3 Interactive Programming

Teacher: We just saw what becomes possible when we think of computer programs as mathematical entities, but that view leaves out the interaction between a human and a computer through which programs are created. Can we approach programs from the opposite perspective and talk about interactive programming?

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

Pythagoras: Programming environments that you use to create programs are interactive, but they support writing of programs in a programming language. Even sophisticated environments for languages like Java are still built around languages.

Socrates: Your assumptions about what programming is determine your answer! We take for granted that program is a static text in some language and we forget that alternative views are possible. Thinking about programming as an interactive process reveals a different perspective and I'm glad we can explore it here!

Archimedes: Are you thinking of something like the famous 1969 demo by Douglas Engelbart, retrospectively known as 'The Mother of All Demos', where he presented his oN-Line System (NLS) that featured mouse, video-conferencing, hypertext and many other aspects of modern interactive computers?

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

Diogenes: It is also starting way too late. The first real interactive programming was done at MIT once the hackers[1] 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. This was the first time you could program a computer interactively through a terminal.

Xenophon: You can hardly call toying with TX-0 programming! The TX-0 was an experimental machine to test transistor technology for the SAGE defence system. The Whirlwind computer that was used at SAGE before was already interactive, but it was programmed properly, through a carefully controlled process.[2] And anyway, sitting in front of a console in the late 1950s would not scale up as you would be wasting scarce computing resources. You had to do your thinking away from the machine.

Pythagoras: 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 was soon used in the interactive way on time-sharing systems.

Teacher: This raises the question as to whether the emergence of interactive programming enabled new ways of using computers?

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:07:52, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.004

Archimedes: Easier and more efficient? Yes. You could get quick feedback and other help from your programming tools. But conceptually, you still have programmers who create programs, mostly by writing source code, and users who work with those. . .

Socrates: I disagree. Interactivity was the key component of the 'man–computer symbiosis' vision that saw computers as tools for thinking or even as a new kind of literacy. Douglas Engelbart's work on 'augmenting human intellect' places this way of thinking in the broader West coast counterculture movement, whose experiments with LSD used a similar terminology of augmenting the human mind and consciousness.[3] This vision, largely funded by ARPA, inspired a whole new generation of technologists!

Diogenes: Man–computer symbiosis is exactly what you do when you program by directly engaging with the computer. Lisp systems in the 1960s made it possible to construct programs gradually by telling the system how to change the program operation.[4] This is a logical step up from modifying the code at textual level!

Archimedes: Isn't this just about building good engineering tools though? Smalltalk, which came out of the same community in the 1970s introduced many of those. It pioneered object-oriented programming and the use of a graphical programming environment that looked much like the Integrated Development Environments (IDEs) of modern programming languages today.

Socrates: You are looking at Smalltalk from a present-day engineering perspective, but this is not how it was thought about at the time. Smalltalk was a part of the Dynabook project, which was to be a 'personal computer for children of all ages'[5] (Figure 3.1) that would let children experience the excitement of thought and creation!

Xenophon: 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?[6]

Socrates: Smalltalk was envisioned 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. . .

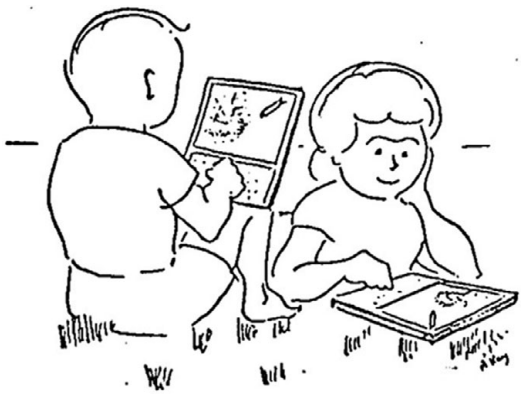


Figure 3.1 A sketch of children using the Dynabook from Alan Kay's 1972 essay.[7] The two 9-year-olds are able to connect their computers, play a game and reprogram its logic. (Source: Courtesy of Alan Kay. Reproduced with permission)

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:07:52, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.004

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

Xenophon: In practice, you do not need a 'meta-medium'. You 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.

Socrates: Realising the Smalltalk vision is not in the commercial interest of big technology companies. We may have tablets that look like the Dynabook, but you cannot reprogram them from within themselves and share your creation with others! And even Xerox PARC had to label its work as the 'office of the future'.

Xenophon: What is wrong with that? The Xerox Star system that Xerox eventually built in the 1980s (Figure 3.2) failed commercially, because it was too expensive, but it gave you something that was remarkably close to computer systems that office workers used in the 2000s.

Socrates: The problem is that Xerox Star removed all that made Smalltalk interesting! It adopted the desktop metaphor 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.

Diogenes: The vision of a system that you can understand and improve is compelling, but even the earlier systems built at Xerox PARC suffered from a growing complexity at both the hardware and software level.[9] For one thing, it required specialised hardware that existed only inside Xerox PARC!

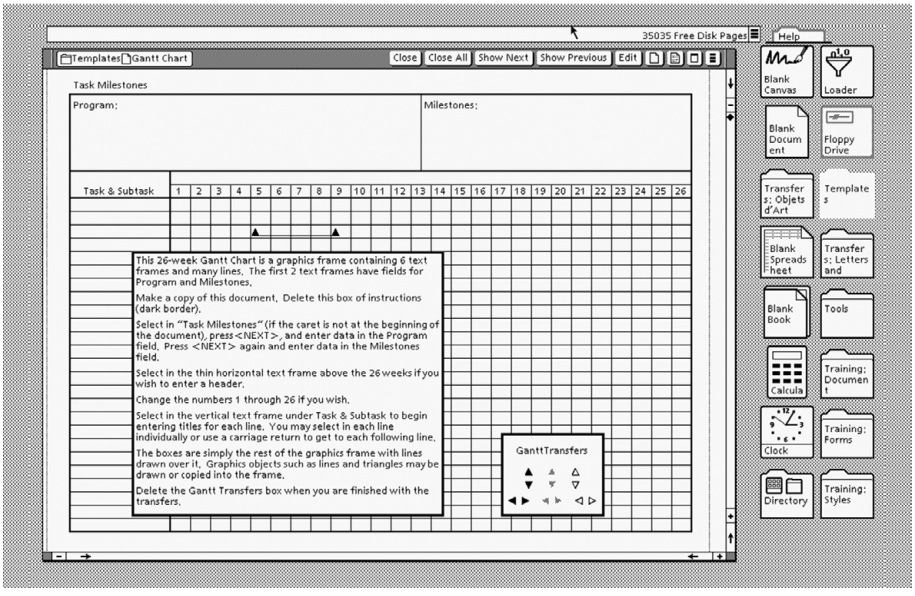


Figure 3.2 Xerox Star desktop showing a template for a Gnatt chart on the left and desktop with further icons on the right. (Source: Screenshot created by the author, using the Darkstar emulator[8])

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:07:52, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.004

Teacher: Was Smalltalk a dead end for interactive programming then? What were the next steps? Did it influence subsequent systems?

Archimedes: 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. . .

Diogenes: I'm sure these are useful if you think building software is like building bridges! Computers are more flexible and interactive programming is a way to directly control them. In the 1970s, this became clear with the rise of microcomputers, which were small, inexpensive and anyone could build one in their own garage!

Xenophon: You mean toys like the Altair 8800 microcomputer? You had to program that by flipping switches and through a teletype terminal like the TX-0! I do not see how this was the next step for interactive programming. . .

Diogenes: Right, but microcomputers were simple, easy to improve and attracted a huge community of hobbyists. Very soon, the microcomputers of the '1977 trinity' all had screens and an interactive programming environment based on BASIC.[10]

Socrates: This is far from the graphical interfaces of Xerox systems, but I see the connection. Many microcomputers booted directly into the 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.[11]

Pythagoras: It is a shame that microcomputers adopted BASIC and not a sensible programming language like Algol. I agree with Dijkstra,[12] who once wrote that BASIC mentally mutilated a generation of programmers beyond hope of regeneration!

Socrates: 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.

Xenophon: 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. . .

Socrates: Great interactive system, ruined again by commercial interests!

Teacher: Care to elaborate?

Socrates: Interactive programming often emerges with new technologies, like the TX-0, desktop computers at Xerox and microcomputers. They give users more control over the computers and their programming. Businesses find this either inefficient or hard to commercialise, so they repeatedly move to a more closed model where programming software is clearly separated from using software.

Teacher: Has modern software development adopted the closed non-interactive model?

Archimedes: Sorry for repeating myself, but modern software engineering adopted many ideas from interactive programming. This includes interactive shells[13] to prototype ideas and Test-Driven Development (TDD) to get quick feedback on your code edits. Those are all great interactive programming tools!

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:07:52, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.004

Socrates: This is relegating interactive programming to a secondary role. Live coding where you interactively control music synthesis and visual effects does much more. It turns programming into a performance and probes how efficient the interaction can be, because you have to respond to what other musicians are doing.[14]

Pythagoras: This is an odd example, but I have a more sensible one. If you look at how data analysts work in Jupyter notebooks, it is very interactive. You load some data into memory, run various commands to process it, immediately visualise the results and then refine your scripts. Interactivity makes a lot of sense in such a specialised context.

