about the C language, but the power of C is in that it lets you represent and freely communicate with anything on your computer. The C language captured this idea when it was created and it still follows this basic principle.

Gamma: I too think that the early MIT spirit is a source of many good ideas, but I would not choose UNIX and C as the example. To me, these two sound like the exact opposite of systems showing the creative potential of computers! I much prefer more creative use of programming like the Spacewar! game or the graphical system Sketchpad. A contemporary example from this tradition would be a live coding system though. My favorite example is a live coding environment called Sonic Pi.

Epsilon: What do you mean? Live coding as in people showing how to create small programs live during conference presentations or lectures?

Gamma: I mean live coding in the sense of using programming in live musical or multimedia performances. In 2019, there were 70 different Algorave events all over the world where you can see live coded audio-visual performances in action in a club! It shows the creative potential of programming in a very transparent way. Sonic Pi is used in classrooms all over the world to teach programming to kids.

Omega: When *Alpha* started talking about hacker culture, I thought the discussion was becoming a bit mystical, but using live musical performances as an example of the most notable example of programming makes absolutely no sense! How is that using computers for anything useful?

Gamma: Do you not find education and exploring creative potential of computers for a new kind of art useful? If you want to question what kind of use of computers is useful, I would worry more about the ways in which computers get used by large corporations!

Teacher: Do you have anything specific in mind, *Gamma*?

Gamma: One example would be the secret resume screening AI tool built by Amazon that journalists uncovered in 2015.[6] This was supposed to identify good candidates and Amazon trained it on resumes received in the past 10 years. It turned out that the algorithm was heavily biased against women and would penalize resumes including phrases such as "women's chess club captain".

Teacher: This might be an interesting case to talk about. Even though we disagree about the nature of programming and have different heroes, perhaps the plurality of views will let us better identify what the issues with programming are and, eventually, find a better way of doing it.

Tau: I agree the Amazon system is unacceptable. But this is not an issue with the algorithm itself. The issue is that it was used poorly. Presumably, the algorithm just learned a bias that was already present in the training data set.[7]

Gamma: Training data is one issue, but not the only one. It is a perfect example of why you need to think about programming as a human activity. Algorithms are designed by humans, built by humans and used by humans. A human bias can enter the scene at any point during programming and operation of software.

Tau: According to a dictionary definition[8], an algorithm is "a set of mathematical instructions or rules that, especially if given to a computer, will help to calculate an answer to a problem." A set of mathematical instructions is a mathematical entity. An algorithm cannot be any more biased than an addition or a monoid!

Alpha: You are wrong. You can have an instruction `if (gender="M") salary+=1000`, which increases the salary for all men and that is quite clearly biased...

Tau: But this is mixing data with your algorithms! What I mean by an algorithm is a fully generic procedure. An artificial neural network on its own does not contain any hard-coded information about a specific problem in this way.

Alpha: Even if the algorithm is generic, programming always relies on implicit practical human knowledge. In an artificial neural network, you have to set the number of layers, propagation functions, hyperparameters and so on. All of these can be a source of bias. The issue with algorithms like neural networks is that it is very hard to gain the practical experience that you need in order to avoid such issues.

Epsilon: Practical experience can never be enough, but you can build more sophisticated systems that guarantee fairness. For systems that make decisions about individuals, you can use the idea of "counterfactual fairness"[9]. This makes sure that the result given by the algorithms is the same regardless of the demographic group of the individual.

Omega: I have to admit, this discussion is beyond me. I can see that there are more and less problematic algorithms, but which one should I use if I need to, for example, conform with the 'right to explanation' regulation required by the EU GDPR legislative? The industry needs to agree on standards that we could adhere to in order to avoid such problems.

Gamma: A regulation might solve your immediate problem in the short-term. In the long-term, programming needs liberal arts education that includes social sciences, arts and philosophy. Much of the discussion we're just having is already present in Heidegger's work "The Question Concerning Technology" published in 1954!

Teacher: We are getting back to arguing about the nature of things, this time with programs and algorithms. But I'm still curious if we can use our different perspectives in a more productive way.

Omega: I don't see how I could have a productive conversation with anyone who thinks about programming without thinking about the actual business context in which programming is typically done.

Alpha: It will always be counter-productive to just talk. What matters is code!

Teacher: Would you care to clarify what you mean by this *Alpha*?

Alpha: For example, I enjoy working with the Rust programming language. Rust uses static types to check that your program uses memory correctly and this is based on ideas from theoretical programming language research. Now, I do not understand much of the theoretical gibberish about types and I certainly don't plan to read Bertrand Russell's Principia Mathematica, but it is clear that the academic research contributed to the implementation of Rust. We should talk less and code more!

Omega: Exchanging technical ideas through code may work, but if you work in business context, you get to a point where you do not actually write much code, but instead lead and manage teams. I don't see how that can be guided by code.

Gamma: How about object-oriented design? The idea of objects is something that appeared in the creative context of Sketchpad that I already mentioned. To the displeasure of some of the original contributors, it later evolved into an object-oriented design methodology with complex UML diagrams and things like that.

Omega: Hm. I think you may be right here. UML modelling is certainly a useful method for business software development. It lets you capture the requirements and the design of a system in a way that can be easily implemented, but without worrying about unnecessary details.

Epsilon: I can relate to object-oriented programming as well. It definitely makes programming easier because of the many tools that you can create around it. A modern software development environment for an object-oriented language like Java helps you navigate around code, search through the classes that exist in the system or automatically offers available methods.

Alpha: I'm not sure if you need all of this, but object-oriented programming certainly did get combined with the C programming language that I was talking about earlier, resulting in the widely used language C++, so I can also see some merit in those ideas.

Tau: I for one do not think object-oriented programming is a particularly good idea, but at least it is something you can model formally. And I'm happy to say that mathematics has proven itself useful in this context as well by uncovering the principles of objects.[10]

Teacher: I'm happy that we can conclude our discussion on a positive note. We may not be able to agree about basic principles such as the nature of programming and the best way of doing it, but our different views can contribute to the development of specific programming concepts! I'm looking forward to seeing how this works out in the upcoming chapters…

Introduction

The 440 Million Dollar Bug

The day did not start well for the financial services firm Knight Capital on August 1, 2012. Before the trading opened, the firm deployed a new version of its automated routing system for equity orders known as SMARS. The system received parent orders from other components of the Knight's trading systems and turned them into one or more smaller child orders sent to stock exchanges. In the 45 minutes after the trading opened on August 1, SMARS received modest 212 parent orders, but generated millions of child orders because of a software issue. These orders resulted in 4 million executed trades of 154 different stocks, greatly exceeding the intended number of trades. Knight Capital lost $440 million as a result of these trades.

The bug illustrates that trading algorithms are not simply "faithful delegates of human beings" but take the role of more active actors.[11] The bug also reveals the many subtleties of programming and operation of computer systems that we need to understand if we are to make sense of how algorithms and programs affect our society. The subtleties are well-documented thanks to an investigation conducted by the Securities and Exchange Commission that eventually made Knight Capital pay further $12 million for violating a rule that requires firms to adequately control risk associated with direct market access.[12]

The new version of the SMARS system, deployed on August 1, contained new functionality for accessing Retail Liquidity Program (RLP) provided by the New York Stock Exchange. This was supposed to replace older "Power Peg" code that was no longer in use and the developers repurposed a flag that originally enabled Power Peg to instead activate RLP. On August 1, the new version of SMARS was successfuly deployed on seven out of eight SMARS servers, but one server kept running the old version of the system. One this server, the newly enabled flag thus activated the old Power Peg code. This code was no longer tested and because of other changes in the system, it was not correctly checking that enough child orders have been generated and kept issuing more and more orders. When the technical staff at Knight Capital started investigating the issue, their initial thought was that there is an error in the new functionality and they temporarily rolled-back the previous version of SMARS. This only made matters worse by running the Power Peg code on all eight servers and the whole system was eventually disconnected from the market.

What is interesting about the Knight Capital bug is not just its complex nature, but also the speculations about how the error could have been prevented that it provoked. Envisioning future computing revolutions is a popular pastime in the field of computer science[13] and counterfactual speculations about what could have been are often used to argue for a specific vision or agenda. The speculations also reveal the different ways of thinking that are specific to the different cultures of programming. I would like to discuss three examples that specifically reference this bug.

The Knight Capital bug makes for a perfect example for a financial startup pitch. Indeed, it was featured in the invited keynote "Formal Verification of Financial Algorithms, Progress and Prospects"[14] by Grant Passmore at a 2017 workshop dedicated to the ACL2 theorem prover. The speaker, a co-founder of an automated reasoning startup believes that financial algorithms are "the killer app for formal methods". The specific vision outlined in the talk is that matching of financial orders could be mathematically formalized and analyzed to formally prove that it follows the various required financial regulations. Knight Capital is mentioned not as a specific example for this goal, but as a general example of "glitches" that make financial markets "notoriously unstable".

What is left implicit in the talk is the idea that formal mathematical methods provide a way for solving all the various issues that plague financial software. This is exactly a type of, often unfulfiled, promise "See what computers are on the verge of doing. It will be revolutionary!" often made by computer specialists.[15] In reality, the Knight Capital issue is way more subtle, because the error was caused not just by buggy code for the Power Peg, but also by incorrect manual deployment of the system. To avoid the issue, we would not just need to exactly specify what the desired trading behaviour is, but we would also need to automate and formally verify the deployment. This is, perhaps unsurprisingly, a subject of various ongoing research projects, but it also shows that there will always be potential sources of error, outside of the scope of what has already been formalized.

Treating programs as mathematical entities that can be formally analyzed is the cornerstone of the mathematical culture of programming. As we will see in Chapter 2, the idea is commonplace in academic computer science. It often finds its way to industry, but often through envisioned revolutions that have not quite happened yet. The fact that the mention of Knight Capital appears in an invited talk at an academic workshop is also not inconsequential, because academic venues are often a mechanism through which knowledge is shared in the mathematical culture of programming.

The Knight Capital bug also prompted many comments from the professional software development community. It happened at a time when the "DevOps movement" was gaining popularity in industry and Knight Capital served well as a cautionary tale. The DevOps movement envisioned a very different way of avoiding the issue than that imagined by the mathematical culture. DevOps is a general term that refers to a range of engineering practices that more closely integrate the "development" of software with its "operation". DevOps aims at a "rapid IT service delivery" and "utilize technology – especially automation tools"[16] to make the deployment process more efficient, reliable and free from potential human mistakes. A blog commentary on Knight Capital written by Doug Seven, working at Microsoft at the time, reiterates the belief that "deployments need to be automated and repeatable" and speculates that this would have prevented the Knight Capital disaster: "Had Knight [Capital] implemented an automated deployment system [the error] would have been avoided."[17]

Again, the response illustrates several cornerstones of the programming culture that it emerged from, that is, the engineering culture of programming. Much of the work done in the context of the engineering culture originates from practitioners. Terms like DevOps or Agile, which we will encounter in Chapter 4, often capture sets of practices that have developed organically in the community. The ideas spread through industry conferences, books, blogs, reports and also through commercial trainings offered by respected engineers. The engineering culture recognizes that humans inevitably make mistakes and tries to find tools and processes to make those mistakes less frequent and less severe. In our

case, we have a respected practitioner writing blog post that recommend the use of tools for deployment automation. Such tools do not offer a formal guarantees, but they would likely be sufficient to eliminate the error. The engineering culture would also advise Knight Capital to remove "dead code", that is the Power Peg code that was no longer in use, which would also avoid the issue.

Last but not least, the Knight Capital bug also prompted a direct response by the Securities and Exchange Commission (SEC) that investigated the firm for violating "Rule 15c3-5" that has been adopted by the comission in 2010 to control the risks associated with automated trading. Amongs other obligations, the rule requires trading firms with direct market access to implement "financial risk management controls and supervisory procedures" that "prevent the entry of orders that exceed appropriate pre-set credit or capital thresholds, or that appear to be erroneous."[18] The SEC investigation of Knight Capital[19] documents that the "system of risk management controls and supervisory procedures [adopted by Knight Capital] was not reasonably designed to manage the risk of its market access" as required by the rule 15c3-5. The investigation stops short of saying that having such controls would have avoided the bug, but doing so was the motivation for introducing the rule 15c3-5 in the first place.

The analysis of the Securities Exchange Commission illustrates a way of approaching programming that I refer to as the managerial culture. The rule 15c3-5 is a good example of work done in this culture, because it attempts to address a programming issue through a higher level regulation of the system. It requires "risk management controls and supervisory procedures" that may have different form, but must include several checks before an order is submitted and human oversight of the executed trades. The compliance with the rule is not checked through some formal proof or by a tool, but by a human process. The CEO of the trading firm is required to certify, on an annual basis, that the controls and procedures are implemented. The format of the regulation is again typical for the managerial culture. It takes form of a rather lengthy document written in formal English, produced by an organization. We will see that this is often the case for the managerial culture, for example when we encounter the IEEE standards for testing, documentation and software verification in Chapter 4.

The three responses, mathematical, engineering and managerial do not cover all the cultures that I am writing about in this book, but they are the perfect introduction to the ways in which cultures of programming differ. They show that cultures adopt different ways of thinking about programming, envision different ideals, as well as share knowledge in different ways and have very different requirements for trusting software. The remaining two cultures that I write about are the hacker culture and the humanist culture. The former values direct engagement with the machine and views programming as a highly individual skill. The hackers would likely be surprised that deployment was done by a separate person and that the technical staff needed so much time to understand that something is going wrong. Finally, the humanist culture views programming as the extension of human thought and often envision broader social implications of technology. They would likely wonder if fully automated trading, without any human involvement, was a socially beneficial idea in the first place.

A New Perspective on the History of Programming

It is easy to get lost in the various proposal for how the Knight Capital bug could have been prevented. Should the firm have implemented more supervisory procedures, used formal verification, employed more automation or built a system with more immediate feedback? It is likely that any of these would have been sufficient, but how do programmers decide which path to advocate? The perspective of *cultures of programming* that I develop in this book provides an effective structure for analyzing developments and debates such as this one.

In the case of the Knight Capital bug, we looked at the arguments and proposals coming from different cultures of programming, but we saw only little interactions between the cultures. Yet, the interactions between the cultures will be the focus of this book. We will see that the proponents of the different cultures of programming often clash when the basic nature of programming is at stake. Is programming a matter of constructing the source code of a program that is then compiled or executed, should we see it as a a process of interacting with a stateful computer system, or should we see it as large-scale engineering effort that needs to be carefuly planned in advance?

The proponents of the different cultures of programming can also productively collaborate. This is often the case when multiple different cultures contribute to a shared technical concepts. A technical concept such as a programming language, test, type or an object may mean a different thing to each of the cultures, but this does not prevent other cultures to come up with new ways of using it and extend it.[20] For example, when the idea of a programming language emerged in the 1950s, it represented a formal mathematical language to the proponents of the mathematical culture, a clever trick for making programming easier for the hackers and a way of building software that is independent on a specific machine for the managerially minded users of computers. Despite the different interpretations, each of the cultures was able to contribute something to the shared notion of a programming language, be it a commercial motivation, compiler implementation or a language definition. I will even argue that such collaborations often advanced the state of the art of programming in ways that would unlikely happen within a single consistent culture.

In the following five chapters, I retell the history of five different strands in the history of programming, viewing the interesting moments in the history as interactions between the different cultures of programming. We will see that this perspective sheds new light on interesting events throughout the history of programming, ranging from the birth of programming languages in the 1950s to the development of Agile programming methodologies of the 2000s. Although this book is primarily about history, the perspective of cultures of programming is equally useful for making sense of contemporary controversies around programming and possibly even for imagining different directions for the future of programming. I refrain from speculations about the future in this book. I do, however, briefly discuss how the perspective of cultures of programming sheds light on current debates about program correctness, understanding of algorithms and the issue of programming education later in this chapter.

The concept of a culture of programming is a post-hoc construction that provides an explanatory narrative for the history of programming. Most of the cultures are based on notions that are a part of the computing folklore, but this book adds more depth. Throughout the book, I will discuss how the different cultures think about programming, what meth-

ods they use in their work and how they exchange information and build knowledge. Still, the cultures of programming do not exist in some objective epistemological sense. They are derived from the broad range of examples that I discuss in this book and they provide a fitting explanation for those. This does not mean that programmers are aware of those in their own work or that there is some objective way of determining what work originates in which culture or what individual belongs to which culture. In fact, many of the computing pioneers that we will encounter in this book combined the perspectives of multiple cultures of programming, which may well be the reason why they are remembered.

For these reasons, the characters that appear in the opening dialgoues of should not be seen as actual historical actors. They are idealized representations of the different ways of thinking about programming and methods of working. The next five sections provide a gentle introduction to the five cultures. Those are the mathematical culture, hacker culture, managerial culture, engineering cultur and humanistic culture, represented throughout the book by *Tau*, *Alpha*, *Omega*, *Epsilon* and *Gamma*.

Program as a Mathematical Entity

The key characteristic of the mathematical culture of programming is that it treats programs as mathematical entities. Consequently, a program can be studied using formal methods and it becomes possible to prove that it is correct. This may seem obvious to a reader familiar with the basics of computer science, but it is not at all obvious if we look at the history of programming. The first computer programmers came from a variety of backgrounds and included engineers, scientists, mathematicians and female "human computers".[21] In the early days, the mathematicians were involved more in planning of the computation and in the numerical analysis of the equations to be computed by the computer than in producing codes to instruct the machine. In fact, the ENIAC, which was the first programmable general-purpose electronic computer, was initially programmed by physically plugging wires to connect components. A program was the physical setup of the machine, rather than some mathematical entity!

The path to the mathematical way of looking at programming relied on both technical and social developments. On the technical side, programming evolved from plugging of wires to, first, writing sequences of machine intructions and, later, to writing of symbolical formulas that were processed by a computer program such as the FORTRAN formula translator. On the social side, the mathematization of programming was a part of a broader attempt to establish computer science as a respectable discipline in the university context.[22] The mathematical perspective provided the, politically much needed, rigorous theoretical foundations.

The ALGOL programming language, which appeared in the late 1950s, is a paradigmatic example that illustrates both of these developments. It differed from its predecessors in that it has been formally specified and was independent of any particular machine. ALGOL was designed by a committee set up by the ACM, a professional organization that was one of the key forces aimed to turn computer sicence into a respectable academic discipline. Although ALGOL was never widely used and was a failure as a practical programming language, it was regarded as an "object of beauty" by the proponents of the mathematical culture.

The ALGOL specification also included several variants of the language, one of which was so-called "publication language" to be used in publications involving ALGOL code. This peculiarity has a significance too, because academic publications are the primary way of exchanging knowledge in the mathematical culture of programming and this was directly supported by ALGOL. The new way of thinking about programming, established by ALGOL, inspired a plethora of theoretical and practical developments. We return to the birth of programming languages and ALGOL in Chapter 2 and discuss one specific development, the notion of types, in Chapter 5. For example, variants of the ALGOL publication language were often used to write algorithms that were published in the regular "Algorithms" column published regularly by the ACM magazine, Communications of the ACM, throughout the 1960s and the 1970s.

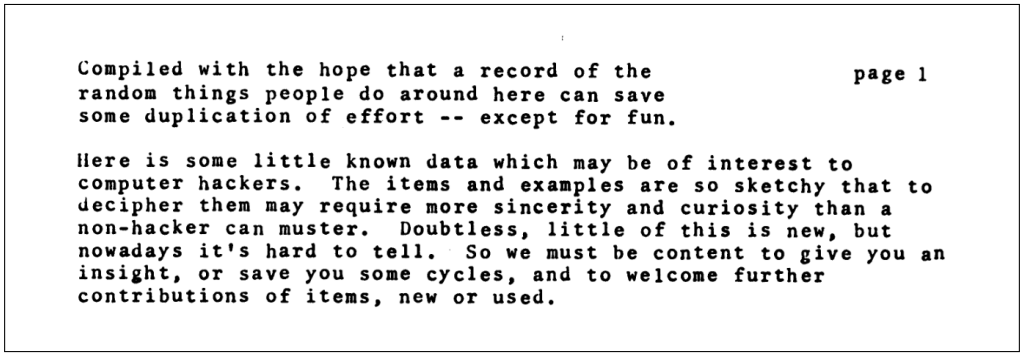
Perhaps the most basic question that arises from the mathematical way of thinking about programs is whether it is possible to prove that a program is correct. The proponents of the mathematical culture believe that this is, indeed, the case. For them, a computer program is a mathematical entity just like a logical proposition and so we can formally verify its properties. The fact that this view was not universally accepted generated a heated debate that we cover in Chapter 2. A number of objections were raised. First, it is often hard to specify what a 'correct' behavior is. Second, programs consisting of millions of lines of code cannot be compared to one-line logical propositions, even if just because of their size. Finally, programs are not merely abstract, but have causal effects on the physical world. Today, proving that a program is correct remains an inspiring challenge for those who share their basic conception of programs with the mathematical culture of programming and a deluded illusion for others.

The case of the sel4 microkernel and the unmanned AH-6 helicopter, mentioned in the opening dialogue, shows some of the challenges. The control software is too large to be formally verified as a whole. The authors instead focus on showing the correctness of some of the key components and giving a formal guarantee that untrustworthy parts cannot interact in unexpected ways. Most of the proofs are also too large for a human to check. Instead, they are checked by another computer program, a proof assistant. There also remains a large number of assumptions, both about the hardware and about some of the source code (such as 1200 lines of startup code in the kernel).[23]

The practical difficulties explain why the ALGOL language is a better paradigmatic example for the mathematical culture of programming than, for example, the formally verified sel4 microkernel that controls the AH-6 helicopter. Another reason is that it is reasonably possible to see ALGOL as a product of one particular culture of programming. It would be hard to make such claim about the sel4 microkernel. Although it partially fulfils a dream of the mathematical culture, its development required a non-trivial number of hacker programming tricks and huge and well-managed engineering effort.

The Hands-On Imperative

Whereas the mathematical culture aims to keep a distance from the technical details of program execution, purportedly to work with a more basic and universal notion, the hacker culture seeks the exact opposite. Direct engagement with the machine and the nitty gritty details of program execution are the cornerstone of the hacker way of thinking about programming. This way of thinking may appear more logical, historically. The first programmers of the first digital computers were often also their creators and so direct engagement



Compiled with the hope that a record of the random things people do around here can save some duplication of effort -- except for fun.

page 1

Here is some little known data which may be of interest to computer hackers. The items and examples are so sketchy that to decipher them may require more sincerity and curiosity than a non-hacker can muster. Doubtless, little of this is new, but nowadays it's hard to tell. So we must be content to give you an insight, or save you some cycles, and to welcome further contributions of items, new or used.

Figure 1.2: An introduction from HAKMEM, an MIT AI Lab technical report

with the computer was a necessity at first. But as we will see in Chapter 4, the machine soon started to disappear from the picture. In the 1950s, most comptuers were operating in the "batch processing" mode where programmers submitted their jobs on punched cards to an operator and then had to wait for several hours to get their results back. The hacker way of thinking about programming had to be reinvented in the early 1960s.

At MIT, computers were studied since before the digital era. We may speculate that the early hands-on experience contributed to the reinvention of the hacker culture in the 1960s. Some thirty years earlier, Vannevar Bush created his differential analyzer, which was a mechanical analog computer for solving differential equations. During the World War II, MIT was the home of the Radiation Laboratory that brought together numerous engineers and scientists to work on the radar technologies. Military applications were also behind the Whirlwind computer, built at MIT at the turn of the 1950s, which was one of the first digital computers that operated on real-time input. Finally, in an effort to prototype the follow-up to Whirlwind, MIT engineers built TX-0, which was an experimental machine utlizing the new transistor and magnetic core memory technologies, completed in 1955.

As with the mathematical culture of programming, the reinvention of the hacker culture in the 1960s relied on a combination of social and technological developments. The TX-0, alongside with its commercial offspring PDP-1, made it possible to directly enter instructions and immediately observe the results, but the two machines also became accessible to the MIT community under a very liberal access policy. Together, these two characteristics made the machines appealing to a community of tinkerers, many of whom came from the MIT's Tech Model Railway Club (TMRC), who started calling themselves "hackers" and began exploring the possibilities of the machines.

The MIT hacker culture that emerged from this configuration has become a part of the computing history folklore and many accounts ignore the broader social counter-cultural context of the hacker culture.[24] In particular, they fail to question the contrast between the masculine reality of the hacker culture and the alleged hacker principle that hackers should be judged merely by their skill and not "bogus criteria such as degrees, age, race, sex, or position."[25] Nevertheless, the hacker ethic, documented as part of the history folklore, does a good enough job explaining the way of thinking about programming and the programming practices followed by the hacker culture.

First of all, the hackers believe in direct access to the computer, which gives them the opportunity to understand and improve things. They are keen to use computers for various playful and not immediately useful purposes, such as the development of the famous

Spacewar! game that I return to in Chapter 3. Hackers do not care about getting degrees, professional recognition or publications. Instead, they want to be regarded as wizards by their fellow hackers. They believed that all information should be free and started using the ARPAnet (a predecessor of the internet) for sharing knowledge as soon as it became possible. The hacker approach to knowledge is perhaps best illustrated by the introduction from an MIT technical report known as HAKMEM[26], shown in Figure 1.2, which collects 191 assorted items of knowledge ranging from the variance of a pseudo-Gaussian distributed random variable to an electrical circuit for an amplifier ("submitted without further explanation or cautions").

The MIT hackers shared many of their beliefs about programming with a broader community of similarly minded programmers who did not necessarily call themselves "hackers" but had the same interest in direct engagement with computers, improving systems and information sharing.[27] Outside of the liberal "ninth floor" at the MIT Tech Square building where TX-0 and PDP-1 were housed, it was not always possible to follow all the principles of the hacker ethic. The best example of this is the group of programmers at Bell Labs that created the UNIX operating system and the C programming language at the end of the 1960s. UNIX and C were a product of what I call the hacker culture. They were tools created by hackers for other hackers and supported a very direct working with the machine. Although UNIX was created in a commercial research environment, its authors saw themselves as "rebels against soulless corporate empiress"[28] and it was effectively distributed as free software. The MIT hackers and UNIX hackers eventually fused into a single community, partly thanks to the sharing of software and partly thanks to the ARPAnet, at least until the early 1980s when AT&T started distributing UNIX as a commercial product. Many MIT hackers could not accept the commercial nature of UNIX, which led to the development GNU project and the "free software" movement in the 1980s.[29]

In this book, I identify traces of the hacker thinking about programming in a wide range of programming practices and artifacts. Because of its disregard for professional recognition and focus on free sharing of information, the specific contributions and influences are harder to follow than in the mathematical culture, with its academic publications and citations. In many cases, an idea originats in the hacker culture, but is then adopted and subverted (or "further developed") by other cultures. Examples include automatic coding discussed in Chapter 2 and the idea of objects discussed in Chapter 6. Some hacker practices are, however, strangely elusive to such attempts. Two examples that we encounter in this book are interactive programming in Chapter 3 and debugging in Chapter 4.

Software Development Lifecycles

Both the mathematical culture and the hacker culture of programming emerged from the university environment. However, the computers of the 1940s and the 1950s were used first for military purposes and later for business applications. Many of those systems were large-scale and required a highly organized and coordinated development effort. The difficulty of programming was a surprise to both individual programmers and to managers.

A prime example that illustrates the challenges of programming is the software for the Semi-Automatic Ground Environment (SAGE), an air defence system consisting of a network of radar stations, connected to a network of computers that was designed to detect a Soviet bomber formation approaching the US to perform a nuclear strike (Figure 4.10). The designers of SAGE approached the problem with the mindset of electrical engineer-

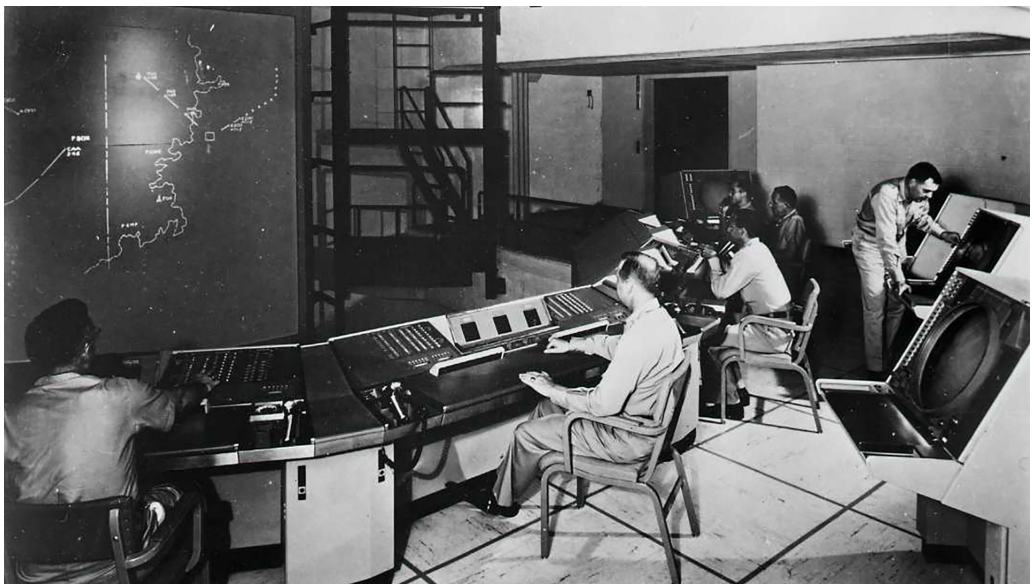


Figure 1.3: A control room of the Semi-Automatic Ground Environment (SAGE)

ing, physics and mathematics and did not see programming as a major challenge.[30] When SAGE was conceived, the designers imagined the programming task could be specified and then assigned to a contractor, just like the production of conventional electornic components. However, it soon turned out that the development of the SAGE control system was a task of unprecedented complexity. According to an expert at IBM, building the system would require 2,000 programmers – at a time when the total estimated number of skilled programmers in the United States was just 200.[31]

Large-scale military and business programming projects, such as SAGE, gave rise to yet another way of thinking about programming that I refer to as the managerial culture of programming. In this culture, programming is seen more as a production-like activity. Consequently, the managerial way of thinking places emphasis on specification, planning and organizational structures. The context from which the culture emerged differs from the hacker and mathematical culture both in the environment and in the kinds of projects that it tackled. However, as the case of UNIX (and later Linux) shows, production of large-scale software is not limited to the managerial culture.

The SAGE system was eventually built and became operational in 1959. It used the Whirlwind computer built at MIT that I mentioned when discussing the hacker culture and MIT was also initially involved, before establishing the MITRE spin off for the more production-oriented work.[32] Similarly, the RAND corporation that was initially responsible for the software development established a spin-off company, SDC, which trained over 7,000 programmers and defined one of the first methodologies for managing the development of large-scale computer systems. As discussed in Chapter 4, the development was organized in phases, starting from operational plan and machine specifications; coding was preceded by coding specification that defined the interfaces between components and was followed by several testing and evaluation phases. The careful planning, typical for the managerial approach, aims to minimize the dependence on individuals. Rather than being highly individualistic hackers, programmers are seen as replaceable factory workers who

are hired, trained and then integrated into a structured development process, where they complete small-scale programming tasks according to the given coding specifications.[33]

The development of the Apollo guidance computer software is another early example of a mission-critical software system that pioneered the idea of careful management of the development process. This was even more necessary because the software development was done at MIT rather than directly at NASA. For example, all modifications to the specification of the on-board software had to be approved by the Software Configuration Control Board and MIT "could not change a single bit without permission."[34] The development process itself followed the, later standard, sequence of phases. It started with the requirements gathering phase in which NASA and MIT jointly prepared the Guidance and Navigation System Operations Plan (GSOP) that specified the required functionality to the level of equations to be computed. This was followed by coding, validation that consisted of unit and integration testing and, finally, production. NASA also defined four review points that resulted in the acceptance of outcomes from individual phases.

Despite the best managerial efforts, the process was a source of concerns and NASA and MIT produced "high quality software, primarily because of the small-group nature of development at MIT and the overall dedication shown by nearly everyone associated with the Apollo program."[35] In other words, even the Apollo guidance software, which was very much a product of the managerial culture of programming, relied on some degree of hacker qualities.

The development of SAGE and Apollo guidance computer are just two of the early examples of programming projects that emphasized structuring of the development process and controlling work done by programmers over actual code. This approach is in stark contrast with the hacker culture where code is the only thing that matters. The managerial culture was crucial for the development of "Software Engineering", a topic that I follow in Chapter 4, which can also be seen as a revolt against the early hacker-dominated approach to programming. However, most of the cases where I discuss the managerial culture in this book are not about struggle for control, but follow another pattern. It is one where the managerial culture adapts a previously mainly technical concept so that it can be used not just for structuring programs, but also for structuring the teams developing them. This is the case with structured programming in Chapter 2, testing in Chapter 4 and also object-oriented programming in Chapter 6.

A Proper Engineering Discipline

During the 1950s, it became clear that the task of creating programs was much harder than initially expected. Programming was "a black art" that relied on "private techniques and inventions" of individual programmers.[36] The hackers had no problem with this and were happy to continue developing their own private inventions, share them with other hackers and continue exploring what can be done in this way. However, outside of the hacker culture, this state of the art was seen as problematic at best. The belief that "the black art of programming [has] to make way for the science of software engineering"[37] culminated with the 1968 NATO conference on Software Engineering. The conference showed a resounding agreement on what the problem is, but very little agreement on how the problem should be addressed.[38]

Those with mathematical background started searching for ways of specifying programs in more rigorous ways, with the hope that mathematical methods could be used to tackle the complexity of programming. Those with managerial inclinations started coming up with processes to organize the work, with the hope that division of labor can lessen the reliance on individual programmer skills. However, a third approach to managing the complexity also started appearing.[39]

The way of thinking that I refer to as the engineering culture of programming recognizes the difficulty of programming. It does not aim to reduce it, either to a mathematical or to an organizational problem. Instead, it aims to tackle it more directly, using a combination of good engineering practices, including testing, monitoring and overengineering. The practices are often supported by tools that utilize the computer itself to simplify the task of programming.

A recurring theme in the engineering culture is an attempt to find a more scientific way of programming and an attempt to approach the problem with a high degree of professionalism.[40] The engineering culture of programming also draws on ideas from other cultures. After all, the willingness to learn from all possible sources is one of the characteristics of the professionalism of the engineering culture.[41] The three types of outcomes from the engineering cultrue of programming that I will discuss in this book are programming styles, programming practices and programming tools.

The idea that good structuring of code can make programming easier is at the core of many developments arising from the engineering culture. One of the earliest examples is the idea of structured programming, popularized by Dijkstra in 1968.[42] At the time, many programmers were still used to writing code as a linear sequence of instructions with jumps that transfer the control to another location in the sequence. Structured programming replaced this with higher-level constructs such as loops, which execute a certain block repeatedly, either for a given number of times or until a condition is met. This made programming easier because the structure of the code more directly indicates what happens when the code is executed. I return to structured programming in Chapter 2 and we will also see the same general principle, structuring code so that it is cognitively easier to work with, when discussing object-oriented programming in Chapter 6.[43]

Another way of making programming easier, in addition to structuring code in a certain way, is to structure the programming process in a certain way. Over much of the history of programming, the question of programming methodology has been dominated by the managerial culture of programming, which interpreted it as the problem of structuring teams. However, the engineering culture found its own approach to the problem by focusing less on structing teams and more on structuring programming. Early work on this idea appeared in the 1970s,[44] but the most prominent example of the engineering approach to programming methodology is the eXtreme Programming (XP) that emerged at the end of the 1990s.

Extreme Programming was developed by software engineers and it treats coding as the central activity. It introduces a range of practices that help programmers produce better code such as pair programming, where code is written by two programmers on a single machine. Another practice, which has the appeal of scientific method, is test-driven development (TDD). In this method, program exist alongside a collection of tests that can be run automatically. When programming, programmers first write a test to specify the required functionality. The test should fail, because the functionality itself has not yet been implemented. The programmer then completes the implementation, making the test pass

successfully. The scientific appeal of the method is that the collection of automated tests ensures that previously implemented functionality remains correct and so, in principle, the method should gradually coverge to a desired result.

The history of testing, which I discuss in Chapter 4, is particularly interesting, because it shows how a single concept can be shared by multiple cultrues of programming and used for different purposes over time. Prior to 1979, the notion of testing was seen as a step in a managerial software development process and a way of showing that program satisfies its requirements. After 1979, the view gradually shifted and testing became an engineering approach to finding errors, which was supported by a range of testing tools. Finally, test-driven development turned testing into a driving force in an engineering-oriented development methodology. This illustrates both of the interactions between cultures that are central for this book. On the one hand, there is a collaboration around a technical concept where different cultures contribute to the notion of a test. On the other hand, there is a struggle for control over programming. Following the publication of "The Manifesto for Agile Software Engineering" in 2001, the light-weight methods of the engineering culture gradually replaced the heavy-weight managerial approaches as the dominant way of thinking about professional programming.

Media for Thinking

The mathematical, hacker, managerial and engineering cultures would provide a good enough structure for writing about the majority of important practical developments in the history of programming. However, doing that would leave a regrettable gap. There is yet another way of thinking about programming. This culture has a less coherent perspective than the others and includes a loosely connected group of people, works and ideas. It is maintained by artists, educators and computer scientists influenced by arts and humanities including philosophy and media theory. This is also the culture that has been the hardest to name. I refer to it as the humanistic culture of programming, because the concern for humans and their relationship with computers and programming is often at the core of the work done in the culture, be it work on education, systems that envision the future of computing or artwork involving programming. The humanistic culture of programming is less concerned with how to construct particular programs and cares more about what programs can be created and how can humans interact with them. It often treats programming not as a mere tool, but as a medium for thinking.

Perhaps the earliest instance of this way of thinking about computers dates back to before the first digital electronic computer, the ENIAC, became operational. In 1945, The Atlantic published an essay "As We May Think" by Vannevar Bush, who we encountered earlier as the creator of the differential analyzer at MIT. In the essay, which was partly based on his pre-war ideas, Bush responded to the problem of information explosion with the idea for a device in which "an individual stores all his books, records, and communications, and which is mechanized so that it may be consulted with exceeding speed and flexibility."[45] These ideas have since been regarded as precursors of the internet, hypertext and online encyclopedias, although Bush assumed the device would be a mechanical computer, like the differential analyzer, based on microfilm.

Vannevar Bush himself does not fit squrely into any of the cultures and has contributed to a wide range of work on computing, but his essay illustrates two typical characteristics that will appear repeatedly in the humanistic culture of programming. First, it presents a

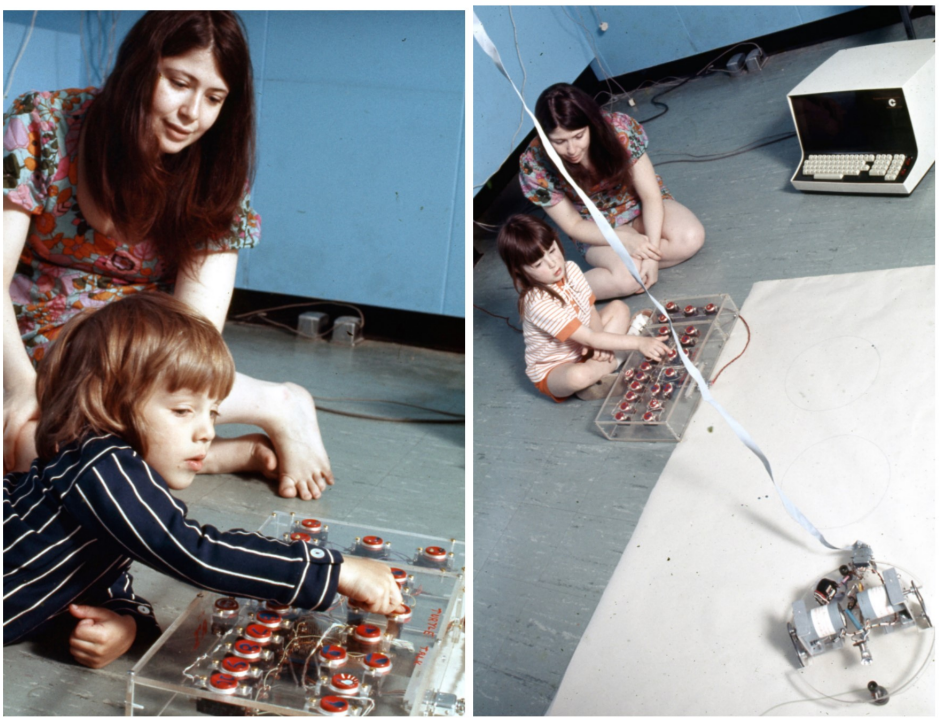


Figure 1.4: Radia Perlman and children playing with a turtle robot, programmable through LOGO, using the Button Box interface designed by Perlman.

vision of the future of human and computer interaction and, second, it imagines a system that will assist humans with thinking and their intellectual endeavours.

Almost 20 years later, the "As We May Think" essay beacame one of the inspirations for Sketchpad, a computer system created in 1963 by Ivan Sutherland. Sketchpad used a graphical user interface to let users construct geometric shapes. At the time when most computers were used in batch processing mode and had no screen, Sketchpad offered a glimpse of a possible distant future. It was also a feat of engineering. It ran on the TX-2 machine, a more powerful successor of TX-0, and utilized the experimental ligth pen input device. The writing about Sketchpad exemplifies both the focus on humans and a way of thinking about programming. It was motivated by the desire to make computers "more approachable" by using "drawing as a novel communication medium."[46]

The humanistic culture of programming thrives when new ways of interacting with computers become possible. Sutherland's Sketchpad was the first computer program with graphical user interface, but it relied on using the most powerful computer available at the time, with custom hardware modifications. This is also the case for the LOGO programming language, which was a "learning environment where children explore mathematical ideas."[47] LOGO is perhaps best known for turtle graphics, a microworld where children control a turtle by entering commands interactively on a terminal. The authors also built a physical robot that could later be controlled using a "button box" that made LOGO accessible to even younger children (Figure 1.4). As I discuss in detail in Chapter 3, LOGO again combines a newly developed interactive way of programming with focus on human thinking and critical reflections on education.

Perhaps the most influential programming language that was significantly influenced

by the humanistic culture of programming is Smalltalk. Despite becoming a general-purpose language that contributed to a wide range of programming concepts, the origins of Smalltalk are quite different from those of other early programming language. A glimpse of the context can be found in the "Personal Dynamic Media"[48] article written by Alan Kay and Adele Goldberg. The article describes the vision of "a personal dynamic medium the size of a notebook" which "could have the power to handle virtually all of it's owners information-related needs." Smalltalk is described as a communication system and programming is referred to as "talking to Smalltalk." In many ways, this article reads reads much like the next iteration of the vision from the "As We May Think" essay, except that Smalltalk now provided a realistic prototype. The fact that Smalltalk is now remembered more as the source of object-oriented programming, which has became largely the subject of the engineering culture of programming is another story and I return to it in Chapter 6.

Yet another revolution in how users can interact with comptuers happened in the 1970s when microcomputers allowed increasing number of people to buy an inexpensive personal computer. This led to a number of more practical developments, discussed in Chapter 3, but it also allowed one novel and eccentrically creative use of programming and computers. In the 1980s, the computer music "The Hub" brought microcomputers to clubs and used them as musical instruments for their performances. The Hub allowed the audience to see their screens, which became a mantra of the flourishing live coding community[49] The work education, rooted in LOGO, and the work on live coded music that started with "The Hub" eventually connected in the form of Sonic Pi, mentioned in the opening dialogue, which is, a live coding system that has been used both in live performances, but also to teach music and programming in schools.

The work in the humanistic culture can take a wide range of forms, but the way of thinking about programming is sufficiently coherent to justify labelling the work as belonging to the same culture. In some cases, there is a direct link that we can follow. For example, a number of individuals and systems were more or less directly influenced by essays like "As We May Think" and other hallmark publications. However, the way of thinking also repeatedly reappears without a direct connection, for example in the context of computer education or computer art. The humanistic culture is also, at least to an extent, truthful to broader humanistic goals. It often aims at greater inclusivity and has a number of widely recognized women contributors such as Cynthia Solomon and Radia Perlman working on LOGO and Adele Goldberg and Diana Merry-Shapiro working on Smalltalk.[50]

The Past and The Present of Programming

The cultures of programming that I introduced in the preceding five sections capture particular ways of thinking about programmnig. Those ways can be traced back to the 1950s or the 1960s, but they can be equally found in contemporary discussions about programming. As we just saw, the different cultures are, to some extent, formed by different communities, such as academic computer scientists or professional software engineers, and by means of exchanging knowledge, such as academic publications or business magazines. However, this is not the case for all of them and individuals can move between cultures or even produce work that combines approaches from multiple cultures.

Vannevar Bush, who we encountered repeatedly in the sections above, is one example. Bush had an engineering background and his pre-war work on the differential analyzer was rooted in traditional engineering. During the Second World War, Bush worked

as an administrator and coordinated much of the US scientific military research. Despite this background, which includes engineering and management roles, his later "As We May Think" essay inspired many working in the humanistic culture of programming.[51]

The computing pioneer John McCarthy is an even better fitting example. In Chapter 2, I will discuss his 1963 paper "Towards a Mathematical Science of Computation", which is a manifesto of the mathematical culture of programming that envisions much of the later work on formal treatment of computer programs. However, McCarthy was also a supporter of the hacker culture at MIT, affectionately referred to as "Uncle John", and hired number of MIT hackers for various projects. He pioneered the idea of time-sharing, which made computers more interactive and enabled lively hacker exploration of programming that I return to in Chapter 3. Finally, he was also one of the founders of the field of Artificial Intelligence (AI). His work on AI and other creative uses of computers, at a time when computers were mainly scientific instruments, would also mark him as a member of the humanistic culture of programming.

The concept of a culture of programming is an interpretation obtained by looking at the past seventy years of programming. This inevitably simplifies some aspects of the complex history, but it also makes it possible to tell an overarching story about the history of programming.[52] The fact that we keep encountering the same cultures of programming throughout the 70-year history of programming makes those concepts also useful for thinking about contemporary debates and issues involving programming.

One example of this was the discussion about the Knight Capital bug that I discussed in the opening of this chapter. In the next three sections, I sketch how the framing in terms of cultures of programming can shed light on three present-time programming issues, namely the questions of program correctness, understanding of algorithms and programming education. There may even be ways in which cultures of programming help us imagine the future of programming. We could, for example, speculate how specific programming concepts used by one culture might evolve if they were adopted by another culture. As interesting as this may be, I refrain from such speculations and focus on the present, in the next three section, and the past of programming, in the rest of the book.

Developing Correct Software

At the end of the 1940s, many believed that programming errors would, along with hardware faults, become less frequent over time.[53] As the many cases of programming errors that we encountered in this chapter show this was an optimistic view and it was soon left behind. Today, many believe that program errors are inescapable. Sometimes, as in the case of Knight Capital, program errors are the source of catastrophic failures. Sometimes, this is not the case. The program errors in the Apollo guidance computer were well understood and carefuly mitigated. In some cases, a program error may even be useful as a source of inspiration. Glitch art (Figure 1.5) uses errors for aesthetic purposes, while an error made in a live coding performance can be used as a source of creative ideas.[54]

Different cultures of programming understand program errors differently, but the attempt to eliminate programming errors has often been a driving force behind many developments. However, different cultures do not just disagree how to best produce error-free programs. They also disagree about the very notion of "correct software". For the proponents of the mathematical culture, programming is a mathematical activity and the kind of correctness that can be studied is thus also a mathematical property. This is typically

Figure 1.5: An example of glitch art. Vernacular of File Formats (2009-2010) by Rosa Menkman. The artwork explores how introducing an error in a compressed file results in different visual effects. The example here is that of the GIF format.

a correspondence between a program implementation and some, more abstract, mathematical specification. The problem of how to obtain the right specification is not a mathematical concern and is outside the scope of mathematical methods. Dijkstra, for example, refers to the problem of obtaining the right specification using the somewhat derogatory term "pleasantness problem"[55] and suggests that it is best tackled by psychology and experimentation. Nevertheless, even this carefully constrained perspective does not avoid all problems with using mathematical methods for reasoning about programs and I will return to some of the challenges in Chapter 2.

To the engineering and managerial cultures, the question of program correctness is a pragmatic one. Does the system work in the way that is required by the customer? The two cultures differ in the methods that they advocate as the right approach to solving the problem. I will discuss the approaches that emerge from the two cultures in Chapter 4. The managerial culture believes that correct software can be built by following a rigorous process that structures the development, carefully integrating a process of requirements gathering, to define the scope of the system, development, testing and maintenance. A stereotypical artifact of the managerial culture is a detailed specification, capturing the requirements of the desired system. Unlike in the mathematical culture, the specification is not intended to be fully formal. For the managerial culture, obtaining the right specification is seen as an important part of software development and so it views "pleasantness" as inseparable from "correctness". As we will see in Chapter 4, the engineering culture places more emphasis on programming methodologies and tools that connect the requirements with the actual system implementation. For example, tests can be seen both as capturing specific requirements, but also as certification that the functionality has been implemented correctly.

Finally, to the proponents of the creative culture, the question of correctness is an ill defined one. Any software has effects on the world and we need to think about such effects. The culture views programming as a social and a political act and so the effects are analysed at such level. A correct software is one that, for example, empowers desirable social structures as in the vision of "a truly free society" developed in "The Telekommunist Manifesto"[56].

The key lesson here is that the culture determines what methods can be used for studying the question of correctness. Such methods, in turn, delineate what is in scope of "programming" and what is outside of the scope. The more systematic the methods, the narrower the scope and the more technical the interpretation of what correctness means. The case of the engineering culture is particularly interesting because, as part of its attempts to capture what software can reliably be built, it had to develop its own new way of reasoning. As we will see in Chapter 4, this relies on logical or even mathematical arguments involving somewhat informal but rigorous notions such as accidental complexity, rate of change in software or forgiveness of the environment.

As the debate about the Knight Capital software error reveals, the different cultures of programming can interpret the same event in multiple ways. The different interpretations are rooted in different worldview and are, to some extent, incommensurable. Not in the sense that they cannot be comprehended, but because the solutions they lead to solve problems that other cultures do not recognize as relevant. The remarkable fact is that programming has been able to maintain this pluralism throughout most of its history.[57] The different views have coexisted and different cultures also often contributed to a single technical artifact. One example of this, which I return to in Chapter 4, is the notion of testing which can be seen as a phase in a managerial software development process, a formal mathematical entity, as well as a tool for engineers. Despite occasional clashes and misunderstandings, the overall practice of programming seems to follow the pluralistic approach of "many things go!"[58]

Understanding Programs

Most software systems are very complex and opaque. When they work as desired, this is not a reason for concern. Alas, complex software systems often exhibit behavior that is unacceptable. Machine translation tool that learns to translate "Russia" as "Mordor"[59] may be amusing to some, but a biased job applicant screening tools,[60] racist predictive policing algorithms,[61] and search engines that discriminate against women of color[62] highlight a serious problem with the state of software. All of these cases raise questions about our understanding of how the complex and opaque software systems that surround us actually operate. Although my focus in this book is primarily on technical aspects of programming, the perspective of cultures of programming can help us think about the different ways in which the working of computer programs can be understood and, in turn, shed some light on how different cultures of programming may view, or be blind to, socio-technical issues.

To the proponents of the humanistic culture, the issues with understanding technology are nothing new. They may point to the French philosopher Gilbert Simondon who, already in the 1950s, raised the issue of our relationship with technology and noted that the lack of understanding makes us passive operators of the machines (or software systems) and alienates the user (or the programmer) from the system.[63] This leads to a situation where the user is controlled by the system rather than the other way round. The proponents of the humanistic culture also firmly believe that there are no technological solutions to social problems. They may pursue a range of different ways for raising awareness of the issue, for example through creative art projects that make the hidden implicit biases in computer systems explicit.[64] They may also pursue projects that aim, inspired by Simondon, to counter the alienation from technology, for example by supporting educational projects that increase the diversity among programmers.

The hackers believe that systems should be designed in a way that "fits a single brain."[65] If a system can be fully understood, its behaviour can also be fully understood and so a system built with a hacker spirit should be free from hidden surprises and emerging behaviour. Even if undesirable behavior is found, the users should have "the freedom to study how the program works, and change it so it does [their] computing as [they] wish."[66] This view, of course, has many problematic assumptions. Most importantly, it assumes that other people using the system are also hackers, who will want to and will have the resources necessary to understand and modify it. That may have been the case for MIT hackers in the 1960s, a community that I discuss in Chapter 3, but it is not the case for any of the problematic systems I mentioned earlier. Yet, the belief that producing systems that can be understood and modified by their users has enabled numerous developments in areas such as data privacy.[67]

The proponents of the mathematical culture would be ready to admit that the complexity of modern software has outgrown scale that can fit a single brain. To follow the framework used by MacKenzie in his account of the mechanization of proofs,[68] the complexity is an inevitable symptom of living in a society of "high modernity". In such society, critical infrastructure consists of complex technical systems and the understanding of such systems is delegated to systems of expertise. The proponents of the mathematical culture believe that formal mathematical proofs as the most trustworthy system of expertise. As with the problem of software correctness, the issue becomes correctly defining the mathematical requirements of software systems.

There are two other approaches to the issues of understanding software and explainability sketched in the opening dialogue. The engineering culture would recognize the earlier examples as engineering failures and would perhaps allude to codes of ethics that govern more traditional engineering disciplines. This does not, however, provide any direct answer as to how an engineer should approach the development of such systems. After a cereful ethical consideration they may recognize certain AI systems as unethical and refuse to work on those. For other systems, the engineering approach may consist of being aware of the threats and building tools to mitigate potential issues. For example, an engineer may develop a tool that attempts to detect and eliminate bias from the underlying system. They may also develop a series of tests to ensure that a system does not exhibit algorithmic bias. Finally, the managerial culture would approach the problem from a similar starting position, but would seek a more explicitly defined solution. One approach may be an industry-wide standard or a government regulation that provides guidance for using artificial intelligence algorithms.[69]

Programming Education

The last topical issue that can be illuminated by framing it in terms of cultures programming, which I want to discuss in this chapter, is the issue of programming education. It is easy to see how contetious the issue is today. A quick search on popular Q&A web sites finds hundreds of questions along the lines of "do I need a computer science degree to get a programmer job?" with hundreds of answers providing contradictory opinions.

Discussions about programming education are revealing, because they are indirectly pointing at the nature of knowledge about programming. Most would agree that programming education should cover fundamental ideas of the discipline, but there is a little agreement on what those fundamental ideas are. The discussion about education shows

that different cultures of programming have a very different understanding of what their discipline is. Historically, we can see this conflict in the very naming of the discipline. Those who favored the mathematical and engineering approach preferred the term *computer science* or *informatics*, business oriented programmers preferred to talk about *information systems* and *information science*, while many saw computing as just one component of a broader field of *cybernetics*.

The first computer science degrees emerged in the 1960s at the intersection of the hacker, mathematical and engineering culutres. They directly reflected the different disciplines involved in building and programming computers at the time. Programming was taught by someone from the university computing centre (typically a service organization hosting a computer for the use of the whole university), numerical analysis by a mathematician and switching theory and logic design by a person from the electrical engineering department. The focus varied depending on whether the course was based in the department of electrical engineering or mathematics. More business-oriented curricula for information systems appeared in early 1970s and included development of computer systems, courses on financial and accounting systems, but also on operations theory and social implications of computing systems.[70]

In later years, the mathematical culture grew in prominence and computer science curricula shifted emphasis to a more abstract notion of programming and algorithms, while the altnerative direction, foused on information systems and motivated by more engineering and managerial interests became secondary. The humanistic culture has, perhaps surprisingly, not been involved in the design of curricula around computing, but has been pursuing its vision as part of broader educational efforts. The LOGO programming language, which was created at MIT at the end of the 1960s, was explicitly designed for teaching, but it was not designed to teach programming. It was designed to teach "powerful ideas" such as mathematical thinking which can be experienced and studied through the use of programming and computers.

Today, the opinions on the best programming education in different cultures of programming differs as ever. A formal university education is shaped primarily by the mathematical culture of programming. A typical computer science degree aims to teach "fundamental" computer science knowledge and, for the mathematical culture, this entails topics such as algorithms, logic and formal languages. Core theoretical topics are typically complemented with material rooted in the engineering culture of programming such as development practices and methodologies. Those evolve more rapidly than mathematical knowledge, making it difficult to keep a curriculum up-to-date. The hacker culture typically contributes an odd course that involves low-level systems programming. The hacker knowledge is, however, also difficult to convey in a formal academic setting as it often takes the form of tacit knowledge learned through practice.[71] An evidence for this is the marked absence of debugging in computer science education that I revisit in Chapter 4.

The different cultures of programming thus have very different opinions on what would be the ideal model of programmer education. For the mathematical culture, mathematical theories of programming are the most basic form of knowledge and they should be taught at universities. The engineering culture values programming practices and tools which evolve more quickly. This makes nontraditional formats provided by industry, such as coding bootcamps, an appealing alternative to formal education. The hackers view formal education as even less important. If programming is a practical skill learned through practice, aspiring programmers should just start programming, learning from code and

guidance of more experienced hackers. Finally, both the managerial and the humanistic culture view computing from a broader perspective and would like to see programming education positioned in a broader context, be it business studies or arts and humanities.[72]

The contributions of the mathematical culture to the discipline of programming may well have played an important role in establishing computer science as a reputable academic discipline,[73] but it also dominated what we consider as fundamental academic computer science knowledge and, in this, alienated the academic mathematical computer science from other cultures of programming.

One place where the alienation is apparent is in job interviews. A common kind of interview questions is about algorithms, even though this knowledge is not always needed in a programming job. The situation is best understood as a cultural mismatch. Mathematical culture of programming built a solid body of fundamental knowledge. This happens to be easy to use in job interviews, even if most programming jobs require, at best, a combination of knowledge from multiple cultures of programming.

How Cultures Meet and Clash

There is much more to the history of programming than can possibly fit a single book. There have been thousands of programming languages, probably even greater number of programming tools and a wide range of programming methodologies used in innumerable projects. Those are not merely technical entities, but they exist in a broader context that often involves struggles for control, intellectual disputes, biases, stereotypes and discrimination as well as numerous other social, political and economic factors.

The aim of this book is to provide a perspective that sheds new light on important episodes from the history of programming, some of which have not been documented in detail before. As I hope to show in the upcoming 5 chapters, the perspective of cultures of programming helps us make sense of developments involving a wide and diverse set of topics ranging from programming languages, object-oriented programming and types to software engineering and interactive programming. It also helps us make sense of some of the contemporary debates about programming, including program correctness, understanding of programs and programming education discussed in this chapter.

The five cultures of programming that I identify in this book attempt to capture different basic principles, assumptions and beliefs concerning the nature and practice of programming that emerge recurringly amongst programmers. This includes different ideas about what constitutes programming knowledge and how to best acquire it, beliefs about what kind of activity programming is and also what it includes in addition to instructing the machine. The five cultures provide meaningful explanation of the history I look at, but they are inevitably simplifications and different accounts of the history of programming may need different or a more fine-grained structure.[74] Yet, I believe that recasting the history of programming as interactions between five different cultures of programming that I attempt in this book provides as good overarching narrative for the history as possible. To borrow a programming term, cultures of programming are a useful abstraction.

Although a reader who is a programmer or a computer scientist might recognize themselves (or their colleagues) in one of the characters in the opening dialogues in this book, an individual does not have to strictly belong to a single culture of programming and accept its assumptions unequivocally. In reality, may of those who contributed to the development of programming combine traits from multiple cultures of programming and change

their views over time. Despite their different basic assumptions, proponents of the different cultures of programming do not "live in different worlds."[75] They can understand each other and exchange ideas, although they may not agree about the foundations behind such ideas and their importance. Furthermore, the different cultures also share a common ground in the form of program code and concrete software artifacts.[76]

The key idea that I put forward in this book is that the most interesting developments in the history of programming over the last 70 years happen happen when two or more cultures interact. The next five chapters provide a wealth of evidence. Throughout the book, we will repeatedly encounter two kinds of interactions.

On the one hand, different cultures of programming often clash about the basic principles of and assumptions about programming. The controversies around program verification (Chapter 2) arise when the cultures clash over what program correctness means. The history of interactive computing (Chapter 3) is a struggle between cultures that is centered around the way in which a human is involved in programming. Finally, in the software engineering debate (Chapter 4), the cultures disagree about the best avenue towards producing software reliably and on budget.

On the other hand, different cultures of programming often contribute to the development of a shared technical artifact, including the very idea of a programming language (Chapter 2). The notion of a type (Chapter 5), which appears in many programming languages today, takes a shape when the hacker, engineering and mathematical cultures meet. The concept of testing (Chapter 4) appears as a hacker method, but is refined by the proponents of engineering and managerial cultures. Similarly, the idea of objects and object-oriented programming (Chapter 6) is first developed in humanistic and hacker cultures, but it later evolves with engineering focus in mind and is further adapted by the managerial culture.

In the 1950s, the existence of different cultures in computing and programming could have been explained by the fact that the field was emerging at the intersection of electrical engineering, mathematics and logic, military and business, psychology and many other disciplines. Seventy years later, the existence of the same cultures of programmin is a sign that programming is and will remain an inherently pluralistic discipline. The historical episodes discussed in this book show how programming has benefited from this structure. The clashes over basic principles eventually deepen our understanding of the nature of programming. Ideas from other cultures often reinvigorate concepts that have been developed in other cultures and remained stale. The existence of multiple cultures also keeps a greater number of approaches that programmers, as a community, have ready at hand for tackling technical challenges that emerge as the field of computing evolves.[77]

Although the main focus of this book is historical, there is also a forward looking aspect to this work. To those contemplating the future of programming, the book might point at new, yet unexplored, possibilities that can appear by viewing an existing technical ideas from one culture through the perspectives of other cultures.

Notes

1. As pointed out by Neff and Nagy (2016), chatbots ranging from Weizenbaum's 1966 ELIZA to Microsoft's 2016 Tay are often used in ways that their designers did not expect and their position between society and technology raises difficult questions about accountability and agency.
2. The struggle for control in programming is discussed by Ensmenger (2012) and the ways in which early

programming in the UK, US and the later emergence of "software engineering" led to masculinization of the discipline are discussed by Hicks (2010), Ensmenger (2010) and Abbate (2012), respectively.

3. In one case, documented by Tomayko (1988), the onboard guidance computer and the ground control computer calculated the time for the de-orbit burn differently and the crew had to manually key the numbers transmitted from the ground control into the Apollo onboard computer.
4. Klein et al. (2018) provides an overview of some of the formal verification efforts ranging from formally verified compilers to the AH-6. We return to the topic of what exactly formal verification means in these cases in Chapter 2.
5. As we discuss in Chapter 3, the term 'hacker' used here refers to a programming sub-culture documented by Levy (2010) and Tozzi (2017), rather than to security hackers who break into computer systems; a discussion of a broader context, including some of the overlaps between the two can be found in work on piracy by Johns (2010)
6. See Dastin (2018) for this particular case. The general issue of algorithmic bias has become a widely recognized issue and is discussed, for example, in a popular account by O'Neil (2016). An important case of how search engines reinforce racism is discussed by Noble (2018).
7. The simplistic position that "algorithmic bias is a data problem" is indefensible, even if we take a very technical perspective on the nature of AI algorithms as shown by Hooker (2021).
8. `https://dictionary.cambridge.org/dictionary/english/algorithm`, Retrieved 4 May 2022
9. Kusner et al. (2017)
10. Abadi and Cardelli (1996)
11. This has been pointed out by MacKenzie (2014). More generally, computer bugs play the role of "Heideggerian hammer" in that it forces us to examine the role of a system at a more basic level.
12. Murhpy (2013)
13. As pointed out by Mahoney (2005), "most declarations of the 'computer revolution' have rested on future promises rather than on present or past performance."
14. `https://www.cs.utexas.edu/users/moore/acl2/workshop-2017/slides-accepted/Passmore-AI-ACL2-2017-export.pdf`, Retrieved 17 June 2022
15. Mahoney (2005)
16. `https://www.gartner.com/it-glossary/devops`, Retrieved 30 April 2022
17. `https://dougseven.com/2014/04/17/knightmare-a-devops-cautionary-tale`, Retrieved 30 April 2022
18. `https://www.sec.gov/rules/final/2010/34-63241.pdf`, Retrieved 30 April 2022
19. `https://www.sec.gov/litigation/admin/2013/34-70694.pdf`, Retrieved 30 April 2022
20. The technical concepts play the role of boundary objects, as introduced by Star and Griesemer (1989). They often have concrete technical implementations that have enough shared content, but they are also flexible enough to be used and interpreted differently by different cultures.
21. Ensmenger (2012)
22. Priestley (2011) documents the technical developments leading from ENIAC to programming languages, whereas Ensmenger (2012) documents the social developments of computer science. Mahoney (1997) follows, more specifically, mathematical theories that contributed to computer science.
23. `https://sel4.systems/Info/FAQ/proof.pml`, Retrieved 21 June, 2022
24. Levy (2010); Raymond (1997)
25. The contrast is discussed by Adam (2005); hacking likely follows the pattern of software engineering, documented by Abbate (2012), which emerged as a more masculine redefinition of programming which remained, in its dominant office worker variant, somewhat accessible to women.
26. Beeler et al. (1972)
27. Turner (2010) tells the history of West coast cyberculture, which combines the hacker culture of the MIT with the West coast counterculture movement.
28. Raymond (1997), quoted by Tozzi (2017)
29. The history of UNIX is briefly discussed in Raymond (2003) and a more detailed historical account of free software, starting from the UNIX hackers, has been written by Tozzi (2017)
30. As pointed out by Slayton (2013), they saw programming as more flexible than traditional phyiscal electronics and so presumably easier. Programmability was also advertised as an advantage of the system to administrators and policymakers, disincentivizing the recognition of the difficulty of programming.
31. Slayton (2013); Ensmenger (2012) talks about the "labor crisis" in programming.
32. Redmond and Smith (2000)

33. Ensmenger (2012) views this through the perspective of struggle for control between the managers and programmers.
34. Quoted by Tomayko (1988). For a more recent detailed historical account, see Mindell (2011)
35. Tomayko (1988), p.41
36. John Backus, inventor of FORTRAN, quoted by Ensmenger (2012).
37. Ensmenger (2012)
38. As noted by Thomas Haigh (2010), the conference is sometimes seen as a crucial turning point by historians of computing, but its actual impact is becoming "harder to square with the actual historical record." Nevertheless, the proceedings of the 1968 and the follow-up 1969 conferences (Naur et al. (1969), Buxton et al. (1970)) serve as a good account of thinking about programming, in a particular community, at the time.
39. As documented by Thomas Haigh (2010), some of the new work was, at least in part, a response to the failure of the work on Algol 68 in the mathematical culture of programming.
40. There is, however, a difference between professionalism arising from the needs of software engineers and professionalism imopsed from the outside. For example, many of the attempts to develop certification schemes for programmers in the 1960s documented by Ensmenger (2012) seem more aligned with the managerial culture and the same would be the case with many contemporary certifications.
41. The "Papers We Love" movement (`https://paperswelove.org`, Retrieved 24 June 2022) is a recent example exhibiting this characteristic of the engineering culture.
42. In his famous "Go to statement considered harmful" letter (Dijkstra (1968)), but also in a working paper on "structured programming" that appeared in the NATO 1969 conference proceedings.
43. The emphasis on code that programmers can understand in Dijkstra (1968) is remarkably long-lived. For example, recently published book by Seemann (2021) has the idea of "code that fits your head" in its very title.
44. The obvious place to look for such work would be the IFIP WG 2.3 on Programming Methodology, established in 1969. However, as the report by Gries (1978a) indicates, most work in the group focused either on structuring of code and data or on topics such as program correctness that are more aligned with the mathematical culture. Two exceptions from this are Niklaus Wirth's contribution on "Program development by stepwise refinement" and Douglas T. Ross' contribution on "Structured analysis".
45. Bush (1945)
46. Sutherland (1963)
47. Solomon et al. (2020)
48. Kay and Goldberg (1977)
49. The principle was captured by the slogan "Obscurantism is dangerous. Show us your screens." formulated by Ward et al. (2004) of TOPLAP, an organization founded in 2004 to promote live coding.
50. As documented by many, including Misa (2011); Abbate (2012); Hicks (2017), pioneering contributions made by women often remain hidden from history, so the notable fact here is perhaps not that there are women contributors, but that their contributions have been recognized. One should not be overly optimistic though as two of the four found career in programming only after working as secretaries.
51. As the well-researched biography of Vannevar Bush by Zachary (2018) shows, his reservations towards humanities make Bush an unlikely contributor to the humanistic culture of programming.
52. The approach of developing an overarching grand narrative of computing history has recently been followed by Haigh and Ceruzzi (2021). The present book is less ambitious and focuses specifically on programming, but it shares the ambition of providing a comprehensible story that explains many developments throughout the history.
53. This is pointed out by (Priestley, 2011, p.254). At the time, "programming" was seen as the mathematical design of the program and "coding" as a translation of the design to machine language; the belief was that coding errors would become infrequent.
54. Blackwell and Collins (2005)
55. Dijkstra (1993)
56. Kleiner (2010)
57. This is not unlike the case of competing interpretations of a scientific experiment. As documented by Chang (2012), in the early 19th century, some saw electrolysis of water as a process producing phlogisticated and dephlogisticated water, while others viewed it as a process splitting water into Hydrogen and Oxygen. In case of chemistry, however, one view eventually dominated.
58. An approach that Chang (2012) advocates for in his work on history and philosophy of science.
59. `https://www.rferl.org/a/27468516.html`, Retrieved 2 July 2022

60. Dastin (2018)
61. O'Neil (2016)
62. Noble (2018)
63. Simondon (2016)
64. An example of this approach is the ImageNet Roulette project by Crawford and Paglen (2019), which uses an AI algorithm to assign, often problematic, categories to a person based on their uploaded photo.
65. The Mu project by Agaram (2020) is an extreme example of this approach in that it attempts to build a system that is comprehensible, starting from low-level machine code.
66. The Free Software Foundation (2022)
67. For example, see the privacy-focused smartphone operating system /e/OS: `https://e.foundation`, Retrieved 3 July 2022
68. MacKenzie (2004)
69. For example, the work by Cihon (2019) at the Oxford Future of Humanity Institute reviews ongoing work on and argues for such standards.
70. Atchison (1985) provides a review of early computer science education; an early curriculum for "information systems" degree is proposed by Ashenhurst (1972)
71. Using the notion of "tacit knowledge" as introduced by Polanyi (1958)
72. A good example of how such rethinking of computing from the perspective of arts and humanities may look is the recent work by Sack (2019).
73. The rise to prominence of the notion of an algorithm and its contribution to the growth of computer science is documented by Ensmenger (2012); Mahoney (1997, 1992) provides a detailed account of the history of computer science and the evolution of the conceptualizations of the discipline.
74. For example, Mahoney (1988) talks about the tripartite nature of computing consisting of electrical engineering, computer science and software engineering, which provides an account of computing as a whole. In later work, Mahoney (2005) discusses more fine-grained communities of computing including data processing, management, mathematical calculation, mathematical logic, human augmentation, artificiall intelligence and computational science. Each of those comunities would predominantly follow beliefs of one particular culture of programming, in the sense used in this book.
75. In the classical sense of belonging to different research paradigm as introduced by Kuhn (1962). The notion of culture of programming is perhaps closer to that of "system of practice", developed by Chang (2012) to talk about the history of chemistry.
76. We might see concrete programming languages and tools as trading zones through which cultures can exchange ideas, even if they interpret them differently. A prime example of this is the notion of "type" discussed in Chapter 5, which is used by many cultures, but in subtly different ways.
77. Much has been written about pluralims in physics. Galison (1997) documented the two sub-cultures of particle physics and pointed out that the structure makes the discipline more stable, perhaps in a similar way in which the different cultures of programming contribute to its stability over time. Chang (2012) talks about different "systems of practice" in the early history of chemistry and uses the account as an argument for greater pluralism in scince, a call that programming in many ways, but sometimes unconsciously, already follows.

Chapter 2

Mathematization of Programming

Teacher: This chapter follows one of the developments through which programming established itself as a professional respectable discipline. To open the discussion, where in the history of programming do we find the first steps towards the professionalization of programming?

Tau: I have serious doubts whether programming is a respectable discipline even today, given that we are still unable to produce correct programs. But to give a more serious answer, I will again say that the turning point from which we can even think about making programming respectable is the publication of the preliminary report on the Algol programming language in 1958.

Alpha: The proliferation of programming languages at the end of the 1950s was certainly an advance, but it was a gradual progression. We were slowly learning how to program better. What was so wrong with programming in the 1950s that suddenly changed?

Tau: In the 1950s, programming lacked the sound body of knowledge that could support it as an intellectually respectable discipline![1] Treating programs as mathematical entities, which was the intellectual achievement of Algol, made it possible to start building such knowledge in the form of fundamental algorithms...[2]

Epsilon: Well, I'm not sure if algorithms are necessarily the right kind of fundamental knowledge, but I would agree that the problem with programming in the 1950s was that it was completely unsystematic. There were no accepted development methodologies, standard tools and processes, so programming was highly unreliable.

Omega: I agree about the unreliability of programming back then, except that it was not due to a lack of algorithms or development methodologies, but a personnel issue. The industry was growing too rapidly and there was a shortage of qualified programmers.[3]

Gamma: It is ironic that the attempts to "professionalize programming" at the time of a labor crisis can also be linked to the exclusion of women from the workforce! Many of the professionalization efforts tried to turn the perception of programming from a low-skilled clerical work, associated with women, to a high-skilled masculine work. What programmers actually did has not changed much. The perception did.[4]

Teacher: I do not want to brush aside an issue that is of utmost importance today, but let's get back to programming languages for a minute. Would you agree that the birth of programming languages, and specifically Algol, was a notable milestone in the professionalization of programming?

Omega: There is an interesting shift associated with the birth of programming languages, which is that programming languages became entities separate from the specific machines. This did not solve all the problems of the 1950s crisis, but it partly solved the portability problem that many in the business of programming at the time were facing. With a bit of luck, you could take your programs in the new programming languages, compile them and run them on a new, more powerful, machine without having to start coding from scratch.

Alpha: You all keep talking as if programming languages were some sudden new invention at the end of the 1950s, but there were many systems that preceded the better known programming languages. There was the A-0 system by Grace Hopper and Autocode by Alick Glennie, both created in 1952. Already at the end of the 1940s, the EDSAC programmers used an "external form" of instructions and interpretive routines, which are early tricks resembling later programming language ideas![5]

Tau: If you look at the work on mathematical logic, you can trace the history of programming languages even further back. The work on lambda calculus, Turing machines and Post production systems in the 1930s pretty much captured the essence of later functional, imperative and concurrent programming languages! The definition of Algol also went hand-in-hand with mathematical work on formal language theory by Chomsky and others.[6]

Gamma: This may be a nice story computer scientists like to tell themselves, but you cannot talk about programming languages before there were any actual physical computers to be programmed! You are treating logical theories as something very different from what they were at the time.[7]

Tau: Do you want to deny that mathematical ideas on formal languages contributed to the first computer programming languages?

Gamma: I have serious doubts about the direct influences of work on mathematical logic, but work on formal languages was soon adopted by computer scientists working on programming languages. In fact, it is interesting to see how the idea of a programming language illustrates the point that we made in the previous chapter. It is a single programming concept that we all share that combines ideas coming from different backgrounds. It brings together theory of formal languages that *Tau* talked about, implementation techniques from earlier systems mentioned by *Alpha* and concrete business motivation that *Omega* found important.

Teacher: That is a nice observation. But do we also have a shared understanding of what programming languages are?

Gamma: I doubt that. For me, programming languages are a medium for thinking, but this way of thinking about programming languages is arguably more recent. The 1950s perspective that I find the most appealing is to think of programming languages and their compilers as translators from a language that a human understands to a language that a machine understands.[8]

Omega: I do not think this is exactly wrong, but it is not a very interesting perspective. The first programming systems in the 1950s were also called "automatic coding" systems and they tried to make programming easier, ideally so that it does not have to be done by skilled programmers. It makes much more sense to see programming langauges as one of the approaches to tackling the labor crisis of the 1950s.[9]

Tau: Sure, programming languages involve translation and, sure, they make programming practically easier. But what makes them interesting is that they allow a new mathematical thinking about programming. They make it possible to treat programs as mathematical entities and formally analyze them using the devices of formal logic.

Teacher: Let's focus on this idea of treating programs as mathematical entities for a moment. What exactly does this mean?

Tau: If you treat computer programs as syntactic structures in a formal language defined by the rules of logic, then you can infer properties of programs from their texts by purely deductive methods. In other words, you can write a program and then prove that it is correct.

Epsilon: That sounds nice, but how do we even know what a correct program is?

Tau: You need to start with a formal specification that defines the correct behaviour!

Omega: I'm not convinced. A specification is a document intended primarily for humans. It gives enough information to a human reader, but I doubt you could use any real-world specification in mathematical proofs. That would make writing the specification as hard as doing the programming itself…

Alpha: I also wonder how do you even make sure that your compiler does what your mathematical model says it does. You make it sound very easy to specify what a program means, but real programming languages and compilers are enormously complicated. After all, one of the criticisms of Algol was that it was very hard to implement properly.

Tau: This is something that people started soon working on. Since the 1960s, there were several efforts to define the Algol language itself fully formally, so that you could prove that a compiler correctly implements the specification and, consequently, also that programs have no errors. It was a visionary idea at the time, but the CompCert project, created in the 2000s, does exactly this for the C programming language and has been used commercially in mission-critical systems.

Alpha: So, it took just 60 years to do this and only for a single programming language?

Tau: The programming practice had to evolve in order to make such projects possible. We had to figure out how to write programs in a more compositional way, so that a correct program can be composed from individual correct components.

Teacher: I expect that we can relate the ideas on better ways of organizing programs to our more general discussion of professionalization of programming. What were some of the first concrete ideas on how to write programs better?

Epsilon: One practical idea that was broadly adopted in the 1960s was that of structured programming. It means organizing code and data using logical structures with meaning, such as loops and conditionals for control-flow and records and collections for representing data.

Gamma: Loops and conditionals are clearly useful, but this way of structuring programs also makes it harder to interact with the programming environment in certain ways. Commodore 64 BASIC has `GOTO`, but it also uses the line-based program structure to allow a very simple way of editing code.[10]

Tau: I do not understand why you would want to have `GOTO` in a programming language! The Böhm-Jacopini theorem from 1966[11] clearly proved that any program containing `GOTO` can be transformed to an equivalent structured program…

Gamma: This is where the mathematical perspective leads you. You are thinking only about programs as syntactic entities, but not about how programs are written or, as was the case of BASIC programs, printed on paper and shared through magazines!

Omega: The idea of structured programming was neat, but for a different reason. It made it possible to manage large development efforts in a top-down way. It inspired a new managerial approach to programming where a skilled chief programmer draws up the overall system design and divides it into modules that are then assigned to junior programmers. This led to a successful development methodology adopted by IBM and others in the 1970s.[12]

Epsilon: The whole point of structured programming is to allow individual programmers to write more readable programs and be more confident in their correctness. Treating structured programming as a managerial practice that turns programmers into unskilled workers goes directly against this idea!

Omega: I do not see why you are so upset. Didn't we finally find a shared concept that we both find useful?

Epsilon: If you want to call top-down management "structured programming", so be it. I'll find another term for what I actually talk about.[13]

Teacher: Let's get back to the issue of mathematical proofs of computer programs. It seems that the basic ideas necessary for this way of thinking were around at least since the 1960s, yet proving programs correct is something that programmers rarely do today. Do you have any insights into why this is the case?

Tau: This is just a matter of poor economic decisions. Proving programs correct increases the initial costs, so people do not do it. They foolishly ignore the fact that eliminating program errors would save them money in the long run…

Epsilon: I'm not entirely sure if proofs about computer programs are really the same kind of thing as ordinary mathematical proofs. For one thing, mathematical definitions tend to be quite simple and formal specifications of programs can be pretty lenghty. Good mathematical proofs often involve ingenious tricks, but proofs about programs are just tedious.

Gamma: What you are saying is pretty much exactly the objection raised by DeMillo, Lipton and Perlis in 1977.[14] They pointed out that correctness of mathematical proofs is ensured by the social processes that surround them, such as mathematicians excitedly discussing new elegant proofs at the blackboard during a seminar or even a coffee break. Proofs about computer programs are too long and boring to support similar social processes and so they cannot be trusted.

Epsilon: This is an interesting problem, but I would not throw proofs of programs out the window yet. Perhaps there is some other way of building trust in program proofs…

Omega: You have to realize that you are not trying to produce correct programs for some intellectual enjoyment. If a company decides it is economically worth it, it can simply demand that the programmers write and carefully review the programs. You can set a target small error rate and adjust your development methodology to guarantee that.[15]

Tau: I do not understand why you would accept "small error rate" when the purpose of mathematical proofs is to give you an absolute certainty.

Epsilon: I thought we agreed that program proofs are not like mathematical proofs and cannot be trusted to the same degree.

Tau: I do not understand this objection. If you have a proof where each step is derived from the assumptions and the previous steps using an inference rule, then it is a proof. It does not matter who is talking about it and how! In fact, you can even automate proof checking and write a computer program that will check that each step of a formal proof is correct. That way, there can be absolutely no doubt about its correctness.

Teacher: It is worth noting that DeMillo et al. also talk about automatic verifiers and they fear what they call the Titanic effect. A failure is still possible, but because of our uncritical trust in the system, it would have catastrophic consequences.

Gamma: There is a more fundamental issue with proofs of programs that philosophers know as a category mistake. The issue is that proofs refer to mathematical a priori knowledge, but programs refer to empirical a posteriori knowledge. What you can prove about programs as formal syntactic entities is distinct from the actual effects that programs have in the real world.

Epsilon: I see what you're saying, but I do not think this is a problem. We are simply moving from the realm of mathematical reasoning to the realm of software engineering. An automatic proof checker is also a program, existing in the empirical world, but if it is correctly engineered then I'm happy to trust what it says. The issue that still remains today is that this is just extremely hard and time-consuming to do for any real system.

Tau: I also think you are underestimating the usefulness of small elegant mathematical proofs about programs. If you study simple formal models of programming languages, you may not be able to prove that the actual implementation is correct, but you will often find where the tricky corner cases are, so that you know where you need to pay special attention during the implementation.

Teacher: It seems to me that the idea of using mathematical methods for thinking about programming is a firmly established one today, so this is certainly something that has changed since the 1950s. However, we still have very different ideas about how exactly should mathematical methods be used. This is quite remarkable given that mathematical knowledge appears as something that does not admit multiple interpretations! This plurality of perspectives has been practically useful, but it seems that we are bound to clash once the discussion turns more philosophical.

