A Brief History of Compilers

Asher Mancinelli

October 9, 2025

Contents

Preface 4

1 Introduction 5

2 Dawn, 1940-1960 7
- 2.1 Where Does it Start? 7
- 2.2 Development of the Z4 7
- 2.3 The ETH's Aquisition of the Z4 8
- 2.4 Grace Hopper 9
- 2.5 Hopper and the Mark I's Manual 10
- 2.6 Postwar Collaboration 11
- 2.7 Hopper at EMCC 12
- 2.8 After the A-2 13
- 2.9 Laning and Zierler at MIT 14
- 2.10 John Backus 14
- 2.11 IBM Mathematical FORmula TRANslating System 15
- 2.12 Timeline 18

3 Crisis and Systems, 1960-1980ish 20
- 3.1 The Software Crisis 20
- 3.2 Aho Before Bell Labs 21
- 3.3 Aho, Ullman, and Bell Labs 21
- 3.4 Compiler-Compilers 23
- 3.5 The Dragon Book 24
- 3.6 Commoditization 25
- 3.7 Timeline 26

4 Freedom, 1980-2000 28
- 4.1 GNU 28
- 4.2 Adoption of Linux 28
- 4.3 Low-Level Virtual Machine 28
- 4.4 Timeline 30

5 Codesign 32
- 5.1 Accellerators 32
- 5.2 MLIR 32

Quotes **33**

5.3 *A Catalogue of Optimizing Transformations* 33
5.4 "Keynote Address" 33
5.5 *The First Computers: History and Architectures* 33
5.6 *Grace Hopper and the Invention of the Information Age* 34
5.7 *Dealers of Lightning: Xerox PARC and the Dawn of the Computer Age* 38
5.8 *A New History of Modern Computing* 39
5.9 "Konrad Zuse's Z4: architecture, programming, and modifications at the ETH Zurich" 39

Bibliography **41**

Preface

The history of compilers is rich and deeply connected to the broader history of computing, however, I believe that no comprehensive work has tied together the threads of this history with a specific focus on compilers. There are wonderful tellings of the very first compilers on Konrad Zuse's Z4 computer at the ETH Zurich and Grace Hopper's pioneering work on the A-series compilers for the UNIVAC I, and John Backus' work on the first commercial compiler for FORTRAN at IBM, but these are often isolated stories. When they are woven together, they are often not connected to the *next* developments in compiler technology at Bell Labs: Aho and Ullman's *Principles of Compiler Design*, the development of Lex and Yacc, the C programming language, Bjarne Stroustrup's first C++ compiler `cfront` (inspired by Alan Kay's vision of object-oriented programming). The subsequent decades of open-source compiler development, advances in optimization techniques, and the explosion of new programming languages and compilation paradigms (e.g. just-in-timem compilation) are then followed by the rise and necessity of hardware-software codesign and domain-specific languages seen most evidently in projects based on MLIR and LLVM. The threads between these points in history offer a deeper understanding of each individual piece and context that motivates modern compiler development.

This work intends to weave these threads together. Compiler technology has been handed down and built upon by generations of computer scientists and engineers; while prior works have primarily focused on individuals or companies or have covered computer history more generally, this book follows the individuals as they develop compiler technology, and follows it to the next generation, from the genesis of the first compilers to the present day.

My hope is that each chapter stands on its own, but that reading them in order will give a more complete picture. The reader ought to be able to read a particular chapter that suits their needs at the time. This book does not assume the reader is deeply familiar with compiler engineering or computer science. The book is structured as a chronological narrative of the history of compilers, intending to keep the focus on compiler technologies and the people behind them without focusing on any particular company or product.

1 Introduction

While the modern programmer may consider the term *compiler* to be a specific one, it is still often misunderstood. Moreover, in the beginning, the term was more ambiguous than it is today. Brian Kernighan described compilers in this seemingly general way[36]:

> *A compiler is a program that translates something written in one language into something semantically equivalent in another language. For example, compilers for high-level languages like C and Fortran might translate into assembly language for a particular kind of computer; some compilers translate from other languages such as Ratfor into Fortran.*

However, this is not sufficiently general. Two counter examples are TEX and METAFONT, which are compilers that transform text into PDF documents and renderable fonts, respectively. Transforming text into an executable program is not the same operation as transforming text into a document or font, yet both are considered compilation.

When students are first introduced to compilers, the first sort of program they are contrasted with is interpreters. This distinction is not necessarily meaningful. The conventional notion of an interpreter is a program that performs the actions specified by the source code as chunks of the code are consumed; perhaps this was the case with early BASIC interpreters and it may be a useful mental model when introducing interpreters to new programmers, but one is hard-pressed to find a modern interpreter that does not perform a compilation step. The most popular interpreters for today's most popular interpreted languages, Python and Javascript, both use relatively sophisticated compilation techniques. There are even Python *libraries* that perform just-in-time (JIT) compilation, targetting CPUs, GPUs, and other specialized hardware [8][47][56][63].

To capture the full spectrum of compiler technologies, the definition we will use in this book is intentionally broad:

> *A compiler analyzes, transforms, and produces code, based on source code.*

In the case of TEX and METAFONT, the source code may not entail a *program* in the conventional sense; TEX source code describes a document rather than a sequence of instructions to be performed. The code being produced by the compiler may be the encoded PDF format, for instance. So too are interpreters which produce code in some form during the interpretation process. The CPython interpreter produces bytecode before executing the program, meaning it is a compiler by our definition. Perhaps CPython it is a compiler the consumes Python code and produces CPython bytecode, which also happens to ship a CPython bytecode virtual machine which typically executes the bytecode as soon as it is produced.

While teaching a course on compilers at Columbia University, one of Alfred Aho's students wrote a compiler called Up♭eat which produces music based on input data; given input data in some format, Up♭eat produces output code in the form of music [2]. The student's final presentation was to set the input data to the ticker for some symbol from the New York Stock Exchange, playing happy and upbeat music whenever the ticker went up, and sad depressed music when it went down. Another Bell Labs creation was the connection of two programs: a program that translated numbers into words (e.g. 123 to ”one hundred twenty three”), and a program that turned text into data representing sound waves. Connecting these two compilers (on that compiled text with numbers into text with words sounding out the numbers, and one that compiled text into sound waves) allowed Bell Labs employees to send messages that would be turned into sound waves and broadcast on a speaker in the computer room. The point of this section is not pedantry, but to establish a broad definition of compiler technology before embarking on the details of its development.

We will primarily focus on compilers that have run on a machine. We largely exclude theoretical works like Ada Lovelace's notes on claculating Bournoulli numbers and the development of automata theory, for instance. Each step in the development of compiler technology builds on the previous steps, and while the prior steps are important, our focus is on compiler programs and not the theoretical works that preceded them, except where espeically relevant.

2
Dawn, 1940-1960

2.1 Where Does it Start?

There is significant debate about who created the first compiler, in no small part due to ambiguous nomenclature. The term *software* came into use sometime between 1959 and 1962, as Rojas and Hashagen note:

> [E]xpressions such as ”hardware”, ”software”, ”machine language”, ”compiler”, ”architecture” and the like... were unknown in 1950. They only arrived a decade later, but the underlying concepts were quite familiar to us. [58]

This Honeywell advertisement *A Few Quick Facts on Software* sought to clarify these terms as well:

> Software is a new and important addition to the jargon of computer users and builders. It refers to the automatic programming aids that simplify the task of telling the computer ’hardware’ how to do its job....

At this time, hardware was the only piece that mattered to customers. Software was an afterthought, if a thought at all. The instruction set of the machine was important, because that was the user’s interface to the machine. It should come as no surprise, then, that the origins of our modern understanding of the term *compiler* are similarly murky, especially considering the fact that *compiler* already carried meaning in English, and was repurposed for computing.

Once could argue that any of these efforts consituted the first compiler:

- Konrad Zuse’s run-programs for the Z4 at the ETH Zurich in 1950
- Laning and Zierler’s compiler for the Whirlwind at MIT in 1950
- Grace Hopper’s A-0 and A-1 compilers for the UNIVAC I at Remington Rand in 1951

2.2 Development of the Z4

Konrad Zuse, a German civil engineer, began work on the Z4 during World War II. Funded partially by his family and partially by the Nazi government, his prior works demonstrated significant creativity and ingenuity, and they were leveraged to build precursors to modern cruise missiles and guided bombs. Most of Zuse’s machines prior to the Z4 were destroyed during the war.

[TODO: Zuse’s Z4 was a strange machine with bespoke memory and instruction set. This affected how the compilers for it were designed.]

2.3 The ETH's Aquisition of the Z4

There were several early efforts to create programs that produced punch cards which contained machine code instructions, which could then be fed back into the machine as input punchcards. The programs produced by these early compilers were called *run-programs*, and the process of using them was called *automatic programming*, a term later coined by Grace Hopper. The first of these programs was run on a machine called the Z4, designed by Konrad Zuse in Germany.

Professor Eduard Stiefel, shortly after establishing the Institute of Applied Mathematics to study numerical analysis at the Swiss Federal Institute of Technology (ETH) in Zurich, began searching for a computer for the institute. He learned of the computing advancements in the United States, Great Britain, and Germany, but no machines were readily available at the time. He sent his assistants Heinz Rutishauser and Ambros P. Speiser to the US to study the latest developments in computing; they spent most of 1949 with Howard Aiken at Harvard and John von Neumann at Princeton.

> Before we returned, that is, in the middle of 1949, Stiefel was informed about the existence of Konrad Zuse's Z4. At that time Zuse was living in Hopferau, a German village near the Swiss border. Stiefel was told that the machine might be for sale. He visited Zuse, inspected the device, and reviewed the specifications. Despite the fact that the Z4 was only barely operational, he decided that the idea of transferring it to Zurich should by all means be considered. Stiefel wrote a letter to Rutishauser and me (we were at Harvard at the time), describing the situation and asking us to get Aiken's opinion. Aiken's reply was very critical - the future belonged to electronics and, rather than spending time on a relay calculator, we should now concentrate our efforts on building a computer of our own.[61]

The Z4 was a bespoke machine with unique components; the computational logic was wired together with telephone relays and the memory was entirely mechanical.

> The Z4 could be used as a kind of manually triggered calculator: the operator could enter decimal numbers through the decimal keyboard, these were transformed into the floatingpoint representation of the Z4, and were loaded to the CPU registers, first to OR-I, then to ORII. Then, it was possible to start an operation using the "operations keyboard" (an addition, for example). The result was held in OR-I and the user could continue loading numbers and computing. The result in OR-I could be made visible in decimal notation by transferring it to a decimal lamp array (at the push of a button). It could also be printed using an electric typewriter. [57]

It notably featured instructions for conditional branching and subroutine calling, which both proved essential for the compiler development that would follow at the ETH. Stiefel was undeterred by Aiken's criticism, and convinced the ETH to purchase the Z4. In 1950, Heinz Rutishauser at Switzerland's ETH obtains a Z4.