Teacher: There seem to be more examples of interactive programming today than I thought, but they pop up in unexpected contexts.

Socrates: Well, that's what I was saying earlier! Interactive programming often emerges with new technologies, be it microcomputers, data analytics or live coding! Alas, managerial and engineering motivations often call for less flexible but easier to control methods, and so interactive programming is replaced with specialised tools. I just wonder how soon this will happen to data analytics and live coding!

3.1 Spacewar!

At the turn of the 1960s, most computer programming was rather dull. You punched your program on a deck of 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.'[15] 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 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 3.3), a game inspired by the 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 Hingham 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) talked about the idea with other MIT computer hackers. He hoped someone else would write the game and he'd just play so he kept looking for excuses not to do the work himself. He knew he needed sine and cosine routines,

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:07:52, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.004

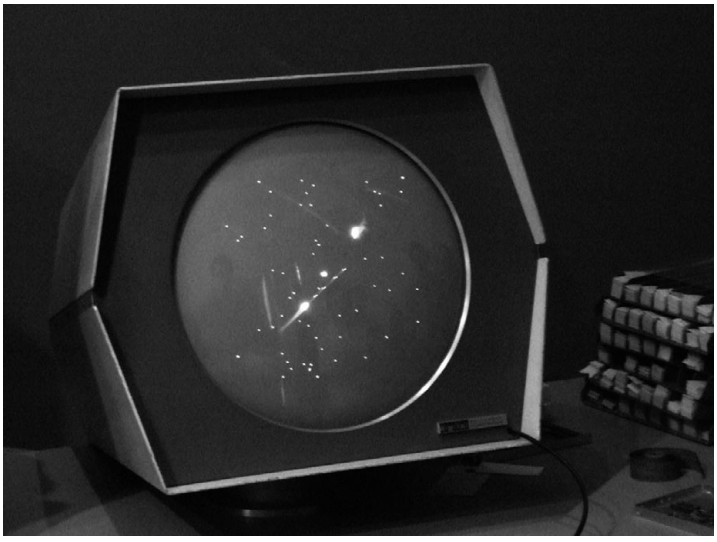


Figure 3.3 Spacewar! game running on PDP-1 at the Computer History Museum in 2007. (Source: Computer History Museum, Photo by Kenneth Lu,[16] CC-BY 2.0)

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?'[17]

After some more hesitation, Steve Russell did write the first version of Spacewar! The game became available to everyone else with access to PDP-1 and, after some debugging and polishing, it became 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.[18] For this chapter, it is interesting for two other reasons. First, Spacewar! was created interactively. 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. Its programmers were using the terminal to modify the code and the screen to test the game. This may not seem surprising today, but it was surprising in the early 1960s when there were just two computers in the whole department, each the size of a room. To use the machine, you would put your name on a signup sheet. Professors and graduate students had some number of allocated hours they could book, but when no one 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.

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:07:52, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.004

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.[19] 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 calculate 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 the 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 had 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 hacker ethic encourages sharing of knowledge, but the way it spreads 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.

3.2 Transistorised Experimental Computer Zero

Towards the end of World War II, the United States Military approached MIT with the problem of designing a flight simulator that could be used to test designs of new airplanes and analyse their aerodynamic stability. In the 1940s, the leading 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 the development of Whirlwind, a new digital computer designed to solve the problem.[20]

Whirlwind became operational in 1951. Its application domain posed a number of interesting challenges and as a result, Whirlwind was a revolutionary machine in many ways. Instead of completing individual computational tasks in batches, it had to be able to accept real-time inputs and produce real-time outputs. It also needed a graphical output system and interactive capabilities. Whirlwind 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. The

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:07:52, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.004

most prominent among them was in the enormous cold war SAGE (Semi-Automatic Ground Environment) air defence system.[21] 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 'transistorised experimental computer zero'. After a successful test, the computer was no longer needed and was transferred to the MIT Research Laboratory of Electronics (RLE) on a long-term loan in 1958. Here, it became available for use to qualified users, including the members of the TMRC, 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 monitored 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 unforeseen event had not been taken into account.[22]

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 3.4). This enabled the user to manually set the values of sixteen 18-bit registers in an auxiliary memory storage or set values directly in the magnetic core memory. However, the high-speed large capacity magnetic core memory made possible 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).[23]

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 Chapter 4. A more outlandish utility was developed for debugging a long-running voice recognition software. The utility enabled the user to follow the control path through a program in a flowchart, using a rather elaborate method that involved drawing the flow-chart by hand on a transparent foil and placing it over the computer screen.[24] It also used the toggle switches to slow down or speed up the program.

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:07:52, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.004



Figure 3.4 TX-0 with the control panel, including the Toggle Switch Storage, 12 1/2" cathode-ray oscilloscope display tube and Flexowriter teletype terminal behind the operator. (Source: Courtesy of the Computer History Museum)

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 and after midnight, he was able to get direct access to it. 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 TMRC, 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 could be achieved 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 programs 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.[25]

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:07:52, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.004

The hacker culture forming around TX-0 encouraged information sharing that had lasting effect on the software industry. It enabled the collaboration around Spacewar! but it also inspired the influential Free Software movement. The limited capabilities of the TX-0 also gave rise to a certain hacker aesthetic.[26] 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,[27] which presented a collection of random MIT hacker tricks a decade later.

3.3 Symbol Manipulating Language

The first experiments with interactive programming were happening at the same time as another development that was making programming easier. This was the emergence of high-level programming languages in the second half of the 1950s. I already wrote about the FORTRAN formula translator, FLOW-MATIC and COBOL that focused on business data processing and ALGOL 58 that made 'programming language' into an independent mathematical entity. All of these were used in the tedious batch-processing mode and did not appeal to the hacker aesthetic. Hackers would eventually find their high-level programming language of choice in Lisp. To follow its origins, we need to get back to a workshop organised 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 chose the term Artificial Intelligence for the name of the workshop, coining 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 Russell'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.[28]

Logic Theorist was implemented in a programming language called 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 language 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 a good fit for his own research, but he also wanted a language that would follow the intuitive algebraic notation of FORTRAN. McCarthy explored the idea of implementing support for list processing in FORTRAN in his role 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

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:07:52, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.004

MIT, McCarthy recruited a group of graduate students and hackers to create Lisp, a more flexible list processing language, for the IBM 704 machine.

Although Lisp was the brainchild of John McCarthy, the language has origins in multiple cultures of programming. This is partly because McCarthy himself belonged and contributed to multiple cultures. His example shows not only that an individual can move between cultures of programming, but also that 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.

As we have seen in Chapter 2, McCarthy later played a pivotal role in the mathematical culture of programming and Lisp adopted the lambda notation for functions from Church's lambda calculus. However, McCarthy himself also acknowledged that the notation was the only influence as he had not studied lambda calculus before designing Lisp.[29] The Lisp language 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 of the early artificial intelligence research. This provided a new kind of programming problem, based on thinking about human thinking, an approach that inspired much work in the humanistic culture of programming that I will discuss later in this chapter.

Finally, 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 group 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 realisation of a hacker was at the source of one of the unique Lisp characteristics that is now referred to as 'meta-circularity', that is, the fact that it can largely be 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'.[30]

3.4 Augmenting Human Intellect

Before we continue with the story of Lisp and interactive programming, it will be useful to return to the background of a culture that influenced much work on interactive programming. The proponents of this culture 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. For this reason, I choose to name it the humanistic culture of programming. Unlike, for example, the mathematical culture, humanistic thinking about programming involves a wider range of perspectives. In many ways, the multiplicity of views on programming is one of the defining characteristics of the culture. The work on artificial intelligence, which was concerned with modelling of human thinking, was one of the reasons for the birth of Lisp. At the same

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:07:52, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.004

time, interest in human creativity inspired programming languages for education and work on computer art.

As the starting point for characterising 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 licence linked to Romanticist ideas through the work of Ada Lovelace,[31] 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.[32]

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 commercialising some of his work. He led a team that built the differential analyser, 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 organisations 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 that 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',[33] Bush introduces the *memex*, a hypothetical device that would store all of an individual's books, records and communications and enable navigation 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 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 of many interactive programming tools a decade later. In 1960, Licklider wrote the second landmark essay of our story, 'man–computer symbiosis'.[34] The essay presented a vision of artificial intelligence where the computer is not replacing a 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'. The essay was accessible, widely read and set a well-defined agenda for the humanistic culture of programming.

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:07:52, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.004

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 the Information Processing Techniques Office (IPTO). To Jack P. Ruina, the director of ARPA, who recruited him to lead IPTO, Licklider was known for his work on human factors of SAGE, but his vision of man-computer symbiosis also resonated well with the desire for using computers in military command and control systems.[35]

Licklider never directly collaborated with Vannevar Bush, but he worked on a project studying 'Libraries of the Future'[36] 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 humanistic vision of the future where computers are used to augment human mental capabilities, be it by assisting them with information organisation 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 computers to this day. One of the earliest examples of this was the Sketchpad system developed by Ivan Sutherland in 1963 and presented in his PhD thesis, 'Sketchpad: A man-machine graphical communication system'. The system enabled 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 ran 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 utilised as the primary input device.

Sketchpad went beyond just drawing. It supported editing, specification of constraints (e.g., two lines are parallel) and the 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), a topic I return to in Chapter 6.