Mathematization of Programming

Giant Electronic Brains

On February 15, 1946, the recently built electronic general-purpose computer, ENIAC, completed a number of calculations during a public demonstration where it was unveiled in front of some 100 prominent guests. One of the example programs was a calculation of a trajectory of a shell, which it completed in twenty seconds. The same task would take female computers about forty hours to complete using a mechanical desktop calculator, which is how the military made those calculations before.

As the ENIAC computed the trajectory, the attendees could literaly see the numbers being processed on a front panel of the machine. The panel showed the contents of ENIAC accummulators using a grid of faint neon lights. For the demonstration, the inventors of ENIAC, John Mauchly and J. Presper Eckert placed halves of Ping-Pong balls over the lights and painted them with numbers each light represented. When one of the engineers turned off the lights in the room during the actual computation, the result was a captivating show of flashing lights produced by a machine that was soon nicknamed a "giant brain" by the press.

Although the audience for the demonstration was almost exclusively male, the programmers who setup the machine to undertake the calculation were mostly female. One of them, Betty Jean Jennings (later Jean Bartik) recalls how she and a collague, Betty Snyder were busy making the last corrections on the day before the demonstration.[16] Last-minute debugging was apparently as common in 1940s as it is 80 years later. According to Jennings the machine was calculating the trajectory correctly, except that it did not stop when the shell hit the ground. When the programmers left around midnight to catch the last train home, the calculation was still buggy. By the morning, Betty Snyder figured out what was wrong. The program was doing one additional iteration after testing for collision with the ground. She flipped one of the three thousand ENIAC switches to fix this and the calculation was ready for the presentation.

The ENIAC was initially designed and built to calculate artillery firing tables for the United States Army. But even before it was publicly unveiled in 1946, it was programmed to perform scientific simulations of neutrons passing through various materials to assist with the design of the hydrogen bomb. This classified work was going on, in parallel with firing trajectory calculations, at the time of the public demonstration of the machine.

Most of the non-classified programming of the ENIAC was done by a group of six women (Figure 2.7), initially referred to as *operators*. They were recruited from a group of human computers hired to calculate ballistics tables by hand, using mechanical desk calculators and a differential analyzer. Computer operator was, at least initially, seen as low status job compared to engineers working on the electronic and mechanical aspects of the computer. This, together with the ongoing labor shortage resulting from the on-

Figure 2.1: The first ENIAC programmers, US Army/ARL Technical Library Archives

going war effort, were also the reasons why women were, at least initially, hired for the work. Despite the low-status of the computer operators, programming the ENIAC required ingenuity and posed at least as many problems as the engineering work.

The ENIAC was, as Betty Jean Jennings eloquently put it, “son of a bitch to program”. It was, at least initially, programmed by setting switches and plugging wires to connect individual components of the machine. Those included *accumulators*, which were used for addition, subtraction and number storage; specialized *divider and square rooter*, *multiplier* for multiplication and *master programmer* that could be used to control looping and branching. The components operated in parallel and had to be synchronized by connecting cables from pulse outputs (indicating the completion of an operation) to pulse inputs (to start the next operation). This was further complicated by the fact that different components took different amount of time to complete their work. For some, such as the divider, the time dependend on the values with which they computed. Programming the ENIAC was more like constructing a new elaborate scientific equipment than like programming as we know it today.

The first computers also suffered from frequent mechanical or electronic failures. During the first runs of the ENIAC, the engineers were continually fxing the machine, replacing blown vacuum tubes, resolving short circuits and correcting other failures. But as the story told by Betty Jean Jennings shows, programmers of the first electronic computers also had to cope with some of the same difficulties as programmers do today. To make the matters worse, early computers like ENIAC were idiosyncratic and had very limited computing power. The limited computing power meant that programmers had to invent clever tricks to make computers do even the most basic calculations. This included both technical tricks, to overcome machine limitations, and mathematical tricks, to implement calculations more efficiently. At the same time, each machine was different and so each required very different kinds of programming tricks. The programming tricks were rarely documented and became a part of each programmers personal repertoire. This unwritten knowledge later contributed to their reputation for being “geniuses and mavericks”[17]. In

a later recollection of John Backus, creator of the FORTRAN programming langauge, "Programming in the 1950s was a black art, a private arcane matter."[18]

Over time, the mechanical and electronic failures became less frequent. In the case of ENIAC, the engineers realised that vacuum tubes often blow because of power surge created when the machine was turned on. By keeping it always on, they managed to reduce the failure rate to one tube every couple of days. Many believed that programming errors will, over time, become as infrequent as hardware errors.[19] The industry needed almost a decade to fully realise that this was not going to be the case. One of the first computer pioneers who understood that programming is going to be a major concern was Maurice Wilkes, who designed and helped build the EDSAC computer at University of Cambridge in 1949. Wilkes recalls that, while working on a programming challenge "the realization came over me with full force that a good part of the remainder of my life was going to be spent in finding errors in my own programs."[20] Maurice Wilkes was ahead of the curve with his realisation and most others only realized the importance of programming when computers entered the commercial sector. This started almost as soon as the ENIAC was built.

Following a dispute with the Moore School of Electrical Engineering about patent rights, the two ENIAC creators J. Presper Eckert and John Mauchly resigned from their university positions and formed the Eckert–Mauchly Computer Corporation. They hired three of the original six ENIAC programmers and started building commercial computers designed for business applications. Their first client was the U.S. Census Bureau, but later installations included a number of manufacturing and insurance companies, as well as other government departments.

As computers left the university environment, companies needed to recruit personnel to program the machines. This was well before the role of a programmer had any clear meaning. The early programmers came from a variety of backgrounds including engineering, physics and mathematics. The growth of the computing industry expanded the range of people who were involved with computers even further to include statisticians, managers and salespeople. All companies that were entering the computing industry faced the challenge of how to find programmers. One recommendation was to look for creative people who like intellectual challenge like chess players or mathematical puzzle solvers,[21] but the reality was that nobody knew what a programmer looks like.

Not only it was hard to find programmers, it was also difficult to train them. Much of the knowledge that early programmers needed to have was intuitive and local. It consisted of tricks that worked on specific machines, anecdotes and rules of thumb. It was also difficult to generalize to be more widely applicable. This kind of knowledge is representative of what I call the *hacker culture* of programming. Hacker knowledge is difficult to write down as it relies on hunches and experience with a large number of past situations. It is acquired through practice or by working with those who already have the skills. The way hackers learn their craft is not through any formal education, but through practice and apprenticeships. The hacker knowledge is what Michael Polanyi calls "tacit knowledge"[22] and represents things that we know how to do, but that we cannot explain in writing.

Sound Body of Knowledge

Programming as a discipline that relies on tacit knowledge of the hacker culture was good enough in the early days of digital computers when there were only a few distinctly unique machines. As the comptuer programming sector started growing and diversifying in the

1950s, programming increasingly became the key limiting factor. The growth of the electronic data processing industry in the early 1950s led to an imminent shortage of programmers. Throughout the 1950s, the shortage developed into an industry crisis referred to as "programming gap".[23] The gap was widely reported and discussed in industry journals. It was certainly due to the novelty of the problems and the rapid growth of the industry, but the nature of the knowledge needed by the early programmers may have also played its part. Tacit knowledge that a programmer in the 1940s and 1950s had to master is slow to acquire and it cannot be easily distilled into textbooks or training courses.

The ongoing industry crisis made many people involved with computers ponder how to make programming into something else than a practical skill learned through practice. The variety of ideas about what programming should look like serve as an early example illustrating the differences between several cultures of programming.

The practitioners started calling for the profesionalization of programming. For managers, the key concern was making programming less reliable on unique skills of individual programmers. For the programmers themselves, the key concern was career paths and being recognized as good programmers. The topics discussed in the practically-oriented computer magazine Datamation, launched in 1957, demonstrate all those interests. They include novel equipment for data management, articles on programming education and training, machine compatibility and management techniques. At the same time, the professional organizations bringing together practioners in the field of data processing focused primarily on the development of professional certifications such as Certificate in Data Processing, which originated in 1960.[24]

The fact that programmers relied so much on tacit hacker knowledge was equally alarming to those with academic aspirations, but their concerns were different than those of industry practitioners. Edsger Dijkstra who was a theoretical physicist by training, but started programming in 1952. He later recalled his dissatisfaction when he realised, some three years after he began working as a programmer, that programming lacked "the sound body of knowledge that could support it as an intellectually respectable discipline."[25] For those with academic interests, this was not only intellectually dissatisfying, but it also threatened programming to remain a low-status profession.

In the academic circles, many of those interested in computers joined the Association of Computing Machinery (ACM) after it was established in 1947. The ACM focused on computer science as an academic discipline as opposed to computer programming done by practitioners. Its members believed that a rigorous approach to programming should be modeled after applied mathematics.

The focus point of this effort eventually became the notion of an *algorithm*,[26] which is a "finite set of rules that gives a sequence of operations for solving specific type of problem".[27] The concept of an algorithm gave computer scientists a practical agenda with many interesting unsolved puzzles. The Communications of ACM journal that was established in 1958 dedicated a large part of its content to discussion of algorithms. A prime example of a work done within this newly created mathematical paradigm of computer science became the book series "The Art of Computer Programming" by Donald Knuth[28] who set to document the most fundamental algorithms. As of today, the first three and a half volumes of this ever growing monograph have been published, consisting of over 3,000 pages.

The developments are characteristic of the three cultures of programming that we encountered so far. The programmers belonging to the *hacker culture* keep mastering their craft and inventing new tricks to make programming more manageable. Those with

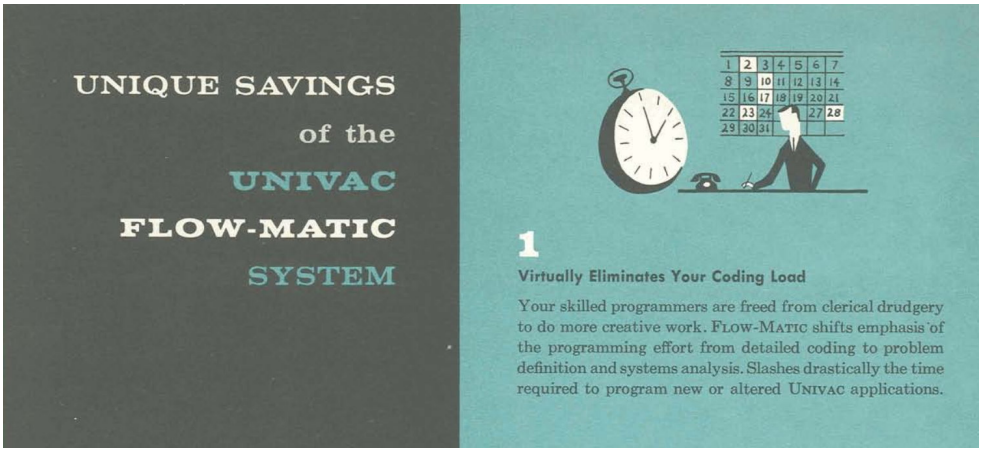


Figure 2.2: A page from a brochure "Introducing a New Language for Automatic Programming Univac Flow-Matic", Sperry Rand Corporation, 1957.[31]

commercial focus that I associate with the *managerial culture* want to make programming more predictable, so that software can be built according to a specification and within a fixed budget. Those with academic interests that I refer to as the *mathematical culture* focused on mathematical analysis of algorithms, which allowed them to establish computer science as a reputable discipline in the modern university environment that values theory higher than practice.[29] The developments also show how the different cultures begin to diverge and clash.[30] In particular, the focus on abstract algorithms distanced academic computer science from the concerns of commercial programmers, while the managerial attempts to make programming more structured goes directly against the interest of the hacker culture programmers.

Given their divergent interests, it seems unlikely that the different cultures could all contribute to a single programming invention. Yet, this is exactly what happened and it revolves around what is quite possibly the greatest invention in programming of the 1950s.

How Technology Became Language

Programming computers by plugging wires between components, as done in the early days of the ENIAC, fortunately did not stay around for long. Most later machines were instead programmed through numerical codes that were entered by flipping switches and using punched cards. However, each machine had its own encoding with different numerical codes, as well as its own idiosyncratic limitations. This meant that programs were created for just a single machine.

During the 1950s, programmers realized that they can use the computer itself to make programming easier.[32] The Short Code system, built in 1950 for the UNIVAC I computer, allowed programmers to specify programs in a more human-readable, although still numerical, code that resembled infix mathematical expressions. The computer run a separate program, Short Code interpreter, which read such instructions and performed the specified computations. A more efficient approach implemented by other systems within a few years was to create a program that would read such human-readable code and translate it to the code used by the machine, which would then be executed directly. A computer scientist will recognize these two methods as predecessors of modern interpreters and

compilers. The approach of writing programs using more expressive pseudo-instructions and translating those to the machine language became known as *automatic coding*. As illustrated by the FLOW-MATIC promotional brochure (Figure 2.2) it promised to "virtually eliminate your coding load".

A popular description of programming at the time was that the programmer had to come up with a program and then translate it into a language the machine understood. This used the anthropomorphic metaphor of computers as giant electronic brains, that was introduced by the media and further popularized by the ongoing developments in cybernetics. The metaphor made it natural to describe programming as a translation from human language into a machine language. The metaphor was not used just by popular media. A computer scientist, Grace Hopper, illustrated her 1954 programming course at MIT with a drawing (Figure 2.3) that describes the A-2 programming system, a predecessor to FLOW-MATIC, as a robot translating between two languages.

During the second half of the 1950s, the notations used by automatic coding systems became known as *programming languages*. The term also "lost its anthropomorphic connotation and acquired a more abstract meaning, closely related to the formal languages of logic and linguistics."[33]. The FORTRAN programming language illustrates this gradual shift. The "Specifications for the IBM Mathematical FORmula TRANslating System FORTRAN", published in 1954, did not use the term programming language yet, but it already consistently referred to the notation as the "FORTRAN language".[34]

The most revolutionary aspect of the development was that programming languages became standalone objects, independent of hardware that supported them, and began to be studied on their own. The FORTRAN language is, again, a witness of this development. It was first built for the IBM 704 machine in 1957, but by 1958, IBM was planning to make FORTRAN available for other machines. The early versions of FORTRAN for other machines were not fully compatible with each other, but it became possible to think that they should be.

At least three different cultures of programming played their role in the birth of the idea of a programming language. The managerial culture provided an important commercial motivation for detaching programs from individual machines. A programming language makes this, at least in principle, possible and "the early users of the term *programming language* were principally computer-user groups and computer-installation managers attempting to bring about the cross-machine compatibility of programs; common or universal notations would facilitate the exchange of programs among and within organizations, and would also provide a suitable vehicle for teaching programming in universities."[36] Technically, programming languages were the next step of the development that started with automatic coding systems such as Short Code, A-2 and FLOW-MATIC. The "black art" programming skills and arcane tricks of the hacker culture of programming were behind the implementations of the interpreters or compilers for the emerging programming languages.

The work on machine-independent programming languages was proceeding on two fronts. The largest computer-user group, SHARE, established a committee to study universal programming languages. The committee failed to make a specific language proposal and some members felt this problem is a "bucket of worms"[37]. The work was complicated by practical challenges, such as machine differences, but it may also have been hindered by the lack of suitable tools for defining a programming language.

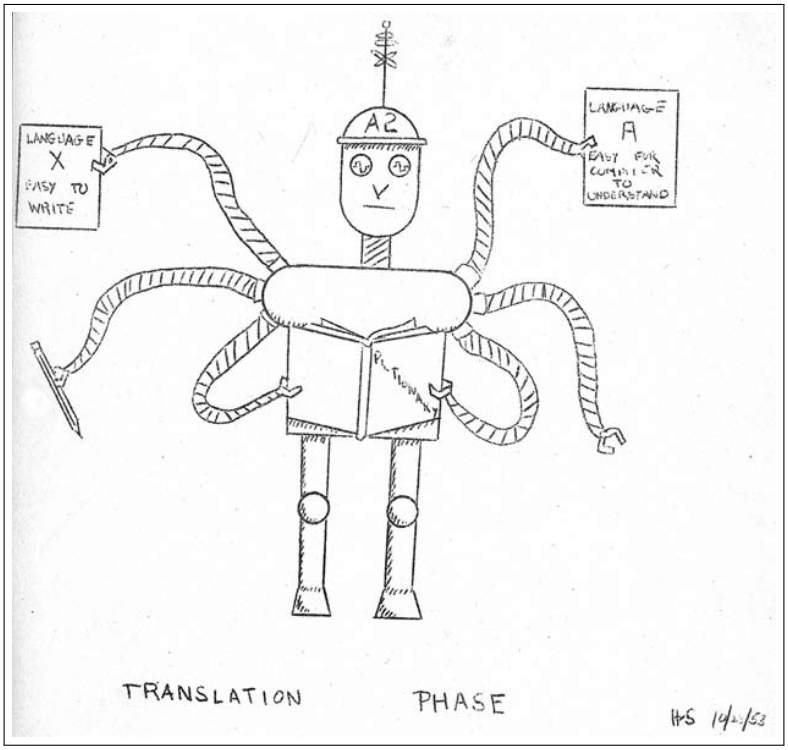


Figure 2.3: An illustration by Grace Hopper that describes the functioning of the A2 compiler for the UNIVAC computer.[35]

The academically-minded members of the Association for Computing Machinery (ACM) similarly established a committee in order to develop a universal language to be used in academic publications and for sharing scientific calculations. The ACM committee eventually developed the International Algorithmic Language (IAL), which was soon renamed to just Algorithmic Langauge, or Algol. The members of the ACM committee had various academic backgrounds, but unequivocally represented what I refer to as the mathematical culture of programming. This background had strong influence on the definition of the Algol language. The language was defined using mathematical tools, such as formal grammar, that made it possible to treat programming languages as mathematical objects.[38]

The managerial, hacker and mathematical cultures of programming all contributed to the idea of a programming language. They all recognize the importance of the idea, but they view it from a somewhat different perspective. To mathematicians, programming langauges are abstract entities that can be formally studies; to hackers, they are tools to help them with coding and to managers, they are a way to make programs run on multiple machines. Those views are not incommensurable and many individuals span multiple cultures and combine their perspectives. Yet, the notion of a programming language plays the role of boundary object,[39] in that it provides a common object of interest for multiple cultures. Although the brith of programming languages brought the different cultures closer, it did not permanently bind them together. The idea of programming languages is flexible enough to allow the different cultures to keep their own distinct perspectives. Three influential programming languages that were designed in 1958 and 1959 illustrate this point.

The Algol language originated with the mathematical culture. It contains a reference to the the formal notion of an algorithm in its very name and emphasized the use of mathematical methods for programming. It was recognised as an "object of stunning beauty"[40] and a remarkable achievement of computer science. It was not widely used for building commercial software, but it became the de facto language in programming textbooks and academic publications. It was also at the core of many developments in mathematization of programming that I will follow in the rest of this chapter.

The language that eventually originated from the commercial efforts to create a portable language for business data processing was COBOL ("Common business-oriented language").[41] The language was born from the managerial culture of programming. Its designers came from commerce and government and did not include any academic computer scientist. The language aimed to be easier to use, even if less powerful and aimed to broaden the range of who can use it, for example by using English-like syntax. It was widely adopted by the data processing industry and remains in use in many legacy systems today. At the same time, COBOL was criticised by academics for its poor structure and verbosity and has never been adopted and studied at universities. COBOL's emphasis on data processing meant that it had innovative language features for working with data, a topic that I will return to when discussing types in Chpater 5.

The third language appearing at the end of 1950s was LISP, which was designed by John McCarthy at MIT.[42] The cultural origins of LISP are harder to untangle than those of Algol and COBOL. The popular story of LISP today is that it was a more or less direct implementation of the lambda calculus, a formal system developed in mathematical logic based on functions. This would place LISP firmly into the realm of the mathematical culture of programming. As pointed out by the historian of programming Mark Priestley,[43] the reality is more complex. LISP was motivated by work on symbolic artificial intelligence (AI) that required sophisticated list processing. It was inspired by a range of existing practical projects in AI and FORTRAN as well as mathematical theories including recursive function theory and the lambda calculus. LISP certainly has mathematical origins, but it was always a pragmatic language that borrowed ideas from all possible sources in order to solve practical problems. However, LISP soon became the language of choice of the MIT hackers, possibly because it was suitable for interactive and experimental way of programming favored by the hacker culture, a history that I revisit in the next chapter.

Mathematical Science of Computing

The Algol language may not have been adopted for commercial programming, but it had an enormous influence on academic computer science. The Algol report[44] is written in a combination of formal and informal language. For describing the language syntax, it uses a format that is very similar to that used in formal language theory. For description of what the individual language constructs mean, that is the semantics of the language, it relies on English language description, albeit very structured and precise.

The report also clearly shows that the use of mathematical formalisms in computer science is very different from their use in mathematics. In particular, the whole report is a definition and there are no theorems, proofs or conjectures about the language. Yet, the semi-formal definition was enough to get computer scientists to think about programs in a new mathematical way. It was also formal enough to get computer scientist to point out

issues with the Algol language definition itself, leading to a revised version of the report published three years later.[45]

The Algol report showed how to think about programs as mathematical entities and made it conceivable to use mathematical methods for their study. Suddenly, academic computer scientists could join the effort to formally study concrete programs, develop mathematical tools for proving their properties and mathematically analyze programming languages themselves. As pointed out by Mark Priestley, the Algol report played a crucial part in transforming the vague feeling that programming should be seen as a mathematical activity into a concrete scientific *research paradigm*.[46] The paradigm provided the mathematical culture of programming with basic assumptions about what programs are, what problems are worth studying and what the valid scientific methods for tackling the problems are.

One of the first to think about the possibilities of mathematical reasoning about programs was John McCarthy. In a manifesto of the Algol research programme that is relevant to this day,[47] he identifies computer science as a mathematical science in which "it is possible to deduce from the basic assumptions, the important properties of the entities treated by the science". He sees programs as such entities and outlines a number of program properties that one can study: Are two procedures equivalent? Does one procedure take fewer steps than another? Does a translation algorithm correctly translate procedures between two programming languages? Many of the questions outlined by McCarthy are still studied by academic computer scientists 60 years later. McCarthy also hopes that the new mathematical science of computing will almost eliminate debugging and claims that, "instead of debugging a program, one should prove that it meets it sspecifications".

To give a sense of the leap from plugging cables on the ENIAC and writing idiosyncratic numerical codes to formal reasoning about Algol programs, let's go through one example discussed by McCarthy. He illustrates the various formal tools he proposes by proving that a program implements the mathematical factorial function. A factorial, written $n!$ is 1 for $n = 0$ and $(n-1)! \times n$ for any greater n. McCarthy defines a function $g(n, s)$, which computes $n! \times s$ as:

$$g(n, s) = \texttt{if}\ n = 0\ \texttt{then}\ s\ \texttt{else}\ g(n-1, n \times s)$$

The function calculates the factorial by going down from the highest number to 0. To obtain a factorial of n, you call $g(n, 1)$. In each step, the function multiplies s by the current n and then it continues for $n-1$ until it counts down to 0 and then it returns the current s. When calling $g(3, 1)$, the value of s at the end will be $1 \times (2 \times (3 \times 1)) = 6$.

The definition is recurisve, meaning that g is defined in terms of g and it uses the conditional expression `if` introduced in the Algol report. The conditional expression evaluates to the result of one of the branches, depending on whether the condition $n = 0$ is true or false. This may not seem revolutionary, but it is quite a step from the instructions that machines actually execute. Only some 10 years earlier, EDSAC programmers had to program using memory addresses and two special accummulators instead of variables. To encode conditions, they had to use the `EnS` instruction, which checked if the value in the accummulator is positive and if so, jumped from the current location to the memory location `n` and then continued executing more instructions.

McCarthy proves that the $g(n, s)$ function actually implements mathematical factorial. As we will talk about proofs about computer programs repeatedly in the rest of this chapter, it is useful to see how this is actually done. I will show this in more details than

McCarthy, so that each step shows just one idea. We start with the obvious truth that $n! \times s = n! \times s$. We refine the formula in a number of steps, making sure that we always replace it with one that means the same thing, until we get to our definition of g:

$$\begin{array}{rcll} n! \times s & = & n! \times s & (1) \\ n! \times s & = & \texttt{if } n = 0 \texttt{ then } n! \times s \texttt{ else } n! \times s & (2) \\ n! \times s & = & \texttt{if } n = 0 \texttt{ then } 1 \times s \texttt{ else } ((n-1)! \times n) \times s & (3) \\ n! \times s & = & \texttt{if } n = 0 \texttt{ then } s \texttt{ else } (n-1)! \times (n \times s) & (4) \\ g(n,s) & = & \texttt{if } n = 0 \texttt{ then } s \texttt{ else } g(n-1, n \times s) & (5) \end{array}$$

The fact that proving the correctness is quite tedious, even for such a simple example, is partly the point of this example, so please bear with me. We start with an obvious truth (1) and then add a conditional expression (2). The conditional expression evaluates to either the `then` or the `else` branch, but those are both the same original expression. Next, in (3) we rewrite the expressions using facts about $n!$ In the `then` branch, we know that $n = 0$ and so $n! = 1$. In the `else` branch, we know that $n \neq 0$ and so we can replace $n!$ with $(n-1)! \times n$. We then simplify the expressions (4) and now we have our program, if we just replace $n! \times s$ with $g(n,s)$ in the formula.

The example shows that one can mathematically prove that a simple program is correct, but if we look at the articles published in the Communications of the ACM, a primary journal of the mathematical culture of programming throughout the 1960s, we can see that writing tedious formal proofs had not become a commonplace activity. The journal however adopted Algol as the language for publishing algorithms and those were written in a formally precise style. The Communications included a column containing a number of new algorithms each month. Between 1961 and 1973 the algorithms were numbered and reached number 472 when it stopped being a regular column. This confirms the point, made by Nathan Ensmenger[48] that the notion of algorithm has became central for what I refer to as the mathematical culture of programming. The magazine, however, simply published useful algorithms with a comment on what they do and did not study them formally. Publications with complete formal proofs of program correctness were relatively rare. An example is the 1969 paper "Prevention of System Deadlocks,"[49] which presents a resource allocation algorithm together with a proof that the algorithm never gets into a stuck state known as deadlock.

Being aware that proving actual programs correct was very tedious, a number of computer scientists focused on finding easier and more practical methods for doing so. Most of those were centered around some way of attaching formal mathematical propositions to specific locations in a program. The idea first independently appeared as *snapshots* in the work of Danish computer scientist Peter Naur[50] and *interpretations* in the work of American Robert W. Floyd[51]. As was typical at the time, both Naur and Floyd got involved with computers through a programming job after their studies of a different scientific disicipline, astronomy and physics, respectively. However, the method that gained notoriety in the mathematical culture of programming was described by C. A. R. Hoare in 1969 and became known as "Hoare triples". Hoare studied classics and philosophy, followed by statistics, but became a programmer for a small British computer manufacturer in 1960. A year later, he started implementing a compiler for Algol and eventually became chief engineer at the company. In 1968, he moved to academia and began working on axiomatic description of programs. Building on the work of Naur and Floyd, he proposed a method for formally reasoning about programs in "An Axiomatic Basis for Computer Programming"[52]. Here, Hoare

introduced a notation written as $\{P\}c\{Q\}$. In the formalism, c is a program instruction also called a command and P, Q are formulas of mathematical logic called pre-condition and post-condition, respectively. A Hoare triple $\{P\}c\{Q\}$ asserts that, if a pre-condition P holds before the execution of a command c, then the post-condition Q will hold after the program runs.[53]

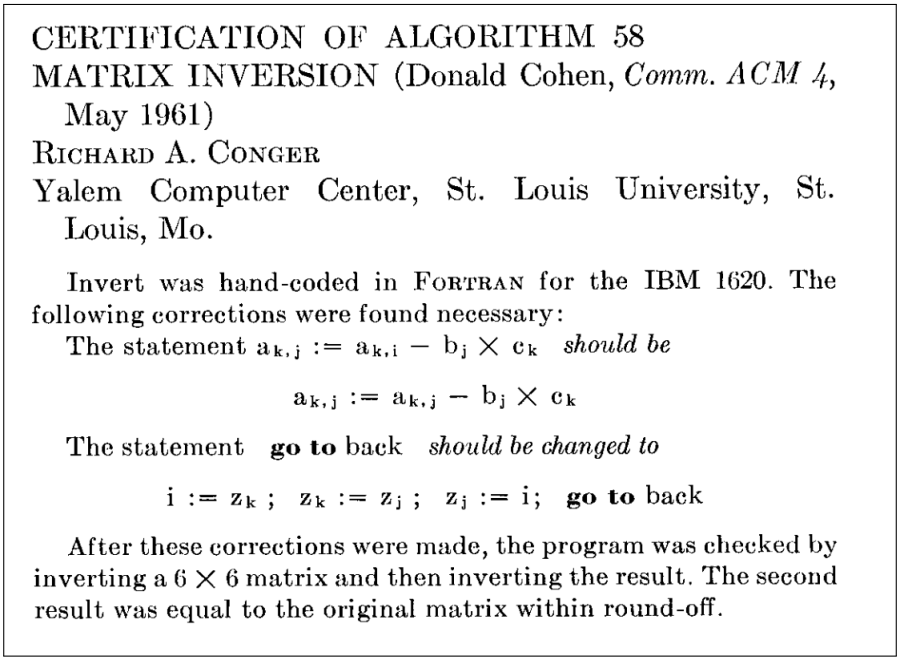
There are two interesting aspects of the axiomatic approach introduced by Hoare. The first aspect is that, the pre-conditions and post-conditions P, Q in $\{P\}c\{Q\}$ do not need to talk about everything that is happening in the program. They can only involve aspects that are necessary for the proof and ignore anything that is irrelevant. The second aspect is that the approach is compositional. Let's say that we have two commands c and d with properties $\{P\}c\{Q\}$ and $\{Q\}d\{R\}$ where Q is the post-condition of the first command and, at the same time, pre-condition of the second command. Now, if we write a program $c; d$ that runs the two commands in a sequence, we know that $\{P\}c; d\{R\}$ because the post-condition of the first command is the same as the pre-condition of the second command. In the first paper, Hoare used his approach to prove the correctness of a simple program that performs division with a remainder, but he later used a similar approach to prove the correctness of a somewhat more complicated program FIND that partially sorts an array of data, which was published in the Communications of the ACM.[54]

To offer a sense of proofs about programs based on Hoare triples, let's look at a different way of calculating the factorial of a number n. This time, the program is written as a loop that iterates from $i = 1$ to $i = n$ and, in each steps, multiples the result s by i:

$$
\begin{array}{l}
s = 1; i = 1; \\
\texttt{while}\ i < n\ \texttt{do} \\
\quad i = i + 1; \\
\quad s = s \times i; \\
\texttt{end}
\end{array}
$$

The value s is first set to 1. It is then gradually multiplied by $1, 2, 3, \ldots n$ and so the final value will be $s = 1 \times 2 \times 3 \times \ldots \times n$. But how can we formally prove that this actually works? Using Hoare triples, we can focus on the key part of the program, which is the body of the loop. It turns out that, for the two assignments in the body, the following holds: $\{s = i!\}i = i + 1; s = s \times i\{s = i!\}$. This means that, if s is a factorial of i before the body runs then, after the two assignments are executed, s will again be the factorial of i. The trick is that i is now incremented by 1 and s has been changed accordingly. We also know that, as the program starts, $\{\}s = 1; i = 1; \{s = i!\}$. This simply tells us that, when we set s and i to 1, it will be true that $s = i!$ because $1 = 1!$ The compositionality of the method means that we can now put all these arguments together. We start with $s = i!$ and this is also the pre-condition and post-condition of the loop body, which means that $s = i!$ will be true after the loop finishes looping. But then, $n = i$ and so at the very end $s = n!$ which is exactly what we wanted to show.

It may not seem like that from my examples, but the axiomatic approach did make reasoning about more complex programs easier. This is due to the two aspects that I mentioned earlier. We need to choose our axioms so that they talk only about what matters and the final proof can be composed from proofs about individual parts of programs. Together, these two aspects also enabled later developments of semi-automatic program verification tools, a topic that I will return to at the end of this chapter.



CERTIFICATION OF ALGORITHM 58
MATRIX INVERSION (Donald Cohen, *Comm. ACM 4*, May 1961)

RICHARD A. CONGER
Yalem Computer Center, St. Louis University, St. Louis, Mo.

Invert was hand-coded in FORTRAN for the IBM 1620. The following corrections were found necessary:

The statement $a_{k,j} := a_{k,i} - b_j \times c_k$ *should be*

$$a_{k,j} := a_{k,j} - b_j \times c_k$$

The statement **go to** back *should be changed to*

$$i := z_k ; \quad z_k := z_j ; \quad z_j := i; \quad \textbf{go to} \text{ back}$$

After these corrections were made, the program was checked by inverting a 6×6 matrix and then inverting the result. The second result was equal to the original matrix within round-off.

Figure 2.4: An algorithm certification published in the Communications of the ACM[56]

In retrospect, it is difficult to say what the expectations of the authors contributing to the mathematical culture of programming were. Hoare treats programming almost as a branch of mathematics. His hope likely was that that programmers would use deductive reasoning themselves to ensure their programs are correct. McCarthy was aware that this is a lot of work and suggests that programs should be involved in checking of the proofs, "because we can require the computer to do much more work in checking each step than a human is willing to do."[55] This idea is closer to the way mathematical methods are used in software verification tools today. However, building such tools required both furhter theoretical innovations and the involvement of the engineering culture of programming in order to build tools that can be used to check non-trivial practical software systems.

Many Definitions of Algol

Early examples of rigorously applied formal mathematical reasoning to computer programs involve very simple programs or algorithms. Even the aformentioned proof of the FIND program by Hoare, which shows "the construction of the proof of a useful, efficient, and nontrivial program"[57] studies an algorithm that has some 20 lines of code. In practice, even the proponents of the mathematical culture relied heavily on empirical testing. The Communications of ACM magazine confirms this fact. In addition to publishing algorithms themselves, the algorithms column also published "certifications" of previous algorithms, such as the one in Figure 2.4. Those were certinaly not proofs. Instead, they were personal reports confirming that the reader implemented the algorithm, tested it, possibly corrected it and obtained a correct result.

A prime example that illustrates the struggles with applying rigorous mathematical methods to large software systems is the quest for a formal description of the Algol programming language.[58] Algol emphasized its mathematical structure. As an object of stun-

ning beauty for the theoreticians, it is an obvious target for formal mathematical treatment. Yet, formally defining Algol turned out to be more difficult than expected. The difficulties shed light on problems that computer scientists still face 60 years later.

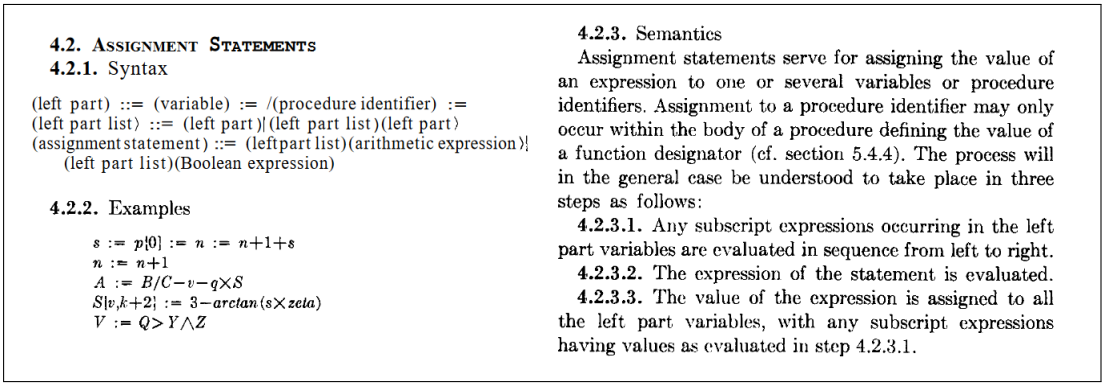
Algol was developed by a committee of American and European computer scientists in a meeting in 1958 in Zurich and revised in a subsequent meeting in 1960 in Paris. Figure 2.5 shows an excerpt from a canonical description of Algol, "Revised report on the algorithmic language Algol 60"[59], which "gives a complete defining description of the international algorithmic language Algol 60". The report defines the syntax of the language, provides numerous examples and defines the semantics of individual constructs. The syntax is specified formally using the Backus-Naur form, a notation technique developed for the report and is still used in modern programming language specifications. The semantics is described in English, which is precise enough for a human reader, but does not define the programming language as a formal mathematical entity.

Contrary to the expectations about mathematical publications, the report does not specify any theorems about the Algol language. It is merely a 17-page long definition. This illustrates an important discrepancy between mathematics and programming. Mathematical theorems are shaped by their proofs, counter-examples and the mathematical striving for simplicity and elegance.[60] In contrast, the structure of theorems about programs is mainly shaped by the programs they talk about. The programs are, in turn, shaped by the aim of building software that can be run to achieve some task. Algol may be a carefully designed simple programming language, but as a mathematical object, it remains very complex. The complexity of Algol poses a significant challenge to those who aim to study properties of the language using mathematical methods.

The first task for the proponents of the mathematical culture of programming was to produce a mathematical definition that was fully formal, rather than relying on informal English langauge description. McCarthy[61] made the first step in 1964 by defining a formal semantics of a minimal subset of Algol that he called Microalgol. Microalgol includes the tricky `goto` construct, but omits many other features of Algol. The semantics is defined as a recursive function $\texttt{micro}(\pi, \xi)$, which defines a single step of the program execution. Given a program π and a current state of execution ξ, the result of $\texttt{micro}(\pi, \xi)$ is a new state ξ' that describes what has changed after running a single statement of the program. The states ξ, ξ' consist of vectors of values assigned to variables together with a number representing the next statement to be executed.

McCarthy acknowledges that Algol is considerably more complicated and lists a number of specific issues, but believes that "those difficulties can be resolved and that a clear description of the state of an Algol computation will clarify the problem of compiler design". The case of Microalgol also illustrtes two important aspects of programming langauge definition. The first is the meta-langauge used. This is the language used to talk about the programming language. McCarthy uses normal mathematical notation with concepts like vectors and functions. The second aspect is, how is the meaning of programs defined. In case of Microalgol, the mathematical definition works as a kind of interpreter. It uses formal mathematical methods to define how programs execute and, step by step, transform a mathematical representation of the state of the computer executing the Algol program.

Hoare, who introduced the $\{P\}c\{Q\}$ notation for reasoning about programs, uses a different mathematical formalism. Rather than specifying how individual language constructs such as the assignment operator transform the state of the computer, he gives ax-



4.2. Assignment Statements

4.2.1. Syntax

⟨left part⟩ ::= ⟨variable⟩ := | ⟨procedure identifier⟩ :=
⟨left part list⟩ ::= ⟨left part⟩ | ⟨left part list⟩⟨left part⟩
⟨assignment statement⟩ ::= ⟨left part list⟩⟨arithmetic expression⟩ |
⟨left part list⟩⟨Boolean expression⟩

4.2.2. Examples

$s := p[0] := n := n+1+s$
$n := n+1$
$A := B/C-v-q\times S$
$S[v,k+2] := 3-arctan(s\times zeta)$
$V := Q > Y \wedge Z$

4.2.3. Semantics

Assignment statements serve for assigning the value of an expression to one or several variables or procedure identifiers. Assignment to a procedure identifier may only occur within the body of a procedure defining the value of a function designator (cf. section 5.4.4). The process will in the general case be understood to take place in three steps as follows:

4.2.3.1. Any subscript expressions occurring in the left part variables are evaluated in sequence from left to right.

4.2.3.2. The expression of the statement is evaluated.

4.2.3.3. The value of the expression is assigned to all the left part variables, with any subscript expressions having values as evaluated in step 4.2.3.1.

Figure 2.5: An excerpt from the Revised report Algol 60 Backus et al. (1963)

ioms about individual langauge constructs. For example, the Axiom of Assignment[62] states that, if predicate $p(x)$ holds before the assignment $x := f$ assigns a value f to a variable x, then a predicate $p(f)$ will hold afterwards. Using Hoare's notation, $\{p(x)\}x := f\{p(f)\}$. The method used by Hoare is quite different from that of McCarthy, but he relies on the same general approach of using a simple subset of the actual Algol language. He omits the tricky `goto` construct and acknowledges that this might be difficult to reason about because it is not compositional. He however believes that other aspects of Algol will not cause any great difficulties. Hoare also shares McCarthy's overall optimism and believes that "the practical advantages of program proving will eventually outweigh the difficulties."

Scaling from a small subset of Algol to the full language proved to be significant undertaking and required the development of new techniques for specifying programming language semantics. Peter Mosses, who worked on a formal definition of Algol as part of his PhD at University of Oxford, followed the style developed by his PhD advisor Christopher Strachey. Rather than using mathematics to define an interpreter or provide axioms, the new method assigns each statement of the language a meaning, independently of the rest of the program. The meanings are mathematical objects, specifically functions written using the lambda calculus formalism. The individual meanings can be composed to get the meaning of an entire Algol program. The 1974 definition by Mosses[63] covers almost the entire Algol language and uses a highly regular mathematical notation that a reader with mathematical background (and sufficient dedication!) is expected to understand. The length of the document made Christopher P. Wadsworth, another PhD student from Oxford studying under Strachey's supervision, question the simplicity of Algol. In a letter to his advisor, he wrote about Mosses' definition of Algol:

> *I must admit I still feel a little surprised it's as long as it is–I guess Algol 60 is just not nearly as 'well-behaved' as one tends to think it is.*[64]

The case of the Algol language definition developed by Mosses reinforces the point that I made earlier about mathematical methods applied to programs. Despite being mathematical, they are nothing like normal mathematical texts. The report by Mosses consists of a 50-page long definition, but no theorems. The complexity of the definition comes from the programming language it is attempting to model. The challenge in producing the definition is in how to structure it so that it can account for all the complexities of the language, but there are no elegant mathematical tricks.

The definition by Mosses, which was published as a technical report by the Oxford University Computing Laboratory in 1974 also illustrates that, by this time, the mathematical culture of programming had an established a methodology for doing normal science. The report has only a brief 1-page introduction, which states that it models Algol, as described in the Revised Report, except that 'own' declarations are omitted. The author does not need to explain what this means and does not need to make any attempt to convince the reader that the model is correct, for example by comparing the behaviour of the mathematical model with programs produced by a specific Algol compiler. The mathematical culture leaves implicit the fundamental assumption that the model is relevant to actual software. As we will soon see, both the complexity of the definition and the unclear relationship with actual software will be identified as problematic by those not strongly committed to the mathematical culture.

Using the ordinary language of mathematics as a meta-language for complex programming language definitions is a limiting factor. Such a mathematical definition has to be written by hand and it has to be checked by other human readers. The computer scientists working on programming languages at the Vienna IBM laboratory hoped that they could develop a compiler for a programming language systematically from its definition through a series of step-wise refinements. For this, the meta-language should not be just the language of human-written mathematics, but instead another formal language that can be processed by a computer. To do this, the Vienna IBM team created a formal language known as Vienna Definition Language (VDL) for writing the semantics of programs.

The IBM team used the newly developed VDL meta-language to give a series of definitions of Algol. The first was a 67 page definition from 1968[65] that followed the operational style used earlier by McCarthy. Another version developed four years later[66] aimed to simplify the first one by handling jumps like `goto` in a different way. Finally, the definition that appeared a couple of years later, in 1978,[67] adopted the denotational approach used by Mosses earlier, aiming to produce "an equally abstract but more readable definition" that is simple enough to fit in a book chapter.

The many attempts to formalize Algol illustrate the variety of work done within the mathematical culture of programming, as well as influences of other cultures. Approaches that formalize only a subset of Algol generally aim at explaining interesting aspects of the language to a human. This kind of work is close to traditional mathematics in that it is primarily focused on other human readers. Much of contemporary academic work on programming language design follows this style. A well-chosen language subset makes it easy to illustrate interesting aspects of design choices or programming language features. It also typically limits the scope to an elegant mathematical object. The obvious limitation is that such formal definition cannot be used to provide any guarantees about the full language.

Approaches that aim to formalize the full programming language depart from traditional mathematical practice. They inevitably involve long and complex definitions that are shaped by practical programming concerns rather than mathematical elegance. The methods of defining formal semantics of a full language also often involve the development of new software tools such as VDL built in the Vienna IBM laboratory. We might speculate that the influence of engineering culture of programming was stronger in an industrial research laboratory than at University of Oxford where similar definition used a more traditional mathematical notation.[68]

The Minority Report

For the mathematical culture of programming, the 1960s were a period of active development. The first stand-alone university departments of computer science started to appear, the ACM published its first curriculum recommendation in 1968 and universities started to offer undergraduate courses and degrees in computer science. Academic computer scientists were busy developing new algorithms, programming languages and mathematical tools for reasoning about them.

The efforts to formalize Algol were underway, but the language was also still under development. Soon after the publication of Algol 60, the Algol working group started discussing the requirements for the next version of the language. Proposals included more powerful data structuring mechanisms that I return to in Chapter 5, as well as plethora of other ideas, such as more unified syntax, obtained by removing the distinction between statements and expressions, the ability for the language to manipulate its own programs, different mechanisms for handling I/O, as well as new ways of structuring the formalism used in the language definition.[69]

In 1965, the Algol working group made a decision to adopt a proposal for a completely new design of the language, rather than one based on gradual improvements in Algol 60, which was the preferred choice earlier. The new design aimed at even a greater mathematical elegance and uniformity, but many of the working group members saw it as "obscure", "incomprehensible" and overly complex. The design was eventually published as Algol 68, but the publication also included a "Minority Report" written by working group members who disagreed with the final design. The dissidents felt that the language description methodology used in the project has failed.[70]

The authors of the Minority Report argued that the problems faced by programmers have changed significantly over the last decade and that programmers find themselves "faced with tasks of completely different and still growing scope and scale." In an insightful analysis of the Algol dissidents' thinking, historian of computing Thomas Haigh pointed out[71] that the group was gradually becoming less interested the problem of the "expression [of programs] in some language" and more interested in the problem of "reliable creation of programs to perform specified tasks." In other words, they started to see "languages as tools designed to support or enforce good practice in the design of complex programs." This new way of thinking presents a shift from the mathematical culture of programming to what I refer to as the engineering culture of programming that was emerging at the time. The Algol dissidents went on to establish a working group on Programming Methodology (IFIP WG 2.3). As a later collection of articles published by the authors[72] shows, the group kept a strong mathematical emphasis, but was gradually interested in broader software engineering issues, architectural metaphors for software development, program structuring and the activity of programming.

Many of the Algol dissidents were also involved in an event that has been widely regarded as a turning point in the history of computing. The event was the NATO Software Engineering conference, which took place in October 1968 in Garmisch, Germany (Figure 2.6). The conference is often linked with the broader industry crisis and the hopes to turn "the black art of programming" into a "science of software engineering."[73], although as convincingly argued by Haigh, the purported impact of the conference is hard to "square with the actual historical record". In reality, the legacy of the meeting has largely been shaped

Figure 2.6: One of the photographscirculated by the organizers after the NATO 1968 conference.

by the Algol dissidents, many of whom presented their work at the conference and who also served as editors of the final, highly quotable, conference report.[74]

The conference brought together some of the Algol dissidents, but also a broader computing community from both academia and industry, yet there was a heavy emphasis on those working on complex and academically interesting software systems such as compilers and operating systems. The phrase "software engineering" was novel at the time and was deliberately chosen as being provocative. It implied the need to move, both from the question of programming languages to the question of software production, as well as the need to move from hacker practices to a development process based on theoretical and practical foundations, following the model of established branches of engineering.

Many of the participants left the Garmisch conference with a sense of excitement. Dijkstra recalled that the meeting was a success "largely because most of the people present were sufficiently high in their local hierarchies that they could afford to be honest, and were." The common recognition that there was a software crisis was the most exciting aspect for all the participants. To capitalize on the excitement the organizers immediately planned a follow-up conference that was held a year later in Rome. The organizers hoped to move from the recognition of the software crisis to concrete proposals for addressing it. The aim was to focus on technical rather than managerial problems. This did not turn out as well as hoped. In the words of the attendees, the conference just "never clicked" and they "left with an enormous sensation of disillusionment".[75]

The two conferences brought together different cultures of programming. This did not prove to be an issue in 1968. The conference was focused on waht is wrong with current methods and everyone was able to contribute from their own perspective. There was enough common understanding among the participants to accept points made by others, even if they felt those were not the most fundamental issues. The aim of the follow-up event was to discuss specific issues in detail and, possibly, come to agreements. This proved to be more difficult. As the editors of the second conference report acknowledge "a lack of communication between different sections of the participants became a dominant feature"[76] of the second conference. Thinking about different cultures of programming offers a useful perspective for understanding this lack of communication.

The organizers wanted to focus on technical rather than managerial problems. By doing so, they already excluded the issues that mattered to the managerial culture of programming, which saw structuring of work and team management as the primary way towards more satisfactory results. Even among the more technical participants, there were widely different perspectives. The discussion about proofs of program correctness was of interest only to the mathematical culture of programming. Time-sharing systems, which we will discuss in Chapter 6, were seen as a powerful new approach by the hackers, but as a tool that teaches sloppy habits by more mathematically inclined participants. The Algol dissidents and like-minded attendees, who were starting to establish what I refer to as the engineering culture of programming, were primarily interested in the development of new ways of structuring programs and in practical tools to simplify programming, but not in management practices or purely mathematical puzzles. The case that perhaps best demonstrates the disagreements between the different cultures is that of structured programming.

Go to Considered Harmful

The development of new reasoning techniques, like the axiomatic method based on Hoare triples, came hand in hand with the effort to write programs in a way that makes reasoning about them easier. The primary construct in Algol that made such reasoning difficult was the `goto` statement. The `goto` statement is a bit like the `EnS` instruction from EDSAC that I talked about earlier. It unconditionally transfers the control, i.e. the position in the program that is currently being executed, to another location that is defined by the argument passed to `goto`. To see the construct in action, we can compare our earlier factorial program with a version written using `goto` instead of `while`:

$s = 1; i = 1;$		$s = 1; i = 1;$
`while` $i < n$ `do`	$L1$	`if` $i = n$ `then goto` $L2$
$\quad i = i + 1;$		$i = i + 1;$
$\quad s = s \times i;$		$s = s \times i;$
`end`		`goto` $L1;$
`print`$(s);$	$L2:$	`print`$(s);$

The two programs work in exactly the same way. The only difference is that the program on the left uses the `while` statement, whereas the version on the right implements the logic of the looping using `goto`. When $i = n$, it jumps to the end of the program to print the result. Otherwise it continues to run the sequence of instructions to update the values of i and s and then jumps back to the label $L1$ to again check if it is done.

The use of `goto` complicates reasoning about programs. In the version on the left, the code has a clear structure that makes it easier to see that the block nested inside the `while` block will run repeatedly in a loop. If we want to reason about the code formally, we know that we need to understand the pre-conditions and post-conditions of the body of the loop and then compose those with what we know about the code before and after the loop. In the version on the right, the program is just a flat sequence of instructions. This breaks compositionality, because you cannot analyze the behaviour of `goto` $L1$ without looking at the rest of the program and locating the instructions at the location $L1$.

Even though commercial computer programmers did not reason about their programs using Hoare triples, they needed to think about programs informally and this is much easier

for programs written using higher-level language constructs like `while`. Many computer scientists believed that this way of writing code is preferable. The most vocal advocate of avoiding `goto` was Edsger Dijkstra, thanks to his letter "Go to statement considered harmful".[77] Dijkstra explains the issues with reasoning about programs that include `goto` and observes, with his typical eloquence, that "the quality of programmers is a decreasing function of the density of `goto` statements in the programs they produce". Following the publication of the letter, Dijkstra also introduced the term *structured programming* for the style of programming that avoids `goto` in a working paper presented at the NATO 1969 Software Engineering conference in Rome.[78]

Dijkstra's letter sparked a debate in the computer science community. Most of those joining the debate accepted that `goto` should not be overused. Some argued that new programming language should not include it at all, while others gave various arguments in favor of it and examples where it is useful. Some also pointed out that any program can be rewritten without the use of `goto`. The canonical reference for this claim, which became known as the *Structured Programming Theorem* bcame a 1966 paper by Böhm and Jacopini[79] that Dijkstra cited to make this point. The style of the paper is very technical and it uses a somewhat different formulation of the problem and has been "apparently more often cited than read in detail."[80] Nevertheless, the theorem was a useful weapon for the proponents of structured programming. As later recalled:

> *Us converts waved this interesting bit of news under the noses of the unreconstructed assembly-language programmers who kept trotting forth twisty bits of logic and saying, 'I betcha can't structure this.'* [81]

The quote is worth reading carefully, because it gives a number of hints about different cultures of programming. The reference to *assembly-language* programmers suggests that those who opposed structured programming preferred a more direct access to the machine, a trait of the hacker culture of programming. Hackers were also concerned about efficiency, which was another argument in favor of `goto` in the debate. It is also interesting to see that a purely theoretical paper, which discusses normalization procedure for flow diagrams, provided such valuable argument to the proponents of structured programming. The paper has no explicit mention of `goto` or, indeed, programming languages and even Dijkstra quotes is cautiously, saying that it "seems to have proved the superfluousness of `goto`", but that the mechanical translation used in the paper is "not to be recommended", because it would produce result that is likely much less readable than the original unstructured code. Yet, theoretical results are deemed as a prime form of knowledge in the mathematical culture of programming and so the result served as a strong argument.[82]

Regardless of the debate, the structured programming approach was soon widely known and adopted, not just by those members of the mathematical culture of programming who were interested in proofs about programs, but also by the broader programming community. As we will see next, the idea also captured the imagination of programming managers, who were less concerned with reasoning about programs and more with control over complexity, budgets and workforce.[83]

Chief Programmer Teams

Edsger Dijkstra, who introduced the term *structured programming* cannot be pigeonholed into a single culture of programming. He was one of the Algol dissidents and was hoping to find sound body of knowledge for the discipline of programing. His work was rooted in mathematical methods, but he was not using mathematics to solve mathematical puzzles, but to improve applied programming. His writing on structured programming illustrates many aspects of what I refer to as the engineering culture of programming. Structured programming is presented as a way that helps the engineer, with their inherent limitations, do their job better. There is also a strong emphasis on individual responsibility. One can almost imagine that using structured programming when possible would be a requirement in Code of Ethics that all software engineers should adhere to. This engineering perspective on structured programming remains well-aligned with the mathematical view that I discussed above. Structured programming simplifies formal reasoning, which is what mathematicians care about, but it also supports informal human reasoning that an engineer needs to undertake.

However, the idea of structured programming also influenced management of software projects. Managers saw structured programming as a systematic approach where you divide the overall problem into smaller sub-problems that can be solved independently by less skilled programmers. This was appealing to 1970s managers as the top-down software development approach resembled the hierarchical top-down structure of large corporations at the time.[84]

The success story of the management approach to structured programming was an information retrieval system developed by IBM for the New York Times.[85] According to its developers, the system was completed several times faster than comparable systems and ran with almost no errors. The programming followed a top-down methodology inspired by structured programming that the authors called Chief Programmer Team (CPT). In CPT, a single "chief programmer" was responsible for the overall system design and was supported by a group of assistants who were assigned simpler programming tasks that did not require a full understanding of the system. The chief programmer designs the top-level program structure and directs a team of supporting programmers who solve specific well-defined problems as required by the chief programmer. The hierarchical structure in the program, provided by structured programming, is thus reflected in the structure of the development team.

The originator of the term *structured programming* Edsger Dijkstra never accepted the managerial interpretation of the idea. He was horrified that programming started to turn from an intellectual discipline to an industrial activity with the prevailing American "management philosophy aiming at making companies as independent as possible of the competence of their employees".[86] According to Dijkstra, managerial software engineering methods aim for an anti-intellectualism characterized by "How to program if you cannot."[87] Despite the disputes, the different takes on the idea often appeared together. The December 1973 issue of the Datamation magazine[88] was dedicated to stuctured programming and included the engineering account of `GOTO`-less programming, an article on the Chief Programmer Team methodology as well as an overview of the structured programming theorem and a light-hearted article proposing to replace the universally hated `GOTO` construct with a "revolutionary" `COMEFROM` feature.[89]

The concept of structured programming and the disagreements that it provoked illustrate a common pattern. Structured programming undeniably benefited multiple different cultures. To the engineering culutre, it offered a useful tool for structuring programs so that they are easier to read. It supported compositional reasoning about programs that was appealing to the mathematical culture. The managerial culture used structured programming as a mechanism for structuring workforce, rather than programs. Yet, as illustrated by Dijkstra's remarks, the proponents of different cultures held conflicting perspectives on the concepts they shared. Those perspectives were not incommensurable, but they were rooted in conflicting assumptions and led to contentious debates.

The cases of programming languages and structured programming are both instructive. Programming languages appeared thanks to a productive meeting of managerial, hacker and mathematical cultures of programming. Programming languages serve as boundary objects that the different cultures could share, but that is interpreted differently by each of them. However, programming languages also allow further communication between different cultures. We can thus also view programming languages as trading zones[90] through which the different cultures exchange ideas. This is exactly what happened with structured programming. It appeared as a way of writing code using loops and other structured constructs instead of `goto`. This shaped mathematical formalizations of Algol which sometimes omitted `goto`, seeing it as inelegant and unnecessary complication. Structured programming also led to the managerial idea of structuring teams to match the structure of code.

In the case of programming languages and structured programming, the collaboration between the different cultures of programming was enabled by having a shared technical concept that can be implemented and has an existence of its own. Concrete technical concepts play a role similar to experiments in physics which, can also "have a life of their own."[91] When deadling with a more abstract idea, direct collaboration is harder. The existence of multiple cultures of programming is beneficial in that, when a particular development reaches a dead end, another culture of programming may adopt an idea and find a different way forward. In a way, this is what happened in response to the difficulties with manually writing proofs of program correctness at the end of the 1970s.

Program Proofs and Social Processes

The opening keynote speaker at the 1968 NATO Software Engineering conference was a prominent computer scientist, Alan Perlis, who had received the Turing Award in 1966 "for his influence in the area of advanced programming techniques and compiler construction."[92] Perlis was a mathematician by training, although with a clear interest in computers. His PhD thesis, completed in 1950, describes an algorithm for solving integral equations. It does so from a mathematical perspective, but with computing applications in mind.

Despite his mathematical background, Perlis' keynote and comments at the NATO conference made it clear that he did not see programs as just another mathematical entity. In the opening keynote, he says that the concern of the conference are objects which depend only in "very weak ways on the laws of physics" and whose "structure depends as much on the social laws governing their usage as on their internal constraints."[93]

Perlis also attended the follow-up 1969 conference and participated in a discussion on software correctness. In response to Hoare, who argued that you should convince yourself

that a program works by inductive mathematical proofs, Perlis expressed skepticism and suggested that suitably selected test cases are often sufficient. To this, Dijkstra responded with his famous remark that "testing shows the presence, not the absence of bugs."[94]

Some five years later, Perlis was sitting in a seminar at Yale University. The speaker was Richard DeMillo, who was visiting his collaborator Richard Lipton. DeMillo was presenting his research on verifying programs and included an extensive proof of one of the algorithms. Perlis raised his hand and asked "Why does it take 20 pages to prove aprogram that is obviously correct?"[95]

The question pointed to a fundamental difference between proofs of mathematical theorems and proofs of computer programs. A difference that DeMillo and Lipton, also trained mathematicians working as computer scientists, soon recognized. The question prompted a collaboration that resulted in a 1977 paper[96], which triggered a controversy in the computer science circles. In the paper, DeMillo, Lipton and Perlis argue that program verification is bound to fail because of the nature of proofs that it demands.

What a mathematical proof is may appear simple at a first glance. In logic, proofs are finite sequences of propositions, each of which is an axiom, an assumptions or follows from earlier propositions by a rule of inference. In practice, proofs are much richer. They use a mix of natural and formal language and they are written to convince another, adequately qualified, colleague. Mathematical proofs are central to mathematical practice. They are taught at universities, shared with colleagues and discussed in whiteboard sessions where mathematicians marvel at the elegance of a proof and clever tricks that make it work. If they trust a proof, they believe that it could, in principle, be written as a sequence of propositions, but they never bother writing it in such formal way.

In programming, proofs are bound to be secondary. Unlike mathematical theorems that are written to be proved, computer programs are written to be executed. A proof exists merely as a certification that a program matches its specification. Most proofs of program correctness are also quite dull. They typically have a very repetitive structure. Rather than using ingenious tricks, they enumerate a large number of fairly obvious cases.

The point made by DeMillo, Lipton and Perlis is that mathematicians gain confidence about mathematical proofs thanks to the rich informal social processes of actual mathematics. Since no comparable social processes can take place among those who work on proving programs correct, formal mathematical approach to programming can never actually eliminate program errors. Although the authors focused merely on the confidence that we can have in the correctness of programs, DeMillo was also aware of the idea by philosopher of science, Imre Lakatos,[97] that mathematical knowledge is built through a process of proofs and refutations. The authors knew that if computer scientists treat proofs merely as certifications, they lose an invaluable force that shapes knowledge in mathematics.

Although DeMillo, Lipton and Perlis were all mathematicians by training and inclination, they did not uncritically accept the assumption of the mathematical culture of programming that programs are mathematical entitites like any other. They believed that programs can be studied using mathematical methods, but accepted that their structure is sufficiently different from ordinary mathematical theorems.

The paper was first presented at the Principles of Programming Languages (POPL) conference[98] and a revised version was later published in Communications of ACM[99]. The reponses to the two publications show that questioning formal proofs about programs touches on a topic that different cultures treat very differently. Many of the positive letters published in a follow-up issue of the Communications of ACM came from readers working

in industry. One reader thanked authors for their wisdom, fairness, style, rigor, and wit while another reader thanked the editors for giving him the first article he enjoyed since joining the ACM.[100]

DeMillo later reflected[101] that the three of them likely became spokespeople for computer industry practitioners who felt that academics were pushing solutions that were not going to work for them. To one manager, the article explained why he "should not look for development of formal proof of the 'correctness' of our programs" while another reader argued that he "cannot recall a single instance in which a proof of a program's correctness would have been useful". He argued that actual bugs are easy to discover through testing; more serious errors are errors in specification that would not be caught by proofs. As a better approach, the reader points to a review of software engineering techniques, including the Chief Programmer Team approach.

Given their background, DeMillo, Lipton and Perlis cannot be clearly associated with any single culture of programming. All three of them have strong mathematical backgrounds, but their view of programs and programming seems more aligned with one that I associate with the engineering culture of programming. They recognize the irreducible complexity of computer programs as well as fundamental human limitations, both of which are typically ignored by the mathematical culture of programming. They also contributed to or advocated for the use of testing in programming; Perlis in his 1968 keynote and DeMillo in his subsequent research on mutation testing. Their critical paper was certainly appreciated by those favoring the managerial and the emerging engineering perspectives.

There were also a number of responses that criticized DeMillo, Lipton and Perlis from the perspective of the mathematical culture. In a letter to editor, Leslie Lamport pointed out that the recent work of Floyd, discussed earlier in this chapter, "taught us that a program is a mathematical object" and likens computer scienctists before Floyd to "geometers before Euclid". Lamport believes that "theorem either can or cannot be derived from a set of axioms" and that their correctness is not decided by a social process. In the letter, he also criticizes the practice of Communications of ACM to publish algorithms without correctness proofs in the "Algorithms" column and notes that the published "Certifications" provide only a flawed social process. Edsger Dijkstra went even further and labelled the article "A political pamphlet from the Middle Ages".[102] Dijkstra does not disagree that communication with colleagues is an essential component of the mathematical culture, but he argues that the authors "give a complete caricature of program verification" and that how to prove program properties more elegantly is the subject of lively interchange in the field. Dijkstra's response thus does not defend a purely formal view of proofs as that of Lamport. His perspective is more appreciative of the human side, but he believes that human programmers should aspire to mathematical rigor and elegance.

Many historians have written about the debate triggered by the DeMillo, Lipton and Perlis paper and tried to assess its significance. Donald MacKenzie[103] suggested that it likely contributed to the decrease of US government funding for formal verification, but computer scientists committed to the mathematical culture who responded to the paper typically disregarded the arguments it made. Yet, looking at the follow-up developments, it seems that the paper captured issue that the mathematical culture was, perhaps only unconsciously and rather reluctantly aware of. If the mathematical culture insisted on using conventional mathematical proofs, it seems likely that it would soon reach a dead end. Real-world software systems were simply too complex for this.

I suggest that this is a case where the pluralism of programming with its multiple cul-

tures came to the rescue. Rather than stagnating and eventually falling out of interest, the work on program verification received new impetuses, coming from two other cultures. The resulting two developments were not a direct response to the critique, but they both address the lack of social processes surrounding program proofs. The managerial Cleanroom methodology explicitly creates a new social process, while the engineering work on mechanized proofs replaces the need for human checking with machine checking.

Cleanroom Methodology and Mechanized Proofs

As we have seen, formal proofs of program correctness done in the mathematical culture mostly focused on simple algorithms in the 1960s. In the 1970s, there was much pioneering development on novel tools and methods for program verification that was to bear fruit later. This included formal languages for writing specifications, such as the Vienna Definition Language (VDL) and the Z notation, tools like the LCF theorem prover that I will return to in Chapter 5, as well as theoretical developments that later allowed program verification using model checking. Yet, the mathematical proofs of programs constructed in the 1970s still focused on simple academic programs.

Some of those working in industry were aware of the developments and attempted to adapt them to help with large scale software development. We already saw one example of this. The Chief Programmer Team (CPT) methodology adapted the idea of structured programming, shifting focus from structuring programs to structuring teams. However, CPT only addressed the problem of structuring teams. The quality of the resulting software still depended primarily on the skills of the chief programmer.

The Chief Programmer Team methodology was developed by Harlan Mills, who had background in mathematics and worked as a research fellow at IBM. With a PhD in game theory, but an industry job, Mills was perhaps the best person to import ideas from the mathematical culture into the world of applied software development. The CPT methodology was tested and developed through two case studies. The first one, preliminary and somewhat inconclusive, was presented at the NATO 1969 conference in Rome. The second successful case study of an information retrieval system for the New York Times was widely discussed, including in the December 1973 issue of the Datamation magazine. The CPT methodology focused on effective organization of work, but it did not address the issue of correctness. However, making sure that a software system or a software component correctly implements its specification was an equally important issue. As Mills later put it, clearly showing his appreciation for mathematical methods, “software engineering without mathematical verification is no more than a buzzword.”[104]

To address the challenge of quality control, Mills proposed an extension of the CPT methodology that he called as Cleanroom Software Development[105]. Again, the approach was inspired by the work within the mathematical culture, but adapted it for more managerial purposes. It also took further an analgoy with surgical team that Mills used when designing the Cheif Programmer Team structure. Like the software development team in CPT, a surgical team has a single lead surgeon, supported by a group of assistants. As the “Cleanroom” name suggests, the new methodology adds a controlled environment, akin to the operating theatre used by surgeons. In Cleanroom, systems are built by following a detailed specification. A programmer is required to review the conformance to the specification rigorously, first to convince themselves and then to convince other team members. “It is obvious” is an acceptable proof if everyone agrees. The process is fundamentally hu-

man and the required degree of rigor is a managerial choice. To ensure correctness of the final product, the Cleanroom methodology employs statistical quality control, measuring "reliability over a probability distriution of usage scenarios".

The development of the Cleanroom methodology was not a reaction to the DeMillo, Lipton and Perlis critique, but it inadvertently addresses the issue that the critique raises, that is the lack of social processes required for checking the correctness of a proof. Convincing colleagues that a software component matches its specification in the Cleanroom methodology may be as tedious as constructing a formal mathematical proof about non-trivial algorithm. However, the social structure needed to make the proof trustworthy is recreated through business organization. The enlightened managers who choose to follow the Clearnoom method recognize that the time spent while reviewing tedious conformance to the specification will save time and money later.[106].

A different answer to the problem of social processes and program proofs was suggested long before the critique by DeMillo, Lipton and Perlis, in a 1963 manifesto of the Algol research programme by John McCarthy,[107] where he wrote that "one should prove that [a program] meets its specifications, and this proof should be checked by a computer program." McCarthy himself published a paper on computer programs for checking mathematical proofs and such programs later became known as mechanical theorem provers.[108] The word prover might be misleading. A mechanical theorem prover is not expected to derive a proof automatically. This was, and remains to be, an unrealistic expectation. It is merely used to check a proof written by a human in a formal and often exasperatingly detailed way. In other words, "a mechanical theorem prover does not act as an oracle that certifies bewildering arguments for inscrutable reasons but as an implacable skeptic that insists on all assumptions being stated and all claims justified."[109]

In the 1950s and the 1960s, the optimism about Artificial Intelligence fueled belief that computers will soon be able to prove complex theorems automatically. The earliest work in this area focused on classic mathematical theorems. The Logic Theorist, created by Allen Newell and Herbert Simon in the 1950s,[110] used heuristics to automatically derive mathematical proofs and was able to prove 38 of the first 52 theorems in chapter 2 of the Principia Mathematica. The work on provers for mathematical theorems prepared the ground for research on theorem provers working with proofs about programs. The aim of the Boyer–Moore theorem prover, developed in 1971 in Edinburgh, was to automatically prove properties of LISP programs. Yet, the prover was not able to complete non-trivial proofs without a stroke of luck or detailed human guidance, but it was a pioneering system that inspired follow-up research. This includes the ACL2 prover, which was developed in the 1990s as an "industry strength"[111] version of the Boyer-Moore prover. The ACL2 was mentioned in Chapter 1, because a conference on this system featured a talk about verifying financial systems with a reference to the Knight-Capital bug.

An influential theorem prover, that was designed for checking proofs about programs was the LCF prover introduced by Robin Milner. LCF was motivated by the aim to prove the correctness of programs such as a compilers.[112] To simplify construction of LCF proofs, Milner's group developed a meta-language (ML), which could be used to write procedures that construct proofs. The idea is that humans should not have to write all steps of the proof. They can, instead, write programs that generate all the steps. The programs to generate proofs are, in turn, checked to be correct using types, a concept that we return to in Chapter 5. This powerful combination gave rise to a family of interactive theorem

provers that are still popular today and the ML language soon took a new form as a standalone programming language that influenced modern languages such as OCaml and F#.

Systems that would automatically prove the correctness of a given program still remain elusive today, but work on theorem provers and other program verification tools remains an active field. On the one hand, semi-automatic mechanical theorem provers are able to do more work automatically. On the other hand, fully automatic tools are able to detect an increasing number of potential bugs. In both cases, the developments rely as much on theoretical innovations as on large-scale engineering efforts. A crucial theoretical development that enabled numerous modern verification tools is separation logic, which appeared in the early 2000s and extended Hoare triples with a mechanism for reasoning about data structures.[113] Practical tools that implement the idea, such as the Infer Static Analyzer developed by Facebook, are complex systems. For example, the authors of Infer describe it as a "general analysis framework which is an interface to the modular analysis engine which can be used by other kinds of program analyses" with instantiations "for security, concurrency and in other domains"[114]

The critique of proofs about programs by DeMillo, Lipton and Perlis came at an interesting point in the history of the mathematical culture of programming. It received a heated response, perhaps because it questioned the central assumption of the culture that computer programs are mathematical entities like any other. At the same time, this is something that many were about to realize anyway. Proofs about programs done using a chalk on a blackboard are educational for simple programs, but cannot be used to prove properties of large-scale software. The mathematical culture did not change how it thinks about proofs about programs in response to the critique, yet a significant change was already in the process of happening. We can see the two developments just discussed as influences of two other cultures.

The Cleanroom development methodology combines a managerial approach, of constructing a suitable business structure, with a mathematical idea of rigorous program analysis. Its originator, Harlan Mills, also illustrates that individuals can bridge multiple cultures of programming. As an academic with PhD in mathematics, he was well acquainted with the mathematical culture. As an IBM research fellow, he had close industrial links and his work on development methodologies was a contribution to the managerial culture.

While the typical approach of the managerial culture is to devise an appropriate organizational structure, the typical approach of the engineering culture is to develop a new tool or a method. Mechanized theorem provers thus arise from a combination of mathematical and engineering cultures of programming. Mechanized proofs are no longer the complex human constructs that mathematicians know as proofs. They are computer representations of sequences of propositions where the prover can derive each proposition from earlier sentences, axioms or assumptions. This notion is close to the notion of proof in logic, with the caveat that we are now talking about computer representations. The mathematical culture of programming gradually adopted mechanized proofs as the hallmark of mathematical computer science research. In doing so, the mathematical culture of programming began diverging from its pure mathematical origins where many mathematicians remain cautious about the use of computers in proofs.[115]

The lack of social processes and the complexity of definitions were not the only challenges for proofs of programs that the mathematical culture of programming had to face. Another difference between proofs about programs and proofs in mathematics led to an equally heated debate in the mathematical culture of programming a decade later.

Figure 2.7: Air Force technicians work at tracking monitors in the Tactical Operations Room on the Ballistic Missile Early Warning System site.[118]

Fundamental Limits of Program Proofs

By the end of 1980s, the techniques and tools of the mathematical culture became powerful enough to prove properties of non-trivial systems, including both computer programs and hardware devices. By this point, the use of mechanical theorem provers became commonplace. What may still have looked like an influence of the engineering culture a decade earlier became fully integrated into the mathematical culture. A subsequent version of the Boyer–Moore theorem prover mentioned above was, for example, used for verifying properties of a small multitasking operating system kernel, a low-level assembly language and aspects of a microcoded CPU.[116]

Interestingly, the adoption of mechanical theorem provers and their use for proving properties of software and even hardware with real-world effects did not trigger an immediate reflection on the nature on program proofs. Many still saw a program as a mathematical entity and a program proof as a formal mathematical proof like any other. Yet, some practitioners started to realize that there are limits to program proofs. A large verification effort that illustrates this was the Software Implemented Fault Tolerance (SIFT) system, which was an experimental fault-tolerant flight control system. Boyer and Moore, who worked on the verification of a small but critical part of the system, pointed out that proof could only be completed if they also had a precise specification of how the underlying microprocessor executes instructions and also the timing of such execution.[117] The timing of execution, in particular, is a property of the real hardware that a mathematical model used of a microprocessor is likely to ignore. In some cases, such details may not even be fully deterministic. The timing can, for example, depend on the temperature. The issue, in short, is that programs are not just mathematical theorems, but are executed on real hardware.

A similarly motivated critical reflection was written by Brian Cantwell Smith, who worked at Xerox PARC and was a founder of Computer Professionals for Social Responsibility, an organization opposing the use of computers in warfare. Smith[119] starts with a story of the

American Ballistic Missile Early-Warning System that mistook the rising moon for a ballistic missile attack in 1960 and asks whether such system, if it was formally verified, should be trusted with launching a counter-attack. According to Smith, the answer is no, because "there are inherent limitations to what can be proven about computers and computer programs". The core of his argument is that computer programs always work with a model of the real world, which inevitably ignores some aspects of reality. Computer programs "are not, as some theoreticians seem to suppose, pure mathematical abstractions, living in a pure detached heaven." They trigger actions in the real world that escape the abstractions of the model.

Despite such limitations, Smith still finds formal verification valuable, just for a different reason. A proof of correctness is really a proof of the compatibility between two formal objects: program and specification. This is still useful, because a program and a specification are typically expressed in a different way. For example, a specification for a refrigerator will say that it is a device which maintains an internal temperature between 3 and 6 degrees. A program controlling a refrigerator has to specify how exactly this is done. Proving that a program satisfies a specification is thus a useful added assurance, even if it cannot guarantee anything about an actual refrigerator.

The observation by Boyer and Moore was published as part of an 600-page long interim report submitted to NASA[120], which was the project funder, while the critical reflection by Smith was initially published as a technical report. Consequently, neither of the two triggered a heated debate in the community.

The controversy had to wait for a later paper by a philosopher James Fetzer, which was published by the Communications of the ACM.[121] Fetzer became aware of the critique of program proofs by De Millo, Lipton and Perlis. He thought that the lack of social processes was not a major issue, but believed that there is a more fundamental problem, the one that Boyer, Moore and Smith all alluded to earlier. Although both programs and theorems about them are syntactic structures, a program is not a pure mathematical entity. The meaning of a program is that it controls a physical machine. Consequently, as Fetzer claimed, the very idea of program verification is nonsensical, because it confuses mathematical a priori knowledge with empirical a posteriori knowledge.

According to Fetzer, program verification can mean two things depending on how we treat the nature of the machine that executes the program. If we see the machine as an abstract entity then program verification is a legitimate mathematical activity, but the proof is about a mathematical model, rather than the actual running programs. If we see the machine as an empirical description of a physical system, then we obtain possibly useful, more or less reliable, empirical knowledge about a program, but not a mathematical proof.

Fetzer's critique provoked a fierce response from the verification community. In a letter to the editors of Communications of the ACM, ten prominent computer scientists claimed that his paper "is not a serious scientific analysis of the nature of verification" and accused the editors of Communications of the ACM for "abrogat[ing] their responsibility, to both the ACM membership and to the public at large" by publishing "the ill-informed, irresponsible, and dangerous article." The response does not, in fact, object to the main point raised by Fetzer. It argues that Fetzer attacks a parody of formal verification when he assumes that "the purpose of program verification is to provide an absolute guarantee of correctness with respect to the execution of a program on computer hardware" and the authors give a number of references "in which the limits of proofs are carefully drawn."

One of the practitioners who was very well aware of this issue was Avra Cohn, who worked as part of the team formally verifying the VIPER microprocessor. VIPER has been developed at Royal Signals and Radar Establishment, a research establishment within the UK Ministry of Defence. It was advertised as "the first commercially available microprocessor with both a formal specification and a proof that the chip conforms to it".[122] To Cohn, this advertisement was dangerously misleading. In her reflections on the work, she carefully points out that "a device can be described in a formal way, and the description verified; but (...) there is no way to assure the accuracy of the description. Indeed, any description is bound to be inaccurate in some respects, since it cannot be hoped to mirror an entire physical situation".[123] In other words, even if we formally verify a system at all its levels, ranging from the low level chip to high-level programming language, we are still only working with abstract models. Again, this does not make formal verification useless in practice. It just means that it should not have a unique status among other methods for building correct systems.

As with the critique by DeMillo, Lipton and Perlis, the paper by Fetzer identified an issue that was, perhaps reluctantly or subconsciously recognized by at least some members of the mathematical culture of programming. According to one commenter who, nevertheless, accuses Fetzer of doing "a disservice to the cause of the advancement of the science of programming",[124] the idea that no deductive argument can guarantee that a system will work perfectly is a fact obvious to practically everyone.

The likely reason for the raging controversy caused by the critique is that it attacked the idea that programs are mathematical entities, an assumption that was still at the core of the mathematical culture of programming. In author's response published in a later issue of the Communications of the ACM, Fetzer tries to support his argument that the limitations are not obvious to practically everyone in the community. He does this by quoting a number of key papers in the field, such as Hoare, who states that "programming is an exact science in that all the properties of a program and all the consequences of executing if can, in principle, be found out from the text of the program itself by means of purely deductive reasoning."[125] As Fetzer points out, such phrasing is clearly in contrast with the idea that limitations of formal verification are obvious to practically everyone.

There are two possible explanations to this discrepancy and I believe that both are a part of the truth. First, the ideas about program proofs in the mathematical culture have evolved over the 20 years since Hoare's pioneering work, quoted by Fetzer, appeared. Mechanical theorem provers that draw from the engineering culture did not just provide new tools, but they also imported a more engineering-oriented thinking into the mathematical culture of programming. Second, the idea that programs are mathematical entities was and remains an accepted, but idealized assumption. It provides justification for the formal approach, but does not shape how it is practically done. As such, it is more a rhetorical device, used likely without elaborate thinking about it.

Proofs for Machines and Proofs for Humans

The two contentious debates that I just discussed share a number of characteristics. In both cases, they point to an issue that is related to core assumptions about what programs are. They also make explicit an issue that those working within the mathematical culture do not publicly acknowledge and, perhaps, are not even consciously aware of. At the same time, both of the critiques came at a time when the mathematical culture was already

transforming how it works. In both cases, the transformation was adopting approaches from another culture of programming. In the case of DeMillo, Lipton and Perlis, two ways of replacing the usual social processes of mathematical proofs appeared. The short-lived Cleanroom methodology creates an explicit social process through a managerial structure. The now widely accepted engineering approach based on mechanized theorem proving replaces human checking with machine checking.

In the case of Fetzer, the complications with proving properties of software that controls physical systems were already becoming clear. A few observers started writing about those problems before Fetzer, although in a more reserved way. At the same time, in 1980s, two subcultures were already appearing within the mathematical culture. Those who wanted to produce as complete proofs as possible treated the problem of program proofs increasingly as an engineering issue. Those who wanted to gain pure mathematical insights treated the problem as a formal mathematical one. This required working with minimal, formally tractable models and it became cornerstone of programming language theory research.

The first development started with the use of mechanical theorem provers, which are essential for ensuring that all cases in a long and tedious proof about complex system are covered. A proof constructed using a mechanical theorem prover is really a part of the program. Just like an ordinary mathematical proofs are written to convince another human that it would be, in principle, possible to construct a finnite sequence of propositions representing the formal proof, a mechanized proof is essentially a program that instructs a machine to construct a sequence of (computer representations of) propositions. Over time, the specification of a correctness property and its proof displaced the program itself as the main object that needs to be created. In modern theorem provers, the programmer only needs to write their proofs and the executable code of a corresponding program is generated based on the source code of the proof. A good recent example using this method is the formally verified CompCert compiler,[126] where a single program represents both the compiler and its correctness proof. In such systems, the proof becomes a technical artifact that is clearly distinct from a formal deductive mathematical proof.

The second development remained more faithful to traditional mathematics. It uses severely simplified mathematical models of programming languages or hardware devices. It is clear that proofs about such models do not provide guarantees about the reality. However, it makes is possible to reason about formal mathematical programs using standard mathematical methods, such as a hand-written proof on a blackboard. This way of working does not guarantee correctness of an actual system that is modelled, but the model may reveal tricky corner-cases that are equally tricky for the real implementation. A failure to prove correctness often points to an issue that also exists in the actual system and can be empirically observed.

The idea of using severely simplified mathematical models in proofs about programs and programming langauges appeared well before the 1980s. It dates back to McCarthy's 1964 Microalgol, which is "a very small subset of Algol" where "all the difficult aspects of Algol are eliminated."[127] Although the work on small mathematical models is rooted in the same mathematical culture as the work resulting in complex mechanized proofs, the language and presentation used in talking about them has gradually became more distinct, possibly highlighting the fact that the object of study is a mathematical object rather than an actual program or a programming language.