> We also made some hardware changes. Rutishauser, who was exceptionally creative, devised a way of letting the Z4 run as a compiler, a mode of operation which

> Zuse had never intended. For this purpose, the necessary instructions were interpreted as numbers and stored in the memory. Then, a compiler program calculated the program and punched it out on a tape. All this required certain hardware changes. Rutishauser compiled a program with as many as 4000 instructions. Zuse was quite impressed when we showed him this achievement.[61]

Thus were the first run-programs produced. This is what we will consider **the first compiler**, though it was not called that at the time. Shortly after Stiefel's assistants' stints in the US and correspondence with Aiken, one of Aiken's engineers would find considerably more success exploring related ideas.

2.4 Grace Hopper

While Grace Hopper may not have been the first to create a program that punched out another program as its output, she pioneered the field of compilation to the extent that many consider her the inventor of compilers. Her innovations were also more readily adopted than those at the ETH. Consider Figure 2.12, and the pace of development in Hopper's time compared to the years prior. Note that while we have ample data on *how* Hopper's compiler worked and how she and her team developed it, the intuition behind those developments is foggy at best. We have recollections from Hopper and her contemporaries, but only from long after the fact. It was not understood at the time how important her work was, so we have only to speculate and piece together oral histories.

Originally a mathematics professor at Vassar College, Hopper obtained waivers for her age and weight and joined the U.S. Navy in 1943, eventually graduating first in her class from Midshipmen's School. She was assigned, somewhat unexpectedly, to Commander Howard Aiken's Harvard Computation Laboratory in 1944 as the third programmer of the Automatic Sequence Controlled Calculator (Mark I), the world's first operational computer. Although it was significantly slower than the ENIAC, it was *programmable*; the ENIAC had to be physically rewired to change its program. To write a compiler for the ENIAC, one would need to plug the phone lines in the back of the machine together to create the compiler, feed in the input program as data, and a human operator would have to take the punchcards it produced and manually rewire the machine to run that program.

Aiken built the Mark I in collaboration with IBM, though it is unclear how much either side contributed in its development. The proportion of credit given to either party would be disputed in numerous documents and press releases in the following years, including the *Manual of Operation for the Automatic Sequence Controlled Calculator*, the technical manual for the Mark I. In the fall of 1944, Aiken decided that his team needed to produce a book documenting the technical developments at the Harvard Computation Laboratory and how the Mark I was intended to be used. For this task, he chose Hopper; though she protested, her reputation as a clear and thoughtful communicator and writer had already earned her the job. Hopper was known for her writing ability, and it was a point of emphasis in her teaching career. She would assign her mathematics students onerous writing tasks to emphasize that "it was no use trying to learn math unless they could communicate with other people." [19, interview on 5 July, 1972]

Aiken and Hopper both understood that, for the Mark I to be a success, it would be used and understood by a large and diverse audience, and for that to happen, they needed a detailed, compelling, and accessible manual.

2.5 Hopper and the Mark I's Manual

Here we drift into some general computing history, mostly because it was so formative for Hopper, who was in turn so formative in the development of compilers. Her manual for the Mark I began with a detailed and dramatic retelling of computing history, opening with the following quote from Charles Babbage and culminating with the Mark I [54]:

> If, unwarned by my example, any man shall undertake and shall succeed in really constructing an engine embodying in itself the whole of the executive department of mathematical analysis upon different principles or by simpler mechanical means, I have no fear of leaving my reputation in his charge, for he alone will be fully able to appreciate the nature of my efforts and the value of their results.

Her history went on to cover:

- Blaise Pascal's counting machines, ”foundation on which nearly all mechanical calculating machines since have been constructed.”
- Leibniz; stepped wheels system for mul/divs.
- Charles Babbage; most significant part of the manual dedicated to him. Difference engine, idea for computing machine. Invented punchcard system to feed in information, made after textile looms. G H emphasized the machine would take 2 decks of cards, one for data, one for instructions (not von neumann).
- Ada King, Countess of Lovelace; series of essays on Babbage's machine. described possibly the first computer program. This could never run and would have to wait for the Mark I before the dream could come true.
- Aiken's Mark I

At one point, all new hires into Aiken's lab were required to read Charles Babbage's autobiography. Hopper was first exposed to Ada King in this text: ”she wrote the first loop. I will never forget; none of us ever will.” [TODO: Coworkers cast Aiken as Babbage, Hopper as Ada King.]

[TODO: Commander Edmund Berkeley, 1946, detailed report about how bad the conditions were in the lab.]

> In his report, Berkeley systematically detailed the unfavorable conditions at the Computation Laboratory, including the length of the work day and the isolation of the staff from similar projects at MIT and the University of Pennsylvania. He named eleven talented people who had left or been dismissed by Aiken between August 1945 and May 1946, noting that all were ”very bitter over the conditions on the project.” The root of the problem, according to Berkeley, was that ”in the Computation Laboratory there is no provision for appealing any decision or ruling whatsoever made

> by the project manager." He was amazed that no one at Harvard and no one in the Navy seemed to have jurisdiction over the rogue director, so that Aiken was able to rule with near absolute authority.

Aiken ran a rigidly heirarchical and meritocratic lab, which allowed Hopper to produce quality work and placed her on more equal footing with her male coworkers; Aiken openly disliked being assigned a female officer, but Hopper's competence outweighed any such sentiments. Her competence did not, however, shield her from the pressures of the environment. She leveraged her computing knowledge to assist the war effort, which included the bombing of Nagasaki. The stresses of the war effort and Aiken's overbearing management drove her to substance abuse in this time period.

2.6 Postwar Collaboration

Aiken started a symposium on large scale digital computing machinery in 1947 to foster collaboration that was not possible during the war.

[TODO: 1947 Aiken grew in stature. Could relocate military staff to Harvard, which he did with Hopper and two others. 1944-1947 Presper Eckert, John Mauchly, and female operators proved valuable bc of how well they could use computers. this generated a wave of new projects: BINAC+UNIVAC at ECC, Mark II and Mark III at Harvard, SSEC at IBM, Whirlwind at MIT, MANIAC at Inst for Advanced Study, ERA (engineering research associates). everyone was isolated during the war, now aiken wanted to start sharing ideas. Symposium on Large Scale Digital Calculating Machinery, June 1947.]

> We'd all been isolated during the war, you see, classified contracts and everything under the sun. It was time to get together and exchange information on the state of the art, so that we could all go on from there.

[TODO: connect her discontent at harvard to her leaving.]

After a brief stint of unemployment, Hopper joined a startup called the Eckert-Mauchly Computer Corporation (EMCC) where she found a more congenial environment to continue her work on compilers.

> According to her friend and former Harvard colleague Edmund Berkeley, Hopper turned to alcohol during this period as a way to deal with the compounding pressures at the Harvard Computation Laboratory. She had dedicated herself fully to the overwhelming task of bringing Howard Aikens machines to life. She used the machines to solve critical military problems, including one that resulted in an explosion over Nagasaki. As the psychological strains became increasingly pronounced, alcohol seemed to serve as an effective outlet, freeing Hopper to express emotions and to temporarily forget obstacles real and imagined. According to Berkeley, the expiration of Hoppers Harvard research contract was the best thing that could have happened to her, although in the short term unemployment added to the stress. During the last week of May 1949, the 43-year-old programmer packed up her belongings, headed to Philadelphia, and bet her future on two younger men who believed they could create the first commercial computer company.[19]

Hopper's contract at Harvard expired in May 1949, and seeking a new path, she moved to Philadelphia to join the Eckert-Mauchly Computer Corporation (EMCC) as a senior mathematician

2.7 Hopper at EMCC

The company Hopper joined was one of the earliest pure computer-focused ventures, founded by J. Presper Eckert Jr. and John Mauchly (designers of the ENIAC, or the Electronic Numerical Integrator and Computer). This startup environment contrasted sharply with the academic rigor of Harvard and the industrial scale of IBM. There she found an open-minded and welcoming environment to develop her ideas; Mauchly, who was to become Hopper's boss, was characterized as "very broadminded, very gentle, very alive, very interested, very forward looking,"[19] creating a tolerant, flexible company atmosphere in contrast to the pressure she experienced at Harvard. A majority of their programming staff consisted of mathematically inclined women who had served as ENIAC operators at the Moore School.

When Hopper arrived in 1949, EMCC had two major projects underway: the BINAC (Binary Automatic Computer), which was close to completion, and the UNIVAC I (Universal Automatic Computer), which would be running within a year. The work environment and upcoming UNIVAC project excited Hopper and enticed her to join the company after walking out of an interview with IBM.

While the organization was grounds for fruitful and innovative research and development team, EMCC was under financial strain; they depended on partial payments for UNIVAC-I orders to stay afloat. The unexpected death of EMCC's chairman Henry Straus forced Eckert and Mauchly to seek a buyer, which they found in Remington Rand, a typewriter and office equipment manufacturer.

Hopper and her team at Remington Rand developed three "compilers" in rapid succession, the A-0, A-1, and A-2, for the UNIVAC I. I quote "compilers" because the A-0 and A-1 were not compilers in the modern sense. Her work was grounded in intellectual openness, collaboration, and accessability; she pioneered the debuggability of programming languages, compiler error reporting, and new ways to share code and collaborate, for example.

> Hopper's recollections point to motivations ranging from an altruistic desire to allow "plain, ordinary people" to program to dealing with her own laziness. Naturally one must be skeptical of such claims, for they were made years after the fact. In 1951 it was difficult for even a visionary like Hopper to imagine the eventual ubiquity of computer technology, and one can be pretty confident that Hopper was not a lazy person.[19]

She began work on the A-0 in October 1951 in her spare time in order to address the mounting crisis facing Remington Rand: they were unable to fully support their customers, and their sales teams were, to put it kindly, incompetent with respect to their product, and they supported their customers as well as one might expect.

> Another Hopper programmer, Adele Mildred Koss, was assigned to Commonwealth Edison when the utility approached the Chicago sales office concerning a potential purchase of a UNIVAC for billing and payroll. At the time, Koss was 7 months

> pregnant and working part time. Since her pregnancy precluded travel, Commonwealth Edison management was forced to come to Philadelphia in order to discuss their billing needs. In the end, the utility did not buy a UNIVAC, but instead purchased an IBM 701 when it became available. Koss recalled: "I remember Grace Hopper's memo to management saying 'This is a multi-million dollar client and you are not treating them like one. You have only assigned a part time programmer to work with.'" [19, Adele Mildred Koss, interviewed by Kathy Kleiman]

Shortly after this fiasco, Hopper's team was tasked with supporting UNIVAC I customers at the US Census Bureau, a task she thought no one in the company was prepared for.