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 subtitle of Sutherland's thesis points to the two organisms, collaborating in a symbiotic relationship. As put by Sutherland himself, Sketchpad provides a more efficient way 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.

Sketchpad envisioned using graphics for drawing and sketching of diagrams, but not for programming the computer itself. It did not take long for this idea to

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:07:52, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.004

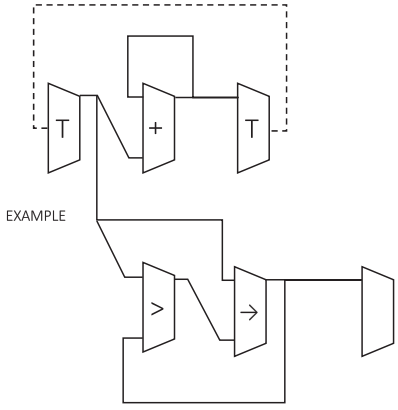


Figure 3.5 A visual program that accepts numbers from terminal and prints their running sum and the maximum, as constructed using Sutherland's system.[37]

ensue. A follow-up project by Bert Sutherland, an older brother of Ivan, added this aspect and described a visual language for 'online graphical specification of computer procedures'[38] (Figure 3.5) in his own PhD thesis in 1966, three years after his younger brother. Not long after, the idea of using flowcharts for visual programming was implemented in the GRAIL system that used a new tablet device.[39]

In the early 1960s, Sketchpad was probing the limits of what it was possible to do. To offer a glimpse of personal computing, it required the full power of the 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 realise this was Wes Clark, the engineer who designed TX-0 and TX-2 and believed that personal computers were the future of interactive computing. Clark set out to build such a 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.[40]

Despite the enthusiastic reception, LINC was only moderately successful. The team continued their work and built the first dozen 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 that the cost of computers would drop enough to make the vision a reality. 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 the 1960s, but it only gained prominence with the work at Xerox PARC in the 1970s that I return to later in this chapter. Meanwhile at MIT,

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:07:52, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.004

the work on interactive programming was taking a different, perhaps less visionary but more practical trajectory.

3.5 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 $1.2M in 2024 in terms of purchasing power). At MIT, the cost of PDP-1 inspired a joking series of program names such as 'Expensive Planetarium' (for the code to draw Spacewar! background), 'Expensive Synthesiser' (for the music playing program) and 'Expensive Typewriter' (for an early interactive text editor). At most other installations, the cost of computer time 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 the 1950s, was to let multiple users 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 had just moved from Dartmouth to MIT, started to call this technique 'time-sharing'.[41] To him, the idea 'seemed quite inevitable' and he thought that 'everybody had in mind to do' it.[42] 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 MIT was being given by IBM. He saw time-sharing primarily as a way to 'reduce the time required to get a problem solved on a machine'. To get support for his plan, he first had to convince the MIT Computation Centre, which managed access to most computers owned by MIT. 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 McCarthy read Licklider's essay on man-computer symbiosis, he

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:07:52, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.004

thought 'the whole notion was obvious'.[43] 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,[44] but the efforts eventually succeeded. At MIT, the first major implementation was the Compatible Time-Sharing System (CTSS), which was developed as part of the ARPA-funded Project MAC. By the mid 1960s, it ran 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 computers with display screens, but they were good enough to enable interactive use of high-level languages like Lisp. This gave the MIT hackers a 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.

3.6 A Step towards 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 labelled NYBOL1 and six cards labelled 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 soon made it possible to use Lisp interactively via the Lisp-Flexo system, using the same kind of terminal as when programming the TX-0. The system evaluated entire Lisp expressions entered by the user. The designers of the system recognised that typing a whole expression at once is difficult and offered the so-called TEN-Mode in which an expression could be entered in smaller chunks into 10 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.[45]

This started in 1963, when L. Peter Deutsch created an interactive Lisp for PDP-1 at Bolt Beranek and Newman (BBN). Deutsch realised that an interactive way of programming requires 'tools for rapid modification of already existing LISP programs within the LISP system'[46] 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 gave programmers a way 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:[47]