To follow the evolution, we can look at a number of small model languages for studying problems related to concurrent programming, i.e. programming that involves multiple processes executing concurrently and communicating, which appeared between the 1970s and the 1990s. The Communicating Sequential Processes model, developed in 1978[128] is defined in a way inspired by the Algol 60 specification. The paper discusses existing programming languages that support concurrency and makes references to potential implementation. A model that appeared just two years later, A Calculus of Communicating Systems[129] is still compared with existing programming languages, but it is presented as a "calculus" rather than a "language". The paper also uses a more mathematical format of definitions. Even more recent model, the π-calculus developed in 1992[130] is presented as a fully formal mathematical model, more akin to the λ-calculus developed in formal logic than to a programming language.

The development of two subcultures with two kinds of proofs was not a response to the Fetzer's critique. It was clearly already happening when the critique was published. Yet, it shows that the issue at the heart of Fetzer's critique is something that has been shaping the mathematical culture. Again, the developments arguably benefited from the pluralism of the field of programming. One of the two directions was influenced by the engineering culture, whereas the other adopted a more purist mathematical approach.

Mathematization of Programming

To a contemporary computer scientist, the general idea of using mathematical methods for the analysis of programs and their correctness seems unproblematic. They are comfortable using the many diverse concrete approaches that emerged from the mathematical culture over time. Their only lament is usually that the use of mathematical mehtods in software industry is not more commonplace. Despite their complicated history, mechanical theorem provers and code analysis tools like the Facebook Infer Static Analyzer are seen as ways of bringing rigorous mathematical methods to practice. Many academic papers and grant applications that focus on developing new mathematical theories of programs and new tools motivate the work by claiming that only rigorous mathematical approach to programming can guarantee that software will work correctly. As we have seen in this chapter, this way of thinking about programming and its history is not as straightforward as it may seem. It is the result of a long and involved process. Over time, developments within the mathematical culture of programming as well as interactions with other cultures reshaped what such "mathematical methods" mean.

Computers were always closely affiliated with mathematical thinking. Many computer pioneers had mathematical background and the early computers were used to solve mathematical problems. Yet, programming itself started more as a craft learned through practice than as a structured mathematical activity. Those involved with computing started looking for better ways of programming, but this meant very different things to different people. Hackers were looking for better tools to control the machine, managers were looking for better ways of organizing software production and mathematicians started to think about programming in terms of algorithms. The three cultures, almost miraculously, came together in late 1950s and collectively established the idea of a programming language as an independent entity. For mathematicians, the language was a formal mathematical object, which also made it possible to think of programs, written as concrete source code, as formal entities.

Despite coming together around the concept of programming languages, the different cultures did not integrate to develop a single unified way of thinking about programming. Programming was a pluralistic discipline before the 1950s and remains a pluralistic discipline today. The several different cultures that I identify in this book each evolve, but they keep a surprisingly stable set of core asssumptions and methods. For example, the hacker culture that we encountered in the early days of programming emphasizes thorough understanding of and direct engagement with computers. It relies on a kind of knowledge that is learned through practice and is hard to write down. The culture did not disappear once programming languages and methodologies inspried by structured programming gave programming more rigid structure. As we will see in Chapter 3, it quickly reappeared once computers became more interactive and it was again possible to work with a machine directly and acquire craft-like skills.

The story of the mathematization of programming that I followed in this chapter is primarily a story of the mathematical culture of programming, its development and its interactions with other cultures. Many such interactions are productive and provide a useful impetus for moving forward in a new direction.

Two interactions that I wrote about involved concrete technical ideas that were either shared by multiple cultures or that crossed the boundary between cultures. The brith of programming languages was motivated by managerial needs and relied on previous hacker knowledge. Yet, it gave the mathematical culture a new way of looking at programs and made it possible to treat them as formal entities written in a formal language akin to that used by logicians. The development of structured programming was motivated by aim of writing better programs and is perhaps best aligned with what I call the engineering culture, but it was also closely connected with the development of compositional reasoning methods in the mathematical culture. However, the idea also inspired new ways of managing the software development process.

Another kind of interaction occurs when one culture borrows general ideas or methods from another culture. The two times this happened in the story told here, the borrowing took place at a time of controversial debates, sparked by the questioning of basic assumptions of the mathematical culture. When DeMillo, Lipton and Perlis pointed out that proofs of programs were different from mathematical proofs and lacked the necessary social processes, the mathematical culture was already in the process of adopting mechanical theorem provers that replace human checking with machine checking. This move took a more engineering perspective on the nature of mathematical proofs and I suggest that it resulted from the borrowing of methods from the engineering culture. The fact that the developments happend at the same time as the controversy may be just a coincidence, but it is also possible that both were a reaction to problems in the mathematical culture that were slowly emerging on the horizon. Traditional mathematical ways of writing proofs worked for small programs, but it would not work for more complex software. This triggered reflection in the community and it also motivated its members to look for ideas outside of their own culture.

The history of mathematization of programming and the controversies that we encountered in this chapter has been told in detail in a number of excellent accounts that I have been drawing on in this chapter.[131] What is new in my retelling of the story is the focus on pluralism within the discipline. If we look at how concepts like programming language, structured programming or mechanized proofs came to be, we can see that this is a product of rich interactions between different groups of individuals who have different

assumptions and different methods of working.[132] Although different cultures of programming do not share assumptions and use different methods, they do not form incommensurable bodies of knowledge. They can exchange idea through individuals who sometimes pass between multiple cultures, but they can also exchange ideas through shared technical concepts such as programming languages, programming constructs and sometimes shared programming practices.

Most of the developments discussed in this chapter were rooted in academic computer science. They were motivated by real-world problems and shaped the practice of programming, but remained, to a large extent, of interest primarily to academics. The development that I follow in the next chapter also emerged from the academic environment, also keeps a strong independent identity and also resulted in innumerable practical applications, but it looks at programming from a completely different angle.

Notes

1. Dijkstra (1972)
2. The establishment of algorithm as a basic concept of computer science is documented by Ceruzzi (1988) and further discussed by Ensmenger (2012), while Mahoney (1992) discusses search for mathematical foundations of computing.
3. Ensmenger (2012) provides a detailed account of the 1950s labour crisis.
4. Documented in various contexts by Hicks (2010), Ensmenger (2010) and Abbate (2012)
5. The external form allowed programmers to use relative addressing (See Wilkes et al. (1951)), while interpretive routines made it possible to use pseudo-instructions, e.g., for working with floating-point numbers (See Campbell-Kelly (1980)).
6. Mahoney (1997)
7. Tedre (2014) documents how the birth of computers created a different kinds of problems for mathematical study of computation.
8. We return to the media-theoretic perspective in Chapter 3. The perspective seeing compilation as a translation process is documented by Nofre et al. (2014).
9. Ensmenger (2012); Priestley (2011)
10. We return to the topic of interacting with a programming environment in Chapter 3. For more on Commodore 64 BASIC, see also `http://tomasp.net/commodore64`, Retrieved 5 August, 2022.
11. The theorem refers to the result by Böhm and Jacopini (1966). As noted by Harel (1980), the result in the paper is of a more technical nature and has been misinterpreted in the same way as done here by *Tau*.
12. MacKenzie (2004)
13. As quoted by MacKenzie (2004), Edsger Dijkstra, the originator of the term structured programming said that "Since IBM stole the term 'structured programming' I don't use it anymore myself."
14. See DeMillo et al. (1977). A detailed account of the history of program proofs can be found in the work of MacKenzie (2001)
15. We return to this topic when discussing the "Cleanroom" methodology later in this chapter.
16. The history of ENIAC programming has been documented by historians, especially Haigh et al. (2016), and also by Bartik (2013) herself. There are disputes about some aspects of the history of the early ENIAC programming work. I generally follow the first-hand account of Jean Jennings Bartik, but Haigh et al. (2016) provide a more neutral perspective.
17. Ensmenger (2012)
18. John Backus, quoted by Ensmenger (2012)
19. Priestley (2011), p.254
20. Wilkes (1985)
21. Ensmenger (2012), p.67
22. Polanyi (1958)
23. Documented by Ensmenger (2012)
24. Ensmenger (2012)
25. Dijkstra (1972)
26. Ensmenger (2012), p.121

27. Knuth (1968)
28. Knuth (1968)
29. Here, the field of programming follows the suit of physics, whose preference for theory over experimental work has been documented by Hacking (1983), p.150.
30. Many of those clashes and struggles are documented by Ensmenger (2012)
31. Univac (1957)
32. The early history of programming systems has been documented by Priestley (2011). The birth of programming languages is told by Nofre et al. (2014), in a paper whose title inspired the heading of this section.
33. Nofre et al. (2014)
34. Backus et al. (1954)
35. Cited in Nofre et al. (2014)
36. Nofre et al. (2014)
37. Quoted by Nofre et al. (2014)
38. The preliminary (International Algebraic Language) report by Perlis and Samelson (1958) and the final report (Algol 60) by Backus et al. (1960) are the primary sources. The development of Algol has been documented in a first-hand account edited by Perlis (1978) and Naur (1978)
39. Star and Griesemer (1989)
40. Perlis (1978)
41. First described by Bosak et al. (1962). First-hand account of the history has been written by Sammet (1978)
42. First-hand account of the history has been written by McCarthy (1978).
43. Priestley (2017)
44. Backus et al. (1960)
45. Backus et al. (1963)
46. Priestley (2011) refers to research paradigms in the sense of Kuhn (1962), although with some caution, because the Algol paradigm exists in parallel with other ways of thinking and participants can exchange ideas with others. This is what I attempt to capture with the notion of “culture of programming” in this book. Many of the pioneering works of the Algol paradigm can be found in a valuable collection by Colburn et al. (2012).
47. McCarthy (1963)
48. Ensmenger (2012)
49. Habermann (1969)
50. Naur (1966)
51. Floyd (1967)
52. Hoare (1969)
53. A first-hand account of the history can be found in Hoare (1981).
54. Hoare (1971)
55. McCarthy (1963)
56. Conger (1962)
57. Hoare (1971)
58. In this section, I rely on the detailed account of the history of formal descriptions of Algol written by Jones and Astarte (2016); Astarte and Jones (2018)
59. Backus et al. (1963)
60. Lakatos (1976)
61. McCarthy (1964)
62. Hoare (1969)
63. Mosses (1974)
64. Christopher P. Wadsworth, cited in Astarte and Jones (2018)
65. Lauer (1968)
66. Allen et al. (1972)
67. Henhapl and Jones (1978)
68. As reported by Astarte and Jones (2018), Mosses was working on a tool that would generate compilers from formal language specifications. His Algol description was written in a style used by the tool, but he never actually used the Algol description as input for his system.
69. Lindsey (1996a)
70. Hoare (1981)
71. Haigh (2010)

72. Edited by Gries (1978b)
73. Ensmenger (2012)
74. Haigh (2010)
75. Pelaez Valdez (1988)
76. Buxton et al. (1970)
77. Dijkstra (1968)
78. The working paper appeared in Buxton et al. (1970) and in a longer, privately circulated version Dijkstra (1970). Author's later reflections on the history appear in Dijkstra (2002).
79. Böhm and Jacopini (1966)
80. Harel (1980)
81. Plauger (1993)
82. See also historical reflections on the `goto` statement by Leavenworth (1972).
83. Ensmenger (2012), p.198
84. Ensmenger (2012), p.199,209
85. Reported in the Datamation magazine by Baker and Mills (1973)
86. Dijkstra (1980)
87. Dijkstra (1988)
88. McCracken (1973), Available at `https://archive.org/details/bitsavers_datamation_34111538`, Retrieved 12 August 2022
89. Discussed by Slayton (2013), p.166
90. Using the concept introduced by Galison (1997)
91. Hacking (1983)
92. `https://amturing.acm.org/award_winners/perlis_0132439.cfm`, Retrieved 12 August, 2022
93. Naur et al. (1969)
94. Buxton et al. (1970)
95. Documented by MacKenzie (2004)
96. DeMillo et al. (1977)
97. Lakatos (1976)
98. DeMillo et al. (1977)
99. De Millo et al. (1979)
100. Discussed by MacKenzie (2004)
101. Quoted by MacKenzie (2004)
102. Dijkstra (1978)
103. MacKenzie (2004)
104. Mills et al. (1987)
105. Dyer and Mills (1981)
106. The methodology has been extensively discussed by MacKenzie (2004)
107. McCarthy (1963)
108. McCarthy (1962)
109. Rushby and Von Henke (1993), quoted byMacKenzie (2004),p.272
110. Newell and Simon (1956)
111. Moore (2019)
112. The system was first presented by Milner (1972); work on proofs is discussed later in Milner (1979).
113. A first-hand account on the development has been written by Pym et al. (2019)
114. `https://fbinfer.com/docs/about-Infer`, Retrieved 13 August 2022
115. For mathematician's reflections on proofs, see for example the collection by Gold and Simons (2008)
116. The applications are discussed by Boyer and Moore (1988), p.9.
117. Discussed by MacKenzie (2004), p.19
118. `https://commons.wikimedia.org/wiki/File:BMEWS_Tac_Ops_Room.jpg`, Retrieved 14 August 2022
119. Smith (1985)
120. Boyer and Moore (1983)
121. Fetzer (1988)
122. Quoted in MacKenzie (2001), p. 248
123. Cohn (1989)
124. Pleasant (1989)
125. Hoare (1969)
126. Leroy (2009)

127. McCarthy (1964)
128. Hoare (1978)
129. Milner (1980)
130. Milner et al. (1992)
131. Especially the history of early programming documented by Priestley (2011), computer personnel and power stuggles by Ensmenger (2012), history of proofs about programs by MacKenzie (2004), history of software engineering as seen through millitary software by Slayton (2013) and the history of the programming discipline by Tedre (2014). The book edited by Colburn et al. (2012) collects many of the papers involved in critical reflections on formal program verification and proofs of programs.
132. I use the term *cultures of programming* to tell the story, but the concept is similar to the idea of *systems of practice* introduced by Chang (2012) when analyzing the history of chemistry.

Chapter 3

Interactive Programming

Teacher: In the last chapter, we saw what becomes possible when we view programming as a formal matehamtical activity. Let's see if talking about interactive programming reveals another perspective!

Omega: Wait, what does interactive programming even mean? Programming langauges and algorithms are established topics that everyone understands, but having interactive programming as a topic on par with the mathematization of programming is very odd.

Epsilon: I can imagine many forms of interactive programming. Look at programming systems like Smalltalk, microcomputers like Commodore 64 or notebooks for data science. Even modern programming environments for Java are interactive in some sense! Although, I do not see how those are connected.

Tau: You are confusing things. Are you talking about programming languages, computers or software applications? If we want to talk about programming, we should think about Smalltalk, BASIC, Python or Java. The come with various editors and tools, but languages is what is important.

Gamma: This is a matter of perspective. We are so used to thinking about programming languages that any view which puts interaction first sounds odd. I think this is wrong. Interactivity can completely change how you program, regardless of the langauge that is behind it. I for one am very glad to have this chatper.

Teacher: Let's go ahead then. *Epsilon* already mentioned some well-known examples like Smalltalk, but where should we start from? How about the 1969 demo, retroscpectively known as "The Mother of All Demos" by Douglas Engelbart where he presented his oN-Line System (NLS) that featured mouse, video-conferencing, hypertext and many other aspects of modern interactive computers?

Epsilon: This is a great reference, but Engelbart's demo was mainly about using the system, not about programming it.

Alpha: Not only that, but it is also starting way too late! The first real interactive programming was done at MIT once the "hackers" from the Tech Model Railroad Club got their hands on the TX-0 computer that was loaned to the MIT Research Laboratory of Electronics in 1958.[1] This was the first time you could program a computer interactively through a terminal.

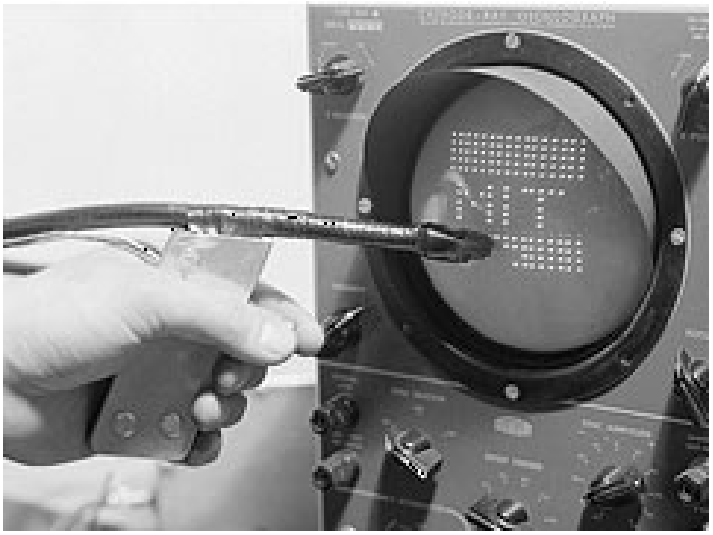


Figure 3.1: Drawing on the Whirlwind cathode-ray tube screen suing the light pen

Omega: Mind you, TX-0 was an experimental machine to test the transistor technology for the SAGE defence system. The Whirlwind computer that was used at SAGE before TX-0 was built was already interactive.

Gamma: Well, Whirlwind did give us the light pen (Figure 3.1), an input device used in the revolutionary graphical programming system Sketchpad in 1963. But Whirlwind itself was programmed, like most other computers at the time, by feeding it instructions on a punched tape that you produced after carefuly working out the program on paper.[2]

Teacher: So, shall we conclude that the TX-0 machine and the PDP-1 that soon followed is the place where interactive programming started?

Omega: That may be the case, but only if you want to call the toying with TX-0 and PDP-1 "programming". To me, this is not real programming. It does not scale. You would never be able to build a large-scale software system if you did it at the console and needed that expensive dedicated computer for each programmer doing the work! The reason why batch processing was the only way to program in the 1960s was that you could do all the thinking away from the machine and utilize the expensive hardware more efficiently.

Tau: Yes, but this was soon solved with time-sharing, which made it possible to connect multiple users to a single machine and run their programs concurrently. You could already do this at the start of the 1960s and the LISP programming language soon started to be used in the interactive way on time-sharing systems.

Teacher: This raises the question whether the rise of interactive programming allowed new ways of using computers?

Epsilon: It made programming easier, because you could get more rapid feedback on whether your code is correct or not and support the programming activity through various interactive tools, but I would not call that a new way of using a computer...

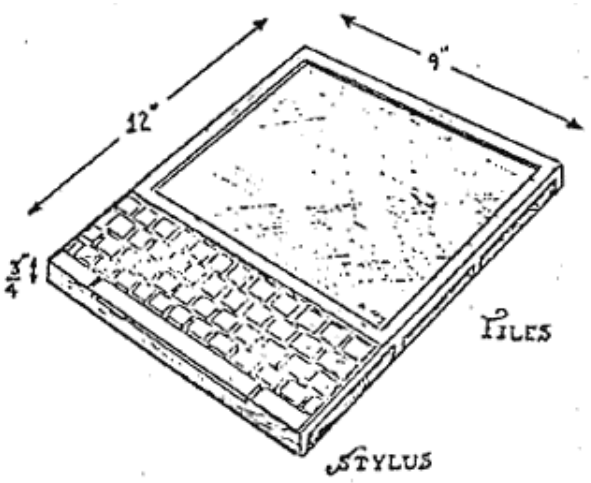


Figure 3.2: A sketch of "Dynabook" from Alan Kay's 1972 paper "A personal computer for children of all ages" Kay (1972)

Gamma: Oh, it absolutely did allow a completely new way of using computers! The immediate feedback is crucial for creative use of computers and also in education.[3] A good example is the LOGO programming language that was designed for children and allowed them to programmatically control a turtle and draw on a computer screen. This kind of learning can only work if you can write code interactively and immediately see what the behaviour is.

Teacher: I think we are now getting back to the time of the oN-Line System demo in the 1969. Even if that was not about programming, did it have any influence on interactive programming?

Gamma: First of all, it is important to see Engelbart as part of a bigger community. His work was partly funded by the ARPA Information Processing Techniques Office (IPTO) that developed a community around the vision of "augmenting human intellect" and he was also at least connected to the West coast counterculture movement, whose experiments with LSD used a very similar terminology of augmenting the human mind and consciousness.[4]

Alpha: By the way, the vision of "augmenting human intellect" also shaped earlier work on interactive programming with LISP,[5] but let's focus on later developments…

Epsilon: Undoubtedly the most influential work on interactive programming that emerged from this community was Smalltalk, which pioneered many ideas of object-oriented programming and was used through a graphical programming environment that looked much like the Integrated Development Environments (IDEs) of modern programming languages like Java and C# today.

Gamma: Ah, well, your way of looking at Smalltalk is shaped by present-day engineering view of programming, but that is not how it was thought about at the time. Smalltalk was a part of vision of a "personal computer for children of all ages"[6] (Figure 3.2) that would allow children to experience the excitement of thought and creation!

Omega: That is the sort of vision you could expect from the 1960s West-coast counter-culture, but how does that help solving real computing problems like modelling the resource requirements of a hospital that a decision theorist may want to program?[7]

Gamma: The whole point of Smalltalk was to serve as a general purpose meta-medium that anyone can adapt to their particular needs. A kid may turn it into a painting system, a musician may turn it into an audio synthesis system and a decision theorist can turn it into an agent-based simulation system...

Teacher: If Smalltalk was so powerful, how come we do not all program in Smalltalk today?

Omega: In practice, you do not really need a "meta-medium". You just need a good special-purpose software for building simulations, rather than something that you first need to laboriously turn into a system for doing the same thing. I think special-purpose software just always ends up being better.

Gamma: The Smalltalk vision was just too far ahead and it still is. We may have tablets that look like the Dynabook, but they are still not programmable in the way envisioned at the time. You still cannot reprogram the iPad from within itself and share your creations with others!

Alpha: Well, realising this vision is not in the commercial interest of Apple, just like it was not in the commercial interest of Xerox. To the management, the work on Smalltalk at Xerox PARC, had to be presented as the vision for the "office of the future".

Omega: What is wrong with that? Yes, Xerox never managed to turn the research into a commercial success, but the Xerox Star system that they eventually built in the 1980s (Figure 3.3) gave you something that was remarkably close to computer systems that office workers used in the 2000s.

Gamma: The problem is that Xerox Star removed all that made Smalltalk interesting! It looks the same on the surface, but you are now a mere user of something that is given to you. You can no longer shape the system to suit your needs. You cannot learn how it works and make it better.

Epsilon: The whole Xerox Star system failed not just for business reasons, but also for technical resons. It was cursed by "biggerism", a growing and unnecessary complexity at both the hardware and software level that cursed many of the systems at PARC.[8]

Alpha: Yes, the work at Xerox PARC was a source of many good ideas, but it became less and less interesting over time as it grew complex, relied on increasingly obsolete hardware and was too disconnected from the underlying machine.

Teacher: What was the next step for interactive programming then? Did Smalltalk influence other interactive programming systems?

Epsilon: It took some time, but Smalltalk had a lot of influence on later object-oriented programming systems, as well as software engineering methodologies. You can also still use Smalltalk today through modern and more efficient implementations...

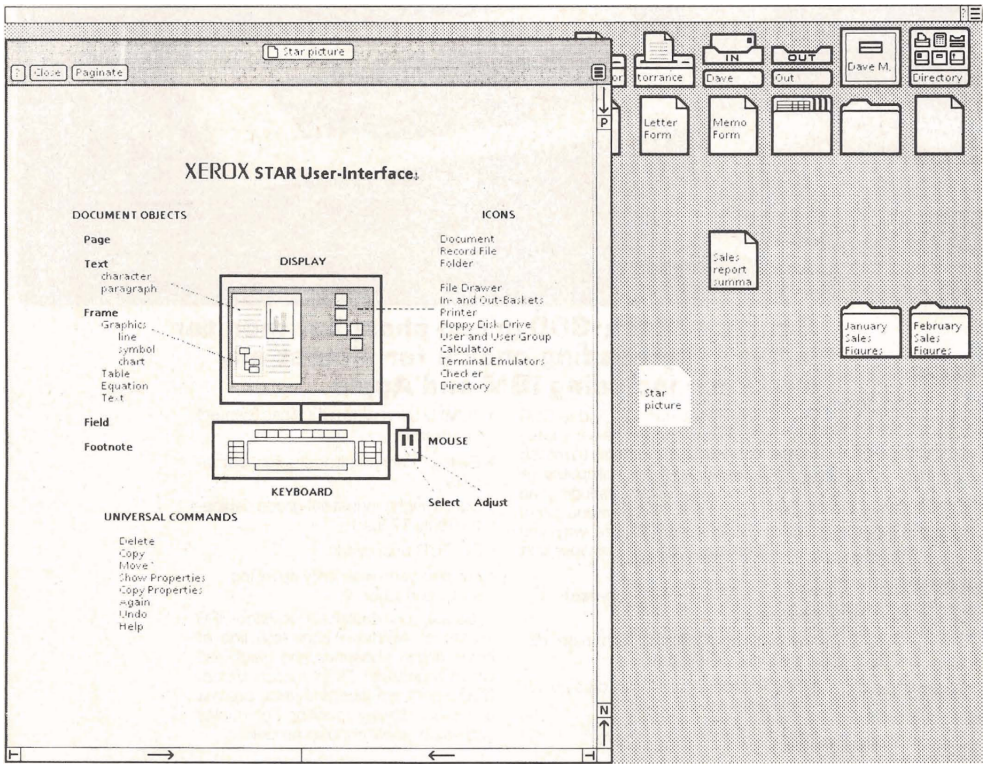


Figure 3.3: Xerox Star desktop showing an open document on the left and several commonly used "icons" on the right (from Kimball and Harslem (1982))

Alpha: Yes, but this is moving away from the idea of interactive programming. The area where interactive programming really took off at the end of the 1970s was programming for microcomputers, a new kind of small and inexpensive computers that became possible thanks to the general-purpose microprocessors that started to be available then.

Omega: You mean toys like the Altair 8800 microcomputer that you got as a kit and first had to assemble it? You had to program them by flipping switches and through teletype terminal like the TX-0! I do not see how this was the next step for interactive programming...

Alpha: That was the starting point, but microcomputers soon attracted a huge community of hobbyists. Because they were so simple, they were also easy to improve. Soon, microcomputers were built and supported by both established and new companies and the "1977 trinity" of computers all had screens and interactive programming environment based on BASIC.[9]

Gamma: This is obviously very far from the graphical interfaces of Xerox systems, but I see the connection. Many microcomputers booted directly into BASIC prompt and so you were doing interactive programming even if you just wanted to load and run a game. Magazines that published source listings for games and other programs further encouraged people to learn how to program.[10]

Tau: It is a shame that microcomputers adopted BASIC and not a sensible programming language like Algol. I agree with Dijkstra[11], who once wrote that BASIC mentally mutilated a generation of programmers beyond hope of regeneration!

Gamma: Say what you want about BASIC, but the simple kind of interactive programming that became popular on microcomputers would not work with a more structured language. BASIC systems on those machines had a simple line-based interface. If you typed a command, it would run it; if you started with a line number, it would edit the program in memory. This way, you could easily enter programs line-by-line, correct errors and insert more code as needed.

Alpha: Exactly. The line-based interface was essentially a structure editor for your code. In a way, it is a new take on the 1960s terminal-based structure editor for LISP.

Omega: I'm glad people soon figured out that you need a more user-friendly operating system and applications you can buy and use without copying the source code from a magazine. Microcomputers did play an important role in this, because they made it commercially viable to sell and use software like VisiCalc, the first spreadsheet application software...

Gamma: Another attept at interactive programming for all ruined by commercial interests!

Teacher: Care to elaborate?

Gamma: It seems to me that more interactive approaches to programming often emerge with new technologies, like TX-0, desktop computers at Xerox and microcomputers. They give users more control over the computers and their programming, which also often relies on sharing code with others. Businesses find this either inefficient or hard to commercialize, so they repeatedly move to a more closed model where programming software is clearly separated from using software.

Teacher: Where do we stand today then? Does modern software development follow a closed software development model?

Epsilon: I guess we are mostly building end-user applications, but I do not think they are always closed. After all, increasing number of software is open-source and developed collaboratively on platforms like GitHub...

Gamma: Even if the code is available, the system does not encourage you to engage with it. If you are looking for interactive programming, you again need to look outside of mainstream commercial programming. One example is live coding where you use interactive programming to control music synthesis and visual effects (Figure 3.4). I find this amazing, because it takes the interactivity of programming to the extreme. You use a programming system as a musical instrument and need to make programming interactive enough so that you can respond to what other musicians are doing.[12]

Omega: What an odd example! I can understand how the past examples eventually resulted in something useful, but I really do not see how this kind of interactive programming can influence anything that commercial programmers need to do.

Figure 3.4: A poster advertising an Algorave event in Kensington, "an evening of alien rhythms and funky visuals by live-coding DJ's."[13]

Epsilon: Maybe... I can think of a few cases where more rapid feedback is useful in the context of ordinary software development. Many programming environments have some form of REPL where you can type code and evaluate it immediately, but people typically use this just in the early prototyping phase. There is also live reloading in many web development frameworks where a web application is automatically updated when you save your code edits. Finally, you could also say that the Test-Driven Development (TDD) methodology is a kind of interactive programming. Again, you can have a tool that monitors your file changes and re-runs relevant tests automatically to give you rapid feedback.

Tau: I think I have a better example. It is also not regular programming, but it is exploring interactivity in a more sensible context. If you look at how data analysts work with data in environments like RStudio or Jupyter, it is also a very interactive kind of programming. You load some data into memory, run various commands to process it, immediately visualize the results and then refine your scripts. I do not see how this would be useful outside of data science, but it makes a lot of sense in that context.

Teacher: Interesting, I did not expect that we would find that many examples of interactive programming in the programming people do today!

Omega: All those individual examples, both past and contemporary, are no doubt interesting, but I remain unconvinced. I think *Alpha* and *Gamma* are right that all the cases are similar. They are often about having more direct access to the computer, getting the computer to assist the programmer and making programming accessible to more people. But I still do not see any connection between the different cases.

Gamma: I think interactive programming is a very basic approach to programming that is maybe not that hard to come up with. It seems that it is often at odds with how computers are used in the commercial context. Perhaps that is why the idea keeps being rediscovered over and over when new technologies emerge!

Interactive Programming

Spacewar!

At the turn of the 1960s, most computer programming was rather dull. You punched your program on a deck of punched cards and handed the deck over to a computer operator. The next day, you would get back a listing with “Program expects a comma. Abort.”[14] because your program had a trivial syntax error. The next day, you would fix your error and try again, without ever getting near the actual computer.

A very different kind of working with computers was taking place in the Building 26 at MIT. By 1961, the building became the home of two computers, TX-0 and PDP-1, which were available for interactive use. You could sit in front of those machines, flip the control switches, enter your programs using a Flexowriter terminal, control the attached Cathode Ray Tube (CRT) screen and even draw on the screen using a light pen, an input device that identifies the screen location at which it is pointing.

The summer before the arrival of the more powerful of the two machines, the PDP-1, to the MIT Electrical Engineering Department in 1961, the staff and students started to think about a way of showcasing the capabilities of the new computer. This inspired a group of current and former MIT students and research assistants to create Spacewar! (Figure 3.5), a game inspired by science fiction novels of E. E. "Doc" Smith’s Lensman series. In Spacewar! two players controlled rockets in space that had a star with gravitational pull in the centre and tried to shoot down the opponent’s rocket using torpedoes.

Spacewar! was conceived by Steve “Slug” Russell, Martin Graetz and Wayne Wiitanen. The three, who mockingly referred to their shared residence at Hingham Street as an Institute established the Higham Institute Study Group on Space Warfare and started discussing the idea further. Russell, who was also a member of the Tech Model Railroad Club (TMRC) discussed the idea with other MIT computer hackers. He hoped someone else would write the game and he’d just play and kept looking for excuses not to do the work himself. He knew he needed sine and cosine routines, but claimed he did not know how to write them. In response, Alan Kotok who was a fellow TMRC member and a TX-0 hacker, drove his brand-new Volkswagen Beetle to the headquarters of DEC, the manufacturer of PDP-1. He got a copy of sine and cosine routines that they provided and said “Okay, Russell, here’s the sine and cosine routine. Now what’s your excuse?”[15]

Eventually, Steve Russell wrote the first version of Spacewar! The game became available to everyone else with access to PDP-1 and, after some debugging and polishing, it bacame popular enough that the lab had to introduce a policy that playing Spacewar! was the lowest-priority thing the computer could be used for.

Spacewar! was hugely influential and was included in the first 10 computer games to be included in the Game canon, a software preservation effort at the Library of Congress.[16] For this chapter, it is interesting for two other reasons. First, Spacewar! was created in-

Figure 3.5: Spacewar! game running on PDP-1 (at the Computer History Museum in 2007)

teractively. Rather than punching the source code on a deck of punch cards, much of the game was created and tweaked while sitting in front of the computer, using the terminal to modify the code and the screen to play and test the game. This may not seem surprising today, but it was surprising in the early 1960s when there were just two room-sized computers in the whole department. To use the machine, you would put your name on a signup sheet. Professors and graduate students had some number of hours they could book, but when noone was using the machine, it became available to unsponsored projects and undergraduate students. This encouraged a culture of late night hacking. Many of the TX-0 and PDP-1 hackers were hanging around the room at night when the machines were in less demand and signed up users sometimes did not show up.

Second, Spacewar! is an early example that illustrates the hacker ethic which emerged at MIT in the 1960s. Its principles encouraged sharing of source code and improving it.[17] If a program was not deemed good enough by a hacker, the hacker was encouraged to make it better. This happened in a number of ways in the case of Spacewar! In the first version, stars were generated randomly. There was also no gravity, because it was not clear how to calcualte it fast enough. Dan Edwards figured out how to make the spaceship rendering faster by generating the instructions to do the drawing for each angle, a kind of just-in-time compilation. This freed enough time to calculate gravity effect on the two spaceships, but not the torpedoes. Pete Samson, who was an astronomy hobbyist, was dissatisfied with the random stars and decided this has to be fixed. He produced a table of roughly 1,000 stars using an actual star map and wrote code to render them efficiently enough on the screen so that they could be included as the background of Spacewar!

The two aspects are typical of the hacker culture of programming. First, the practical hacker knowledge spreads through personal experience rather than through written materials or formal education. Despite using many innovative techniques, including one of the first just-in-time compilers, the authors did not turn any of those into an academic publication. The knowledge spreads through sharing that is encouraged by the hacker ethic, but this also keeps it local and inaccessible to outsiders. Second, hackers value direct engagement with computers. An idea or a beautiful theory is worthless if you cannot actually implement it.

Although Spacewar! is a great example of an interactive program that was also programmed interactively, if we want to trace the origins of the interactive approach to programming, we need to go back a couple more years to the Whirlwind computer, the precursor of TX-0.

Transistorized Experimental Computer Zero

Towards the end of the World War II, the U.S. Military approached the MIT with the problem of designing a flight simulator that could be used to test designs of new airplanes and analyze their aerodynamic stability. In the 1940s, the first-choice approach was to build an analog electromechanical system. This soon proved too slow and inflexible. The person responsible for the project, Jay Forrester, heard about the ENIAC from a colleague at MIT and decided that a fast digital computer would be up for the task. In 1946, the military agreed to fund development of Whirlwind, a new digital computer designed to solve the problem.[18]

Whirlwind became operational in 1951 and was a revolutionary machine in many ways. Its application domain posed a number of new interesting challenges. Instead of completing individual computational tasks in batches, Whirlwind had to be able to accept real-time inputs and produce real-time outputs. It also needed a graphical output system and interactive capabilities. It had a cathode-ray tube (CRT) screen and was later also equipped with a light pen that was used for screen input.

The interactive features of Whirlwind opened the room for new computer applications, most notably the enormous cold war SAGE (Semi-Automatic Ground Environment) air defense system.[19] However, programming the Whirlwind was as tedious as programming other early digital computers. As with the likes of EDSAC, all Whirlwind programming was done away from the machine and debugging mostly involved making sense of lengthy printouts.

Whirlwind used magnetic core memory, a new kind of fast random-access memory, but it was still based on vacuum tubes. To test the use of transistors for a potential future version of the machine the MIT Lincoln Lab, which housed the Whirlwind, started building TX-0, the “transistorized experimental computer zero”. After successful test, the computer was no longer needed and was transferred to the MIT Research Laboratory of Electronics (RLE) on a long-term loan. Here, it became available for use to qualified users, including the members of the Tech Model Railroad Club, without much bureaucracy on a round the clock, seven days a week basis. This arrangement promoted a new more direct kind of computer programming:

> *The idea of working on-line caught on immediately. It was easily demonstrated that a researcher could get a working program off the ground in much less time, and secondly, the output could be mointored so that a useless amount of data was not generated when it was initially apparent that the program was not fully debugged, or that some unforseen event had not been taken into account.*[20]

The new way of working was enabled by the informal access policy, but also by technical characteristics of the machine. The most basic way of manually controlling the TX-0 was through the Toggle Switch Storage (Figure 3.6), which allowed the user to manually set

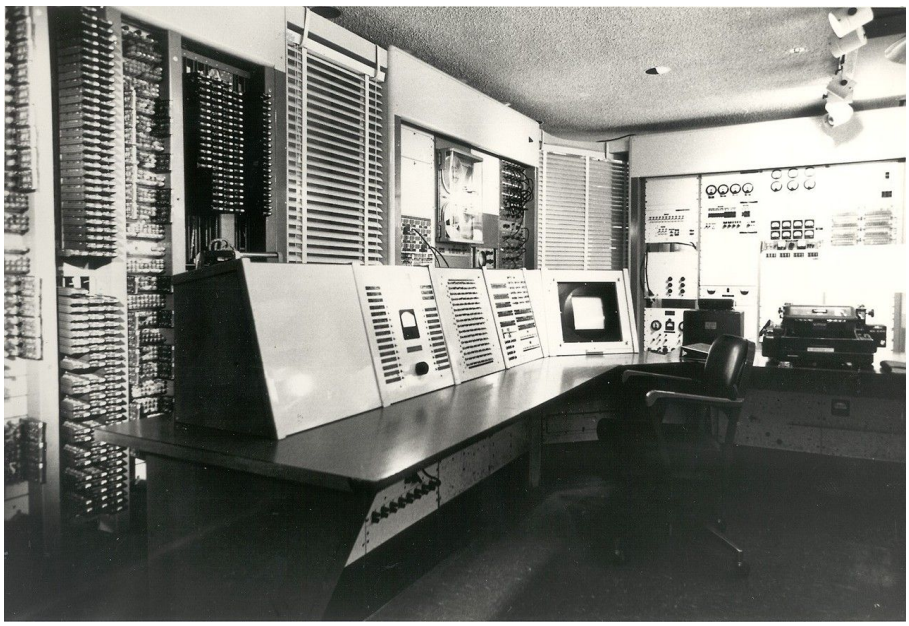


Figure 3.6: TX-0 with the control panel, including the Toggle Switch Storage, 12 1/2" cathode ray oscillososcope display tube and Flexowriter teletype terminal in the back.

the values of sixteen 18-bit registers. However, the high-speed large capacity magnetic core memory allowed much more than this. In the initial setup, a part of the memory was allocated as a secondary storage medium that could contain program in a symbolic format. This could then be transformed into executable instructions by a primitive compiler. The memory was also used to store a range of utility programs that could co-exist with the user program. The most notable one was a program named Utility Tape 3 (UT-3).[21]

UT-3 enabled interactive programming of TX-0. When loaded, it would wait for commands from the Flexotype terminal. The commands, written in a symbolic language, could be used to examine and modify the contents of registers, search the memory for a particular value or references to a particular address, modify the memory and identify changes between the in-memory program and its version on tape, as well as to run another program in memory using the `go to` command.

The UT-3 was followed by a large number of other utilities, including a more sophisticated symbolic assembler (Macro) and a debugging tool (Flit) that I return to in the next chapter. A more outlandish utility was developed for debugging a long-running voice recognition software. The utility allowed the user to follow the control path through a program in a flow chart, using a rather elaborate method that involved drawing the flow-chart by hand on a transparent foil and placing it over the computer screen[22]. It also allowed to slow down or speed up the program using the toggle switches.

The person in charge of TX-0 after it moved to the Research Laboratory of Electronics in Building 26 was Jack Dennis. As an undergraduate, Dennis worked on Whirlwind. After midnight, he was able to get direct access to Whirlwind. He knew this was a more efficient way of programming than using printouts and the experience convinced him about the value of direct working with computers. Dennis himself contributed to many of the aforementioned TX-0 tools, including the Macro assembler and the Flit debugger, but he also attracted a new generation of hackers. He gave a number of introductory talks on TX-0 to

the Tech Model Railroad Club, with which he was formerly affiliated, and established the informal rules that made TX-0 widely accessible.

Many of the TMRC hackers, including Kotok and Graetz who were later involved in the development of Spacewar!, were curious about computers and soon started exploring what can be done with the machine. Their way of working included many of the principles of the hacker culture and hacker ethic that I wrote about in the context of Spacewar! To get as much access to the machine as possible, they often used it during the night. Many of their programs were designed for the sake of programming itself, that is to make programming of the TX-0 easier. Many other were curious demonstrations of what is possible, such as a program that controlled an audio speaker to produce computer music. Jack Dennis suggested this problem to Peter Samson who was able to produce a simple melody on TX-0, but was not very satisfied. He eventually installed a hardware extension to TX-0 so that he could get a three-part harmony out of it.[23]

The hacker culture forming around TX-0 encouraged information sharing that had lasting effect on the software industry. It enabled the collaboration around Spacewar! that I mentioned above, but it inspired the influential Free Software movement. The limited capabilities of the TX-0 also gave rise to a certain hacker aesthetic. The hackers prided themselves in finding clever tricks to make programs as compact as possible in order to save valuable memory space. On TX-0, writing programs this way was the only option, but low-level assembly was still ubiquitous in the HAKMEM memo[24], which presented a collecton of random MIT hacker tricks a decade later.

Symbol Manipulating Language

The development of high-level programming languages was making programming easier in the second half of the 1950s. I already wrote about FORTRAN, which became available for IBM 704; FLOW-MATIC and COBOL, which focused on business data processing and ALGOL 58, which made "programming language" into an independent (mathematical) object. But all of these had to be used in the tedious batch-processing mode and did not appeal to the hacker aesthetic. Hackers too would eventually find their high-level programming language of choice. To follow its origins, we need to get back to a workshop organized at Dartmouth in the summer of 1956 by John McCarthy, before his move to MIT. The workshop brought together a small number of researchers interested in intelligence, learning, abstraction, creativity and computers. McCarthy choose the term Artificial Intelligence for the name of the workshop, conining the name for a new branch of computer science.

Two of the attendees of the workshop were Herbert Simon and Allen Newell. Together with a programmer Clifford Shaw, the two created a program they called Logic Theorist earlier in the year. Logic Theorist was a program for automated reasoning using propositional logic and was eventually able to automatically prove 38 of the first 52 theorems from Whitehead and Russel's Principia Mathematica. It worked by selecting and applying appropriate rewrite rules, such as substitution and modus ponens, to a given theorem using heuristics to choose a suitable rule.[25]

Logic Theorist was implemented in a programming language IPL created by Newell, Simon and Shaw. The language was an assembly language, but it was centred around the idea of list processing. For 1956, this was a revolutionary idea. The langauge was designed not for numerical computation, but for symbolic list processing and lists proved a powerful tool for representing theorems and implementing their transformations.

After learning about the Logic Theorist and IPL, McCarthy realised that list processing would be good fit for his own research, but he also wanted a language that would follow the intuitive algebraic notation of FORTRAN. McCarthy first explored the idea of implementing support for list processing in FORTRAN as an advisor for an IBM project on reasoning about plane geometry. This led to a language FLPL, which was a set of list processing routines for FORTRAN. Shortly after moving to MIT, McCarthy recruited a group of graduate students and hackers to create LISP, a new list processing language for the IBM 704 machine.

Although LISP was the brainchild of John McCarthy, the language has origins in multiple culturse of programming. This is because McCarthy himself belonged and contributed to multiple cultures. His example shows not only that individual can move between cultures of programming and the views of individual cultures are not incommensurable. His example also supports the hypothesis that the most interesting work on programming arises at the intersection of multiple cultures.

LISP demonstrably brought together ideas from many different cultures. McCarthy himself later played pivotal role in the mathematical culture of programming and LISP adopted the lambda notation for functions from the Church's lambda calculus. However, McCarthy himself acknowledged, the name was the only influence as he had not studied lambda calculus before designing LISP.[26] LISP was also strongly influenced by the engineering issues of list processing and the fact that this was first explored in the context of FORTRAN further shows the influence of the pragmatic engineering culture. LISP would also never happen without the influence the early artifical intelligence research. This provided new kind of programming problems, based on thinking about human thinking, an approach that inspired much work in the humanistic culture of programming that I will discuss shortly.

Some of the interesting innovations in LISP were also due to the involvement of the MIT hackers. One of those hired to work on LISP was Steve Russell, who later became famous for his work on Spacewar! In order to explore the computational power of LISP, the team implemented a function called `eval` that took a LISP expression represented using lists and evaluated it. Russell then pointed out that `eval` can itself be used as a basis of an interpreter for LISP, hand coded the function for IBM 704 and LISP got its first interpreter. Thus an almost accidental realization of a hacker was at the source of one of unique LISP characteristics that is now referred to as "meta-circularity", that is the fact that it is largely defined in itself. Despite the appeal of LISP, working on it on the IBM 704 computer in batch processing mode was, in the words of Steve Russell, a "horrible engineering job".[27]

Augmenting Human Intellect

Before we continue following the story of LISP and interactive programming, I need to retrace some of the background of the culture that influenced much work on interactive programming. The proponents of this view include artists, psychologists, engineers, educators and many others, but they all share a distinct concern for the human and the relationship between the human and the computer. I refer to this as the *humanistic culture* of programming. Unlike, for example, the mathematical culture, humanistic thinking about programming involves a wider range of perspectives. I already talked about the influences of artificial intelligence, which was concerned with understanding and modelling human thinking; I will soon discuss creative influences that inspired work on computer art.

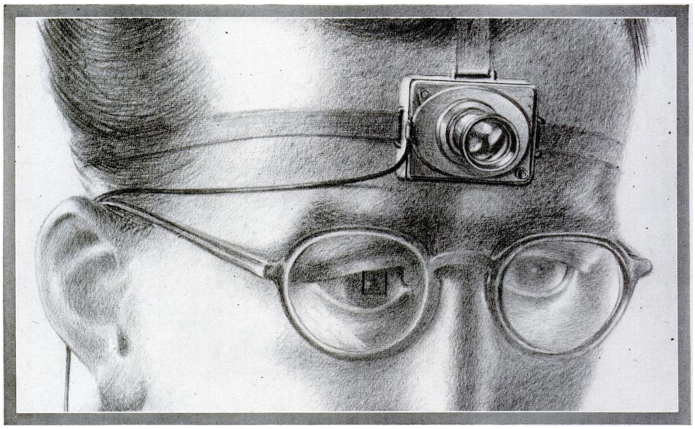


Figure 3.7: Illustration for a condensed version of "As We May Think", republished by the Life Magazine (Vol. 19, No. 11) in October 1945 (*"A scientist of the future records experiments with a tiny camera, fitted with universal-focus lens"*)

As my starting point for characterizing the humanistic culture of programming, I use two essays that sketched futuristic visions of the relationship between the human and the computer. To use a term adopted later, the two essays envisioned the potential of computers for augmenting the human intellect. The humanistic culture has many different origins and could be with some poetic license linked to Romanticist ideas through the work of Ada Lovelace[28], but the two essays directly inspired some of the important follow-up work and serve well to illustrate the aims and ways of thinking of the humanistic culture.[29]

The first essay was written by Vannevar Bush, who was an engineer, inventor and science administrator. Bush joined the Department of Electrical Engineering at MIT in 1919 and worked on a number of electronic devices, successfully commercializing some of his work. He led a team that built the *differential analyzer*, an analog computer for solving differential equations. The machine combined mechanical and electric components and was capable of solving equations with up to 18 independent variables. However, Bush was also a successful research administrator. He became the dean of the MIT School of Engineering and, during the Second World War, managed various scientific research organizations established by the government to support the military efforts.

Before the war, Vannevar Bush started to think about the "library problem," that is the fact that investigators in any modern scientific enterprise need to find and plough through hundreds of materials. He thought some kind of invention, based on microfilm and analog electronic devices would be able to make such information management much more efficient. After the war, Bush returned to the idea and published an influential article that presented the idea as a direction for peacetime research. In "As We May Think"[30], Bush introduces the *memex*, a hypothetical device that would store all individual's books, records and communication and allow navigating through it via *associative trails*. The vision is strikingly similar to the World Wide Web with its hypertext links, although Bush's thinking was in terms of analog devices, microfilms and he imagined the device as a desk.

The second early proponent for the humanistic way of thinking is J. C. R. Licklider. His background was psychoacoustics, but he became interested in information technology and joined MIT in 1950 where he was involved in the SAGE radar data processing system for which the interactive Whirlwind computer was built. He later moved to Bolt Beranek and

Newman (BBN), a company with close links to MIT which played a key role in the development many interactive programming tools a decade later. In 1960, Licklider wrote the second landmark essay of our story, "Man-Computer Symbiosis"[31]. The essay presented a vision of artificial intelligence where the computer is not replacing human in specific tasks, but instead forms a symbiotic system with the human user. In his own words, the vision is for "men and computers to cooperate in making decisions and controlling complex situations without inflexible dependence on predetermined programs". To John McCarthy, the ideas were not surprising[32], but their presentation was read well and set a well-defined research agenda.

In 1962, Licklider got a chance to pursue this agenda in concrete terms when he joined the Department of Defense Advanced Research Projects Agency (ARPA) as a director for its recently established Command and Control Research office, later renamed to Information Processing Techniques Office (IPTO). To the ARPA director, Licklider was known for his work on human factors of SAGE, but his vision of man-computer symbiosis also resonated well with the vision of using computers in military command and control systems.[33]

Licklider never directly collaborated with Vannevar Bush, but he worked on a project studying "Libraries of the Future"[34] that was inspired by the memex and was started before his time at ARPA, only to be completed after the end of his two-year term as the IPTO director. Both Bush and Licklider outlined a similar humanstic vision of the future of computers where computers are used to augment human mental capabilities, be it by assisting them with information organization or with making sense of and solving complex problems.

The vision of augmenting human intellect outlined in "As We May Think" and "Man-Computer Symbiosis" keeps inspiring programming system designers interested in more creative ways of using computer to this day. One of the earliest examples of this was the Sketchpad system developed by Ivan Sutherland in his 1963 PhD thesis, "Sketchpad: A man-machine graphical communication system". The system allowed users to construct computer drawings using an interactive graphical user interface. This may seem trivial today, but it was a major undertaking in the early 1960s and it required efficient use of cutting edge computing technology. Sketchpad run on TX-2, which was a successor of the TX-0, built at the Lincoln Lab two years later. TX-2 was not as easily accessible as TX-0, but it was more powerful and was equipped with the light pen, which Sketchpad utilized as the primary input device.

Sketchpad went beyond just drawing. It supported editing, specification of constraints (e.g. two lines are parallel) and definition of new reusable symbols. Sketchpad symbols later served as one of the inspirations for object-oriented programming as modifying the definition of a symbol (a class) resulted in change to all its uses (instances).

Ivan Sutherland was inspired by both Vannevar Bush and his ideas around memex and by J. C. R. Licklider and his ideas on man-computer symbiosis. The reference to "communication system" in the substitle of Sutherland's thesis points to the two organisms, collaborating in a symbiotic relationship. As put by Sutherland himself, Sketchpad provides a more efficient ways of communication between the human and the computer:

> *The Sketchpad system makes it possible for a man and a computer to converse rapidly through the medium of line drawings. Heretofore, most interaction between men and computers has been slowed down by the need to reduce all communication to written statements that can be typed (...) For many types of communication, such as describing the shape of a mechanical part or the connections of an electrical circuit, typed statements can prove cumbersome.*

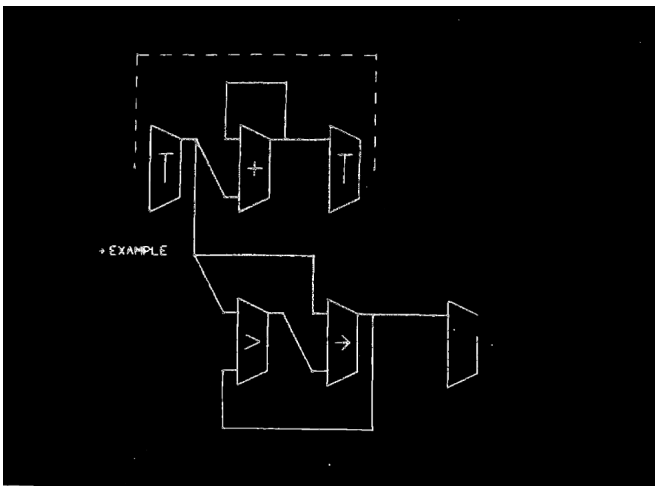


Figure 3.8: A sample visual program which accepts numbers from terminal and prints their running sum and the maximum.[38]

Sketchpad envisioned using graphics for drawing and sketching of diagrams, but not for programming the computer itself. The wait for this idea to appear was not long though. A follow-up project by Bert Sutherland, an older brother of Ivan, completes the task and describes a visual language for "online graphical specification of computer procedures"[35] (Figure 3.8) in his own PhD thesis in 1966, three years after his younger brother. Not long after, the idea of using flow-charts for visual programming was implemented in the GRAIL system[36] that used a new tablet device.[37]

In the early 1960s, Sketchpad was probing the limits of what was possible to do. To offer a glimpse of personal computing, it required the full power of gigantic and enormously expensive TX-2 machine. Only few could imagine that such power would become available for computers that could be owned and used by individuals. One of those who did realize this was Wes Clark, the engineer who designed TX-0 and TX-2 and believed that personal computers were the futrue of interactive computing. Clark set out to build such machine in 1961 after completing TX-2. In just one year, he completed LINC, which is sometimes referred to as the first personal computer. The machine supported real-time data processing and was first introduced as a tool for neurophysiological researchers. Following the first public demonstration at National Institutes of Health, the participants could clearly see the potential of the machine:

> *It was such a triumph that we danced a jig right there around the equipment. No human being had ever been able to see what we had just witnessed.*[39]

Despite the enthusiastic reception, LINC was only moderately successful. The team continued their work and build the first dozen of machines for biomedical applications by 1963. Outside of medical systems, the idea was perhaps too far ahead of its time. Only a few could imagine the dropping cost of computers dropping enough to make the vision realistic. As a result, the team did not receive further funding from MIT, and had to move, which hindered the progress. The work on personal computers continued in various forms throughout 1960s, but it only gained prominence with the work at Xerox PARC in the 1970s that I will return to soon. Meanwhile at MIT, the work on interactive programming was taking a different, perhaps less visionary but a more practical trajectory.

A Time Sharing Operator Program

The hackers at MIT had access to a single interactive computer at night hours, but this was an exception from the rule. Most other computers at the time, especially those used by businesses, were used through batch-processing. New smaller computers like PDP-1, which was a commercial follow-up to TX-2, were becoming cheaper, but they still cost over $120,000 in 1960 (equivalent to roughly $1M in 2021 in terms of purchasing power). At MIT, the cost of PDP-1 inspired a joking series of program names such as "Expensive Plantarium" (for the code to draw Spacewar! background) and "Expensive Synthetizer" (for the music playing program). At most other installations, the cost practically prevented interactive use. If interactive programming was to become more commonplace, interactive computing capabilities needed to become accessible to a much wider audience in some way despite the high cost of computers.

One possible answer, which was already in the air at the end of 1950s, was to allow multiple users to interact with a single, more powerful, computer at the same time. Humans interacting with a computer through a terminal spent most of the time thinking and typing, so a single machine could, in principle, handle concurrent requests from multiple users. John McCarthy, who just moved from Dartmouth to MIT, started to call this technique "time-sharing"[40]. To him, the idea "seemed quite inevitable" and he thought that "everybody had in mind to do" it[41]. That was not the case and it took quite some work to get the idea off the ground.

In 1959, McCarthy wrote a memo "A Time Sharing Operator Program for our Projected IBM 709", in which he argued that time-sharing should be the primary way of using a new machine that the MIT was being given by the IBM. He saw time-sharing primarily as a way to "reduce the time required to get a problem solved on a machine". He first had to convince the MIT Computation Centre, which managed access to most computers. This was not hard as the centre was well aware of irate users, complaining about the 3 to 36 hour response time for running programs in the batch processing mode.

McCarthy proposed to do a small-scale experiment using the existing IBM 704 machine. Interestingly, this turned out to require not just writing new code, but also modifying the hardware so that an interrupt is triggered when the machine receives input from a user. McCarthy asked Steve Russell to implement an interactive version of LISP for the IBM 704 time-sharing system and held the first demonstration in 1960. The idea caught on, attracting both the MIT Computation Centre, which saw time-sharing as a more effective use of their hardware, as well as J. C. R. Licklider, who saw it as a way towards his idea of man-computer symbiosis. In fact, the idea was so closely related that when McCarhty read Licklider's essay on man-computer symbiosis, he thought "the whole notion was obvious."[42] Licklider later provided significant funding for time-sharing in his role of the IPTO director at ARPA in the mid-1960s.

The development of a large-scale time-sharing system suffered various technical and political setbacks[43], but the efforts eventually succeeded. At MIT, the major first implementation was the Compatible Time-Sharing System (CTSS), which was developed as part of the ARPA-funded Project MAC. By mid-1960s, it run on a dedicated IBM 7094 machine with over 100 terminals and was able to support 30 concurrent users.

Time-sharing systems were more limited than early interactive comptuers with display screens, but they were good enough to allow interactive use of high-level languages like LISP. This soon gave the MIT hackers a high-level programming language that was *fun* to

use. Interactive programming would soon no longer require the use of low-level machine instructions, which was an interesting challenge for hackers, but made it difficult to produce programs of greater complexity.

A Step Toward Man-Computer Symbiosis

The standard way of running a LISP program at MIT in 1960 was using punched cards. Your program had to be preceded by five cards labeled NYBOL1 and six cards labeled LCON, which you could fetch from the "Utility Decks" drawer in room 26-265 in the MIT Computation Centre. LISP came with a tracing functionality that let you debug programs away from the computer. Thanks to the early work on time-sharing, the MIT deployment on IBM 704 also allowed users to use LISP interactively using the LISP-Flexo system, using the same kind of terminal as when using the TX-0. The system evaluated entire LISP expressions entered by the user. The designers of the system recognized that typing a whole expression at once is difficult and offered so called "TEN-Mode" in which expression could be entered in smaller chunks into ten buffers (in any sequence with the option to overwrite earlier inputs) before evaluating it. The interactive mode of LISP I was rather basic, but it offered a glimpse of a not-too-distant future.[44]

This started in 1963, when Peter Deutsch created interactive LISP for PDP-1 at Bolt Beranek and Newman (BBN). Deutsch realised that interactive way of programming requires "tools for rapid modification of already existing LISP programs within the LISP system"[45] and started working on an interactive LISP editor. The editor leveraged the fact that LISP was a list-processing language and LISP programs themselves are represented as lists that contain either symbols or further nested lists. This means that what Deutsch needed to do was to create an editor for nested lists and then use that to edit LISP programs.

The LISP editor allowed programmers to navigate through the nested list structure and make changes to it. As an example Deutsch uses a LISP function to append two lists with a number of errors and then describes how to correct those. One such error is illustrated in the following snippet:

```
(LAMBDA (X) Y (COND ((NUL X) Y)
  (T (CONS (CAR X) (APPEND (CDR X) (Y))))))
```

The function, defined using the lambda notation, is supposed to take two arguments, `X` and `Y`. If the first is empty `(NUL X)`, it returns `Y`; otherwise, it takes the first element of `X` using `CAR X` and appends the remaining ones using recursion before adding the removed element. The error here is that the arguments of the function, `X` and `Y` should be written in a single list as `(X Y)`.

After invoking the interactive editor on the function, the programmer can type commands such as `P` to print the currently edited expression. Numbers such as `3` navigate to an n-th element of a list and `0` navigates back to a parent list, so the following prints the expression, focuses on the second sub-list, prints it and then returns back (lines starting with `*` are typed by the user):

```
*P
 (LAMBDA (X) Y (COND & &)).
*2 P
 (X)
*0
```

When printing, the editor does not print the entire expression, but only the first few levels. More deeply nested lists, such as the arguments of `COND` are replaced with the `&` symbol. To modify the list, programmers could type a number in parentheses to remove or replace n-th element of a list, so the following corrects the error by first removing third element and then replacing the second with a new list `(X Y)`:

```
*(3)
*(2 (X Y))
*P
 (LAMBDA (X Y) (COND & &)).
```

The LISP editor was first developed as a stand-alone tool, but it was later revised and improved by others at BBN and incorporated into a LISP distribution named BBN-LISP in 1971.

Deutsch was a high-school student when he started implementing LISP for the PDP-1, but he was already an accepted member of the hacker community thanks to his programming skills. The LISP editor, which he developed as a student at University of California, Berkeley, is also a product of the hacker culture. Despite being one of the first structure editors for programming languages, a technique that would become an active research field in the 1980s and 1990s, Deutsch's LISP editor was ever only documented in manuals and various internal reports.[46] The project also follows the typical pattern where hackers are dissatisfied with their own programming tools and set out to improve them.

Another development that changed how interactive LISP programming was done was taking place at MIT at the same time as Deutsch created his LISP editor. It was done as doctoral research project and so it was approaching the problem from a more academic perspective. In the 1960s, LISP was the de facto language for artificial intelligence research. The practice of AI research provided motivation for Warren Teitelman, who developed an interactive programming system for LISP named PILOT:

> *In artificial intelligence problems (...) it is important for the programmer to experiment with the working program, making alterations and seeing the effects of the changes. (...) PILOT is a programming system that is designed specifically for this purpose. It improves and raises the level of interaction between programmer and computer when he is modifying a program.* [47]

Although PILOT shares technical characteristics with systems produced by hackers, Teitelman positions his work in the context of "symbiotic systems", adopting the perspective of "man-computer symbiosis" of J. C. R. Licklider. This is perhaps a necessity as work emerging from the hacker culture was rarely treated as being worthy an academic publication. Together with references to systems like Sketchpad, this links PILOT to the humanistic culture of programming.

The central piece of PILOT was a language for specifying list transformations called FLIP. This was then used in various ways to create, edit and modify LISP programs. PILOT treated programs as collections of procedures and it allowed two kinds of modifications. *Editing* was the process of changing individual procedures. The `EDIT` function allowed the user to look for patterns in procedure code and transform the matching parts. *Advising* was the process of changing the interface between procedures. The `ADVISE` function could be used to insert other procedure at the entry or exit points of any other procedure. This was used, for example, to insert procedures that would record the history of program execution.

In advising, the PILOT system anticipates the concept of Aspect-Oriented Programming that would appear in the early 2000s. However, PILOT is more interesting for another reason. In his PhD thesis, Teitelman views LISP as a *programming system* that the programmer interacts with rather than a *programming language* in which the programmer writes code. This perspective is in stark contrast with the mathematization of programming and the formalization of programming languages that followed the publication of the Algol report. This schism remained open for several decades, until, in the 1990s, the "language" way of looking dominated.[48]

The work on the LISP editor, BBN-LISP and PILOT all came together in Interlisp, which was an implementation of LISP notable for its many innovative programming tools. It was designed to be an interactive programming system. The programmer using Interlisp was interacting with a running LISP environment that contained both data and code. The users did not have to worry about files, which were only used when saving the state of the system to an external backup media. Interlisp also featured an auto-correction system called DWIM ("Do What I Mean"), also created by Teitelman, that was triggered when running a program resulted in error. It attempted to correct common errors such as typos or incorrect parenthesization. All such corrections were done in the Interlisp environment, so the corrected version was used next time the code was executed.[49]

The perspective that LISP is a programming system rather than a language became even more dominant in late 1970s. The increasingly complex artificial intelligence projects required more memory than the most common research computer could provide and new stock machines were less suitable for running LISP, primarily because their architecture made common operations with lists hard to implement efficiently. In response, Richard Greenblatt and Thomas Knight at MIT began building a computer specifically designed to run LISP efficiently. This was the birth of "LISP machines"[50], a class of computers that would be the dominant way of running LISP in 1980s. LISP machines were personal computers with displays. LISP was used both as the systems language to implement everything on the machine and as the way of interacting with the computer and, in many ways, LISP fulfiled the role of an operating system.

The case of LISP is interesting, because unlike ALGOL and COBOL, it was the product of a mix of cultures. It was used in AI research, it was partly inspired by mathematical formalisms and it was developed by hackers. Interlisp pioneered many of the programming tools that will later become the bedrock of engineering approach, even though this still required further development. For one, the DWIM auto-correction was reportedly somewhat idiosyncratic to errors made by Warren Teitelman himself. Those hackers making different kinds of errors claimed the acronym DWIM stood for "Damn Warren's Infernal Machine."[51] Different cultures of programming also developed different dialects of LISP. A good example is Scheme, created by Sussman and Steele in 1975 and influenced by Sussman's study of Algol. Scheme was not only more directly inspired by mathematical programming roots. It was also used in a more mathematical work on programming. In particular, it resulted in an influential series of papers on programming language semantics ("Lambda the Ultimate") and this was the main force for its success[52]. Another example of a system that takes a distinct perspective on LISP was LOGO, which emerged from the humanistic culture and was focused on education. I will return to LOGO after a brief remark on computer art that is useful for explaining the context of the humanistic culture.

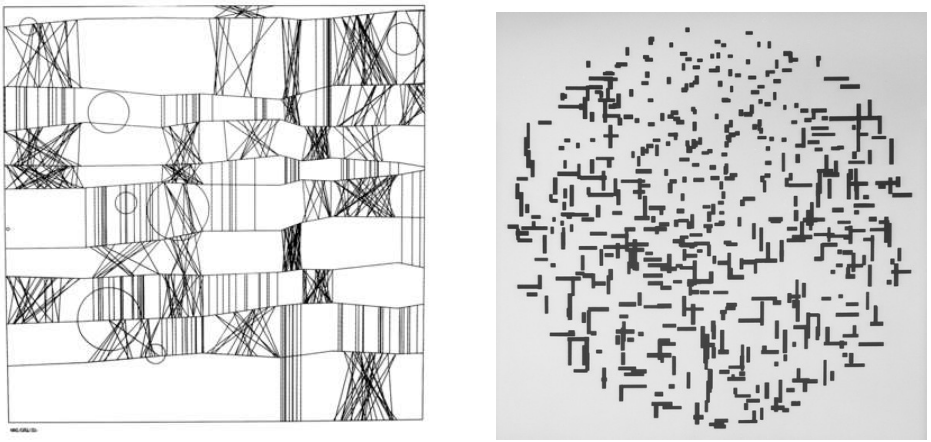


Figure 3.9: Two examples of early computer art. Homage to Paul Klee, Frieder Nake, 1965 (left) and Computer Composition with Lines, A. Michael Noll, 1964 (right)[53]

There Should be No Computer Art

The humanistic culture of programming is characterized by the concern for the relationship between the human and the computer. The futuristic visions of man-computer symbiosis that I discussed above are one case of this, but the relationship can be probed and explored in many ways. The early use of computers in the context of art approaches the question from a different perspective, but often poses similar questions.

The early work on computer generated art dates back to 1950s, but first public exhibitions took place in 1965. Frieder Nake exhibited his work in Stuttgart and A. Michael Noll presented his work in the same year in New York. As Figure 3.9 shows, early computer generated art creatively used the limited capabilities of early computers and single-color plotters and much of the work involved abstract art, utilizing the basic shapes that computers can produce. Early computer artists also illustrate another important trait of the humanistic culture of programming. It is the self-consciousness and critical reflection of their role in the society. Not long after the first exhibitions, Frieder Nake wrote a provocative essay "There should be no computer art"[54] in which he criticises the fact that computer art is being co-opted by the commercial art world, in order to create "another fashion for the rich and ruling". Nake does not want computers to become a "source of pictures for galleries" and argues instead for a use leading to critical reflections on art and the society. Could the kind of art created first by Nake be a basis for a more positive transformation in the society, rather than hanging on the walls of rich art collectors?

In the vision for children education pursued by Seymour Papert and his colleagues at MIT, this was exactly the case. The origins of this work are surprisingly technical. In 1961 at a conference in England, Marvin Minsky and Seymour Papert presented similar papers on a technique that became known as reinforcement learning and is the basis of many successful applications of AI to this day. Minsky was an co-founder of AI from MIT and the author of numerous PDP-1 hacks, while Papert was working with psychologist Jean Piaget in Geneva studying education. The two were both interested in thinking about thinking; Minsky in thinking in machines and Papert in thinking in children. They soon started a collaboration that lasted several decades and brought Papert to MIT in 1963.[55]

Papert joined Minsky as a co-director of the MIT Artificial Intelligence lab. He soon got fascinated by the hacker community, which now favored LISP as their programming language of choice. In the community, Papert found "extraordinary examples of learning and problem solving that took place organically."[56] He also soon started working with the education research group at Bolt Beranek and Newman (BBN) that was led by a mathematician and a musician Wallace (Wally) Feurzeig and included Dan Bobrow and Cynthia Solomon, who previously took a job as Marvin Minsky's secretary so that she can learn programming and was now, among other things, involved with the BBN-LISP project.

Papert was inspired by Piaget's theory of active learning, which believes that students should construct their own knowledge by actively interacting with concrete materials. After coming to MIT, he started looking for the right kind of "concrete materials" that would enable children learn mathematical ideas using the computer. The technical developments such as time-sharing and interactive computing made it possible to think of programming beyond scientific computations and business data processing. The MIT hackers thought of programming as something that every dedicated hacker can master, but it took another leap of thought to see programming as something that every child can learn. This step was taken by the group around Papert, Feuerzeig and Solomon who started thinking about a programming language, eventually called LOGO, that would make such learning possible.

Children, Computers and Powerful Ideas

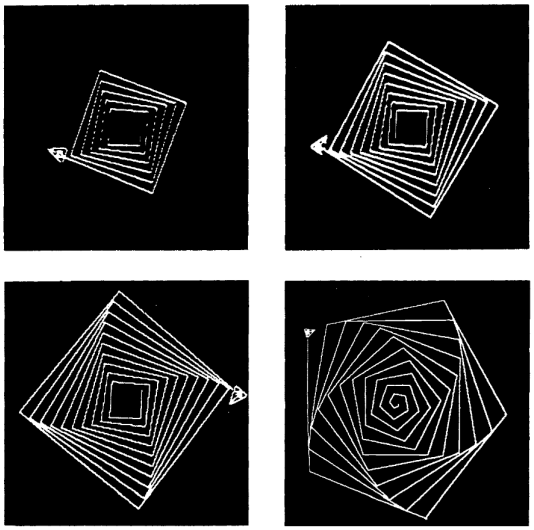
Technically, LOGO is a programming language inspired by LISP, but its aims and culture are very different from anything seen in programming prior to 1967. At the time, professional programmers were learning to program in order to solve business problems; MIT hackers were learning to program in order to do programming, because programming was fun. Papert did not think that children should learn programming for either of those reasons. Instead, he saw programming as a way of communicating with a living native speaker of mathematics. The view is best captured in his later book:

> *The computer can be a mathematics-speaking and an alphabetic-speaking entity. We are learning how to make computers with which children love to communicate. When this communication occurs, children learn mathematics as a living language. Moreover, mathematical communication and alphabetic communication are thereby both transformed from the alien and therefore difficult things they are for most children into natural and therefore easy ones.*[57]

The group was aware of the early work on computer art and also saw LOGO as a vehicle for art. But most importantly, they saw computers as the means that is best used for creative ends. *"As in a good art class, the child is learning technical knowledge as a means to get to a creative and personally defined end."*[58]

Logo was based on LISP because of LISP's interactive programming capabilities and because it made list processing easy. The designers explicitly did not want to create a simple language just for children. They wanted a langauge that would allow students progress from a simple environment to the most powerful kind of programming available. The key design idea that allowed this progression was later captured by the term *microworlds*.

Microworlds are small environments for thinking about particular problems. The first microworld was built around simple word manipulation and could be used, for example, to



```
TO SPI DISTANCE
FORWARD DISTANCE
RIGHT 90
SPI DISTANCE + 5
END
```

Figure 3.10: Drawing Turtle graphics using Logo. `SPI` is a recursive procedure that moves the turtle forward, rotates it and calls itself with greater distance. The output shows interesting effects achieved by varying the angle on line 3.

create a program that constructs a sentence by generating a random sequence of words of the form "adjective, noun, verb, adjective, noun". The resulting sentences, allow children to naturally get interested about the structure of natural languages and grammar:[59]

```
RED GUINEA PIGS TRIP FUZZY WUZZY DONKEYS
PECULIAR BIRDS HATE JUMPING DOGS
FAT WORMS HATE PECULIAR WORMS
FAT GEESE BITE JUMPING CATS
```

A more famous microworld is the later Turtle graphics where the child controls a turtle that draws lines in a two-dimensional space. Turtles first existed in the form of a floor Turtle, a robot that moved around actually drawing on the ground, but screen Turtle provided a similar experience using a display (Figure 3.10). The Logo designers thought of Turtle as an object-to-think-with. Although Papert later said that Turtle should be seen as just one example of such objects-to-think-with, it remains an unparalleled example of a microworld for a number of reasons.

First, the children can easily learn to see the world through the perspective of the turtle. When programming, they can think what they want the turtle to do in relative terms. This makes graphics easier in contrast to using absolute Cartesian coordinates. In fact, when Cynthia Solomon later realised that young children had hard time typing longer commands, she developed a command `TEACH` that lets children instruct the turtle using a simple language of single-alphanumeric commands and is capable of recording and replaying such instructions. Second, the Turtle metaphor also allows children to debug programs by "becoming the turtle" and literaly stepping through the code. Even more interestingly, when the program misbehaves, a child can think "the turtle has a bug" rather than thinking "I made a mistake", allowing them to avoid negative emotions.

The goal of LOGO was to teach powerful ideas to kids. The group used the term "powerful idea" to refer to concepts that are more general than the examples through which they are taught, such as the idea of anthropomorphization (becoming a turtle) or metalanguage (language for talking about a language). In many ways, the work on LOGO was rooted in the humanistic culture of programming. For one, LOGO was more than just a programming language. "It was also a computer environment made up of people, things,

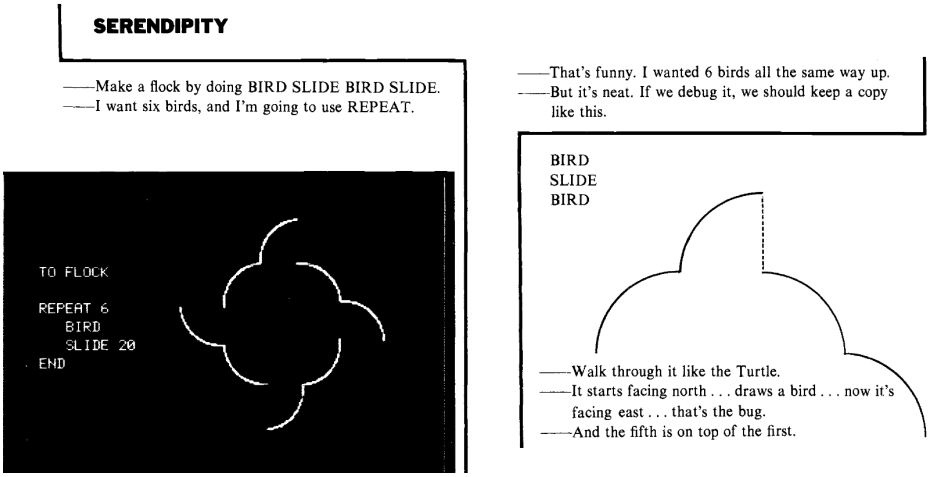


Figure 3.11: An excerpt on serendipity from Mindstorms Papert (1980), showing a "conversation between two children who are working and playing with a computer".

ideas. And it was a computer culture: a way of thinking about computers and about learning and about talking about what you were doing."[60] Logo designers also viewed their work in a broader social and political context. This is well documented in Papert's Mindstorms book where he points out that "there is a world of difference between what computers can do and what society will choose to do with them.". He also acknowledges that his work is political and proposes a revolutionary change, because "schools as we know them today will have no place in the future." The work arising from all cultures of programming had a political side, be it struggle for control over software production, an attempt to establish programmer as a more masculine professional or striving for academic respect.[61] What is unique about the humanistic culture of programming is that it often acknowledges and discusses the political side of programming, even if it often paints an overly rosy picture.

The cultural background had a number of interesting technical implications. In particular, the Logo culture is interesting in how it approaches program errors. Those are seen as sources of serendipity that provide an opportunity for learning. Figure 3.11 shows an example showing the debugging of an unexpected situation in Logo using the Turtle graphics. In the hypothetical situation, the child first notes that the erroneous result is nice, before proceeding to debug it by "becoming the turtle".

Logo designers were also conscious of aspects of the system that are important for communication, but are ignored by most other language designers. This includes the syntax, naming but also error messages. A telling story is that the response of early Logo to an unknown command such as `love` was `love has no meaning`. The designers wanted to avoid telling such message to children and discussed printing either `I don't know how to love` or `You haven't said how to love`. Both of these were criticised. The former overly anthropomorphizes the computer while the latter puts too much blame on the child.Solomon et al. (2020) Finally, the designers were also not talking about just the programming language, but about the entire educational environment. This includes solving problems using Logo on computer, but also group interactions and physical activities. Famously, the teaching done by the Logo group also involved activities like juggling and the Mindstorms book explains different styles of juggling as they would be encoded in Logo.

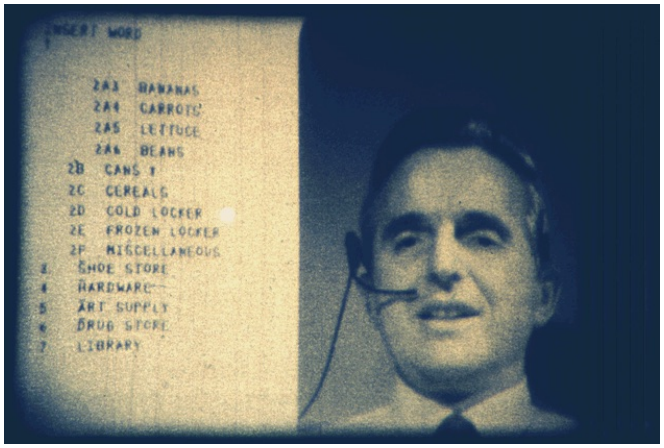


Figure 3.12: Collaborative document editing in the oNLine System (NLS) as demonstrated by Douglas Engelbart in "The Mother of All Demos" in 1968[63]

The Logo programming environment is perhaps the earliest and most prominent programming system that was primarily influenced by the humanistic culture of programming, but it had strong influences and contributions from other cultures. The initial motivation for LOGO was to teach mathematical ideas and much of the technical developments that it crucially relied on were being built thanks to large-scale engineering efforts, like time-sharing, and playful excitement of hackers who often shared the physical space with the LOGO designers.

At the same time as LOGO was designed to help children learn to think on the East coast, another project heavily influenced by humanistic visions focusing on improving human capabilities was getting ready for its debut on the West coast.

The Mother of All Demos

On December 8, 1968, Douglas Engelbart presented the oNLine System (NLS) at the Fall Joint Computer Conference in San Francisco. At a time when most interaction with computer was through a teletype, connected to a time-sharing system, the demonstration gave a glimpse of the future. It used graphical user interfaces throughout and featured a way of navigating between documents akin to modern web browsing, interactive display-based text editing not unlike the present-day wikis, a new device for controlling the computer called the "mouse", as well as a tele-conferencing system bringing some of the speakers virtually from Menlo Park. For all these reasons, the demonstration is often retroactively referred to as "The Mother of All Demos" (Figure 3.12).

Engelbart's work was directly inspired by Vannevar Bush. He read the "As We May Think" essay while serving as radio and radar technician with the United States Navy. It greatly influenced him and motivated him, in 1951, to return to university and pursue graduate studies at Berkeley. After completing his studies, he joined the Stanford Research Institute (SRI) in Menlo Park. He became a valued contributor in the SRI group working on magnetic storage, but his true interest was in pursuing his vision that he started to refer to as "augmenting human intellect".[62]

By the end of the 1950s, Engelbart got a small grant to work on his ideas and developed a detailed account of his vision, but without actually implementing any of it yet. He

outlined the vision in a report "Augmenting Human Intellect: A Conceptual Framework"[64]. The report is positioned in the humanistic culture of programming through its references to Vannevar Bush and J. C. R. Licklider, as well as its motivation. The objective of augmenting human intellect is:

> *Increasing the capability of man to approach complex problem situation to gain comprehension to suit his particular needs and to derive solutions to problems. Increased capability in this respect is taken to mean mixture of the following more rapid comprehension, better comprehension, the possibility of gaining useful degree of comprehension in situation that previously was too complex, speedier solutions, better solutions and the possibility of finding solutions to problems that before seemed insoluble.*

The report is a combination of a general analysis, specific examples and idiosyncratic details, which are all typical for later Engelbart's work. He analyses general structures involved in working with information, such as different sources of intelligence. He describes the envisioned system that is referred to as H-LAM/T (Human using Lauguage Artifacts Methodology in which he is Trained), next research steps needed to build it, but he also gives concrete examples such as that of an "augmented" architect designing a building using the system:

> *He sits at working station that has visual display screen some three feet on side; this is his working surface and is controlled by computer (his "clerk") with which he can communicate by means of small keyboard and various other devices. (...) With "pointer" he indicates two points of interest, moves his left hand rapidly over the keyboard, and the distance and elevation between the points indicated appear on the righthand third of the screen.*

Here, the report describes what seems like a modern CAD system controlled through a graphical user interface. It is also closely related to the Sketchpad system that was being developed at MIT by Ivan Sutherland at the same time and was also inspired by ideas on man-computer symbiosis. In the report, Engelbart mentions hearing about the Sketchapd project, but notes that further information was unavailable at the time of writing.

Engelbart's report eventually found its way to Robert Taylor who worked as a research program manager at NASA. Taylor never pursued PhD, but he had background in psychology and did research on psychoacoustics. He was familiar with the pioneering work of the humanistic culture of programming, including the essays of Bush and Licklider, and he soon started supporting Engelbart's work through grants from NASA.