> Inspired by Holberton's Sort-Merge Generator, Hopper conceived the idea of writing a program to create a program, or in modern day terms, building a compiler. The idea was to get commonly used subroutines automatically inserted into another program based on calculated offsets. Most people at the time considered this impossible. As Hopper recalled:[35]

> The Establishment promptly told us, at least they told me, quite frequently that a computer could not write a program; it was totally impossible; that all computers could do was arithmetic, and that it couldn't write programs.[42]

[TODO: the A-2 was a front-end translator to the A-0, more like a complete toolchain with a front end to a language, a linker and a standard library.]

2.8 After the A-2

[TODO: later developed the A-3 and AT-3/MATH-MATIC; FLOW-MATIC (B-0, 1956/58), FORTRAN, COBOL]

> B-0, as the business language compiler was designated, became available to UNIVAC customers at the start of 1958. Before its completion, Remington Rand merged with Sperry Gyroscope Corporation to form Sperry Rand. The marketing department of the new company renamed the business language FLOW-MATIC. (In addition, AT-3 was renamed MATH-MATIC.) The completed version of FLOW-MATIC had a rich library of oper-ational verbs that appeared to meet the application needs of most businesses.23 These verbs included editing commands so information could be formatted before output. Furthermore, FLOW-MATIC provided unparalleled flexibility in data designation, thus allowing file names to be given complicated descriptions.

> her superiors that the proposed business language would only be in English. B-0, as the business language compiler was designated, became available to UNIVAC customers at the start of 1958. Before its completion, Remington Rand merged with Sperry Gyroscope Corporation to form Sperry Rand. The marketing department of the new company renamed the business language FLOW-MATIC. (In addition, AT-3 was renamed MATH-MATIC.) The completed version of FLOW-MATIC had a rich

library of oper-ational verbs that appeared to meet the application needs of most businesses.23 These verbs included editing commands so informa-tion could be formatted before output. Furthermore, FLOW-MATIC provided unparalleled fl exibility in data designation, thus allowing fi le names to be given complicated descriptions.

This was what she originally called the data processing compiler in January, 1955; it was soon to be known as "B-0," later as the "Procedure Translator" [KM 57], and finally as flow-matic [HO 58, TA 60], This language used English words, somewhat as math-matic did but more so, and its operations concentrated on business applications. The following examples are typical of flow-matic operations; [43]

2.9 Laning and Zierler at MIT

[TODO: Laning and Zierler, 1954. Early 1950s. Inspiration for Backus/FORTRAN. Worked more like a modern compiler than Hopper's A-0 and A-1.]

The first programming system to operate in the sense of a modern compiler was developed by J. H. Laning and N. Zierler for the Whirlwind computer at the Massachusetts Institute of Technology in the early 1950s. They described their system, which never had a name, in an elegant and terse manual entitled "A Program for Translation of Mathematical Equations for Whirlwind I," distributed by MIT to about one-hundred locations in January 1954.26 It was, in John Backus's words, "an elegant concept elegantly realized." Unlike the UNIVAC compilers, this system worked much as modern compilers work; that is, it took as its input commands entered by a user, and generated as output fresh and novel machine code, which not only executed those commands but also kept track of storage locations, handled repetitive loops, and did other housekeeping chores. Laning and Zierler's "Algebraic System" took commands typed in familiar algebraic form and translated them into machine codes that Whirlwind could execute.27 (There was still some ambiguity as to the terminology: while Laning and Zierler used the word "translate" in the title of their manual, in the Abstract they call it an "interpretive program.")28 [36]

2.10 John Backus

IBM did not feel that Aiken and the Harvard Computation Laboratory had given them sufficient crediti for their contributions to the Mark I, which left Thomas Watson Sr. and the IBM folks bitter about the experience and eager to produce a new device entirely in-house. This device would become the Selective Sequence Electronic Calculator, or the SSEC. It was built on 57th Street in Manhattan, and it was monstrous. Roughly 50 by 100 feet with a giant console and hundreds of toggle switches and tape units and relays behind glass panels; there were giant windows that allowed passersby to see the machine in action.

One such passerby was John Backus, a recent Masters graduate from Columbia University. He was intrigued by the machine, which he mentioned to his tour guide, who suggested he go upstairs and talk to the boss about it. Robert ”Rex” Seeber gave him a puzzle, which he solved, and he was hired on the spot [13].

In 1942, Backus majored in Chemistry at the University of Virginia where he struggled academically. He was expelled due to poor attendance within the first year before being drafted into the US Army. He commanded an antiaircraft battery at Fort Steward, Georgia, remaining in the US for the remainder of WWII.

While he did not at first find success in academia, he got very good marks on military aptitude tests. He was directed to the University of Pittsburgh’s engineering program and later to a premedical program at Haverford College near Philadelphia (which is where he grew up). In 1945 he attended the Flower and Fifth Avenue Medical School in NYC, but he was still struggling with the academy. He was uninterested in medicine, feeling that it was all about memorization. He dropped out after less than a year.

He entered a radio technician school and became interested in math, which led him to enroll in the math program at Columbia University. The SSEC that would intrigue him at the IBM computing center was designed at the Watson Scientific Computing Laboratory at Columbia.

2.11 IBM Mathematical FORmula TRANslating System

At IBM, Backus worked on the SSEC and later the IBM 701 and 704. The main use of the SSEC was aerospace calculations; programming calculations to predict the position of the moon was one of the first tasks he was given at IBM. He would continue writing programs for these machines in spite of their poor usability. His team’s techniques would be used in the lunar missions of the 1960s.

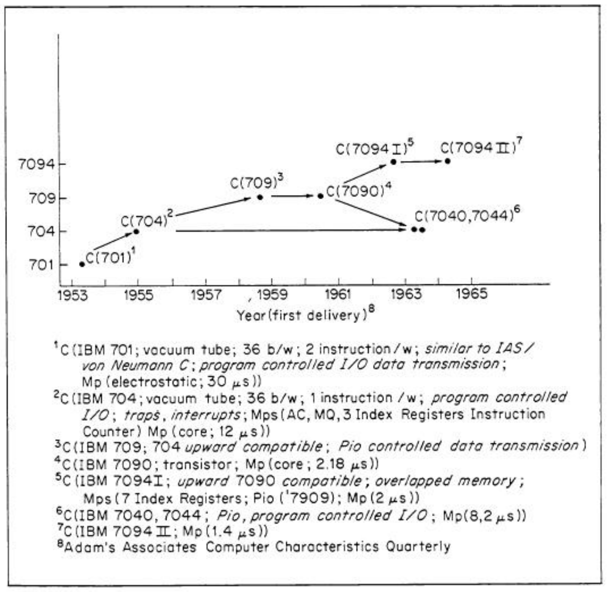
The pain of writing programs for these early machines entirely in machine code drove him to explore new ways to program. The first of these was a symbolic notation for floating point arithmetic and address expression calculation called Speedcoding[13]:

> **Grady Booch:** So then from your experience with the SSEC, you then went on to produce Speedcoding, the Speedcoder...What were sort of the things that influenced you to create that in the first place?
>
> **John Backus:** Well, programming in machine code was a pretty lousy business to engage in, trying to figure out how to do stuff. I mean, all that was available was a sort of a very crude assembly program. So I figured, well, let’s make it a little easier. I mean it was a rotten design, if I may say so, but it was better than coding in machine language.

The IBM 701 did not have an index register, so calculating addresses for array operations was tedious and error-prone. Speedcoding provided a way to express these calculations symbolically.

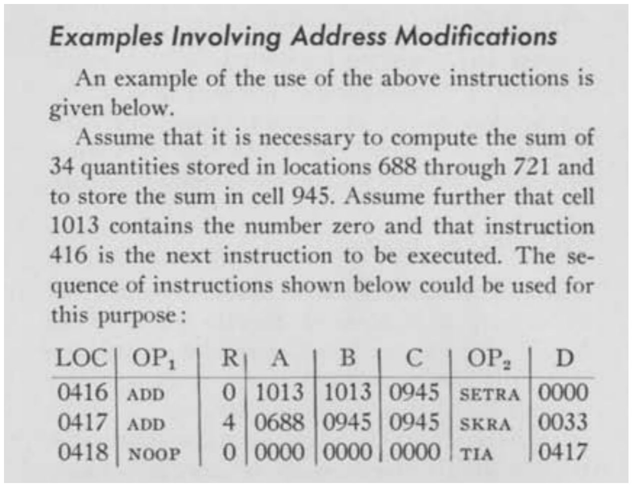
The 704 was the first machine to have such a register; it also had floating point instructions and core memory, more or less obviating the need for Speedcoding: ”we were moving to the 704, which had built in floating point, built in index registers, which was all that Speedcoding was supposed to supply. So what the hell?” [13] He credits himself with getting index registers and floating point into the 704.

In 1953, based on his experience with Speedcoding on the 701, Backus proposed yet another language to elevate the programming experience on the 704. IBM management supported the



[1]C (IBM 701; vacuum tube; 36 b/w; 2 instruction/w; *similar to IAS/ von Neumann C; program controlled I/O data transmission*; Mp (electrostatic; 30 μs))
[2]C (IBM 704; vacuum tube; 36 b/w; 1 instruction/w; *program controlled I/O; traps, interrupts*; Mps (AC, MQ, 3 Index Registers Instruction Counter) Mp (core; 12 μs))
[3]C (IBM 709; 704 *upward compatible; Pio controlled data transmission*)
[4]C (IBM 7090; transistor; Mp (core; 2.18 μs))
[5]C (IBM 7094I; *upward* 7090 *compatible; overlapped memory*; Mps (7 Index Registers; Pio ('7909); Mp (2 μs))
[6]C (IBM 7040, 7044; *Pio, program controlled I/O*; Mp(8,2 μs))
[7]C (IBM 7094 II; Mp (1.4 μs))
[8]Adam's Associates Computer Characteristics Quarterly

Figure 2.1: Excerpt from *The IBM 701–7094 II Sequence: A Family by Evolution* [38], illustrating the instruction structure for summing quantities.



Examples Involving Address Modifications

An example of the use of the above instructions is given below.

Assume that it is necessary to compute the sum of 34 quantities stored in locations 688 through 721 and to store the sum in cell 945. Assume further that cell 1013 contains the number zero and that instruction 416 is the next instruction to be executed. The sequence of instructions shown below could be used for this purpose:

LOC	OP_1	R	A	B	C	OP_2	D
0416	ADD	0	1013	1013	0945	SETRA	0000
0417	ADD	4	0688	0945	0945	SKRA	0033
0418	NOOP	0	0000	0000	0000	TIA	0417

Figure 2.2: Excerpt from *IBM Speedcoding for the Type 701 Electronic Data Processing Machine* [44]

proposal. He formed a ten-person team of his own choosing based out of IBM's Manhattan headquarters. They released *Preliminary Report, Specifications for the IBM Mathematical FORmula TRANslating System, FORTRAN* after about one year of working together.