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:07:52, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.004

```
(LAMBDA (X) Y (COND ((NULL 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 list is empty `(NULL X)`, the function returns `Y`; otherwise, it takes the first element of `X` using `CAR X` and appends the remaining ones recursively 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 the parent list. The following then prints the expression, focuses on the second sub-list, prints it and then returns back (lines preceded by `*` 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 the n-th element of a list, so the following corrects the error by first removing the 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 the 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 only ever documented in manuals and various internal reports.[48] 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 a doctoral research project and so it was approaching the problem from a more academic context. In academic research in the 1960s, Lisp was the de facto language for

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:07:52, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.004

artificial intelligence (AI) 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.[49]

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' developed by J. C. R. Licklider. This is perhaps a necessity as work emerging from the hacker culture was rarely treated as being worthy of an academic publication, but it also suggests that humanistic visions were often easy to combine with the practice of hackers. The thesis further references systems like Sketchpad, further linking 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 enabled two kinds of modification. *Editing* was the process of changing individual procedures. The `EDIT` function let the user 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 procedures 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.

The notion of advising was later further developed by object-oriented systems for Lisp in the 1980s that I mention in Chapter 6, but the concept also inspired Aspect-Oriented Programming that would appear in the early 2000s. However, PILOT is 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 mathematisation of programming and the formalisations 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.[50]

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 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 parenthesisation. All such corrections were done in the Interlisp environment, so the corrected version was used next time the code was executed.[51]

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:07:52, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.004

The perspective that Lisp is a programming system rather than a language became even more dominant in the 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',[52] a class of computers that would be the dominant way of running Lisp in the 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. In many ways, the Lisp system fulfilled 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 would later become the bedrock of the 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'.[53]

Different cultures of programming also developed different dialects of Lisp. An interesting example is Scheme, created by Gerald Sussman and Guy Steele in 1975. Their initial motivation was to make sense of the Actor model developed by Carl Hewitt.[54] They decided to implement a toy implementation of the programming model in Lisp and adopted lexical scoping of Algol that Steele was studying at the time. The language initially supported a way of defining functions (lambda) and actors (alpha), but the two researchers soon discovered that the mechanisms are, in fact, the same.

The influence of the mathematical culture on the design of Scheme is easy to retrace, but there is also a more subtle methodological influence of the humanistic culture. In the artificial intelligence research, a common approach of understanding a behaviour was to try to write a program to simulate it.[55] Writing a dialect of Lisp to understand the Actor model follows this approach. Scheme can thus be seen as a product of the methodology of the humanistic culture with ideas of the mathematical culture. It was later used by the authors to explore more mathematical problems of programming languages. In particular, the work resulted in an influential series of papers on programming language semantics ('Lambda the Ultimate') and this was the main force for its success.[56]

Another system that takes a distinct perspective on Lisp was Logo, which emerged from the humanistic focus on education. I will return to Logo after a brief detour that is useful for explaining the typical characteristics of the humanistic culture.

3.7 There Should Be No Computer Art

The humanistic culture of programming is characterised by the concern for the relationship between the human and computer. The futuristic visions of man-computer

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:07:52, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.004

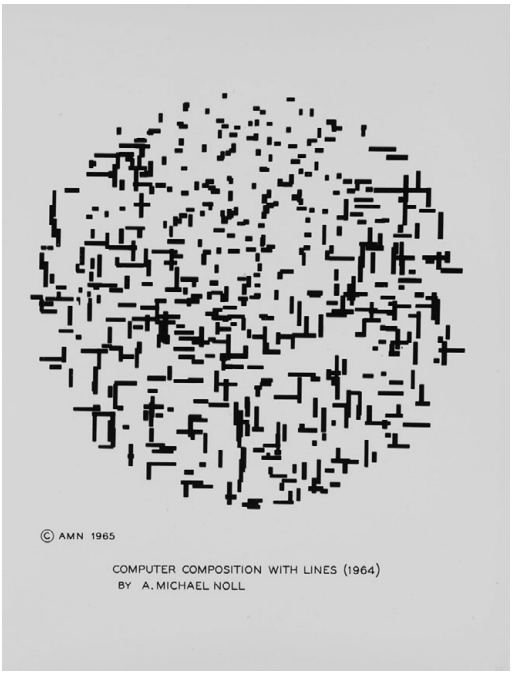


Figure 3.6 Computer Composition with Lines by A. Michael Noll (1964).[57]
(Source: Courtesy of A. Michael Noll. Reproduced with permission)

symbiosis that I discussed above are one instance of this, but the relationship can be probed and explored in a variety of 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 the 1950s, but the 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 3.6 shows, early computer generated art creatively used the limited capabilities of early computers and single-colour plotters and much of the work involved abstract art, utilising the basic shapes that computers could produce. Despite the limited computer capabilities, the work foretold a number of techniques that would appear in computer art later. In some cases, it also followed the humanistic methodology that I just discussed in the context of Scheme. For example, A. Michael Noll later recruited 100 participants for a study that compared Mondrian's 'Composition with Lines' with a picture he generated using a computer, pointing out that only 28 per cent were able to correctly identify the computer generated picture (and over a half preferred it). Using the program to generate the pattern thus served as the means for probing the nature of art and human perception of it.

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 society. Not long after the first exhibitions, Frieder Nake wrote a provocative essay

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:07:52, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.004

'There should be no computer art'[58] 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 by Nake and other computer art pioneers be a basis for a more positive transformation in society, rather than hanging on the walls of rich art collectors?

In the vision for children's 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 a co-founder of the field of artificial intelligence 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.[59]

Papert joined Minsky as a co-director of the MIT Artificial Intelligence Lab. He soon became fascinated by the hacker community, which now favoured Lisp as their programming language of choice. In the community, Papert found 'extraordinary examples of learning and problem solving that took place organically'.[60] 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 had previously taken a job as Marvin Minsky's secretary so that she could learn programming and was then, among other things, involved in the BBN-LISP project.

Papert was inspired by Piaget's theory of active learning, which posits 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 material' that would enable children to 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 could master, but it took another leap of thought to see programming as something that every child could learn. This step was taken by the group around Papert, Feurzeig and Solomon who started thinking about a programming language, eventually called Logo, that would make such learning possible.

3.8 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;

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:07:52, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.004

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.[61]

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.'[62]

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 language that would empower students to progress from a simple environment to the most powerful kind of programming available. The key design idea that made this progression possible 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 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 made children curious about the structure of natural languages and grammar:[63]

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

A better known 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 3.7). The Logo designers thought of a turtle as an object-to-think-with. Although Papert later said that a 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. When Cynthia Solomon later realised that young children had a 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

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:07:52, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.004

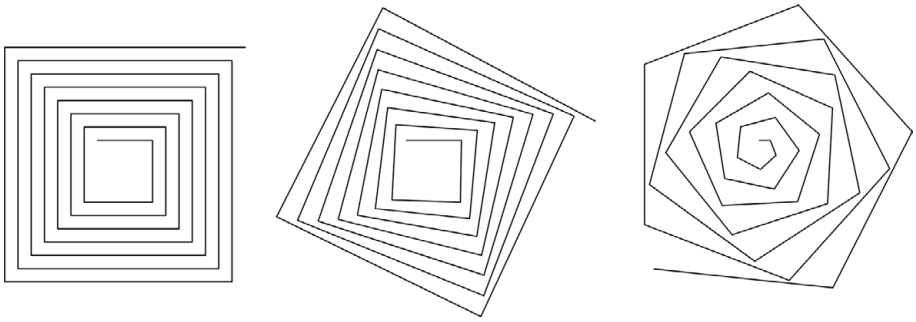


Figure 3.7 Drawing Turtle graphics using Logo. `SPI` is a recursive procedure that moves the turtle forward, rotates it and calls itself with a greater distance. The output shows interesting effects achieved by varying the angle on line 3.
(Source: Recreated by the author using Curly Logo[64])

and replaying such instructions. Second, the turtle metaphor also lets children debug programs by 'becoming the turtle' and literally 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', avoiding 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 anthropomorphisation (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, ideas. And it was a computer culture: a way of thinking about computers and about learning and about talking about what you were doing.'[65] Logo designers 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 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, support for the countercultural movement, an attempt to establish programmer as a more masculine professional or striving for academic respect.[66] 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 of the Logo language 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. In a hypothetical situation discussed in Papert's *Mindstorms* book, a child wants to draw a flock of six birds 'all the same way up'. When they see four rotated birds instead, they first note that the erroneous result is nice and should be kept for later, before proceeding to debug it by 'becoming the turtle'.

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:07:52, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.004

Logo designers were also conscious of the 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 of the response of early Logo implementations to an unknown command such as `love`. The initial response was 'love has no meaning', but the designers wanted to avoid telling such a 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 anthropomorphises the computer while the latter puts too much blame on the child.[67] 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. But those are then encoded as Logo programs, following again the humanistic theme of using programs to understand a problem.

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.

3.9 The Mother of All Demos

On 8 December, 1968, Douglas Engelbart presented the oN-Line System (NLS) at the Fall Joint Computer Conference in San Francisco. At a time when the most interactive way of using a 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'.