In 1962, Taylor met Licklider who was newly appointed as the director of IPTO. Licklider soon began asking Taylor about his psychoacoustics research. The two became close and Taylor eventually succeeded Licklider as the IPTO director, after it was briefly led by Ivan Sutherland. The NASA and later IPTO provided Engelbart with funding to setup Augmentation Research Center at the SRI and allowed him to recruit a number of early collaborators. The first of those was Bill English, an engineer who, among other things, worked with Engelbart on the development of computer "mouse" and managed many of the key systems of the 1968 presentation, including the video and audio link between the San Francisco auditorium and the Menlo Park office.

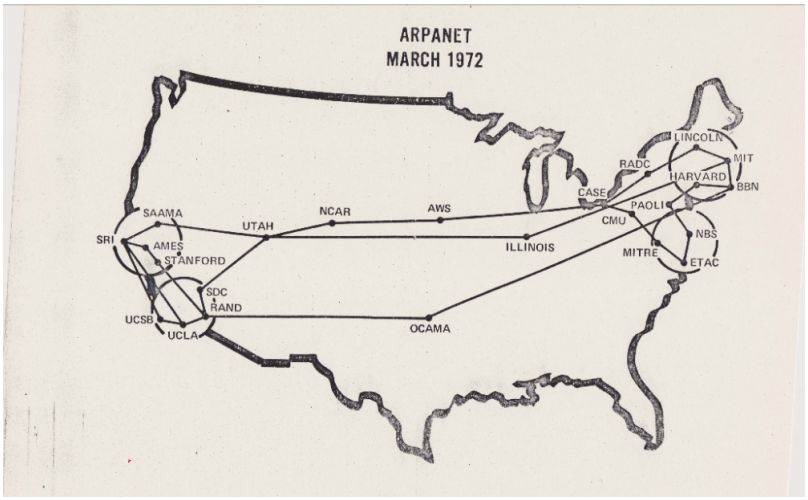


Figure 3.13: A map of the ARPANET from 1972, showing the first 29 connected nodes.

The ideas behind NLS were revolutionary, but the system itself required a lot of expertise in order to be used. As sarcastically remarked by Larry Tesler, who worked on word processing in 1970s at Xerox PARC, "they had to justify the fact that it took people weeks to memorizethe keyset and months to become proficient, so they came up with this whole mystique about 'augmenting intellect'."[65]

The NLS implementation was equally elaborate. It was built using a number of small languages that a modern-day programmer would call domain-specific. There was, for example, a Machine-Oriented Languag for low-level coding and a Content Analysis Language for text manipulation. The languages themselves were implemented in another special langauge, Tree-Meta, designed for writing compilers. This allowed the team to quickly change the system, but there was still a clear distinction between editing the code of the system and running it. That said, the NLS demonstration was not focused just on videoconferencing and collaborative document editing. It showed a number of features that prefigured what will appear in integrated developer environments several decades later. When viewing NLS source code, the programmer is able to collapse structures such as IF statements and each file also contains a list of known bugs and collaboratively edited notes recording, for example, possible future improvements.

Douglas Engelbart established a group of devoted disciplies that "regarded his vision with almost a religious awe". As one of them remarked, "he not only made sense; it was like someone turning on a light"[66]. However, it took several more years before others tried to implement similar ideas outside of the tight-knit Engelbart group.

Almost Anything Goes

In the 1960s and the 1970s, work on interactive programming was done at a small number of research centres and institutions. They were loosely connected through conferences and meetings, including those organized by IPTO, but also through the movement of individuals between the centres and later virtually through the ARPANET. The map of ARPANET from 1972 (Figure 3.13) shows many of the centres. It includes Harvard, Lincoln Lab, MIT and BBN in Boston where the first computer hackers experimented with interactive pro-

gramming on TX-0, TX-2 and PDP-1 and where LISP, LOGO and time-sharing were developed. It includes the University of Utah where Ivan Sutherland of the Sketchpad fame was based and where Alan Kay, who we will encounter in the next section did his PhD. It also includes the West coast hub with SRI where Engelbart worked and UCLA from which the first packet-switched message was sent to SRI in 1969.

A key role in the development of the community was played by the ARPA Information Processing Techniques Office (IPTO), which was initially called Command and Control Research office. The office was established by ARPA in order to kill two birds. First, they wanted to get into command and control research, but second, they also had a rather expensive computer, built as a backup for the SAGE air defence system, that they no longer had any use for. The computer was sent to System Development Corporation (SDC) in Santa Monica and used for war game scenario simulations, but the IPTO soon started planning further research. The focus on command and control was fortunate as it aligned with much of the thinking around interactive programming. This is clear from the initial purpose of IPTO, which was "to support research on the conceptual aspects of command and control and to providea better understanding of organizational, informational, and man-machine relationships."[67]

After establishing the IPTO, the ARPA managers approached J. C. R. Licklider to become the first IPTO director. Licklider was a fortunate find as his work on man-computer symbiosis addressed exactly the kind of command and control problems that the IPTO was supposed to focus on. Under Licklider's leadership, IPTO started funding work on time-sharing, graphics, augmenting human intellect and others. Crucially, the projects also formed an "ARPA community" that brought together a diverse range of people from the hacker, engineering and humanistic cultures of programming. The ARPA community continued to develop under the leadership of Licklider, Ivan Sutherland and later Bob Taylor who moved from his earlier position at NASA. The project investigators met regularly at meetings that stroke the right balance between being competitive and collaborative, but much of IPTO funding went to students and helped to "win the hearts and minds of a generation of young scientists."[68] Under Taylor's leadership, the IPTO also started organizing regular meetings specifically for the students it funded, which provided nourishing environment for future leaders of the field.

The IPTO was pivotal in the development of time-sharing and ARPANET, a direct predecessor of the Internet, but by the start of the 1970s, its focus has shifted from basic research towards applied work. Fortunately for the ARPA community, another organization opened in 1970 and provided a new home for many from the ARPA community. The new organization was the famed Xerox Palo Alto Research Center (PARC). Over the next decade, Xerox PARC played a renowned role in the development of modern graphical user interfaces, local networking technologies like Ethernet, object-oriented programming and laser printers. Xerox PARC taped into the same mix of hacker, engineering and humanitarian cultures as the ARPA community. Soon after its founding, its leadership hired Bob Taylor who eventually became the manager of the Computer Science Laboratory at PARC. As in IPTO, he played a role of a visionary and community builder who, somehow, managed to get a large group of computer scientists with diverse backgrounds and interests to work together.

Much has been written about the history of IPTO, Xerox PARC, their key figures and their intertwined histories, as well as the influence of the 1960s counterculture on the developments.[69] The detailed accounts reveal many links and sometimes unexpected con-

nections. For example, Stewart Brand who was the publisher of the counterculture magazine Whole World Catalog assisted Doug Engelbart as the video assistant during "The Mother of All Demos". Three years later, in 1972, Brand visited Xerox PARC and wrote an article "Spacewar" for the Rolling Stone magazine, a title of which was a nod the computer game that I opened this chapter with. The article portrayed the Xerox PARC researchers and programmers, including Bob Taylor and Alan Kay, as brilliant, uninterested in conventional goals and with *"plenty of time for screwing around"*, a portrait that shocked the Xerox management and caused much tensions between PARC and the higher Xerox management. The countercultural influences were also present in the experiments with LSD, which were often done as part of (more or less) controlled research. Engelbart was interested in those[70] as his work on augmenting human intellect was closely paralleled with the idea of enhancing creativity with psychedelic drugs. Engelbart's own experience with LSD did not lead to major new discoveries, but the experiences of many others differed. For example, the Hypercard system that I discuss later in this chapter and which was a groundbreaking hypertext system, was conceived by Bill Atkinson during an LSD trip in 1985.

The ARPA community and Xerox PARC are, however, also interesting as a place where multiple different cultures of programming meet. Not just in terms of the broader social environment, but also in more specific terms of how they thought about programming and computing. The early researchers at Xerox PARC came from a wide range of backgrounds and brought with them different ways of thinking about computers. PARC hired Peter Deutsch, a well-known hacker and a former member of the Tech Model Railroad Club (TMRC) who implemented LISP for PDP-1 as a high-school student and created the first version of the LISP editor as a student. The group also recruited Bill English, an engineer who was the first member of Douglas Engelbart's group and was instrumental to producing The Mother of All Demos. Further engineers included Chuck Thacker, who worked on hardware design for a time-sharing system at Berkeley previously. His engineering spirit is captured by his lifelong fight against "biggerism"; in his computer designs, one "never found a logic gate or a ground wire out of place."[71]

Finally, PARC also had a fair share of members inflenced by the humanistic culture. The best known is perhaps Alan Kay. Kay struggeld with the rigid education system, but eventually completed PhD with Ivan Sutherland in Utah and joined Xerox PARC soon thereafter. He publicly declared that it is fine to use $3M machines to play games and "screw around" in the Roling Stone article by Stewart Brand. Kay saw computers as creative tool. He joined PARC in order to build Dynabook, a personal computer "which could be owned by everyone and could have the power to handle virtually all of its owner's information-related needs."[72]. Kay's influences clarly link him to the humanistic culture of programming. He was influenced by Marshall McLuhan's work on media theory, Jean Piaget's psychology, but also, more directly, by Seymour Pappert's work on LOGO and Ivan Sutherland's Sketchpad.

Xerox PARC eventually settled on the research program of envisioning the office of the future. This was able to accommodate a wide range of hacking, engineering work and research inspired by humanistic concerns. It included the work of Alan Kay, who was really interested in computers for children, but built systems that were equally interesting for future office systems. However, the focus excluded some of the more serendipitous creative work done by artist. For example, Richard Shoup and Alvy Ray Smith who briefly collaborated at PARC in 1974 and produced first computer animations departed too far from the core research program and did not stay around for long. Shoup continued to

work on graphics software independently and Alvy Ray Smith went on to co-found Pixar Animation Studios.

The meeting of cultures around IPTO and Xerox PARC provides an example where interesting new developments occur when multiple cultures meet. This was the case for the birth of programming langauges in Chapter 2 and it will be the case for the development of types in Chapter 5. The advances on interactive programming are no different. The case of interactive programming is perhaps exceptional in that it brought together people from many different cultures for a relatively long time. In addition to the hacker, engineering and humanistic cultures, the mathematical culture also played its part although less prominently. First, both Papert and Kay were interested in computers as a medium for teaching rigorous mathematical thinking, even though their own work did not lead to the typical outcomes of the mathematical culture. Second, many of the MIT hackers had clearly mathematical interests, as illustrated by the many mathematical puzzles discussed in the HAKMEM memo[73] which is a canonical output of the hacker culture. Most prominent among them is John McCarthy, who wrote seminal papers of the mathematical culture discussed in Chapter 2. Finally, Xerox PARC also employed direct contributors to the mathematical culture of programming, such as James H. Morris, whose PhD thesis on lambda-calculus models of programming languages is covered in Chapter 5 when talking about types.

One culture that is conspicuously absent from the list is the managerial culture of programming. In the history of interactive programming, there seems to be little room for well-designed organizational structures, careful advance planning and formalized processes. To paraphrase Paul Feyerabend's mantra of epistemological anarchism, when it comes to interactive programming *almost* anything goes. The community was keen to embrace different methods and principles and often overcome the differences, but this did not apply to managerial methods. This may have been for multiple reasons. First, interactive programming requires direct and often uncontrolled access to computers which is often at odds with managerial methods. The countercultural leaning of the community may be another reason, especially at a time of growing tensions between programming personnel and management[74] illustrated by the "Unlocking Computer's Profit Potential" report that I return to in Chapter 4. There are also the personal characters of the individuals. Bob Taylor is widely regarded as an exceptional research manager who built a devoted team of researchers, but his aversion to rigorous management methods resulted in frequent clashes and, ultimately, his departure from Xerox PARC. One may only speculate whether more openness towards the managerial culture of programming would, for example, make more of the ground-breaking technologies invented at Xerox PARC easier to commercialize.

Personal Dynamic Media

The meeting of cultures at Xerox PARC created an environment where many thought about programming in a very different way than the creators of COBOL, Algol or even modern programming languages like Java, Python or Haskell. Alan Kay is now best known as the designer of the Smalltalk programming language, but his interest at Xerox PARC was always building a portable computer that would be easy enough to use for children, while being powerful enough to offer the full flexibility of a programmable computer.

Following this humanistic vision, Kay also saw *using* the computer and *programming*

Figure 3.14: Alto with storage disks, computer, vertical display, mouse and a keyboard.

it as closely interconnected. When you see a computer for the first time, you may only try running the code that is already there, but as you learn, you will want to be able to customize and modify the preexisting programs and eventually, develop new programs on your own.[75] In other words, an iPad or other modern tablet may look much like the tablet computer sketched by Kay in Figure 3.2, but the way of working with it is nothing like what Kay was trying to achieve. The fundamental difference is that programming was an inherent part of using the Dynabook. An iPad makes a strict distinction between a user and a programmer and you cannot gradually progress from one to the other without leaving the system. The Dynabook was supposed to allow this.

Kay imagined that hardware developments will make it possible to build an actual Dynabook within a decade or two. To experiment with ideas, he wanted to build an "interim Dynabook", which would be powerful enough to show what working with Dynabook would feel like. This coincided with the interests of two PARC engineers, who took it as an opportunity to build a personal computer with an even broader range of uses.[76] The work resulted in an influential machine Xerox Alto shown in Figure 3.14. About 30 machines were built initially. They soon became popular across PARC, as well as outside, and over 1000 of Altos was eventually built in total.

Equipped with a machine, Alan Kay and his team started working on implementing the Dynabook vision. The best account of the overarching vision is the "Personal Dynamic Media"[77] paper, co-authored with a PARC collaborator and co-inventor of Smalltalk, Adele Goldberg. The paper uses a language influenced by media theorist Marshall McLuhan and describes the goal of the group as desinging "dynamic media which can be used by human beings of all ages" that "could have the power to handle virtually all of its owner's information-related needs." Smalltalk is presented not primarily as a programming language, but as "a new medium for communication" with the machine.

Kay and Goldberg saw computers as a *metamedium* that can be used to simulate, or *become*, all other media, including a "wide range of existing and not-yet-invented media". The example programs described in the paper capture the expected way of using

the system. They include “An Animation System Programmed by Animators”, “A Hospital Simulation Programmed by a Decision-Theorist” or an “Electronic Circuit Design by a High School Student”. The expectation is that a user will first use the metamedium of Smalltalk to construct a medium for solving the kind of problems they are interested in, be it animation, modelling or circuit design. The user will then use the resulting medium, or program, to create animations, model decision theory problems or design circuits. Programming is thus a new kind of literacy that everyone will master in order to use computer as a flexible tool for their work.

The idea that anyone would be able to use a computer in an unrestricted way through Smalltalk was an appealing vision to the proponents of the humanistic culture of programming. The reality turned out to be more problematic. The team taught the first practical version of the language, Smalltalk-72 to a local middle school children and, despite some positive experience, many kids struggled with rudimentary Smalltalk concepts. Many non-professional adults working at PARC who Kay and Goldberg tried to teach Smalltalk faced similar challenges.[78]

In January 1976, during a team research retreat Kay tried to convince his team to start afresh and make a new attempt at producing a communication medium for anyone to use. However, by this time, Smalltalk was already a sizeable project that had a life of its own. Dan Ingalls, who did most of the system implementation, certainly did not want to abandon Smalltalk, but instead wanted to turn it into a full programming system. Kay started to work on a new notebook and language called NoteTaker while Ingalls reengineered the Smalltalk implementation, producing Smalltalk-76 and then led the development of subsequent versions. Here, the Smalltalk language followed the suit of the Xerox Alto machine. The PARC community was eager to use its own tools and both Alto and Smalltalk started to be used by others for their own purposes and experimentation.

Smalltalk may not have became the universal medium for communication with a computer, but even as a programming language, it retained many of the ideas from which it was born. Most importantly, Smalltalk was not a programming language in which you would write a complete program that would then be run. Instead, you were interacting with an environment that already contained various definitions that you could reuse and modify. In this, it was similar to Lisp systems like Interlisp, but the programming model of Smalltalk was based on objects, a topic we will return to in Chapter 6. Smalltalk 72 also pioneered using graphical interface for programming. You wrote your commands in a dialog window at the bottom of the screen. When editing a definition, the window becomes a structured editor, logically similar to that developed by Deutsch for LISP, but controlled using a mouse and menu, rather than a terminal console. Smalltalk 72 also followed LOGO in the attempt to be more friendly through the use of non-technical language. The manual written by Adele Goldberg and Alan Kay[79] introduces programming as “talking to Smalltalk” and the language itself uses graphical symbols like a hand for defining a new symbol and a smiley face, as a value representing a turtle, both of which appear in Figure 3.15 (left).

After the January 1976 retreat, Smalltalk was not abandoned and it did, indeed became a more comprehensive programming environment. This effort was led by Dan Ingalls and resulted in Smalltalk 76, which was more efficient, fully supported object-oriented concepts like inheritance and came with a graphical interface, shown in Figure 3.15 (right). The graphical interface included the iconic Smalltalk object browser, a window that can be used for browsing the different class definitions that exist in the system and for viewing or editing their methods.

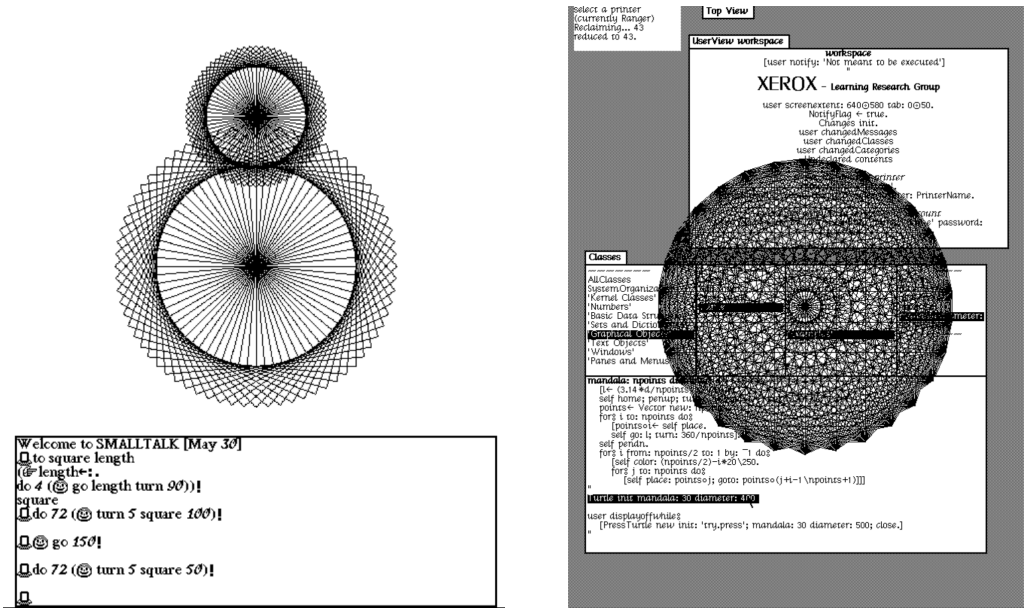


Figure 3.15: Turtle in Smalltalk 72 (left) and Smalltalk 76 (right). In Smalltalk 72, the screenshot shows a dialog window with entered code and the result of running a turtle code. In Smalltalk 76, the screenshot shows the "mandala" operation in the class browser and the result of running an example included in the method definition.

Alan Kay started working on ideas leading to Smalltalk during his PhD at University of Utah, which he did in 1966–1969. This is only a few years after the Algol language appeared in 1958. Many of the important developments in the Algol research programme that I discussed in Chapter 2 happened around the same time. This includes the rising popularity of structured programming, development of mathematical tools for reasoning about programs and the very idea that a program should be treated as a mathematical entity that can be formally analyzed. Smalltalk was influenced by Algol in various technical aspects, but it belongs to a very different culture of programming. In Smalltalk, you use a computer by interacting with it, sometimes through writing code. You work by modifying a rich environment. There is no clear "program" that you could consider in isolation. Smalltalk is the culmination of ideas leading to interactive programming. It combines the interactive graphical way of using computers, envisioned by humanists augmenting human intellect and implemented in Engelbart's oNLine System (NLS), with the direct way of interacting with running programs developed by hackers that culminated with Interlisp.

The development of interactive programming no doubt benefited from the mixing of different cultures of programming. Smalltalk appeared as a system motivated primarily by the humanistic vision of programming for everyone, but had many other influences and contributors. It did not achieve its original goal, but it was adopted by the hackers and engineers at Xerox PARC for other projects. It is easy to imagine that, if it was not for the people leaning towards other cultures of programming who took leadership of the project and developed Smalltalk further, it would remain just an abandoned attempt at programming for "human beings of all ages".

Articulate Languages for Communication

Today, Smalltalk is remembered as the programming language that crucially contributed to the development of object-oriented programming. This is true and I follow this strand in Chapter 6. However, Smalltalk is also a flagship interactive programming system that brought together the humanistic vision of programming as communication with graphical user interfaces. In this, it has been followed by number of programming systems from the late 1970s until today. Although the systems following Smalltalk were not as influential, it is worth looking at two examples as they illustrate the way of thinking about programming in the humanistic culture.

The first is Pygmalion,[80] which appeared in 1975 and is a "two-dimensional, visual programming system" implemented on top of Smalltalk-72. The second is Boxer[81] which appeared in 1985 and is an "integrated computational environment suitable for naive and inexperienced users". Aside from sharing the cultural context, the two systems are both visual programming systems in that they represent the program in a graphical way and let the user construct programs through visual representation. In Smalltalk, graphical interface was used for structured editing (in Smalltalk-72) or for browsing through available objects (in Smalltalk-76 and newer). In Boxer and especially Pygmalion, graphical interface is used for constructing the program logic itself. Furthermore, Pygmalion and Boxer both motivate this idea through arguments that are centered around creative thought.

Pygmalion was created by David Canfield Smith during his PhD at Stanford University. Smith joined Stanford in 1967 with a goal to develop computer that would be able to learn. He eventually changed his direction and approached Alan Kay to be his thesis supervisor. Following Kay's advice, Smith read a number of books on philosophy and art and his work was greatly influenced by ideas on creative thought and schematic thinking from "Art and Illusion" by Ernst Gombrich.[82]

Pygmalion uses two-dimensional graphical representation because, as Smith argues, visual images provide more descriptive power. Using a term borrowed from Gombrich, the aim of Pygmalion is to design an *articulate* langauge for communication with a computer, i.e., a langauge that "corresponds closely to the form used in the mind in thinking about the problem". Pygmalion, shown in Figure 3.16 (left) is based on two key principles. The first is the principle of *icons*, concrete visual representations of an idea that appear on the screen. In Figure 3.16, the icons are mostly labels in boxes, but the thesis also includes a more graphical example of electric circuits. The second principle is *programming by demonstration*. A program is constructed not by "telling" the computer what to do but instead by "doing" it. The Pygmalion system implements a "remembering editor for icons", which stores the sequence of actions done by the user and is able to re-execute them. Incidentally, the same recording mechanism was also created in 1974 to teach LOGO programming to young children (Figure 3.17).

The Pygmalion programming system itself could have been described in a narrow technical way, but Smith's PhD thesis does the exact opposite. It's very subtitle "A Computer Program to Model and Simulate Creative Thought" makes a reference to creativity and its first two chapters discuss "a psychological model of creative thought, forming the basis for the Pygmalion design principles." The thesis also contains numerous references, arguments and examples that connect it with the humanistic culture. These include references to the LOGO programming language, Turtle graphics, Sketchpad, but also the Spacewar!

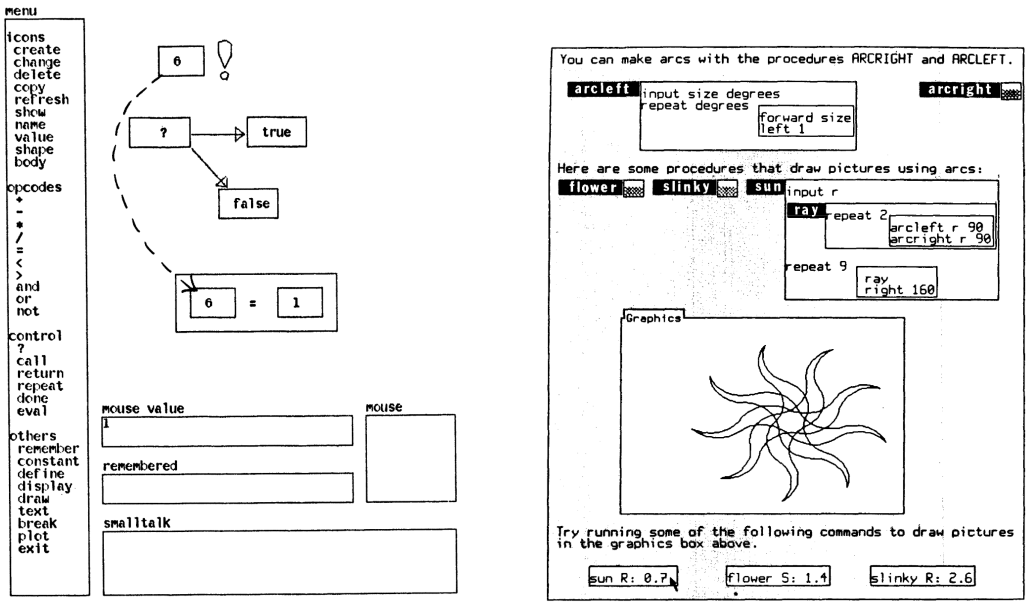


Figure 3.16: Pygmalion from Smith (1977) (left) and Boxer from diSessa and Abelson (1986) (right). The Pygmalion example shows a step in the process of computing the factorial function and the Boxer example shows a simple turtle graphics program.

game developed by the MIT hackers for PDP-1 in 1962. Those references serve as *cultural pointers*[83] that connect the thesis to the broader humanistic culture of programming.

The motivation for Boxer was more specific. Andrea diSessa, who started the project is a trained physicist, but became interested in education and joined Papert's LOGO group at MIT. The goal for Boxer was to build a medium that could be used for creating an interactive physics textbooks. A book on optics might include structured text, alongside with simultations that perform ray tracing through lens and mirrors. The medium for such book should be easy to program, but more importantly, it should be *reconstructible*. The user should be able to see the programming, understand it and modify it to fit their needs.

Boxer, shown in Figure 3.16 (right), implements a reconstructible computational medium using two key concepts. The first concept the use of a *spatial metaphor*. Programs are represented visually, in a two-dimensional space as nested boxes. The boxes can contain text or code, itself represented using nested boxes. For example, in the figure, a box `repeat 2` contains a nested box with the code to be repeated twice. Boxes can also be collapsed to save screen space. The second concept is *naive realism*, which is the idea that what you see on the screen are the computational objects themselves. In Boxer, the program is not hidden. You see it all in the form of nested (possibly collapsed) boxes. A procedure is a box, code to call a procedure is another box and visual output generated by running it is yet another box. An interesting consequence is that the programming language *is* the user interface. There is no need for buttons, because you can run a simulation by clicking on the box that represents the code for the simulation.

There are many reasons for viewing Boxer as a product of the humanistic culture of programming, including the focus on education, the fact that diSessa was affiliated with the LOGO group at MIT, but also the different arguments that support various design choices in the papers about Boxer. Like Pygmalion, Boxer's use of a visual representation is moti-

Figure 3.17: Button box developed by Radia Perlman allowed young children (aged 3-5) control and progrma LOGO turtle by issuing instructions, but also by recording and then replaying sequences of instructions.

vated by an argument about human cognition. People have a commonsense knowledge about space and the system can leverage that to help users "interpret the organization of a computational system in terms of spatial relationships". The references in papers on Boxer include Pygmalion and a large number of "integrated computational environments" that target experienced programmers such as Smalltalk and Interlisp.

Ideas from Pygmalion and Boxer all found their way into modern computing and programming tools, but often in ways that leave some of the original motivations behind. David Canfield Smith, who created *icons* in Pygmalion later joined Xerox to design the user interface of Xerox Start, a computer office system that was an attempt to commercialize some of the technology developed at PARC. The design used icons in a way we would recognize today, but without any of their original computational nature.[84] Boxer led the way to contemporary block-based visual programming environments for kids like Scratch, but those only keep the idea of visual programming, not that of reconstructible media.

Projects such as Pygmalion and Smalltalk aimed to make programming accessible to everyone, but they run on computers like Xerox Alto that were still too expensive to be owned by an individual. The systems started as research projects and their creators correctly expected that the exponential growth of computer power will soon solve the issue of cost. The cost of computers did, in fact, decrease, but this brought very different kind of machines from those that Alan Kay and his group imagined.

Homebrew Computer Club

Since its establishment, Xerox PARC was subject to tensions between the researchers who pursued curiosity-oriented approach and trusted scientific serndipity and the Xerox management that saw PARC largely as a way of preparing for potential future threats to their existing business model.[85] In 1971, PARC researchers reluctantly respondend to the pressure from management, embodied by Xerox planning executive Don Pendery, and produced a brief report on the future of computing that was cheekily labelled "PARC Papers

Figure 3.18: Popular Electronics magazine cover featuring the Altair 8800 microcomputer.

for Pendery and Planning Purpose"[86]. The authors described some of their visions including many relevant for the "office of the future" and estimated that their technology should be commercializable in the early 1980s. Xerox eventually tried to do this. In 1977, it established the Systems Development Department, which started to building a system inspired by Xerox Alto that would be usable for office tasks. The system was released in 1981 and was in many ways an impressive technology. It featured graphical user interface based on icons as we know them today, a modern word processing application, all connected using Ethernet with laser printers. But the system remained too expensive and Xerox was unable to find an effective marketing strategy for seeling it. Smalltalk itself was not publicly released until 1980 and even then, it took several more years before it became efficient enough on commodity hardware to be used for business application development.

In the meantime, a new kind of computers appeared. Throughout the 1960s, hardware manufacturers started producing increasingly sophisticated integrated circuits until finally, in 1971, Intel released a complete general-purpose microprocessor, Intel 4004, available as a single chip. Intel was at first unsure about marketing the chip, but they eventually began offering the 4004 as a "computer on a chip". This allowed a small company named MITS, which initially focused on producing telemetry modules for rocket models, to produce a cheap microcomputer that they called Altair 8800. Altair was a kind of machine that its creators wanted for themselves and they started advertising it to fellow computer hobbyists. Many of the early adopters did not have any specific use for it; they just wanted to play with a computer of their own. Following an article in the Popular Electronics magazine (Figure 3.18), where it could be ordered for $397 (the equivalent of about $2000 today), the machine became an instant hit.[87]

Altair 8800 was nothing like a modern personal computer. It was also nothing like the Xerox Alto that run Smalltalk. It was much more like the 1955 TX-0 machine, but even that came with a built-in tape reader and a 12" oscilloscope screen. The Altair 8800 was initially sold as a kit that you had to assemble yourself. The only way to control it was through

Figure 3.19: Front panel of Altair 8800, showing the status lights, a row of switches for specifying addresses and entering data and a row of control switches.

a sequence of switches on the front and the only way it could respond was by flashing lights. You might even be reminded of the ENIAC computer with its flashing lights covered by halves of Ping-Pong balls in 1946 that I mentioned in the opening of Chapter 2. The dramatic change was that Altair 8800 was affordable enough to reach a new generation of hackers, eager to get their hands on a new electronic gadget and explore what can be done with it.

Using just switches and lights, you couldn't do all that much with the Altair, but the system had a detailed specification that allowed you to modify it. You could build or buy extensions that let you connect the Altair to a paper tape reader, teletype or a video terminal. To load a program from a paper tape, you first had to use the front panel switches (Figure 3.19) to manually enter instructions of a bootloader, which was a very compact program that copied data from a tape into the Altair memory. Then, you would flip a switch to read the program and another to actually run it. If you connected the Altair to a teletype or a video terminal, the program could then interact with the user by printing to and accepting input from the terminal.

To the engineers at PARC, Altair 8800 and its early successors looked desperately primitive and the hackers working with those machines appeared as mere kids with toys. But the community of enthusiasts around those machines grew and started creating a wide range of programs. The most influential part of the community was the Homebrew Computer Club in Menlo Park, California. Many of the founding members of the club built their own microprocessor-based computers, even before the Altair was released.

The early microcomputer community that developed around the Homebrew Computer Club shared some of its characteristics with earlier contributors to interactive programming that I discussed earlier, but it also brought new perspectives. It shared much of its spirit with the 1960s MIT hacker culture in that it emphasized individual technical achievements, information sharing and the Homebrew members beleived that software should be free and shared with everyone. The club was also rooted in the same countercultural social context as Engelbart's Augmentation Research Center and Xerox PARC. The members believed in a do-it-yourself approach that was a strong focus of Stewart Brand's

countercultural bible, the Whole Earth Catalog.[88] In contrast to the early MIT hackers, the Homebrew Computer Club was more open towards commercial interests.

The fact that commercial companies were increasingly involved in the development of hardware and software for microcomputers did not initially have an effect on what cultures of programming were involved. The microcomputer community brought together hackers interested in tinkering with the electronics with humanists who saw programming as a way of advancing the human condition. They were interested in building systems that "would allow the public to take advantage of the huge and largely untapped reservoir of skills and resources that resides with the people" and believed that computer intelligence should be directed towards "demystifying and exposing its own nature, and ultimately giving [the user] active control."[89] The community believed in decentralization that is at odds with the managerial approaches to programming that were emerging in large-scale commercial computing enterprises. Attempts to transition from "black art" of programming to the science of programming, which I follow in the next chapter, were also largely ignored by the microcomputer programmers.

Programming of microcomputers was also initially not influenced by commercial companies because the most notable early businesses focused on building and selling hardware, rather than developing software. The most famous company that originated in the Homebrew Computer Club is undoubtedly Apple, co-founded in 1976 by Steve Wozniak and Steve Jobs, which started with the Apple I do-it-yourself kit. The idea that information should be free led to conflicts with those trying to profit from building software. This was the case of Bill Gates and Paul Allen, who realized the commercial potential of microcomputers, founded Micro-Soft and implemented an Altair 8800 interpreter for the programming language BASIC. The interpreter was licensed to MITS who started selling it on behlaf of Micro-Soft. In line with their belief that information and software should be free, the Homebrew community started copying and sharing the BASIC interpreter, to which Gates responded with a contentious "An Open Letter to Hobbyists" that accused them of stealing.[90] This dispute, however, did not prevent BASIC from becoming the de facto programming language for microcomputers.

In 1977, three companies released personal computers that sold in millions over the next few years and made computers available to a broad public. The three computers were Apple II, Commodore PET and Radio Shack TRS-80 and they were all influenced by the Altair 8800 or the Homebrew Computer Club in some way. Apple II was a successor of Apple I. It was also designed by Steve Wozniak, but it marks a shift from a machine for hobbyists to a machine for end-users. It was preassembled, with all electronic circuitry hidden in a user-friendly box. The Commodore company briefly considered purchasing the Apple design before producing their own machine and TRS-80 was co-designed by another Homebrow member and an Altair kit owner working at a company that owned RadioShack. Perhaps due to the Altair 8800 influence, all three 1977 microcomputers came with a BASIC interpreter that partly served as a primitive operating system. BASIC was not the only way of programming microcomputers and programmers could always choose to write code in low-level assembly. This was favored by many more sophisticated users and, later, also by companies producing advanced software for microcomputers. But BASIC was the first language that many new computer users would see and it shaped their experience with computers.

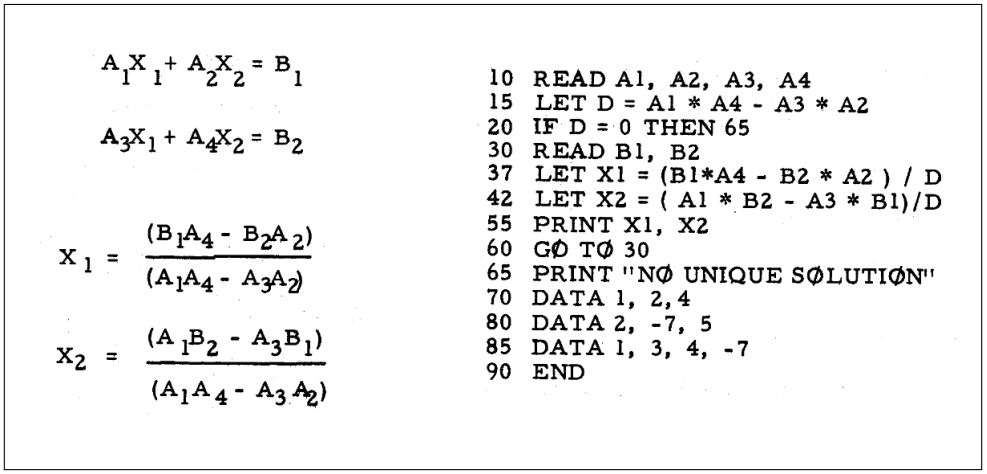


Figure 3.20: An introductory example from the 1964 BASIC manual. A program for solving a pair of linear equations (left top) implementing a simple calculation (left bottom) as a BASIC program (right).

Beginner's All-purpose Symbolic Instruction Code

The history of the BASIC language dates back to 1964 when it was designed by John G. Kemeney and Thomas E. Kurtz at Dartmouth College. BASIC ("Beginner's All-purpose Symbolic Instruction Code") was created as an language to be used for educational purposes for the new Dartmouth time-sharing computer system. It was initially designed for writing numerical programs in a way that would be easier than using FORTRAN. The idea was that every Darmouth student should have access to it and should learn it during their studies. As such, BASIC was designed to be easy to use, but also to work well on an interactive terminal.

The BASIC language itself was very simple. The introductory example from the 1964 BASIC manual (Figure 3.20) shows many of the available constructs. Programmers could define variables, read data from the `DATA` block at the end of a program, use conditionals and the `GO TO` statement for transferring the control to another part of the program. In the example, this is used to read multiple inputs and solve multiple pairs of equations. In addition, the 1964 version of BASIC also had a way of implementing sub-routines using the `GO SUB` command, `FOR` loop for iteration and a way of defining multi-dimensional variables to represent lists and tables.

Despite the simplicity of the language, there were a few features that made it powerful and interesting enough for hackers. Since the Altair, BASIC had commands `PEEK` and `POKE` that allowed the programmer to read data from and write data to any memory address specified as an argument. This gave the programmers access to everything on the machine that was not exposed in another way, such as drawing on arbitrary screen location. However, since BASIC program itself was stored in the memory, you could also write programs that looked at their own source code and modified it. In addition to `PEEK` and `POKE`, there was also the `SYS` command, which allowed invoking code at arbitrary memory location, including various system routines and assembly code written by the programmer. So despite the simplicity of the core language, the system provided enough backdoors for hackers to explore once they knew all the basic tricks.

Figure 3.21: After turning on, Commodore PET starts and immediately runs Commodore BASIC. Any further interactions were through the BASIC command prompt.

Perhaps more interesting than the language itself is the way of interacting with it. On Commodore PET, BASIC was not just a programming environment, but the operating system of the computer. When the machine started, the BASIC command line appeared (Figure 3.21) and the user could start by typing BASIC programs. Even if you wanted to play a game that you got from a friend on a tape or a floppy disk, you had to do this by invoking the BASIC command `LOAD`, which takes a file name and device number as arguments and then typing `RUN` to run it. This loaded the BASIC source code for the program into the memory of the interpreter. You could then also easily view the source code and try to modify it. Although Commodore BASIC is certainly not how Alan Kay and Adele Goldberg imagined "personal dynamic media", it was in some ways close to the idea. Unlike programming languages born from the mathematical and engineering traditions like FORTRAN or Algol, BASIC did not impose a strict distinction between programming and running a program.

Using BASIC on Apple II, Commodore PET and TRS-80 had an interactive nature not just because you sometimes typed commands in the prompt. The other factor was its ingenious, if sometimes ridiculed, use of line numbers. As Figure 3.20 shows, every line of a BASIC program started with a line number. When the program runs, the numbers are used by the `GO TO` statement. For example, line 60 in the sample program contains `GO TO 30`, meaning that when the program gets to this point, it will continue running code on line 30. However, the line numbers are also used for editing. If you load a program like the above and type a line starting with a number, BASIC will put the new line into an appropriate place in the program, possibly replacing existing code. This way, it is possible to use the same simple command prompt for both running commands and editing code. In a way, the prompt serves as an elementary structure editor. It is a very simple one, because the structure of BASIC code is just a list of lines, whereas structure editors in Interlisp or Smalltalk 72 had to let users navigate through a much more complex tree structure.

The authors of BASIC were attempting to make programming that would normally be done in FORTRAN easier, but their design was also influenced by Algol and they referred to BASIC as an "algebraic language". However, they did not adopt some of the Algol features that were to become influential in the mathematical culture of programming and became

recognized as good engineering practices. First, BASIC did not support rich data types which were long established in COBOL and were becoming standard in mathematically-minded work on programming. Second, the structure of a program in BASIC is a list of lines, rather than that of nested blocks as in Algol. This makes it harder to use ideas of structured programming and it was one of the reasons that led Edsger Dijkstra, a proponent of the mathematical approach to programming, to condemn BASIC in his 1975 note:

> *It is practically impossible to teach good programming to students that have had a prior exposure to BASIC: as potential programmers they are mentally mutilated beyond hope of regeneration.*[91]

The quote is perhaps best interpreted as a clash between the mathematical culture and the hacker cultures of programming. Using the mathematical lens, BASIC programs do not have the elegant structure of Algol programs and the interactive way in which they are created makes them harder to treat as mathematical entities. Yet, to a hacker, BASIC on a microcomputer allows you to easily experiment with computers, have fun and (occasionally) also write a program that does something interesting or even useful.

The era of microcomputers happened thanks to the combination of engineering advances that made the microprocessor possible, work of educators who thought that every student should be able to program a computer and hacker efforts to build computers one could cheaply play with. The times of Apple II, Commodore PET and TRS-80 was also perhaps the last time programming was at the centre of using computers. A typical user was a hacker who wanted to explore what can be done with computers and, accordingly, at least one of the operating systems available for each of those machines was the BASIC programming environment. Microcomputers were personal, so nobody needed to introduce a policy that "playing Spacewar! is the lowest-priority thing a computer could do." Most users played games and run other programs, but they were equally interested in fiddling with the machine to see what it can do.

This way of looking at computers came to an end when commercially minded programmers realised that useful programs for microcomputers can be built for and sold to non-hackers.

The Birth of an Industry

At the turn of the 1980s, microcomputers turned from playthings for hackers into commercial tools. This was enabled not just by the rise of more user-friendly microcomputers and their low cost, but crucially, also by the development of new ways of using computers. The purchase of an mainframe computer from IBM a decade earlier was a major investment that was typically also associated with the, equally expensive, development of custom software for a specific business task. The low cost of microcomputers would not be so interesting if their purchase also involved the development of expensive custom software. However, new kinds of software that started to appear at the end of the 1970s meant that custom software was often not needed. In particular, the growth of the microcomputer industry and its acceptance by business is largely due to the development of end-user software applications for text processing, spreadsheets analysis and database management.[92]

In 1974, Butler Lampson and Charles Simonyi developed the Bravo editor at Xerox PARC for the Alto machine. Bravo was a WYSIWYG (What You See Is What You Get) editor that

supported text formatting, fonts and many other features familiar today. Bravo directly influenced Xerox Star, but the work was not known outside Xerox. The first text processor for microcomputers, Electric Pencil, appeared in 1976 without any influence from Bravo. The software was rather rudimentary when compared to Bravo, but it became popular in the microcomputer community and was soon imitated and surpassed by WordStar and later WordPerfect. Yet, the capabilities of Bravo were only matched by graphical text processors for Apple Lisa and Apple Macintosh in mid 1980s.

The case of spreadsheets is more interesting for the history of interactive programming because working with spreadsheets is closer to programming. The first spreadsheet software for microcomputers was VisiCalc, developed by Dan Bricklin and Bob Frankston in 1979.[93] VisiCalc was initially sold for Apple II and it became the reason why many companies purchased the machine. It was a powerful tool for various financial calculations such as examining alternative budget plans ("what if" problems). VisiCalc owed its debt to the previous generation of interactive computers. Both Bricklin and Frankston worked on the time-sharing system Multics at MIT and Bricklin later worked on word processing software at DEC. VisiCalc was a commercial product of an entepreneurial spirit. It was not intended as a programming tool, but rather a specific business tool. The 1983 advertisement for Apple with VisiCalc makes the focus clear: *"Teamed with VisiCalc financial software, an Apple can help anyone make better, smarter, faster business decisions."* Like Bricklin and Frankston, many others who worked on earlier interactive programming projects moved to end-user software development for microcomputers at the turn of the 1980s. David Canfield Smith who created the Pygmalion system later joined Xerox to work on the Xerox Star user interface. Here, he developed the idea of icons from Pygmalion into something much more akin to present day icons in Mac OS or Windows, but in 1983, he moved to VisiCorp, a company selling VisiCalc. Similarly, Charles Simonyi, who created the Bravo document processing system, joined Microsoft in 1981 and started working on a program that would become the first version of Microsoft Word.

The third must-have application for business use of microcomputers became dBase, a database management system first released in 1979. In the first version, dBase allowed users to define a database, manually enter and modify data, but also to write simple data queries. Unlike modern database systems, which operate as services that other applications connect to, dBase was an end-user application. When you defined a database, dBase automatically created a text-based user interface for entering data into the database. Most operations with dBase database were triggered through commands entered in a simple interactive prompt (much like that in BASIC). Working with data was done through a "record pointer" that referenced one particular row in the database. You could navigate over the records using commands such as `GOTO`, which jumped to a specified index, or `FIND`, which searched the database for a record containing a certain value.

Both VisiCalc and dBase allowed limited kind of interactive programming in a narrow domain, but they made the basic kind of programming they supported accessible to a wide range of non-experts. Incidentally, the same remains the case with modern spreadsheet applications like Microsoft Excel. In a way, VisiCalc and dBase are two specific examples of new media that Alan Kay and Adele Goldberg wished for in their "Dynamic Personal Media" article. Their dream did not come true. Visicalc and dBase were not created using metamedium such as Smalltalk that would allow their users to modify the systems themselves. They were implemented in assembler and were commercial applications designed for a specific task. This is, finally, where the managerial culture of programming started to

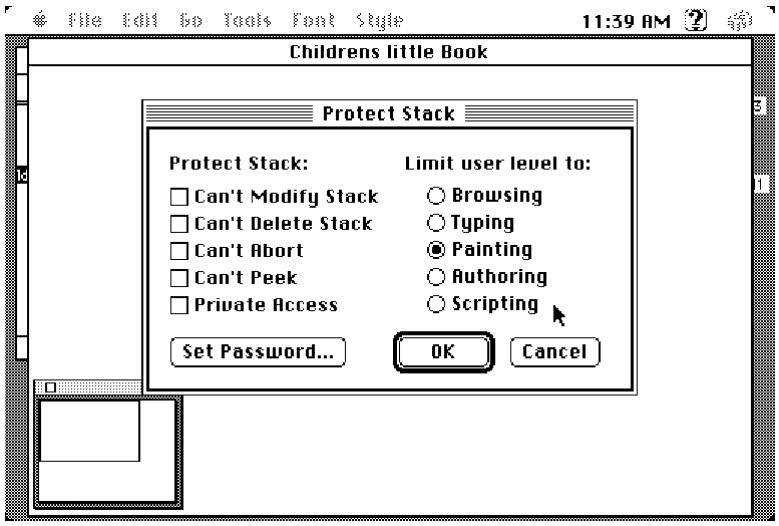


Figure 3.22: Options for protecting the stack and choice of user levels in HyperCard

shape the programming and using of microcomputers. Not in the sense that VisiCalc and dBase were commercial applications, but in the sense the they were well-defined products that draw a clear distinction between their developers and their users. This distinction is not typical in the hacker and humanitarian cultures. For hackers, everyone interacting with a computer should be a hacker. The humanitarian visions typically aim to be more inclusive. You start as a user, but can gradually progress and gain access to the capabilities that are available to developers.

A remarkable software application that brought back some of the perspectives of the humanistic culture is Hypercard, which was released by Apple in 1987 for their Macintosh system. Although Hypercard was an application, rather than a general purpose programming system like LISP or Smalltalk, it had no fixed application domain. In HyperCard, users created “stacks” of “cards” that could contian user interface elements and data. Like modern-day web pages, cards can be connected through links. Hypercard stacks are used and created through the same user interface. The interface displays the current card and the user can interact with it by clicking on buttons and following links. The interface, however, supports multiple modes (Figure 3.22). In browsing mode, you could not modify the stack; the typing mode allowed you to change text on cards; the painting mode allowed you to change the visual properties of user interface elements, while the authoring mode allowed you to create new controls and specify their functionality through menus, for example to create a link to another card. Finally, the scripting mode allowed you to specify more complex behavior using the HyperTalk programming language. The progression from a user to a programmer, which we saw implicit in many systems originating in the humanistics culture of programming, is made explicit in HyperCard. However, Hypercard was also mindful of commercial users in that it allowed them to “protect the stack” to prevent others from modifying it (Figure 3.22).

Despite being a commercial application like VisiCalc or dBase, many characteristics of Hypercard link it to the humanistic culture of programming, as well as the coutnercultural origins of interactive computing. The creator of Hypercard, Bill Atkinson, recalled in an interview how the idea for Hypercard came to him while sitting on a park bench at night after taking a dose of LSD:[94]

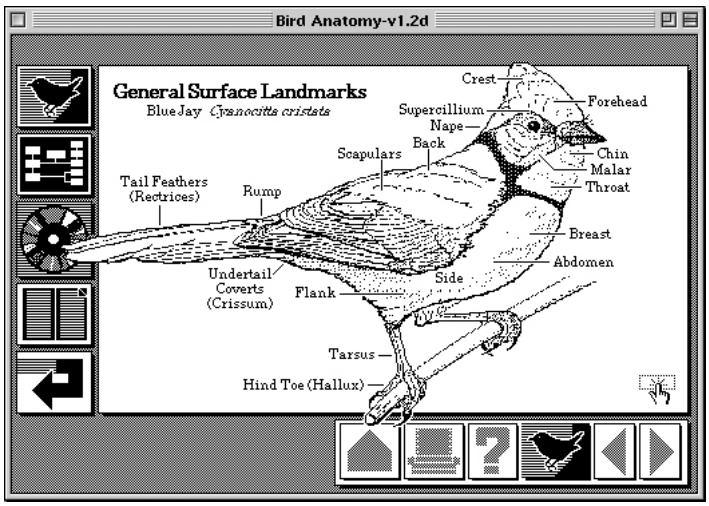


Figure 3.23: HyperCard Bird anatomy stack

> *Poets, artists, musicians, physicists, chemists, biologists, mathmeticians, and economists all have separate pools of knowledge, but are hindered from sharing and finding the deeper connections. (...) How could I help? By focusing on the weak link. (...) It occurred to me the weak link for the Blue Marble team is wisdom. Humanity has achieved sufficient technological power to change the course of life and the entire global ecosystem, but we seem to lack the perspective to choose wisely between alternative futures. (...) I thought if we could encourage sharing of ideas between different areas of knowledge, perhaps more of the bigger picture would emerge, and eventually more wisdom might develop. (...) This was the underlying inspiration for HyperCard, a multimedia authoring environment that empowered non-programmers to share ideas using new interactive media called HyperCard stacks.*

The motivation for Hypercard resembles motivations that we saw earlier in this chapter, be it the man-machine symbiosis of J. C. R. Licklider or augmenting human intellect of Douglas Engelbart. Atkinson wanted to let people passionate about any subject, who are not programmers, use computers to their full potential. HyperCard maybe did not change how the "Blue Marble team" manages wisdom, but it succeeded in the goal of empowering non-programmers to share ideas. It was surrounded by an enthusiastic community that used it for a wide range of projects, ranging from creative uses like interactive games, educational materials (Figure 3.23) and art pieces to commercial applications like databases and control systems. HyperCard gave users much more flexibility than VisiCalc or dBase and is perhaps closer to systems like Boxer. Yet, the user still needs to operate within the constraints of the stacks of cards format.

Show Us Your Screens!

Microcomputers became a new platform for work on interactive programming. They provided an impetus for the development of the BASIC programming language and enabled the birth of commercial software. However, the same technology also allowed new creative uses of computers in the art scene. As was the case with the early computer art which

influenced the educational LOGO language, the computer art done using microcomputers also later influenced interactive programming tools for education. This time though, we need to look at computers and music.

In the 1970s, a number of musicians from the experimental music scene in Oakland started to incorporate microcomputers in their work. Many of them were already building their own electronic systems, but microcomputers allowed the electronic system to become "a musical actor, as opposed to merely a tool."[95] The community started using the KIM-1 microcomputer, which had similar capabilities as the Altair 8800. KIM-1 was a single printed circuit board with a 6 digit display and 23 buttons. It was indended as a development board for a new MOS 6502 microprocessor, but it became popular with hobbyists, partly because its expansion connectors allowed it to connect to devices such as lights, switches and speakers. Several of the experimental musicians used KIM-1 to generate early algorithmic music, but a more radical use of the machine was to directly control musical devices. Moreover, you could also connect multiple KIMs in a way where one player could respond to actions of another. By 1978, an Oakland group started performing using KIMs under the name "The League of Automatic Music Composers".

The group played for several years, but they soon realised that they had to make complicated ad-hoc connections between the individual computers each time they played (Figure 3.24). As one of the group members recalls, "this made for a system with rich and varied behavior, but it was prone to failure, and bringing in other players was difficult."[96] To "clean up the mess", the group designed a new architecture where a central microcomputer, The Hub, is used for passing messages between the individual players connected to it. The architecture gave name to a reborn computer music group, The Hub.[97] Their performances evolved in many ways over the next decade. They adopted the MIDI communication protocol, when it appeared at the end of the 1980s. They also started using computer screens that the audience could view to see how the performers interact with their instruments, which was in a way a logical next step to showing the bare KIM-1 circuit board that was used in the early experiments.

The history of computer music meets with the history of interactive programming in a number of ways. The Hub was a part of the same counterculture movement as the Homebrew Comptuer Club and early adopters of microcomputers. Some of the early programmable music synthesis systems used the LISP language for its interactivity. At a more fundamental level, using a computer as a music instrument is only possible through an interactive programming system that allows immediate program execution.

At the turn of the millenium, a new generation of computer musicians appeared. As more powerful computers became available, their focus shifted to controlling music composition by interactively writing code in high level programming languages, an approach that became known as "live coding". During a live coding performance, musicians type code on a computer and run it interactively. The performers also typically project their screens so that the audience can see their work and they may also use code to control visual effects on screen. Incidentally, there is an interesting parallel between the development of live coding, which originates in the humanistic culture of programming, and the development of agile methodologies, in the engineering culture at the same time, which I discuss in Chapter 4.[98] In particular, the idea of sharing code on screen with the audience is similar to the idea of pair programming where two programmers work on a single machine, one typing code and one watching, commenting and reviewing code.

In 2004, the live coding community came together to form TOPLAP (Temporary|Trans-

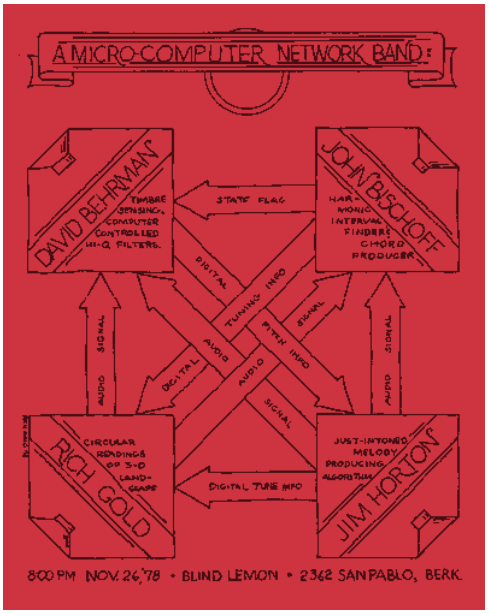


Figure 3.24: Flyer from the League's Blind Lemon concert of Nov. 26, 1978 featuring diagram of League topology. Designed and drawn by Rich Gold.

national|Terrestrial|Transdimensional) Organisation for the (Promotion|Proliferation|Permanence|Purity) of Live (Algorithm|Audio|Art|Artistic) Programming. The TOPLAP manifesto[99] exhibits many of the aspects of the humanistic culture of programming. It puts code at the center with the slogan "Obscurantism is dangerous. Show us your screens". Unlike some of the earlier works originating in the humanistic culture, it takes a more realistic view on readability of code and acknowledges that "it is not necessary for a lay audience to understand the code to appreciate it." The organization also emphasizes social aspects of live coding and the statement, "live coding is inclusive and accessible to all," resonates with Nake's 1971 criticism of computer art. As with the 1950s visual computer art that later influenced work on education, one of the ways in which live coding benefited the society is through work on education.

The system that made the use of live coding in education possible is Sonic Pi created by Sam Aaron.[100] It was released in 2012 and was initially built to teach programming and music in schools using the low-cost Raspberry Pi computer. When using Sonic Pi, performers write code in the Ruby programming language to instruct a synthesizer to play notes and samples. They can select aind immediately run portions of the code to play their music. Almost 50 years after the appearance of LOGO, the Sonic Pi system shares a number of important principles that defined LOGO. First, Sonic Pi is designed for both schoolchildren and professional musicians, which mirrors the aim of LOGO to bring the "most powerful programming language available", LISP, to everyone. Second, Sonic Pi and live coding more generally adopt a liberal approach to errors. In "musical genres that are not notated so closely (...), there are no wrong notes - only notes that are more or less appropriate to the performance." This means that live coders "may well prefer to accept the results of an imperfect execution"; they might "even accept the result as a serendipitous alternative to the original note."[101] As in LOGO, errors are seen as sources of serendipity that provide an opportunity for learning.

Live coding is clearly inspried by earlier work on computer art and so it orginates in the humanistic culture of programming. The idea has been, however, influenced by other cultures of programming. The hacker culture influence was prevalent in the early computer music work where performers built their own instruments, following the do-it-yourself mantra of the 1960s counterculture. The extent to which live coding is influenced by mathematical culture is open to a debate. On the one hand, the importance of algorithms, a cornerstone of the mathematical culture, is made clear in the TOPLAP manifesto which "recognises continuums of interaction and profundity, but prefers insight into algorithms." On the other hand, live codes are performers who use the computer as a music instrument rather than to construct linguistic entities. A typical performance using a system like Sonic Pi involves programming concepts like loops and processes, but typically few, if any, algorithms.

In this, programming education using live coding dissents from systems that teach concepts based on the academically dominant mathematical culture. For example, the visual programming language for education Scratch, which was developed at MIT Media Lab in the early 1990s takes a different approach. Scratch was inspired by LOGO and evolved from the idea of "visual LOGO". It allows children to create graphical programs like games, but it fully adopts the idea of programming as the construction of linguistic entities. Programs in Scratch are constructed visually by arranging blocks, which makes it more accessible to children, but the thinking that it encourages is more directly product of the mathematical culture. The difference between Sonic Pi and Scratch is a clear example of the importance of recognizing different cultures of programming. The notion of computational literacy very much depends on which culture of programming is advocating for it.

Reinventions and Adaptations

Looking at the history, we can broadly identify two ways of thinking about programming. The first one is to treat it as a process of constructing textual programs, which are created, modified, formally analyzed and then compiled and run. The second is to see programming as interacting with a programmable medium. The two ways of thinking do not result in completely different technologies. A mathematical programming language like Algol also exists as part of an environment comprising editors, tools and compilers that the programmer interacts with, while an interactive programming systems like Smalltalk or Interlisp still have a language at their centre. The two ways of thinking, however, determine what the programmers focus on when creating or using the technology.

The two perspectives are noteworthy for my story because each of them is favored by different cultures of programming. On the one hand, the mathematical culture largely exists around the idea of treating programs as textual entitites, written in a formal language of mathematics. On the other hand, the hacker culture and the humanistic culture often rely on treating programming as interaction with a machine or a medium, respectively. To hackers, interactive way of programming offers direct access to the computer and it lets them fully utilize the machine in ingenious, unrestricted ways. To artists, visionaries and educators, interactive way of programming is a more natural view that emphasizes the crucial ongoing process, be it exploration, learning or communication with a machine.

When discussing the mathematization of programming in Chapter 2, it was possible to see direct historical influences and follow the development of the idea and concepts that emerged from it. I did not try to document every single influence and concept, but it

was possible to see a continuity, both in the community and in the ideas that it developed. One reason for this is that the hotbed of the mathematical view was in academia, which publishes work as papers and books that can be easily shared and followed. Another reason is that the mathematical treatment of programs is a very general idea, not linked to a particular machine or application. This means that it can continually develop even when machines, systems and programming languages change.

The history of interactive programming does not follow such clear historical line. Work on interactive programming often results in code, hacks, memos and demos rather than publications with references, which complicates the tracing of direct influences. The progression from MIT hackers to the ARPA community and Xerox PARC is an exception from the rule as there is a well-documented continuity in the community. The same cannot be said about the emergence of the new generation of microcomputer hackers, who share many values with the early MIT hackers, but do not form a part of the same community. Consequently, many ideas of interactive programming reappear, likely independently, in different contexts. Their presentation may be quite different, but the core ideas have surprisingly much in common. For example, viewing a computer as a programmable dynamic medium, which appeared in the humanistic vision around Smalltalk is not a million miles away from the interactive programming of microcomputers using BASIC, which served, at least for some time, as a primary medium for not just programming, but also interacting with the machine.

The interactive approach to programming often reappears and evolves when the context in which programming is done changes. Many of the developments that I looked at in this chapter were caused by new hardware or system developments. TX-0 gave the MIT hackers a direct access to the machine, which led to the development of new online debugging tools like UT3 while time-sharing provided a new kind of direct interaction and motivated the development of LISP editor, which made program modification on a teletype more manageable. Last but not least, the rise of personal computers led to the development of new interactive programming systems, both graphical and text based. However, new contexts that inspire ideas on interactive programming can also be new application domains. Two examples that I looked at above were programming in the educational context and programming as a tool for computer music, but there are numerous examples that also bring interactive programming into contexts that are of interest to other cultures of programming.

In the mathematical culture, interactive style of programming appeared in the context of interactive theorem proving. Here, the programmer first defines a theorem they want to prove and then enter commands that transform the goal. After each command, the programmer can review the current state of the system such as remaining sub-goals. Interactive theorem provers can be used for proving mathematical theorems, but thanks to an equivalence between types and propositions that I return to in Chapter 5, interactive programming using a theorem prover can also be used to construct programs.

Last but not least, new impetus for the development of interactive programming has also been provided by the field of data analytics and data science. In this context, various interactive programming systems have been developed since 1988 when Mathematica 1.0 was released for the Macintosh. Much of the recent work in this context is around notebook systems that interleave text and equations with code and outputs of running code such as charts and tables. Notebooks are used interactively; the analyst writes code in small chunks, runs it interactively and immediately sees the results. In a somewhat lim-

ited way, modern data science systems reinvent the naive realism concept of Boxer, that is the idea that what users see on the screen is their whole computational world.

What is striking about the many reappearances of interactive programming in different contexts is that the different incarnations of the interactive style of programming often follow a similar historical pattern. They appear in the creative, humanistic or hacker culture. They are used by hackers to explore new ways of using a computer, visionaries to imagine new ways of thinking using computers and artists to pursue creative goals. Along the way, they produce various ancillary ideas and programming techniques that are interesting on their own and that appeal to more practically minded engineers and managers. To satisfy engineering and commercial interests, the ancillary ideas are often stripped away from their original context and they develop independently. The examples of this pattern are ubiquitous in this chapter. Xerox Star adopted user interface of Smalltalk and icons from Pygmalion, but removed the underlying programmable medium, while microcomputers stopped booting into an interactive BASIC console and started running spreadsheet and word processing applications instead.

In other words, each reappearance of the idea of interactive programming enriched the state of the art of programming with new concepts, ranging from debugging tools and structure editors to graphical interfaces and spreadsheets. However, behind those interesting new concepts often lie more fundamental visions and ideas such as those of man-human symbiosis, augmenting human intellect or programmable metamedium. Those provide inspiration or conceptual framework for the specific technical developments, but those fundamental ideas are often lost as specific new technical concepts become established and find use in everyday commercial programming.

So far, we looked at developments that pursue somewhat idealistic visions for programming. However, both the mathematical perspective on programming and the humanistic and hacker ideals of interactive programming lead to a certain reservation towards practical programming as done in industry. In the next chapter, I follow developments that were more directly concerned with practical software developments. They made programming more accessible and reliable, but it happened through other means than those advocated by the mathematical, hacker and humanistic cultures.

Notes

1. The term “hacker” here is used to refer to a programmer-hacker, a sub-culture that is a part of the computing folklore documented by Levy (2010) and Tozzi (2017), rather than to security-hackers. The two cannot, however, be fully separated as programmer-hackers were open to violating security for (what they saw) as non-malicious goals.
2. Programming of Whirlwind was documented by Fedorkow (2021), as part of an effort to recover some of the original Whirlwind software artifacts.
3. Halvorson (2020) notes that rising accessibility of computers led to questions such as whether computational thinking may be valuable for its own sake, which inspired the development of LOGO.
4. For the connections with counterculture, see Markoff (2005) and Turner (2010). The story of the IPTO community, which has gained almost mythical status among some programmers, has been told by Waldrop (2001) and Hiltzik et al. (1999).
5. Most notably, the PILOT system developed by Teitelman (1966)
6. A phrase used as the title of a speculative essay by Kay (1972); the motivation has also been described by Kay (1996) in retrospective article on the history of Smalltalk.
7. The example is based on that given in Kay and Goldberg (1977)
8. The term “biggerism” was used by Charles “Chuck” P. Thacker who worked at Xerox at the time and used the term to describe unnecessarily complicated technology (Hiltzik et al. (1999))

9. Montfort et al. (2014) documents some of the early history and illustrates the creative spirit of the community around microcomputers through reflections on a single BASIC program `10 PRINT CHR$(205.5+RND(1)); : GOTO 10`
10. Halvorson (2020) tells the story of microcomputers, BASIC and how those contributed to the movements to democratizate programming.
11. Dijkstra (1982)
12. For this view on interactive programming, see Blackwell and Collins (2005).
13. `https://twitter.com/rbkc/status/1143904409019199489`, retrieved 18 May 2022
14. Kotok (2005)
15. As recollected by Greenblatt (2005); Russell (2017); Samson (2017); Kotok (2005)
16. Campbell-Kelly (2004) discusses the broader context of home and recreational software that followed early games like Spacewar!
17. The hacker ethic and culture has been described by Levy (2010). This is a canonical, but a naively optimistic reference. Kelty (2008) describes the stories of early hackers as avowedly Edenic and not reflecting inevitable commercial constraints and Tozzi (2017) also provides a more critical perspective. As pointed out by Ensmenger (2015), the idealised view of Levy (2010) and Brand and Crandall (1988) also helped to establish the masculine "stereotype of the bearded, besandaled computer programmer" at the time when "in actual practice women were still very much present in most corporate computer departments."
18. Forrester and Everett (1990); broader history of Whirlwind has been told by Redmond and Smith (2000) and Ornstein (2002) provides a personal account of Whirlwind programming.
19. The SAGE system is at the start of a long history of defence software, followed by Slayton (2013), that shaped the public debate about nuclear defence. The complexity of SAGE is illustrated by Ensmenger (2012) who points out that in 1956, the company developing software for SAGE employed three-fifths of all programmers in the US and went on to double the number of programmers in the country over the next few years.
20. McKenzie (1974)
21. **?**Jr. (1958)
22. The "dynamic flow chart program" is described in "The Computer Museum Report", Volume 8, 1984 available at `http://www.bitsavers.org/pdf/mit/tx-0/TX-0_history_1984.txt`, retrieved 21 May 2022
23. Samson (2017)
24. Beeler et al. (1972)
25. The technical details of Logic Theorist have been described by Gugerty (2006). MacKenzie (2001) discusses broader history of proof automation and identifies two strands of work. Logic Theorist aimed to simulate how humans reason, while other systems aimed to use the most efficient computer solution.
26. LISP is often described as "the first functional programming language", but as pointed out by Priestley (2017), who documents the origins of LISP, this is misleading and simplistic. Personal recollections by McCarthy (1978) also refrain from making such claims.
27. Levy (2010)
28. Tan (2020) considers work of Ada Lovelace through the framework of poetics and uses this as a starting point for arguing for a more profound understanding of code.
29. The various influences of the two essays are documented by Waldrop (2001) who also, in many ways, illustrates the techno-optimism of the humanistic culture of programming.
30. Bush (1945)
31. Licklider (1960)
32. Waldrop (2001)
33. Slayton (2013) discusses the work of Licklider in the context of military command and control. Despite actively contributing to this aspect of military research, Licklider was skeptical of automatic anti-ballistic missile systems that excluded human interaction.
34. Licklider (1965)
35. Sutherland (1966)
36. Ellis et al. (1969)
37. Visual programming languages are still being developed today and are frequently aligned with the humanistic culture, such as its focus on education. An example is the block-based visual language Scratch developed by Resnick et al. (2009).
38. Sutherland (1966)
39. November (2004)

40. McCarthy (1992)
41. Quoted in Waldrop (2001)
42. Quoted in Waldrop (2001)
43. Documented in Waldrop (2001)
44. Fox (1960); McCarthy (1978)
45. Deutsch (1967)
46. The report,Deutsch (1967), does not seem to be cited by any of the later work on structure editors. However, Deutsch published a paper “An online editor” Deutsch and Lampson (1967) about a general-purpose text editor that resembles the LISP editor and is more widely known.
47. Teitelman (1966)
48. Gabriel (2012), who reflects on the history of the schism, suggests to view it as two scientific paradigms. I would rather associate the “systems” and “languages” views with the hacker and the mathematical cultures of programming, respectively.
49. This is documented in the Interlisp manual, Teitelman (1974), and in a paper on the history of Lisp Steele and Gabriel (1996)
50. Bawden et al. (1974)
51. This is quoted in the humorous hacker’s “Jargon file” that was maintained by the hacker community in various formats at MIT and Stanford and was later published, e.g., in Raymond (1996)
52. According to Steele and Gabriel (1996)
53. Noll (2016)
54. Nake (1971)
55. Solomon et al. (2020)
56. Solomon et al. (2020)
57. Papert (1980)
58. Papert (1980)
59. This example is given by Solomon et al. (2020)
60. Solomon et al. (2020)
61. Many of those are documented, for example, by Ensmenger (2012); Abbate (2012); Hicks (2017).
62. Parts of the story of Doug Engelbart has been told by many, including Rheingold (2000); Hiltzik et al. (1999); Waldrop (2001) and perhaps most comprehensively by Bardini (2000).
63. Available to watch online Engelbart and Victor (1968)
64. Engelbart (1962)
65. Hiltzik et al. (1999)
66. Quoted in Hiltzik et al. (1999)
67. Barber (1975)
68. Waldrop (2001)
69. For the former, see Waldrop (2001); Hiltzik et al. (1999); for the latter, see Markoff (2005); Turner (2010)
70. Markoff (2005)
71. Hiltzik et al. (1999)
72. Kay and Goldberg (1977)
73. Beeler et al. (1972)
74. Documented by Ensmenger (2012)
75. This envisioned progression was made more explicit later, in Reenskaug (1981)
76. Some of the motivations are discussed in the internal “Why Alto” note by Lampson (1972)
77. Kay and Goldberg (1977)
78. Hiltzik et al. (1999)
79. Goldberg and Kay (1976)
80. Smith (1977)
81. Di Sessa (1985); diSessa and Abelson (1986)
82. Gombrich (1961)
83. A term introduced by Lennon (2019).
84. The design has been documented by Johnson et al. (1989)
85. According to an interpretation by Hiltzik et al. (1999)
86. Damouth et al. (1971)
87. This history is a part of broader context of the development of a personal computer, documented by Ceruzzi (2003).

88. Interactive programming is a technological link between Engelbart's NLS, Kay's Smalltalk and microcomputers. Their role in the countercultural movement provides another link, documented by Turner (2010)
89. From 1975 issue of the People's Computer Company, quoted by Turner (2010), p.115
90. The development of BASIC and the now-legendary letter are prominently discussed by Freiberger and Swaine (1984); Ceruzzi (2003); Montfort et al. (2014)
91. Known as EWD498 and later published as Dijkstra (1982)
92. For a more detailed account of the birth of the personal computing industry, see again Ceruzzi (2003)
93. The history of VisiCalc has been told by Grad (2007)
94. Atkinson (2016)
95. Chris Brown and John Bischoff (2002)
96. Chris Brown and John Bischoff (2002)
97. The history of The Hub has been documented by Gresham-Lancaster (1998)
98. The parallel has been pointed out by Zmölnig and Eckel (2007)
99. Ward et al. (2004)
100. Aaron (2016)
101. Blackwell and Collins (2005)

Chapter 4

Software Engineering

Teacher: The last time, we talked about the many ways in which mathematical methods can be used to make software development more rigorous. Looking at industry practices, it seems that those methods are not used nearly as often as *Tau* would like. Yet, computers mostly work. Out of the 1100 or so deaths caused by computers before 1992, only thirty or so were due to a software bug and out of these, twenty-eight were caused by a faulty software in the Patriot missile system. How is it possible that the number is this low, despite the very limited use of formal methods in practice?[1]

Tau: Practitioners deserve a cautious congratulation. Looking at the industry practices, I think the answer is a mix including debugging, testing and quality assurance, rigorous management processes and development methodologies, as well as over-engineering.[2]

Gamma: What I find remarkable is a fact that you did not mention. It is that 90 percent of the 1100 or so deaths were caused by poor human-computer interaction. Does it really make sense to spend so much time on formal methods when better understanding of human factors could have an order of magnitude more possitive impact?

Tau: Formal methods play a direct role in practice, especially in mission-critical systems where a failure could be catastrophical. More importantly though, they deepen our basic understanding of programming, promote the best of current practice and point to directions for future improvement.[3]

Teacher: Let's have a look at *Tau*'s list in a historical order. What were the first industry practices that helped us produce more reliable systems?

Alpha: Well, debugging programs got a lot easier thanks to the birth of time-sharing and interactive computing at the end of the 1960s and in the 1970s, because you could interact with the computer while it was running, explore the state of the program, correct it and resume the execution. A primitive form of debugging goes back to the first digital computers that often had a way of running programs step-by-step. Some even had lights where you could see the current state of registers!

Gamma: Ironically, debugging got worse with digital computers! A light to show you a value in your registers is nice, but nowhere near what we had before.

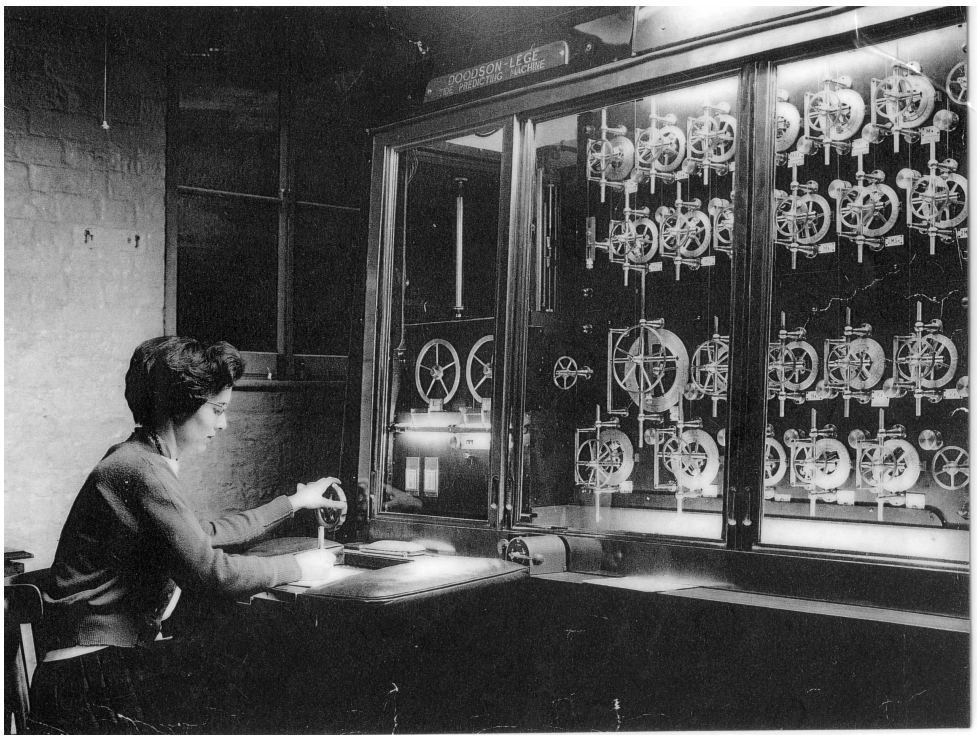


Figure 4.1: Valerie Gane in the basement of Bidston Observatory, operating the Doodson-Légé tide prediction machine, which was built circa 1950.

Epsilon: What do you mean? Debugging calculations made by hand on a piece of paper?

Gamma: It's easy to forget, isn't it? Before digital computers started to dominate in 1950s, there was quite a number of analog machines. They were less powerful and inflexible, but they didn't just calculate an answer; they invited you to make a tangible model of the world. Your program becomes a living thing that you could inspect as a whole![4]

Tau: I admit that some of the analog computers are works of beauty. You can certainly see how the calculation works when using the analog Doodson-Légé tide prediction machine (Figure 4.1), much better than whan using a model implemented in FORTRAN.

Alpha: You are right that mechanical analog computers were transparent thanks to their nature, but similar debugging capabilities existed for early digital computers. On the EDSAC machine, you could debug programs by running them instruction-by-instruction and inspecting the state of a part of memory on a CRT screen…

Omega: Such a waste of valuable computer time! It's not surprising that this use of digital computers was soon banned on most installations.

Epsilon: In the 1940s, computer time was understandably more expensive than programmer time. As is often the case with imposed limitations, it inspired programmers to come up with new approaches. It resulted in the first debugging tools, many of which we use to this day. In case of EDSAC, those were postmortem dumps and interpreters that printed diagnostic information while running the program.

Teacher: That sums up the state of debugging in the 1940s and the 1950s, but let's get back to the development of time-sharing and interactive computing that *Alpha* mentioned earlier. Did debugging became easier in the 1960s when computers could be used interactively by an individual.

Tau: I was wondering about this, but it is really hard to find any information about the early debugging techniques in the academic literature!

Alpha: You can check out the FLIT and DDT utilities, which were built in 1960 and 1961 for TX-0 and PDP-1 machines. They are documented well enough in memos published by the MIT Lincoln Laboratory. An academic paper would probably not be very useful. They are simple tools and hackers learned how to use them through practice...

Omega: I see! This is the black art of programming again!

Epsilon: That is a fair criticism, but people started taking the issue of debugging more seriously in the 1960s. Once hardware got reliable and high-level programming languages made it possible to write useful programs, the fact that the number of bugs in programs was not getting any smaller became the main limiting factor for the use of computers.[5]

Tau: Very well, but what new debugging capabilities did this give us?

Alpha: I'm not so sure. The debugging tools that I was talking about were quite powerful already. You could step through instructions one-by-one, set breakpoints and even create conditional breakpoints. You could also patch your program on the fly to see if your fix resolves the issue. This worked initially for assembly languages, but in the mid-1960s, similar tools started to appear for LISP and, later, other high-level languages.

Tau: Should I be impressed or disappointed? Honestly, this sounds pretty much like the modern debugging tools that I use today!

Epsilon: Well, modern debugging tools are better in a number of ways. They use efficient incremental compilation rather than interpretation and reverse or time travel debugging makes it possible to step back in time. But the primary improvement is greater conceptual clarity. In the 1960s, debugging became clearly distinct from testing, which allowed debugging and testing tools to evolve independently.

Teacher: This is a good segue to testing, which was the next technique mentioned by *Tau*. There is a lot to be said about testing in the context of modern software development, but let's start with the history. What did software testing mean in the 1960s and the 1970s, when it became separated from debugging?

Omega: Proper testing only came when programming started becoming "software engineering" and replaced the black art of programming that *Alpha* keeps talking about with a more rigorous methodology following the 1968 NATO conference. Testing became a part of the software development process. It was used to make sure the program solves the specified problem.

Tau: Excuse me, but testing shows the presence of bugs, never their absence. In other words, you cannot use testing to show that your program operates correctly.[6]

Omega: You are missing the point. The role played by testing in the development processes used in the 1970s was to certify that a system has been implemented according to requirements an can be handed over to the customer. How comprehensive testing is required was a matter to be agreed with the customer.

Epsilon: Today, we think of testing as being primarily about finding bugs, but *Omega* is correct that, historically, testing has been thought of as a way of showing that programs satisfy the requirements. It was a productive view for some time. For example, it encouraged the development of automatic testing tools.

Tau: This is a silly idea. If you guarantee program correctness just through tests, it is surely easy to write a program that passes all the tests, but is completely wrong!

Omega: That is why the testing phase of the development process needs to be undertaken by a different team than the development phase. That way, the developers do not know what the tests look like.

Epsilon: Do not be ridiculous. Programmers are professionals who want to do a good job. We understand that our goal is to produce effective software, not software that merely passes tests. The secretive way of testing used in the 1970s not only make development expensive, but also create an unhealthy relationship between testers and programmers.[7] Well-designed tests used in an earnest way give you satisfactory practical guarantees.

Tau: Well, thorough thinking about the meaning of *well-designed* inspired some interesting analyses at the time. Two influential papers from 1975 treat the issue of testing using mathematical methods. One tries to find a good way of generating input data for tests, while the other develops a probabilistic theory based on how much of code gets covered by tests.[8] Mind you, this still gives you weaker guarantees than formal verification, but it is, at least, a rigorous method.

Teacher: We are having a much more interesting debate about testing than we had about debugging. I wonder if we can explain why that is the case?

Gamma: It seems to me that a "test" is an entity that can mean something to each of us. You can find a meaning for "test" even in my favorite domain of live coding, but it is more like a practice in an art performance. Anyway, the consequence is that each of us can work on tests and multiple cultures contributed to the concept.

Teacher: Does this mean that the history of testing is richer than that of debugging?

Epsilon: Certainly! And we did not even get to the most interesting developments that followed once people stopped following the impractical heavyweight managerial development methods that treated "testing" as a separate stage after "development" and, instead, started using tests as part of Test-Driven Development…

Tau: Excuse me, but isn't Test-Driven Development still using tests just as a poor substitute for formal verification?