[TODO: Backus wasn't really that involved in fortran II and III and IV. The rest of the team took over. He never wrote much fortran and doesn't have many thoughts about it today other that the "we should be making things higher level."] [TODO: Backus moved into functional programming. Was never fruitful. He became a lab fellow at IBM after Watson Jr started the fellowship program, so he could kinda play with functional programming and not really get much done.] [TODO: He didn't like Hopper's ideas for COBOL, thought they were crazy and too complex. Didn't like algol either. Really just wanted to make things simpler and higher level and easier to use. Pushed back against the priesthood.]

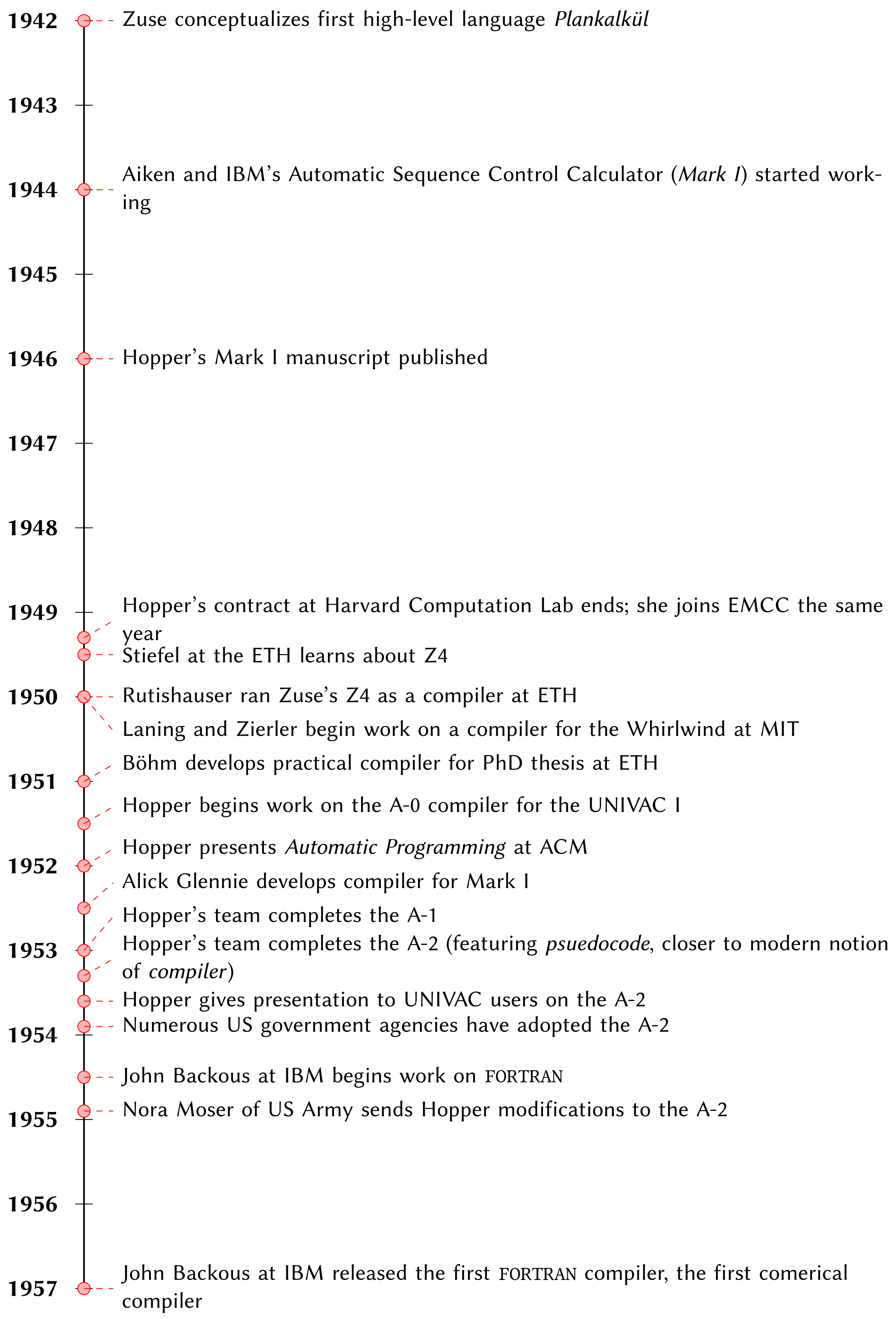
Though the FORTRAN operator's manual was completed by the fall of 1956, the compiler itself was not distributed to IBM 704 installations until April 1957. Within a year after distribution, half of the IBM 704 installations were using FORTRAN to solve more than half of all mathematical prob-lems.16 Subsequently, compilers were produced for the IBM 705 and the IBM 650, quickly making FORTRAN the most widely used automatic program of its day. By 1961, UNIVAC users demanded a compatible FORTRAN compiler and aban-doned Hopper's MATH-MATIC. [19]

It describes the system which will be made available during late 1956, and is intended to permit planning and fortran coding in advance of that time [p. 1], Object programs produced by fortran will be nearly as efficient as those written by good programmers [p. 2], "Late 1956" was, of course, a euphemism for April 1957. Here is how Saul Rosen described fortran's debut [RO 64, p. 4]: Like most of the early hardware and software systems, fortran was late in delivery, and didn't really work when it was delivered. [43]

It is not my intention to give a complete description of either; hence this section will describe only the main highlights of FORTRAN development. The earliest significant document that seems to exist 1s one marked "PRELIMINARY REPORT, Specifications for the IBM Mathematical FORmula TRANslating System, FORTRAN", dated November 10, 1954 and issued by the Programming Research Group, Applied Science Division, of IBM. [60]

Laning and Zierler's algebraic compiler served as evidence that prestigious institutions such as MIT were taking automatic programming seriously, prompting Backus to write Laning a letter shortly after the May symposium. In the letter, Backus informed Laning that his team at IBM was working on a similar compiler, but that they had not yet done any programming or even any detailed planning.12 To help formulate the specifi cations for their proposed language, Backus requested a demonstration of the algebraic compiler, which he and Ziller received in the summer of 1954. Much to their dismay, the two experienced fi rsthand the effi ciency dilemma of compiler-based language design. The MIT source code was commendable, but the compiler slowed down the Whirlwind computer by a factor of 10. Since computer time was so dear a commodity, Backus realized that only a com-piler that maximized effi ciency could hope to compete with human programmers. Despite this initial disappointment, Laning and Zierler's work inspired Backus to attempt to build a compiler that could translate a rich mathematical language into a suffi -ciently economical program at a relatively low cost.13 [19]

2.12 Timeline



1942 Zuse conceptualizes first high-level language *Plankalkül*

1943

1944 Aiken and IBM's Automatic Sequence Control Calculator (*Mark I*) started working

1945

1946 Hopper's Mark I manuscript published

1947

1948

1949

Hopper's contract at Harvard Computation Lab ends; she joins EMCC the same year

Stiefel at the ETH learns about Z4

1950 Rutishauser ran Zuse's Z4 as a compiler at ETH

Laning and Zierler begin work on a compiler for the Whirlwind at MIT

Böhm develops practical compiler for PhD thesis at ETH

1951

Hopper begins work on the A-0 compiler for the UNIVAC I

Hopper presents *Automatic Programming* at ACM

1952

Alick Glennie develops compiler for Mark I

Hopper's team completes the A-1

1953

Hopper's team completes the A-2 (featuring *psuedocode*, closer to modern notion of *compiler*)

Hopper gives presentation to UNIVAC users on the A-2

Numerous US government agencies have adopted the A-2

1954

John Backous at IBM begins work on FORTRAN

Nora Moser of US Army sends Hopper modifications to the A-2

1955

1956

1957 John Backous at IBM released the first FORTRAN compiler, the first comerical compiler

Figure 2.3: Early compiler history, 1942–1957

3 Crisis and Systems, 1960-1980ish

[TODO: shit got crazy; became important, needs more focus/attention.]

3.1 The Software Crisis

> Despite great strides in software, programming always seemed to be in a state of crisis and always seemed to play catch-up to the advances in hardware. This crisis came to a head in 1968, just as the integrated circuit and disk storage were making their impact on hardware systems. That year, the crisis was explicitly acknowledged in the academic and trade literature and was the subject of a NATO-sponsored conference that called further attention to it. Some of the solutions proposed were a new discipline of software engineering, more formal techniques of structured programming, and new programming languages that would replace the venerable but obsolete COBOL and FORTRAN. Although not made in response to this crisis, the decision by IBM to sell its software and services separately from its hardware probably did even more to address the problem. It led to a commercial software industry that needed to produce reliable software in order to survive. The crisis remained, however, and became a permanent aspect of computing. Software came of age in 1968; the following decades would see further changes and further adaptations to hardware advances. [27]

[TODO: DEC PDP-8 and PDP-11; IBM System/360 and OS/360; Multics; Unix; C]

As the 1960s progressed, the notion of *software* became more established, and the programs being written served the authors to a much greater extent. Prior to the 1960s, programs were often tailor-made for a specific machine. There was no hope of re-using the program on another machine. For most of the users, their organization had spent a considerable portion of their budget on their system, and they were expected to use it for a long time. Retargetable compilers did not exist.

Michael Mahoney made a strong statement to this end rather early on [58, The Structures of Computation]:

> The kinds of computers we have designed since 1945 and the kinds of programs we have written for them reflect not the nature of the computer but the purposes and aspirations of the groups of people who made those designs and wrote those programs, and the product of their work reflects not the history of the computer but the histories of those groups, even as the computer in many cases fundamentally redirected the course of those histories.

Despite great strides in software, programming always seemed to be in a state of crisis and always seemed to play catch-up to the advances in hardware. This crisis came to a head in 1968, just as the integrated circuit and disk storage were making their impact on hardware systems. That year, the crisis was explicitly acknowledged in the academic and trade literature and was the subject of a NATO-sponsored conference that called further attention to it. Some of the solutions proposed were a new discipline of software engineering, more formal techniques of structured programming, and new programming languages that would replace the venerable but obsolete COBOL and FORTRAN. Although not made in response to this crisis, the decision by IBM to sell its software and services separately from its hardware probably did even more to address the problem. It led to a commercial software industry that needed to produce reliable software in order to survive. The crisis remained, however, and became a permanent aspect of computing. Software came of age in 1968; the following decades would see further changes and further adaptations to hardware advances.

[TODO: 1945 was too early for this strong of a statement; how can one argue that software reflected the authors when it was so dependent on the hardware?]

3.2 Aho Before Bell Labs

3.3 Aho, Ullman, and Bell Labs

[TODO: Software (and compilers!) starts to become a real discipline! Ullman was older and further along than Aho, and Hopcraft came to Princeton and became Aho's advisor.]

One of the first people that I met at Princeton was a Columbia graduate by the name of Jeffrey Ullman. He had just gotten his undergraduate degree from Columbia University and also had come to study digital systems in the EE department at Princeton. So, he and I became close friends. When we graduated from Princeton, we both joined the newly formed Computing Sciences Research Center at Bell Labs. There we developed a lifelong collaboration on subjects ranging from algorithms, programming languages, to the very foundations of computer science. I was very fortunate to have met some of the greatest people in the field and to have gotten to know them and work with them. You learn so much by working with the best people in the field. So, I felt very blessed because I had this kind of background...