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 to return to university and enrol in a graduate program at Berkeley in 1951. After completing his studies, Engelbart 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, which he started to refer to as 'augmenting human intellect'.[68]

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:07:52, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.004

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'.[69] 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:

> [I]ncreasing the capability of a man to approach a 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 a mixture of the following: more-rapid comprehension, better comprehension, the possibility of gaining a useful degree of comprehension in a 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 of Engelbart's later 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 Language 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 a working station that has a visual display screen some three feet on a side; this is his working surface and is controlled by a computer (his 'clerk') with which he can communicate by means of a small keyboard and various other devices. …With a '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 right-hand third of the screen.

Engelbart describes what seems like a modern CAD system controlled through a graphical user interface. The vision is 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 Sketchpad project, but notes that further information was unavailable at the time of writing.

Engelbart's report eventually found its way to Bob Taylor who worked as a research program manager at NASA. Taylor had a 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 set-up the Augmentation Research Centre at the SRI and enabled 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

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:07:52, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.004

of a computer 'mouse'. English also 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.

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 the 1970s at Xerox PARC, 'they had to justify the fact that it took people weeks to memorise the keyset and months to become proficient, so they came up with this whole mystique about "augmenting intellect"'.[70]

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 Language for low-level coding and a Content Analysis Language for text manipulation. The languages themselves were implemented in another special language, Tree-Meta, designed for writing compilers. This enabled 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 video-conferencing and collaborative document editing. It showed a number of features that prefigured tools that appeared in integrated developer environments several decades later. When viewing NLS source code, the programmer was able to collapse structures such as IF statements and each file also contained a list of known bugs and collaboratively edited notes recording, for example, possible future improvements.

Douglas Engelbart established a group of devoted disciples 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'.[71] However, it took several more years before anyone tried to implement similar ideas outside of the tight-knit Engelbart group.

3.10 Almost Anything Goes

In the 1960s and 1970s, work on interactive programming was done at a small number of research centres and institutions. They were loosely connected through conferences and meetings, but also through the movement of individuals between the centres and later virtually through the ARPANET. The map of ARPANET from 1972 (Figure 3.8) shows many of the centres. It includes Harvard, Lincoln Lab, MIT and BBN in Boston where the first computer hackers experimented with interactive programming 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 Sketchpad fame was based and where Alan Kay, whom 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 organisation in the development of the community was the ARPA Information Processing Techniques Office (IPTO), initially called Command and Control Research office, of the US Department of Defense. The office was established by ARPA in order to kill two birds. First, they wanted to get into command and control research, but

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:07:52, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.004

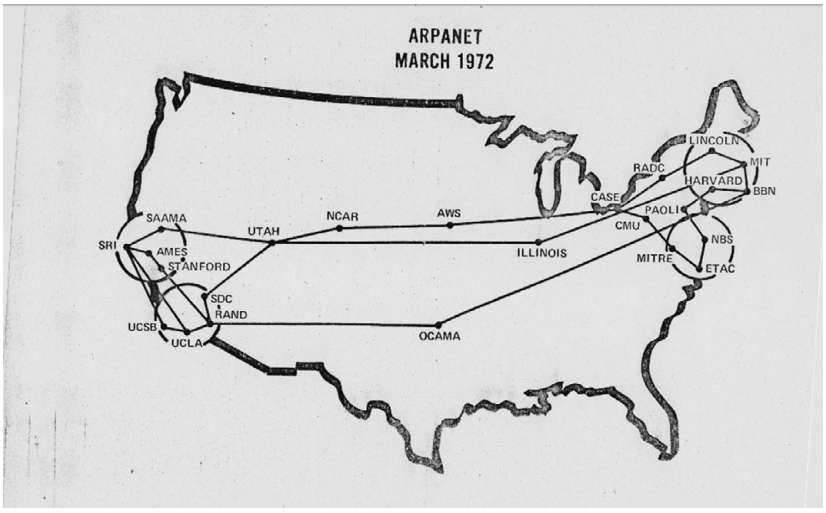


Figure 3.8 A map of the ARPANET from 1972, showing the first 29 connected nodes. (Source: UCLA and BBN, Wikimedia Commons,[72] CC-BY-SA 4.0)

second, they also had a rather expensive computer, built as a backup for the SAGE air defence system, for which they no longer had any use. As one of the earliest projects, the expensive computer was sent to System Development Corporation (SDC) in Santa Monica and used for war game scenario simulations, but 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 provide a better understanding of organisational, informational, and man-machine relationships'.[73]

After establishing the IPTO, the ARPA managers approached J. C. R. Licklider to become its first director. Licklider was a lucky 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 related topics. 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 struck 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'.[74] Under Taylor's leadership, the IPTO also started organising regular meetings specifically for the students it funded, which provided a 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. By the start of the 1970s, the political context changed and

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:07:52, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.004

forced IPTO to shift focus from basic research to applied work. Fortunately for the ARPA community, another organisation opened in 1970 and provided a new home for many from the ARPA community. The new organisation was the famed Xerox Palo Alto Research Centre (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 tapped into the same mix of hacker, engineering and humanistic 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 the 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.[75] The detailed accounts reveal many links and sometimes unexpected connections. 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'[76] for *Rolling Stone* magazine, the title of which was a nod to the computer game that I mentioned at the beginning of this chapter. The article portrayed the Xerox PARC researchers and programmers, including Bob Taylor and Alan Kay, as brilliant, uninterested in conventional goals and plenty of time for messing around, a portrait that shocked the Xerox management and caused much tension 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 for intellectual reasons as his work on augmenting human intellect was closely paralleled with the idea of enhancing creativity with psychedelic drugs.[77] Engelbart's own experience with LSD did not lead to any 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 reportedly conceived by Bill Atkinson during an LSD trip in 1985.

The ARPA community and Xerox PARC are interesting as places 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 range of backgrounds and had past experience with different ways of thinking about computers. PARC hired L. 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 in producing 'The Mother of All Demos'. Further engineers included Chuck Thacker, who previously worked on hardware design for a time-sharing system at Berkeley. His engineering spirit is captured by his lifelong fight against

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:07:52, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.004

'biggerism'; in his computer designs, one 'never found a logic gate or a ground wire out of place'.[78]

Finally, PARC also had a fair share of members influenced by the humanistic culture. The best known is perhaps Alan Kay. Kay struggled with the rigid education system, but eventually completed a PhD at the University of 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 a *Rolling Stone* article by Stewart Brand. Kay saw computers as a 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'.[79] Kay's background clearly links 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 Papert'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 and engineering work, as well as research motivated 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. The focus still excluded some of the more serendipitous creative work done by artists. For example, Richard Shoup and Alvy Ray Smith, who briefly collaborated at PARC in 1974 and produced the 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 plenty of evidence illustrating the hypothesis that interesting new developments occur when multiple cultures meet. This was the case for the birth of programming languages 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 clearly had mathematical interests, as illustrated by the many mathematical puzzles discussed in the HAKMEM memo[80] which is a canonical output of the hacker culture. 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 organisational structures, careful advance planning and formalised processes. To paraphrase Paul Feyerabend's mantra of epistemological anarchism, when it comes to interactive programming *almost* anything goes. The community was

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:07:52, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.004

keen to embrace different methods and principles and often overcame the differences, but this did not apply to managerial methods. There are multiple possible reasons for the absence of managerial culture of programming in the mix. First, interactive programming requires direct and 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[81] illustrated by the 'Unlocking the 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 the ground-breaking technologies invented at Xerox PARC easier to commercialise.

3.11 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 Python. Alan Kay is now best known as the designer of the Smalltalk programming language and the pioneer of object-oriented programming, but his interest at Xerox PARC was always in 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* the computer 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 customise and modify the pre-existing programs and eventually, develop new programs on your own.[82] In other words, an iPad or other modern tablet may look much like the tablet computer sketched by Kay in Figure 3.1, 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 make this possible.

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.[83] The work resulted in the influential machine Xerox Alto shown in Figure 3.9. About 30 machines were built initially. They soon became popular across PARC, as well as outside, and over a thousand Altos were eventually built.

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:07:52, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.004

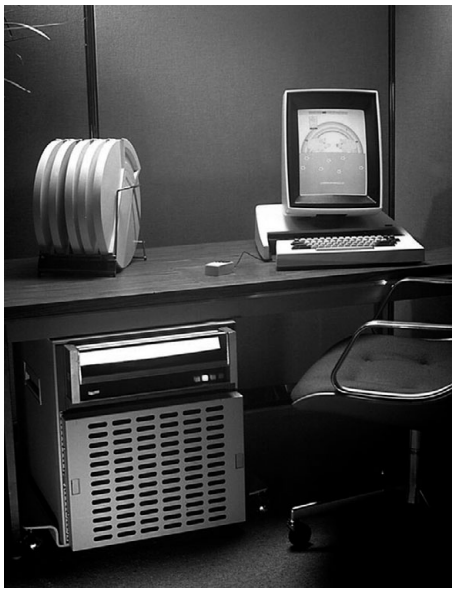


Figure 3.9 Alto with storage disks, computer, vertical display, mouse and a keyboard. (Source: Computer History Museum. Courtesy of PARC. Reproduced with permission of SRI International)

Equipped with a machine, Alan Kay and his team started working on implementing the Dynabook vision. The best account of the concept is the 'Personal Dynamic Media'[84] paper, co-authored with a PARC collaborator and co-developer of Smalltalk, Adele Goldberg. The paper uses a language influenced by media theorist Marshall McLuhan and describes the goal of the group as designing 'dynamic media which can be used by human beings of all ages'. 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 *meta-medium* that can be used to simulate, or *become*, 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' and an 'Electronic Circuit Design by a High School Student'. The expectation is that a user will first use the meta-medium 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 the 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 local middle school children and,