Epsilon: No, no, no! In modern use, testing is doing much more than just writing checks to see if you implemented a certain part of a specification correctly. In TDD, you create tests through a conversation with a customer, so tests capture the requirements and become a lightweight specification. You can then use tests to get rapid feedback during development. It also structures your work so that it follows the scientific method. You run a test (i.e., run an experiment) and if it fails, you revise your theory (i.e., your program), until the test passes.[9]

Tau: I understand what you say about structuring development workflow and getting a quick feedback, but using tests to create your specification is the wrong way round. Surely, you should write tests based on your specification so that you are checking for sensible properties.

Epsilon: The problem is that, in practice, you almost never have a detailed specification of the system. The idea of Test-Driven Development is that you first write a test and only then implement the code. When writing the tests, you have to think about what the right behaviour should be. The tests should be as simple as possible, so that your colleagues or even business domain experts can review them and make sure that your specification is correct.[10]

Gamma: This is an interesting remark. If tests are something that you discuss with your colleagues and look at in order to understand how a software should work, then they provide a focal point for valuable social processes. Exactly the kind of social processes that proofs of program correctness were lacking!

Epsilon: Another useful property is that tests do not get out of date. You run them automatically after every change and make sure that all tests pass. If your tests fail, you either fix your code, or revise your specification and update the test.

Omega: When you put it in this way, the idea that engineers not only see, but even produce the tests makes sense. What you are describing is an appealing light-weight software development methodology. If you have enough time and resources, then I think you should still produce a detailed specification up-front. However, if your timeframe and resources are limited, then your method where specification is developed gradually and written in form of tests sounds quite effective.

Alpha: This debate has taken an unexpected turn! When I think of testing, I just think of writing a simple script to explore an idea, or running a command interactively. There may be some tools to help with that, but I do not think it is useful to turn that into a specialization with a three-letter acronym and expensive certifications!

Teacher: We seem to be getting ahead of ourselves. We started with testing as an engineering practice, but are now talking about testing as a development methodology. This was, alongside with rigorous management, another factor suggested by *Tau* earlier. Let's take a step back and focus on these two aspects. What were the first development methodologies that people used?

Epsilon: I guess the answer is the Waterfall methodology that was described by Winston Royce in 1970.[11] It is a heavyweight process consisting of requirements gathering, analysis, program design, coding, testing and operations. To be honest, I do not understand how anyone was able to build software this way.

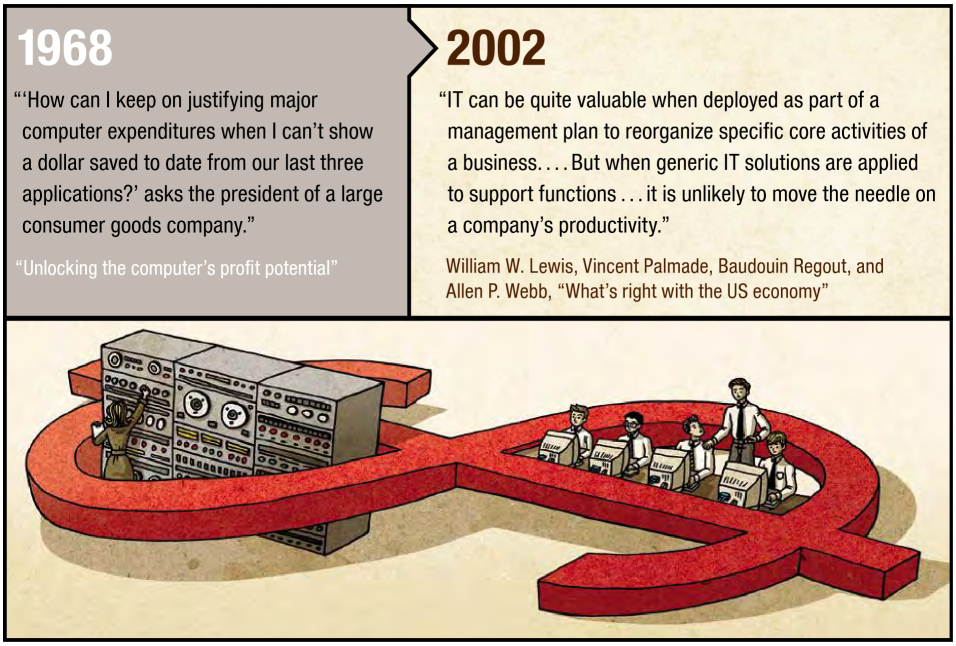


Figure 4.2: A quote from the McKinsey "Unlocking the computer's profit potential" report, reprinted with a 2002 quote under a title "Some things haven't changed much..." in the McKinsey Quarterly 50th anniversary issue in 2014.

Teacher: Can you elaborate on what was wrong with the model?

Epsilon: The issue is that Waterfall directs you to only progress forward, step-by-step. It assumes that you never need to revisit your earlier decisions, but in practice, this inevitably happens.

Omega: First, the idea is older than you think. The process later known as Waterfall was used in the 1950s for large-scale military software where careful requirements analysis and design were a must. It worked very well in that context. Second, the often cited paper by Royce is actually about adjusting Waterfall to allow going back if a decision needs to be revised. People were already refining the method at the end of the 1960s.

Tau: So, was that write-up produced in response to the 1960s NATO conferences on Software Engineering that we keep encountering?

Omega: I do not think so. Everyone says that the conference hosted a diverse audience, but almost no development managers were among the participants. The organizers of the subsequent 1969 conference explicitly decided avoid management issues! A better wake-up call for many in the management roles was the "Unlocking the Computer's Profit Potential" (Figure 4.2) report published by McKinsey.[12]

Alpha: I've heard of Waterfall and Test-Driven Development, but I really cannot think of any other methodology. Were there any major developments in between these two?

Omega: Many development methodologies improved the Waterfall model. This included adding a rapid prototyping phase, multiple iterations of the process or more thorough validation methods.

Alpha: Everyone keeps frowning when I say that getting better at programming just requires more time and experience, yet, this seems to be exactly what happened with the development methodologies!

Omega: A more significant development happened on another level. Pepole started to realize that different methodologies may work for different organizations, but what matters is to have a methodology. Models like the Capability Maturity Model (CMM), which was developed in 1986, define the important characteristics of a methodology. It allows an organization to gradually progress from an initial chaotic model to repeatable, defined, capable and efficient methodology.

Epsilon: That sounds even more heavy-weight than Waterfall. Not only you need a process, but you need a process for having a process!

Omega: I guess it worked well enough for some software systems in the 1970s and 1980s, but in the 1990s, the PC market and the rise of the internet changed what kind of applications companies needed. It was crucial to build software faster, more collaboratively and adapt to changing market requirements.

Alpha: Is this where the Agile methodologies come from?

Epsilon: It was a gradual process. Extreme Programming and Scrum were two methodologies that tried to address the situation, but you are right that the term Agile is the most famous one. The Agile manifesto, written by 17 thought-leaders at the famous Snowbird meeting in Utah in 2001 gave those methodologies a common name and popularized them.

Gamma: It is sad that it takes 17 men to re-invent something that has been described more clearly by a woman over a decade earlier!

Epsilon: Oh, I did not know this. Please tell us more.

Gamma: Well, Christianne Floyd, who was a professor in Berlin from 1978 was working on software engineering methods inspired by the Scandinavian idea of "participatory design" where you closely work with users of a product during its design. She wrote a paper "Outline of a Paradigm Change in Software Engineering"[13] that contrasts the, dominant at the time, product-oriented approach with the emerging process-oriented approach. The latter is based on close collaboration with customer, learning and building of a shared understanding.

Epsilon: That does, indeed, sound almost the same as the key ideas in the Agile movement!

Teacher: On that note, let's conclude our discussion about development methodologies and move to the next topic. The next item from *Tau*'s list was over-engineering.

Epsilon: Technically speaking, that is what I would call exception handling. It also has an interesting history and involves a couple of interesting programming langauges that we have not discussed yet, including PL/I and, a bit later, Clu and Ada.

Alpha: That is looking at the history through contemporary lens! LISP had exceptions well before that, but they were often handled interactively by the programmer, so the concept of exception handling overlaps with debugging that we talked about earlier.

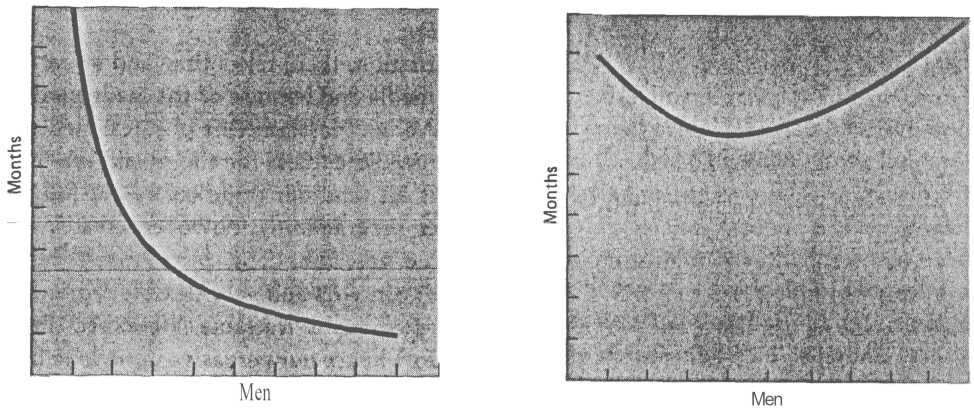


Figure 4.3: Time versus the number of workers for a “perfectly partitionable task” (left) and a “task with complex inter-relationships” (right)[15]

Tau: It is clear we could talk about the history of exceptions, but I think we are missing the forest for the trees. Is there some fundamental idea behind all the different engineering methods? I’m sure we can use all these methods and perhaps build software a bit more reliably, but it is lacking an analytical framework.

Epsilon: What do you mean by an analytical framework?

Tau: Well, if you treat programs as mathematical entities, you can not only construct programs, but also say what programs can and cannot, in principle, be created. You cannot do this if you just talk about specific engineering methods.

Teacher: Let’s not be rude. I would be curious to hear more about exception handling.

Epsilon: No, no. I think *Tau* is making a fair point and I would like to answer, but if you only accept deductive reasoning, this will not satisfy you. The engineering community has, in fact, found its way of talking about the limits of what can be built.

Omega: This may be tricky, but I suppose you can run some empirical studies to verify the effectiveness of different methodologies and tools. If you want a fair comparison, you will have a hard time setting up multiple teams with equal skills and getting enough resources to develop the same software multiple times, just so that you can compare results.

Epsilon: I was not going to suggest you should run a controlled study…

Tau: If you reject both formal arguments and rigorous empirical studies, then I do not know how else you can make a serious scientific argument.

Epsilon: Software development is inevitably a human task. You cannot hide the inherent complexity and make simplifying claims. However, you can identify common patterns, characterize different classes of programs and their general properties and also make rigorous analytical arguments grounded in the understanding of the relationship between the human world and software.[14]

Teacher: It may be useful to look at a concrete case. *Epsilon*, could you give an example of the kind of reasoning you have in mind?

Epsilon: Take the well-known example from "The Mythical Man-Month"[16] by Fred Brooks, that is often called Brooks' law. It states that "adding manpower to a late software project only makes it later."

Omega: In my experience, that is, indeed, the case, but I do not have enough data to prove this with certainty![17]

Epsilon: Brooks' argument does not rely on experience though. He explains the 'law' in terms of communication overhead. Look at the two charts in Figure 4.3. If you have a task that can be perfectly divided among team members, the amount of time needed decreases proportionally with the number of people. But programming complex systems requires communication between team members and, in general, you need each member to talk to every other member. So you have to add an overhead of $n*(n-1)$. This is quadratic and so, even for a small amount of communication, it eventually dominates the gains you get by dividing the work.

Omega: Well, you could measure the number of minutes needed for programming and communicating and give an exact formula, but I doubt it would match the reality very precisely.

Gamma: I do not think we need concrete numbers though. The logic is convincing enough without that. I guess this is what *Epsilon* referred to as rigorous analytical argument. I wonder to what other problems can this kind of reasoning apply...

Tau: It was used quite extensively in debates about the limits of software engineering, for example in the debates about anti-ballistic missile systems (ABM).[18] Another example was an argument that ABM cannot be reliably built because the rate of the change in the environment is greater than the rate of the change we can maintain in the software system. This also relies on a sort of mathematical argument that is convincing enough, even without specific quantities.

Teacher: I see, *Tau*, that you are still not convinced, even though *Epsilon* has given us two examples of mathematical arguments!

Tau: It may be the best you can do for dealing with the soft human aspects of software, but I am interested in software you can trust and I do not see how these kinds of arguments help with that.

Gamma: I thought we already agreed that each of us is interested in different kinds of questions, but it is a shame we did not even find a common language for talking about the human aspects of software.

Teacher: Let's conclude by getting back to the last point made by *Tau*, which is trusting software. This sounds like something very basic that each of us will have something to say about. How do you decide whether to trust software or not?

Tau: My answer will not surprise anyone. Trustworthy software should come with a formal specification and a correctness proof.

Omega: That is unrealistic and cannot work in practice. You need a solid process to ensure that system will have the chracteristics it needs.

Alpha: Ultimately, software is built by people. I only trust programs created by people that I can trust.

Epsilon: Well, I think we just talked about all the aspects that I find important. You need a combination of good people who take responsibility for their work, which cannot be replaced by a process, as well as good tools and methods of working, which cannot be replaced by personal individualistic wizardry.

Gamma: Do not forget, though, that software is written by people for people. I only trust software that is trying to do the right thing, because you can never separate the technical from the social and, honestly, the social-side of software seems to be much harder to get right these days.

Software Engineering

How Did Software Get So Reliable without Proof?

In March 1996, the International Symposium of Formal Methods Europe took place in Oxford. The symposium evolved from a series of events focused on the Vienna Development Method (VDM), a formal method that started with the work on semantics of programming languages in the IBM Vienna lab at the end of 1960s. The organizers stroke a positive note about the industrial applicability of formal methods and a quarter of the presentations at the symposium were reports on industrial use of formal methods.

The opening keynote of the symposium was presented by a veteran of the field, C. A. R. Hoare, who developed foundational theories for reasoning about programs that we encountered in Chapter 2. Hoare's keynote, "How Did Software Get So Reliable Without Proof?" asks a question that many proponents of the mathematical culture may be rightfully puzzled by. Hoare acknowledges that "formal methods and proof play a small direct role in large-scale programming" and reflected on this situation:

> *Twenty years ago it was reasonable to predict that the size and ambition of software products would be severely limited by the unreliability of their component programs. (...) Fortunately, the problem of program correctness has turned out to be far less serious than predicted. A recent analysis (...) has shown that of several thousand deaths so far reliably attributed to dependence on computers, only ten or so can be explained by errors in the software (...). Similarly predictions of collapse of software due to size have been falsified by continuous operation of real-time [telecommunications] systems. So [a question arises]: why have twenty years of pessimistic predictions been falsified?*[19]

Hoare's answer is that software gets reliable through a mix of practical engineering practices, including good project management, rigorous testing, debugging, over-engineering and innovations in programming methodology. As one might expect from a proponent of the mathematical culture, Hoare does not see this as a failure of formal methods. Instead, he argues that many such practices owe much to the theoretical ideas developed twenty years prior to his paper. Formal methods and proof may play a small direct role in large-scale programming, but "they do provide a conceptual framework and basic understanding to promote the best of current practice, and point directions for future improvement."

Despite the title, Hoare's keynote was an optimistic one. He concluded that the twenty year gap between theory in practice that he identified "is actually an extremely good sign of the maturity and good health of our discipline" where basic research conducted by theorists provides fundamental basic understanding and where industry adapts and adopts

useful theoretical innovations such as structured programming, informal mathematical reasoning, data types and inforamtion hiding.

Hoare's keynote paints an idealized image of a gap between theory, which explores new directions for advancing the state of the art, and practice, which adopts proven research ideas. In terms that I use to look at the history of the field in this book, Hoare would see the mathematical culture of programming as the primary source of new scientific ideas and the engineering and managerial cultures as those who apply scientific ideas in the real world. As we saw when looking at the origins of programming languages in Chapter 2, and as we will repeatedly see in the chapters to come, the interactions between cultures are not so simple. An idea may emerge in any culture of programming, it may be influenced and reshaped by ideas from other cultures and even return back in a different form. Much like the history of programming languages, the history of management, testing, debugging and error handling that we will follow in this chapter is full of such interesting and unexpected twists.

The Art of Electronic Computer Maintenance

The idea that programming a computer will become a major challenge was completely unexpected when the first digital electronic computers were created. To a contemporary programmer, this may seem naive, but it is less surprising if we recognize that digital electronic computers did not appear out of the blue in 1940s. In many ways, they were the next evolutionary step from analog computers which used mechanical, hydraulic or electrical quantities to model numerical problems. Analog computers did not require any programming and their operation was often fully transparent. A wonderful late example of this idea is the hydraulic computer that models the economy, which was created in 1949 and was called MONIAC, "to suggest money, the ENIAC and something maniacal."[20] In MONIAC, the flow of money in the economy is modelled as flow of colored water. Various mechanisms in the economy are modelled using valves controlled by floats, pulleys and other hydraulic devices illustrated in Figure 4.4. The machine had a limited accuracy, but it was unprecedented in aiding the understanding of complex ideas of economics.[21] The user of the MONIAC can always see the full state of the system and noticing that the machine (mis)behaves is just a matter of watching how the water flows through the system.

Electronic analog computers removed the transparency, but kept the simple fixed structure where a small change in the input leads only to a small change in the output. The last traces of this simplicity were finally lost with the advent of digital computers. The operation of digital computers is not just hidden from sight, but it also involves complex "evolution of a meaning".[22]

In spite of this complexity, the job of an operator who programmed the early digital computers was thought of as a low-status and not particularly challenging one. As told in Chapter 2, many of the original programmers of the ENIAC computer were women, recruited from a group of human computers hired to calculate ballistics tables by hand. In ENIAC the programming was, at first, done using switches and by connecting individual components of the machine using cables. The ENIAC programmers had to figure out how to translate a more high-level mathematical plan of a computation into actual wiring of the machine. At least initially, the program on ENIAC was the physical setup of cables and switches.

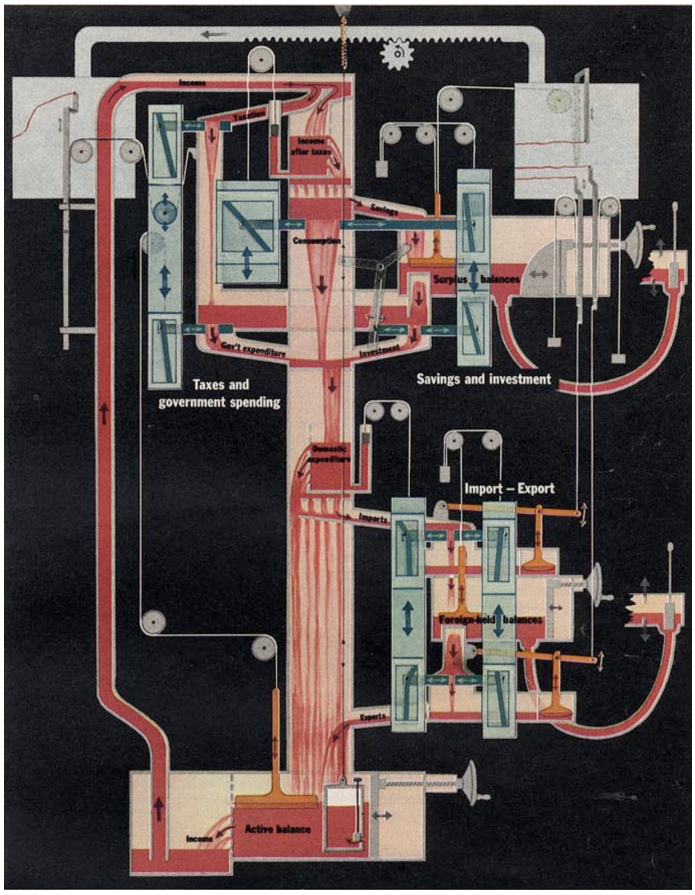


Figure 4.4: Illustration of the structure of the MONIAC computer by Max Gschwind for the article "The Moniac" in the Fortune Magazine, March 1952.[23]

Why start the history of good software engineering practices with something as rudimentary as the ENIAC? As soon as the challengees of programming and debugging appeared, programmers started to come up with ways of dealing with those. Many such methods are not unlike the methods that programmers use to program and debug computers today. First of all, different individual programmers approached the problem of programming in different ways. In her memoir, Betty Jean Jennings (Figure 4.5) who we encountered as on of the first ENAIC programmers in Chapter 2, recalled how her approach to programming differed from that of her colleague Adele Goldstine:

> *Adele was an active type of programmer, trying things very quickly. I was more laid back and given to attempting to figure out things logically before doing anything.*[24]

Computer programmers never resolved this schism and never settled on one right way of programming. We find the same difference in approaches among the hacker and mathematical culture of programming today. On the one hand, mathematically-minded programmers still favor logical reasoning over experimentation. This approach is often taught at universities, in part because it can be more easily written down. On the other hand, hackers often solve problems by quickly iterating on a solution. Being an active type of programmer like Adele Goldstine today may require less formal preparation, but more intuition and experience. Skilled hackers do not try just any random solutions. They approach the solution gradually, in a well-chosen sequence of steps.

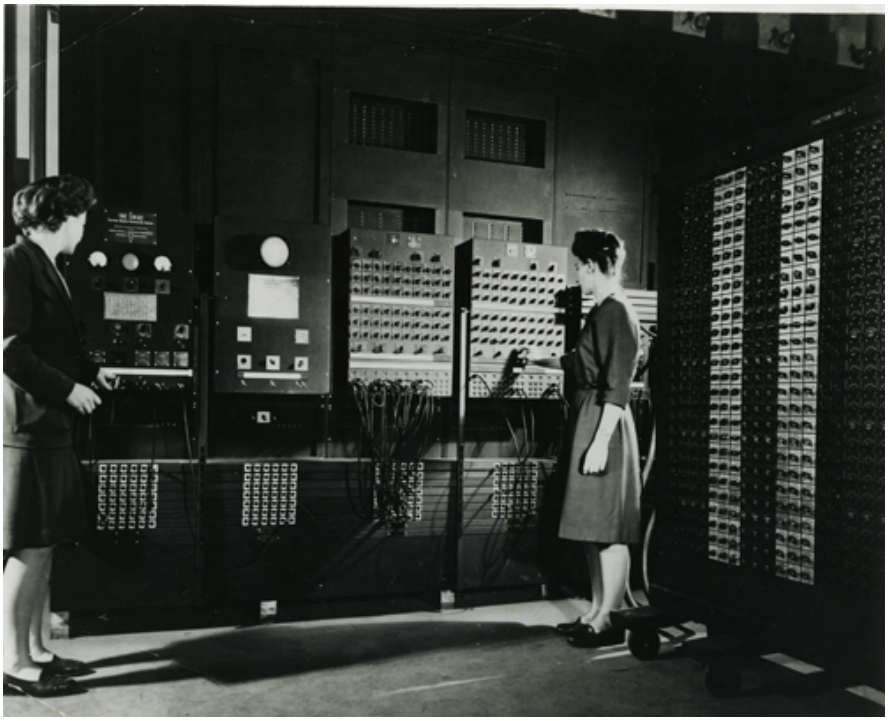


Figure 4.5: Betty Jennings, left, and Frances Bilas, right, setting up the ENIAC.

The programming of the ENIAC also involved two ideas that are characteristic of the engineering culture of programming, which often strives to find a better or a more reliable method to solve a problem. The first engineering method was used when debugging the program for calculating trajectories. When developing the program to calculate trajectories, two of the ENIAC programmers first manually calculated a single trajectory in a way in which the ENIAC was supposed to calculate it. When the machine was set up to calcualte trajectories, it was first run on this test trajectory. If the results differed, the programmers knew something was wrong. They could stop the execution of the ENIAC and run it manually pulse-by-pulse to find the source of the error. This was useful not just for finding errors in program wiring, but also to identify blown vacuum tubes, which was a common hardware failure of the ENIAC. The idea of using a hand-computed test run as a baseline for testing the program is the kind of robust method that arises from the engineering culture. In fact, the approach is not unlike that of a much later Test-Driven Development (TDD) methodology that we will discuss later in this chapter.

The second engineering technique that is illustrated by the history of ENIAC is the idea to use the computer itself to simplify its programming. In 1948, a reprogramming of the ENIAC changed how the machine operated. The new approach was based on the experience with the ENIAC and also ideas for a subsequent machine, EDVAC, that was in the works. The reprogramming wired the ENIAC such that it would execute instructions encoded on punched cards using so called "60 order code".[25] This made programming the machine easier as it no longer required tedious fiddling with cables. The new programmable wiring was kept until the end of ENIAC operation in 1955. Again, using a computer to build a tool that makes the programming easier is an engineering method that we will encounter many times in this chapter.

Despite being electronic and digital, the ENIAC had a certain degree of transparency. The state of the memory was visible using a grid of light indicators on the control panel, which the inventors used to put on a show during the 1946 public demonstration. Running the machine pulse-by-pulse also made it possible to follow what is going on. Even more curious hacker trick was used to make transparent the later BINAC machine, built by the Eckert–Mauchly Computer Corporation in 1949. One of the BINAC engineers discovered that they could connect the output of one of its circuites to a radio and listen to its sound, much like the hacker-minded narrator in "Zen and the Art of Motorcycle Maintenance"[26] who notices an issue just from the different noise the motorcycle engine makes:

> *Mind you, this weird music wasn't like anything being broadcast on the radio– it was just a sequence or pattern of sounds emitted from the radio as the BINAC was running. Al Auerbach said he'd quickly discovered that he could tell whether the BINAC was running correctly by the sounds coming out of the radio when he played the test program.*[27]

In the case of ENIAC, hardware failures were still very common and so debugging using, for example, the manually computed trajectory, often invovled looking for both hardware and programming errors. However, hardware soon started becoming more reliable, partly thanks to the fact that compures were made of a large number of repeated components. No such repetitive structure exist in computer programs and so programming issues grew in prominence. One of the computer pioneers who understood the complexity of programming early on was Maurice Wilkes, the designer of the EDSAC computer, who later recalled a moment in 1949 when "the realisation that a good part of the remainder of [his] life was going to be spent in finding errors in [his] own programs."[28]

The EDSAC computer, desgined by Wilkes at Cambridge shared architecture with many computers built after ENIAC. It was controlled through a program, represented as a sequence of *order codes* that was loaded from a tape or punched cards. The orders, or instructions, were then stored in computer memory and executed. EDSAC recognized 17 different instructions for performing calculations (`AnS` to add the number in storage location `n` into the accumulator), memory access (`TnS` to transfer the contents of the accumulator to storage location `n`), but also for conditional jump (`EnS` to continue executing orders in storage location `n` if the value in the accumulator is greater than or equal to zero).

The practical matter of programming early computers required "black art" skills that are characteristic of the hacker culture of programming. Programmers had to invent ingenious tricks, often using the fact that a program itself was stored in memory, to implement operations that seem obvious today, but were not included in the instruction sets of early machines. For example, EDSAC did not have an instruction for indirect addressing that programmers would use today when accessing an element in an array at a specified index. If you wanted to access a storage based on an index computed by your program, you had to use the transfer instruction `TnS` to first overwrite a part of your program (which was loaded into a known location in the memory) that specified the argument of the next `TnS` transfer instruction. The ACE computer designed by Alan Turing did not even have conditional jump and programmers had to use the very same trick of overwriting the target location of an unconditional jump.[29]

The first non-trivial program that has likely been subject to debugging is a program to calculate the Airy integral, created by Maurice Wilkes for EDSAC. A detailed account of the debugging process has been given by the historian Martin Campbell-Kelly,[30] who analysed

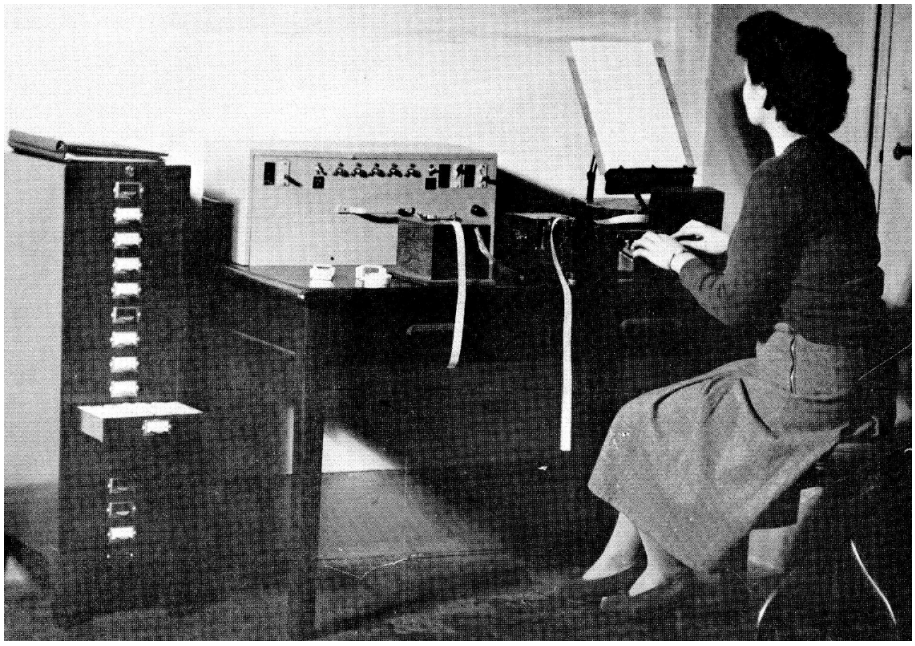


Figure 4.6: Tape preparation equipment with operator and a cabinet housing the EDSAC sub-routine library.

an early tape with the program, finding "approximately twenty errors in the 126 lines of the program." Many of those were simple punching errors, but some likely required notable debugging effort. The unexpected fact that producing correct programs is difficult meant that EDSAC did not initially have any dedicated program debugging tools. As elsewhere by Campbell-Kelly, "the way to debug a program at Cambridge, and at most other places, was to sit at the console and execute the program manually, instruction by instruction, while observing the registers and memory on monitor tubes".[31] This tedious process was referred to as "peeping". On EDSAC, it was done using a "Single E.P" button, which executed a single instruction of a program and a CRT monitor that displyed the contents of a portion of the computer memory. Some early computers went even further and provided switches for manually modifying the instructions in memory.

As was the case with the history of ENIAC programming, the history of EDSAC programming also includes a primordial form of engineering programming techniques that later become standard. On EDSAC, the most interesting methods were allowed by the fact that the program was stored in memory where it could be programmatically modified. This made it possible to use the computer itself to build tools that make programming and debugging the computer easier.

The first trick simplified the construction of "large" programs like the Airy integral calculation. Maurice Wilkes and David Wheeler adopted the idea of a sub-routine that was proposed in the "First Draft of a Report on EDVAC" written by the ENIAC team at Moore School. Sub-routines were sequences of instructions, eventually stored on tapes in a library (Figure 4.6) that implemented common tasks and could be added to the program after the main routine. Doing this manually would be laborious, because EDSAC instructions used absolute addressing in instructions that accessed memory and so all addresses in the sub-routine would have to be adjusted. To avoid this issue, Wilkes and Wheeler

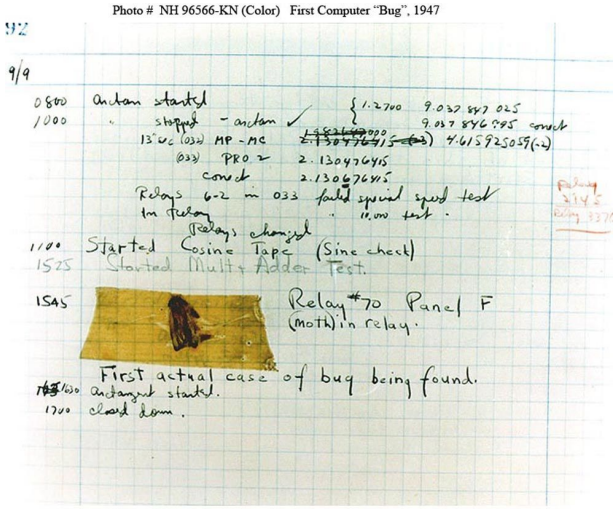


Figure 4.7: "First actual case of a bug being found" – a dead moth removed from the electromechanical Harvard Mark II computer by Grace Murray Hopper.

introduced "initial orders", which was a rudimentary program loader, formed by 40 or so instructions, that loaded sub-routines from a tape and replaced relative addresses in its instructions with absolute ones.[32]

A number of other tricks were devised to simplify the debugging of programs. Debugging programs through peeping, by running them instruction-by-instruction on a real machine was not only tedious but it was also taking valuable computer time. As the demands on the machine grew, peeping was recognised as "extravagant use of computer time"[33] and was outlawed by the designers of EDSAC. Fortunately, the engineering method of using the computer to simplify programming led to two new approaches to the problem of debugging in the new setting.[34]

The first attempt was the development of a "postmortem" dump routine that was triggered when the program terminated abnormally and printed out a region of the computer memory. The programmers could then inspect the state and look for errors offline. The routine is, of course, the precursor of the idea of a core dump or memory dump that is produced, for example, when a contemporary operating system crashes. The postmortem routine did not help when the program worked, but computed an incorrect result. Debugging such issues required a more sophisticated tool. This was implemented by Stanley Gill in 1951. His "interpretive checking routine" was able to print diagnostic information while the main program was running. To do this, it iterated over the instructions of the main program and executed them one-by-one, simulating the behaviour of the EDSAC. This was about 50x slower than running the instructions directly, but it provided an invaluable debugging tool. In modern terms, the routine implemented an EDSAC order code interpreter with detailed logging.

Before we continue, it is useful to review the terminology used to talk about programming issues in the 1950s. This has since evolved and so the historical meaning of the terms differs from the contemporary one. So far, I have been using the term 'debugging' to refer to the work that programmers had to do in order to get their programs to run correctly. In 1950s, this was one of the frequently used terms, although its meaning was broader and included activities that present-day programmers might call testing. The term debugging

has its origins in the engineering slang used at the time. Despite a popular tale, the term 'bug' was not introduced into the vocabulary of programmers when Grace Murray Hopper removed a dead moth from one of the circuits of the Harvard Mark II computer. Hopper has, in fact, removed a dead moth and kept it in the logbook (Figure 4.7), but the term 'bug' has been used by engineers to refer to malfunctions from at least late 19th century. Adoption of the term 'bug' for talking about programming issues was a notable rhetorical move. As pointed out by Kidwell,[35] by using 'bug' rather than 'failure', programmers suggest that the issues are small faults that can be corrected. This proved an ironic naming, given that the estimate cost of work done on fixing the "Y2K bug" was over $300bn.

Another term that was often used alongisde with 'debugging' was 'checking' a program or 'program checkout'. An early example that uses this terminology is a paper "Checking a Large Routine"[36] by Alan Turing. The paper illustrates that the early terminology around getting programs to run correctly was not clearly differentiated, although its use of the term 'checking' is the exception rather than the rule. Turing opens the paper by asking "How can one check a routine in the sense of making sure that it is right?" He suggests that programmers can help the program checker by "making assertions about various states the machine can reach". Program checking in Turing's paper may be read more as an early attempt at proving a program correct than as debugging or testing. In other early systems, "program checkout" was done, for example, by inspecting log traces produced by a tool such as the EDSAC interpretive checking routine. Throughout 1950s, the terminology differentiated and the typical view was that you wrote a program, debugged it to make sure it runs and then "checked it out" to make sure it produces the correct results. The task of 'program checkout' was more akin to what present-day programmers would call testing. Turing's early work on proving programs correct was not immediately followed and his formal approach to program checking was likely independently reinvented in the mathematical culture of programming in 1960s.

The shift from direct interaction with the machine to debugging tools was influenced by the business context. The machine time was expensive and postmortem and checking routines allowed programmers to debug their programs while the computer was available for running other programs. From the early 1950s, the operation of EDSAC was so streamlined that a full-time operator to run programs on behalf of the programmers was hired. The combination of managerial requirements and work of the engineering culture thus sidelined the more direct hacker approach. However the hacker culture did not have to wait long for another chance at transforming how we program and debug.

On-line Debugging Techniques

Running computer in a batch processing mode was the norm in 1950s and so debugging programs using memory dumps and diagnostic logs was the only option for most programmers. The exception from the norm were the MIT hackers who got their hands on the TX-0 computer, which we encountered in the previous chapter, after it was moved to the MIT Research Laboratory of Electronics (RLE) in 1958. The interactive input/output capabilities that were added soon after the move to MIT RLE made it possible to debug programs interactively, although the tools for doing so were initially very limited. The 1958 memo "TX-0 Direct Input Utility System"[37] first goes to great lengths to justify the very idea of "debugging at the console" and then acknowledges the current limitations:

> *Debugging at the console is not new. In fact, it is the oldest form of debugging. For the past few years it has been regarded as a great professional sin, but like most sins, it still exists, is rarely talked about, and hardly ever admitted. (...)*
>
> *At the present time most utility programs do not provide enough immediate information to enable a programmer to find his error while he is still at the console. And if he can find it, few systems are capable of allowing him to make an immediate change without the preparation of a card, paper tape, or some other secondary input medium, especially if he wants to retain the symbolic language of the conversion program.*

The Utility Tape 3 (UT-3) tool, which I mentioned in the previous chapter, allowed users to control the computer interactively, by typing commands on the console, but it was very rudimentary. Among other things, programs had to be typed in octal numerical system. The major advance in interactive debugging was the FLIT utility, written by Thomas Stockham and Jack Dennis. The latter was, in principle, in charge of the TX-0 operation at RLE, but was, in fact, more interested in playing with the machine himself. In a pun typical for the hacker culture of programming, the FLIT utility was named after an insecticide, but was presented as an acronym for Flexowriter Interrogation Tape.[38]

The FLIT utility made it possible to enter programs interactively using a symbolic notation that was very close to the assembly language used when writing the program in the first place. Rather than entering instructions as octal numbers, programmers could work with symbols as in their source programs. FLIT also introduced a debugging technique that remains the bread and butter of debugging to this day. The `break` command allowed programmers to set a breakpoint at a given location in the user program. When the program was run and reached the location, the control was transferred to FLIT, which printed the contents of registers and allowed the user to interactively modify the state and program instructions or resume program execution.

Only one TX-0 computer was ever constructed and so the FLIT utility never left the world of MIT hackers. A successor tool was created by another MIT hacker, Alan Kotok, for the DEC PDP-1 computer as soon as it was delivered to the MIT RLE in 1961. Following the insecticide-inspired naming tradition, this became known as the DEC Debugging Tape (DDT). The DDT utility was similar to FLIT and allowed the users to modify programs using a symbolic assembler, refer to parts of a program using the symbolic names used in the source code as well as set breakpoints. An interesting characteristic of both FLIT and DDT was their concise langauge of commands. In DDT, most operations were triggered using just a single character and so the tools were very efficient in the hands of experienced hackers.

The appearance of the first time-sharing systems in early 1960s renew interest in on-line program development and debugging tools, although much of the work was still happening close to MIT and Cambridge. One such collaboration was between Thomas Evans at Air Force Cambridge Research Laboratories and D Lucille Darley who worked at Bolt Beranek and Newman (BBN), a company with close links to MIT that we encountered in the previous chapter. The two wrote an extensive review of existing on-line debugging techniques and created a new tool named DEBUG.[39] The DEBUG tool addressed two issues with FLIT and DDT. First, it made it easier to modify program by automatically relocating parts of a program to make space for the new code. Previously, this had to be done by inserting the new code in an unused part of memory and inserting error-prone jumps to

the right places in the code. Second, DEBUG also made it possible to easily export a clean corrected version of the debugged program. It kept a table of edits performed manually during the interactive session and was able to print out on an "alter tape" after the end of the debugging session.

In their review of on-line debugging techniques, Evans and Darley remark that time-sharing systems make on-line debugging economically feasible and that facilities for program debugging is an "area of critical importance for effective utlization of such time-sharing systems". The motivation for their review was to document the systems that existed at the time, either in early time-sharing systems or for smaller computers such as TX-0 or PDP-1 that were used interactively at some installations. In doing so, the review also highlights one point that clearly positions work on debugging utilities in the realm of the hacker culture of programming. Evans and Darley noted that "much of the work in this field has been described only in unpublished reports or passed on through the oral tradition". Moreover, some of the tools they review "are far from completely described even in internal memoranda." In other words, debugging relies on personal knowledge gained through practice and mentorship that is difficult to write down. The reliance on oral tradition may also have been a reason why much of the early work on debugging tools was confined to MIT and affiliated organizations.

Tools like FLIT and DDT were a step up from UT-3 and flipping of switches on early machines in that they allowed the programmer to use a symbolic language during their debugging sessions. They no longer had to modify the numeric codes representing the instructions, but could instead type assembly language instructions. In 1966 when Evans and Darely published their survey, on-line debugging techniques existed for both assembly languages and for high-level programming languages including LISP and FORTRAN, although the ones for high-level langauges were "less well-developed and less widely used."

The programming systems with on-line debugging facilities for high-level programming languages started to develop in mid-1960s on early time-sharing systems. The time-shared IBM 7044 computer came with FORTRAN implementation QUICKTRAN and multi-user FORTRAN also existed for the Berkley time-sharing system. Those systems generally allowed users to set breakpoints (by inserting appropriate instruction into a program) and inspect the values of variables (using the symbolic name of the variable). QUICKTRAN was based on an interpreter and so it also allowed users to insert and delete statements in the program source code, which was not possible in systems that executed compiled code. The idea of an incremental compilation, which is used by some debugging tools today, already existed in literature, but was not implemented in a real-world system at the time.

The higher-level language with the most advanced interactive programming and on-line debugging features in mid-1960s was LISP. LISP has been around since the end of 1950s, but the first implementations were written for batch-processing systems. By 1964, LISP was available on a number of interactively used machines such as PDP-1 at Stanford, as well as on a time-sharing systems including the IBM 7094 computer running the Compatible Time Sharing System (CTSS) at MIT. The first time-sharing implementations of LISP featured fairly basic interactivity. The time-sharing LISP running at MIT allowed users to set breakpoints, evalaute LISP expressions interactively and print the backtrace (recording what functions are being called) when an error occured, but this was soon about to change.[40]

Thanks to its nature, LISP is the perfect language for the development of advanced debugging tools and LISP hackers soon started to exploit this characteristic. In LISP both data and code is represented as lists, a feature known as *homoiconicity*. This means that

code of a running program can be easily represented and modified during the program execution. As we saw in the previous chapter, a number of LISP implementations also included editors that allowed users to edit LISP code not as text, but by directly modifying the list structure representing it. As in the case of FORTRAN, running the code modified interactively required having an interpreter, in addition to a compiler that turned the initial program into an efficient assembly. The work came together in MacLisp, which ran on the Incompatible Timesharing System (ITS) at MIT. In the aforementioned survey of debugging tools, Evans and Darley describe a scenario that is made possible by this combination of debugging tools:

> *With some care, it has been possible, for example, to find a bug while at a breakpoint in running a test case, call the [structure] editor to make a correction, run the program on a simpler test case to verify the correctness of the change, then resume execution of the original test case from the breakpoint.*

One interesting aspect of this scenario is how it integrates the different activities involved in getting a program to run correctly. It starts with running a test case, which is an activity we would today refer to as testing, follows with making a correction in a tool that we would now refer to as the debugger and verifies this correction, which is something that a present-day programmer would perhaps do interactively using a REPL (Read-Eval-Print Loop) shell. In other words, the on-line debugging techniques of 1960s do not yet make a clear distintion between development, debugging and testing. More interestingly, error handling is also closely interconnected with the other activities.

A system that illustrates this well is the BBN LISP and the corresponding project manual published in 1967.[41] The authors include Daniel Bobrow, but also D. Lucille Darley (who co-authored the aforementioned survey of debugging tools), Peter Deutsch (who created the structure-based editor for LISP), Daniel L. Murhphy and Warren Teitelman (who we encountered in the previous chapter as the author of the PILOT system).

The connection between debugging and error handling is immediately clear in the report, because the two topics are discussed in a single section, "Error Handling and Debugging Functions". The section opens by explaining that there are two classes of errors, "`H` errors for which the user can provide Help on the spot; and $\overline{\texttt{H}}$ errors for which no help is possible". By default, the $\overline{\texttt{H}}$ errors stop all computation, printing functions that have been entered by the failing computation. The error can be handled in code using the `errorset` function, which behaves as exception handler in modern programming languages. It runs a computation specified as an argument and, if no error occurs, returns its result. If an error occurs, the special value `NIL` is returned.

`H` errors allow the user to fix the mistake and let the program continue. This includes cases where a function or a symbol is undefined (or misspelt). The user can fix such errors interactively by typing the correct name or defining the function during the debugging session. The user can also induce (trigger) a `H` error interactively to break the execution as soon as the next function is entered. In addition to this, BBN LISP supports more conventional breakpoints, set by invoking the `break` function. This modifies a specified function to stop the execution when the function is called, optionally also checking that a certain break condition is satisfied.

By the end of the 1960s, many different tools for on-line debugging, testing and error handling were developed. There was no clear distinction between debugging, testing

and error handling and the different tools were often used together. This required practical hacker knowledge held by the practitioners and more often shared through personal interactions than through written documentation. However, the ongoing shift from the black art of programming to software manufacturing based on rigorous foundations was about to transform the practices of debgging, testing and error handling.

The Debugging Epoch Opens

While the on-line debugging techniques for early interactive computers and time-sharing systems were appearing in the 1960s, most commercial programmers were still debugging their programs using the two methods developed over a decade ago for EDSAC: the post-mortem memory dump and various forms of tracing that printed diagnostic information during the program execution. This was increasingly recognized as a major obstace for software development and, in November 1965, Mark Halpern made the problem clear in his article for the Computers and Automation magazine titled "Computer Programming: The Debugging Epoch Opens".[42] In the article, Halpern commends the recent developments in hardware and programming languages and discusses the next major challenge: "We have machines big and fast enough to execute most useful programs, and software varied and reliable enough to get those programs written; what now? Now: debugging."

Halpern, who had experience working at IBM and was now based in a research laboratory at Lockheed, sees "the debugging problem" as the new limiting factor for the growth of the computer industry. He uses the term debugging in the broader sense of identifying and correcting programming errors and proposes a number of new debugging techniques. One such technique that remains commonplace to this day is distinguishing program operation in "Debugging Mode" and in "Production Mode". Other techniques he proposes were not unlike some of those that were already becoming possible in LISP environments developed at the time by the hacker culture. Halpern identified a problem that was becoming increasingly visible at the time and envisioned some tools that would become available later, but he did not anticipate a different kind of response to the problem that was starting to take shape.

At the end of the 1960s, many stopped seeing the difficulty with software development as "the debugging problem" of eliminating small program flaws and started, instead, seeing it as an industry crisis.[43] This was a major cultural shift. As discussed in Chapter 2, practical programming in 1950s was dominated by the hacker culture of programming. It was a black art that relied on personal knowledge and the results were not reproducible or reusable. This state of the art was not acceptable to either the mathematical culture, which sought formal foundations for programming, nor the managerial culture, which wanted to be able to produce software of a predictable quality at a predictable cost.

Tackling the problematic state of computing became the subject of the NATO Software Engineering Conference held in 1968 in Garmisch. The idea for the conference came from Fritz Bauer, professor at the Technical University of Munich who was involved in the development of ALGOL, although he was not a signatory of the Algol 68 Minority Report that I discussed in Chapter 2.

Bauer and the NATO Study Group on Computer Science decided to structure the conference along the three themes of design of software, production of software and service of software. The organizers identified group leaders for each of the topics, many of whom were previously involved in the ALGOL efforts, and invited a limited number of "carefully

choosen leading figures“ from different backgrounds and different countries.[44] The attendees included a diverse group of academic and industry programmers, but the attendee list notably lacks representatives of the hacker culture of programming. The MIT hackers that we encountered in the previous chapter would likely disagree with many of the assumptions of the conference, but they also did not fit the traditional profile of an academic or industry leader in the field.

Consequently, the conference attendees quickly recognized and agreed on the problem with the “black art” approach of the hacker culture of programming. The agreement on the problem at the 1968 Garmisch conference, however, contrasted with the disagreement on the solution at the follow up conference in 1969 in Rome. The different cultures of programming interpreted the term “software engineering” differently and were more interested in pursuing their own agenda than agreeing on a common approach. According to the mathematical culture, the answer was the development of methods for formal reasoning about programs. The managerial culture saw the answer to the software crisis in scientific management of programming teams. However, the conference also marked the rise of the engineering culture of programming that hoped to address the software crisis by the development of new and better tools and programming practices to assist the programmer.

The development of software engineering brought a more structured approach to thinking about software and its development.[45] A concrete example of this was structured programming that I discussed in Chapter 2 and that soon acquired both engineering and managerial interpretation. However, the structuredness is apparent in the very organization of the Garmisch conference. It was divided, following the steps of the emerging development process, into sessions on design, production and service. The NATO conference report also shows more differentiation in the terminology used for talking about debugging and testing. It only mentions debugging in a general sense, but includes a number of more thorough perspectives on software testing.

First of all, the authors of the working paper “The Testing of Computer Software”[46] relate testing to the aim of the conference to provide a scientific basis for software development. The authors note that “testing is one of the foundations of all scientific enterprise” and argue that software should be “rigorously tested for conformity with the specification”. They go on to distinguish two different aspects of such testing:

> *This testing has two aspects: that carried out by the producer of the software to satisfy himself that it is fit for release to his customers and that carried out on behalf of the customer in order to give him sufficient confidence in the software to justify his paying for it.*

The working paper notes that the unfortunate current strategy is that “most customers accept the software without any form of test” and suggests a testing strategy that includes checking the documentation, system availability, verification of functionality and performance assessment. In other words, they describe the kind of testing that we might today refer to as acceptance testing.

A more fine-grained perspective distinguishing different kinds of tests is from John Nash at IBM whose working paper includes an illustration (Figure 4.8) that distinguishes unit tests, component tests and system tests. As we will see, this kind of division was becoming more commonplace as one aspect of the structured development process that later became known as Waterfall. Finally, another interesting idea from the NATO 1968

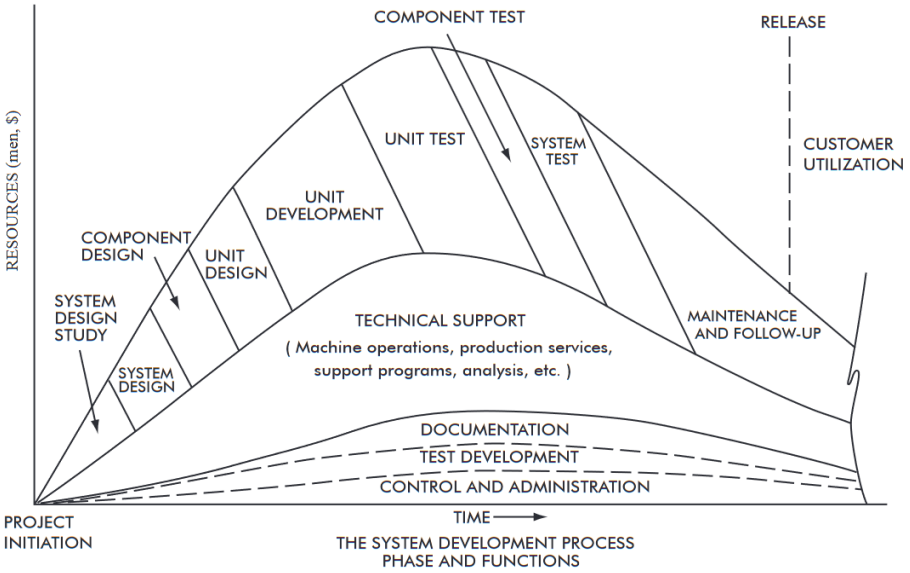


Figure 4.8: Figure “Some problems in the production of large-scale software systems” from the NATO 1968 Software Engineering conference report.

report is from a working paper “Aids in the production of maintainable software”. The author argues that testing should be automated and that “a collection of executable programs should be produced and maintained to exercise all parts of the system.”

The NATO conference proceedings show that the task of getting program to run as required, which was interchangeably called checkout, debugging or testing in 1950s, started to differentiate into a number of distinct concepts. Most importantly, the notion of testing is becoming a multi-faceted concept that covers the more managerial notion of acceptance testing done by a customer, as well as the engineering idea of automated testing as practiced for example by modern unit testing tools.

Testing and debugging also appears in the report from the 1969 Rome conference. Interestingly, most of the discussion appears in the “Software quality” section under the “Correctness” subsection, which covers formal correctness (“how to prevent bugs from ever arising”) and debugging (“how to catch them when you have got them”). It is the section on formal correctness that contains an argument in favor of careful testing by Perlis alongside with a famous remark by Dijkstra that “[t]esting shows the presence, not the absence of bugs.”

The discussion on debugging in the report is also enlightening. The list of participants of the 1969 Rome conference includes Warren Teitelman, who worked on the on-line debugging tools for LISP and is possibly the sole clear representative of the hacker culture in the list of attendees. His working paper, “Toward a programming laboratory” presented much of his work on LISP, including the PILOT system and interactive error correction. The working paper sparked a number of discussions at the conference, in particular on the relative merits of off-line debugging and on-line debugging. Niklaus Wirth, who worked on the Pascal programming language at the time, expressed his worry about the new developments:

> *There has been, since the advance of timesharing and on-line consoles, a very hectic trend towards development of systems which allow the interactive development of programs. Now this is certainly nice in a way, but it has its dangers, and I am particularly wary of them because this conversational usage has not only gained acceptance among software engineers but also in universities where students are trained in programming. My worry is that the facility of quick response leads to sloppy working habits and, since the students are going to be our future software engineers, this might be rather detrimental to the field as a whole.*[47]

Wirth's comment comes from the perspective of a mathematical culture, arguing that programmers should first thoroughly, perhaps even formally, think through the problem before writing any code. The engineering perspective defended by Alan Perlis in response is that "people don't think like that; conversational systems permit programmers to work from the top down and to compute even with incomplete specifications". Teitelman unsurprisingly shares this view and remarked that conversational systems promote "a more efficient, systematic approach to debugging and testing". In the report, Teitelman does not elaborate on what he means by "systematic", but making that explicit would likely pose a challenge. The ambition of the NATO conferences was to transfrom programming into a proper engineering discipline. This had almost no effect on debugging, which remained a hacker practice that is learened from experience and relies on the skills of individual programmers. The shift also did not visibly affect the work on interactive programming, which we follwed in the previous chapter. The conferences did however, mark the start of a transformation of exception handling and software testing.

Orderly and Reliable Exception Handling

When a program running on a batch processing system encountered an unexpected situation, it would terminate and optionally produce a postmortem dump. In interactive programming systems, an unexpected situation would break the execution so that a programmer could analyze the situation, correct it and resume the execution. In both cases, the assumption was that the unexpected situation would eventually be handled by a programmer. However, the premise of software engineering was to produce reliable programs that can gracefully recover from unexpected situations.

In LISP, the burden of handling unexpected situations could be moved from the programmer to the program itself. Since version 1.5, LISP included a function `errorset` that I mentioned about when discussing on-line debugging techniques. When an error occurred inside code wrapped with `errorset`, the function returns `NIL` and the programmer can write code that recovers from the error. The 1965 LISP 1.5 manual is, however, careful to say that continuing computation in a different direction when one path results in an error is only possible if the failing code has not modified any variables or caused other damage. In other words, doing the recovery correctly is still a task for a competent hacker.

Another early programming language that contributed to the development of exception handling is PL/I. The language predates the birth of Software Engineering and was developed at IBM as part of a broader effort to bridge the gap between scientific programming and business data processing. In mid-1960s, IBM developed a single family of computers to cover all possible application and designed the IBM System/360 computer

range as well as the operating system OS/360. The designers of PL/I believed that the strict separation of scientific and business computing was no longer beneficial and that programmers needed a single language to complete all their programming tasks. Some also hoped that having a single language may relive IBM from the burden of maintaining multiple compilers.[48] Although PL/I was modelled after FORTRAN, it also included rich data structuring capabilities and built-in functions for the handling of I/O, both motivated by the requirements of business data processing.

was a false stereotype and programmers needed a single language. Although PL/I was modelled after FORTRAN, it included rich data structuring capabilities and built-in functions for the handling of I/O. Operations for reading data were, however, a common source of program failures, which influenced the design of the language:

> *[Languages before PL/I] had few general facilities to define actions which were to be taken when conditions such as reading an end of file, overflow, bad data, new line, occurred. Often the conditions could not even be sensed in the language, and assembly language patches were required. (...) The [PL/I] designers were determined that entire real applications should be programmable in the language and these programs be properly reliable and safe.*[49]

To make programming of reliable and safe applications in PL/I possible, the langauge designers defined 23 types of error conditions and the `ON` construct which could be used to specify action to execute in the case the error condition occurred. The construct was cumbersome to use, partly because passing information to the exception handler often required using global variables and because the action of registering a handler had to be cautiously explicitly reverted.

Although LISP 1.5 and PL/I introduced much of what is needed for modern exception handling, it took another decade until more sophisticated exception handling started to became commonplace. The developments in LISP 1.5 and PL/I are notable, because they contributed to the differentation of the various tasks involved in getting programs to behave correctly. Whereas dealing with errors in 1950s was closely interconnected with testing and debugging and was a matter for hackers, the constructs like `ON` and `errorset` make it a part of the programming language itself. The terminology shows that this was a gradual process. When talking about the `errorset` function, the LISP 1.5 manual[50] talks about "errors", while PL/I manuals talk about "error conditions" and introduce the `ON` statement in a section on condition-processing statements.

Internalizing error conditions into the programming language moves the problem of error handling into the realm of the engineering and even mathematical cultures. PL/I is a good example. The language originated from engineering efforts at IBM and its design notably lacks criteria that would be typical for the mathematical culture such as the emphasis on solid theoretical basis and the ease of proving correctness. Yet, it was also extensively studied at the IBM Vienna lab, which produced multiple formal definition of the language. For some of the formalisation techniques, Algol served as a testbed, which resulted in the specifications that I covered in Chatper 2.[51]

The software engineering community that started to emerge in the wake of the NATO Software Engineering conferences had software reliability as one of its main concerns. In the mid-1970s, the issue of reliability also became increasinly important to programming language researchers and the resulting influences between the two communities have been well documented.[52] Work on exception handling is a prime example. In 1975,

John B. Goodenough published two papers that made major contributions to exception handling and testing. I return to the latter in the next section. Goodenough was working at a software consultancy developing software engineering tools and solutions, so he came from a markedly engineering background. Yet, his proposal for exception handling mechanism was presented at the ACM Symposium on Programming Languages (POPL) where it appeared among talks mostly rooted in the mathematical culture of programming. It then evolved into a publication in the Communications of ACM.[53]

The exception handling design proposed by Goodenough shares many aspects with modern exception handling mechanism as known, for example, from Java. Exception handlers are statically associated with blocks of code (i.e. using lexical scope) and can specify code to run for various exceptions that may be triggered in the enclosed code block. There is a number of built-in exceptions, but users can also define their own. Furhtermore, procedures are annotated with exceptions they may be raised and Goodenough argues that compiler should check that all possible exceptions are handled. In a feature that goes beyond what is supported by most programming languages today, the handler can also resume the execution of the code that raised an exception and there are three different constructs for raising exceptions: `ESCAPE` requires termination, `NOTIFY` forbids termination and `SIGNAL` where the termination behavior is decided by the exception handler.

Goodenough's paper was a detailed proposal for an exception handling mechanism, but it was theoretical and did not come with a specific implementation. It had, however, a direct influence on Clu and Ada, two programming langauges from the late 1970s that embody the ideals of software engineering in the form of a programming language. Clu was created by Barbara Liskov at MIT and was motivated by the work on programming methodology and information hiding. Its key innovation was the design of abstract data types, which I cover in the next chapter, but Liskov also wanted to support "robust or fault-tolerant programs (...) that are prepared to cope with the prsence of errors".[54] For this reason, Clu adopted an exception handling mechanism that was inspired by both Goodenough's paper and the exception handling mechanism in PL/I. It also made two design decisions that are common in programming languages today. First, it referred to the mechanism as "exception handling" and, second, it only implemented the termination model,i.e. there was no way to resume the program from the original location after an exception. The development of the Ada language was led by the US Department of Defence and was the result of a managerial Software Initiative to reduce the High Costs of Software. The designers believed that "a common language for use throughout the DoD would be a first step to building an engineering discipline of software, to supersede the then current 'black art'."[55] The design of Ada was a product of thorough and detailed analaysis and was described in a 250-page long "Rationale for the design of the Ada programming language".[56]. The Rationale makes a reference to Goodenough, as well as other ongoing research on exceptions and adopts the terminology of exception handling as well as the termination-only mode of exceptions just like Clu.

Whereas the development of exception handling in PL/I was focused on solving the specific problem of handling of specific error conditions related to input and output issues, much of the later work influenced by the rise of software engineering had a broader realm. Already in LISP, the exception handling mechanism based on `errorset` was used "not to trap and signal errors but for more general control purposes (dynamic non-local exits)"[57] and MacLisp later added a pair of primitves `catch` and `throw` so that the original exception handling primitives could be reserved just for error handling. More interest-

ingly, Goodenough also saw the proposed exception handling mechanism "in general, as a means of conveniently interleaving actions belonging to different levels of abstraction".[58] This was the case especially with exceptions that are intended as notifications for the caller who then resumes the execution of the code triggering the notification. Similarly, in Clu, exceptions were seen as "a general way of conveying information from a called procedure to its caller."[59]

In the case of LISP, the LISP 1.5 Manual does not indicate that `errorset` should be used for other purposes than for a graceful recovery from an error. The more general use of the mechanism was likely the result of the hacker culture, which was able to exploit the mechanism for originally unintended clever programming tricks. However, Goodenough's paper and Clu present a cultural shift and exceptions are seen as a langauge mechanism that allows multiple good software engineering practices, including both error handling for reliability, but also good structuring of code into multiple levels of abstraction. In a perhaps surprising plot twist, Java again constrained the use of exceptions to just error handling in the 1990s only to have the trend reversed, by recent work on algebraic effect handlers that are, yet again, an exception-inspired programming language construct for interleaving actions across multiple levels of abstraction.[60]

System Structure for Fault Tolerance

The work on exception handling in programming languages provided a way of recovering from errors, a more general program structuring mechanism, but it also served as the basis for over-engineering, another practical engineering practice that Hoare later saw as a practice contributing to software reliability. In the analysis of exceptions that Goodenough presented in his paper, he classified exceptions into two categories. A *range failure* "occurs when an operation (...) finds it is unable to satisfy its output assertion" and *domain failure* "occurs when an operation's inputs fail to pass certain tests of acceptability." In other words, if something goes wrong when performing the operation, it is a range failure, but when the operation is invoked incorrectly, it is a domain failure.

The former category of program errors is inevitable. A program cannot prevent real-world situations such as corrupted file. However, in an ideal program imagined by the mathematical culture of programming, a domain failure would never happen. A program would be written so that the acceptability of inputs is ensured by the callers of the operation and, in an ideal world, even guaranteed by a formal proof. The idea that an operation should test acceptability of inputs is thus a basic form of over-engineering, or, preempting errors that should not happen in theory, but do happen in pracitce. In modern programming terminology, the practice of checking for invalid input is widely adopted as part of the *defensive programming* methodology that also dates back to mid-1970s.[61]

The software engineering research on software reliability inspired a number of other programming language designs that would simplify over-engineering. One example that illustrates possibilities beyond those that are supported in main-stream programming languages today came from Brian Randell. Randell was a signatory of the Algol 68 Minority Report, attended both of the NATO conferences and was the editor of the proceedings for the first one and dedicated much of his career to work that contributes to the engineering culture of programming. In 1969, he moved from IBM to Newcastle University and started working on software reliability. In 1975, he published an influential paper "System Structure for Software Fault Tolerance"[62] that introduced so called *recovery blocks*.

Recovery blocks are a programming language feature designed for building of fault-tolerant systems that can recover from design inadequacies. A recovery block is similar to exception handler, but the body is guarded by an acceptance condition (a check for a range failure). If the condition is violated by the primary implementation, the block proceeds to execute one of the alternative, possibly simplified, implementations, until it obtains a result that satisfies the acceptance condition. Another idea implementing the idea of over-engineering that emerged from the software engineering community in the mid-1970s was to build systems with a self-checking mechanisms.[63] Such systems would actively monitor themselves to detect range and domain failures. While none of the mechanisms going beyond exception handling appeared in widely used programming languages, they normalized over-engineering as a way of tackling programming issues and introduced a mechanism that was, in various forms, implemented in numerous later programming systems.[64]

Following the birth of software engineering, exception handling moved from a manual task for a programmer into a condition to be handled automatically, by the program itself. Consequently, it also turned from an interactive programming system feature into a programming language construct. Those working on exception handling started to come less from the hacker culture and more from the engineering and mathematical culture of programming. Testing, another aspect of reliable software development, was also soon to be transformed by the birth of software engineering.

Professionalization of Testing

New efforts to organize the community around software testing emerged at the start of the 1970s. In 1972, Bill Hetzel organized the Computer Program Test Methods Symposium at the University of North Carolina that brought together many of those interested in testing. Mirroring the narrative about the state of software development as a whole, Hetzel complained that testing "is all art" and lacks an "established discipline to act as a foundation."[65] The symposium and the book "Program Test Methods"[66] that was edited and published by Hetzel following the symposium were an attempt to put testing on a rigorous basis.

Hetzel's efforts were seen as a promising step. According to a report written for the ACM by one of the symposium attendees, David J. McGonagle, "the programming industry took another step toward maturity"[67] with the symposium. This step was more in the ambition of the symposium than in its contents and the presentations at the event illustrate that there was not yet a coherent testing community. They included managerial reports on the testing done for large software systems, engineering reports on new testing tools, proposal for standardizing software testing practices, but also a number of mathematical talks on proving program correctness. As reported by McGonagle, the event "served to bridge the gap between the proof of correctness people on one hand and the large systems users on the other". McGonagle doubts that the event "convinced any of the non-believers of the critical nature of this area of computer systems work."[68] However, the symposium was just a start. It was followed by a series of events on testing that, over the next two decades, established software testing as a recognized software engineering discipline.

The organizer of the 1972 symposium later co-founded a consultancy focused on software testing together with David Gelperin. The two played an instrumental role in promoting software testing and co-authored a paper "The Growth of Software Testing"[69] that also helped establish testing as a discipline by providing its origin story. In the paper, Gelperin and Hetzel recognize the early 1970s as the era when the level of professionalism in soft-

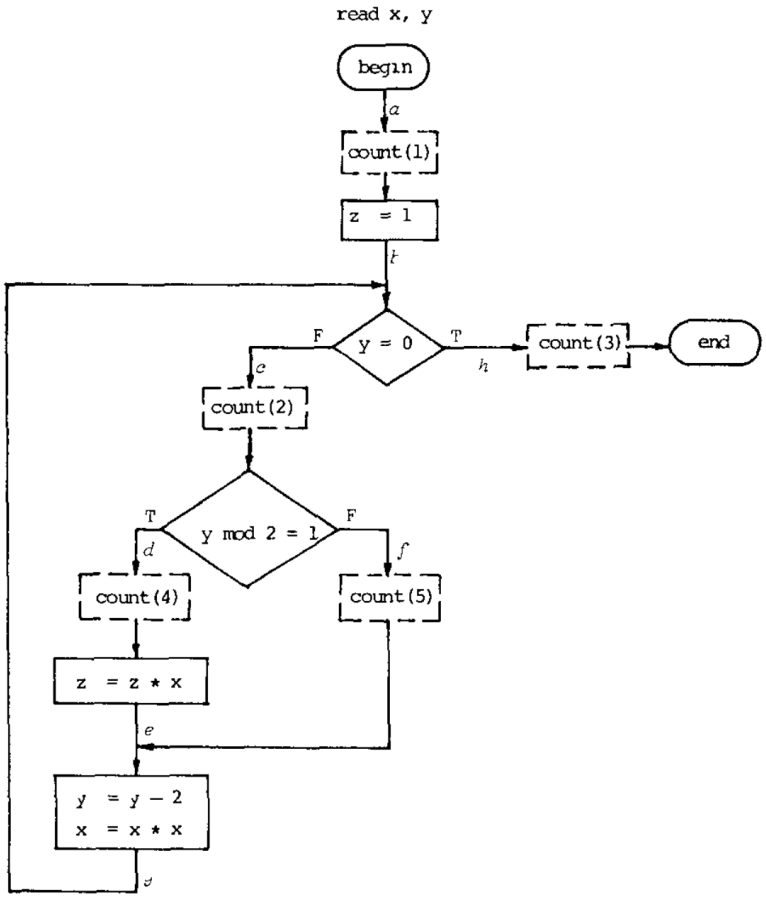


Figure 4.9: A flowchart for a sample program from Huang (1975). Each branch (arrow) is annotated with a counter `count(n)` with different number `n` for different counters. A good set of tests will make sure that each counter is incremented at least once.

ware testing significantly increased. In this period, the first software testing conferences took place and testing became the topic of increasing number of academic papers. Companies also started to recognize software test engineering as a speciality and started to hire specifically for positions such as “test manager” or “test technician”.

In the 1970s, testing became clearly distinguished from debugging. The goal of the latter was to “make sure the program runs” while the goal of the former was to “make sure the program solves the problem”. As the positive phrasing suggests, testing was seen as a way of showing that the software satisifes its requirements. As such, it had the same ultimate objective as the work on proving programs correct, but it utilized engineering rather than mathematical methods. This led to a debate between the proponents of the engineering culture and the mathematical culture, which questioned the extent to which testing can guarantee program correctness.

At the 1969 NATO conference, C. A. R. Hoare argued that one can easily show the “ultimate futility of exhaustive testing.” In the report, this is contrasted with a point made by Alan Perlis that “[m]uch of program complexity is spurious and a number of [well-chosen] tests will exhaust the testing problem”.[70] In the 1970s, much work on tools and methods for testing focused on finding the right tests. But what counts as the right test?

I suggest that we can see the debate about limits of testing as a clash between the mathematical and the emerging engineering culture of programming. The most convincing arguments trying to show that testing can show the absence of bugs came from engineers who utilized mathematical methods to counter the arguments made by the proponents of the mathematical culture of programming by using their own means.

Three prominent examples of this approach appeared in mid-1970s.[71] The first was from J. C. Huang whose PhD was in switching and automata theory, but who also had practical experience as an engineer.[72] Huang acknowledges that doing exhaustive testing, even for a simple program that takes two 32-bit integers as inputs and completes in 1 millisecond would still take 50 billion years. Huang's answer is to not treat the program as a black box, but analyze its structure and find a set of tests such that each branch in the program runs at least once (Figure 4.9). John B. Goodenough, who we encountered in the section on exception handling, also proposed a method for choosing test inputs.[73]. He also indirectly responds to Hoare's comment when he claims that "properly structured tests are capable of demonstrating the absence of errors in programs." Finally, at the same time, Thomas J. McCabe, who was employed by the National Security Agency and later established his own software consultancy, published the "cyclomatic complexity" measure.[74] The measure is calculated based on the number of branching points, paths and loops in a program flowchart. Although this was later used without reference to testing, McCabe suggested to use the metric as a basis for a testing strategy. If the number of tests is lower than that required for a program of a certain complexity, either more tests need to be added or the program structure needs to be simplified.

The next period of software testing that Gelperin and Hetzel refer to as "destruction-oriented" started in 1979 with the publication of the textbook "The Art of Software Testing".[75] It is ironic to see "art" involved yet again, but the author does not advocate "black art". Instead, the title is intended to suggest that the book takes a practical rather than theoretical perspective. The author acknowledges that "many subjects related to testing such as the idea of mathematically proving the correctness of a program were purposefuly excluded". The book adopts an engineering perspective that draws on methods for test selection developed throught the preceding decade. Yet, it also still recognizes the hacker heritage of software testing. In section on "Error Guessing", Myers admits that some people seem to have a knack for smelling out errors and that their error-guessing technique is largely an intuition and ad hoc process.

The book is a significant milestone not just because it is the first book solely dedicated to software testing, but because it shifts the perspective on the purpose of testing. Meyers sees testing as "the process of executing a program with the intent of finding errors". This way of thinking about testing presents a shift from the earlier view which was to show program correctness. It is also a shift from a managerial thinking of testing (acceptance testing done by a customer as part of the development process) to an engineering thinking (testing done by engineers to help with their job). It also happens to be more practically effective as it encourages test selection that is more likely to find faults. It prompts testers to look for edge cases that might not be considered when the aim is to make sure that a program works as required. Finally, the new perspective on testing aligns it with other approaches to fault detection. For example, Myers' book talks about code inspections and code walkthroughs and labels those as *non-computer-based* (or human) *testing*.

The halmark method of the engineering culture is the create new tools that assist with solving the problem at hand. Such tools tend to be less well documented in the literature,

but there is a number of good examples for the destruction-oriented period of software testing. The idea for one such tool came from Richard Lipton, who had it as a student in 1971, long before he became known as a co-author of the 1977 paper "Social Processes and Proofs of Theorems and Programs" that I talked about in Chapter 2. The idea was termed "mutation testing" and it was published both as a paper and as a tool presentation in 1978[76]. The idea of mutation testing is to automatically change some aspect of the program logic and see if this change is detected by the tests. For example, a program that sorts a list of numbers contains a comparison `X<=Y` to check if one element is less than or equal to another element. A mutation testing tool might change this check to, for example, `X!=Y` testing for inequality instead. A well-designed test suite can detect such changes in logic. If the tests fail to detect that the modified program is incorrect, it is a hint that more tests need to be added.

The developments that I followed in this section mostly focused on the engineering aspects of testing, but testing was also a crucial part of the managerial approach to software engineering where it was one of the phases in the application development lifecycle. To document this, I first need to return to the end of the 1960s and follow the managerial response to the industry crisis.

A Slightly Ominous Note for Information Processing Management

The 1960s started on a high note for the managers working in the commercial computing industry. The industry was growing at an impressive annual rate of 27 percent and the number of computers installed in the United States grew from 54 hundred to 74 thousand during the decade.[77] Yet, many were also increasingly aware of the limitations and issues with the use of computers. There was a growing body of anecdotal evidence of problems with complex computer systems that that experts in the field were familiar with.[78] The Mariner 1 spacecraft was lost due to a programming error in its guidance system in 1962. The IBM OS/360, delivered in 1967, was delayed by 9 months and cost the company half a billion dollars, four times the original budget. Fred Brooks Jr., the project manager of the delayed IBM OS/360 system later likened the development of large software systems at the time to a tar pit striggle:

> *No scene from prehistory is quite so vivid as that of the mortal struggles of great beasts in the tar pits. (...) Large-system programming has over the past decade been such a tar pit, and many great and powerful beasts have thrashed violently in it. Most have emerged with running systems—few have met goals, schedules, and budgets. Large and small, massive or wiry, team after team has become entangled in the tar. No one thing seems to cause the difficulty–any particular paw can be pulled away. But the accumulation of simultaneous and interacting factors brings slower and slower motion.*[79]

The reasons for the issues were both technical and managerial. The rapid growth of the computing industry meant that computers were used to solve increasingly large and complex problems. It was no longer possible for a small group of highly skilled hackers to keep the structure of an entire system in mind while building it and software development started to require large teams which, in turn, meant that much more coordination and communication was needed. The programming languages at the time provided only basic

mechanisms for structuring and organizing large code. The fact that a "subroutine library" referred to a filing cabinet with punched cards is merely a testimonial to that.

For the engineering culture, the NATO Software Engineering conferences are often seen as the pivotal moment. The conferences brought together participants from universities, industrial research labs, computer manufacturers, computer software and service firms as well as the governments. But as pointed out by Thomas Haigh[80] a more careful look reveals that most attendees were involved in the development of technically interesting software such as operating systems, programming languages and compilers. Notably absent were discussions of application software, that is the bread and butter of programming such as travel booking or payrol systems.

The managerial perspective on the growing problems with the rise of computers was best captured in a report 'Unlocking the Computer's Profit Potential"[81] published by McKinsey and Company in 1968. The report was widely discussed in computer magazines after its publications. The "Editor's Readout" of the Datamation magazine from August 1968 labels the report as "a slightly ominous note for information processing management", while the republication of the report in the Computers and Automation magazine in April 1969 ensured it reached an even broader audience. The authors of the report studied the use of comptuers at 36 companies. Despite the technological advances, from a profit standpoint, "computer efforts in all but a few exceptional companies are in real, if often unacknowledged, trouble."

The basic problem, according to the report, is that most companies completed "conversions of routine administrative and accounting operations to computer systems", but are now finding it difficult to effectively apply computers in other areas. The computer staff who completed the conversion projects are now leading computerization efforts, but are "seldom strategically placed (or managerially trained) to assess the economics of operations fully or to judge operational feasibility." The McKinsey recommendation was for the top management to take a more active role in the management of computer projects:

> *Many otherwise effective top managements, however, are in trouble with their computer efforts because they have abdicated control to staff specialists (...). Only managers can manage the computer in the best interests of the business. The companies that take this lesson to heart today will be the computer profit leaders of tomorrow.*

The report coincided with a shift in thinking about the nature of programming job.[82] Before the end of the 1960s, programming was seen as a creative activity that required unique skills and often also unique treatment. Towards the end of the decade, the literature aimed at top management stopped talking about programmers as weird but ingenious hackers and started seeing them as any other kind of worker. The managerial approach to turning programming into a "proper engineering discipline" was thus modelled after industrial factories and industrial production lines. The managers hoped that such model can lead to reliable and predictable software production, without relying on particular skills of individual programmers. They started looking for a methodology that could control the development process, not so much by appropriately controlling and structuring the code being written, but by appropriately controlling and structuring the team writing it.

Many different attempts at managing programmer teams soon appeared. In Chapter 2, I talked about the Chief Programmer Team methodology and the Cleanroom software engineering approach, which were modelled after the structure of a surgical team and were

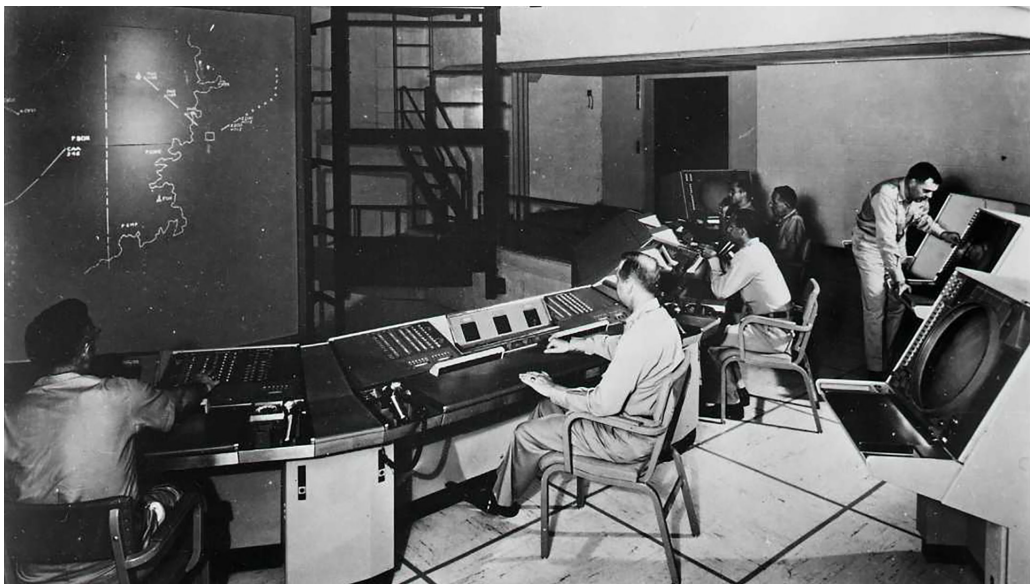


Figure 4.10: The Semi-Automatic Ground Environment (SAGE) Air Defence System

developed by Harlan Mills at IBM throughout the 1970s. Other approaches include egoless programming,[83] which was a democratic management structure focused on collaboration with a rotating leadership. However, the dominant approaches were top-down development methodologies inspired by the, initially mathematical, idea of structured programming. The specification of the system had to be written, decomposed into parts that can then be developed according by different teams. The most prominent structured approach to software engineering became known as the Waterfall development model.

Production of Large Computer Programs

The software development model that became known as Waterfall in the 1970s was, in fact, used already in the 1950s during the development of the Semi-Automatic Ground Environment (SAGE) system (Figure 4.10), which utilized the interactive Whirlwind computer that I briefly mentioned in the last chapter. SAGE was a system of computers to process live data from multiple radar sites across the US that was responsible for identifying a Soviet nuclear strike and coordinating the response. It was an effort of a unique scale. The core development team "grew from 125 personnel in 1956 to 1,400 in 1962" and the organization behind SAGE "trained more than 700 programmers between 1955 and 1957."[84]

The development methodology eventually used at SAGE has been presented in a 1956 paper "Production of Large Computer Programs" at the Symposium on Advanced Programming Methods for Digital Computers in Washington D.C., but without any specific reference to SAGE, which was not yet publicly announced. The paper came perhaps too early and was largely forgotten, until it was reprinted a quarter of a century later.[85] According to the paper, the development of the main SAGE control program proceeded in nine phases documented in Figure 4.11. The operational plan defines the design requirements and is used to produce operational specifications. This should be sufficiently detailed so that programmers can perpare the program using only machine and operational specifications. Coding specifications then describe the structure of the program with enough detail that make it

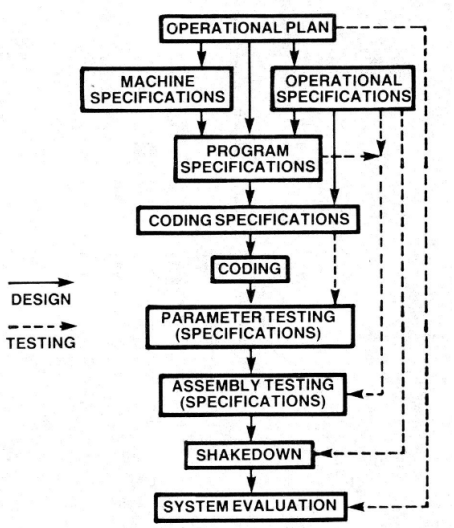


Figure 4.11: "Production of a large program system proceeds from a general operational plan" which defines broad design requirements for the complete control system and is jointly prepared with the user of the system. The dashed line indicates that testing is guided by a specification, not by the coded program. Adapted from Benington (1983).