Hsu: Before we jump into Bell Labs more deeply, could you maybe explain– talk about your PhD thesis, but try to explain it to somebody who, maybe like a museum goer who doesn't really know much about computer science and linguistics.

Aho: This is interesting. As I mentioned, Hopcroft told me, "Find your own research problem." He did teach a course in automata and language theory, so I got introduced to formal language theory and automata theory, at least, as it was known at that time. I was interested in programming languages and compilers. What I noticed was that a programming language has a syntax and a semantics. All languages have a syntax and a semantics. If you want to write a translator for a programming language, or even a natural language, you have to understand the syntax and semantics of your source language and the target language...

Hansen: 1967, and you followed Ullman there. He had already joined Bell Labs before. Aho: A few months before me. Hansen: A few months before. And what group was it that you joined? Aho: I was interviewed by a department head by the name of Doug McIlroy. He was an applied mathematician from MIT. He had been at Bell Labs for a few years before me. Amongst other things, he had coinvented macros for programming languages and he's also in this class of one of the smartest people I've ever met.

Jeff wanted to go to academia a little bit earlier than I did, like many years earlier. He stayed at Bell Labs for a few years and went to Princeton University where he joined the faculty of the electrical engineering department, but he would come and spend one day a week consulting at Bell Labs. His consulting stint was he would come Fridays and sit in my office all day. The conversations that we'd have would range over all sorts of topics, and sometimes he'd mentioned that he was working on a problem with a colleague at Princeton, and after describing the problem, I might say, "You're kidding," and he said, "Oh, you're right. The solution is obvious, isn't it?" I don't know whether I would say dynamic programming or whatever, but several papers came out of this intense collaboration, and we got to the point where we could communicate with just a few words. We had a very large, shared symbol table. [2]

But as Unix was being developed, Ken Thompson created the first two versions of Unix using assembly language. He had joined Bell Labs at roughly the same time I had. He was there maybe six months or so ahead of us, and he had been assigned to work on the Multics project that Bell Labs was part of with MIT and GE. When Bell Labs got tired of pouring money into Multics and not getting the operating system that it had wanted, it abandoned the project and left Ken Thompson to his own devices. Ken thought there were some good ideas in Multics. Being the genius that he was, he said, I can do it much more simply and much more elegantly. So he created a rudimentary version of Unix and then kept writing and polishing it. Dennis Ritchie came on the scene. Ken had also created a programming language, B. The B was maybe the first letter of BCPL. Who knows? But when Dennis Ritchie looked at it, he said, what B needs is a decent type system. So he put a decent type system on B, and created the C programming language. Thompson and Ritchie wrote the third version of Unix using the newly created C programming language. I became an early adopter of C, and I had C wired in my fingertips, so I could write C programs quite readily, and of course, there were all these neat tools that accompanied the programming environment on Unix. There were the text editors. I don't know whether you've ever heard of the ED editor or the QED editor that was at MIT as part of Multics. QED had regular expressions in it. This triggered my interest in regular expressions. Ken Thompson had written a program called grep for doing pattern matching on text files, and it had a very limited form of regular expressions when I encountered it. [2]

Collaboration with Ullman Aho is best known for the textbooks he wrote with

Ullman, his co-awardee. The two were full time colleagues for three years at Bell Labs, but after going back to Princeton as a faculty member Ullman continued to work one day a week for Bell.

They retained an interest in the intersection of automata theory with formal language. In an early paper, Aho and Ullman showed how it was possible to make Knuth's LR(k) parsing algorithm work with simple grammars that technically did not meet the requirements of an LR(k) grammar. This technique was vital to the Unix software tools developed by Aho and his colleagues at Bell Labs. That was just one of many contributions Aho and Ullman made to formal language theory and to the invention of efficient algorithms for lexical analysis, syntax analysis, code generation, and code optimization. They developed efficient algorithms for data-flow analysis that exploited the structure of ”gotoless” programs, which were at the time just becoming the norm. [37]

The Early History of Software, 1952-1968 101

In the early 1960s computer science struggled to define itself and its purpose, in relation not only to established disciplines of electrical engineering and applied mathematics, but also in relation to—and as something distinct from—the use of computers on campus to do accounting, record keeping, and administrative work.58 Among those responsible for the discipline that emerged, Professor George Forsythe of Stanford's mathematics faculty was probably the most influential. With his prodding, a Division of Computer Science opened in the mathematics department in 1961; in 1965 Stanford established a sepa-rate department, one of the first in the country and still one of the most well-regarded.59 [27]

[TODO: Dragon book; all the books Aho, Ullman and others worked on together.]

3.4 Compiler-Compilers

[TODO: Yacc and Lex made with Aho's help. then everyone started making mini languages. AWK. ”Kernighan and Cherry developed a little language for specifying mathematics called EQN using these tools”]

People started using the Kernighan and Lorinda Cherry EQN tool to specify mathematics in their documents and in the research papers that they were writing. They would feed the EQN specification into the typesetting program roff...

Knuth adopted the EQN language to include in the TeX typesetting system, and in LaTeX. It's basically Kernighan and Cherry's way of specifying mathematics. These software tools had a great deal of influence, and Kernighan and Cherry enjoyed the fruits of parsing theory and formal language theory in using the tools Lex and Yacc to create their EQN typesetting language. Knuth has this saying that the best theory is motivated by practice and the best practice by theory. I internalized that with my early experience in the Computing Sciences Research Center because I found that the theory that we were developing in computer science could be applied to document preparation systems, programming languages, compilers, and so on. It was really a very productive environment. I taught courses on compiler design at local universities, and then when I went to Columbia, I would teach the course on programming languages and their translators…

I might point out that the first Fortran compiler developed by IBM in the 1950s took 18 staff years to create. In my programming languages and compilers course, I organized the students into teams of four or five. Each team had to create their own programming language, and then write a translator for it, and in all the time that I taught the course for almost 25 years at Columbia to thousands of students, never did a team failed to deliver a working compiler in the 15-week course, and I attribute that to the abstractions [2]

Aho: Okay. AWK is a programming language that was created by me, Brian Kernighan, and Peter Weinberger. Hsu: And it's your three initials that are in. Aho: Yes. I'm the A in AWK. Weinberger is the W in AWK and Kernighan is the K in AWK.

We thought that it was just a throwaway tool for us, nobody really would be interested in it. But it's amazing how much routine data processing there is in the world.

The reason the language got to be known as AWK was because when our colleagues would see the three of us in one office or another, and when they'd walk past the open door, they'd say, AWK, AWK, AWK as they were going down the corridor. So we had no choice but to call it AWK because of the good-natured ribbing we got from our colleagues, and because at some Unix conference, they passed out t-shirts that had AWK, and the error message saying "bailing out on or near line five" on them.

[TODO: Ratfor, AMPL, other Kernighan languages.] [TODO: continue with typesetting...]

3.5 The Dragon Book

Jeff had bought into this idea that it's good for your career to write a book about what you're working on. In the '70s, with all this work on Unix and C, there was a lot of interest in creating new programming languages and compilers. As with the algorithms book, what we did was we performed research on efficient algorithms for parsing and for some of the other phases of compilation, wrote papers on those and presented them at conferences. But we took the important ideas that we developed and the community had developed over several decades and codified them into what are now called the dragon books. The first dragon book was published in 1977.

We did have theorems and proofs in the book, and Jeff had this brilliant idea that the book should have a cover with a fierce dragon on it representing the complexity of compiler design, and then a knight in armor with a lance. The armor and the lance were emblazoned with techniques from formal language theory and compiler theory to slay the complexity of compiler design…

In the 1980s, more was known about how to construct efficient compilers. We invited Ravi Sethi as a third coauthor, he was at Bell Labs at the time, to join us in

> creating the second version of the dragon book. In the first version, the dragon was in red. This second version, the dragon was– sorry. In the first version it was in green. In the second version the dragon was in red. What was interesting about the red dragon book was there was a movie that was created in 1995 titled Hackers with a young Angelina Jolie in it, and in the movie, there is the uber hacker that's explaining to the new hackers what you have to read to become an uber hacker. He shows them 10 papers and books that you must read, and one of them was the red dragon book. When my two children saw this movie, and they had seen the red dragon book at home, this is the first time they thought their old man was really something because he had one of his books in a Hollywood movie. It shows what you have to do to impress your kids these days. The red dragon book was 800 pages. In 2007, we invited Monica Lam as a fourth coauthor to create a third version of the dragon book that had a purple dragon on the cover and it was close to a thousand pages. None of us had the heart to write a fourth book at this point because it just shows how much new knowledge had been created in the area of programming languages and compilers and their translators, and we continued to do research in this area to keep up with it. [2]

[TODO: Bjarne Stroustrup, C++ (1979); Dennis Ritchie, C (1972); Ken Thompson, B (1969); Brian Kernighan, AWK (1977), AMPL (1976), co-author of The C Programming Language (1978)]

3.6 Commoditization

[TODO: Bill Gates and Paul Allen (Microsoft) | Microsoft BASIC (1975) | Developed the first critical piece of commercial software for personal computers, establishing the doctrine that software should be a purchased, proprietary commodity. Sun microsystems, each part of the company needed to sell to all the others, reason why their compiler was paid; proprietary Unix;]