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:07:52, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.004

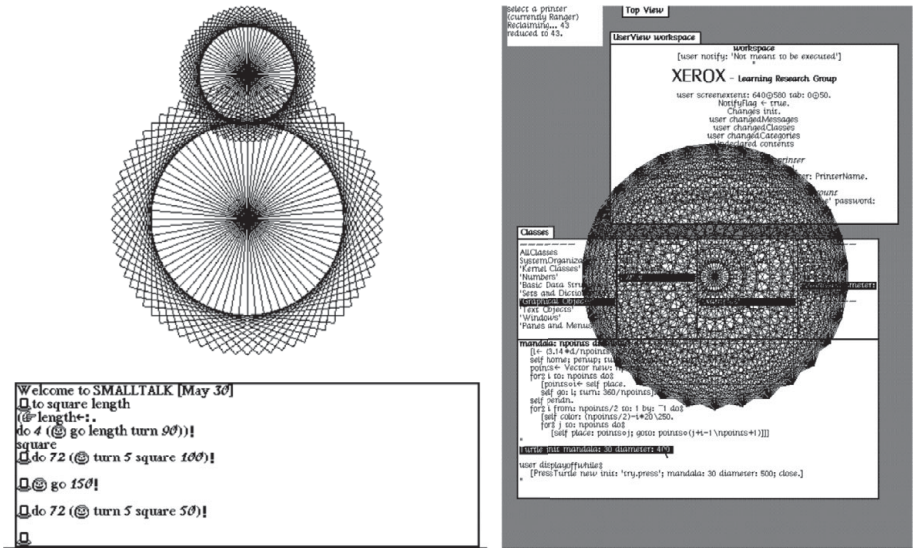


Figure 3.10 Turtle in Smalltalk-72 (left) and Smalltalk-76 (right). In Smalltalk-72, the screenshot shows a dialog window with entered code and the result of running a turtle code. In Smalltalk-76, the screenshot shows the 'mandala' operation in the class browser and the result of running an example included in the method definition.
(Source: Created by the author, using emulators from the Smalltalk Zoo[85])

despite some positive experiences, 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.[86]

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 sizable 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. He went on to create Smalltalk-76, which was a more efficient, fully supported object-oriented concept like inheritance and came with a graphical interface, shown in Figure 3.10 (right). The graphical interface also 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. At the same time, Alan Kay shifted his focus to a new, more accessible computer and a language called NoteTaker. Here, the Smalltalk language followed the same pattern as 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, but typically for their own purposes and experimentation that was somewhat different from the motivations of their creators.

Smalltalk may not have become 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

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:07:52, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.004

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 structure 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[87] 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 3.10 (left).

The development of Smalltalk happened in parallel to a major development rooted in other cultures of programming. 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 analysed. 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 oN-Line 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, 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'.

3.12 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

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:07:52, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.004

programming system that brought together the humanistic vision of programming as communication with graphical user interfaces. In this, it has been followed by a 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,[88] which appeared in 1975 and is a 'two-dimensional, visual programming system' implemented on top of Smalltalk-72. The second is Boxer[89] which appeared in 1985 and has been described as 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 means. In Smalltalk, graphical interface was used for structure 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 centred 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 a 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.[90]

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* language for communication with a computer, that is, a language that 'corresponds closely to the form used in the mind in thinking about the problem'.[91] Pygmalion, shown in Figure 3.11 (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 3.11, the icons are mostly labels in boxes, but the thesis also includes a more graphical example of electronic 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.

The Pygmalion programming system itself could have been described in a narrow technical way, but Smith's PhD thesis does the exact opposite. Its 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 humanistic culture. These include references to the Logo programming language, Turtle graphics, Sketchpad, but also the Spacewar! game developed by the MIT hackers for PDP-1 in 1962.Those

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:07:52, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.004

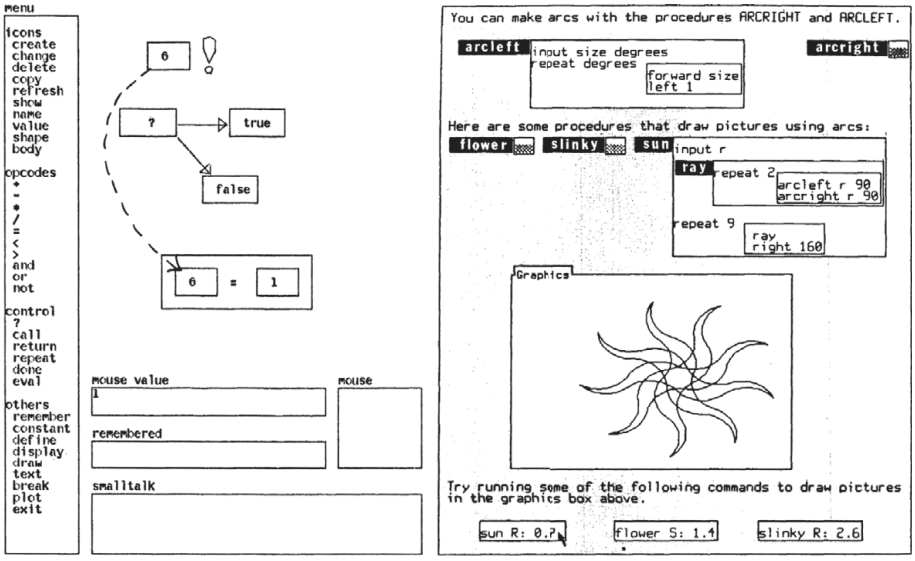


Figure 3.11 Pygmalion (left) and Boxer (right). The Pygmalion example shows a step in the process of computing the factorial function and the Boxer example shows a simple Turtle graphics program.
(Source: Left: Used with permission of Springer Nature BV, from Smith (1977). Right: Used with permission of ACM, from diSessa and Abelson (1986). Permissions conveyed through Copyright Clearance Center, Inc.)

references serve as *cultural pointers*[92] 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 interactive physics textbooks. A book on optics might include structured text, alongside simulations that perform ray tracing through lens and mirrors. The medium for such a 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 3.11 (right), implements a reconstructible computational medium using two key concepts. The first concept is the use of a *spatial metaphor*. Programs are represented as nested boxes in a two-dimensional space. 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.

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:07:52, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.004

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 visual representation is motivated by an argument about human cognition. People have a commonsense knowledge about space and the system can leverage that to help users 'interpret the organisation 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 Star, an office system that was an attempt to commercialise some of the technology developed at PARC. The design used icons in a way we would recognise today, but without any of their original computational nature.[93] 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 kinds of machines from those that Alan Kay and his group imagined.

3.13 Homebrew Computer Club

Since its establishment, Xerox PARC was subject to tensions between the researchers who pursued a curiosity-oriented approach and trusted scientific serendipity and the Xerox management that saw PARC largely as a way of preparing for potential future threats to their existing business model.[94] In 1971, PARC researchers reluctantly responded 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 for Pendery and Planning Purpose'.[95] The authors described many of their visions, some relevant for the 'office of the future' and estimated that their technology should be commercialisable by the early 1980s. Xerox eventually tried to do this. In 1977, it established the Systems Development Department, which started to build a system inspired by Xerox Alto that would be usable for office tasks. The resulting Xerox Star system was released in 1981 and was in many ways an impressive technology. It featured a 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

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:07:52, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.004

unable to find an effective marketing strategy for selling 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 computer appeared. Throughout the 1960s, hardware manufacturers started producing increasingly sophisticated integrated circuits. Finally, in 1971, Intel released a complete 4-bit 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 was soon followed by a more powerful family of 8-bit microprocessors, including the Intel 8080 that caught the eye of a small electronics company called MITS. The company was initially producing telemetry modules for rocket models, but it shifted its focus in the early 1970s and used the Intel 8080 microprocessor to create a cheap microcomputer that they called Altair 8800. Altair was a kind of computer that its creators wanted for themselves. Like many of the early adopters, they did not have any specific use for it. They just wanted to play with a computer of their own. This growing community of electronics hobbyists was served by a number of monthly hobbyist magazines. When the editors of one such magazine, *Popular Electronics*, started looking for a computer project based on the Intel 8080 microprocessor, they learned about the ongoing MITS project. Altair 8800 was featured in the January 1975 issue of *Popular Electronics* and the readers could directly order it for $397 (the equivalent of about $2,500 in 2024). Following the publication of the article, the machine became an instant hit.[96]

Altair 8800 was nothing like a modern personal computer. It was also nothing like the Xerox Alto that ran Smalltalk. It was much more like a tiny version of 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 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, which 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 could 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 enabled you to modify it. You could build or buy extensions for connecting 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 3.12) to manually enter the 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

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:07:52, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.004



Figure 3.12 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.
(Source: Living Computers Museum, Photo by Michael Dunn,[97] CC-BY 2.0)

creating a wide range of programs. The most iconic 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 emphasised individual technical achievements, information sharing and the Homebrew members believed 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*.[98] In contrast to the early MIT hackers, the Homebrew Computer Club was more open towards commercial pursuits.

The fact that for-profit 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'.[99] The community believed in decentralisation that is at odds with the managerial approaches to programming that were emerging in large-scale commercial computing enterprises. Attempts to transition

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:07:52, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.004

from the 'black art' of programming to the science of programming, which we looked at in Chapter 2, were also largely ignored by the microcomputer programmers.

Programming of microcomputers was also initially not influenced by for-profit 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 realised 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 behalf 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.[100] As argued by Joy Lisi Rankin, BASIC was already the 'language of the people'[101] and the dispute did not prevent it 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 broader 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 pre-assembled, 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 Homebrew member and an Altair kit owner. 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 favoured by many more sophisticated users and, later, also by companies producing advanced software. But BASIC was the first language that many new computer users would see and it shaped their experience with computers.

3.14 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 a 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 Dartmouth student should have access to

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:07:52, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.004

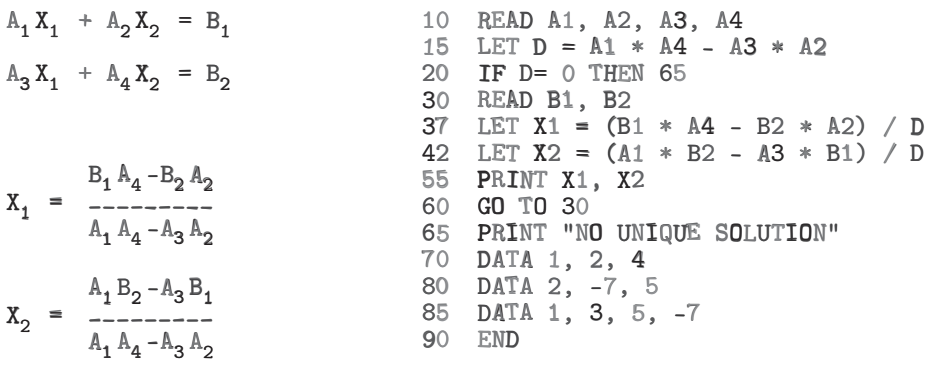


Figure 3.13 An introductory example from the 1964 BASIC manual. A program for solving a pair of linear equations (left top) implementing a calculation (left bottom) as a BASIC program (right).
(Source: Dartmouth College Computation Center[102])