"possible to predict precisely the output of [each component] subprogram for any configuration of input items." The coding is followed by two-phase testing. In parameter testing, each component is tested according to its specification in isolation, in "an environment that simulates pertinent portions of the systems program". This is then followed by a testing of the assembled program, first using simulated and then using live data.

The development methodology of SAGE may not have directly influenced later managerial development methodologies, but it has many aspects that are characteristic of methods that became widely used in the coming decades. Most importantly, it splits the process into a number of phases that are undertaken in sequence. Testing is treated as a separate phase and tests are based on the developed specifications, rather than being based more directly on the system implementation. The goal of testing in the SAGE methodology is to show that the system satisfies the specification, i.e., it falls into the "demonstration-oriented" period of testing as classified by Gelperin and Hetzel. The author of the write-up is, however, also anticipating the later debate about the viability of this approach:

> *It is debatable whether a program of 100,000 instructions can ever be thoroughly tested – that is, whether the program can be shown to satisfy its specifications under all operating conditions.*

The individual phases of the SAGE development methodology are remarkably similar to those of the "Waterfall" development model that became widely known and used from the start of the 1970s. The Waterfall model is sometimes attributed to Winston Royce, who describes it in a paper "Managing the Development of Large Software Systems" published in 1970,[86] although without using the term "Waterfall". The model, illustrated in Figure 4.12 starts with requirements capture, follows with analysis and program design, actual coding, testing and concludes with program operation. However, Royce describes the basic model only as a starting point for further discussion. He "believes in this concept", but points out that the basic method is risky and invites failure.

Royce starts with a naïve version of the process that proceeds only in the forward direction, but soon notes that the methodology needs to be iterative. The chief difficulty,

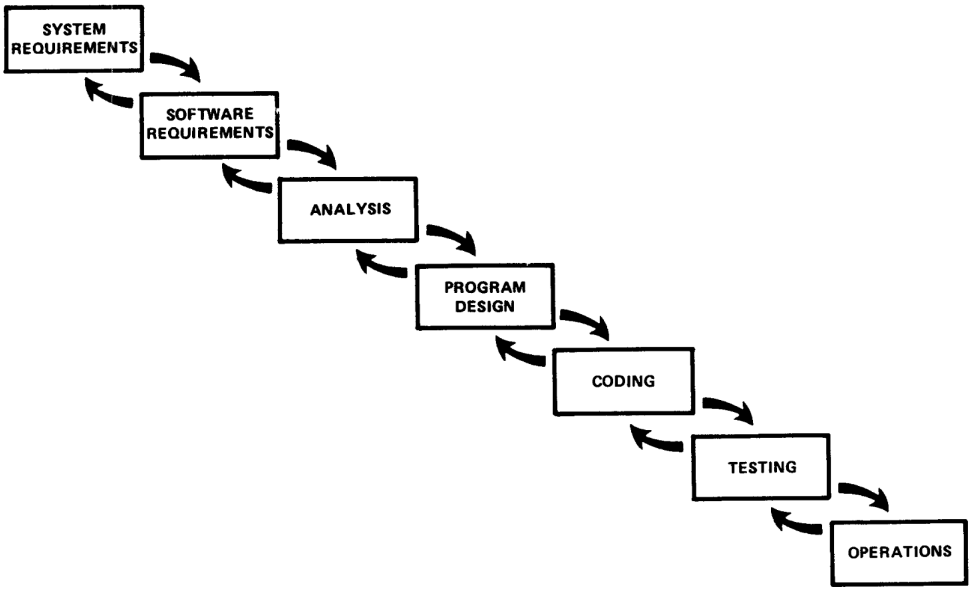


Figure 4.12: "Implementation steps to develop a large computer program for delivery to a customer" as proposed by Royce (1970) with iterative interactions, hopefully confined to successive steps.

according to Royce, is that we may sometimes need to go back by multiple steps. For example, the testing phase, which only occurs after coding is done, may reveal timing, storage and input/output issues. Those are not precisely analyzable upfront and may require design changes, which then, in turn, require new implementation and "result in 100-percent overrun in schedule and/or costs." Royce goes on to suggest five ways in which the process can be improved, including two that will become major themes in later developments that I will return to later in this chapter. The first is to arrange matters so that the version finally delivered is actually a second version, which is what Fred Brooks Jr. recommends in his "Plan to throw one away"[87] essay. The second is to involve the customer prior to the final delivery, which is the essence of a shift that started in the late 1980s and eventually led to the Agile development methods.

Yet another way in which the process can be improved focuses on testing and is documented with a diagram shown in Figure 4.13. Here, Royce recommends to involve a dedicated team of "specialists in software product assurance" who should test the system based on a detailed documentation, performing manual inspection of code to catch simple errors and manual checkout for identifying more complex bugs. An interesting proposal is to "test every logic path in the computer program at least once with some kind of numerical check." This is an informal sketch of a testing approach that was to be developed more formally over the next five years in the work on test data selection that I discussed in the preceeding section.

Software development methodologies like Waterfall impose a structure on the development process. They organize the work into phases, each of which results in outputs such as documentation, specification or code. This makes it possible to define milestones and monitor project progress. In other words, such methodologies make the opaque hacker process of building software legible to a managerial outsider. The early work on devel-

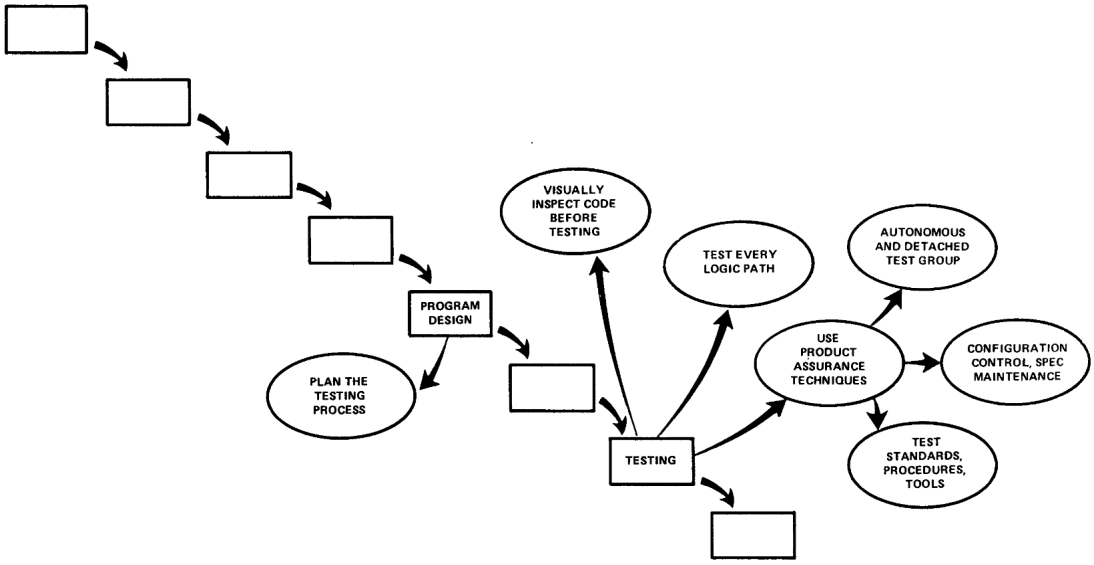


Figure 4.13: Additional aspects for testing outlined by Royce (1970) in a section “Plan, control and monitor computer program testing“

opment methodologies had origins in the managerial culture of programming and it illustrates a typical way of working in this culture. In order to professionalize software development, its proponents attempt to develop industry standards for the development process.

There are three notable organizations that contributed to the standardization of software development processes and the role of testing in those. The Institute of Electrical and Electronics Engineers (IEEE) established a task group on software engineering in 1979. The task group did not, at first, attempt to standardize the process or the best practices for testing. It started by developing a standard for test documentation,[88] which should be used for documenting test activities that occur as part of the software development process. This was followed by two more specific standards, specifying the process for ensuring that products meet their requirements (verification) and that they satisfy user needs (validation),[89] as well as a detailed process for testing individual software components (unit testing)[90]. At the same time, the National Bureau of Standards (NBS) published its “Guideline for Lifecycle Validation, Verification, and Testing of Computer Software”,[91] which was intended primarily as a guideline for federal agencies developing new software.

Gelperin and Hetzel choose the 1983 publication of the NBS guideline as the beginning of the “evaluation-oriented” period of software testing. In this era, tests are used not only to find bugs, but also to assess the project progress. The NBS guideline recommends a number of different kinds of testing throughout the software development lifecycle. Unit testing, integration testing and system testing is done at the end of the programming and testing phase while acceptance testing is done manually, by the customer, at the end of the software installation phase. The guideline also includes regression testing, in order to prevent the reintroduction of errors during the operation and maintenance phase. The tests further increase the legibility of the development, becuse managers can easily track the testing progress.

The standards published by IEEE and NBS were merely recommendations, but several organizations soon started requiring that software providers follow a specific process.

Most notably, the US Department of Defense made its own standard of the software development process, published under the code DOD-STD-2167A in 1985, a requirement. A contractor for the Department of Defense was required to follow a process consisting of a number of activities such as system and software requirement analysis, preliminary and detailed design, coding, as well as unit testing, integration testing and system testing.

Gelperin and Hetzel, the authors of the 1988 paper "The Growth of Software Testing"[92] whose testing eras I adopted in this chapter, were not historians, but software engineering consultants. They established a joint consulting business in 1986, started organizing a conference series on software testing and published a well-regarded book on software testing in 1988.[93] Their retrospective on testing was not written just to document the history of testing, but also to define the new trends. After discussing the "evaluation-oriented" period, the authors turn to an advocacy of their own approach which they label as "prevention-oriented" and which purportedly started a year before the publication of their paper and the book.

In the envisioned "prevention-oriented" period, test planning, test design and development and testing is an ubiquitous activity that happens in parallel with all other steps of the software development lifecycle. Testing is no longer done in certain steps of the process, but throughout the development. Gelperin and Hetzel capture this idea in their consulting company motto, "Test, than code", which they wore proudly on a blue pin at the Fourth International Conference on Software Testing in 1987.[94] Gelperin and Hetzel discuss their ideas using the vocabulary of the managerial culture. They talk about development phases, testing lifecycle and outputs such as the "Master Test Plan". Yet, some of their ideas are closely related to Agile development methods that we will encounter later in this chapter. But first, I want to discuss a more conceptual change that followed the birth of software engineering.

Disciplinary Repertoire of Software Engineering

The engineering culture of programming that grew in prominence in the 1970s was behind new practical development methodologies and new approaches to testing and error handling. More importantly, it also established a new way of thinking and reasoning about software systems, for which Rebecca Slayton coined the term "disciplinary repertoire" of software engineering.[95] The disciplinary repertoire allowed software engineers to search for fundamental laws and principles of software engineering and use those to shape public debates about software systems, such as the missile defence documented by Slayton.

As we saw in Chapter 1 the central notion of debates in the mathematical culture of programming became that of an *algorithm*. The disciplinary repertoire for reasoning about algorithms include primarily formal mathematical methods. For example, the *undecidability of the halting problem* is a mathematical theorem with a proof. It states that it is impossible to construct an algorithm, which would determine whether an arbitrary computer program will finish running or continue running forever. For the mathematical culture, this has wide-ranging theoretical implications about what kinds of programs can never be constructed. But to the engineering culture, it seems like a theoretical curiosity that tells only little about practical limits of building large-scale software applications. The rise of software engineering gradually led to the recognition that a software *system* has a very different nature than a computer *algorithm* and that a different language for talking about them is needed.

The difference between a computer algorithm and a software system was crucial for the critique of program verification by DeMillo, Lipton and Perlis that I discussed earlier:

> *[T]here is no continuity between the world of FIND or GCD and the world of production software, billing systems that write real bills, scheduling systems that schedule real events, ticketing systems that issue real tickets.*[96]

The fundamental difference is that "the specifications for algorithms are concise and tidy, while the specifications for real-world systems are immense, frequently of the same order of magnitude as the systems themselves. The specifications for algorithms are highly stable, stable over decades or even centuries; the specifications for real systems vary daily or hourly."[97] "These are not differences in degree. They are differences in kind."[98] In other words, the disciplinary repertoire for talking about algorithms may work well for algorithmic programs, but it is not applicable to large software systems.

The large software systems that software engineering needs to think about are not "composed of nothing more than algorithms and small programs". According to DeMillo, Lipton and Perlis, "the colorful jargon of practicing programmers seems to be saying something about the nature of the structures" that programmers work with. The implementation of any such large systems is made of "patches, ad hoc constructions, bandaids and tourniquets, bells and whistles, glue, spit and polish, signature code, blood-sweat-and-tears, and, of course, the kitchen sink." From the late 1960s, engineers started to figure out what kind of fundamental laws and principles may apply to software systems constructed in such a way.

Software Aspects of Strategic Defence Systems

The most notable steps in the development of the disciplinary repertoire of the engineering culture of programming were triggered by reflections on or by thinking about concrete software systems. This is a pattern characteristic for the engineering culture of programming. We saw that the work on software development methodologies started in the 1950s with the reflections on the development of the SAGE system and we will see this pattern in most of the laws and principles that I look at in this section.

The case of the anti-balistic missile (ABM) defence,[99] illustrates the development of the disciplinary repertoire of software engineering from the the end of the 1960s until the late 1980s. The development started at the time of the NATO Software Engineering conferences, although the most active participants in the discussion about ABM attended were absent from the two conferences. The US has been developing various forms of the ABM defence since the 1950s, but the announcement of the new "Safeguard" system in 1969 raised public profile of the program and triggered a new debate. The Safeguard anti-ballistic missile system was intended to protect US nuclear launch sites from limited forms of enemy attacks. A number of policymakers as well as computer scientists opposed it. This was often for political reasons, but opposers understood that a more effective argument needs to be based on technical reasons why such a system cannot reliably work.

The first contribution to the debate came from J. C. R. Licklider, who was instrumental in the development of time-sharing systems and the internet during his time as a research administrator in ARPA and was now a professor at MIT. Licklider was invited to write a

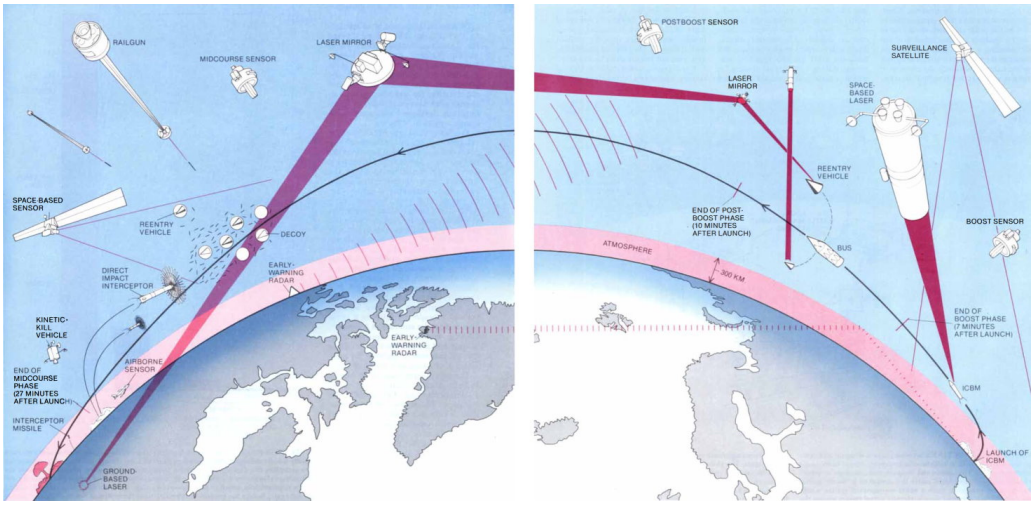


Figure 4.14: Illustration showing the proposed hardware for the Strategic Defnce Initiative from "The Development of Software for Ballistic-Missile Defense" by Herbert Lin

chapter on reliability for a study on the feasibility of the ABM. He soon concluded that the difficulties with the development of similarly complex systems are a common wisdom among the practitioners, but that there is very little well documented evidence of past failures and their reasons. In his report "Underestimates and Overexpectations",[100] Licklider resorts to talking about "common wisdom", which is that "for a certain class of systems, costs and times tend to be grossly underestimated and performance tends to be mercifully unmeasured." The tricky class of systems has a number of characteristics. They are complex, operate in operating conditions that are not fully under the designers control, their task changes over time and they are difficult to test in realistic scenario. Licklider discusses a number of examples, including the SAGE and the American Airlines travel booking system, which both suffered from underestimates and overexpectations. He concludes that:

> *[C]omplex software subsystems can be "mastered" (...) and made to provide useful and effective service if they can be developed progressively, with the aid of extensive testing (...), and if they can be operated more or less continually in a somewhat lenient and forgiving environment.* [101]

The question then is whether the ABM system satisfies the conditions that are needed in order for it to be mastered. As a complex system that cannot be dependably tested and which operates in an evolving hostile environment, Licklider's analysis raises serious concerns about the possiblity of building the software component of the ABM system. At the same time, Daniel McCracken, who was a well-known programming book author, started a more plainspoken effort, the organization Computer Professionals Against ABM (CPAABM). In a press release, the signatories simply stated that the "Safeguard antiballistic missile system should be abandoned because its control computer will not be able to perform the complex job assigned to it."[102] This is because the "pattern of development that must of necessity be followed with Safeguard, is highly unlikely to lead to a successful computer system." The group argued that there are three main issues with the system. First, its precise task cannot be defined, because the countermeasures developed by the

enemy cannot be expected; second, realistic testing of the system is impossible; and third, the Safeguard system cannot be developed through evolution.

The arguments used by the CPAABM are similar to those of Licklider. They focus partly on the inherent complexity of such system and partly on the properties of the environment in which the system needs to be built and operated. However, the arguments are not based on formal proofs or impeccable empirical evidence. They are generalizations based on the somewhat anecdotal evidence about past systems.

The question of necessary evidence for making such claims sparked a debate among the computing community. Herbert Bright, an ACM council member, was one of those who believed that past experience is not sufficient for claims about future systems. He expressed "a profound distaste for saying we believe it won't work when the truth is merely that we suspect it won't work".[103] The system would not violate any known physical or mathematical law and so "it's unfeasible to develop a sound basis for a proof that the proposed system cannot work". Joseph Weizenbaum did not feel obliged to provide a mathematical proof: "My suspicion is strong to the point of being belief. I don't think that my statement as a professional that I hold this belief obligates me to a mathematical proof."[104] However, Weizenbaum did not base his suspicion merely on personal hunches or the analysis of past failures. His belief, summarized and interrogated by Slayton, was based on an informal, but logical argument:

> *Computer systems could only become reliable if "the environment that they are to control or with which they are intended to cooperate must have a change rate smaller than that of the [computer] system itself." But since the missile defense "environment is actively hostile and destabilizing by intent," the software could not keep up with its changing requirements: "convergence is logically impossible."*[105]

Weizenbaum's argument is an example of the developing disciplinary repertoire of the engineering culture of programming. It uses logical reasoning to talk about things such as "the environment" and "change rate", which are crucial for large complex systems, but can hardly be defined in a fully formal way. We will see this approach to reasoning about software systems repeatedly in works such as "The Mythical Man-month" by Fred Brooks, but also in subsequent debate on the anti-ballistic missile defence.

The Safeguard system was eventually built at one site and was briefly operated, before being shut down, because the politicians accepted that its utlity "will be essentially nullified in the future".[106] This was not the end of the US ABM efforts, however, and in 1983, Ronald Regan's administration proposed a new system, Strategic Defense Initiative (SDI), nicknamed the "Star Wars program". The Star Wars program was intended to protect the US from ballistic missiles by destroying them before they hit their target. The program explored various ways of doing this, ranging from laser beams to projectiles dropped from a satellite (Figure 4.14). Again, the system also required the development of a very complex control software that would detect the attack, calculate missile trjectories, coordinate interception and discern decoys from actual warheads.

One of those invited to study the computer aspects of the SDI was David L. Parnas, a computer scientist known for his software engineering work on information hiding. On June 28 1985, Parnas resigned from the SDI panel and published a series of notes that he wrote while being a panel member. In the notes, republished as "Software aspects of strategic defense systems"[107] by the Communications of the ACM, Parnas concludes that

“we will never be able to believe, with any confidence, that we have succeeded” in building trustworthy software for the SDI and that ”nuclear weapons will remain a potent threat”. Parnas is clear that he opposes the initiative on technical rather than political grounds. His objections are not against any kind of military software, but rather “on characteristics peculiar to this particular effort.” What makes the SDI software unattainable is its inherent complexity, the fact that it cannot be tested in realistic conditions prior to its actual use as well as the fact that there won’t be time for human intervention when the system responds to an attack.

The reason that testing and gradual development is the only way to build reliable software is the inherent complexity of large software systems. In his analysis of “the fundamental technological differences between software engineering and other areas of engineering”, Parnas, like Weizenbaum some 15 years earlier, employs logical reasoning for talking about informal concepts. He explains the reasons for software complexity with reference to analog and digital engineering products.

Analog systems have an infinite number of states within a broad operating range and can be modelled using continuous functions. “Small changes in inputs will always cause small changes in outputs” and so analog systems “contain no hidden surpirses.” It suffices to ensure, through well-understood mathematical analysis, that “the system components are always operating within their normal operational range.” In contrast, digital systems have a finite number of states that cannot be modelled as continous functions. Before the advent of computers, the number of states of such sytems was typically small enough that exhaustive testing was possible. This is no longer the case for digital computers, which have a very large number of states. To obtain correct and reliable hardware, engineers were able to leverage the fact that hardware is built using many copies of smaller digital subsystems, which can be analyzed and exhaustively tested independently. But this cannot be applied to software systems, which have orders of magnitude larger number of states and no repetitive structure. According to Parnas, “this difference clearly contributes to the relative unreliability of software systems and the apparent lack of competence of software engineers.” Most importantly, the logical reasoning used by Parnas in his analysis allows him to conclude that this “is a fundamental difference that will not disappear with improved technology.”

The Mythical Man-Month

The only commercial company that was developing software systems of the same scale as the US military was IBM, which dominated the computer market and was rapidly growing. By the end of the 1950s, IBM had a complex family of products, including multiple ranges of incompatible low-end and high-end computers. In 1961, the company embarked on a new direction and started building a single family of computers that would share the same software and have interchangeable peripherals. The System/360 (Figure 4.15), as the new family was called, was announced in April 1964 at the company’s auditorium and was paralleled with press conferences in 164 US cities and 14 other countries. The announcement was followed by a flood of orders from eager customers, but “it would take all the resources of the company working 60-hour weeks, and in some cases round-the-clock shifts, for nearly two years before the System/360 would be called a success.”[108]

The development of the operating system for the new family of machines, called OS/360, was led by Fred Brooks, who joined IBM several years earlier, after completing PhD in ap-

Figure 4.15: IBM System/360

plied mathematics at Harvard. The OS/360 included two major features that made the development complicated. It supported running multiple programs at once and the ability to handle multiple users through time-sharing interactive use. It also came with 44 new peripherals that all required software support. As the system became delayed, IBM reallocated some of its programmers working elsewhere to join the OS/360 effort. The team working on the system eventually consisted of over 1000 people and the cost of the development, grew from $40 million to $500 million. When Fred Brooks left the company in 1964, the CEO asked him why managing software projects was harder than managing hardware projects. This planted a seed for a series of essays, published as now a classic book "The Mythical Man-Month"[109] a couple of years later.

Brooks's reflection on the development of the OS/360 includes a number of principles that have became well-known slogans in the software engineering community. The most famous one is known as *Brooks's law* and is based on his experience with adding programmers to the OS/360 development effort in order to make up for the slips in schedule. The law states that "adding manpower to a late software project makes it later." The law that was certainly counter-intuitive to Brooks and others at IBM during the development of OS/360 is not just an aphorism based on personal experience. As with the principles I talked about in the previous section, it is supported by a logical argument involving the informal notion of communication.

The reason for the law is that developing a software system in a team requires communication. Most programming tasks cannot be divided into a number of completely independent smaller parts and so the developers will spend some of their time coordinating their work with others. Brooks expresses this point formally when he explains:

> *If each part of the task must be separately coordinated with each other part, the effort increases as $n(n-1)/2$. Three workers require three times as much pairwise intercommunication as two; four require six times as much as two. If, moreover, there need to be conferences among three, four, etc., workers to re-*

> *solve things jointly, matters get worse yet. The added effort of communicating may fully counteract the division of the original task (...).*[110]

In other words, as the team gets bigger, the communication overhead grows faster than the added programming capabilities and the overhead eventually outweights the gains. The mathematical argument is clear enough even if we do not know the specific numbers, such as the percentage of time spent on communication.

Fred Brooks' essays are not merely about misconceptions to be avoided. He also documents a number of developments in software engineering that he believes can have a positive effect. He endorses the Surgical Team structure developed by an IBM colleague Harlan Mills, he argues for a rigorous practice in writing documentation, but he also welcomes new technological development such as high-level programming languages, on-line debugging tools and electronic word processing tools (for maintaining a shared up-to-date documentation).

Brooks' analysis of software engineering did not stop with OS/360 and Brooks's law. In 1986, he wrote a new essay that grew out of his later experience reviewing military software. The essay, titled "No Silver Bullet: Essence and Accidents of Software Engineering" that was reprinted a year later in the IEEE Computer magazine. In the essay, Brooks makes a bold prediction:

> *There is no single development, in either technology or management technique, which by itself promises even one order-of-magnitude improvement within a decade in productivity, in reliability, in simplicity.*[111]

Again, this is not a baseless claim and Brooks supports it with a logical argument. He argues that software construction involves *essential tasks* and *accidental tasks*. The former deal with the "complex conceptual structures that compose the abstract software entity", while the latter are concerned with "the representation of these abstract entities in programming languages and the mapping of these onto machine languages within space and speed constraints." Essential complexity is caused by the problem that the software solves. A software for filling out tax returns in 10 different countries has to do 10 different things. In contrast, accidental complexity is caused by the imperfection of our programming tools.

According to Brooks, the hard part of producing a software system is dealing with the essential tasks, that is the specification, design and testing of the abstract software entity. The accidental complexity has been largely simplified by the development of high-level programming languages, although it is still something that programmers need to deal with. The crux of Brooks' argument is that, unless dealing with the accidental complexity "is more than 9/10 of all effort, shrinking all the accidental activities to zero time will not give an order of magnitude improvement." Again, the exact number does not matter, as long as we believe that it is less than 9/10. If the ratio between essential and accidental complexity was 1:1, reducing the accidental complexity to 0 would only lead to a two-fold improvement.

In "No Silver Bullet", Brooks also elaborates on the two sources of the essential complexity that we already encountered in the discussion about anti-ballistic missile defence, that is the inherent complexity of large digital systems and the complexity of an evolving environment. In the discussion of the former, Brooks shares an argument with Dave Parnas who was, for a time, his colleague at The University of North Carolina at Chapel Hill. Software entities are, according to Brooks, "more complex for their size than perhaps

any other human construct." They are composed of a large number of distinct, discrete elements and, as software gets larger, the situation only gets worse:

> *[S]caling-up of a software entity is not merely a repetition of the same elements in larger size, it is necessarily an increase in the number of different elements. In most cases, the elements interact with each other in some non-linear fashion, and the complexity of the whole increases much more than linearly.*[112]

The second source of complexity of software systems is that they are "embedded in a cultural matrix of applications, users, laws, and machine vehicles. These all change continually, and their changes inexorably force change upon the software product." This characteristics of the environment in which software systems operate has been the primary concern of another influential contribution to software engineering from the 1980s.

Laws of Software Evolution

Like many of those developing the engineering perspective on programming, Meir "Manny" Lehman had both academic and industrial experience. After studying at Imperial College London, he worked at Ferranti, Israel's Ministry of Defense and IBM's research division before returning to the Imperial College. In 1968, Dr. Arthur Andersen who was the Director of Research from Lehman's group at IBM came accross a paper from Bell Labs claiming that a move to interactive programming improved programming productivity three-fold. Andersen came to Lehman exclaiming that "Anything that Bell Labs can do, IBM can do better!" and asked him to investigate what projects could improve productivity of IBM programmers.[113] The question changed Lehman's career trajectory for the next 25 years.

The immediate result of this task was a 1969 report "The Programming Process",[114] which reviewed individual technical advances like interactive programming and high-level programming languages, but concluded that the "solution of the programming problem is seen in the development of a total system-oriented methodology and the associated support technologies and tools." The focus on methodology assisted by tools positions Lehman's approach firmly in the engineering culture of programming. So do the references in his report, which include works of Dave Parnas and the 1968 NATO Software Engineering conference.

Shortly thereafter, Lehman started looking at the evolution of the OS/360 operating system, which provoked his interest in the dynamics of software growth and the laws that govern it. Together with a colleague, Les Belady, he started using the term *software evolution*. After moving back to Imperial, Lehman continued studying software evolution by analysing large real-world software solutions and eventually formulated eight laws of software evolution that were published in an influential paper "Programs, life cycles, and laws of software evolution".[115]

Lehman opens his analysis by pointing out that program evolution has been always associated with the concept of large programs, but that characterizing "large" and "non-large" programs has been challenging. To address this, Lehman introduces an alterntive and more insightful classification of programs, distinguishing between S, P and E-programs. The types of programs are distinguished not based on their size, but based on their relationship to the environment and the degree to which they can be fully specified. An S-program (*specification*) is an implementation of an algorithm. The problem it solves does

not change and the solution can be fully described by a specification. A P-program (*problem*) solve a problem that can be precisely defined, but the quality of the solution is dependent on the environment. Lehman uses a chess playing program as an example. Chess has a clear set of rules, but the solution will involve complex hard-to-specify algorithm and the program quality will depend on the end-user of such program. Finally, E-programs (*environment*) do not even have a fixed problem specification. They "mechanize a human or societal activity," but in doing so, they are embedded in the environment and change it, often also changing the problem statement. Examples given by Lehman include operating systems, air-traffic control and stock control.

The first consequence of the classification, according to Lehman, is on the notion of program correctness. Whereas an S-program can be fully formally specified and its implementation checked against the specification, the validity of an E-program "depends on human assessment of its effectiveness in the intended application." Although Lehman somewhat optimistically suggests "that as programming methodology evolves still further, all large programs (software systems) will be constructed as structures of S-programs," his laws talk primarily about the most challenging kind of programs, the E-programs.

The focus on E-programs, which is typical for many of the discussions followed in this chapter, is in a stark contrast with the typical focus of the mathematical culture of programming, which often assumes that programs can be fully specified and, by doing so, restricts itself to talking about what Lehman classifies as S-programs. An eloquent advocate of the mathematical culture, Edsger Dijkstra, describes his focus clearly in a position paper on software reliability[116] by contrasting the "scientific questions of correctness" with "the unformalized question whether a tool (...) is in such-and-such unformalized and ill-understood environment a pleasant tool to use." The narrow focus of mathematical methods is a frequent source of clashes between the mathematical and the engineering culture. For example, Peter Naur responded to Dijkstra by pointing out that pleasantness is "a somewhat euphemistic word to use about such calamities as airplanes colliding in mid air or atomic reactors exploding."[117]

Lehman's realization that programs need to evolve in parallel with the continually evolving environment in which they are embedded poses a major challenge for top-down software development methodologies that analyze the requirements at the beginning of the process, but ignores the ongoing evolution. In other words, the advances in understanding of software development in the engineering culture of programming outpaced the understanding in the managerial culture, which was busy standardizing the steps of a top-down development process. By the end of the 1980s, engineers started using their understanding to rethink software development processes.

Paradigm Change in Software Engineering

Already in 1970, when Royce wrote his critique of the simple Waterfall software development model, he was aware of issues with a process that proceeds forwards in steps, from analysis to specification, implementation, testing and deployment. Royce pointed out that more iteration and feedback is needed in the process. He suggested a "do it twice" method in which a pilot project is completed to understand the problem before building the real system. This approach was also later advocated by Fred Brooks in his "Plan to throw one away" essay.[118] Another suggestion by Royce was to involve the cutomer not just in the initial requirements gathering stages, but repeatedly throughout the development process.

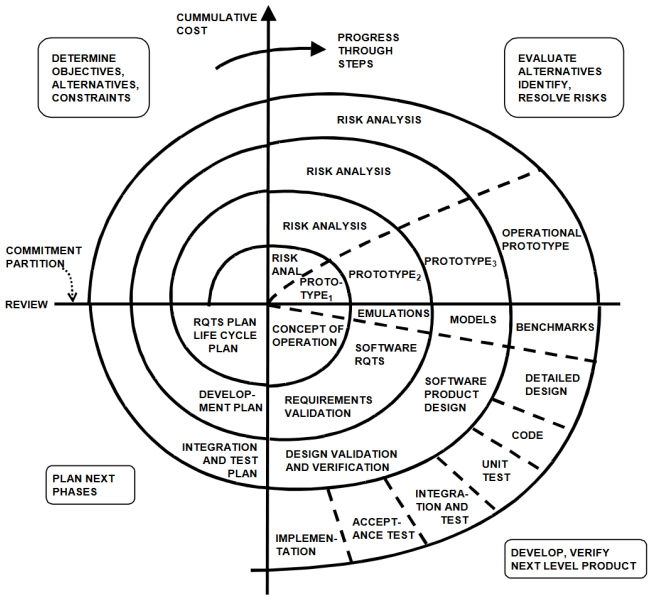


Figure 4.16: A diagram illustrating the Spiral development model from Boehm (1988)

The chief motivation for developing such processes is that fully understanding the problem upfront is almost never possible, a point that was also made by Dave Parnas when discussing the limits of software engineering methods in the context of anti-ballistic missile software. Parnas pointed out that “it is hard to make the decisions that must be made to write [an upfront project specification]. We often do not know how to make those decisions until we can play with the system.”[119]

Throughout the 1980s, a variety of innovations in software development have been proposed to to address the issue. Newly created development models included *prototyping*, where some part of the cycle is done multiple times to allow learning, as well as *incremental* models, where the system is broken down into smaller parts that are developed gradually. A development model that brought many of the preceding ideas together was developed by Barry Boehm. As is typical for the key actors in this chapter, Boehm had academic background, but spent most of his career in industry. In 1988, he developed the *spiral development model* that is illustrated in Figure 4.16. Rather than proposing a single methodology, the model proposes a way of determining the most appropriate process based on the analysis of risks at each stage. Boehm later described the model as “process model generator”, highlighting the fact that the resulting process can dynamically take a number of different shapes.[120]

Despite the numerous innovations, the work on software development models never fully separated itself from the managerial idea of control over the the development process and the programmers involved. In other words, even the more flexible and adaptive development models still aimed to transform the “black art of programming” into a rigorous discipline that is reproducible, perdictable and does not rely on the skills of individual programmers. They do so by wrapping the inherent uncertainty in the programming process with enough additional layers to make software development more deterministic.

A more radical vision of how software should be built came from Christiane Floyd. After completing a PhD at University of Vienna, she worked in several academic and industry

jobs, before becoming the first female professor of computer science in Germany in 1978. Since joining the Technical University of Berlin in 1978, Floyd and her group have been working on a new approach to software development "which is primarily concerned with the development of software as an adequate tool for users."[121] The approach transcends "the conceptions of software engineering prevalent in the [Germany]. These see it as being based on fixed requirements of the software products to be developed, ignoring the social context in which they are embedded."[122]

In pursuing this goal, Floyd closely collaborated with Scandinavian colleagues. Following two field trips to Denmark, Norway and Sweden in 1986, she published a trip report that documents the "Scandinavian Approach"[123] to software development, which makes the user a more integrated part of the process. In Scandinavia, such approach emerged in the 1970s under the term "participatory design". While in Scandinavia, work on participatory design led to the birth of an active research field on human-computer interaction, Floyd used the participatory design approach as a basis for software development methodology that she named STEPS (Software Technology for Evolutionary Participative System Development). Perhaps the most visionary work arising from Floyd's approach appears in a chapter "Outline of a Paradigm Change in Software Engineering",[124] published as part of proceedings of a conference held in Aarhus titled "Computers and Democracy". In the chapter, Floyd describes the dominant paradigm that emerged following the 1968 NATO conference as *product-oriented* and summarizes the *process-oriented* alternative.

According to Floyd, "the product-oriented perspective regards software as a product". It "considers the usage context of the product to be fixed and well understood, thus allowing software requirements to be determined in advance." In contrast, the process-oriented perspective "views software in connection with human learning, work and communication, taking place in an evolving world with changing needs". This approach clearly recognizes the changing environment in which software development happens and the difficulties with producing upfront specifications that were gradually being recognized by many researchers and practitioners of software engineering. In the process-oriented approach, the final product "is perceived as emerging from the totality of interleaved processes of analysis, design, implementation, evaluation and feedback, carried out by different groups of people involved in system development in various roles." The process-oriented approach admits that requirements change and understanding is built gradually.

In the product-oriented perspective, process aspects such as requirements definition, quality assurance, user acceptance and software modifiability are considered "as additional aspects outside the realm of systematic treatment." Floyd holds that "this situation is inherently unsatisfactory and can only be remedied if we adopt (...) the richer process-oriented perspective as our primary point of view."

The perspective from which Floyd approaches software engineering cannot be easily classified as belonging to any of the individual cultures that appear repeatedly in this book. Like other development methodologies, STEPS can be seen as emerging from a managerial attempt to structure the development process. However, the process-oriented methodology also makes individual engineers more autonomous as they are involved in learning about the problem and shaping the final product. Finally, Floyd's work also emphasizes human learning and the Scandinavian participatory design, which connects it to the humanistic culture that we followed in the previous chapter.

When Christiane Floyd wrote about the paradigm change in software engineering, she talked about an "ongoing controversy between rivalling ideas and attitudes". She clearly

saw a glimpse of an emerging new approach to software engineering, but it took 15 more years before the new paradigm took a form that allowed it to dominate the field of software engineering.

The New New Approach

At the same time as Christiane Floyd was thinking about the Scandinavian approach to software development and about the ongoing paradigm change, a similar idea emerged in the US about product development more generally. Hirotaka Takeuchi and Ikujiro Nonaka had management and operational theory backgrounds and gained experience in both industry and academia in US and Japan. In 1986, they published an article "The New New Product Development Game"[125] in the Harvard Business Review journal, in which they analyzed a "new approach to managing the product development process".

Takeuchi and Nonaka looked at six specific products including a Honda city car and two Canon cameras and reported that companies are "increasingly realizing that the old, sequential approach to developing new products simply won't get the job done" in a fast-paced, competitive world where speed and flexibility are essential. They liken the new approach of the pioneering companies to rugby "where a team tries to go the distance as a unit, passing the ball back and forth." Using the new approach:

> *The product development process emerges from the constant interaction of a hand-picked, multidisciplinary team whose members work together from start to finish. Rather than moving in defined, highly structured stages, the process is born out of the team members' interplay.*

In contrast to the humanistic Scandinavian approach focusing on shared knowledge building, the wording here is more managerial and focuses on teams and initiative. Yet, the ultimate vision of collaborative and more dynamic approach is very closely related. The authors talk about a new product development organization, but the organization is nothing like that of the managerial structures of the 1970s. In the new approach, teams have higher degree of autonomy, learning and knowledge is exchanged between team members and the process is controlled by a more subtle indirect mechanisms.

Takeuchi and Nonaka talked about product development more generally, but in the early 1990s, the ideas about the "new approach to product development" served as a motivation for the next step in the evolution of software development methodologies. The SCRUM methodology, developed by Ken Schwaber and Jeff Sutherland, was first presented at a workshop on Business Object Design and Implementation,[126] associated with the ACM OOPSLA conference. The authors recognize that sofware development occurs under rapidly changing circumstances and that the development requires maximum flexibility. They contrast SCRUM to other development methodologies, notably Waterfall, Iterative and the Spiral models. In SCRUM, the development is done in short "sprints" which are always followed by review that makes it possible to adapt more rapidly to the lessons learned during the development.

SCRUM remains rooted in the managerial approach to programming. It describes how teams and the development process should be organized, rather than, for example, recommending particular tools. It does, however, reflect a shift in the managerial culture. Whereas the managerial approach to software development in the 1970s attempted to

eliminate dependence on individual programmers and their skills, the new methods that emerged in the 1990s see programmers as creative and responsible professionals that should be trusted with making technological decisions and empowered to learn and contribute to the product development.[127]

Extreme Programming

The need for a new development methodology was in the air at the end of the 1980s and new ideas have emerged from a variety of programming cultures. The Scandinavian process-oriented approach adopted many humanistic ideals, the Scrum methodology followed a managerial approach, but new ideas were also materializing in the engineering culture of programming. The resulting development method, which emerged in the 1980s and gained prominence in the 1990s is Extreme Programming (XP). As with the work of Chrisitane Floyd and the Scrum methodology, XP tackled the typical issues of its time, that is the development of software in an environment where requirements are vague and rapidly changing. What makes XP interesting for my story is not the various ways in which it is similar to and differs from Scrum, but the fact that it has quite different origins.

Extreme Programming was not developed by academics or management consultants, but by software engineers. The methodology was developed and later publicized by Kent Beck,[128] who nevertheless credits many ideas to "common knowledge" and also his collaborator Ward Cunningham. Both Beck and Cunningham were consultants, working on commercial object-oriented systems developed using the Smalltalk programming language. They were looking for better ways of building systems and explored a range of ideas, such as building on the concept of pattern languages conceived by the architect Christopher Alexander that I return to in Chapter 6. They were also a part of the Smalltalk community, which had a long tradition of empowering individuals and allowing them to shape their tools. All of these aspects are mirrored in the XP methodology, which is based on decentralized decision making, rapid feedback and adaptation as well as focus on tools and best programming practices. The clear affiliation with the engineering culture of programming is perhaps best summarized in Erich Gamma's forweword of Beck's book, which opens by saying that "Extreme Programming nominates coding as the key activity throughout a software project. This can't possibly work!"[129]

The engineering origins of the Extreme Programming methodology shift the attention from managerial issues to engineering, but it still talks about a number of management aspects. In particular, XP identifies a number of different roles in the team. The two most prominent ones are a programmer and a customer, who is now a direct part of the team, in accordance with what Christiane Floyed envisioned. The role of the customer is a challenging one. The customer decides what the software should do, specifies the tests for the required behaviour, while doing their regular job. In the first widely discussed project completed using the XP methodology, the Chrysler Comprehensive Compensation System (C3) payroll system, the customer was able to do all this work, but eventually found the work too stressful and moved on to a different position. The team soon realised that finding another "XP customer" was more challenging than expected, which resulted in an ominous presentation "Will Extreme Programming kill your customer?" presented at the 2001 OOPSLA conference.

Extreme Programming shows its engineering origins in that it describes a number of specific programming practices. Many of those are not new. As explained by Kent Beck,

“To some folks, XP seems like just good common sense. So why the "extreme" in the name? XP takes commonsense principles and practices to extreme levels.”[130] Beck lists 12 practices covering areas such as project planning, customer involvement and testing. For example, “pair programming” requires that all code is written by two programmers at one machine. The practice should not be used, for example, only when solving a particularly tricky problem, but all the time. “Testing” requires continually written unit tests created by programmers and customer-created tests demonstrating that individual features are completed. In contrast to earlier methodologies, testing in XP becomes the driving force behind development. Another principle is that of “refactoring”, which asks programmers to continually restructure code in order to improve its strucutre, simplify it and ensure it communicates the intent better.

Extreme Programming brings back to prominence a practice of software testing, which keeps appearing, disappearing and evolving throughout this chapter. When discussing the professionalization of testing, I pointed out that testing became clearly distinguished from debugging in 1970s. Companies started hiring “test technicians” to work on automated software testing tools and, with the start of the destruction-oriented period, testing started to focus specifically on finding errors in programs. In most of the software development processes that I talked about so far, testing exists as a separate phase to be done at a specific point during the development. Sometimes, the process also draws an explicit distinction between unit testing, which focuses on individual units of functionality, integration testing that focuses on the overall system and acceptance testing to show that the product matches requirements specified by the customer.

The idea of Extreme Programming was to take existing engineering practices to the extreme. In case of testing, the most extreme version of the idea became known as Test-Driven Development (TDD). Here, automated test for a required functionality need to be created even before the programmer writes any code to actually implement the functionality. The idea was not new. In fact, Gelperin and Hetzel’s consulting company Software Quality Engineering promoted the motto “Test, then code” as early as 1986, and, as I mentioned earlier, even used the motto for lapel pins at a software testing conference in 1987.

Test-Driven Development, which eventually evolved into an independent software development methodology,[131] fully completes the shift from the managerial culture of programming to the engineering culture. It turns tests from something that can be done manually to a software artifact that has to be automated. Tests are not used just for ensuring that a certain functionality is implemented correctly, but they serve as a specification of the system. In TDD, programmers start by writing tests for a minimal system that could be possibly run and then gradually add tests that describe further functionality, ensure that the tests pass by implementing the necessary logic and then proceed to make the code simpler and more reusable before moving on to the next feature. The tests can also be seen as capturing the understanding of the problem, jointly developed by programmers and the customer, and can even be treated as a partial specification, an idea that the proponents of the mathematical culture would find absolutely inadequate.

TDD shows its connection to the engineering culture in that it is structured around tools, automated test runners. At the same time, it makes another attempt at fulfilling the vision outlined at the NATO 1968 Conference on Software Engineering, which was to turn the black art of programming into a science of software engineering. Already in 1968, A. I. Llewelyn, working for the short-lived UK Ministry of Technology, pointed out that “testing is one of the foundations of all scientific enterprise.”[132] Test-Driven Development devel-

ops the parallel between programming and science in that it outlines a sytematic software development process based on testing. But just like actual scientific work is more complicated than its naive image, so is software development even when following the TDD methodology.

The Debugging Paradox

In 1965, Mark Halpern concluded his article "The Debugging Epoch Opens", which I referred to at the start of this chapter, by caliming that what "is certain is that the debugging problem must be confronted squarely and soon if the computer is to take on some of the critical roles it is presently being cast for." He warned that that "the danger is real and imminent" and argued that the "remedy will require the commitment of the programming community's full resources."

In many ways, Halpern was correct. Eliminating program bugs and, more generally, making sure that programs perform correctly has become a major effort over the next several decades and remains as important issue as ever. In one way, however, his prediction was mistaken. The interesting developments that we followed in this chapter focused on testing and software development processes, but very little on *debugging*. This may be a historical accident, because the meaning of phrases like program checkout, testing and debugging kept shifting and evolving throughout the 1960s and the 1970s. Yet, when Halpern wrote his article, debugging was already increasingly associated with "making sure that programs run" whereas testing referred to "making sure that it solves the problem".[133]

So, why didn't I have more to say about debugging? The reason is that the basic practices of debugging today are not unlike that discussed in a review of on-line debugging techniques by Evans and Darley in 1966.[134] Just like when debugging programs using a LISP system in 1966, programmers today set breakpoints to pause program execution, explore the values of variables, modify the program and then test the result. Unlike in 1966, this works reliably for high-level programming languages, which is a notable technical achievement, but there have been few conceptual innovations. It is difficult to say what the reasons for this is with any certainty, but thinking about the interactions between the different cultures of programming in this chapter offers one possible answer.

In the 1950s, programmers talked about debugging, testing, program checkout and error handling somewhat interchangeably. This was the era of black art of programming and dealing with errors was a "private arcane matter".[135] As testing became distinguished from debugging, the term "test" slowly started to be used not just as a verb, but also as a noun. A "test" first became a description of a check to be performed manually and later automatically. This, however, also made a test into an artifact that could be shared with programmers across cultures. In other words, a test became a boundary object.[136] This may have made it easier for other cultures of programming to contribute to work on testing. Managers could make running of tests a part of their processes, mathematicians could study tests formally and engineers could build new testing tools. Eventually, Test-Driven Development made an executable test into a primary mechanism for controlling the software development process.

The case of error handling shares similar characteristics with that of testing. In the early days of programming, the handling of program errors was fully within the realm of the hacker culture. In the 1960s, LISP and PL/I added a programming language construct for working with errors; in the 1970s, the exception handling mechanism appeared and

was adopted by Ada and many programming languages that followed. Here again, the vague programming practice results in an artifact, or a boundary object, that can be shared across cultures. Engineers started implementing programming languages with exceptions, mathematicians were able to formally analyze programs using them, as well as to propose new ways of working with exceptions.

Like tests eventually gave rise to a new development methodology, exceptions gave rise to a new programming style in the 1980s. While working on the development of fault-tolerant systems for telephone exchange, Joe Armstrong at Ericsson came up with a new way of handling exceptions for their programming language Erlang. In telecommunications, systems are designed to "run forever" and must accept that hardware failures will happen. As such, other computers must respond to errors. To support this, Erlang programs are structured as sets of lightweight processes, which may execute on separate machines. When an exception occurs, the affected process is killed. The crash will then be handled by a dedicated supervisor process that may restart the process or activate another system. In Erlang, exceptions became a key building block for building reliable fault-tolerant systems.[137]

In contrast to testing and exception handling, the debugging remains a hacker practice with no simple boundary objects that could be exchanged with other cultures. As is characteristic of the hacker culture of programming, the knowledge is often unwritten and it remains the case that "much of the work [on debugging] has been described only in unpublished reports or passed on through the oral tradition."[138] Even the internal memoranda documenting debugging tools often merely provide a reference manual for the tools, rather than documenting how they should be used.[139] Building a sophisticated modern debugging tools is an engineering challenge and mathematical work on program analysis contributed valuable ideas for debuggers, but the process of debugging is still a private arcane matter.

Different Ways of Trusting Software Systems

Some of the most interesting developments discussed in this chapter happened when different cultures interact, often by exchanging and contributing to a specific technical concept. Each culture brings its own methods and ways of thinking about programming. While the concept of debugging remained within the realm of the hacker culture, concepts such as testing were broadly adopted and developed. Managers turned a test into a part of a software development process, mathematicians studied the formal guarantees that a set of tests can provide and engineers contributed to testing by developing new tools and even used tests as a basis for a systematic software development methodology. Different cultures of programming are also apparent in the way programmers think about the limits of what software can be built and what they see as sufficient criteria for trusting that a system is correct.

As we saw in Chapter 2, for the mathematical culture of programming, a formal correctness proof is the ultimate goal. A fully trustworthy software system should come with a proof showing that the implementation corresponds to a formal spcification. The trust in proofs may be established through a social process or through mechanical checking. In reality, proving an entire system correct is often unachievable. In practice, the proponents of the mathematical culture will also, even if uneasily, trust systems where the formal proof concerns a simplified model or only a core part of the system.

The mathematical culture also recognizes mathematical laws as the only limitation of what software can be built. To put it simply, it is impossible to build a system that solves the halting problem, but anything else is possible. The proponents of the mathematical culture believe that there is no fundamental conceptual difference between a compiler and a software to detect and target balistic missiles. Any software system can be implemented, provided that we are given a sufficiently precise specification. The question whether such specification can exist is out of scope for computer science. To make this point explicit, Edsger Dijkstra distinguishes between the problem of correctness (whether a system meets a specification) and of pleasantness (whether the system is one we wanted to have).[140]

The managerial and engineering cultures are more concerned with practical constraints of building complex software systems that lack clear specifications and are built in an evolving environment. The engineering disciplinary repertoire developed to talk about the limits of software development concerning such systems. This made it possible to claim that certain systems are impossible to build, for example when the change rate of the environment is higher than the change rate at which the system can be built or when progressive development with continuous testing is impossible.

What exactly is required by the managerial and engineering cultures in order to trust a system has been evolving over time. In the managerial cutlure of the 1970s, a system can be trusted if it was built by correctly completing all the required steps of the selected software development methodology including, most notably, acceptance testing by the customer. The trust is thus mediated through a process agreed ahead of time. In the case of the engineering culture, the trust in software system relies on the responsibility of individual programmers, but it can be supported by tools. For example, in Test-Driven Development, a tool can measure code coverage, which is a metric of what proportion of code has been executed during testing. While some engineers will emphasize that this is a simplisitic metric, many are more likely to trust systems with higher code coverage.

Finally, in the hacker culture of programming, the trust is mainly placed in individuals. In the 1950s, those were the masters of "black art" of computer programming, but a similar level of trust still exists in highly technical, especially open-source, programming projects such as the Linux kernel or in the development of programming languages. For example, some open-source communities award the Benevolent Dictator for Life (BDFL) title to the project leader who is trusted with making ultimate decisions about the project.

Donald MacKenzie discusses the issue of trust in the context of mechanized proof by referring to historical and sociological research on risk and trust in the societies of high modernity. He summarizes that "in traditional communities face-to-face interactions typically obviated the need for formal structures of objectivity." In a society of a high modernity which, among other things, relies on increasingly complex computer systems, "face-to-face solutions are no longer viable. Modernity's 'trust in numbers' is a substitute for absent trust in people."[141]

In the case of software development, the question of trust does not follow a simple progression from traditional communities to a society of high modernity. The hacker culture still relies on the equivalent of face-to-face interactions, whereas the mathematical culture is fully commited to the modernist trust in numbers. The managerial and engineering cultures are somewhere in the middle of the spectrum. Although individual responsibility is important in both engineering and managerial culture, the two cultures rely on processes and tools, respectively, as crucial components for trust.

Notes

1. As discussed below, this question was posed by Hoare (1996) with reference to an analysis by historian Donald MacKenzie. I quote numbers from a more recent publication MacKenzie (2004).
2. Based on the answer suggested by Hoare (1996)
3. *Tau*'s answer follows that given by Hoare (1996).
4. One of those who appreciated this aspect was Vannevar Bush, discussed by Waldrop (2001), who we encountered in Chapter 3.
5. Inspired by Halpern (1965), who predicted that the 1960s and the 1970s will be the epoch of debugging
6. *Tau* is paraphrasing Dijkstra, as quoted in the proceedings of the NATO 1969 conference edited by Buxton et al. (1970)
7. This part of the dialogue follows the account written by Gelperin and Hetzel (1988)
8. Goodenough and Gerhart (1975),Musa (1975)
9. This recent way of thinking about testing has been documented, for example, by Mugridge (2003)
10. This is even more the case in a related technique known as Behavior-Driven Development. See Solis and Wang (2011)
11. Royce (1970)
12. According to Ensmenger (2012), the report by McKinsey (1968) was widely cited in business and technical literature soon after its publication.
13. Floyd (1987)
14. The list is based primarily on works by Lehman (1980), Brooks Jr (1975) and Parnas (1985) that will be discussed later in the chapter.
15. Brooks Jr (1975)
16. Brooks Jr (1975)
17. As pointed out by Licklider (1969), lessons learned in the 1950s and the 1960s have resulted in "common wisdom". His work was one of the first attempts to make such knowledge more explicit.
18. Documented in detail by Slayton (2013).
19. Hoare (1996)
20. for (1952), quoted in Bissell (2007)
21. Bissell (2007)
22. To use the term used by von Neumann and Goldstine (1947). The historical significance of the problem of controlling the evolution of meaning is discussed in detail in a chapter on notations by PROGRAMme (2022)
23. for (1952)
24. Bartik (2013)
25. Haigh et al. (2016)
26. Pirsig (1999)
27. Bartik (2013)
28. Wilkes (1985)
29. Documented by Priestley (2011).
30. Campbell-Kelly (1992)
31. Campbell-Kelly (2011)
32. Initial orders and sub-routines have been described in a textbook by Wilkes et al. (1951). A useful historical summary can be found in Campbell-Kelly (2011).
33. Campbell-Kelly (1992)
34. Both presented in Wilkes et al. (1951) and documented by Campbell-Kelly (2011).
35. Kidwell (1998)
36. Turing (1949)
37. Jr. (1958)
38. As is typical for the hacker culture of programming, the system was only ever described in an internal memo, Stockham and Dennis (1960), written for the benefit of the MIT hacker community.
39. See Evans and Darley (1966) for the review and Evans and Darley (1965) for the DEBUG tool
40. As usual, the systems have been described in various internal memos including ones by Martin and Hart (1964), Teitelman (1965) at MIT and one by Russell (1963) from Stanford.
41. Bobrow et al. (1967)
42. Halpern (1965)
43. This view is documented by Ensmenger (2012).
44. The history of the conference has been documented by Pelaez Valdez (1988).

45. The extent to which the NATO conferences contributed to the rise of the engineering culture is open to discussion. Ensmenger (2012) talks about “major cultural shift”, while Thomas Haigh (2010) argues that the impact is hard to “square with the actual historical record”.
46. Llewelyn and Wickens (1968)
47. Buxton et al. (1970)
48. The context in which the PL/I language was developed has been documented by Astarte (2019).
49. Radin (1978)
50. McCarthy et al. (1962)
51. For a detailed account of the history of PL/I and its formalizations, see Astarte (2019).
52. Ryder et al. (2005)
53. Goodenough (1975a,b)
54. Liskov (1993)
55. As recalled in the history of the Ada project written by Whitaker (1993).
56. Ichbiah et al. (1979)
57. Steele and Gabriel (1996)
58. Goodenough (1975a)
59. Liskov (1993)
60. The idea has been introduced by Plotkin and Pretnar (2009) and a more accessible review of the feature can be found in a report by Chandrasekaran et al. (2018).
61. The modern term for the specific practice is *offensive programming* and seems to have been introduced in the late 1990s, but the term defensive programming dates back to a 1975 book by Yourdon (1975)
62. Randell (1975)
63. A notable example of this is the system designed by Yau and Cheung (1975).
64. Most notably, in the Erlang programming langauge that appeared in the 1980s.
65. Quoted in MacKenzie (2004)
66. Hetzel (1973)
67. McGonagle (1972)
68. McGonagle (1972)
69. Gelperin and Hetzel (1988)
70. Buxton et al. (1970)
71. The three examples are referenced by MacKenzie (2004), p. 42.
72. Huang (1975)
73. Goodenough and Gerhart (1975)
74. McCabe (1976)
75. Myers et al. (1979)
76. A demo, presented at the AFIPS conference by Budd et al. (1978) and a paper by DeMillo et al. (1978)
77. Ensmenger (2012), p.148
78. Slayton (2013), p.109
79. Brooks Jr (1975)
80. Thomas Haigh (2010)
81. McKinsey (1968)
82. The shift has been documented in detail in the work of Ensmenger (2012)
83. Introduced in “The Psychology of Computer Programming” by Winberg (1971)
84. See Slayton (2013), who documents not just the development of the SAGE, but the entire socio-technical context of missile defence software.
85. Benington (1983)
86. Royce (1970)
87. Brooks Jr (1975)
88. IEEE Standard for Software Test Documentation, ANSI/IEEE (1983)
89. IEEE Standard for Software Verification and Validation Plans, ANSI/IEEE (1986)
90. IEEE Standard for Software Unit Testing, ANSI/IEEE (1987)
91. National Bureau of Standards. (1984)
92. Gelperin and Hetzel (1988)
93. Hetzel (1988)
94. This and many other events in the history of testing are documented in the excellent online reference by Meerts and Graham (2010).
95. Slayton (2013)
96. De Millo et al. (1979)

97. ibid.
98. ibid.
99. Wonderfully documented by Slayton (2013)
100. Licklider (1969)
101. Licklider (1969)
102. Available at `https://digitalcollections.library.cmu.edu/node/16631`, Accessed 21 September 2022
103. Quoted in Slayton (2013)
104. Quoted in Slayton (2013)
105. Slayton (2013)
106. **?**
107. Parnas (1985)
108. As reported in the Boyer (2004)
109. Brooks Jr (1975)
110. Brooks Jr (1975)
111. Brooks (1987)
112. Brooks (1987)
113. As recealled by Lehman in an oral interview conducted by William Aspray, Lehman (1993)
114. Later republished as part of a joint book, Lehman and Belady (1985)
115. Lehman (1980)
116. Dijkstra (1977)
117. Naur (1993)
118. Brooks Jr (1975)
119. Parnas (1985)
120. Boehm (1988)
121. Floyd et al. (1989)
122. ibid.
123. Floyd et al. (1989)
124. Floyd (1987)
125. Takeuchi and Nonaka (1986)
126. Schwaber (1997)
127. As noted by Fowler (2001), the shift did not happen solely in software development, but is a more general move in product design and manufacturing.
128. The publication that popularized the idea is a book by Beck (2000)
129. Beck (2000)
130. Beck (2000)
131. Outlined by Beck (2003)
132. Naur et al. (1969)
133. Gelperin and Hetzel (1988)
134. Evans and Darley (1966)
135. backus-1980-programming
136. Using the terminology introduced by Star and Griesemer (1989).
137. Armstrong (2007)
138. Evans and Darley (1966)
139. Using terminology of Polanyi (1958), debugging is personal knowledge.
140. Dijkstra (1993)
141. MacKenzie (2004)

Chapter 5

Programming with Types

Teacher: Types are a good follow-up after a chapter on software engineering as types are another widely used concept that makes programming safer and more reliable. Was the history of types similar to that of the concepts we talked about the last time, such as tests and exceptions?

Tau: Types are more fundamental than tests or exceptions! They originated in the work on formal logic. They reveal a basic link between programming and logic and a proof that programming is a branch of mathematics.

Omega: Well, as far as I know, types existed in programming langauges at least since Algol. Programming language theoreticians like to use them to prove properties about programs, but I'm not sure how that makes types more fundamental than other programming concepts that we talked about.

Tau: Types existed in mathematics well before there were any computers. The earliest use of types is in the work on foundations of mathematics by Bertrand Russell, who uses types as a mechanism for avoiding paradoxes of the kind "sets of all sets that are not members of themselves". This appeared as early as 1903 and it has been adapted for the λ-calculus by Alonzo Church in 1940. Programming languages with a reasonable notion of type directly follow the schemes invented by Church.

Epsilon: This is a fascinating history, but your retelling is misleading. The notion of types that is similar to the one introduced by Church in λ-calculus, which you choose to call "reasonable", only appears in typed functional programming langauges in 1970s!

Teacher: Let's try to retrace the history then, starting from the early programming languages. Were there any types in FORTRAN?

Epsilon: If you search for the term 'type' in the 1956 reference manual for the FORTRAN automatic coding system for the IBM 704,[1] you'll find that FORTRAN has 32 types of statements, two types of constants and two types of variables, three basic types of decimal-to-binary and binary-to-decimal conversions... None of these sound like the kind of types that we are looking for.

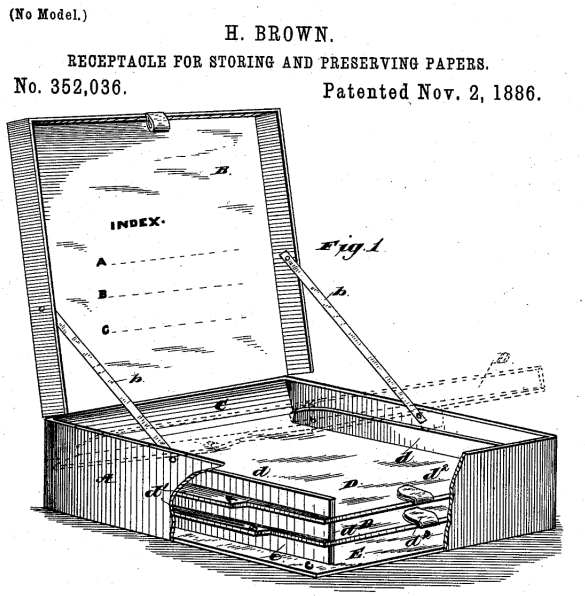


Figure 5.1: Sketch of a "receptacle for storing and preserving papers", the predecessor of the filling cabinet for storing (paper) records, patented in 1886 by Henry Brown.[2]

Alpha: I suspect that what you are looking for is the concept of *modes* of functions and their arguments. This is likely the predecessor of types. Modes were used to decide how the bits that represent numbers in memory should be interpreted. As the manual describes, arguments of a function can be in either fixed or floating point mode. Similarly the function itself may be in either of these modes. You indicate that a function is in the fixed point mode by starting its name with the letter "X" and it also has to end with "F" to indicate that it is a function. So, for example `XSIN1F` was a function taking and returning a floating-point number.

Epsilon: I can see how this is similar to different primitive types that we have in modern programming languages, but I don't think you can talk about types if you can't represent richer data types or even just textual data.

Alpha: FORTRAN provided a subscript notation for working with 1, 2, and 3-dimensional arrays of variables. A data type for strings was introduced in FORTRAN 77. Prior to that, you could represent text as numbers using Hollerith constants. In other words, I think this notion worked well enough for scientific computing at the time.

Teacher: Where should we look for the origins of richer data types, then?

Omega: If you looked outside of the scientific programming bubble, you would discover that the data processing industry already solved this problem. The programming language FLOW-MATIC, created in 1955, supported the concept of records, which was clearly inspired by the existing use of paper records. You could say that the true predecessor of types is the filling cabinet (Figure 5.1) used for storing records since the end of the 19th century!

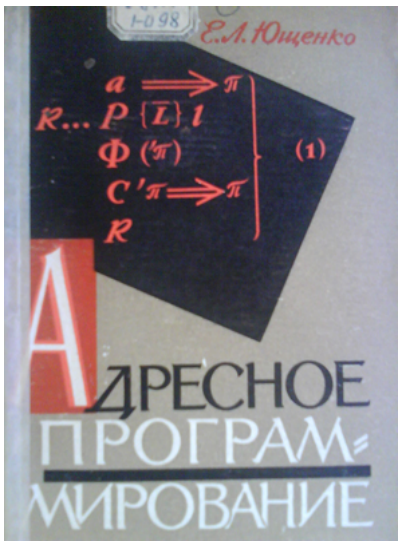


Figure 5.2: Cover of the "Address programming"[5] book, written by Ekaterina Yushchenko in 1963.

Tau: Well, yes, records in FLOW-MATIC and later COBOL were an inspiration for the later work on richer data types in theoretical computer science, but much work was needed for getting the field of business data processing on a firm theoretical footing.[3]

Epsilon: Records in COBOL look very similar to records in some of the contemporary programming languages. What kind of theoretical footing were they missing?

Tau: As I said a number of times, programming is a mathematical activity, so you need to think about the mathematical objects involved rather than bits in computer memory or paper records in a metal case! For records, the first step towards formal foundations was given by John McCarthy in 1961 when he proposed to model basic numerical data spaces as sets and then construct derived data spaces using set operations such as a product to model records.[4]

Gamma: I'm sorry, but I'm getting a bit lost. I hoped to learn what's the deal with types, but then *Alpha* talked about "modes" and now *Tau* is talking about "data spaces". This is very confusing! Are we even talking about the same thing?

Tau: I can see a relationship here. Modes are a way of specifying what the inputs and outputs of a function are. Data spaces do that too, but at a more fundamental theoretical level, using constructions already familiar to mathematicians such as sets, Cartesian products, discriminated unions and functions.

Teacher: It seems to me that a type is yet another concept that has brought together work from many different cultures. It unifies modes, which were mainly used to correctly interpret bits in memory, records used for business data processing and mathematical models that understand types as sets!

Alpha: I'm happy to use the term type, but I object to the idea that we should think of types as sets. This may be useful when talking about numbers and records, but it does not work universally.

Tau: Can you give an example of where using sets as the model does not work?

Alpha: Take the Address programming language, developed by Ekaterina Yushchenko from Ukraine (Figure 5.2). The language was based on variables that represent memory addressess, a notion that will be called pointers in later programming languages. I do not see how you could represent pointers as sets.[6]

Tau: You really should stop thinking about programming as fiddling with bits. What do you need addresses for? To represent data structures such as lists or trees. Those can be mathematically modeled as recursive structures, which gives you higher level of abstraction, more suitable for reasoning about programming.

Epsilon: I agree with *Alpha* that we should not think of types as sets, but there is a more fundamental reason for that. Since 1970s, some of the authors writing about types started to think less about what types *are* and more about how types are *used* and this requires a different way of reasoning about types.

Teacher: So, what did this perspective lead to?

Epsilon: People realized that types were used to solve two problems. The *authentication* problem is to make sure that program operates only on values that are valid representations of its expected inputs. This is what types as sets can model well. The *secrecy* problem is to make sure that programmer only works with values using the provided procedures, rather than relying on the underlying representation. The secrecy problem is not something that the "types as sets" view can capture.[7]

Omega: Now, we are getting back to software engineering! The concept of secrecy is closely related to that of *information hiding*, which was introduced as a better way of structuring programs in the early 1970s. This is a great concept for managing a large team of programmers, because it lets you decompose a large system into smaller parts that can be managed and developed independently. You can, of course, do this without types, but automatic type checking provides a useful guarantee that information hiding has not been violated.[8]

Teacher: Has the idea of types became a managerial concept like structured programming?

Epsilon: Types may be useful for that, but they retained a strong identity as programming langauge features. However, the work on types and secrecy definitely shaped later programming languages like Clu and Ada. An "abstract data type" in Clu was a custom type that could be used only through the specified operation. Its representation was hidden and only available within the "cluster" that implemented it.

Tau: Did Clu actually guarantee these properties in a formal sense?

Epsilon: Are you asking how the types were chedked? In the early version, Clu used a combination of compile-time type checking, where possible, and run-time type checking for cases where compile-time checking was not possible.

Tau: See, this is another case where getting back to the logical origins of the concept of a type would have helped, which is what John Reynolds did in 1974![9] It turns out that if you extend the simply typed lambda calculus with a notion of user-defined types, you can check even the more complex cases at compile-time and also formally prove that programs do not rely on their data representation!

Teacher: Types seem to be used in a number of different ways. I think this raises an important question of whether it is helpful to use the same term for all of those uses?[10]

Gamma: Well, it does make the discussion somewhat confusing, but I think there is value in having a single notion of type. It allowed multiple people, following different perspectives, to contribute to a single concept. For example, as we just saw, the logical tradition provided a checking mechanism for the engineering notion of an abstract data type. This may not have happened if we used different terms!

Tau: If we can model all the different aspects of types as extensions of the types in simply-typed lambda calculus, then I do not see why we would need different notions. This can clearly be done for different primitive types, records and the problem of secrecy. Are there any other uses of types that I'm missing?

Alpha: I can see the utility in joining some of those ideas. Invoking an invalid operation on a value may well be the programming equivalent of a logical paradox, but I'm not sure about information hiding. This seems more like a social control mechanism than a useful tool for a programmer.

Tau: You probably wouldn't expect me to defend any idea advocated by *Omega*, but there is more to information hiding than just controlling access to data. Abstract data types were, in fact, fundamental for the use of types in the automated theorem prover LCF developed in the 1970s.

Omega: I imagine that is a complex system that needs to be divided into independently developed modules, so this is not a surprise...

Tau: Not quite! LCF used abstract data types to represent provable theorems. Axioms are pre-defined theorems and inference rules are functions that you can call to derive new theorems. The fact that a theorem is an abstract data type means that you cannot just arbitrarily create theorems. You can only derive them from axioms and so you always specify a valid proof!

Epsilon: Now we are getting to an interesting topic. The metalanguage used for composing theorems in LCF later became a stand-alone practically useful programming language ML. This only happened once ML was extended with features like records and module systems, but the basic idea of the language originated in the work on LCF.

Tau: Most importantly, all those later extensions to the ML type system did so without losing and of its nice theoretical properties.

Gamma: You talk about nice theoretical properties, but what exactly makes types in ML more rigorous than types in, say, Algol 68, which also had an extensive specification?

Tau: The Algol 68 specification is an informal English text. As we saw in the history of formalizing Algol, this is not enough for precisely specifying a programming language. The ML community had higher standards and when the different versions of the language were formalized as Standard ML, the langauge came with a formal mathematical definition of its behaviour. This setup makes it possible to prove that well-typed programs do not go wrong!

Alpha: You ignore the practical fact that many errors in real programs are related to input and output. It may be that the ML type system prevents many errors that may happen inside programs, but it ignores many more other errors.

Tau: That is true, but the appraoch pioneered in ML is very general. Since its conception, it has been used for checking how programs communicate over network, what resources they access, how they work with memory, as well as to control aliasing in object-oriented programs![11]

Gamma: That is an impressive list, but as far as I can see, none of these made it into a widely-used programming language. Am I missing something?

Epsilon: The idea of ownership types has been adopted by Rust, but I think the problem with this approach is that you need a new type system extension for each property that you want to check. That way, your types soon get out of control!

Tau: There is another option. You can make the basic type system more flexible and let the programmer use that for checking whatever properties they want. For example, you can implement a general mechanism for tracking what "effects" a program has. This can then be used for handling exceptions, I/O accesses, asynchronous communication, memory accesses and many other kinds of behavior![12]

Teacher: But now you are talking about the behavior of programs and not about values! I wonder if this is stretching the concept of a type?[13]

Tau: That depends on how you think about types. You are right that effects do not make sense if you think of types as sets of values. For that, you need to adopt a more general perspective. The most common interpretation today is to see types as a lightweight formal method.[14]

Omega: Can you please enlighten us on what is that supposed to mean?

Tau: Type describe some aspect of how a program should behave. They do so in a way that can be automatically checked and, if a program is well-typed, the types guarantee that the program will not contain certain kinds of errors. In other words, you can think about types as a limited form of specification.

Omega: Wait a minute! The whole point of a specification is that it is not a part of the program. It should be written by someone who is a domain expert, not a programmer. A specification facilitates a discussion about the problem. It sounds to me like your fancy types just mean that you have to write the logic twice, once as an actual program and once as a type. This must be quite inefficient and expensive!

Epsilon: Actually, types facilitate these conversations.[15] In the process of writing a type definition, you have to precisely understand the domain in order to devise the correct representation. This often involves talking to a customer and other team members, or even showing the type definition to the customer in order to decide what the domain allows. Types are then used to make invalid states unrepresentable.[16]

Gamma: It seems to me that types achieve one thing that proofs of programs failed to do. Earlier, we talked about the objection that program proofs cannot be trusted, because they are not supported by the same social processes as mathematical proofs. Well, if people regularly review and talk about types in their programs, then this provides a healthy social process that ensures the types are correct!

Teacher: I hear fewer disagreements than usual. Is understanding of types as a lightweight formal method something that we could all agree on?

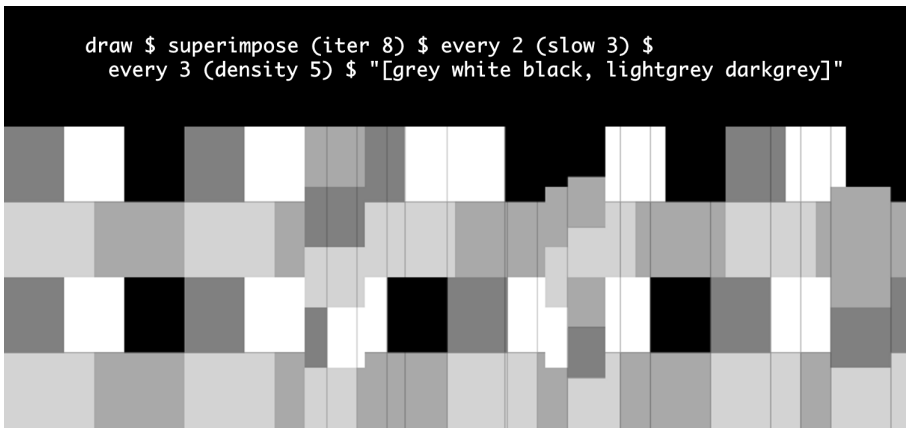


Figure 5.3: A visual pattern produced using the code snippet above in Tidal Cycles.[17]

Epsilon: We really do not need to try to sound so clever. Types are a programming assistance tool, just like test runners, linters and many other tools.

Gamma: I think they are just obstacles that get in the way of creative thinking! Types may be fine if you have a clear specification, but normal software needs to evolve and types only make that harder.

Tau: Do you think there is nothing positive at all about types, then?

Gamma: Well, Tidal Cycles, a system for visual and musical pattern composition is implemented in Haskell, so it has some types under the cover (Figure 5.3). You can use the inferred type information to make sense of code when writing it interactively, so that aspect is useful. But in this one case, types are not getting in your way!

Teacher: I see that we still disagree. But is that only bad, or does the plurality of interpretations allow the concept of types to evolve in multiple useful directions?

Gamma: I can imagine a useful direction for types after all. I recently saw a conference talk where the speaker used a language with a fancy type system, but she was using the system interactively. She first wrote a type of a function, then the system automatically suggested a part of the implementation based on the type information. She then filled the remaining holes in the code, again using the programming environment as an assistance tool along the way. I can imagine how this way of using types contributes to the vision of man-computer symbiosis that I advocated earlier!

Epsilon: I believe she must have been using a dependently-typed programming language like Idris or Agda. These languages are similar to ML that we talked about earlier, but they rethink types in a funny way. In ML, you wrote the code and the system inferred the types for you. In Idris or Agda, you write the types and the system helps you write the code!

Tau: You need to distinguish between the language and the editor. The editor may be interactive, but the types in the finished program are still specifications. In dependently typed language, the types are more precise, so the editor can help you more. For example, you can define a type `Vect` $n\ \alpha$, which represents a vector of length n containing values of type α. A type of a function can sometimes fully determine its implementation. Can you guess what a function of the following type does?

$$\texttt{Vect}\ n\ \alpha \to \texttt{Vect}\ m\ \alpha \to \texttt{Vect}\ (n + m)\ \alpha$$

Alpha: Well, if it takes vectors of lengths n and m that contain the same thing, and returns a single vector with a length $n+m$, then I guess the only thing this can do is to append them. Could you trick the system, maybe by repeating the same value many times?

Tau: Right. In this case, you can trick it, because there are multiple possible functions of the same type. You could, of course, make the type even more precise and, for example, disallow copying of elements from the list. Then the editor may be able to give you the only possible correct implementation fully automatically!

Epsilon: I do not understand why you insist on thinking about types as a formal specification when you keep talking about how types are used by the editor to help you write code! That is exactly how I want to use types too…

Tau: All I insist on is that types also give you correctness guarantees.

Epsilon: But they do not have to do that in order to be useful as tools.

Tau: What a crazy idea! Can you please explain what you mean?

Epsilon: Happy to. Look at the TypeScript programming language. Its type system does not give you absolute guarantees. The langauge extends JavaScript and adds type checking that has a number of carefully considered aspects where the designers choose simplicity and compatibility with JavaScript over safety.[18]

Tau: But this is ridiculous! How is that supposed to be useful?

Epsilon: It works very well in practice. Types in TypeScript support the structuring of large programs, they serve as a useful documentation, they help you write code via editor auto-complete and they help you avoid many common programming errors. You do not need a formal correctness for that.

Tau: So you are happy to lose safety and expose your users to potential vulnerabilities?

Epsilon: Safety is not an issue in TypeScript. It compiles to JavaScript, which has full runtime checking. A bit like the Clu language, which also relied on runtime checks for some of the abstraction guarantees.

Teacher: I would like to conclude by reflecting on the history of types briefly. We tried agreeing on various definitions of what a type is, but it seems that we have not been very successfuly at this, perhaps with the exception of the brief moment when all the different ideas about types came together in the ML programming language. Has this recurring disagreement been a curse or a blessing?

Tau: The fact that people do not have a clear definition makes them make confused claims and the lack of a clear definition is an obstacle to the progress in programming language research. It is unfortunate that the engineering notion of data type, modelled later as sets, got confused with the logical notion of a type, modelled after the lambda calculus.[19]

Gamma: As I said already, I think the fact that those overlapping conceptions of a type share the same name makes it easier to exchange ideas between different communities. We saw how Clu adopted the logical notion of type checking for the engineering notion of abstract data types. I wonder if the ML language would ever be born if we kept all those notions separate…

Tau: That might be a valid point, but how do you want to talk about something that you cannot even clearly define?

Gamma: That might be a problem for a mathematician, but humans do that using natural language very day. The meaning is use. The meaning of a 'type' is given by its use in the programming practice and in theory of programming languages.[20]

Tau: This is a very clever-sounding answer that does not actually say anything. How do you expect that science can progress with such lazy thinking where anything goes?[21]

Alpha: You might not be able to directly compare and synthesize different theoretical contributions concerning types, but that is not what programming is about! In the end, we care about what programs we can write using types. The capabilities enabled by types are something you can assess regardless of your definitions. Many different kinds of types can be used to build developer tooling, so just look at the end results!

Teacher: This is a good point to conclude our discussion. For better or worse, I think that this statement characterizes the history of types very well. Programmers and computer scientists exploit what they find useful for the design of more elegant, practical and usable artifacts. In this, they have been able to bring together good engineering practices, ideas from mathematical logic as well as ad-hoc programming tricks.[22]

Programming with Types

Are Types Invented or Discovered?

In February 2016, some 350 functional programmers came to Karków to join the programming conference LambdaDays. The opening keynote titled "Propositions as Types"[23] was presented by a computer scientist, Philip Wadler, who is well known for his many contributions to the theory of programming languages.

The topic of Wadler's keynote can be accurately related in two ways. A dull technical summary is that the talk described a correspondence between two formal systems, a natural deduction system for intuitionistic logic and the simply typed lambda calculus. A more exciting account is that the talk showed a remarkable correspondence between logic and functional programming, suggesting that functional programming languages are not mere human inventions, but are instead rooted in deeper truths about the universe.

Seen through the perspective of the initiated, the talk documents how the idea of types, which originated in Bertrand Russell's work in formal logic, made it into the first functional programming language, the lambda calculus, before ending in programming languages that functional programmers use today, including Haskell, OCaml and F#.

A historian or a philosopher of mathematics attending the talk would raise many objections.[24] First of all, the two systems related in the talk were both developed in response to the same problems in work on the foundations of matehematics and they are carefuly constructed to match, so their correspondence is not as surprising as it may seem. A listener who is not acquainted with the assumptions of the mathematical culture may be surprised by another aspect of the talk. Wadler talked about the relationship between two formal systems that originated in the field of logic before the first computers existed. Yet, he clearly suggests that one side of the relationship is about programming languages, for example by saying that the "lambda calculus is the world's oldest programming language, being defined in the 1930s, and it is the coolest because it was defined decades ago before the first stored program computers were built". Such claim only makes sense if we see programming langauges as abstract mathematical ideas, but not if we see programming languages as the result of human engineering efforts..

Wadler's keynote makes for a good opening of a chapter on types in programming because it illustrates many of the perspectives typically accepted by the mathematical culture of programming, which had a strong influence on the idea of types in programming languages. In Chapter 2, the proponents of the mathematical culture included mainly academic computer scientists. The increasing popularity of functional programming made such views common among many practitioners, such as those attending the LambdaDays conference, who however combine mathematical assumptions with applied engineering methods.

The origin story of types in programming languages, as told by the mathematical culture, starts with the history of formal logic in the early 20th century. The idea of 'type' that appeared in the early work on logic has, indeed, influenced types in programming languages. Except this influence is not completely direct and takes place some 15 years after types first appear in programming languages, most likely independently of formal logic. With this caveat, I will start by reviewing the history of types in the early 20th century logic, before returning to programming in the late 1950s.