3.7 Timeline

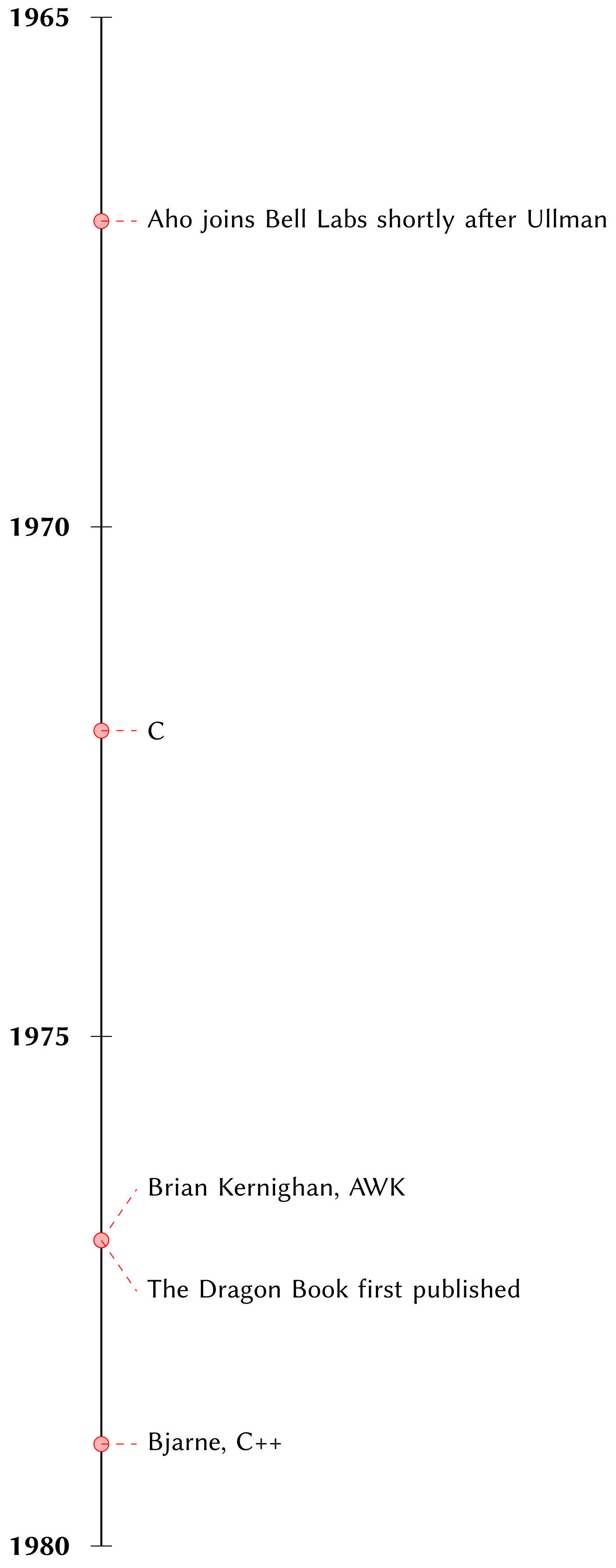


Figure 3.1: TBD, 1965–1980

4
Freedom, 1980-2000

[TODO: gnu linux c llvm python; Facebook php->c++ compiler;] [TODO: lattner tried to get llvm in the gnu project; new licensing, permissive licensing.]

The discussion of the GNU/Linux operating system and the "open source" software movement, discussed last, likewise has deep roots. Chapter 3 discussed the founding of SHARE, as well as the controversy over who was allowed to use and modify the TRAC programming language. GNU/Linux is a variant of UNIX, a system developed in the late 1960s and discussed at length in several earlier chapters of this book. UNIX was an open system almost from the start, although not quite in

the ways that "open" is defined now. As with the antitrust trial against Microsoft, the open source software movement has a strong tie to the beginnings of the personal computer's invention. Early actions by Microsoft and its founders played an important role here as well. We begin with the antitrust trial. [27]

4.1 GNU

4.2 Adoption of Linux

4.3 Low-Level Virtual Machine

Key innovation is that LLVM is a collection of compiler *libraries* that have thin programs wrapping them. Other open-source compilers like GCC tend to be monolithic programs, which can be harder to compose. LLVM contains a collection of sub-projects that can be used independently as tools *or libraries*; key examples being the LLVM subproject of LLVM itself (which contains LLVM IR, the intermediate representation, the optimizer, interpreter and code generator) and Clang, a C/C++/Objective-C compiler front-end for LLVM. Any other compiler project could emit LLVM IR and fully leverage the LLVM project after the front-end. LLVM's intermediate representation has become the lingua-franca of the compiler world.

From its beginning in December 2000, LLVM was designed as a set of reusable libraries with well-defined interfaces [LA04]. At the time, open source programming language implementations were designed as special-purpose tools which usually had monolithic executables. For example, it was very difficult to reuse the parser from a static compiler (e.g., GCC) for doing static analysis or refactoring. While scripting languages often provided a way to embed their runtime and interpreter into larger applications, this runtime was a single monolithic lump of code that was included or

> excluded. There was no way to reuse pieces, and very little sharing across language implementation projects. [22]
>
> The name ”LLVM” was once an acronym, but is now just a brand for the umbrella project. While LLVM provides some unique capabilities, and is known for some of its great tools (e.g., the Clang compiler2, a C/C++/Objective-C compiler which provides a number of benefits over the GCC compiler), the main thing that sets LLVM apart from other compilers is its internal architecture. [22, p. LLVM]

Lattner published his thesis on LLVM in 2002, and joined Apple in 2005.

4.4 Timeline

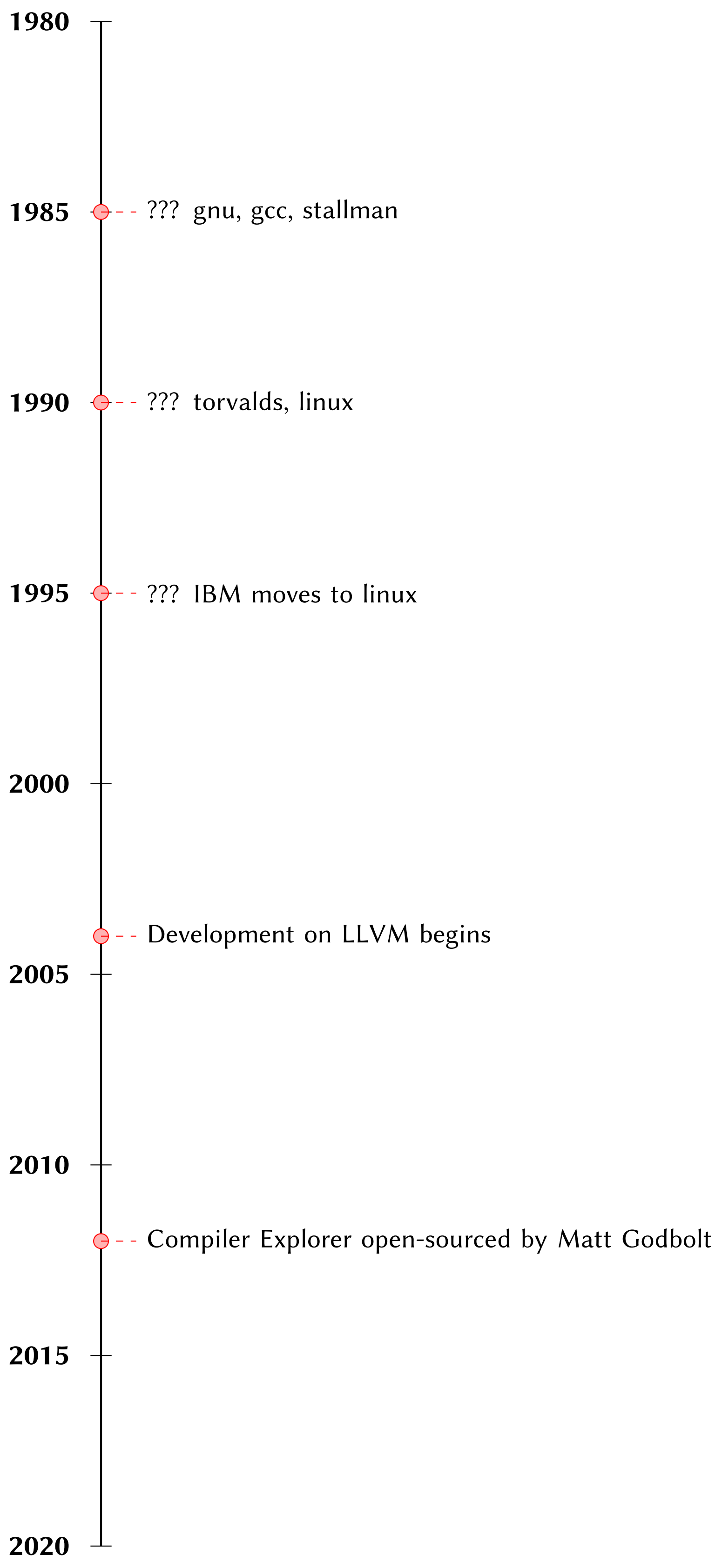


Figure 4.1: Open Source Compiler Development Timeline, 1980–2020

5
Codesign

In each age, contemporaries have attempted to place their work in the overarching history of computing, and some have attempted to look forward and guess at the field's future; in either case, they are often wrong. Computing is a young, volatile industry, and it is often much easier to place place a time period in history retrospectively than it is to do so in the moment. Nonetheless, this is what I will attempt to do in this chapter. I am writing about compilers here in the 2020s, the time period in which I work on compilers. Out of hubris I will attempt to contextualize today's compiler work in the context of history. Please let my future readers be forgiving.

> But there are still lots of interesting open problems left and one of the most intriguing aspects of compiler design is can we use AI, machine learning, and large language models like GPT-3 to create code automatically from written or spoken specifications. That's still an unfolding story and I'm not willing to trust any program created by an AI program at this point. I wouldn't want it in my pacemaker. I wouldn't want it in my self-driving car or in my airplane. But maybe for a computer game, it's okay. This is what they're creating with these at this time. So even the area of programming language translation is undergoing new approaches and how successful they will be is yet to be determined. [2]

5.1 Accellerators

5.2 MLIR

Quotes

This section should not be included in the final copy; it contains quotes and bibliographic references that may be useful in the writing of the book.

5.3 *A Catalogue of Optimizing Transformations*

Allen and Cocke [6]

One of the earliest publications about compiler optimizations. Mentions inlining/IPO.

> The term *optimization* is a misnomer in that it is not generally clear that a particular, so called, optimizing transformation even results in an improvement to the program. A more correct term would be "amelioration."

5.4 "Keynote Address"

Hopper [42]

> Mark I had built-in programs for sine, cosine, exponential, arctangent, .-... On the other hand, because they were wired into the machine, they had to be completely general. Any problem that we solved, we found we did not need complete generality; we always knew something about what we were doing–that was what the problem was. And the answer was–we started writing subroutines, only we thought they were pieces of coding. And if I needed a sine subroutine, angle less than zr/4, I'd whistle at Dick and say, "Can I have your sine subroutine?" and I'd copy it out of his notebook. We soon found that we needed just a generalized format of these if we were going to copy them, and I found a generalized subroutine for Mark I. With substitution of certain numbers, it can be copied into any given program. So as early as 1944 we started putting together things which would make it easier to write more accurate programs and get them written faster. I think we've forgotten to some extent how early that started.

5.5 *The First Computers: History and Architectures*

Rojas and Hashagen [58]

> The chief programmer of Mark I, Richard M. Bloch, kept a notebook in which he wrote out pieces of code that had been checked out and were known to be correct. One of Bloch's routines computed sines for positive angles less that 45 degrees to only ten digits. Rather than use the slow sine unit built into the machine, Grace Hopper simply copied Dick's routine into her own program whenever she knew it would suit her requirements. This practice ultimately allowed the programmers to dispense with the sine, logarithm, and exponential units altogether. Both Bloch and Bob Campbell had notebooks full of such pieces of code. Years later, the programmers realized that they were pioneering the art of subroutines and actually developing the possibility of building compilers.
>
> There were sets of instructions for integers, floating-point numbers, packed decimal numbers, and character strings; operating in a variety of modes. This philosophy had evolved in an environment dominated by magnetic core memory, to which access was slow relative to processor operations. Thus it made sense to specify in great detail what one wanted to do with a piece of data before going off to memory to get it. The instruction sets also reflected the state of compiler technology. If the processor could perform a lot of arithmetic on data with only one instruction, then the compiler would have that much less work to do. A rich instruction set would reduce the "semantic gap" between the English-like commands of a high-level programming language and the primitive and tedious commands of machine code. Cheap read-only memory chips meant that the designer could create these rich instruction sets at low cost if the computer was microprogrammed.