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 3.13) shows many of the available constructs. Programmers could define variables, read data from the `DATA` block at the end of a program, and 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 subroutines 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` for reading data from and writing data to any memory address specified as an argument. This gave programmers access to everything on the machine that was not exposed in another way, such as drawing on arbitrary screen location. Since the 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 for invoking code at an arbitrary memory location, including various system routines and assembly code written by the programmer. Despite the simplicity of the core language, the system provided enough backdoors for hackers to practise their craft once they mastered all the basic tricks.

Perhaps more interesting than the language was 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 3.14) 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, you had to do this by invoking the BASIC command `LOAD`, which takes a file name and device number as arguments and

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:07:52, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.004



Figure 3.14 After turning on, Commodore PET starts and immediately runs Commodore BASIC. Any further interactions were through the BASIC command prompt. (Source: Courtesy of Terry Stewart.[103] Reproduced with permission)

then you had to type `RUN`. This loaded the BASIC source code for the program into the memory of the interpreter and ran it. You could then also 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 including FORTRAN and 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 use of line numbers, which was a remnant from the era of teletype terminals and was sometimes ridiculed, but it proved remarkably useful. As Figure 3.13 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. Their design was also influenced by Algol and they

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:07:52, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.004

referred to BASIC as an 'algebraic language', even though they did not adopt some of the Algol features that were to become influential in the mathematical culture of programming and became recognised 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.[104]

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 lets you easily experiment with computers, have fun and (occasionally) also write a program that does something interesting or even useful.

The era of microcomputers came about thanks to a 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 era of Apple II, Commodore PET and TRS-80 was also perhaps the last time programming was at the centre of using computers.[105] 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 ran other programs, but they were equally interested in fiddling with the machine to see what it could 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.

3.15 The Birth of an Industry

At the turn of the 1980s, microcomputers turned from playthings for hackers into commercial tools. This was not just because of the rise of more user-friendly microcomputers and their low cost, but crucially, also because of the development of new ways of using computers. The purchase of a 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

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:07:52, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.004

of microcomputers would not be so interesting if their purchase also involved the development of expensive custom software. However, new kinds of software, which 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.[106]

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. The capabilities of Bravo were only matched by graphical text processors for Apple Lisa and Apple Macintosh in the 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. It was initially sold for Apple II and it became the reason why many companies purchased the machine and sometimes even referred to it as the 'VisiCalc machine'. The system 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. As documented by Nooney,[107] 'Bricklin's willingness to develop for a microcomputer rather than a time-sharing system wasn't based in a passion for the hack or a desire to prove anything about the potential of microcomputing at all but in a deliberate set of business decisions largely governed by his most interested potential investor.' VisiCalc was a commercial product of an entrepreneurial spirit. It came with an 'extensive documentation, reference guides, model spreadsheets, and packaging that made users feel like they were dealing with a serious product'.[108]

VisiCalc was not intended as a programming system, but as 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 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.

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:07:52, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.004

The third must-have application for business use of microcomputers was 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 the dBase database were triggered through commands entered in a simple interactive prompt (much like that of 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 a 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 'Personal Dynamic Media' article. Their dream did not come true. VisiCalc and dBase were not created using meta-medium such as Smalltalk that would enable their users 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 shape the programming and use 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 humanistic cultures. For hackers, everyone interacting with a computer should be a hacker. The humanistic 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 visions 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 contain user interface elements and data. Like modern-day web pages, cards could be connected through links. HyperCard stacks were used and created through the same user interface. The interface displayed the current card and the user could interact with it by clicking on buttons and following links. The interface, however, supported multiple modes (Figure 3.15). 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 behaviour using the HyperTalk programming language. The progression from a user to a programmer,

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:07:52, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.004

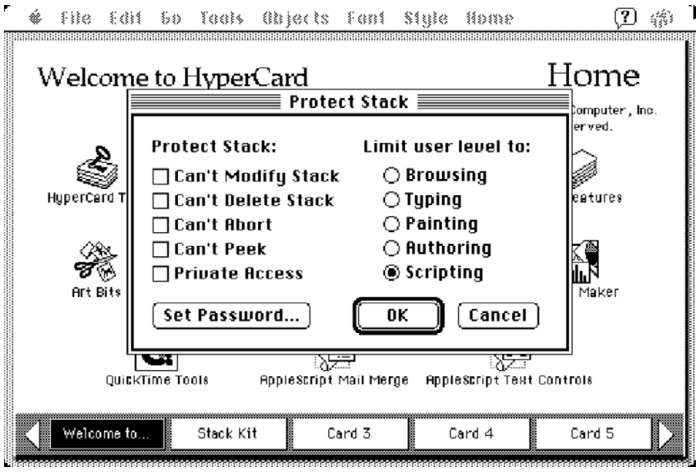


Figure 3.15 Options for protecting the stack and choice of user levels in HyperCard. (Source: Screenshot created by the author, using the Internet Archive HyperCard emulator[109])

which we saw implicit in many systems originating in the humanistic 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 3.15).

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 countercultural 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:[110]

> Poets, artists, musicians, physicists, chemists, biologists, mathematicians, 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

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:07:52, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.004

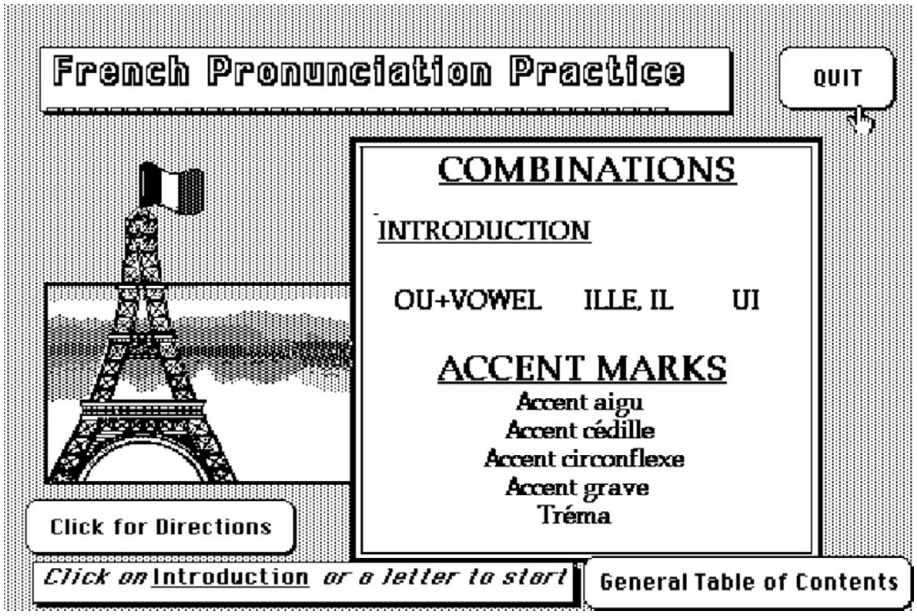


Figure 3.16 French Pronunciation stack created by Charles W. Gidney.
(Source: Screenshot created by the author, using the Internet Archive HyperCard emulator[111])

creative uses like interactive games, educational materials (Figure 3.16) and art pieces to commercial applications like databases and control systems. HyperCard gave users much more flexibility than VisiCalc or dBase and was perhaps closer to systems like Boxer. Yet, the user still had to operate within the constraints of the stacks of cards format.

3.16 Show Us Your Screens!

In the late 1970s, 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. The same technology also supported new creative uses of computers in the art scene. As was the case with the early computer art which influenced the educational Logo language, 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 made it possible for the electronic system to become 'a musical actor, as opposed to merely a tool'.[112] 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

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:07:52, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.004

23 buttons. It was intended as a development board for a new MOS 6502 microprocessor. It became popular with hobbyists, partly because its expansion connectors allowed it to connect to devices such as lights, switches and speakers. Several 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 the 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 3.17). As one of the group members recalls, 'this made for a system with rich and varied behaviour, but it was prone to failure, and bringing in other players was difficult'.[113] To 'clean up the mess', the group designed a new architecture where a central microcomputer, The Hub, was used for passing messages between the individual players connected to it. The architecture gave name to a reborn computer

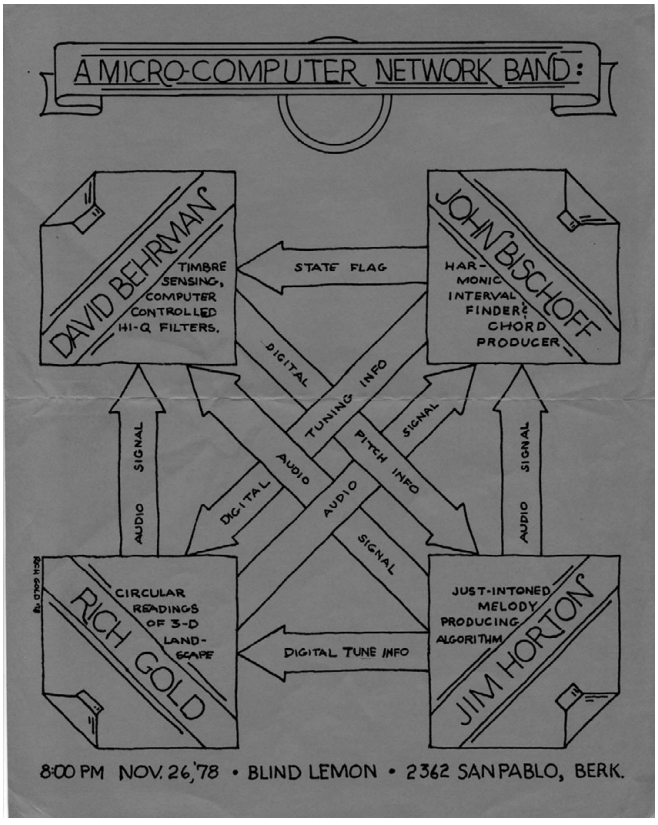


Figure 3.17 Flyer from the League's Blind Lemon concert of Nov. 26, 1978 featuring diagram of League topology. Designed and drawn by Rich Gold. (Source: Courtesy of Rich Gold. Reproduced with permission of Marina deBellagente LaPalma)

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:07:52, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.004

music group, The Hub.[114] 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 a logical next step to showing the bare KIM-1 circuit boards that were used in the early performances.

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 Computer Club. 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 supports immediate program execution.

At the turn of the millennium, 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.[115] 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|Transnational|Terrestrial|Transdimensional) Organisation for the (Promotion|Proliferation|Permanence|Purity) of Live (Algorithm|Audio|Art|Artistic) Programming. The TOPLAP manifesto[116] exhibits many of the aspects of the humanistic culture of programming. It puts code at the centre 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 organisation also emphasises 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, one of the ways in which live coding benefited 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.[117] 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 synthesiser to play notes and samples. They can select and 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 characterised Logo. First, Sonic Pi is designed for both schoolchildren and professional musicians, which mirrors the aim

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:07:52, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.004

of Logo to bring the 'most powerful programming language available', that is, Lisp, to everyone. Second, Sonic Pi and live coding more generally adopt a liberal approach to errors. The performers recognise that 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'.[118] As in Logo, errors are seen as sources of serendipity that provide an opportunity for learning.

Live coding is clearly inspired by earlier work on computer art and so it originates 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 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 coders are performers who use the computer as a musical instrument rather than to construct linguistic entities. A typical performance using a system like Sonic Pi involves programming concepts like loops and processes, but does not typically involve the programming of any algorithms.

In this, programming education using live coding dissents from systems that teach concepts based on the academically dominant mathematical culture. The most prominent example of the latter is the visual programming language for education Scratch, which was developed at MIT Media Lab in the early 1990s. Scratch was inspired by Logo and evolved from the idea of 'visual Logo'. It lets children 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 rooted in the mathematical culture of programming. The difference between Sonic Pi and Scratch illustrates the importance of recognising different cultures of programming. The notion of computational literacy will differ significantly, depending on which culture of programming is advocating for it.