Resolving Logical Paradoxes with Types

In 1902, Bertrand Russell wrote a letter to Gotlob Frege to let him know that he discovered a paradox in his formal system published in 1879. Using a modern notation, the system made it possible to define predicates such as $p(x)$ to denote that a property p holds for an argument supplied in place of the variable x. Negation of a predicate, written $\neg p(x)$ would denote that the property does not hold. Russell realised that Frege's system made it possible to define a predicate $p(x)$ such that it is true if and only if $\neg x(x)$.

This is a clever trick, because it uses a variable x for both the predicate and the argument that it should be talking about. The definition is not just clever, but it is also paradoxical. If we assume $p(p)$ then, by substituting p for the variable x in the definition, we get $\neg p(p)$. But then, again by substituting p for x in the definition of p, we get $\neg\neg p(p)$, which is equivalent to $p(p)$. Russell concluded that both $p(p)$ and $\neg p(p)$ hold in Frege's system, which is a contradiction.

Russell's solution evolved gradually over several years.[25] The "Appendix B" of his 1903 book "The Principles of Mathematics"[26] (not to be confused with later and better known "Principia Mathematica") defines the so called "doctrine of types". Here, Russell introduces a hierarchy of types for classifying objects and classes (think sets containing individuals or other sets). In the doctrine of types, the lowest type represents *individuals*, the next type *classes of individuals*, the next *classes of classes of individuals* and so on. Russell does not go as far as imposing types on *propositions*, which he considers "harsh and highly artificial". This means that types do not apply to predicates like p, which was the source of inconsistency in Frege's system. Consequently, the doctrine of types from "Appendix B" does not eliminate all contradictions from the system.

The flaw is corrected in a 1908 paper "Mathematical Logic as based on the Theory of Types".[27] Here, Russell describes the "vicious-circle principle" and requires that "whatever contains an apparent variable must not be a possible value of that variable." The type of a proposition that contains a variable must be higher than the type of values assigned to the variable. This rules out the contradiction caused by $p(p)$ discussed earlier. In the definition $p(x)$, the proposition p contains a variable x. If x is a variable of a certain type, the type of p must be higher and so p cannot be used as a value of x. Writing $p(p)$ is ilegal.

The next major step in bringing types to programming, at least in the eyes of mathematically-minded programmers, was made in 1940 by Alonzo Church. Like Russell, Church was interested in foundations of logic. Around 1928, he began working on a formal system that was based on the concept of functions, rather than sets (or classes). The system was published in 1932 and later became known as the lambda calculus. In 1940, still before any modern digital computers existed, Church published a paper "A formulation of the simple theory of types"[28]. In the paper, Church describes a variant of Russell's theory of types which "incorporates certain features of the calculus of λ-conversion" or lambda calculus

as we now know it. Church's hierarchy of types for the lambda calculus defines ι as a type of individuals, o as a type of propositions and $\alpha\beta$ as a type of functions with a parameter (independent variable) of type α and result (dependent variable) of type β.

The purpose of types, as introduced by Russell, was to eliminate paradoxes from a formal logical system. To serve this role, we do not need to know much about what types *are*. Russell defines a type as the "range of significance of a propositional function, i.e., as the collection of arguments for which the said function has values."[29] However, he also later notes that it is unnecessary to know what objects belong to the lowest type and that, in practice, only the relative types of variables are relevant. Church makes a similar remark when he "purposely refrains from making more definite the nature of the types [of propositions and individuals]".[30] This is in stark contrast with types as they first appear in programming languages, where there are no relative types and the nature of the primitive types is all that matters.[31]

Evaluating Arithmetic Expressions in Two Modes

As in formal logic, types in contemporary programming languages are often used for capturing what the range of possible values of a variable and terms are. If we have a variable `COUNT` of type integer, this indicates that its values may be numbers such as $1, 2, 3, \ldots$, but not a floating-point number like 3.14 or a string `"Bertrand"`. As in logic, this may be used for restricting what programs are valid. In *statically typed* programming languages, types are checked before program is run. If you write a program that attempts to multiply `"Bertrand"` by 3.14, the program will be ruled out as invalid. The compiler will know that multiplication can only be applied to arguments of numerical types. The value `"Bertrand"` is not of a numerical type and so the checking procedure will report an error.

Unlike in logic, the nature of primitive types is central in programming. A compiler typically uses types to check that primitive operations are called with values of correct types as arguments. It may also use the fact that a variable is of a certain type to allocate appropriate number of bytes of memory for storing it. However, different modern programming languages use types very differently and none of the above is universally true. It should, however, provide a good basic intuition.

The idea of types is so commonplace in contemporary programming that it is perhaps surprising how long it took before programmers started consistenlty using the term 'type'. The IBM 704 FORTRAN manual,[32] published in 1956 uses the term 'type' in its everyday English sense, often interchangeably with words such as 'class'. The report talks about "32 types of statements", "three basic types of decimal-to-binary or binary-to-decimal conversions", but also "two types of variables":

> *Two types of variable are also permissible: fixed point (restricted to integral values) and floating point. Fixed point variables are distinguished by the fact that their first character is I, J, K, L, M, or N.*

In a more formal description of functions, expressions and arithmetic formulas (Chapter 3 of the manual), the authors use the term 'modes'. For functions, each of the arguments as well as the result can be in either of the two modes. This yields a number of different mode configurations and a separate function name must be given for each of the supported configurations. The IBM 704 version of FORTRAN also required a particular naming of

functions. All function names must end with `F` and, if the mode of the function is fixed, the name must also start with `X`. For example, a function `XINTF` truncates an argument in the floating point mode and produces a fixed point number.

Arithmetic expressions are also in one of the two modes and this is used to define permissible expressions. For example, if `E` and `F` are expressions of the same mode, then `E+F`, `E-F`, `E*F` and `E/F` are also expressions of the given mode. In other words, numerical operators cannot be applied to expressions in different modes. In retrospect, the similarity with types in formal logic is not too hard to see. Just like Russell used types to make writing of the self-referential proposition $p(p)$ invalid, FORTRAN uses modes to make writing `E+F` invalid for expressions representing different kinds of numerical values. Yet, it took almost 10 years before someone made this connection. But as we saw in Chapter 2, at the time when FORTRAN was being created, the very idea of programming languages as independent entities, more so mathematical ones, was only just appearing.

It seems possible that FORTRAN adopted the term 'mode' because the mode of an expression was used to determine what kind of machine code the compiler produced for the expression. The machine code for `X+Y` was different in the fixed and the floating-point mode. In other words, the mode of an expression determines one of two possible modes of compilation. If this is the case, the terminology used in FORTRAN was shaped by the hacker culture of programming in that the name of the concept was derived from a technical implementation trick that was used to actually compile FORTRAN code. This would not be too surprising. After all, it was the author of the 1956 FORTRAN manual, John Backus, who later said that "programming in 1950s was a black art". Interestingly, the term 'mode' outlived its hacker culture origins. It did not disappear immediately and we will see that it still appears in late 1960s and 1970 when it was adopted by the mathematical culutre, possibly before it fully embraced the term 'type' with its links to the early 20th century formal logic.

The first modern usage of the term 'type' is in the 1958 preliminary report on the Algol programming language,[33] which was still known as International Algorithmic Language (IAL) at the time. The term was introduced during a meeting between the American ACM group and European group working on IAL in Zurich at the end of May 1958. Tracing the history of types, Simone Martini suggested that "the technical term appears to be just a semantical shift from the generic one; in particular, there is no clue that in this process the technical term 'type' from mathematical logic had any role".[34] Many of those present at the meeting were notable proponents of the mathematical culture of programming, so it is possible that they were familiar with the use of types in logic, but there is no acknowledgement of influence. Even if the term was, in fact, imported from logic, the authors of the report did not find this significant enough to discuss the relationship explicitly.

Types or modes in FORTRAN and Algol 60 were rather basic compared to what programmers know as types today. Algol 60 had three primitive types, integer and real numbers and a Boolean type to represent logical values. It also supported arrays, but those were not treated as a type, but rather as a "subscripted variables" where the subscript determined a position (index) of a value in the array. The designers considered other types including complex numbers, vectors and matrices but those did "not seem to have stirred much attention during the discussions".[35] One feature that was notably lacking from Algol 60 was a mechanism for representing structured data such as records consisting of multiple fields. It is striking that the mathematically-minded designers of Algol were thinking about vectors and matrices, but not about a mechanism for representing structured data that was becoming essential in business data processing at the very same time.

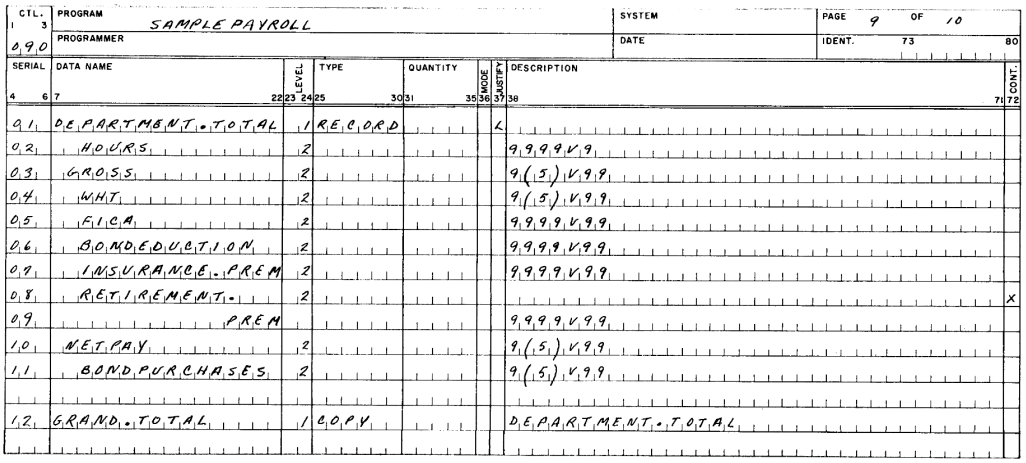


Figure 5.4: A sample data description card from the IBM Commercial Translator manual[36] specifying a structured record with total payroll spending per department.

Structuring Scientific and Business Data

As we saw in Chapter 2, the late 1950s was when the idea of programming language as a stand-alone object, not linked to a specific computer emerged. This brought together the managerial need for portability, mathematical tools for thinking about languages and hacker expertise in implementing programs that translated human-readable code to machine code. A number of programming languages came out of this effort. FORTRAN was a product of the "black art" of programming in 1950s. It was a successor of various automatic programming systems developed previously, but it was also shaped by the engineering focus on supporting efficient scientific applications. Algol 60 kept the same focus on scientific applications as FORTRAN, but approached the issue from a mathematical perspective.

At the same time, computers were also becoming increasingly used for commercial applications. The Eckert–Mauchly Computer Corporation, created by the ENIAC inventors, ran into financial difficulties and was sold to Remington Rand in 1950, where it continued as a division and started selling the UNIVAC I computer in 1951. IBM soon joined the electronic computer market with the IBM 701, which became available in 1952. Programmers at both of the companies faced the problem of creating business data processing systems. The implementation of those relied on the same amount of hacker ingenuity as the implementation of systems for scientific applications, but the structure of the problems they posed was shaped by business needs rather than scientific needs. In particular, processing business records required ways of working with structured data consisting of both textual and numerical information stored in files. Two early programming systems that tackled this challenge, and were both products of meeting of the managerial and hacker cultures, were Flow-Matic and Commercial Translator (COMTRAN). Flow-Matic was developed by Grace Hopper at Remington Rand and it was released for UNIVAC to customers in 1957. COMTRAN was developed at IBM, under the leadership of Bob Bemer and was publicly presented in 1957, although the implementation only became available later.

Flow-Matic and COMTRAN may seem like obscure historical artefacts, but they became sources of ideas for a programming language that appeared in 1959 and is still in use today, the Common Business-Oriented Language (COBOL).[37] The language was produced by the

Committee on Data Systems Languages (CODASYL) that was established with the goal of establishing a common langauge for data processing. When CODASYL started looking into the idea, it first established a Short-Range Committee that would conduct study existing business compilers and provide materials for a task group that would design the language. In a perhaps unsurprising turn of events, the Short-Range Committee was later tasked with defining interim language, that would be used before the final design was agreed. The interim language was published as COBOL 60 and received wide-spread adoption in industry, while the final design never materialized.

Flow-Matic and COMTRAN influenced COBOL and future commercial programming languages in a number of ways. One notable design influence of Flow-Matic was the use of readable English words rather than symbolic code. However, the most interesting feature for our discussion about types is the fact that both Flow-Matic and COMTRAN separated "data description" from "program description". The data description specifies the way data is stored in terms of files, records and fields. Figure 5.4 shows a data description from the COMTRAN manual for a sample payroll system. Records are hierarchical and nesting is indicated by the 'level' column on the data description card. (Nesting was supported in COMTRAN, but not in Flow-Matic.) Type `RECORD` defines a new record whereas the `COPY` type indicates that the structure of the value is the same as that of another record. For individual fields, the 'description' column specifies the format of the data. For example, `9(5)V99` indicates that a numerical value formed by up to 5 numerical characters, followed by a decimal point and two decimal digits. The COBOL language, which was developed in 1959, adopted both the general concept of records, as well as the formatting specifier (known as the "PICTURE clause").

A contemporary programmer can easily see COMTRAN records as precursors of the record data type that exists in many recent programming languages, but the way of thinking about data descriptions in business-oriented programming langauges was very different from thinking about types (or modes) in scientific programming. In COBOL, data description became one of three "divisions" that make a COBOL program, alongside with "procedure division" used for specifying program code and "environment division" which specifies machine-specific information such as input and output devices or machine configuration. As the following extract from a "Detailed description of COBOL"[38] shows, the COBOL designers thought of data division as part of the program that specifies the physical structure of data on a tape, even though in a somewhat high-level way:

> *The* `DATA DIVISION` *is concerned with information from files, data which are developed during the program and placed into working storage, and constants defined by the user. (...) The basic concept used in the data organization is that of 'logical record' which can be defined to be any consecutive set of information. (...) It is important to note that several logical records may occupy a block on a tape (i.e. physical record), or a logical record may extend across several physical records.*

Making the connection between data definitions in business programming languages and types in Algol required bridging the difference in thinking about data and types. Moreover, the exchange of ideas was hindered by the social circumstances and prejudices. The discussion about types and Algol happened among the mathematical and engineering culture, often in conferences and publications supported by the Association for Computing Machinery (ACM). This did not involve programmers working on business data processing

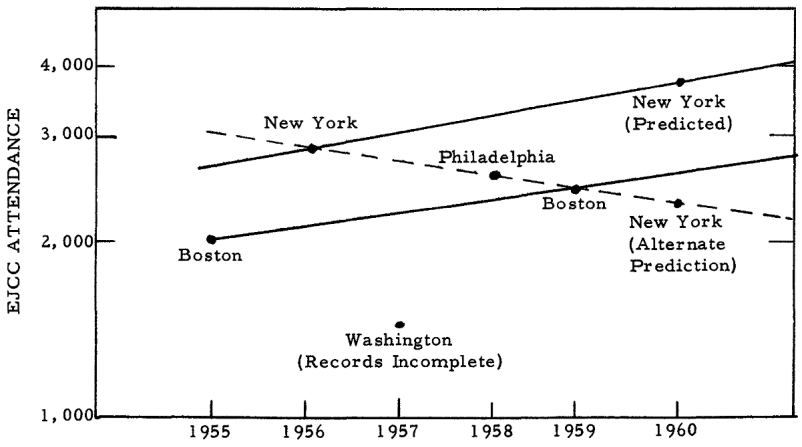


Figure 5.5: Predicted number of attendees at the EJCC based on growth in attendance in Boston (full line) or based on overall eastern trend (dashed line). The General Chairman noted that, if the optimistic prediction is correct, there is no hotel in New York that can hold the EJCC.[41]

who read the Datamation magazine, met through the volunteer-run SHARE user group and collaborated through CODASYL. The division between the mathematical and the business culture was not complete though. Since 1951, the Western and Eastern annual Joint Computer Conferences (WJCC and EJCC) provided a venue where communities mixed and in 1960, the conference was expected to attract some 3,000 attendees (Figure 5.5).

The mathematically minded members of the ACM were aware of the work done in the data processing industry, but felt that their publications "failed to report fundamental research in the data processing field".[39] The ACM members were not interested in "how someone else solved his payroll problem", because they believed that "the solution is almost sure to be too specifically oriented to the company for which it was solved and [lacks] general applicability."[40] Yet, learning how COBOL programmers solved their payroll problem is exactly what was needed for the next step in the evolution of types.

Towards a Universal Programming Language

The limitation of scientific programming languages for working with other kinds of data structures were widely understood at the time and many mathematically-minded computer scientists attempted to design a way of representing richer data structures modelled after established mathematical structures.[42] One of those mathematically-minded computer scientists was Jonh McCarthy who we encountered in the preceding chapters as the author of LISP and a proponent of the mathematical culture. At the culture-bridging Western Joint Computer Conference, McCarthy presented a paper "A Basis for a Mathematical Theory of Computation",[43] which was the first step of his effor to develop mathematical science of computation. According to McCarthy, his paper contains contributions to a number of goals, the first of which is the development of a "universal programming language" that would support both scientific and business-oriented programming.

McCarthy believes that "Algol is on the right track but mainly lacks the ability to describe different kinds of data", which has become a key feature of business data processing languages. In McCarthy's view, "languages such as Flow-Matic and COBOL have made a

start in this direction, even though this start is hampered by concessions to what the authors presume are the prejudices of business men". Interestingly, the concessions do not have much to do with the way data descriptions are structured. McCarthy mainly criticises the use of English-like notation in COBOL, which he finds unsuitable for formal treatment. It is worth noting that McCarthy was exercising his own prejudices here. The syntax of COBOL was defined in a formal meta-language that was not unlike the Backus-Naur form used in the definition of Algol 60.[44] The semantics of COBOL was described, just like the semantics of Algol, using plain English, albeit with perhaps less technical rigor.

To McCarthy and other proponents of the mathematical culture, introducing the ability to describe different kinds of data first required the development of mathematical theory of data types. The specific details of a concrete programming language would only come later. In an acknowledgement that remains typical for work of the mathematical culture, McCarthy confirms this and says that "the formalism for describing computations in [his] paper is not presented as a candidate for a universal programming language because it lacks a number of features, mainly syntactic, which are necessary for convenient use."

Much of McCarthy's paper is dedicated to various ways of defining functions in terms of base functions, conditional expressions and recursive definitions. Inspired by Church's lambda calculus, he also discusses functions taking functions as arguments and adopts the λ notation used by Church. When he gets to talking about data, McCarthy models the values that functions can accept as arguments and return as results using mathematical sets. To make the terminology in this chapter even more confusing, McCarthy refers to such sets of values as "data spaces". He does not adopt the term "type" even though he cites lambda calculus and the term "type" was already in use by the definition of Algol. This is surprising only in retrospect. Types in lambda calculus were used to avoid logical paradoxes, whereas types in Algol described the kind of data accepted by a sub-routine. In both cases, types could rule out certain invalid expressions, but this link was not necessarily obvious at the time.

The specific ideas presented by McCarthy include a number of ways of defining new data spaces in terms of existing data spaces. This is done using the usual mathematical operations for working with sets. A Cartesian product of data spaces A, B creates a data space $A \times B$ consisting of pairs (a, b) such that $a \in A, b \in B$. Although McCarthy does not say so explicitly, the construction roughly corresponds to records in COBOL, which also combine multiple values of other primitive types. McCarthy also defines union of non-intersecting sets $A \oplus B$ and a power set A^B which models function values and remarks on the importance of recursive definitions. The only example given by McCarthy is the recursive data space S defined as $S = A \oplus (S \times S)$. This is the model of S-expressions, a notation used in his new programming language LISP. The non-intersecting union $\oplus$ operator represents a choice, so an S-expression is either an atom $a \in A$ or a pair (x, y) composed of two S-expressions $x, y \in S$.

The notion of a "data space" as used by McCarthy, type as used by Russell and type as used in Algol come closer together in a monograph "Structured Programming".[45] The book was published in 1972 and includes three chapters. In the first chapter, Edsger Dijkstra writes about his notion of structured programming. In the second chapter, C. A. R. Hoare refines and extends the line of work started by McCarthy. He adopts the term "type" and explicitly discusses the similarities between the notion of type used by programmers, citing the notion of types in Algol, logicians, citing Russell's use of type to avoid paradoxes, as well as in informal mathematical writing. Finally, the third chapter by Ole-Johan Dahl and

```
type local car = (make:manufacturer; regnumber:carnumber;
                  owner:person; first registration:date);
type visitor car = (make:manufacturer; regnumber:carnumber;
                    origin:country);
type car = (local:localcar,
            foreign:foreign car).
```

```
type car = (make:manufacturer;
            regnumber:carnumber;
            (local:(owner:person;
            first registration:date),
            foreign:(origin:country))
            ).
```

Figure 5.6: Two ways of representing data about cars using records (written using ";") and discriminated unions (written using ","). Above, a car is a either local (including its owner and a first registration date) or foreign (including country of origin). Below, the type of a car is a record containing the shared attributes and a nested discriminated union containing additional information about either a local or a foreign car.

C. A. R. Hoare discuss program structuring in SIMULA, an object-oriented programming language discussed in Chapter 6. This is presented as a synthesis of "the design of data and the design of programs," that is the topics of the first two chapters of the monograph.

Hoare presents his ideas on data structuring using a hypothetical un-implemented programming language with concrete syntax for type definitions and includes records, fixed-size arrays, potntially infinite sequences, but also discriminated unions that represent a choice of alternatives, powersets (to model function values) and recursive type definitions. For each of those, Hoare provides a set-theoretical model and notes on implementing them. The set-theoretical models of records and discriminated unions are like the $\times$ and $\oplus$ operators introduced by McCarthy. Hoare makes it clear that he is talking not just about mathematical models, but about real computer programs. Consequently, his examples that include a range of realistic problems, for example a data structure to store information about cars shown in Figure 5.6.

Meanwhile, the designers of Algol 60, united in the international IFIP Working Group 2.1, embarked on the task of designing a successor version of Algol. Improving the limited support for data structuring in Algol 60 and including richer data types was one of the goals for a successor language. As I mentioned a couple of times already, the design process was far from smooth and the process involved "discord, resignations, unreadable documents, a minority report, and all manner of politicking."[46] We do not need to get into the particulars of this contentious history, but it is worth noting that the dissenters who felt that the proposed language fails to offer a suitable tool for structuring programs included the two authors of the later "Structured Programming" monograph discussed above, C. A. R. Hoare and Edsger Dijkstra.

A common criticism of Algol 68 was that the definition was unreadable to an uninitiated reader. This also plagued the terminology of data structuring. In a section of the Algol 68 report[47] that compares Algol 68 with Algol 60, the authors note that:

> *Whereas ALGOL 60 has values of the types integer, real and Boolean, ALGOL 68 features an infinity of "modes", i.e., generalizations of the concept "type".*

The authors of the Algol 68 Report decided to kept the term 'type' for primitive types, but re-introduced the term 'mode' when referring to structures that are constructed by 'mode declarers' which compose other modes or primitive types. Rather than adopting the business term 'record', the report talks about structured values. The notion of mode in Algol 68 still keeps some of the original implementation-centric hacker aspects. For example, a mode can specify whether a value will be stored on the global heap or in a local stack. In most other respects, however, Algol 68 is a product of the mathematical culture. This is apparent in its attempt to achieve "orthogonal design" that minimizes the number of primitive concepts, formal writing style of the report as well as idiosyncratic typography and new terminology.

The re-introduction of the term "mode" for talking about composed types in the Algol 68 report was only a brief diversion in the history of types. Another of the dissenters from the design committee, Niklaus Wirth, went on to develop the programming language Pascal,[48] which was arguably the most successful attempt to resurrect the spirit of Algol 60 and develop it into a universal programming language. Pascal not only switched the terminology back and referred to custom data structures as *types*, but it also adopted forms of structuring data that were familiar to COBOL programmers including records (representing fixed number of components of possibly different types) and files (representing a sequence of components). Records in Pascal can also have multiple variants, which is arguably a more business-friendly approch for implementing McCarthy's union ($\oplus$).

It is worth noting that the idea of introducing new user-defined types in a program was not always seen as an inherent aspect of programming. Some authors used the term "extensible languages" to refer to languages that can be extended through custom type (or mode) declarations in order to create new languages, more suitable for solving problems in a certain domain. An example is the review of "Definition mechanism in extensible programming langauges",[49] which recognizes "mode constructors" as one such mechanism of an extensible language. The paper puts this extension mechanism alongside other extension mechanism such as the ability to define new language syntax using macros.

Despite the objective to produce a language that would be suitable for both scientific computing and business data processing, most of the work on structured data types was done at the intersection of engineering and mathematical cultures. Throughout the process, there was an interesting borderline between more mathematical and more engineering designs. For example, Niklaus Wirth was commissioned by the Algol 68 design committee to produce a report representing the "views of the pragmatists". The pragmatic engineering design included a type of pointers, which are much closer to how machines operate, but break the otherwise elegant mathematical theory based on sets. In contrast, the mathematically-minded idealists favored recursive definitions, because they have an elengant set theoretical model.

The mathematics used for talking about types in the late 1960s was mostly set theory. This provided a model for records, unions and recursive data types. References to types as known from mathematical logic started to appear, but those were not treated as types in the sense used in programming language. In particular, Hoare (1972) only referenced Russell's notion of types for avoiding logical paradoxes, while McCarthy (1961) referenced Church's untyped lambda calculus. The confusing and ever-changing terminology involving types, modes and data spaces only confirms that the notion of a type was still in flux. The link between programming languages and lambda calculus was slowly gaining importance throughout the 1960s in the pioneering work on programming language semantics.

The Next 700 Programming Languages

The mathematical influence on programming languages, through which types in programming and types in logic became one and the same thing, features one of the most bizarre characters of the history of British logic. Mervyn Pragnell was not an academic and never published any papers, yet he appears in the stories told by many British pioneers of theoretical programming language research. Pragnell was the organizer of an underground logic reading group where many of them met. His group was literaly underground. It met in the basement of Birkbeck College, without the knowledge of college authorities, thanks to a lab assistant who Pragnell knew and who would let the attendees in. Pragnell recruited members of the reading group in various ways, such as by lurking around the Foyles bookshop, looking for people buying books on mathematical logic and inviting them to join the group. Several members of the group "recall a distinctly theological atmosphere, taking turns to read aloud pages of hefty books on logic and mime the formulae!"[50]

Although there are no written records of the group meetings, it is likely that the lambda calculus was one of the topics discussed in the reading group. The topic may have come to the group through Christopher Strachey, best known for his work on the semantics of programming languages. As documented by Astarte,[51] Christopher Strachey had been introduced to the lambda calculus by Roger Penrose around 1958 and employed another member of the reading group, Peter Landin, who had also been interested in the lambda calculus in his consulting business between 1960 and 1964. Yet another member of the group and early proponent of functional progrmaming, Rod Burstall recalled that "Landin taught [him] about lambda calculus and functional programming in the pub round the corner from the College, The Duke of Marlborough."[52]

At Strachey's consulting business, Landin worked on a compiler for the Feranti Orion computer. The machine architecture made producing optimal code directly awkward and Landin suggested to use a language inspired by lambda calculus as an intermediate program representation.[53] Based on this idea, Landin produced a series of papers that describe the approach in a more basic research format. In the first paper of the series, "The Mechanical Evaluation of Expressions",[54] Landin introduces a lambda calculus-inspired formalism of "applicative expressions". He describes how "some forms of expressions used in current programming languages", including lists, conditional expressions and recursive definitions, can be modelled using applicative expressions. He then describes a mechanism for evaluating such applicative expressions. The paper reflects Landin's mathematical background and the evaluation mechanism described abstractly, in a formal mathematical langauge. In modern terms, Landin presents an abstract machine, which defines the state of a machine using mathematical structures, shown in Figure 5.7, and models evaluation using rules that turn a state, such as (S, E, C, D), into a new state (S', E', C', D').

The next step in Landin's work came when he was invited to a conference on Formal Language Description Languages (FLDL) in Vienna in September 1964. In his presentation, which was published as proceedings paper 2 years later,[56] Landin extended his method and used it to define the semantics of the Algol 60 programming language. The details of this work were later described in a two-part paper in the Communications of the ACM.[57] The papers follow the same approach as the earlier work. Landin describes how to translate Algol 60 programs to his "applicative expressions", extended with features necessary to support imperative programming like assignments and jumps; he then describes a model

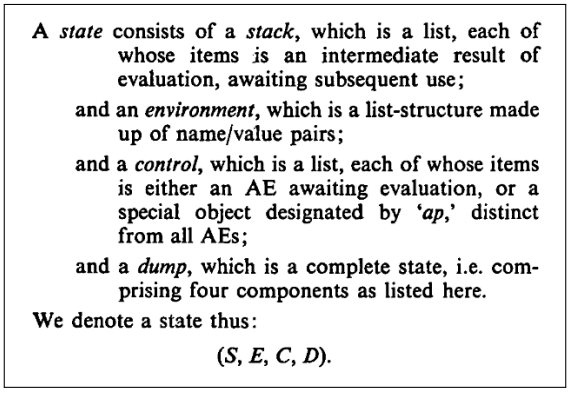


Figure 5.7: The state of an abstract machine for evaluating applicative expressions (AEs). The abstract machine models the evaluation of a lambda calculus-inspired intermediate representation of programs.[55]

of evaluation for those expressions that he now calls the "SECD machine", after the symbols that appear in the definition of the machine state in Figure 5.7.

Landin's use of the lambda calculus in the series of papers published while working with Strachey is certainly a step towards bridging the gap between formal logic and programming. Algol 60 programs are translated to lambda calculus-like language, but the two are still to some extent separate kinds of entities. The papers have also not yet elaborated the connection between the two notions of types. After attending the FLDL conference, Landin moved from Strachey's consulting business to work on experimental programming languages at the UNIVAC division of the Sperry Rand corporation, which acquired Remington Rand in 1955. The move from work on programming language implementation to work on programming language design provided an incentive for developing a different way of using the lambda calculus. In the aforementioned four papers, lambda calculus serves as an intermediate language to define the semantics of another programming language. After moving to UNIVAC, Landin published a paper called "The next 700 programming languages"[58] which uses lambda calculus not just for the semantics, but also as a source of langauge design ideas.

The ISWIM ("If you See What I Mean") language presented by Landin in this paper attempts to provide a general system for compositional programming, an approach that became the cornerstone of the functional family of programming languages. Landin explains his approach in the introduction:

> *ISWIM is an attempt at a general purpose system for describing things in terms of other things, that can be problem-oriented by appropriate choice of "primitives." So it is not a language so much as a family of languages, of which each member is the result of choosing a set of primitives.*

This also explains the number 700 in the paper title. The paper starts with a quotation reporting that there are over 1,700 programming languages used in over 700 application areas. By supplying appropriate sets of primitives, ISWIM can be used for the next 700 of desired application areas. ISWIM consists of a "purely functional" sub-system with impure extensions including assignment and a form of jump (using so-called program points). Programs are written as expressions with auxiliary definitions given using `where` expressions.

In a number of places, the paper makes explicit detailed references to mathematical logic. Landin is not just borrowing the λ notation as McCarthy did for LISP, but he is actually marrying programming language theory with work on lambda calculus. An example is his discussion of the β equivalence rule, which ensures that an expression L `where` $x = M$ is equivalent to a program obtained by substituting the expression M for a variable x in the expression L. The rule makes it possible to eliminate `where` from programs, which Landin connects to the Church-Rosser theorem of formal logic.

ISWIM itself was intended as an abstract description of a family of programming languages and it remained unimplemented. However, at the end of 1960s, two early functional programming languages turned the ideas from ISWIM into reality. They were the PAL language at MIT in 1968 and the GEDANKEN language at Argonne National Laboratory in 1969. Both followed the same structure as ISWIM with a core functional sub-language based on function application. Both PAL and GEDANKEN were missing features typically associated with types in functional languages today, including static checking of types, type inference and data structures such as records and unions that were already studied in theoretical work and present in the Algol 68 design.

Landin was directly involved in the implementation of the PAL language, having moved to MIT briefly in 1966-67. The initial LISP implementation was written by Landin together with James H. Morris Jr. who was a PhD student at MIT at the time. Morris was inspired by Landin's "incisive analyses of programming languages"[59] and his work is particularly interesting for our story because he extended the correspondence between programming languages and the lambda calculus from just the applicative structure used in ISWIM and PAL to the structure of types.

Types Are Not Sets

James H. Morris is one of the computer scientists who illustrate the fact that individuals can birdge and contribute to multiple cultures of programming. When Morris came to MIT for his PhD, he joined Project MAC, which was the home of the 1960s hacker culture at MIT that I discussed in Chapter 3. He worked on the interactive programming system OPS-3 and made it faster by a factor of 25 to 200 by replacing an interpreter with an (on-line) compiler,[60] a feat that would appeal to the practically minded hackers who care about making computer systems better. However, his PhD thesis "Lambda-calculus models of programming languages"[61] is a clear contribution to the mathematical culture of programming. Not long after his PhD, Morris joined Xerox PARC to work, most notably a modular programming language Mesa and the Cedar programming environment (Figure 5.8). This work had a strong engineering ethos. This is clear from its description; Cedar is a "software equivalent of the kind of machine shop needed by an engineering laboratory."[62]

Morris' PhD thesis is analytical in the same sense as Landin's earlier papers. It does not design a new programming language. Instead, it aims to explain two constructs used in programming languages, recursion and types, using the lambda calculus as a model. In a chapter on recursion, Morris relates recursion in programming languages to fixed point combinators in the lambda calculus. In a chapter on types, Morris presents a simple type system for a lambda calculus extended with operations for basic arithmetic. The extent to which this was directly inspired by Church's simply typed lambda calculus is not clear. Morris does not make an explicit reference to it and only cites the untyped lambda calculus and a review of combinatory logic that mentions types, but only in passing.

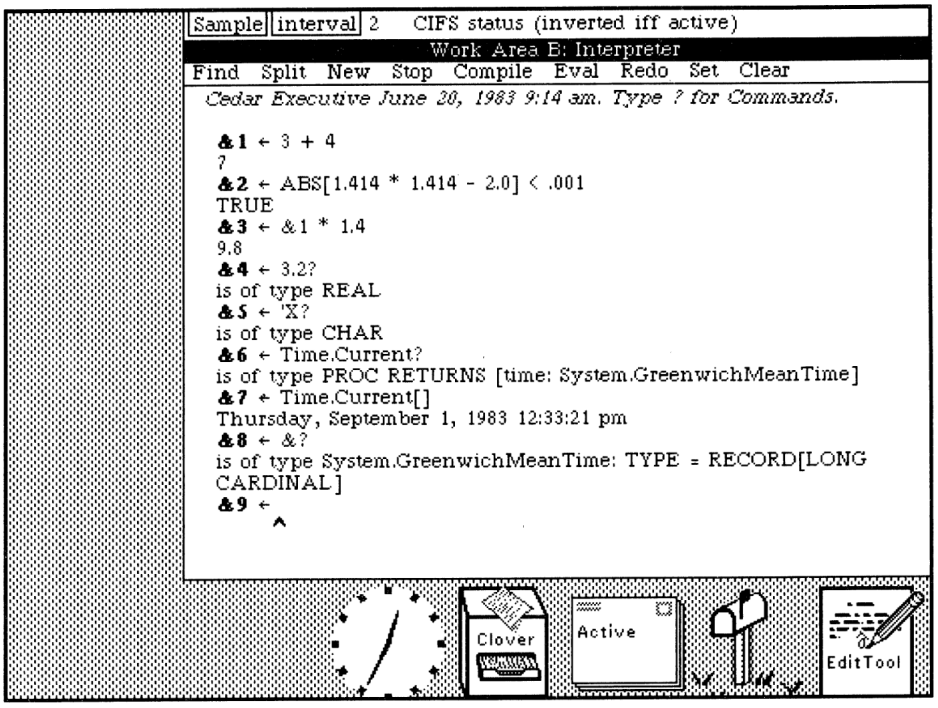


Figure 5.8: The Cedar programming environment developed at Xerox PARC.

Nevertheless, the material in the chapter on types would be familiar to a contemporary programming language researcher. Morris proves "sufficiency" of the type system, which guarantees that a well-typed program does not result in `ERROR`, a special kind of value produced by invalid program operation. Morris also presents an algorithm for type assignment, which finds a type for a program without explicit type annotations. The structure of his type assignment algorithm is similar to that of the later Hindley-Milner algorithm from 1978, but simpler, because Morris' type system is not polymorphic. In a brief final section of the chapter, Morris extends his language with pairs and unions, inspired by McCarthy's data spaces and Algol 68 types. He also adopts the now familiar notation of $\mathtt{T}_1 \times \mathtt{T}_2$ and $\mathtt{T}_1 + \mathtt{T}_2$ for pairs and unions, respectively.

Morris understands types in a way. Despite making a reference to McCarthy's set-theoretic model of types when talking about pairs and unions, his view of types is closer to that of Russell in the context of formal logic. Morris does not discuss what types are or how they should be modelled. Types are just a formal construction for identifying invalid programs. This new way of thinking about types in programming languages is clearly captured by a paper "Types are not sets"[63] that Morris wrote four years after completing his PhD thesis, while working at Xerox PARC. The paper was shaped by his PhD research, but also by his experience working on the Modular Programming Language (MPL), a predecessor of Mesa, that used types for program structuring. In other words, Morris brings together the mathematical perspective, derived from the lambda calculus with an engineering perspective on what types in programming languages are good for in the context of modular programming.

Morris first recognises the dominant thinking about types in the mathematical culture, which focuses on what types are, noting that "there has been a natural tendency to look to mathematics for a consistent, precise notion of what types are. The point of view

there is extensional: a type is a subset of the universe of values. While this approach may have served its purpose quite adequately in mathematics, defining programming language types in this way ignores some vital ideas." The vital ideas arise not from thinking about what types are, but what types can be used for: "rather than worry about what types are I shall focus on the role of type checking. Type checking seems to serve two distinct purposes: authentication and secrecy."

The problem of *authentication* is to guarantee that only correct kind of values are admitted for processing. Type checking can, for example, ensure that only Boolean values (true and false) are passed to an operation that expects a Boolean. Thinking of types as sets addresses this problem well. We can see types as a mechanism for ensuring that only an appropriate subset of values is passed to an operation. The problem of *secrecy* is to ensures that "only the [given] procedures can be applied to objects of [a certain] type." In other words, type checking should ensure that programs do not unnecessarily depend on the particular representation that a programmer chooses for their objects. Such representation should be hidden from most of the program and it should be visible only to a small number of privileged procedures that are used by the rest of the program and can be easily modified in case the representation needs to change.

The second problem put forward by Morris is the use of types for what we might now call *information hiding*. Information hiding appeared in the engineering culture in the 1970s as an appeoarch to modular programming. It was described by David L. Parnas, first in a technical report in 1971 and later in an influential paper "On the Criteria To Be Used in Decomposing Systems into Modules".[64] In the later paper, Parnas proposes a way of structuring programs into independent modules that expose a limited number of operations and hide the representation of data they use. This makes a system easier to change and understand and it also enables independent development of individual components. Morris has been thinking of protection mechanism that could be used to ensure information hiding in programming languages and published a paper on this topic[65] less than a year before "Types are not sets". Morris does not explicitly cite Parnas in either of the papers, but he clearly suggests to use types and type checking to ensure correct hiding of information in a modular programming language. Interestingly, Morris first describes a dynamic type system that seals and unseals data into or from opaque representations at runtime. He only later notes that some of those checks could be done statically, i.e. before the program is executed.

The notion of type fully acquired its new information hiding sense when Barbara Liskov and Stephen Zilles introduced the concept of "abstract data types"[66] and described its implementation in the Clu programming language. Liskov and Zilles use the term "type" to relate the mechanism to built-in types in programming languages. An abstract data type is a mechanism for introducing abstract values that are defined by the operations they support rather than by their representation. Liskov and Zilles marry the engineering motivation of Parnas with the use of types advocated by Morris and they reference both of the strands of prior work. In Clu, a programmer can define an abstract data type, which is an abstract description of a module in terms of its public operations, and then implements it using a structure called a *cluster*. As in the work of Morris, the abstraction is, in part, enforced by dynamic checks at runtime. However, the paper also cites personal communication with John C. Reynolds, noting that his recent work "indicates that complete compile-time type checking may be possible."

The mention of Reynolds was likely referring to his work that appeared a year later in a paper "Towards a Theory of Type Structure".[67] Similarly to the work done by Morris in his PhD thesis, the paper brings together the development of types in programming with types developed in the logic tradition. Both Morris and Reynolds use a method that many theoretical programming language papers will use over the following 40 years. They present a simple extension of the typed λ-calculus and then prove that the extension has a certain desirable property.

In case of Morris, the property is that a well-typed program evaluates to a value, which is a precursor of the type soundness property that will become widespread in follow-up work over the. next several decades. The paper by Reynolds proves a formal property that he calls the "representation theorem". It states that the value obtained by evaluating a program does not "depend upon the particular representations used to implement its primitive types." In other words, Reynolds talks about information hiding, rather than about values produced by well-typed programs as is the case in Morris' work and in much of the present-day theoretical programming language literature. While Reynold's style and contribution is strongly rooted in the mathematical culture of programming, but the program property that he proves originates in the engineering culture of programming.

In Search of Types

The story of types told so far is that of a concept shaped by diverse commercial needs, theoretical insights and practical constraints. Different programming languages used types in different ways and different cultures of programming approached them from different perspectives. It may even seem that different authors and different languages are really talking about different things. The term 'type' became only popular over time. The precursor of types in FORTRAN was called 'mode', COBOL defined the structure of data in terms of files and records and the early mathematical theory proposed by McCarthy referred to 'data spaces'. In later works, most authors talk about types, but they often further refine the terminology. Algol 68 distinguishes between primitive 'types' and composed 'modes' while the programming language feature to support modular programming was not called just a 'type' but consistently used the qualified term 'abstract data type'.

Yet, treating the different notions of type as different things would hide interesting interactions between different cultures of programming that shaped them. We saw two notable influences across the cultural boundaries so far. First, the idea of record types for structuring data in data processing appeared in the context of commercial computing that I associate with the managerial culture of programming. A record simply moved from a filling cabinet into an electronic computer system. The need for richer data structuring was soon recognized by other programmers, leading first to mathematical models of records and later to engineering work on supporting records in concrete programming languages including Algol W, Algol 68 and Pascal. The second notable influence across the cultural boundaries proceeds in the opposite direction. Much of the work on types and lambda calculus was done in logic, even before computers existed. Thanks to Christopher Strachey, Peter Landin and Mervyn Pragnell's underground reading group, those ideas entered the mathematical culture of programming and influenced James Morris who brought the ideas along when moving from the mathematical culture of his PhD research to the engineering culture at the Xerox team working on modular programming languages.

The story of types is not unlike the story of the definition of a polyhedra told by Lakatos,[68] but the forces that shape a concept in programming are even more numerous. The main forces that shape the definition of a mathematical definition are proofs and counter-examples. In the case of types and programming concepts more generally, the forces originate from the corresponding mathematical models, but also include practical commercial requirements and implementation constraints.

Even though the history is tangled, there are two different emerging themes in how types can be defined and used. The two themes have been identified by Stephen Kell.[69] Interestingly, Kell's divide is orthogonal to the boundaries between different cultures of programming. On the one hand, we have "data types" that primarily exist to structure data, define the representation of data in a computer and ensure that it is accessed correctly. Data types exist in COBOL in the form of records, they can be mathematically modelled as sets of values and are supported in programming languages through concrete mechanisms for defining records, unions or variant records in Pascal. On the other hand, we have "types" as known from logic. The mathematical models of those refrain from stating what types are and instead focus on specifying how to check types. This perspective is derived from work on formal logic and was eventually adopted by Morris and became particularly useful for thinking about information hiding and abstract data types. The distinction between "data types" and logical "types" is something that many computer scientists glance over. However, to some, it was very clear as early as 1960s. In a note about logical types that will play an important role in the next section, first circulated informally in 1969, Dana Scott writes "the first confusion we should avoid is that between *logical* types and what we might call *data* types."[70]

It may seem that, in the 1970s, the notion of a type was bound to diverge into multiple different concepts. There were many differences between how individual cultures thought about and used types. Types in programming languages for business data processing were quite different from types in scientific programming languages. There was also the divide between "data types" and "logical types". Yet and against all odds the opposite happened.

The Definition of Standard Types

Robin Milner was another influential British computer scientist who attended at least one or two of Mervyn Pragnell's underground reading group meetings. As many others in the group, Milner was interested in semantics of programming languages, i.e. using formal mathematical methods for specifying the meaning of programs. After graduating from Cambridge, Milner spent the late 1960s teaching at City University in London and as a research assistant in Swansea. During this time, he got interested in the verification and automated theorem proving. As he recalled later, he was particularly inspired by the double relationship between "the idea of a machine proving theorems in logic, and the idea of using logic to understand what a machine was doing."[71]

This interest led Milner to work on various forms of machine-assisted theorem proving. He is well-known for the development of the interactive LCF theorem prover that I already briefly mentioned in Chapter 3 when talking about formal proofs in computer science.[72] However, when he moved to Swansea, Milner first tried to create a fully automatic theorem prover. He soon "became shattered with the difficulty of doing anything interesting in that direction" and, consequently, became "more interested in amplifying human intelligence

than (...) in artificial intelligence."[73] Milner got a chance to shift his focus when he joined John McCarthy in Stanford in 1971-73 when two interesting developments came together.

The first development was practical. Prior to LCF, most work on theorem provers focused on automatic systems. This included the Boyer–Moore theorem prover, also mentioned in Chapter 3, that was developed in the early 1970s in Edinburgh. In those systems, you would specify a theorem, run the system and wait until it produced a proof or (more often) exhausted computer resources. As Milner soon realised finding proofs automatically was very hard. It was also the only thing you could reasonably try to do using a computer that was operated in the batch-processing mode, by handing a stack of punched cards to an operator and getting a result the next day. Time-sharing systems that I discussed in Chapter 4 appeared at the end of the 1960s and allowed a more interactive use of computers. Users could connect to a computer via a terminal, enter commands and get results immediately. This also made it possible to build interactive theorem provers.

The second development was theoretical. The aforementioned privately circulated note by Dana Scott[74] introduced a formalism for reasoning about recursively defined functions. The formalism made it possible to prove equivalence of partially defined functions, which are structures that often occure in work on programming language semantics. Milner referred to the formalism as Logic of Computable Functions (LCF) and used it as the basis of his theorem prover of the same name.

The goal of the LCF prover was to assist with proofs about programs and programming languages using the formalism developed by Dana Scott. A typical motivating example for the work was proving the correctness of a compilation algorithm.[75] The process of proving was interactive, meaning that you first entered some definitions and a goal, which describes the theorem that you wanted to prove. You then issued commands to transform the definitions and goals. Those include splitting the goal into sub-goals, applying rewrite rules on the definitions, or storing proved theorems into a library for later use. Although LCF was a proof checker rather than an automatic theorem prover, it included an automatic simplification algorithm that made completing proofs somewhat easier.

Constructing proofs in Stanford LCF was still a tedious process. One reason was that the range of commands that one could use to transform definitions and goals was fixed and limited. After moving to Edinburgh in 1973, Milner and his group started working on a "metalanguage" that would make using LCF easier by allowing users to create their own proof construction operations. ML was a functional programming language that made it possible to write scripts that construct LCF proofs by manipulating the terms of the underlying object language. The design of ML was inspired by a number of programming languages, including LISP, which was used to implement Stanford LCF and Landin's ISWIM.

To get a sense of how ML was used in the Edinburgh LCF, we can look at a brief example from a paper accompanying an invited lecture at the Mathematical Foundations of Computer Science conference in 1979.[76] The excerpt in Figure 5.9 proves that $X = X \vee X$ in a Boolean logic theory defined earlier in the paper. The code constructs a term `t` that we want to prove. It then fetches two different collections of axioms, `s1` and `s2`, and simplifies the original term using the built-in `simpterm` function using those two different sets of axioms. The theorems assigned to `th1` and `th2` after those operations are `t==X` and `t==X+X` and the final line uses built-in symmetry and transitivity to construct a theorem `X==X+X` represented by `th`. It is worth stressing that ML was not restricted to simple lists of instructions, but also allowed users to define their own functions, or *tactics*, that implement

```
let  t = "X + (X * ( ¬ X))"  ;;
let  s1 = ss(map (AXIOM `BA`)[`andinv` ; `oride`] );;
let  t1,th1 = simpterm  s1  t ;;
let  s2 = ss (map(AXIOM `BA`) [`ordist` ; `orinv`; `andide`] );;
let  t2,th2 = simpterm  s2  t ;;
let  th = TRANS(SYM th1, th2) ;;
```

Figure 5.9: ML script that derives the proof that $X = X \vee X$ (writing + for logical disjunction and $*$ for conjunction) in a Boolean algebra.[77]

different strategies of searching for proofs. To a contemporary functional programmer, the tactics would appear very much like combinators of a functional domain-specific language.

The challenge for the design of ML was to make it easy to write different strategies for constructing proofs, but prevent users from accidentally constructing non-theorems. To achieve this, ML represents a theorem as an abstract data type `thm`. The only pre-defined theorem values are axioms of a theory and the only operations that a programmer can use to create new theorems are the inference rules of the theory. This means that an ML program can only ever produce valid theorems. The result may not be the theorem you wanted, but it will never be a non-theorem.

Unlike earlier languages such as Clu that initially relied on runtime checks, the correct use of types in ML was checked by a static type system. Using the terminology of James Morris, types in ML ensure the secrecy of abstract data types. There is no way the ML code can inspect the structure of the `thm` type and create a new value from scratch. The ML type system[78] is interesting for three reasons. First, the type system is polymorphic, meaning that types of operations can include type variables. For example, the type of a function that returns the first element of a list in ML is α `list` $\rightarrow \alpha$. Here, α is a type variable indicating that the operation can work on lists containing any elements. The function can be applied to, for example, a list of theorems `thm list` and it will return the first theorem `thm` from the list. Second, the system does not require users to write any explicit type information. It infers a type automatically, based on the implementation. The algorithm is reminiscent of that described by Morris[79] but it is more sophisticated as it also handles polymorphism. Finally, the paper also formally proves the soundess of the type system, which is captured by the slogan "well-typed programs cannot go wrong". Milner defines how programs evaluate. Invalid operations produce special value, written as "wrong" and the proof guarantees that well-typed programs never produce this value.

The ML in Edinburgh LCF supported functions, tuples and unions, but lacked richer mechanisms for working with data, including records and discriminated unions. Those were first added in the HOPE language that was based on ML and later in an implementation of ML known as "ML under VMS" after the VAX/VMS system that it run on. The "ML under VMS" system was implemented by Luca Cardelli, who was working on his PhD thesis, focused on hardware verification, at the same time. Despite being developed by an academic computer scientist, ML under VMS had some notable characteristics of a hacker culuture. First, it generated VAX machine code and was significantly faster than the original LISP implementation. Second, the main early source of information about the project

Figure 5.10: The first volume of the Polymorphism newsletter contained a brief report on the November 1982 meeting and a range of contributed articles.[81]

was a plain text file "mlchanges.doc", included with the distribution. The "ML unde VMS" was also later ported to other platforms, most notably UNIX.[80]

By 1982, there were a number of incompatible versions of ML, including the original LCF/ML, HOPE, "ML under VMS" and also "ML under UNIX". In November 1982, a meeting hosted by Science and Engineering Research Council (SERC) brought together 20 programming language researchers interested in ML. The SERC organizers recognized that ML family of languages is becoming a centre of research activity and were keen to promote collaboration. As reported in the first issue of the Polymorphism newsletter (Figure 5.10), the meeting included a discussion on the differences between ML and HOPE. The attendees could not yet envisage the ultimate functional language and believe that "variety and experimentation, rather than standardization, were needed at this stage."[82] Yet, standardization is exactly what happened over a series of meetings in the early 1980s and by 1985, informal drafts for "Standard ML" were already circulated among the group and the efforts focused on producing more formal mathematical definition of the language. This eventually resulted in "The Definition of Standard ML",[83] published by the MIT Press.

The ML language originated from the mathematical culture of programming. Many of the contributors were, at some point, affiliated with Pragnell's logic reading group, the language itself emerged from work on a proof assistant and the ML definition is written in a formal mathematical language. However, the ML type system brings together ideas from many different cultures. It uses types to track how values are stored in memory and uses those for efficient compilation, in a way that dates back to the hacker notion of mode in FORTRAN. It suppers data types such as records and discriminated unions, which are descendantes of records in FLOWMATIC and were further developed at the intersection of engineering and mathematical cultures. The abstract data type `thm` from the original LCF/ML evolved into a sophisticated support for modules that make it possible to define custom abstract data types, which links the ML notion of types to the engineering tradition and work on information hiding.

The notion of types in the ML language finally brings together the "data type" notion that emerged from the needs to represent different kinds of data with the notion of "type" that appeared in mathematical logic, but existed largely independently of programming languages until 1970s. It would be tempting to use Kuhnian perspective on scientific revolutions and see the era before the definition of types in the ML langauge as pre-scientific and the great unification of ideas on types that happened with ML as the birth of a normal science. This would not be entirely unjustified: there was no consensus on a particular the-

ory of types in the era before ML and the birth of the ML paradigm defined a set of methods and the kinds of questions that they can answer. The ML paradigm became hugely influential and has a large following in both the mathematical and the engineering cultures of programming. Yet, the definition of types in ML is not the end of the story and we will soon see the concept developing in multiple different directions, under the influence of the engineering culture and the logical tradition of the mathematical culture. Before that, I briefly recount the line of work on types that emerged from ML.

Types as a Lightweight Formal Method

The ML approach to types has two characteristics that enable it to work as a research paradigm: it is both clearly delineated and sufficiently open-ended. On the one hand, the definition of Standard ML sets out a way of working with types. Types should be checked at compile-time and should prevent runtime errors. They should be defined formally using a particular mathematical language. The definition of Standard ML shows how to do so conveniently using inference rules. A programming language with type checking should make guarantees about the execution of well-typed programs, captured by the slogan "well-typed programs do not go wrong". On the other hand, the phrase "do not go wrong" does not say what kind of wrong behaviors should be eradicated using types. The mathematical style of Standard ML, based on inference rules, is relatively easy to follow in work that captures different uses of types. These two characteristics of ML types allowed a large number of theoretical computer scientists to contribute new notions of type, guaranteeing a wide range of properties about programs.

The research paradigm defined by the ML notion of types caters primarily to the mathematical culture of programming. It shifts focus from issues such as implementation or commercial applications to theoretical work. There is an implicit assumption that the theoretical work is the first step towards developing practical programming languages. The paradigm is perhaps best captured in an introduction of a textbook for advanced undergraduate and graduate students "Types and Programming Languages":[84]

> *Modern software engineering recognizes a broad range of formal methods for helping ensure that a system behaves correctly with respect to some specification (…) of its desired behavior. On one end of the spectrum are powerful frameworks [that] can be used to express very general correctness properties but are often cumbersome to use (…). At the other end are techniques of much more modest power—modest enough that automatic checkers can be built into compilers (…) and thus be applied even by programmers unfamiliar with the underlying theories. (…) [By] far the most popular and best established lightweightformal methods are type systems, the central focus of this book.*

The quote positions the concept of a type unequivocally in the context of formal mathematical approach to programming. In doing so, it blatantly ignores the rich multi-cultural history of the concept. The introduction leaves no doubt that there should be a specification of program behaviour and that we need a formal mathematical method to ensure that an implementation conforms to this specification. In other words, the introduction defines what should be done with types within the ML paradigm.

Thinking about types as a lightweight formal method also forces us to rethink a number of typical assumptions about specifications. First, types are typically written by the same group of programmers who write the implementation. This is in contrast with conventional specifications, which are written by analysts and approved by the managers. Second, checking that an implementation follows a specification is no longer a human activity, as for example in the Cleanroom methodology that I discussed in Chapter 3. Instead, it is a task for an automatic type checker.

In this way, the ML tradition of types provides a new perspective on the critique by DeMillo, Lipton and Perlis that formal proofs about programs lack social processes. Social processes involving types are not centered around the question of proof checking, i.e. verifying that an implementation matches a specification. The type checker does that automatically. However, programmers actively discuss types in at least two ways. First, they talk about types of common functions and what they mean. For example, functional programmers will happily spend a lot of time explaining what a library function does by discussing the details of its type. This is especially the case for very general operations like `map` with polymorphic types like $\texttt{List}\ a \to (a \to b) \to \texttt{List}\ b$. Close reading of such type often provides good enough understanding of what the operation may be doing. Here, the function gets a list of values of type a and a function that knows how to turn values of type a into values of type b. The function produces a list of values of type b. The only way to do this is to go over all the values in the input list, apply the given function to each element and collect the results which is, in fact, what the function does. Second, programmers often ask their peers "why is this code not type checking?" when attempting to make sense of a programming error. They then attempt to follow the logic of the type checker and see where their intuitive understanding of what code does fails to match with what the types say, especially in languages where the types are inferred automatically. Both of these provide important social processes, which ensure that the use of types as a lightweight formal method is not just a black-box that labels programs as 'correct', but a process with a human element. Types bring back human checking and intuition.

Comprehensively documenting the work on types in the ML paradigm is a topic for another publication, but it is worth discussing a few directions. A large body of work within the ML tradition of types in late 1980s and 1990s focused on using types to provide more guarantees about program execution. Types in ML ensure that abstractions are not violated and that expressions evaluate to a value of a right type, but there are many other ways in which a program can go wrong. In particular, many interesting wrong things can happen in a programming language where different parts of a program can write to and read from a shared memory or in programs that access external resources such as the file system or network. Type and effect systems[85] annotate type information with an *effect* that the evaluation of an expression has. They were first used to avoid issues arising from the use of shared memory, but were later also used to check the correct use of resources such as files. Effects can also be used to manage program memory more efficiently. Functional programming languages like ML and LISP typically use garbage collector, which automatically looks for and releases data that is no longer needed. Region-based memory management[86] made it possible to free the memory used by a program more efficiently, while ownership types[87] address issues that occur when the state of an object in an object-oriented langauge is changed by multiple other objects. All this work subscribes to the ML notion of types and belongs to the mathematical culture of programming. As is often the case in programming, the ideas found their way to other cultures. The perfect example of

this is the Rust programming language, which has engineering origins and ethos. Rust was first announced in 2010 and uses ownership and advanced memory management techniques as a basis for the design of a safe systems programming language.

Rust is, however, a somewhat unique case. It adopts two specific techniques for solving engineering issues around memory safety and efficiency. For the Rust designers, this issue is important enogh to warrant direct support in the programming language. Many other extensions developed within the realm of the ML paradigm of types are even more single-purpose. This makes them easy to define using the language of the mathematical culture, but it makes it hard to justify their implementation in an idustrial-scale programming language arising from the engineering culture. As the types got more complex, it also became harder to keep the usability afforded by the type inference, which frees users from having to write types explicitly, and also to keep the simplicity of type checking as implemented in Standard ML. The type system may require more annotations or explicit type specifications, it may become intractable and, at the very least, it becomes harder to understand and use.

Another direction has been taken by the Haskell language which appeared in 1990. Haskell was initially motivated less by a specific approach to types and more by its evaluation strategy. It uses lazy evaluation, meaning that arguments of a function are not evaluated before the function call, but only when they are actually used. In late 1980s, there was a large number of experimental functional languages featuring lazy evaluation and Haskell was a community effort to introduce a common more widely-used language. An explicit goal of Haskell was to design language that could be extended, modified and used for research into language features.

The initial design of Haskell introduced "type classes" which can be used to specify that a function argument is of any type that supports certain operations. For example, a function `sqr` that calculates a square of a number has a type `Num a => (a -> a)`. This specifies that the function works on any type `a` as long as that type is numerical and supports numerical operators such as multiplication. Type classes solved a relatively small problem at first, but they inspired further work on types in Haskell. The Haskell designers later described Haskell as a "type-systems laboratory" and recalled that:[88]

> *An entirely unforeseen development—perhaps encouraged by type classes is that Haskell has become a kind of laboratory in which numerous type-system extensions have been designed, implemented, and applied. Examples include polymorphic recursion, higher-kinded quantification, higher-rank types, lexically scoped type variables, generic programming, template meta-programming, and more besides.*

The list of examples would be even longer if it included extensions implemented after 2007 when the quote was written. Although Haskell originated with the mathematical culture of programming and most such extensions are formally described in an academic paper, it adopts a different approach to types than the research projects born from the ML paradigm. Haskell includes a family of general purpose type-level mechanisms that can be used to express more detailed specifications using types. Using those often requires what I would classify as unwritten knowledge of the hacker culture. Expressing various constraints about code often involves tricks that are not well documented, but are known to the community of Haskell users.[89]

There are many kinds of constraints that can be specified in Haskell through such tricks, but types in Haskell are not used for writing fully general specifications.[90] The language, however, led the way to a more general notion of type that does make it possible to specify arbitrary constraint using types. In order to document this idea, we need to return to the early days of interactive theorem proving, before the appearence of ML.

Automating Mathematics Using Types

As we saw earlier, the ML language, which defined a new paradigm for types in programming languages in the 1970s, has its origins in the work on the LCF interactive theorem prover. Curiously enough, later work on interactive theorem proving, which evolved in parallel with the work on the ML language, reshaped the notion of types in programming languages in the early 2000s.

To document this, we need to return to another early interactive theorem prover. The system was Automath, developed by Nicolaas Govert de Bruijn in 1967.[91] Automath was not widely publicized, but it was not entirely unknown. Interestingly, Donald Knuth mentions Automath in his review of Hoare's chapter in "Structured Programming",[92] and notes that Automath is worth looking at because it "is the epitome of the concept of type." Automath uses the logical notion of type in an even more profound way than the work of Morris on lambda calculus or the work on the ML programming language.[93]

At the core of Automath is the remarkable mathematical equvialence between types and logical propositions that Philip Wadler referred to in the presentation that I talked about in the opening section of this chapter. This relationship is now known as the Curry-Howard correspondence after Haskell B. Curry who observed an early version of the idea in 1934 and William A. Howard, who described the modern version of the idea in 1969, independently of de Bruijn who was already using it in Automath.

The Curry-Howard correspondence captures a relationship between types in the lambda calculus and propositions in intuitionistic propositional logic. The key idea is that logical propositions can be understood as types. To prove a proposition, we need to show that there is a value of the required type (the type is inhabited). For example, let's say that we want to prove that $P \rightarrow (Q \rightarrow P)$. This is not a surprising result, but it will serve well as an example. The proposition states that P implies that $Q \rightarrow P$, in other words, it follows from P that Q implies P. This is true, because if P holds then $Q \rightarrow P$ will necessarily also hold ($Q \rightarrow P$ could only be false if Q was true and P false, but we already know that P is true).

Using the Curry-Howard correspondence, we can see this proposition as a type of a function $P \rightarrow (Q \rightarrow P)$ where P and Q are some arbitrary types. In modern programming terminology, the function is generic (or polymorphic) with two type parameters. This is a function that takes a value of type P and returns a function taking a value of type Q and returning a value of type P. This is easy to define! Using the lambda notation, we can write it as $\lambda x.\lambda y.x$. The same can be written in a programming language like JavaScript as:

```
function(x) {
  return function(y) { return x; }
}
```

The type $P \rightarrow (Q \rightarrow P)$ is inhabited by the above function and so the proposition holds! The correspondence also helps us understand why proving $P \rightarrow Q$ is impossible. We

would need to construct a function that somehow produces a value of an arbitrary type Q using just a value of type P. We do not know anything about those two types, aside from having one value of type P and so there is no way of constructing a value of type Q. The correspondence also works for conjunction and disjunction, which map to pairs of values and union types, respectively. This was already used in Automath and it means that, for example, a logical proposition $P \wedge Q \rightarrow P$ corresponds to a function that takes a pair of P and Q and returns the first component of the pair.

More interestingly, the correspondence can also be extended to quantificators of predicate logic. This idea was developed in systems that followed Automath and I will get to them in the next section. The types corresponding to universal and existential quantification are more subtle than those for functions, pairs and unions and are known as dependent types. A type corresponding to universal quantification is the dependent function type, written as $\Pi_{x:A}B(x)$. This is a type of function whose return type depends on the value of the argument. Given a value x of type A, the return type $B(x)$ is a type obtained based on the value x. The complexity introduced here is that $B(x)$ is itself an expression that calculates the resulting type and needs to be evaluated during type checking. A type corresponding to existential quantification is known as the dependent pair $\Sigma_{x:A}B(x)$ and it is a pair of values (x, y) where the type of the second value depends on the first value. The key idea is that types are no longer just simple types like integers, records or lists of integers, but can include computations that produce types based on values. This makes type checking difficult, but as we will soon see, it has interesting practical applications.

The Curry-Howard correspondence shows that types in functional programming languages are related to propositional logic, but does it justify saying that functional languages are discovered rather than invented as suggested by Philip Wadler in the talk discussed in the opening of this chapter? Mathemtaically-minded computer scientists who accept this argument do so based on a number of assumptions that are typical for the mathematical culture of programming. The first is the unificantion of complex engineered software systems, such as programming languages, with their mathematical models. The implementation of types in an actual compiler is always more complex and involved than the formal model given on paper. Programming languages often also have various pragmatic features that are omitted "for simplicity" from their models that further complicate their formal properties.[94]

The second assumption of the mathematical culture is that it focuses on current mathematical knowledge, but ignores the broader historical perspective. Curry-Howard correspondence relies on a very carefully constructed matching pair of a logic and a type theory that has been obtained by an iterative process of refinements.[95] Moreover, both predicate logic and the lambda calculus arose from the same community of early 20$^{\text{th}}$ century logic, working on the same kinds of issues in the foundations of mathematics. Given this context, the correspondence between types and logic is perhaps less remarkable on its own. The correspondence did, nevertheless, had a remarkable influence on programming languages and types.

Programming in Type Theories

A number of interactive theorem provers were more or less directly inspired by Automath and used the idea of Curry-Howard correspondence as their basis. As explained in the

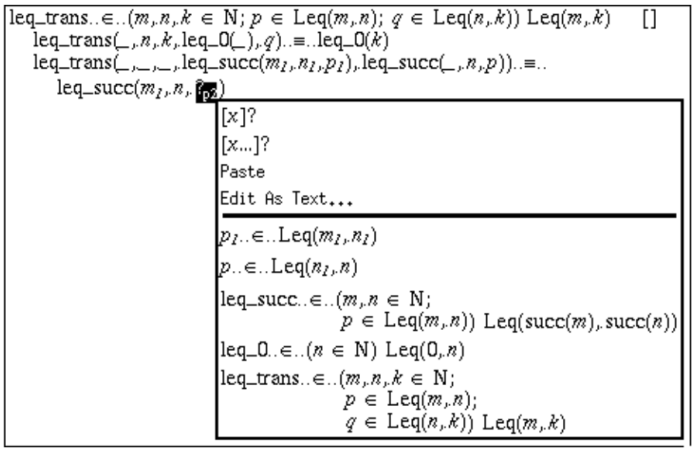


Figure 5.11: Constructing the proof of transitivity of the less than or equal ($\leq$) relation on natural numbers in the interactive theorem prover ALF.[96]

previous section, creating a proof in a theorem prover based on the Curry-Howard correspondence is done by constructing a program in a language derived from the lambda calculus. This may perhaps not be obvious when one looks at an example such as that of the ALF theorem prover in Figure 5.11 and so it took some time until the wider programming language community realised that the same style of working could also be used for constructing programs.[97] The change in thinking happened at the turn of the millenium, mainly in Sweden and France.

In the 1970s, Per Martin-Löf introduced a type theory that included the dependent function and the dependent pair types discussed in the previous section. He developed the theory as part of his work on constructivist foundations of mathematics where existence proofs have to specify how to construct the mathematical entities they postulate. The Curry-Howard correspondence is a perfect fit. A constructive proof created using type theory can be seen as a program that produces the required mathematical object. Seeing such proofs as programs was, however, a later development. After visiting Martin-Löf in the early 1980s, Bengt Nordström from Chalmers started writing a book, "Programming in Martin-Löf's Type Theory". The book envisioned a way of using the type theory as a programming language:[98]

> *[Per Martin-Löf's type theory] is well suited as a theory for program construction since it is possible to express both specifications and programs within the same formalism. Furthermore, the proof rules can be used to derive a correct program from a specification as well as to verify that a given program has a certain property.*

The introduction alludes to the idea that one can write a precise specification as a proposition and, by constructing a proof of the proposition using the type thory, derive a program that implements the specification. Despite having the word 'programming' in the title, the book was more an introduction to type theory with references to functional programming than a book about programming in the conventional sense. The book's example programs, the most sophisticated one being sorting of a sequence of coloured objects, can convince only devoted members of the mathematical culture, but the book inspired the development of a family of theorem provers that eventually brought theorem proving closer to programming, starting with the aforementioned ALF theorem prover,[99] developed in Chalmers in the early 1990s.

```
PrintfType :: String -> #
PrintfType "" = String
PrintfType ('%':'d':cs) = Int -> PrintfType cs
PrintfType ('%':'s':cs) = String -> PrintfType cs
PrintfType ('%': _ :cs) = PrintfType cs
PrintfType ( _ :cs) = Printf Type cs

printf :: (fmt::String) -> PrintfType fmt
```

Figure 5.12: The definition of `printf` function in Cayenne.[100] `PrintfType` is a function that takes a string and returns a type. It is recursively defined by pattern matching over the first characters of the string. For example, if the string starts with `%d`, the result will be a function type taking `Int` and returning whatever type is returned by recursively processing the rest of the list. The `printf` function itself (last line) takes a string `fmt` as an argument and its return type is `PrintfType fmt`, which depends on the value of the format string, given as the first argument.

At the same time, Thierry Coquand introduced the Calculus of Constructions, which was another type theory based on the Curry-Howard correspondence, and started working on the Coq theorem prover in Paris with Gerard Huet and Christine Paulin-Mohring. The work was inspired by Automath, as well as more theoretical developments around types and the lambda calculus. Both ALF and Coq were primarily theorem provers, even though they were clearly designed with program construction in mind. The authors of the Coq theorem prover started experimenting with using it more as a programming language already in early 1990s when they developed a mechanism for extracting Caml (a language based on ML) programs from Coq proofs.

At the end of the 1990s, two experimental programming languages attempted to bring dependent types into programming in a more conventional way. The langauges were Cayenne, created in Chalmers and Dependent ML, created at the Carnegie Mellon University.[101] Dependent ML took a conservative approach. It allowed natural numbers to appear in a type, as in the type `Vect` n α representing a vector of values of type α of length n. Cayenne supported unrestricted dependent types. A practical motivating example used by the authors was the `printfn` function. The type of the function depends on the value of the format string given as an argument (Figure 5.12). For example, the string format `"Hello %s!"` requires a `string` argument and so the type of `printfn "Hello %s!"` is a function function `string` $\rightarrow$ `string` whereas `printfn "%d"` returns a function `int` $\rightarrow$ `string`.

Both Cayenne and Dependent ML combined the idea of dependent types, which until then existed only in the mathematical culture world of theorem provers, with a practical programming language. As the name suggests, Dependent ML extended the ML language while Cayenne was implemented in and inspired by Haskell. Equally, both Dependent ML and Cayenne remained an early exploration of the idea of practical dependent types and were not yet adopted by a broader programming community.

The next wave of dependently typed programming languages appeared in the mid-2000s and included Epigram, Agda and Idris.[102] The latter two evolved into practical programming languages that have been used not just academics, but also by some practitioners and have been maintained ever since. At the same time, the Coq theorem prover was also further developed in ways that make it more usable as a programming language and

was used, for example, to develop a formally verified compiler for the C language called CompCert, a project that started in 2005 and is available under a commercial license.[103]

One interesting aspect of dependently typed programming languages is their interactive programming environment. This is perhaps unlikely. As we saw in Chapter 3, most work on interactive programming was rooted either in the hacker or the humanistic culture of programming. However, interactive theorem provers offer a new kind of mathematical perspective on programs. Rather than seeing programs as mathematical entities that should be proved correct, they see programs as matheamtical entities (proofs) that are interactively constructed.

This was not yet the case in Automath where the users would write "a book" and submit it to be checked.[104] This is perhaps not surprising, given that Automath was created before the emergence of time-sharing and personal computers. However, all later interactive theorem provers and dependently typed programming languages have an interactive editor that provides immediate feedback. There are two basic ways of interacting with a theorem prover. In the first style, the user starts with the desired type (theorem) and enters a sequence of commands that invoke various tacticts that attempt to construct the program. This approach is used, for example, in the Coq theorem prover and in the ALF editor shown in Figure 5.11. The tactics can be simple rewrite rules, but also more powerful tactics such as `auto`, which attempts to find a program automatically, using variables in scope (hypotheses) and other standard rewrite rules. This way of programming is akin to using a REPL. The programmer proceeds step by step, entering commands and observing their effects until they reach the desired goal.

In the second style, the user writes the desired type alongside with source code of a program in a functional language that has the desired type. This is more like writing code in ordinary functional languages, but with two caveats. First, the programmer starts with a type rather than, as done in ML, letting the compiler infer the type from code. This is because the type is the specification of the program we are creating or theorem that we are proving. Second, when writing code, the programmer is supported by an interactive editor that offers hints using clever auto-completion tools and also checks the program in background and reports mismatches with the desired type.

Naïvely, we could see dependent types as an evolutionary step in the work on types in the context of functional programming languages and the ML paradigm. After all, if we view types as a lightweight specification mechanism, then supporting richer types in order to write more precise specifications would be a natural direction. This might have been the motivation for Dependent ML, which was clearly rooted in the ML tradition, but the history is more complicated when it comes to types in languages like Coq, Agda and Idris. All of those reverse one of the basic principles of the ML tradition, that is the fact that types are inferred from code and are checked automatically. In contrast, in Coq, Agda and Idris, types are written up-front and programming is an interactive process in which the programmer constructs an implementation of the right type, supported by various interactive editing tools provided by the programming environment. Seeing them as just the next step of the ML paradigm would thus be wrong.

The development of Coq, Agda and Idris partly repeats the meeting of cultures that was necessary for the birth and popularization of the ML language. The three dependently typed languages have origins in the mathematical world of theorem proving. However, they combine this with engineering motivations. For example, an early motivation for Idris was systems programming, which motivated a sophisticated system for handling

side-effects. To support the interactive development of programs, the languages also required a suitable editing environment. In the early days, those were implemented as extensions to the infamous Emacs and vim text editors. Those are not just the products of the hacker culture, but the rivalry between them is a lasting part of the hacker culture. In other words, the latest generation of dependently programming languages is a product of another meeting of cultures, the fruits of which we are probably yet to fully see.

The Meaning of Types Is Their Use

In the mathematical culture of programming, the primary argument in favor of using types is that they can ensure correctness of programs. In the ML tradition, this was made explicit by the slogan "well-typed programs do not go wrong". However, this is just one thing that can be achieved using types. Even the proponents of the mathematical culture recognize usefulness of types for engineering reasons. The lecture notes for the theoretical "Types" course at University of Cambridge[105] open by documenting five uses of type systems: detecting errors, support for structuring large systems, documentation, efficiency and whole-language safety. The three reasons in the middle of the list are chiefly engineering ones. The engineering culture has found many practical ways of leveraging types in recent history, but it does so using its characteristic approach. That is, by building tools that assist with programming. Many such tools used types and, conversely, stretched the notion of types.

In parallel to the more mathematical line of work that I followed in this chapter, types were adopted by object-oriented programming languages, which I will discuss in depth in the next chapter. Types in object-oriented programming languages have been mainly influenced by the Algol tradition. The object-oriented language Simula used types for checking the correct use of objects and their members and became an inspiration for Bjarne Stroustrup who designed the widely used C++ language in 1979. He later acknowledged this influence, but also emphasized what the Simula tooling based on types enables:[106]

> *I acquired a great respect for the expressiveness of Simula's type system and (..) its compiler's ability to catch type errors. The observation was that a type error almost invariably reflected either a silly programming error or a conceptual flaw in the design.*

In the engineering culture, type information were used for efficient compilation of code, they served as an error checking tool, but they also soon became valuable for the development of sophisticated developer tooling. In the early 1990s, the short-lived Lucid Energize development environment for C++ pioneered numerous developer assistance features to help with code navigation and editing. Similar features were soon incorporated in many major developer tools for C++ (Figure 5.13) as well as for other programming languages.

The engineering culture found types a powerful mechanism for building developer tooling. Most notably, types were used for navigation and for automatic code-completion. When invoking a member of an object, the editor would automatically offer a list with the available members and programmers would be able to choose a member from a list instead of typing its name in full. In the 1990s, this use of types did not reshape the notion of a type in the programming language. With the growing importance of developer tools and the associated engineering perspective on types, engineering trade-offs started to influence the design of types in programming languages.

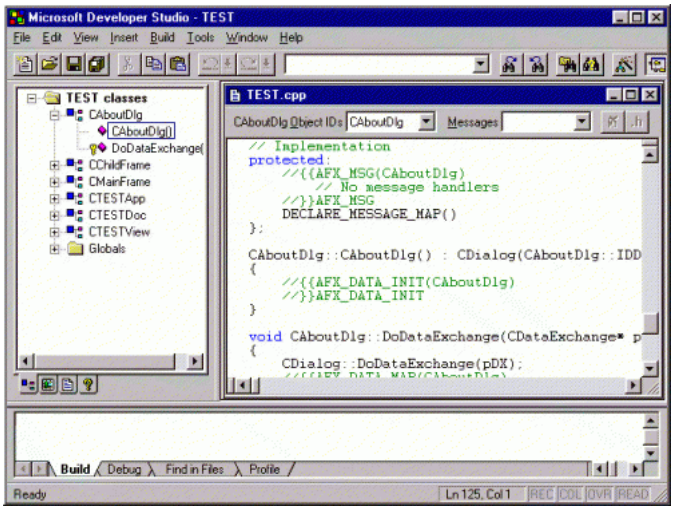


Figure 5.13: Microsoft Visual C++ 4.0, released in 1995, displayed a summary of available types in the "Class View" panel (left) and allowed programmers to navigate to individual class members.[107]

A prominent recent programming language that makes an engineering trade-off that would not be acceptable in the mathematical culture is TypeScript. The langauge was released by Microsoft in 2012 and its primary motivation is to enable better tooling for large-scale JavaScript development. TypeScript extends JavaScript with static type checking in a backwards compatible way. Moreover, the design of the TypeScript type system has been motivated by the patterns that appear in popular JavaScript libraries.[108] TypeScript also makes it possible to annotate an existing JavaScript library with types, through a separate definition file. This allows better developer experience, but the annotations are not (and can not) be checked, since the implementation is in plain JavaScript.

Annotations for external libraries are one source of potential runtime errors. It may be the one that is actually important in practice, but not the one that would worry the proponents of the mathematical culture. The TypeScript type system also intentionally makes a number of design decisions that make the type system unsound. This means that if a type checker determines that a variable has a certain type, this may not actually be the case when program is executed. As explained by a member of the TypeScript development team, "100% soundness is not a design goal. Soundness, usability, and complexity form a trade-off triangle. There's no such thing as a sound, simple, useful type system."[109] In other words, the creators of TypeScript intentionally choose a simpler but unsound type system design in places where soundness would introduce additional complexity.

A proponent of the mathematical culture would be sure to point out that there is nothing like 99% soundness. In theory, a type system either is sound, or it is not. In practice, a flaw that rarely leads to a problem is still a flaw, but one that is worth accepting if it allows the designers better fulfil other goals. Since TypeScript compiles to JavaScript, which performs full runtime checking, the lack of soundness does not lead to errors that would make the system unsafe. The TypeScript design follows a set of coherent design principles which are distinct from the principles that are favoured by the ML paradigm. The focus on tooling and producing simple useful documentation for programmers means that it makes sense to sacrifice soundness for simplicity. Yet, it again brings together a multiple perspectives developed by multiple cultures. The TypeScript type system is rooted in the engineering culture, with its focus on simple tooling. At the same time, it incorporates a number of type system concepts developed within the mathematical culture, including discriminated

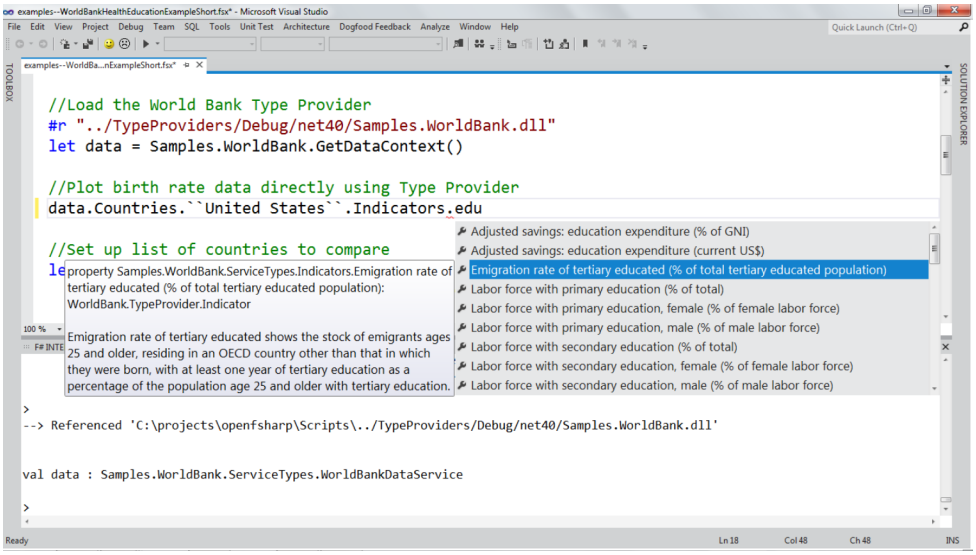


Figure 5.14: WorldBank type provider in F# in Visual Studio. The auto-completion lists shows generated members and allows the programmer to search for US education data.

union types from ML and even a very limited form of dependent types when that was necessary for tackling tricky patterns in existing JavaScript code.[110]

Another use of types motivated by engineering reasons that stretched the notion and also questioned the established notion of type safety were type providers (Figure 5.14), a feature added to the F# programming language in 2012. The F# language is a direct descendant of ML and is thus rooted in the mathematical tradition. It is also a commercially supported product for industrial software engineers and so it has an equally strong engineering influence.[111]

Type providers were motivated by one such practical engineering concern, which is access to external data. Programs that need to work with external data typically have no information, much less guarantees, about the structure of such data. We can trace this struggle to the very first use of types in programming languages. As we saw in Figure 5.4, the business-oriented language COMTRAN featured data description cards that were used to describe the structure of external data in a format understood by COMTRAN. Those descriptions were not unlike meta-data formats for describing the structure of external data in the early 2000s such as XML Schema and WSDL (Web Service Description Language). Unlike COMTRAN data description cards, the meta-data formats that emerged in the 2000s were not understood directly by programming languages.