5.6 *Grace Hopper and the Invention of the Information Age*

Beyer [19]

> Though it is sometimes difficult to identify the motivation behind particular inventions, it appears that a dearth of talented programmers, a personal frustration with the monotony of existing programming techniques, and the lack of resources made available by senior management at Remington Rand to support computer clients led Hopper to invent the technologies and techniques, such as the compiler, that allowed the computers to, in effect, help program themselves. Interestingly enough, as her A-0 compiler evolved into the A-1 and the A-2, Hopper's reason ing in regard to the invention changed. Compilers became less about relieving programmers of the monotony of coding and more about reducing programming costs and processing time.

Programming was considered the action of writing machine code directly; writing code in a high-level language was not even considered programming. The motivation for writing Hopper's first compiler was to offload this task to the computer itself.

> First and foremost, the central motivations for automatic programming were far more personal in the 1952 paper. With the construction of a functioning compiler,

Hopper hoped, ”the programmer may return to being a mathematician.” Though Hopper had sincerely enjoyed the challenge of coding since first being introduced to computers 8 years earlier, she wrote, ”the novelty of inventing programs wears off and degenerates into the dull labor of writing and checking programs. The duty now looms as an imposition on the human brain.” By teaching computers to program themselves, Hopper would be free to explore other intellectual pursuits

These computing pioneers, according to Hopper, created machines and methods that removed the arithmetical chore from the mathematician. This chore, however, was replaced by the new burden of writing code, thus turning mathematicians into programmers. Hopper’s paper boldly offers the next step in the history of computing: shifting the humanmachine interface once again so as to free the mathematician from this new burden, and making ”the compiling routine be the programmer and perform all those services necessary to the production of a fi nished program.”

He [the mathematician] is supplied with a catalogue of subroutines. No longer does he need to have available formulas or tables of elementary functions. He does not even need to know the particular instruction code used by the computer. He needs only to be able to use the catalogue to supply information to the computer about his problem.20 The ”catalogue of subroutines” was a menu that listed all the input information needed by the compiler to look up subroutines in the library, assemble them in the proper order, manage address assignments, allocate memory, transcribe code, and create a fi nal program in the computer’s specifi c machine code.21 A subroutine entry in the catalogue consisted of a subroutine ”call-number” and the order in which arguments, controls, and results were to be stated. The call-number identifi ed the type of subroutine (t for trigonometric, x for exponential, etc.), specified transfer of control (entrance and exit points in each subroutine), and set operating and memory requirements. In fact, language such as ”call-number” and ”library” compelled Hopper to name her program generator a ”compiler,” for it compiled subroutines into a program in much the same way that historians compile books into an organized bibliography.22

A program generated by a compiler could not only be run as a stand alone program whenever desired; it also ”may itself be placed in the library as a more advanced subroutine.” This suggested that subroutine libraries could increase in size and complexity at an exponential rate, thus enabling mathematicians to solve problems once deemed impossible or impractical.23

Hopper ends the paper by establishing a short-term roadmap for the future development of compilers. She describes a ”typeB” compiler, which, by means of multiple passes, could supplement computer information provided by the programmer with self-generated information. Such a compiler, she imagines, would be able to automate the process of solving complex differential equations. To obtain a program to compute $f(x)$ and its first n derivatives, only $f(x)$ and the value of n would have to be given. The formulas for the derivatives of $f(x)$ would be derived by repeated application of the type-B compiler.24 Hopper also admits that the current version of her compiler did not have the ability to produce efficient code. For example, if both sine and cosine were called for in a routine, [TODO: make note about me doing this in NVHPC compilers] a smart programmer would figure out how to have the program compute them simultaneously. Hopper's compiler would embed both a sine subroutine and a cosine subroutine in sequence, thus wasting valuable memory and processing time. Hopper states boldly that the skills of an experienced programmer could eventually be distilled and made available to the compiler. She concludes as follows:

Although the test results appear to be a smashing endorsement of the A-0 compiler, Ridgway dedicates a substantial amount of his paper to the ineffi ciency of run-programs. (A "run-program" was the fi nal product of the compiler process. Today, such a program is called object or machine code.) During the 5 months since Hopper had introduced compilers, critics had pointed out that run-programs generated by compilers were less effi cient than those created by seasoned programmers...

Furthermore, an hour of computer time was far more costly in 1952 than an hour of programmer time.

Ridgway acknowledged that using compilers took up more computer time, both as a result of compiling a program and as a consequence of ineffi cient code. But "in this case," he argued, "the compiler used was the 'antique,' or A-0, the fi rst to be constructed and the most ineffi cient." Ridgway was confi dent that Hopper and her team at the Computation Analysis Laboratory would construct new compilers that "squeezed" coding into "neat, effi cient, and compact little packages of potential computation."29

A closer look at the manual for the A-2 compiler (produced by the Computation Analysis Laboratory during the summer of 1953) suggests that, despite the signifi cant improvements over the A-0 compiler, automatic programming had its limitations. Hopper's vision of intuitive, user-friendly, hardware-independent pseudocodes generating effi cient running programs was far from realization. The A-2 provided a three-address "pseudocode" specifi cally designed for the UNIVAC I 12-character standards. The manual defi ned "pseudo-code" as "computer words other than the machine (C-10) code, designed with regard to facilitating communications between programmer and computer."32 Today we refer to it as source code. Since pseudo-code could not be directly executed by the UNIVAC I, the A-2 compiler included a translator routine which converted the pseudocode into machine code. (See table 8.3.) The manual states that the pseudo-code is "a new language which is easier to learn and much shorter and quicker to write."33

The most groundbreaking change was the A-2's ability to debug pseudo-code and fl ag errors automatically. The compiler generated twelve-character error codes that captured the nature of the error, a miraculous innovation for any programmer who had experienced the pain and monotony of debugging computer code. (See table 8.4.)

Pseudocode was actually the innovation of compilers as we know them today, which didn't wasn't part of Hopper's compilers until the A-2. Prior to that, it was really a way to link/load programs from a library of subroutines.

> First, the designer of the compiler now was a linguist. That is, the compiler programmer had the ability to design the syntax of the pseudo-code.
>
> Not only would it be far easier to learn than machine code; its intuitive logic would help users debug their work. "I felt," Hopper recalled, "that sooner or later . . . our attitude should be not that people should have to learn how to code for the computer but rather the computer should learn how to respond to people because I fi gured we weren't going to teach the whole population of the United States how to write computer code, and that therefore there had to be an interface built that would accept things which were people-oriented and then use the computer to translate to machine code."38
> She mentioned that compilers could be designed to program the machine code of any computer. "A problem stated in a basic pseudo-code can thus be prepared for running on one or more computers if the corresponding compiler and subroutine library is available," she wrote. Just as the compiler freed the user from knowing how to program in machine language, pseudo-code was now liberated from a specifi c type of hardware. A payroll pseudo-code could run on a UNIVAC or an IBM computer, so long as the appropriate compiler was running on both. Hopper stated that as of May 1952 such a benefi t was theoretical, insofar as her laboratory had tried it only once, with inconclusive results.39
> Though interpreters were simpler to use than programming in machine code, Hopper believed the approach was a step in the wrong direction. Compiling the A-2 pseudo-code was time consuming in the short term, but the resultant run-program eliminated these six interpretive steps and thus could run more effi ciently. In her fi nal comparison of interpreters and compilers, Hopper wrote: "In both cases, the advantage over manual programming is very great, once the basic subroutines have been tested and proved. The saving of time for a compiler is usually greater."40
>
> Moreover, Hopper's network of invention attracted the enthusiastic participation of many women in the programming fi eld. Nora Moser (of the Army Map Service), Betty Holberton (at the David Taylor Basin), Margaret Harper (of the Remington Rand/ Naval Aviation Supply Office), and Mildred Koss (of Remington Rand) viewed the compiler as more than just a new programming concept. Indeed, they saw it as the centerpiece of an innovative automated system of programming that they had a hand in creating.
>
> The fact that Hopper wholeheartedly welcomed nonUNIVAC personnel to learn about the A-2 compiler sheds some light on her beliefs concerning intellectual property. Hopper did not view software as a commodity to be patented and sold. Rather, she took her cue from the mathematics community. Like most other academics, mathematicians shared information universally, in order to advance knowledge.

Refl ecting on the negative reactions of some of her fellow programmers, Hopper expressed the belief that arguments focusing on "effi ciency" and "creativity" covered far baser motivations: "Well, you see, someone learns a skill and works hard to learn that skill, and then if you come along and say, 'you don't need that, here's something else that's better,' they are going to be quite indignant." In fact, Hopper felt that by the mid 1950s many programmers viewed themselves as "high priests," for only they could communicate with such sophisticated machines.

Hopper was not the only one who came to this conclusion. John Backus, developer of Speedcode and later of FORTRAN, was conscious of the programming community's reaction to his contributions: "Just as freewheeling westerners developed a chauvinistic pride in their frontiersmanship and a corresponding conservatism, so many programmers of the freewheeling 1950s began to regard themselves as members of a priesthood guarding skills and mysteries far too complex for ordinary mortals."54 But the more the likes of Backus and Hopper preached the benefi ts of automatic programming, the more concerned the programming priesthood became about the spreading technology.

5.7 *Dealers of Lightning: Xerox PARC and the Dawn of the Computer Age*

Hiltzik [40]

Vendor lock-in was tremendously powerful at the time, if only for hardware reasons (expensive, difficult to move/transition). The instruction set was only part of the problem. Compilers gave a way to break out of vendor lock-in at the ISA level by giving programmers a higher-level target, but still depended on orgs being able and willing to switch hardware.

Once IBM sold the system to United Airlines, it could rest assured that the frightful effort of rewriting software, retraining, staff, and moving tons of iron and steel cabinets around would make unit very long and hard before replacing its IBM system by one made by, say, Honeywell.

Xerox and SDS execs wanted to try and break out of scientific computing and into business computing, competing with IBM. When the Xerox folks met with potential business customers, compilers were part of the value chain.

Chapter 7. The Clone.

"How good is your COBOL compiler?" they asked... On hearing the question, Bob Spinrad recognized as though for the first time the enormity of the task confronting the company. Scientific and reserach programmers, like thsoe who worked for SDS and its traditional customers, would not be caught dead working in COBOL, which they considered a lame language suitable only for clerks and drones. He shifted uneasily in his chari.

"It's not a question of how good our COBOL compiler is," he told the visitors.

"Why not?"

"Because we don't have one."

The SDS folks didn't understand software's role in value chain for users, and they weren't willing to make hardware changes.