3.17 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 analysed 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 interactive programming systems like Smalltalk or Interlisp still have a language at their centre.[119] The two ways of

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:07:52, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.004

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 favoured by different cultures of programming. On the one hand, the mathematical culture builds around the idea of treating programs as entities written in a formal language. On the other hand, hacker culture and humanistic culture often rely on treating programming as interaction with a machine or a medium, respectively. To hackers, the interactive way of programming offers direct access to the computer and lets them fully utilise the machine in ingenious, unrestricted ways. To artists, visionaries and educators, the interactive way of programming emphasises the crucial ongoing process, be it exploration, learning or communication with a machine.

When discussing the mathematisation 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 with references that can be easily 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 development of interactive programming does not follow such a clear historical line. Work on interactive programming often results in code, hacks, memos and demos rather than publications with references. This complicates the tracing of direct influences. The progression from MIT hackers to the ARPA community and Xerox PARC is an exception from the rule, because 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 differ, 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 for a time as a primary medium for 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 the LISP editor, which made program modification on a teletype manageable. New contexts that inspire ideas on interactive programming can also be new application domains. Two examples are programming in the educational context and programming as a tool for computer music.

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:07:52, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.004

Interestingly, the interactive style of programming found its place even within the mathematical culture. This happened in the early 1990s in the context of interactive theorem proving. Here, the programmer first defines a theorem they want to prove and then enters 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.

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 the 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.

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–machine symbiosis, augmenting human intellect or programmable meta-media. 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.

This chapter and Chapter 2 have 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 Chapter 4, 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.

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:07:52, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.004

2 Programming of Whirlwind was documented by Fedorkow (2021), as part of an effort to recover some of the original Whirlwind software artifacts.
3 For the connections with counterculture, see Markoff (2005) and Turner (2010). The story of the ARPA Information Processing Techniques Office (IPTO) community, which has gained almost mythical status among some programmers, has been told by Waldrop (2001) and Hiltzik et al. (1999).
4 The notion, known as *advising* was developed by Teitelman (1966) in the PILOT system.
5 A phrase used as the title of a speculative essay by Kay (1972b); the motivation has also been described by Kay (1996) in a retrospective article on the history of Smalltalk.
6 The example is based on that given in Kay and Goldberg (1977).
7 Kay (1972b).
8 https://github.com/livingcomputermuseum/Darkstar.
9 Charles 'Chuck' P. Thacker who worked at Xerox at the time used the term 'biggerism' to describe unnecessarily complicated technology (Hiltzik et al., 1999).
10 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`.
11 Halvorson (2020) tells the story of microcomputers, BASIC and how those contributed to the movements to democratise programming.
12 Dijkstra (1982).
13 Also known as REPL (Read-Eval-Print loop), named because the shell repeatedly reads code as input, evaluates it and prints the result.
14 For this view on interactive programming, see Blackwell and Collins (2005).
15 Kotok (2005).
16 www.flickr.com/photos/24226200@N00/364960084/.
17 As recollected in oral histories by Greenblatt (2005); Russell (2017); Samson (2017); Kotok (2005).
18 Campbell-Kelly (2004) discusses the broader context of home and recreational software that followed early games like Spacewar!.
19 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, besandalled computer programmer' at the time when 'in actual practice women were still very much present in most corporate computer departments'.
20 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.
21 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 USA and went on to double the number of programmers in the country over the next few years.
22 McKenzie (1974).
23 Gilmore (1958a,b).
24 The 'dynamic flow chart program' is described in 'The Computer Museum Report', Volume 8, 1984 available at www.bitsavers.org/pdf/mit/tx-0/TX-0_history_1984.txt, retrieved May 21, 2022.
25 Samson (2017).
26 The topic of code aesthetic from the perspective of different groups that partly overlap with the cultures of programming is discussed by Depaz (2023).

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:07:52, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.004

27 Beeler et al. (1972).
28 The technical details of Logic Theorist have been described by Gugerty (2006). **?** 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.
29 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.
30 Levy (2010).
31 Tan (2020) considers the 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.
32 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.
33 Bush (1945).
34 Licklider (1960).
35 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 sceptical of automatic anti-ballistic missile systems that excluded human interaction.
36 Licklider (1965).
37 Sutherland (1966).
38 Sutherland (1966).
39 GRAIL was described by Ellis et al. (1969). Visual programming languages continues to be an active research field today and is 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).
40 November (2004).
41 McCarthy (1992).
42 Quoted in Waldrop (2001).
43 Quoted in Waldrop (2001).
44 Documented in Waldrop (2001).
45 Fox (1960); McCarthy (1978).
46 Deutsch (1967).
47 The code is based on an example given by Deutsch (1967), but is reformatted, corrected and adapted for modern Lisp systems.
48 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.
49 Teitelman (1966).
50 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.
51 Documented in the Interlisp manual (Teitelman, 1974) and in a paper on the history of Lisp (Steele and Gabriel, 1996b).
52 Bawden et al. (1974).
53 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 (Raymond, 1996).
54 For the first-hand account of the history, see the work by Steele and Gabriel (1996b).
55 A book by Schank and Riesbeck (1981) illustrates this approach by collecting 'miniature' versions of AI programs in order to explain them. For a reflection on this way of thinking

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:07:52, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.004

about computer programs, see also the work by Sack (2022). I am grateful to Richard P. Gabriel for suggesting this interpretation of the history of Scheme.

56 According to Steele and Gabriel (1996b).
57 Noll (2016).
58 Nake (1971).
59 This interpretation of the interests of the two researchers is due to Solomon et al. (2020).
60 Solomon et al. (2020).
61 Papert (1980).
62 Papert (1980).
63 This example is given by Solomon et al. (2020).
64 https://github.com/drj11/curlylogo.
65 Solomon et al. (2020).
66 Many of those are documented, for example, by Ensmenger (2012); Turner (2010); Abbate (2012); Hicks (2017).
67 Solomon et al. (2020).
68 Parts of the story of Doug Engelbart have been told by many, including Rheingold (2000); Hiltzik et al. (1999); Waldrop (2001) and perhaps most comprehensively by Bardini (2000).
69 Engelbart (1962).
70 Hiltzik et al. (1999).
71 Quoted in Hiltzik et al. (1999).
72 https://commons.wikimedia.org/wiki/File:Arpanet_1972_Map.png.
73 Barber (1975).
74 Waldrop (2001).
75 For the former, see Waldrop (2001); Hiltzik et al. (1999); for the latter, see Markoff (2005); Turner (2010).
76 Brand (1972).
77 As documented by Markoff (2005).
78 Hiltzik et al. (1999).
79 Kay and Goldberg (1977).
80 Beeler et al. (1972).
81 Documented by Ensmenger (2012).
82 This envisioned progression was made more explicit later (Reenskaug, 1981).
83 Some of the motivations are discussed in the internal 'Why Alto' note (Lampson, 1972).
84 Kay and Goldberg (1977).
85 https://smalltalkzoo.thechm.org/.
86 Hiltzik et al. (1999); Kay (1996) himself also acknowledges that the children selected from Palo Alto schools were of 'hardly an average background'.
87 Goldberg and Kay (1976a).
88 Smith (1977).
89 Di Sessa (1985); diSessa and Abelson (1986).
90 Gombrich (1961).
91 Smith (1977).
92 A term introduced by Lennon (2019).
93 The design has been documented by Johnson et al. (1989).
94 According to an interpretation by Hiltzik et al. (1999).
95 Damouth et al. (1971).
96 This history is a part of broader context of the development of a personal computer (Ceruzzi, 2003).
97 www.flickr.com/photos/acidhelm/39802981903.

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:07:52, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.004

98 Interactive programming is a technological link between Engelbart's NLS, Kay's Smalltalk and microcomputers. Their role in the countercultural movement provides another link (Turner, 2010).
99 From 1975 issue of the People's Computer Company, quoted by Turner (2010).
100 The development of BASIC and the now-legendary letter are discussed by Freiberger and Swaine (1984); Ceruzzi (2003); Montfort et al. (2014).
101 Rankin (2018).
102 www.bitsavers.org/pdf/dartmouth/BASIC_Oct64.pdf.
103 www.classic-computers.org.nz/blog/2011-07-04-pet-rom-upgrades.htm.
104 Known as EWD498 (Dijkstra, 1982).
105 Possibly with the exception of the era of the 1990s web programming when many web users were also experimenting with creating their own websites and they were able to learn from others through the 'view source' functionality of their web browsers.
106 For a more detailed account of the birth of the personal computing industry, see the excellent account by Nooney (2023) and also by Ceruzzi (2003).
107 Nooney (2023); the history of VisiCalc has also been told by Grad (2007).
108 Nooney (2023).
109 https://archive.org/details/hypercardstacks.
110 Atkinson (2016).
111 https://archive.org/details/hypercard_french_pronunciation.
112 Chris Brown and John Bischoff (2002).
113 Chris Brown and John Bischoff (2002).
114 The history of The Hub has been documented by Gresham-Lancaster (1998) and Brümmer (2021).
115 The parallel has been pointed out by Zmölnig and Eckel (2007).
116 Ward et al. (2004).
117 Aaron (2016).
118 Blackwell and Collins (2005).
119 For a more detailed analysis of the difference between the programming language perspective and the programming systems perspective, see also Jakubovic et al. (2023); Gabriel (2012).

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:07:52, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.004