Type providers were an attempt to get the programming language to understand the meta-data formats.[112] However, thanks to two technical aspects of their design, type providers also challenged the traditional thinking about types. First, type providers can generate types by running arbitrary code. This made it possible to find new unexpected uses of the mechanism and it made them popular among the F# members of the hacker culture.[113] Second, the types generated by type providers are generated on demand. The compiler only asks the type provider for a list of members and their respective types if the programmer attempts to access a member via the '.' operator. This makes it possible to create type providers that map external data into the type system of the langauge at a very fine-grained level. An example of this is the World Bank type provider (Figure 5.14), which generates

types with members for individual countries and development indicators from the World Bank database.

Type providers challenge thinking about types in two ways. First, they make it possible to use very specific types. Typically, "United States" would be a value of type "Country", but with a type provider, "United States" can be a type with only a single value. This shift in the meaning of types is not without a precedent. In the 1970s, the work on custom data types such as records and unions was done as part of the research on *extensible programming languages*[114] that aimed to allow users to extend a programming language so that it more closely matches the domain in which they work. Type providers share this motivation and are the next step in the move from arrays of floating-point numbers to custom data types such as records. Needless to say, formally how to formally define what is the meaning of a type such as "United States" raises some questions for the mathematical culture of programming.

The second challenge posed by type providers to the usual mathematical thinking about types is that the correctness of a program using type providers may depend on an external data source or, indirectly, the state of the real world. The traditional ML slogan that "well-typed programs do not go wrong" now includes an additonal caveat "as long as the world behaves reasonably". We can see this as an engineering interpretation of the mathematical idea of soundness. For the engineering culture, it is natural that programs rely on external resources that may fail. The fact that a program type checks is a useful reassurance, but not an absolute formal proof.

The engineering approach to types departs from the earlier work in a number of ways. It is not worried about formal properties of type systems. Types are useful as long as they have practical benefits and do not get in the way of programmers. The engineering culture is also not concerned with the nature of types and so it does not attempt to define what a type is. Types are not explained as sets or through their logical models. They are simply a mechanism used in the implementation of various tools. In other words, types are defined less by what they are, but more by how they are used in the programming systems and tools.[115]

Stalking the Elusive Type

The meaning of the notion of a type in programming languages shifted dramatically several times throughout its history, but it remained sufficiently stable that a working programmer nowadays can understand most of the different takes on types discussed in this chapter. Different cultures of programming refer to similar things when they talk about types, but think about them differently and look to different origins of the notion. For the hackers, types started as a useful trick to compile programs correctly; for the mathematicians, types import a powerful logical notion for ruling out invalid programs; and for engineers, types are a mechanism for program structuring.

Looking at the 70-year history of types in programming makes it clear that there is no simple answer to the question what is a type. The engineers and hackers do not seem worried about the lack of a definition and use or implement types in whatever way works. The mathematicians attempt to bring order into the chaos by providing definitions, but those do not stop others using types in yet another incompatible way. A type in Algol 58 was simply one of three pre-defined primitive types, but the idea that programming languages should be "extensible" and support data as known from business data processing soon

changed that and types became sets of values. The logical approach that became ubiquitous in the 1980s shifted the focus from what types represents to how they are checked. This have made it easier to develop new notions of types that do not fit the model of "types as sets of values", but it was also less satisfactory to some in the mathematical culture who believed that a type should denote a mathematical object such as a set.

An example of work that tried to redefine what types are emerged from work that extends ML types with units of measure.[116] This makes it possible to annotate numerical types with their physical units, so a function that calculates the square of a number would have a type $\texttt{int}\ u \to \texttt{int}\ u^2$. The function takes a number with any physical unit u and returns a number with the same unit squared. Calling it with a value in meters such as $5m$ results in $25m^2$. The type of this function indicates more than just the fact that it takes a number and returns a number. All functions f of type $\texttt{int}\ u \to \texttt{int}\ u^2$ have the property that $f(k * x) = k^2 * f(x)$ for any constant k. In other words, if we multiply the input by k, the result is multiplied by k squared. The question is, how can this information be captured by the mathematical entity representing the type?

The answer is to represent types not as sets, but as mathematical relations between the input and the output. A relation holds between only some elements of the input and the output sets and so it can, for example, capture the property defined by the units. The idea also proved useful for defining the meaning of types that track the effects that a computation may have. In type systems that track effects of computations, the type $\texttt{int}\ \&\ \{\texttt{read}\ \rho\}$ denotes a function that takes a number and produces a number, but it may also read from a memory region ρ. A mathematical relation can capture the fact that values in memory regions other than ρ cannot affect the result of the function.

This brief example illustrates how the notion of type evolves and how different cultures of programming can participate in such developments. The use of types for tracking units was motivated by engineering concerns. However, implementing the idea turned out to be at odds with the established mathematical models of types. This coutner-example[117] prompted the mathematically-minded computer scientists to revisit their definitions of what types are and develop a new formal theory. When the established scientific approach of modelling types as sets stopped working, when faced with new kinds of types, it firced computer scientists to reopen the black box and develop a new view.[118]

Pluralism and Scientific Progress

Types are shaped by a comlpex network of interactions between different cultures that interpret and use types differently but are, nevertheless, able to exchange ideas thanks to this shared notion and are also able to contribute new aspects to the notion. In a way, types and their concrete implementations in different languages provide a shared language through which the different cultures can communicate. For example, the mathematical culture first realised that types can be fully checked at compile-time. The implementors of abstract data types in the engineering culture used this to eliminate run-time checks and the hacker culture used the idea to improve the efficiency of compilers for languages with types.[119] Types can also play the role of a common language that different cultrues can share. For example, abstract mathematical ideas of category theory can be translated to type definitions, which engineers can understand and use for their own purposes.[120]

Types can also become a concept that requires the skills of more than just a single culture of programming. An example would be types as implemented in the programming

language Haskell. The language is rooted in the mathematical culture and its GHC compiler has become a lively playground for theoretical research on types. The theoretical ideas are, however, complemented with a serious engineering implementation. Furthermore, as the extensions to the GHC type system grew, using them started to require more than just theoretical understanding. The only way to learn how to use the variety of extensions is through practical experience, which is a typical attribute of the hacker culture.

The overall history of types is not linear and the developments do not follow a fixed method. Different cultures contribute their ideas, based on different approaches at different points in time. As a result, the development of types is perhaps best explained by the theory of epistemological anarchism.[121] This is not always appreciated by proponents of more rigorous cultures, who believe that having a clear definition would allow greater collaboration, but there are undeniable benefits of such strucutre. The notion of a type remains a living process and even a successful unification, such as the ML notion of a type, does not permanently freeze the notion. Talking about types in an integrated way that does not rely on a definition and can account for the mutually inconsistent notions of type used by different cultures can perhaps be best done by considering what can be done using types. If we can use types to produce more efficient compiled code, eliminate bugs or build developer tooling, then we obtained knowledge about types that is independent of a particular definition.[122]

Types in programming are then an inherently pluralistic concepts. In his reflection on types, Simone Martini genralizes the idea to the entire discipline of computer science:[123] "The crucial point, here and in most computer science applications of mathematical logic concepts and techniques, is that computer science never used ideological glasses (...), but exploited what it found useful for the design of more elegant, economical, usable artifacts. This eclecticism (or even anarchism, in the sense of epistemological theory) is one of the distinctive traits of the discipline, and one of the reasons of its success." The fact that the notion of 'type' is a multi-cultural mix of ideas that have never been fully integrated might well be the reason behind its success.

Notes

1. Backus et al. (1956)
2. Brown (1886)
3. Paraphrasing a letter to the editor of the ACM by Postley (1960), quoted by Ensmenger (2012), who discusses it in the context of broader tensions between the data processing industry and the academic computer science.
4. McCarthy (1961)
5. (1963)
6. The difficulty of modelling pointers has been pointed out by Priestley (2011). More information about the Address language can be found in Yushchenko (2022). In the mathematical culture in the West, the problem of pointers is dealt with by Hoare (1965) who proposes typed record references, but without discussing their interpretation in terms of sets.
7. Morris (1973b)
8. The concept of information hiding has been described by Parnas (1972) and its usefulness for management is discussed by DeRemer and Kron (1976). This is also discussed as part of review of influence of software engineering on programming languages by Ryder et al. (2005).
9. Reynolds (1974)
10. For the multi-cultural origins of types, see Martini (2016) and for different possible answers to this question, see the reflections by Kell (2014) and Petricek (2015)

11. For these particular examples see the work on session types by Honda (1993), coeffects by Petricek et al. (2014), region-based memory management by Tofte and Talpin (1997) and ownership types by Clarke et al. (1998).
12. This is a reference to the work on "effect handlers", sparked by Plotkin and Pretnar (2009), that was briefly mentioned in the previous chapter.
13. In the context of mathematics, this process, including that of concept-stretching, has been described by Lakatos (1976). Programming concepts like types and tests are subject to similar evolutionary processes.
14. This is the framing used, for example, in the introduction of popular textbook on types by Pierce (2002).
15. This is often informally acknowledged in the functional programming community, e.g., "Being able to just show the code to a client and have him immediately catch domain modelling errors: priceless." (`https://twitter.com/brandewinder/status/570437796113985536`, retrieved 6 Oct, 2022) Description of this way of working with types can be found, for example, in a book by Wlaschin (2018)
16. This phrase is due to Minsky and Weeks (2008)
17. Adapted from `https://www.kindohm.com/posts/2016/2016-09-02-tidal-vis/`, retrieved 6 October, 2022
18. `https://www.typescriptlang.org/docs/handbook/type-compatibility.html`, retrieved 7 October, 2022
19. This is a perspective advocated by Kell (2014), who also provides a number of examples of confused claims resulting from the conflation.
20. This is the position advocated by Petricek (2015)
21. A claim made with a nod to Feyerabend (1975)
22. A call for pluralism in thinking about types and programming more generally has been made by Martini (2016)
23. See recording: `https://www.youtube.com/watch?v=aeRVdYN6fE8`, retrieved 26 October, 2022
24. An accessible summary of my own objections can be found at: `https://tomasp.net/blog/2018/alien-lambda-calculus/`, retrieved 26 October, 2022
25. Documented by Urquhart (1988), while Coquand (2018) discusses a broader context.
26. Russell (1903)
27. Russell (1908)
28. Church (1940)
29. Russell (1908)
30. Church (1940)
31. The early history of types and some of their early uses in the context of programming in the context of the lambda calculus have been discussed by Cardone and Hindley (2006).
32. Backus et al. (1956)
33. Perlis and Samelson (1958)
34. Martini (2016)
35. Naur (1978)
36. IBM (1960)
37. The history of COBOL has been documented by Sammet (1978).
38. Sammet (1961)
39. Postley (1960), quoted in Ensmenger (2012)
40. Postley (1960)
41. Rochester (1960)
42. A review of those efforts has been written by Priestley (2011) and includes work on "plexes" by Ross (1961), ongoing work by McCarthy et al. (1962) on list processing in LISP, as well as the work that I discuss in detail in this section.
43. McCarthy (1961)
44. PROGRAMme (2022)
45. Dahl et al. (1972)
46. Lindsey (1996b)
47. van Wijngaarden et al. (1969)
48. First documented by Wirth (1971), but historical recollections in Wirth (1996) provide another valuable source.
49. Schuman and Jorrand (1970)
50. Astarte (2017); Mervyn Pragnell's underground reading group is further mentioned in various personal recollections, including Burstall (2000); Bornat (2009); Hodges (2001); McBurney (2009).
51. Astarte (2017)

52. Burstall (2000)
53. The history has been thoroughly documented by Astarte (2019)
54. Landin (1964)
55. Landin (1964)
56. Landin (1966a)
57. Landin (1965a,b)
58. Landin (1966b)
59. Morris Jr (1969)
60. `https://ban.ai/multics/doc/MAC-PR-03-648346.pdf`, TODO: link dead
61. Morris Jr (1969)
62. Teitelman (1984)
63. Morris (1973b)
64. Parnas (1972)
65. Morris (1973a)
66. Liskov and Zilles (1974)
67. Reynolds (1974)
68. Lakatos (1976)
69. Kell (2014)
70. Scott (1993)
71. Milner (2003)
72. See Gordon (2000) for a first-hand account of the history of LCF.
73. Milner (2003)
74. Scott (1993)
75. See, for example, Milner and Weyhrauch (1972) and the general discussion in a later tutorial article, Milner (1979).
76. Milner (1979)
77. Milner (1979)
78. Presented by Milner (1978)
79. Morris Jr (1969)
80. Some of the engineering work on ML has been discussed in Abadi et al. (2014).
81. Cardelli and MacQueen (1983)
82. Cardelli and MacQueen (1983)
83. Milner et al. (1990); the history written by MacQueen et al. (2020) documents the development of Standard ML.
84. Pierce (2002)
85. Talpin and Jouvelot (1994)
86. Tofte and Talpin (1997)
87. Clarke et al. (1998)
88. Hudak et al. (2007)
89. An exception that proves the rule is paper by McBride (2002) which documents some of the tricks. It does, however, have a somewhat idiosyncratic, personal and lighthearted style that one might expect from a hacker culture of programming.
90. This has been gradually changing with recent developments that focus on extending Haskell with full support for dependent types.
91. For more information about the history and legacy of Automath, see the work by Dechesne and Nederpelt (2012). Automath is also documented by Laan (1997), who puts it in a wider context of the evolution of type theory in logic and mathematics.
92. Knuth (1973)
93. Harrison et al. (2014) documents the technical history of interactive theorem proving, including the different systems and their influences.
94. The fact that Java and Scala type systems are unsound, as shown by Amin and Tate (2016), is just one example where real implementation proved to be more complex than its formal models.
95. As mentioned earlier, this is much like the process of proofs and refutations described by Lakatos (1976) in the context of mathematics.
96. Altenkirch et al. (1994)
97. How work on interactive theorem proving influenced programming languages is not well documented, but Altenkirch et al. (2005) provides useful starting points.
98. Nordström et al. (1990)

99. Magnusson and Nordström (1993)
100. Adapted from Augustsson (1998)
101. Cayenne was described in Augustsson (1998) and Dependent ML in Xi and Pfenning (1999)
102. Epigram by McBride and McKinna (2004), Idris by Brady (2017) and Agda by Norell (2007), which also has a well-documented interactive Emacs plugin Coquand et al. (2006).
103. Leroy (2009)
104. Paulson (1993)
105. Pitts (2011)
106. Stroustrup (1996)
107. More information about the history of Microsoft code assistance tools can be found at: `https://devblogs.microsoft.com/cppblog/intellisense-history-part-1/`, retrieved 5 November 2022.
108. For examples of this reasonsing, see Rosenwasser (2018)
109. Cavanaugh (2018)
110. It remains to be seen whether the use of types in TypeScript can be reconciled with perspectives of other cultures. So far, most proposals coming from other cultures propose to "fix unsoundness" by making the type system more complex, which goes against the TypeScript design principles (Vekris et al. (2016); Richards et al. (2015)). However, a 2015 article "In Defense of Soundiness: A Manifesto" by Livshits et al. (2015), published in the Communications of the ACM suggests that the engineering culture approach to soundness is gaining acceptance in parts of the computer science community. The article notes that "the dominant practice is one of treating soundness as an engineering choice" and elaborates on how to best make such trade-offs.
111. The development of F# has been documented by Syme (2020).
112. This is the initial motivation for the feature, as outlined by Syme et al. (2013)
113. For examples of "silly type providers", see `http://www.pinksquirrellabs.com/tags/type-providers.html`, retrieved 5 November 2022
114. SOLNTSEFF and YEZERSKI (1974); Schuman and Jorrand (1970)
115. In Petricek (2015), I suggest this may be related to the "meaning is use" approach to philosophy of language proposed by Wittgenstein.
116. Kennedy (1996)
117. I say counter-example with reference to Lakatos (1976), who documents how counter-examples in mathematics force mathematicians to revisit their definitions. The difference here is that the counter-example comes from outside of the mathematical culture and so multi-cultural programming concepts like types are likely more susceptible to such changes.
118. The development fits with that described by Latour (1987)
119. Using the metaphor introduced by Galison (1997), we can see types as entities that exist in the trading zone between (at least) the mathematical and engineering cultures of programming.
120. The most infamous case of such translation is the notion of a monad that I discussed in Petricek (2018).
121. "Anything goes" as Feyerabend (1975) says, but the history of types follows a more refined version of the slogan in that each culture has its own principles that it follows.
122. This view is close to the experimental philosophy of science advocated by Hacking (1983).
123. Martini (2016)

Bibliography

1952. The Moniac. *Fortune Magazine* (March 1952), 100–101.

Sam Aaron. 2016. Sonic Pi–performance in education, technology and art. *International Journal of Performance Arts and Digital Media* 12, 2 (2016), 171–178.

Martin Abadi and Luca Cardelli. 1996. *A theory of objects*. Springer.

Martin Abadi, Philippa Gardner, Andy Gordon, Radu Mardare, and (.Eds). 2014. *Essays for the Luca Cardelli Fest*. Technical Report MSR-TR-2014-104. `https://www.microsoft.com/en-us/research/publication/essays-for-the-luca-cardelli-fest/`

Janet Abbate. 2012. *Recoding gender: Women's changing participation in computing*. Mit Press.

Alison Adam. 2005. Hacking into hacking: Gender and the hacker phenomenon. In *Gender, ethics and information technology*. Springer, 128–146.

Kartik Agaram. 2020. *Bicycles for the Mind Have to Be See-Through*. Association for Computing Machinery, New York, NY, USA, 173–186.

C. D. Allen, D. N. Chapman, and C. B. Jones. 1972. *A Formal Definition of ALGOL 60*. Technical Report 12.105. IBM Laboratory Hursley. `http://homepages.cs.ncl.ac.uk/cliff.jones/publications/Other-TRs/TR12.105.pdf`

Thorsten Altenkirch, Veronica Gaspes, Bengt Nordström, and Björn von Sydow. 1994. *A user's guide to ALF*. Technical Report. Chalmers University of Technology, Sweden.

Thorsten Altenkirch, Conor McBride, and James McKinna. 2005. Why dependent types matter. *Manuscript, available online* (2005), 235.

Nada Amin and Ross Tate. 2016. Java and Scala's type systems are unsound: The existential crisis of null pointers. *Acm Sigplan Notices* 51, 10 (2016), 838–848.

ANSI/IEEE. 1983. *IEEE Standard for Software Test Documentation*. Technical Report 829-1983. Institute of Electrical and Electronics Engineers.

ANSI/IEEE. 1986. *IEEE Standard for Software Verification and Validation Plans*. Technical Report 1012-1986. Institute of Electrical and Electronics Engineers.

ANSI/IEEE. 1987. *IEEE Standard for Software Unit Testing*. Technical Report 1008-1987. Institute of Electrical and Electronics Engineers.

Joe Armstrong. 2007. A History of Erlang. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages* (San Diego, California) *(HOPL III)*. ACM, New York, NY, USA, 6–1–6–26. https://doi.org/10.1145/1238844.1238850

R. L. Ashenhurst. 1972. Curriculum Recommendations for Graduate Professional Programs in Information Systems. *Commun. ACM* 15, 5 (May 1972), 363–398. https://doi.org/10.1145/355602.361320

Troy Astarte. 2017. Towards an Interconnected History of Semantics. In *Fourth International Conference on the History and Philosophy of Computing*. Brno, Czech Republic.

Troy K. Astarte. 2019. *Formalising Meaning: a History of Programming Language Semantics*. Ph. D. Dissertation. Newcastle University.

Troy K. Astarte and Cliff B. Jones. 2018. Formal Semantics of ALGOL 60: Four Descriptions in their Historical Context. In *Reflections on Programming Systems - Historical and Philosophical Aspects*, Liesbeth De Mol and Giuseppe Primiero (Eds.). Springer Philosophical Studies Series, 71–141.

William F. Atchison. 1985. The Development of Computer Science Education. *Adv. Comput.* 24 (1985), 319–377. https://doi.org/10.1016/S0065-2458(08)60370-8

Bill Atkinson. 2016. Bill Atkinson, interviewed by Leo Laporte. https://twit.tv/shows/triangulation/episodes/247.

Lennart Augustsson. 1998. Cayenne—a language with dependent types. In *International School on Advanced Functional Programming*. Springer, 240–267.

John W Backus, Friedrich L Bauer, Julien Green, Charles Katz, John McCarthy, Peter Naur, Alan J Perlis, Heinz Rutishauser, Klaus Samelson, Bernard Vauquois, et al. 1960. Report on the algorithmic language ALGOL 60. *Numer. Math.* 2, 1 (1960), 106–136.

J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, and M. Woodger. 1963. Revised Report on the Algorithm Language ALGOL 60. *Commun. ACM* 6, 1 (Jan. 1963), 1–17. https://doi.org/10.1145/366193.366201

J. W. Backus, R. J. Beeber, S. Best, R. Goldberg, H. L. Herrick, R. A. Hughes, L. B. Mitchell, R. A. Nelson, R. Nutt, D. Sayre, P. B. Sheridan, H. Stern, and L. Ziller. 1956. *Fortran: Automatic Coding System for the IBM 704 EDPM*.

John W Backus, Harlan Herrick, and Irving Ziller. 1954. Specifications for the IBM Mathematical FORmula TRANSlating System, FORTRAN. Preliminary report, Programming Research Group. *Applied Science Division, International Business Machines Corporation, New York* (1954).

Terry F Baker and D Mills, Harlan. 1973. Chief programmer teams. *Datamation* 19, 12 (1973), 58–61.

Richard J Barber. 1975. The Advanced Research Projects Agency, 1958–1974.

T Bardini. 2000. The personal interface: Douglas Engelbart, the augmentation of human intellect and the genesis of personal computing.

Jean Jennings Bartik. 2013. *Pioneer programmer: Jean Jennings Bartik and the computer that changed the world*. Truman State University Press.

Alan Bawden, Richard Greenblatt, Jack Holloway, Thomas Knight, David Moon, and Daniel Weinreb. 1974. *LISP Machine Progress Report*. Technical Report AIM-444. MIT. https://dspace.mit.edu/handle/1721.1/5751

Kent Beck. 2000. *Extreme programming explained: embrace change*. addison-wesley professional.

Kent Beck. 2003. *Test-driven development: by example*. Addison-Wesley Professional.

M Beeler, R. W Gosper, and R Schroeppel. 1972. *HAKMEM*. Technical Report. AI Memo 239, Technical Report, MIT.

H. D. Benington. 1983. Production of Large Computer Programs. *Annals of the History of Computing* 5, 4 (Oct 1983), 350–361. https://doi.org/10.1109/MAHC.1983.10102

Chris Bissell. 2007. Historical perspectives – The Moniac A Hydromechanical Analog Computer of the 1950s. *IEEE Control Systems Magazine* 27, 1 (2007), 69–74. https://doi.org/10.1109/MCS.2007.284511

Alan F. Blackwell and Nick Collins. 2005. The Programming Language as a Musical Instrument. In *Proceedings of the 17th Annual Workshop of the Psychology of Programming Interest Group, PPIG 2005, Brighton, UK, June 29 - July 1, 2005*. Psychology of Programming Interest Group, 11.

Daniel G Bobrow, D Lucille Darley, L Peter Deutsch, Daniel L Murphy, and Warren Teitelman. 1967. *The BBN 940 LISP System*. Technical Report. Bolt Beranek and Newman, Inc. http://www.softwarepreservation.org/projects/LISP/bbnlisp/BBN940Lisp_Jul67.pdf

Barry W. Boehm. 1988. A spiral model of software development and enhancement. *Computer* 21, 5 (1988), 61–72.

Corrado Böhm and Giuseppe Jacopini. 1966. Flow diagrams, Turing machines and languages with only two formation rules. *Commun. ACM* 9, 5 (1966), 366–371.

Richard Bornat. 2009. Peter Landin: a computer scientist who inspired a generation. *Higher-Order and Symbolic Computation* 22, 4 (2009), 295–298.

Robert Bosak, Richard F. Clippinger, Carey Dobbs, Roy Goldfinger, Renee B. Jasper, William Keating, George Kendrick, and Jean E. Sammet. 1962. An Information Algebra: Phase 1 Report&Mdash;Language Structure Group of the CODASYL Development Committee. *Commun. ACM* 5, 4 (April 1962), 190–204. https://doi.org/10.1145/366920.366935

Chuck Boyer. 2004. *The 360 Revolution*. IBM. Online at https://archive.org/details/h42_The_360_Revolution_Chuck_Boyer.

Robert S Boyer and JS Moore. 1983. On Why It Is Impossible to Prove that the BDX90 Dispatcher Implements a Time-sharing System. (1983).

Robert S. Boyer and J. Strother Moore. 1988. *A Computational Logic Handbook*. Academic Press Professional, Inc., USA.

Edwin Brady. 2017. *Type-driven development with Idris*. Manning Publications Company.

Stewart Brand and RE Crandall. 1988. The media lab: Inventing the future at MIT. *Computers in Physics* 2, 1 (1988), 91–92.

Fred Brooks. 1987. No Silver Bullet: Essence and Accidents of Software Engineering. *Computer* 20, 4 (April 1987), 10–19. https://doi.org/10.1109/MC.1987.1663532

Frederick P Brooks Jr. 1975. *The mythical man-month: essays on software engineering*. Addison-Wesley.

Henry Brown. 1886. Receptacle for storing and preserving papers. https://patents.google.com/patent/US352036A/

Timothy A. Budd, Richard J. Lipton, Richard A. DeMillo, and Frederick G. Sayward. 1978. The design of a prototype mutation system for program testing. In *American Federation of Information Processing Societies: 1978 National Computer Conference, June 5-8, 1978, Anaheim, CA, USA (AFIPS Conference Proceedings, Vol. 47)*, Sakti P. Ghosh and Leonard Y. Liu (Eds.). AFIPS Press, 623–629.

Rod Burstall. 2000. Christopher Strachey—understanding programming languages. *Higher-Order and Symbolic Computation* 13, 1-2 (2000), 51–55.

Vannevar Bush. 1945. As we may think. *The atlantic monthly* 176, 1 (1945), 101–108.

J.N. Buxton, B. Randell, and NATO Science Committee. 1970. *Software Engineering Techniques: Report on a Conference Sponsored by the NATO Science Committee*. NATO Science Committee; available from Scientific Affairs Division, NATO.

Martin Campbell-Kelly. 1980. Programming the EDSAC: Early programming activity at the University of Cambridge. *Annals of the History of Computing* 2, 1 (1980), 7–36.

M. Campbell-Kelly. 1992. The Airy Tape: An Early Chapter in the History of Debugging. *IEEE Annals of the History of Computing* 14 (10 1992), 16–26. https://doi.org/10.1109/85.194051

Martin Campbell-Kelly. 2004. *From airline reservations to Sonic the Hedgehog: a history of the software industry*. MIT press.

Martin Campbell-Kelly. 2011. From theory to practice: the invention of programming, 1947-51. In *Dependable and Historic Computing*. Springer, 23–37.

Luca Cardelli and David MacQueen (Eds.). 1983. Polymorphism: The ML/LCF/Hope newsletter. *Polymorphism* 1, 1 (1983). http://lucacardelli.name/Papers/Polymorphism%20Vol%20I,%20No%201.pdf

Felice Cardone and J Roger Hindley. 2006. History of lambda-calculus and combinatory logic. *Handbook of the History of Logic* 5 (2006), 723–817.

Ryan Cavanaugh. 2018. Type-checking unsoundness: standardize treatment of such issues among TypeScript team/community? https://github.com/Microsoft/TypeScript/issues/9825#issuecomment-234115900 [Online; accessed 10-September-2018].

Paul Ceruzzi. 1988. Electronics Technology and Computer Science, 1940-1975: A Coevolution. *Annals of the History of Computing* 10, 4 (1988), 257–275. https://doi.org/10.1109/MAHC.1988.10036

Paul E Ceruzzi. 2003. *A history of modern computing*. MIT PRess.

Sivaramakrishnan Krishnamoorthy Chandrasekaran, Daan Leijen, Matija Pretnar, and Tom Schrijvers. 2018. Algebraic effect handlers go mainstream (dagstuhl seminar 18172). In *Dagstuhl reports*, Vol. 8. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

Hasok Chang. 2012. *Is water H2O?: Evidence, realism and pluralism*. Vol. 293. Springer Science & Business Media.

Chris Brown and John Bischoff. 2002. INDIGENOUS TO THE NET: Early Network Music Bands in the San Francisco Bay Area. http://crossfade.walkerart.org/brownbischoff/IndigenoustotheNetPrint.html [Online; accessed 7-December-2021].

Alonzo Church. 1940. A formulation of the simple theory of types. *Journal of Symbolic Logic* 5, 2 (1940), 56–68. https://doi.org/10.2307/2266170

Peter Cihon. 2019. *Standards for AI governance: international standards to enable global coordination in AI research & development*. Technical Report. Future of Humanity Institute. University of Oxford.

David G Clarke, John M Potter, and James Noble. 1998. Ownership types for flexible alias protection. In *Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. 48–64.

Avra Cohn. 1989. The notion of proof in hardware verification. *Journal of Automated Reasoning* 5, 2 (01 Jun 1989), 127–139. https://doi.org/10.1007/BF00243000

Timothy R Colburn, James H Fetzer, and Terry L Rankin (Eds.). 2012. *Program verification: Fundamental issues in computer science*. Vol. 14. Springer Science & Business Media.

Richard A. Conger. 1962. Certification of Algorithm 58: Matrix Inversion. *Commun. ACM* 5, 6 (June 1962), 347. https://doi.org/10.1145/367766.368188

Catarina Coquand, Makoto Takeyama, and Dan Synek. 2006. An Emacs interface for type directed support constructing proofs and programs. In *European Joint Conferences on Theory and Practice of Software, ENTCS*, Vol. 2.

Thierry Coquand. 2018. Type Theory. In *The Stanford Encyclopedia of Philosophy* (fall 2018 ed.), Edward N. Zalta (Ed.). Metaphysics Research Lab, Stanford University.

Kate Crawford and Trevor Paglen. 2019. Excavating AI: The politics of images in machine learning training sets. *AI and Society* (2019).

O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare (Eds.). 1972. *Structured Programming*. Academic Press Ltd., GBR.

Dave Damouth, John Urbach, J. G. Mitchel, and Alan Kay. 1971. *PARC Papers for Pendery and Planning Purposes*. Technical Report. Xerox PARC.

Jeffrey Dastin. 2018. Amazon scraps secret AI recruiting tool that showed bias against women. In *Ethics of Data and Analytics*. Auerbach Publications, 296–299.

Richard A. De Millo, Richard J. Lipton, and Alan J. Perlis. 1979. Social Processes and Proofs of Theorems and Programs. *Commun. ACM* 22, 5 (May 1979), 271–280. https://doi.org/10.1145/359104.359106

Francien Dechesne and RP Nederpelt. 2012. NG de Bruijn (1918-2012) and his road to Automath, the earliest proof checker. *The Mathematical Intelligencer* 34, 4 (2012), 4–11.

Richard A. DeMillo, Richard J. Lipton, and Alan J. Perlis. 1977. Social Processes and Proofs of Theorems and Programs. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Los Angeles, California) *(POPL '77)*. Association for Computing Machinery, New York, NY, USA, 206–214. https://doi.org/10.1145/512950.512970

Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. 1978. Hints on Test Data Selection: Help for the Practicing Programmer. *Computer* 11, 4 (1978), 34–41. https://doi.org/10.1109/C-M.1978.218136

F. DeRemer and H. H. Kron. 1976. Programming-in-the-Large Versus Programming-in-the-Small. *IEEE Transactions on Software Engineering* SE-2, 2 (June 1976), 80–86. https://doi.org/10.1109/TSE.1976.233534

L Peter Deutsch and Butler W Lampson. 1967. An online editor. *Commun. ACM* 10, 12 (1967), 793–799.

P Deutsch. 1967. *Preliminary Guide to the LISP Editor*. Technical Report Project Genie Document W-21. University of California, Berkeley. http://www.softwarepreservation.org/projects/LISP/bbnlisp/W-21_LISP_Editor_Apr67.pdf

Andrea A Di Sessa. 1985. A principled design for an integrated computational environment. *Human-computer interaction* 1, 1 (1985), 1–47.

Edsger W. Dijkstra. 1968. Letters to the Editor: Go to Statement Considered Harmful. *Commun. ACM* 11, 3 (March 1968), 147–148. https://doi.org/10.1145/362929.362947

Edsger W. Dijkstra. 1970. Notes on Structured Programming. (April 1970). http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD249.PDF circulated privately.

Edsger W. Dijkstra. 1972. The Humble Programmer. *Commun. ACM* 15, 10 (Oct. 1972), 859–866. https://doi.org/10.1145/355604.361591

Edsger W Dijkstra. 1977. A position paper on software reliability. *ACM SIGSOFT Software Engineering Notes* 2, 5 (1977), 3–5.

Edsger W Dijkstra. 1978. On a political pamphlet from the middle ages. *ACM SIGSOFT software engineering notes* 3, 2 (1978), 14–16.

Edsger W Dijkstra. 1980. *America's programming plight*. Technical Report EWD 750. The University of Texas at Austin.

Edsger W Dijkstra. 1982. How Do We Tell Truths that Might Hurt? In *Selected Writings on Computing: A personal Perspective*. Springer, 129–131.

Edsger W Dijkstra. 1988. *On the cruelty of really teaching computing science*. Technical Report EWD 1036. The University of Texas at Austin.

Edsger W Dijkstra. 1993. *In reply to comments*. Technical Report EWD 1058. The University of Texas at Austin.

Edsger W Dijkstra. 2002. Ewd 1308: What led to "Notes on structured programming". In *Software Pioneers*. Springer, 340–346.

Andrea A. diSessa and Harold Abelson. 1986. Boxer: A reconstructible computational medium. *Commun. ACM* 29, 9 (1986), 859–868.

M. Dyer and Harlan D. Mills. 1981. Cleanroom Software Development. In *Sixth Annual Software Engineering Workshop* (Goddard Space Flight Center). NASA.

Thomas O Ellis, John F Heafner, and William L Sibley. 1969. *The GRAIL Project: An experiment in man-machine communications*. Technical Report. RAND Corporation, Santa Monica, CA.

Christina Engelbart and Bret Victor. 1968. The 1968 Demo - Interactive. `https://www.dougengelbart.org/content/view/374/464/` Doug Engelbart Institute, [Online; accessed 16 February-2021].

Douglas C Engelbart. 1962. Augmenting human intellect: A conceptual framework. *Menlo Park, CA* (1962).

Nathan Ensmenger. 2010. Making programming masculine. In *Gender codes: Why women are leaving computing*. IEEE Computer Society and John Wiley & Sons, Inc., Hoboken, New Jersey, 115–141.

N.L. Ensmenger. 2012. *The Computer Boys Take Over: Computers, Programmers, and the Politics of Technical Expertise*. MIT Press.

Nathan Ensmenger. 2015. "Beards, sandals, and other signs of rugged individualism": masculine culture within the computing professions. *Osiris* 30, 1 (2015), 38–65.

Thomas G. Evans and D. Lucille Darley. 1965. DEBUG–an Extension to Current Online Debugging Techniques. *Commun. ACM* 8, 5 (May 1965), 321–326. `https://doi.org/10.1145/364914.364952`

Thomas G Evans and D Lucille Darley. 1966. On-line debugging techniques: a survey. In *Proceedings of the November 7-10, 1966, fall joint computer conference*. ACM, 37–50.

Guy C Fedorkow. 2021. Recovering Software for the Whirlwind Computer. *IEEE Annals of the History of Computing* 43, 1 (2021), 38–59.

James H. Fetzer. 1988. Program Verification: The Very Idea. *Commun. ACM* 31, 9 (Sept. 1988), 1048–1063. https://doi.org/10.1145/48529.48530

P Feyerabend. 1975. *Against Method*. Verso.

]nyt-1975-safeguard John W. Finney. [n. d.]. Safeguard ABM System to Shut Down; $5 Billion Spent in 6 Years Since D

Christiane Floyd. 1987. Outline of a Paradigm Change in Software Engineering. In *Computers and Democracy: A Scandanavian Challenge*, G. Bjerknes (Ed.). Brookfield, Gower Publishing Company, Old Post Road, Brookfield, USA, 191–210.

Christine Floyd, Wolf-Michael Mehl, Fanny-Michaela Resin, Gerhard Schmidt, and Gregor Wolf. 1989. Out of Scandinavia: Alternative approaches to software design and system development. *Human–computer interaction* 4, 4 (1989), 253–350.

Robert W Floyd. 1967. Assigning meanings to program. In *Proc. Symposia in Applied Mathematics, 1967*, Vol. 19. 19–32.

Jay Wright Forrester and Robert R Everett. 1990. The Whirlwind computer project. *IEEE Trans. Aerospace Electron. Systems* 26, 5 (1990), 903–910.

Martin Fowler. 2001. The new methodology. *Wuhan University Journal of Natural Sciences* 6, 1 (2001), 12–24.

Phyllis Fox. 1960. *LISP I Programmers manual*. Internal Paper. MIT, Cambridge. http://bitsavers.org/pdf/mit/rle_lisp/LISP_I_Programmers_Manual_Mar60.pdf

Paul Freiberger and Michael Swaine. 1984. *Fire in the Valley: the making of the personal computer*. McGraw-Hill, Inc.

Richard P. Gabriel. 2012. The Structure of a Programming Language Revolution. In *Proceedings of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (Tucson, Arizona, USA) *(Onward! 2012)*. ACM, New York, NY, USA, 195–214. 978-1-4503-1562-3 https://doi.org/10.1145/2384592.2384611

Peter Galison. 1997. *Image and logic: A material culture of microphysics*. University of Chicago Press.

D. Gelperin and B. Hetzel. 1988. The Growth of Software Testing. *Commun. ACM* 31, 6 (June 1988), 687–695. 0001-0782 https://doi.org/10.1145/62959.62965

B. Gold and R.A. Simons. 2008. *Proof and Other Dilemmas: Mathematics and Philosophy*. Mathematical Association of America. 9780883855676 2008922718

Adele Goldberg and Alan Kay. 1976. *Smalltalk-72: Instruction Manual*. Xerox Corporation Palo Alto.

Ernst Hans Gombrich. 1961. *Art and illusion*. Pantheon Books New York.

John B. Goodenough. 1975a. Exception Handling: Issues and a Proposed Notation. *Commun. ACM* 18, 12 (Dec. 1975), 683–696. 0001-0782 https://doi.org/10.1145/361227.361230

John B. Goodenough. 1975b. Structured Exception Handling. In *Conference Record of the Second ACM Symposium on Principles of Programming Languages, Palo Alto, California, USA, January 1975*, Robert M. Graham, Michael A. Harrison, and John C. Reynolds (Eds.). ACM Press, 204–224. https://doi.org/10.1145/512976.512997

John B. Goodenough and Susan L. Gerhart. 1975. Toward a Theory of Test Data Selection. In *Proceedings of the International Conference on Reliable Software* (Los Angeles, California). ACM, New York, NY, USA, 493–510. https://doi.org/10.1145/800027.808473

Mike Gordon. 2000. From LCF to HOL: a short history.. In *Proof, language, and interaction*. 169–186.

Burton Grad. 2007. The creation and the demise of VisiCalc. *IEEE Annals of the History of Computing* 29, 3 (2007), 20–31.

Richard Greenblatt. 2005. *Oral History of Richard Greenblatt, interviewed by Gardner Hendrie, January 12, 2005*. Technical Report X3056.2005. Computer History Museum.

Scot Gresham-Lancaster. 1998. The aesthetics and history of the hub: The effects of changing technology on network computer music. *Leonardo Music Journal* 8, 1 (1998), 39–44.

David Gries. 1978a. *Programming methodology: A collection of articles by members of IFIP WG 2.3*. Springer-Verlag New York Inc.

David Gries (Ed.). 1978b. *Programming Methodology: A Collection of Articles by Members of IFIP WG 2.3*. Springer-Verlag, Berlin, Heidelberg. 0387903291

Leo Gugerty. 2006. Newell and Simon's Logic Theorist: Historical Background and Impact on Cognitive Modeling. *Proceedings of the Human Factors and Ergonomics Society Annual Meeting* 50, 9 (2006), 880–884. https://doi.org/10.1177/154193120605000904

A Nico Habermann. 1969. Prevention of system deadlocks. *Commun. ACM* 12, 7 (1969), 373–ff.

I. Hacking. 1983. *Representing and Intervening: Introductory Topics in the Philosophy of Natural Science*. Cambridge University Press. 9780521282468 83005132

Thomas Haigh. 2010. Dijkstra's Crisis: The End of Algol and Beginning of Software Engineering, 1968-72. https://tomandmaria.com/Tom/Writing/DijkstrasCrisis_LeidenDRAFT.pdf [Online; accessed 9-August-2022].

Thomas Haigh and Paul E Ceruzzi. 2021. *A New History of Modern Computing*. MIT Press.

Thomas Haigh, Peter Mark Priestley, Mark Priestley, and Crispin Rope. 2016. *ENIAC in action: Making and remaking the modern computer*. MIT press.

Mark Halpern. 1965. Computer programming: The debugging epoch opens. *Computers and Automation* 14, 11 (November 1965), 28–31.

Michael J Halvorson. 2020. *Code Nation: Personal computing and the learn to program movement in America*. ACM Books.

David Harel. 1980. On folk theorems. *Commun. ACM* 23, 7 (1980), 379–389.

John Harrison, Josef Urban, and Freek Wiedijk. 2014. History of Interactive Theorem Proving. In *Computational Logic*, Vol. 9. 135–214.

Wolfgang Henhapl and Cliff B. Jones. 1978. *A formal definition of ALGOL 60 as described in the 1975 modified report*. Springer Berlin Heidelberg, Berlin, Heidelberg, 305–336. 978-3-540-35836-7 https://doi.org/10.1007/3-540-08766-4_12

Bill Hetzel. 1988. *The Complete Guide to Software Testing* (2nd ed.). QED Information Sciences, Inc., USA. 0894352423

W.C. Hetzel (Ed.). 1973. *Program Test Methods*. Prentice-Hall. 9780137296248 72008657

Marie Hicks. 2010. Meritocracy and feminization in conflict: computerization in the British Government. In *Gender codes: why women are leaving computing*. IEEE Computer Society and John Wiley & Sons, Inc., Hoboken, New Jersey, 95–114.

Mar Hicks. 2017. *Programmed inequality: How Britain discarded women technologists and lost its edge in computing*. MIT Press.

Michael A Hiltzik et al. 1999. *Dealers of lightning: Xerox PARC and the dawn of the computer age*. HarperCollins Publishers.

Charles AR Hoare. 1971. Proof of a program: FIND. *Commun. ACM* 14, 1 (1971), 39–45.

C. A. R. Hoare. 1965. Record handling. *Algol Bulletin* 21 (1965), 39–69.

C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12, 10 (Oct. 1969), 576–580. 0001-0782 https://doi.org/10.1145/363235.363259

C. A. R. Hoare. 1972. Structured Programming. Academic Press Ltd., London, UK, UK, Chapter Chapter II: Notes on Data Structuring, 83–174. 0-12-200550-3

Charles Antony Richard Hoare. 1978. Communicating sequential processes. *Commun. ACM* 21, 8 (1978), 666–677.

Charles Antony Richard Hoare. 1981. The Emperor's Old Clothes. *Commun. ACM* 24, 2 (feb 1981), 75–83. 0001-0782 https://doi.org/10.1145/358549.358561

C. A. R. Hoare. 1996. How did software get so reliable without proof?. In *FME'96: Industrial Benefit and Advances in Formal Methods*, Marie-Claude Gaudel and James Woodcock (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–17. 978-3-540-49749-3

Wilfrid Hodges. 2001. A history of British logic. http://wilfridhodges.co.uk/history01.pdf [Online; accessed 25 August-2020].

Kohei Honda. 1993. Types for dyadic interaction. In *International Conference on Concurrency Theory*. Springer, 509–523.

Sara Hooker. 2021. Moving beyond "algorithmic bias is a data problem". *Patterns* 2, 4 (2021), 100241.

J. C. Huang. 1975. An Approach to Program Testing. *ACM Comput. Surv.* 7, 3 (Sept. 1975), 113–128. 0360-0300 https://doi.org/10.1145/356651.356652

Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. 2007. A History of Haskell: Being Lazy with Class. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages* (San Diego, California) *(HOPL III)*. Association for Computing Machinery, New York, NY, USA, 12–1–12–55. 9781595937667 https://doi.org/10.1145/1238844.1238856

International Business Machines Corporation. Data Processing Division. IBM. 1960. *General Information Manual: IBM Commercial Translator*. IBM. Online at http://bitsavers.org/pdf/ibm/7090/F28-8043_CommercialTranslatorGenInfMan_Ju60.pdf.

Jean D. Ichbiah, Bernd Krieg-Brueckner, Brian A. Wichmann, John G. P. Barnes, Olivier Roubine, and Jean-Claude Heliard. 1979. Rationale for the Design of the Ada Programming Language. *SIGPLAN Not.* 14, 6b (June 1979), 1–261. 0362-1340 https://doi.org/10.1145/956653.956654

Adrian Johns. 2010. *Piracy: The intellectual property wars from Gutenberg to Gates*. University of Chicago Press.

Jeff Johnson, Teresa L. Roberts, William Verplank, David Canfield Smith, Charles H. Irby, Marian Beard, and Kevin Mackey. 1989. The Xerox star: A retrospective. *Computer* 22, 9 (1989), 11–26.

Cliff B. Jones and Troy K. Astarte. 2016. *An Exegesis of Four Formal Descriptions of ALGOL 60*. Technical Report CS-TR-1498. Newcastle University School of Computer Science. https://assets.cs.ncl.ac.uk/TRs/1498.pdf Forthcoming as a paper in the HaPoP 2016 proceedings..

John T. Gilmore Jr. 1958. *TX-0 Direct Input Utility System*. Technical Report 6M-5097-1. Massachusetts Institute of Technology.

Alan Kay. 1972. A personal computer for children of all ages. In *Proceedings of the ACM annual conference-Volume 1*.

Alan Kay and Adele Goldberg. 1977. Personal dynamic media. *Computer* 10, 3 (1977), 31–41.

Alan C Kay. 1996. The early history of Smalltalk. *History of programming languages—II* (1996), 511–598.

Stephen Kell. 2014. In Search of Types. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software* (Portland, Oregon, USA) *(Onward! 2014)*. ACM, New York, NY, USA, 227–241. 978-1-4503-3210-1 https://doi.org/10.1145/2661136.2661154

Christopher M Kelty. 2008. *Two bits: The cultural significance of free software*. Duke University Press.

Andrew John Kennedy. 1996. *Programming languages and dimensions*. Ph. D. Dissertation. University of Cambridge, Computer Laboratory.

P. A. Kidwell. 1998. Stalking the elusive computer bug. *IEEE Annals of the History of Computing* 20, 4 (Oct 1998), 5–9. 1058-6180 https://doi.org/10.1109/85.728224

Ralph Kimball and B Verplank E Harslem. 1982. Designing the Star user interface. *Byte* 7, 1982 (1982), 242–282.

Gerwin Klein, June Andronick, Matthew Fernandez, Ihor Kuz, Toby C. Murray, and Gernot Heiser. 2018. Formally verified software in the real world. *Commun. ACM* 61, 10 (2018), 68–77. https://doi.org/10.1145/3230627

Dmytri Kleiner. 2010. *The telekommunist manifesto*. Vol. 3. Institute of Network Cultures Amsterdam.

D.E. Knuth. 1968. *The Art of Computer Programming*. Vol. 1-4. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Donald E. Knuth. 1973. *A review of "Structured Programming"*. Technical Report STAN-CS-73-371. Computer Science Department, Stanford University.

Alan Kotok. 2005. *Oral History of Alan Kotok, interviewed by Gardner Hendrie, November 15, 2004*. Technical Report X3004.2005. Computer History Museum. https://archive.computerhistory.org/resources/access/text/2013/05/102657916-05-01-acc.pdf

T.S. Kuhn. 1962. *The Structure of Scientific Revolutions*. University of Chicago Press. 9780226458144 2011042476

Matt J Kusner, Joshua Loftus, Chris Russell, and Ricardo Silva. 2017. Counterfactual fairness. *Advances in neural information processing systems* 30 (2017).

Twan Dismas Laurens Laan. 1997. *The evolution of type theory in logic and mathematics*. Ph. D. Dissertation.

I. Lakatos. 1976. *Proofs and Refutations: The Logic of Mathematical Discovery*. Cambridge University Press. 9780521290388 75332478

Butler Lampson. 1972. *Why Alto*. Technical Report. Xerox PARC.

P. J. Landin. 1964. The Mechanical Evaluation of Expressions. *Comput. J.* 6, 4 (1964), 308–320. https://doi.org/10.1093/comjnl/6.4.308

P. J. Landin. 1965a. Correspondence Between ALGOL 60 and Church's Lambda-notation: Part I. *Commun. ACM* 8, 2 (Feb. 1965), 89–101. 0001-0782 https://doi.org/10.1145/363744.363749

P. J. Landin. 1965b. A Correspondence Between ALGOL 60 and Church's Lambda-notations: Part II. *Commun. ACM* 8, 3 (March 1965), 158–167. 0001-0782 https://doi.org/10.1145/363791.363804

Peter J Landin. 1966a. A formal description of ALGOL 60. In *Formal Language Description Languages for Computer Programming*. North Holland Publishing Company, Amsterdam, 266–294.

P. J. Landin. 1966b. The Next 700 Programming Languages. *Commun. ACM* 9, 3 (March 1966), 157–166. 0001-0782 https://doi.org/10.1145/365230.365257

Bruno Latour. 1987. *Science in action: How to follow scientists and engineers through society*. Harvard university press.

Peter E. Lauer. 1968. *Formal definition of ALGOL 60*. Technical Report 25.088. IBM Laboratory Vienna. http://homepages.cs.ncl.ac.uk/cliff.jones/publications/OCRd/Lau68.pdf

B. M. Leavenworth. 1972. Programming with(out) the GOTO. *SIGPLAN Not.* 7, 11 (nov 1972), 54–58. 0362-1340 https://doi.org/10.1145/987361.987370

Meir M Lehman. 1980. Programs, life cycles, and laws of software evolution. *Proc. IEEE* 68, 9 (1980), 1060–1076.

Meir M. Lehman. 1993. *An Interview Conducted by William Aspray, IEEE History Center, 23 September 1993*. Technical Report 178. The Institute of Electrical and Electronics Engineers, Inc.

M. M. Lehman and L. A. Belady. 1985. *Program Evolution: Processes of Software Change*. Academic Press Professional, Inc., USA. 0124424406

Brian Lennon. 2019. Foo, Bar, Baz...: The Metasyntactic Variable and the Programming Language Hierarchy. *Philosophy & Technology* (18 Dec 2019). 2210-5441 https://doi.org/10.1007/s13347-019-00387-2

Xavier Leroy. 2009. Formal Verification of a Realistic Compiler. *Commun. ACM* 52, 7 (July 2009), 107–115. 0001-0782 https://doi.org/10.1145/1538788.1538814

Steven Levy. 2010. *Hackers: Heroes of the computer revolution*. O'Reilly Media. 978-1449388393

J. C. Licklider. 1969. Underestimates and overexpectations. *Computers and Automation* 18, 9 (1969), 48–52.

J. C. R. Licklider. 1960. Man-Computer Symbiosis. *IRE Transactions on Human Factors in Electronics* HFE-1, 1 (March 1960), 4–11. 0099-4561 https://doi.org/10.1109/THFE2.1960.4503259

Joseph Carl Robnett Licklider. 1965. *Libraries of the Future*. MiT Press.

C. H. Lindsey. 1996a. *A History of ALGOL 68*. Association for Computing Machinery, New York, NY, USA, 27–96. 0201895021 https://doi.org/10.1145/234286.1057810

C. H. Lindsey. 1996b. *A History of ALGOL 68*. Association for Computing Machinery, New York, NY, USA, 27–96. 0201895021 https://doi.org/10.1145/234286.1057810

Barbara Liskov. 1993. A History of CLU. In *The Second ACM SIGPLAN Conference on History of Programming Languages* (Cambridge, Massachusetts, USA) *(HOPL-II)*. ACM, New York, NY, USA, 133–147. 0-89791-570-4 https://doi.org/10.1145/154766.155367

Barbara Liskov and Stephen Zilles. 1974. Programming with Abstract Data Types. In *Proceedings of the ACM SIGPLAN Symposium on Very High Level Languages* (Santa Monica, California, USA). ACM, New York, NY, USA, 50–59. https://doi.org/10.1145/800233.807045

Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. 2015. In Defense of Soundiness: A Manifesto. *Commun. ACM* 58, 2 (Jan. 2015), 44–46. 0001-0782 https://doi.org/10.1145/2644805

AI Llewelyn and RF Wickens. 1968. The testing of computer software. In *Software Engineering*, *Report on a conference sponsored by the NATO SCIENCE COMMITTEE*, *Garmisch*, *Germany*. 7–11.

Donald MacKenzie. 2001. *Mechanizing Proof: Computing*, *Risk*, *and Trust*. MIT Press, Cambridge, MA, USA. 0-262-13393-8

D.A. MacKenzie. 2004. *Mechanizing Proof: Computing*, *Risk*, *and Trust*. MIT Press. 9780262632959 2001018687

Donald MacKenzie. 2014. A sociology of algorithms: High-frequency trading and the shaping of markets. https://c.mql5.com/forextsd/forum/169/algorithms25.pdf

David MacQueen, Robert Harper, and John Reppy. 2020. The history of Standard ML. *Proceedings of the ACM on Programming Languages* 4, HOPL (2020), 1–100.

Lena Magnusson and Bengt Nordström. 1993. The ALF proof editor and its proof engine. In *International Workshop on Types for Proofs and Programs*. Springer, 213–237.

Michael S Mahoney. 1988. The history of computing in the history of technology. *Annals of the History of Computing* 10, 2 (1988), 113–125.

Michael S Mahoney. 1992. Computers and mathematics: The search for a discipline of computer science. *The Space of Mathematics. Philosophical, Epistemological, and Historical Explorations. Berlin: Springer* (1992), 349–363.

Michael S Mahoney. 1997. *Computer science: the search for a mathematical theory*. Taylor and Francis.

Michael S Mahoney. 2005. The histories of computing(s). *Interdisciplinary science reviews* 30, 2 (2005), 119–135.

John Markoff. 2005. *What the dormouse said: How the sixties counterculture shaped the personal computer industry*. Viking Press.

William Martin and Timothy Hart. 1964. *Time Sharing LISP*. Technical Report AIM-67. MIT. `http://www.bitsavers.org/pdf/mit/ai/aim/AIM-067.pdf`

Simone Martini. 2016. Several Types of Types in Programming Languages. In *History and Philosophy of Computing*, Fabio Gadducci and Mirko Tavosanis (Eds.). Springer International Publishing, Cham, 216–227. 978-3-319-47286-7

Conor McBride. 2002. Faking it simulating dependent types in haskell. *Journal of functional programming* 12, 4-5 (2002), 375–392.

Conor McBride and James McKinna. 2004. The view from the left. *Journal of functional programming* 14, 1 (2004), 69–111.

Peter McBurney. 2009. Guerrilla logic: a salute to Mervyn Pragnell. `http://wilfridhodges.co.uk/history01.pdf` [Online; accessed 25 August-2020].

T.J. McCabe. 1976. A Complexity Measure. *IEEE Transactions on Software Engineering* SE-2, 4 (1976), 308–320. `https://doi.org/10.1109/TSE.1976.233837`

John McCarthy. 1961. A basis for a mathematical theory of computation, preliminary report. In *Papers presented at the May 9-11, 1961, western joint IRE-AIEE-ACM computer conference*. 225–238.

John McCarthy. 1962. Computer Programs for Checking Mathematical Proofs. In *Proceedings of Symposia in pure mathematics, vol 5*. American Mathematical Society.

John McCarthy. 1963. Towards a Mathematical Science of Computation. In *Information Processing* (Amsterdam, The Netherlands), C. M. Popplewell (Ed.). North-Holland Publishing Company, 21–28.

John McCarthy. 1964. A formal description of a subset of ALGOL. In *Formal Language Description Languages for Computer Programming*. North-Holland Publishing Company, 1–12.

John McCarthy. 1978. History of LISP. In *History of programming languages*. 173–185.

John McCarthy. 1992. Reminiscences on the history of time-sharing. *IEEE Annals of the History of Computing* 14, 1 (1992), 19–24.

John McCarthy, Paul W Abrahams, Daniel J Edwards, Timothy P Hart, and Michael I Levin. 1962. *LISP 1.5 programmer's manual*. MIT press. `https://www.softwarepreservation.org/projects/LISP/book/LISP%201.5%20Programmers%20Manual.pdf`

Daniel McCracken. 1973. Revolution in programming: an overview. *Datamation* (1973), 50–52.

J. David McGonagle. 1972. Notes on the Computer Program Test Methods Symposium. *SIGPLAN Not.* 7, 5 (May 1972), 8–12. 0362-1340 https://doi.org/10.1145/987053.987056

John A. McKenzie. 1974. *TX-0 Computer History*. Technical Report 627. MIT RLE.

McKinsey. 1968. *Unlocking the Computer's Profit Potential*. Technical Report. McKinsey & Company, Inc.

Joris Meerts and Dorothy Graham. 2010. The History of Software Testing. http://www.testingreferences.com/testinghistory.php [Online; accessed 14-October-2021].

HD Mills, M Dyer, and RC Linger. 1987. Cleanroom Software Engineering. *IEEE Software* 4, 5 (1987), 19–25.

Robin Milner. 1972. *Logic for Computable Functions: Description of a Machine Implementation*. Technical Report. Stanford, CA, USA.

Robin Milner. 1978. A theory of type polymorphism in programming. *J. Comput. System Sci.* 17, 3 (1978), 348 – 375. 0022-0000 https://doi.org/10.1016/0022-0000(78)90014-4

Robin Milner. 1979. Lcf: A way of doing proofs with a machine. In *Mathematical Foundations of Computer Science 1979*, Jiří Bečvář (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 146–159. 978-3-540-35088-0

Robin Milner. 1980. *A Calculus of Communicating Systems*. Lecture Notes in Computer Science, Vol. 92. Springer. 3-540-10235-3 https://doi.org/10.1007/3-540-10235-3

Robin Milner. 2003. An interview with Robin Milner, interviewed by Martin Berger, September 3, 2003. https://users.sussex.ac.uk/~mfb21/interviews/milner/, Retrieved 28 October, 2022.

Robin Milner, Joachim Parrow, and David Walker. 1992. A Calculus of Mobile Processes, I. *Inf. Comput.* 100, 1 (1992), 1–40. https://doi.org/10.1016/0890-5401(92)90008-4

Robin Milner, Mads Tofte, and Robert Harper. 1990. *Definition of standard ML*. MIT Press. 978-0-262-63132-7

Robin Milner and Richard Weyhrauch. 1972. Proving compiler correctness in a mechanized logic. *Machine intelligence* 7, 3 (1972), 51–70.

David A Mindell. 2011. *Digital Apollo: Human and machine in spaceflight*. Mit Press.

Yaron Minsky and Stephen Weeks. 2008. Caml trading–experiences with functional programming on Wall Street. *Journal of Functional Programming* 18, 4 (2008), 553–564.

Thomas J Misa. 2011. *Gender codes: Why women are leaving computing*. John Wiley & Sons.

Nick Montfort, Patsy Baudoin, John Bell, Ian Bogost, and Jeremy Douglass. 2014. *10 PRINT CHR* $(205.5 + RND(1)); : GOTO10. MITPress.$

J Strother Moore. 2019. Milestones from the pure Lisp theorem prover to ACL2. Formal Aspects of Computing *31, 6 (2019), 699–732.*

James H. Morris. 1973a. Protection in Programming Languages. Commun. ACM *16, 1 (Jan. 1973), 15–21. 0001-0782 https://doi.org/10.1145/361932.361937*

James H. Morris, Jr. 1973b. Types Are Not Sets. In Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages *(Boston, Massachusetts)* (POPL '73). *ACM, New York, NY, USA, 120–124. https://doi.org/10.1145/512927.512938*

James Hiram Morris Jr. 1969. Lambda-calculus models of programming language. *Ph. D. Dissertation. Massachusetts Institute of Technology.*

P. D. Mosses. 1974. The Mathematical Semantics of ALGOL 60. *Technical Report Technical Monograph PRG-12. Oxford University Computing Laboratory, Programming Research Group. http://homepages.cs.ncl.ac.uk/cliff.jones/publications/OCRd/Mosses74.pdf*

Rick Mugridge. 2003. Test driven development and the scientific method. In Proceedings of the Agile Development Conference, 2003. ADC 2003. *IEEE, 47–52.*

Elizabeth M. Murhpy. 2013. In the Matter of Knight Capital Americas LLC Respondent. *Technical Report SEC Release No. 70694, File No. 3-15570. Securities and Exchange Commission.*

J. D. Musa. 1975. A theory of software reliability and its application. IEEE Transactions on Software Engineering *1 (09 1975), 312–327. 0098-5589 https://doi.org/10.1109/TSE.1975.6312856*

Glenford J Myers, T Badgett, and C Sandler. 1979. The Art of Software Testing. John Wiley & Sons, Inc. New York *(1979), 22041–3467.*

Frieder Nake. 1971. There should be no computer art. Bulletin of the Computer Arts Society *(1971), 18–21.*

National Bureau of Standards. 1984. Guideline for Lifecycle Validation, Verification, And Testing of Computer Software. *Technical Report. U.S. Dept. of Commerce, National Bureau of Standards.*

Peter Naur. 1966. Proof of algorithms by general snapshots. BIT Numerical Mathematics *6, 4 (1966), 310–316.*

Peter Naur. 1978. The European Side of the Last Phase of the Development of ALGOL 60. *Association for Computing Machinery, New York, NY, USA, 92–139. 0127450408*

Peter Naur. 1993. The place of strictly defined notation in human insight. In Program Verification. *Springer, 261–274.*

P. Naur, B. Randell, F.L. Bauer, and NATO Science Committee. 1969. Software engineering: report on a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7th to 11th October 1968. *Scientific Affairs Division, NATO.*

Gina Neff and Peter Nagy. 2016. Talking to Bots: Symbiotic Agency and the Case of Tay. International Journal of Communication *10 (2016), 4915–4931.*

Allen Newell and Herbert Simon. 1956. The logic theory machine–A complex information processing system. IRE Transactions on information theory *2, 3 (1956), 61–79.*

Safiya Umoja Noble. 2018. Algorithms of oppression. In Algorithms of Oppression. *New York University Press.*

David Nofre, Mark Priestley, and Gerard Alberts. 2014. When technology became language: The origins of the linguistic conception of computer programming, 1950–1960. Technology and Culture *55, 1 (2014), 40–75.*

A Michael Noll. 2016. The Howard Wise Gallery Show Computer-Generated Pictures (1965): A 50th-Anniversary Memoir. Leonardo *49, 3 (2016), 232–239.*

Bengt Nordström, Kent Petersson, and Jan M Smith. 1990. Programming in Martin-Löf's type theory. *Vol. 200. Oxford University Press Oxford.*

Ulf Norell. 2007. Towards a practical programming language based on dependent type theory. *Ph. D. Dissertation.*

Joe November. 2004. LINC: biology's revolutionary little computer. Endeavour *28, 3 (2004), 125–131.*

Cathy O'Neil. 2016. Weapons of math destruction: How big data increases inequality and threatens democracy. *Broadway Books.*

Severo Ornstein. 2002. Computing in the Middle Ages: A View from the Trenches 1955 1983.

Seymour Papert. 1980. Mindstorms: Computers, children, and powerful ideas. *Basic Books. 255 pages.*

D. L. Parnas. 1972. On the Criteria to Be Used in Decomposing Systems into Modules. Commun. ACM *15, 12 (Dec. 1972), 1053–1058. 0001-0782 https://doi.org/10.1145/361598.361623*

David Lorge Parnas. 1985. Software Aspects of Strategic Defense Systems. Commun. ACM *28, 12 (Dec. 1985), 1326–1335. 0001-0782 https://doi.org/10.1145/214956.214961*

Lawrence C Paulson. 1993. Designing a theorem prover. In Handbook of logic in computer science (vol. 2) background: computational structures. *415–475.*

Maria Eloina Pelaez Valdez. 1988. A gift from Pandora's Box: the software crisis. *Ph. D. Dissertation. University of Edinburgh.*

Alan J. Perlis. 1978. The American Side of the Development of ALGOL. *Association for Computing Machinery, New York, NY, USA, 75–91. 0127450408*

Alan J. Perlis and Klaus Samelson. 1958. Preliminary report: International algebraic language. Commun. ACM *1, 12 (1958), 8–22.*

Tomas Petricek. 2015. Against a Universal Definition of 'Type'. In 2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!) *(Pittsburgh, PA, USA)* (Onward! 2015). *ACM, New York, NY, USA, 254–266. 978-1-4503-3688-8 https://doi.org/10.1145/2814228.2814249*

Tomas Petricek. 2018. What we talk about when we talk about monads. The Art, Science, and Engineering of Programming *2, 3 (2018), 12.*

Tomas Petricek, Dominic Orchard, and Alan Mycroft. 2014. Coeffects: A Calculus of Context-dependent Computation. In Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming *(Gothenburg, Sweden)* (ICFP '14). *ACM, New York, NY, USA, 123–135. 978-1-4503-2873-9 https://doi.org/10.1145/2628136.2628160*

B.C. Pierce. 2002. Types and Programming Languages. *MIT Press. 9780262162098 2001044428*

Robert M Pirsig. 1999. Zen and the art of motorcycle maintenance: An inquiry into values. *Random House.*

Andrew M. Pitts. 2011. Lecture Notes on Types for Part II of the Computer Science Tripos. https://www.cl.cam.ac.uk/teaching/1112/Types/typ-notes.pdf

P. J. Plauger. 1993. Programming on purpose. *PTR Prentice Hall, Englewood Cliffs, N.J. 9780133281057 92045905*

C. Pleasant, James. 1989. The Very Idea (Technical Correspondence). Commun. ACM *32, 3 (mar 1989), 374–381. 0001-0782 https://doi.org/10.1145/62065.315927*

Gordon Plotkin and Matija Pretnar. 2009. Handlers of algebraic effects. In European Symposium on Programming. *Springer, 80–94.*

Michael Polanyi. 1958. Personal knowledge. *University of Chicago Press. 0-226-67288-3*

John A. Postley. 1960. Letters to the Editor. Commun. ACM *3, 1 (Jan. 1960), 0.06–. 0001-0782 https://doi.org/10.1145/366947.366948*

M. Priestley. 2011. A Science of Operations: Machines, Logic and the Invention of Programming. *Springer London. 9781848825550 2011921403*

Mark Priestley. 2017. AI and the origins of the functional programming language style. Minds and Machines *27, 3 (2017), 449–472.*

PROGRAMme. 2022. What is a computer program? *In preparation.*

David Pym, Jonathan M Spring, and Peter O'Hearn. 2019. Why separation logic works. Philosophy & Technology *32, 3 (2019), 483–516.*

George Radin. 1978. The Early History and Characteristics of PL/I. SIGPLAN Not. *13, 8 (Aug. 1978), 227–241. 0362-1340 https://doi.org/10.1145/960118.808389*

B Randell. 1975. System structure for software fault tolerance. IEEE Transactions on Software Engineering *SE-1, 2 (June 1975), 220–232. 0098-5589 https://doi.org/10.1109/TSE.1975.6312842*

Eric S Raymond. 1996. The new hacker's dictionary. *MIT Press.*

Eric S Raymond. 1997. The cathedral and the bazaar. *O'Reilly Media. 978-0596001087*

Eric S Raymond. 2003. The art of UNIX programming. *Addison-Wesley Professional.*

Kent C Redmond and Thomas M Smith. 2000. From whirlwind to MITRE: The R&D story of the SAGE air defense computer. *MIT Press.*

Trygve M. H. Reenskaug. 1981. User-oriented descriptions of Smalltalk systems. Byte, *6 8 (1981).*

Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, et al. 2009. Scratch: programming for all. Commun. ACM *52, 11 (2009), 60–67.*

John C. Reynolds. 1974. Towards a theory of type structure. In Programming Symposium, *B. Robinet (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 408–425. 978-3-540-37819-8*

Howard Rheingold. 2000. Tools for thought: The history and future of mind-expanding technology. *MIT press.*

Gregor Richards, Francesco Zappa Nardelli, and Jan Vitek. 2015. Concrete Types for TypeScript. In 29th European Conference on Object-Oriented Programming (ECOOP 2015), *John Tang Boyland (Ed.), Vol. 37. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 76–100. 978-3-939897-86-6 1868-8969 https://doi.org/10.4230/LIPIcs.ECOOP.2015.76*

Nathaniel Rochester. 1960. Foreword. In Papers Presented at the December 13-15, 1960, Eastern Joint IRE-AIEE-ACM Computer Conference *(New York, New York)* (IRE-AIEE-ACM '60 (Eastern)). *Association for Computing Machinery, New York, NY, USA, 1–9. 9781450378710 https://doi.org/10.1145/1460512.1460513*

Daniel Rosenwasser. 2018. Announcing TypeScript 3.0. https://blogs.msdn.microsoft.com/typescript/2018/07/30/announcing-typescript-3-0/ [Online; accessed 10-September-2018].

Douglas T Ross. 1961. A generalized technique for symbol manipulation and numerical calculation. Commun. ACM *4, 3 (1961), 147–150.*

Winston W. Royce. 1970. Managing the Development of Large Software Systems. In Proceedings IEEE WESCON. *1–9.*

John M Rushby and Friedrich Von Henke. 1993. Formal verification of algorithms for critical systems. IEEE Transactions on Software Engineering *19, 1 (1993), 13–23.*

Bertrand Russell. 1903. Appendix B: The doctrine of types. Principles of mathematics *(1903), 523–528.*

Bertrand Russell. 1908. Mathematical Logic as Based on the Theory of Types. American Journal of Mathematics *30, 3 (1908), 222–262. 00029327, 10806377*

Stephen Russell. 2017. Interview with Stephen (Steve) "Slug" Russell, conducted by Christopher Weaver, January 8, 2017. *Technical Report. Video Game Pioneers Oral History Collection, Archives Center, National Museum of American History, Smithsonian Institution, Washington, DC.*

S. R. Russell. 1963. Improvements in LISP Debugging. *Technical Report AIM-10. Stanford Artificial Intelligence Project. https://www.softwarepreservation.org/projects/LISP/stanford/Stanford-AIM-10.pdf*

Barbara G. Ryder, Mary Lou Soffa, and Margaret Burnett. 2005. The Impact of Software Engineering Research on Modern Progamming Languages. ACM Trans. Softw. Eng. Methodol. *14, 4 (Oct. 2005), 431–477. 1049-331X https://doi.org/10.1145/1101815.1101818*

Warren Sack. 2019. The software arts. *MIT Press.*

Jean E. Sammet. 1961. Detailed Description of COBOL. In Annual Review in Automatic Programming, *Richard Goodman (Ed.). International Tracts in Computer Science and Technology and Their Application, Vol. 2. Elsevier, 197–230. 0074-9141 https://doi.org/10.1016/B978-1-4831-9779-1.50013-6*

Jean E. Sammet. 1978. The Early History of COBOL. SIGPLAN Not. *13, 8 (Aug. 1978), 121–161. 0362-1340 https://doi.org/10.1145/960118.808378*

Peter Samson. 2017. Interview with Peter Samson, conducted by Christopher Weaver, January 9, 2017. *Technical Report. Video Game Pioneers Oral History Collection, Archives Center, National Museum of American History, Smithsonian Institution, Washington, DC.*

Stephen A. Schuman and Philippe Jorrand. 1970. Definition Mechanisms in Extensible Programming Languages. In Proceedings of the November 17-19, 1970, Fall Joint Computer Conference *(Houston, Texas)* (AFIPS '70 (Fall)). *ACM, New York, NY, USA, 9–20. https://doi.org/10.1145/1478462.1478465*

Ken Schwaber. 1997. Scrum development process. In Business object design and implementation. *Springer, 117–134.*

Dana S. Scott. 1993. A type-theoretical alternative to ISWIM, CUCH, OWHY. Theoretical Computer Science *121, 1 (1993), 411–440. 0304-3975 https://doi.org/10.1016/0304-3975(93)90095-B*

Mark Seemann. 2021. Code That Fits in Your Head: Heuristics for Software Engineeing. *Addison-Wesley Professional. 978-0137464401*

Gilbert Simondon. 2016. On the Mode of Existence of Technical Objects. *Univocal Publishing. 978-1-937561-03-1*

R. Slayton. 2013. Arguments that Count: Physics, Computing, and Missile Defense, 1949-2012. *MIT Press. 9780262019446 2012051748*

Brian Cantwell Smith. 1985. Limits of Correctness in Computers. *Technical Report CSLI-85-36. The Center for the Study of Language and Information, Stanford, CA.*

David Canfield Smith. 1977. Pygmalion: A computer program to model and stimulate creative thought. *Birkhauser.*

C. Solis and X. Wang. 2011. A Study of the Characteristics of Behaviour Driven Development. In 2011 37th EUROMICRO Conference on Software Engineering and Advanced Applications. *383–387. 1089-6503 https://doi.org/10.1109/SEAA.2011.76*

N. SOLNTSEFF and A. YEZERSKI. 1974. A Survey of Extensible Programming Languages. In Computer Science and Technology and their Application, *MARK I. HALPERN and WILLIAM C. McGEE (Eds.). International Tracts in Computer Science and Technology and Their Application, Vol. 7. Elsevier, 267–307. 0074-9141 https://doi.org/10.1016/B978-0-08-017806-6.50010-9*

Cynthia Solomon, Brian Harvey, Ken Kahn, Henry Lieberman, Mark L. Miller, Margaret Minsky, Artemis Papert, and Brian Silverman. 2020. History of Logo. Proc. ACM Program. Lang. *4, HOPL (2020), 79:1–79:66. https://doi.org/10.1145/3386329*

Susan Leigh Star and James R. Griesemer. 1989. Institutional Ecology, 'Translations' and Boundary Objects: Amateurs and Professionals in Berkeley's Museum of Vertebrate Zoology, 1907-39. Social Studies of Science *19, 3 (1989), 387–420. https://doi.org/10.1177/030631289019003001*

Guy L. Steele and Richard P. Gabriel. 1996. History of Programming languages—II. ACM, New York, NY, USA, Chapter The Evolution of Lisp, 233–330. 0-201-89502-1 https://doi.org/10.1145/234286.1057818

Thomas G Stockham and Jack B Dennis. 1960. FLIT-Flexowriter Interrogation Tape: A Symbolic Utility Program for the TX-0. *Technical Report M-5001-23. Massachusetts Institute of Technology.*

Bjarne Stroustrup. 1996. A history of C++ 1979–1991. In History of programming languages—II. *699–769.*

Ivan Edward Sutherland. 1963. Sketchpad, a man-machine graphical communication system. *Ph. D. Dissertation. MIT.*

William R Sutherland. 1966. On-line Graphical Specification of Computer Procedures. *Ph. D. Dissertation. MIT, Lincoln Labs Report TR-405.*

Don Syme. 2020. The early history of F#. Proceedings of the ACM on Programming Languages *4, HOPL (2020), 1–58.*

Donald Syme, Keith Battocchi, Kenji Takeda, Donna Malayeri, and Tomas Petricek. 2013. Themes in Information-rich Functional Programming for Internet-scale Data Sources. In Proceedings of the 2013 Workshop on Data Driven Functional Programming *(Rome, Italy)* (DDFP '13). *ACM, New York, NY, USA, 1–4. 978-1-4503-1871-6 https://doi.org/10.1145/2429376.2429378*

Hirotaka Takeuchi and Ikujiro Nonaka. 1986. The new new product development game. Harvard business review *64, 1 (1986), 137–146.*

J.P. Talpin and P. Jouvelot. 1994. The Type and Effect Discipline. Information and Computation *111, 2 (1994), 245 – 296. 0890-5401 https://doi.org/doi.org/10.1006/inco.1994.1046*

Czander Tan. 2020. The Poetics of Computer Code: Tracing Digital Inscription in Ada Lovelace's England. Digital Studies/Le champ numérique *10, 1 (2020).*

Matti Tedre. 2014. The science of computing: shaping a discipline. *CRC Press.*

Warren Teitelman. 1965. EDIT and BREAK functions for LISP. *Technical Report AIM-84. MIT. http://www.bitsavers.org/pdf/mit/ai/aim/AIM-084.pdf*

Warren Teitelman. 1966. PILOT: a step toward man-computer symbiosis. *Ph. D. Dissertation. MIT.*

Warren Teitelman. 1974. INTERLISP Reference Manual. *Technical Report. Xerox Palo Alto Research Center. http://www.softwarepreservation.org/projects/LISP/interlisp/Interlisp-Oct_1974.pdf*

Warren Teitelman. 1984. The Cedar programming environment: A midterm report and examination. *Technical Report. Xerox Palo Alto Research Center.*

The Free Software Foundation. 2022. What is Free Software? https://www.gnu.org/philosophy/free-sw.en.html, Retrieved 3 July 2022.

Thomas Haigh. 2010. Dijkstra's Crisis: The End of Algol and Beginning of Software Engineering, 1968-72. http://tomandmaria.com/Tom/Writing/DijkstrasCrisis_LeidenDRAFT.pdf [Online; accessed 27-August-2021].

Mads Tofte and Jean-Pierre Talpin. 1997. Region-based memory management. Information and computation *132, 2 (1997), 109–176.*

James E Tomayko. 1988. Computers in Spaceflight. *Technical Report. NASA contractor report CR-182505, National Aeronautics and Space Administration, Scientific and Technical Information Division, Washington, DC, USA.*

Christopher Tozzi. 2017. For fun and profit: A history of the free and open source software revolution. *MIT Press.*

Alan Turing. 1949. Checking a large routine. In Report of a Conference on High Speed Automatic Calculating Machines. *University Mathematical Laboratory, Cambridge, 67–69.*

Fred Turner. 2010. From counterculture to cyberculture. *University of Chicago Press.*

Univac. 1957. Introducing a New Language for Automatic Programming Univac Flow-Matic. http://www.computerhistory.org/collections/catalog/102646140 Computer History Museum, catalog no. 102646140 [Online; accessed 10-September-2018].

Alasdair Urquhart. 1988. Russell's Zig-Zag Path to the Ramified Theory of Types.

A. van Wijngaarden, B. J. Mailloux, J. E. L. Peck, and C. H. A. Koster. 1969. Report on the Algorithmic Language ALGOL 68. *Springer Berlin Heidelberg, Berlin, Heidelberg, 80–218. 978-3-662-39502-8 https://doi.org/10.1007/978-3-662-39502-8_1*

Panagiotis Vekris, Benjamin Cosman, and Ranjit Jhala. 2016. Refinement Types for TypeScript. In Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation *(Santa Barbara, CA, USA)* (PLDI '16). *ACM, New York, NY, USA, 310–325. 978-1-4503-4261-2 https://doi.org/10.1145/2908080.2908110*

John von Neumann and HH Goldstine. 1947. Planning and Coding of Problems for an Electronic Computing Instrument, Part II, vol. 1. *Technical Report. Institute for Advanced Study, Princeton.*

M. Mitchell Waldrop. 2001. The Dream Machine: J.C.R. Licklider and the Revolution That Made Computing Personal. *Viking Penguin. 0670899763*

Adrian Ward, Julian Rohrhuber, Fredrik Olofsson, Alex McLean, Dave Griffiths, Nick Collins, and Amy Alexander. 2004. Live algorithm programming and a temporary organisation for its promotion. In Proceedings of the README Software Art Conference, *Vol. 289. 290.*

William A. Whitaker. 1993. Ada—the Project: The DoD High Order Language Working Group. SIGPLAN Not. *28, 3 (March 1993), 299–331. 0362-1340 https://doi.org/10.1145/155360.155376*

M.V. Wilkes. 1985. Memoirs of a Computer Pioneer. *MIT Press. 9780262231220 lc85006667*

Maurice Vincent Wilkes, David J Wheeler, and Stanley Gill. 1951. The Preparation of Programs for an Electronic Digital Computer: With special reference to the EDSAC and the Use of a Library of Subroutines. *Addison-Wesley Press.*

Gerald M. Winberg. 1971. The Psychology of Computer Programming. *Van Nostrand Reinhold.*

N. Wirth. 1971. The programming language pascal. Acta Informatica *1, 1 (01 Mar 1971), 35–63. 1432-0525 https://doi.org/10.1007/BF00264291*

N. Wirth. 1996. Recollections about the Development of Pascal. *Association for Computing Machinery, New York, NY, USA, 97–120. 0201895021 https://doi.org/10.1145/234286.1057812*

Scott Wlaschin. 2018. Domain Modeling Made Functional: Tackle Software Complexity with Domain-Driven Design and F#. *Pragmatic Bookshelf.*

Hongwei Xi and Frank Pfenning. 1999. Dependent types in practical programming. In Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. *214–227.*

S. S. Yau and R. C. Cheung. 1975. Design of Self-checking Software. In Proceedings of the International Conference on Reliable Software *(Los Angeles, California). ACM, New York, NY, USA, 450–455. https://doi.org/10.1145/800027.808468*

E. Yourdon. 1975. Techniques of Program Structure and Design. *Prentice-Hall. 9780139017025 75009728 https://books.google.cz/books?id=T6FQAAAAMAAJ*

Yuriy Yushchenko. 2022. Pointers in programs on the computer MESM. Originalhttp://www.icfcst.kiev.ua/MUSEUM/TXT/YuriYushchenko.pdf For the European Virtual Computer Museum of the History of Information Technologies in Ukraine.

G Pascal Zachary. 2018. Endless frontier: Vannevar Bush, engineer of the American century. *Simon and Schuster.*

Johannes Zmölnig and Gerhard Eckel. 2007. Live coding: an Overview. In Proceedings of the 2007 International Computer Music Conference, ICMC 2007, Copenhagen, Denmark, August 27-31, 2007. *Michigan Publishing. http://hdl.handle.net/2027/spo.bbp2372.2007.063*

. . 1963. . .