> Headquarters executives thought of software as the gobbledygook that made a machine run, like the hamster driving the wheel. They could not understand why the decision between the PDP-10 and the Sigma needed to be any more complicated than, say, choosing an albino rodent over a brown one.
>
> But from a technical point of view, the issue was hardly tha tcasual. Software was the factor that defined the fundamental incompatibility between the Sigma and PDP machines and the superiority, for PARC's purposes, of the latter. The architectures of the two computers were so radically different that software written for the PDP would not properly fit into the memory space the Sigma allocated for data.
>
> Although it was theoretically possible to simply "port" all the PDP software over to the Sigma, the CSL engineers calculated that such a job would mean rewriting every single line of every PDP program, a task that would take three years and cost $4 million dollars.
>
> ...
>
> They had made an issue out of hardware–what machine they could buy –when their real concern was software–what programs they could run.

PARC folks replicated the PDP-10's instructions in microcode.

5.8 *A New History of Modern Computing*

Haigh [36]

> The development of Unix shifted gradually from assembler to a new higher-level language... Instead of writing, a whole operating system, all that was needed was a C compiler able to generate code in the new machines language, and some work to tweak the Unix kernel and standard libraries to accommodate its quirks.

5.9 "Konrad Zuse's Z4: architecture, programming, and modifications at the ETH Zurich"

Speiser [61]

> [Eduard Stiefel] sent two of his assistants, Heinz Rutishauser and myself, to the United States with the assignment of studying the new technology in order to start a similar project at the ETH. We spent most of the year 1949 with Howard Aiken at Harvard and John von Neumann at Princeton, but we also looked at other installations, among them the ENIAC at Aberdeen and the Mark II at Dahlgren. We gratefully acknowledge the hospitality with which we were received and the openness with which we were given information.
>
> Despite the fact that the Z4 was only barely operational, he decided that the idea of transferring it to Zurich should by all means be considered.

Zuse introduced the `undef`/`poison` values in the original Z4!

In the following sections, expressions such as ”hardware”, ”software”, ”machine language”, ”compiler”, ”architecture” and the like are used freely, although they were unknown in 1950. They only arrived a decade later, but the underlying concepts were quite familiar to us.

Konrad Zuse must be credited with seven fundamental inventions:...

4. Look-ahead execution: The program’s instruction stream is read two instructions in advance, testing if memory instructions can be executed ahead of time.

6. Special values

We also made some hardware changes. Rutishauser, who was exceptionally creative, devised a way of letting the Z4 run as a compiler, a mode of operation which Zuse had never intended. For this purpose, the necessary instructions were interpreted as numbers and stored in the memory. Then, a compiler program calculated the program and punched it out on a tape. All this required certain hardware changes. Rutishauser compiled a program with as many as 4000 instructions. Zuse was quite impressed when we showed him this achievement.

Bibliography

[1] A. V. Aho, S. C. Johnson, and J. D. Ullman. “Code Generation for Expressions with Common Subexpressions”. In: *J. ACM* 24.1 (Jan. 1977), pp. 146–160. ISSN: 0004-5411. DOI: 10.1145/321992.322001. URL: https://doi.org/10.1145/321992.322001.

[2] Alfred V. Aho. *Oral History Interview with Alfred V. Aho*. English. Video interview. Interviewer: Hansen Hsu. Computer History Museum Oral History Collection. Catalogue number 102792705. Duration: 3:00:19. MOV format. Gift of Computer History Museum. Chatham, New Jersey, USA: Computer History Museum, June 2022. URL: https://www.computerhistory.org/collections/catalog/102792704/.

[3] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*. USA: Addison-Wesley Longman Publishing Co., Inc., 1986. ISBN: 0201100886.

[4] Alfred V. Aho and Jeffrey D. Ullman. *The theory of parsing, translation, and compiling*. USA: Prentice-Hall, Inc., 1972. ISBN: 0139145567.

[5] Howard Aiken. *Oral History Interview with Howard Aiken*. English. Transcript. Interview conducted February 26-27, 1973, by Henry Tropp and I. B. Cohen. Part of the Computer Oral History Collection, 1969-1973, 1977. Repository: Archives Center, National Museum of American History, Smithsonian Institution. Washington, D.C., USA: Smithsonian Institution, Feb. 1973. URL: https://mads.si.edu/mads/id/NMAH-AC0196_aike73027.

[6] Francis Allen and John Cocke. *A Catalogue of Optimizing Transformations*. Tech. rep. IBM J Watson Research Center, 1971. URL: https://www.clear.rice.edu/comp512/Lectures/Papers/1971-allen-catalog.pdf.

[7] Randy Allen and Ken Kennedy. “Automatic translation of FORTRAN programs to vector form”. In: *ACM Trans. Program. Lang. Syst.* 9.4 (Oct. 1987), pp. 491–542. ISSN: 0164-0925. DOI: 10.1145/29873.29875. URL: https://doi.org/10.1145/29873.29875.

[8] JAX Authors. *JAX: High performance array computing*. https://github.com/jax-ml/jax. 2024.

[9] Babbage. *A History of C Compilers — Part 1: Performance, Portability and Freedom*. Whistle-stop tour of the evolution of C compilers, emphasizing performance, portability, and freedom. The Chip Letter (Substack). May 5, 2024. URL: https://thechipletter.substack.com/p/a-history-of-c-compilers-part-1-performance.

[10] J. W. Backus et al. “The FORTRAN automatic coding system”. en. In: *Papers presented at the February 26-28, 1957, western joint computer conference: Techniques for reliability on - IRE-AIEE-ACM ’57 (Western)*. Los Angeles, California: ACM Press, 1957, pp. 188–198. DOI: 10.1145/1455567.1455599. URL: http://portal.acm.org/citation.cfm?doid=1455567.1455599 (visited on 10/04/2025).

[11] John Backus. "Can programming be liberated from the von Neumann style? a functional style and its algebra of programs". In: *Commun. ACM* 21.8 (Aug. 1978), pp. 613–641. ISSN: 0001-0782. DOI: 10.1145/359576.359579. URL: https://doi.org/10.1145/359576.359579.

[12] John Backus. "The history of Fortran I, II, and III". In: *History of Programming Languages.* New York, NY, USA: Association for Computing Machinery, 1978, pp. 25–74. ISBN: 0127450408. URL: https://doi.org/10.1145/800025.1198345.

[13] John W. Backus. *John Backus Oral History Transcript.* English. Interviewed by Grady Booch, videography by Gardner Hendrie. 42 pages. Catalogue number 102657970. Acquisition number X3715.2007. Computer History Museum. Sept. 5, 2006. URL: https://www.computerhistory.org/collections/catalog/102657970/.

[14] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. "Compiler transformations for high-performance computing". In: *ACM Comput. Surv.* 26.4 (Dec. 1994), pp. 345–420. ISSN: 0360-0300. DOI: 10.1145/197405.197406. URL: https://doi.org/10.1145/197405.197406.

[15] F. L. Bauer and H. Wössner. "The "Plankalkül" of Konrad Zuse: a forerunner of today's programming languages". In: *Commun. ACM* 15.7 (July 1972), pp. 678–685. ISSN: 0001-0782. DOI: 10.1145/361454.361515. URL: https://doi.org/10.1145/361454.361515.

[16] Friedrick Bauer, Peter Naur, and Brian Randell. "SOFTWARE ENGINEERING". In: *SOFTWARE ENGINEERING.* Garmisch, Germany: NATO SCIENCE COMMITTEE, Oct. 1968. URL: https://www.scrummanager.com/files/nato1968e.pdf.

[17] M. Bellia. "Hierarchical development of programming languages". en. In: *Calcolo* 18.3 (Sept. 1981), pp. 219–254. ISSN: 0008-0624, 1126-5434. DOI: 10.1007/BF02576358. URL: http://link.springer.com/10.1007/BF02576358 (visited on 10/04/2025).

[18] Peter J. Bentley. *Digitized: the science of computers and how it shapes our world.* eng. Oxford: Oxford university press, 2012. ISBN: 9780199693795.

[19] Kurt W. Beyer. *Grace Hopper and the Invention of the Information Age.* The MIT Press, 2009. ISBN: 026201310X.

[20] *Biography of Grace Murray Hopper | Office of the President.* en. 2025. URL: https://president.yale.edu/biography-grace-murray-hopper (visited on 10/04/2025).

[21] Corrado Böhm and Peter Sestoft. *Calculatrices digitales: Du déchiffrage de formules logico-mathématiques par la machine même dans la conception du programme.* Fransk. Original title: DIGITAL COMPUTERS On encoding logical-mathematical formulas using the machine itself during program conception. May 2016.

[22] Amy Brown and Greg Wilson, eds. *The Architecture of Open Source Applications, Volume I: Twenty-four designers explain how their software works.* Available online at https://aosabook.org/en/. Online: aosabook.org, 2011. ISBN: 978-1-257-63804-3. URL: https://aosabook.org/en/index.html.

[23] Herbert Bruderer. *Did Grace Hopper Create the First Compiler? Communications of the ACM.* en-US. Dec. 2022. URL: https://cacm.acm.org/blogcacm/did-grace-hopper-create-the-first-compiler/ (visited on 10/04/2025).

[24] Chandler Carruth. *Modernizing Compiler Design for Carbon Toolchain.* YouTube video, CppNow 2023 Conference. Premiered Aug 17, 2023. Discusses modern compiler architecture and the Carbon language toolchain. CppNow, 2023. URL: `https://www.youtube.com/watch?v=ZI198eFghJk`.

[25] Chandler Carruth. *Understanding Compiler Optimization — Opening Keynote, Meeting C++ 2015.* YouTube video of opening keynote on compiler optimization. Meeting C++ / YouTube. Dec. 19, 2015. URL: `https://www.youtube.com/watch?v=FnGCDLhaxKU`.

[26] Paul Ceruzzi. “”Nothing new since von Neumann”: a historian looks at computer architecture, 1945-1995”. In: *The First Computers: History and Architectures.* Cambridge, MA, USA: MIT Press, 2000, pp. 195–217. ISBN: 0262181975.

[27] Paul E. Ceruzzi. *A History of Modern Computing.* 2nd ed. Cambridge, MA, USA: MIT Press, 2003. ISBN: 0262532034.

[28] John Cocke. *Programming languages and their compilers: Preliminary notes.* USA: New York University, 1969. ISBN: B0007F4UOA.

[29] *Collections.* en. URL: `https://computerhistory.org/collections/` (visited on 10/05/2025).

[30] *Computer History Collection.* URL: `https://americanhistory.si.edu/comphist/` (visited on 10/05/2025).

[31] IBM Corporation. *John Backus — The father of Fortran changed programming forever.* English. Biography of John W. Backus, covering his work at IBM, development of FORTRAN, and contributions to programming languages. IBM. Oct. 9, 2025. URL: `https://www.ibm.com/history/john-backus`.

[32] Jon Gertner. *The Idea Factory: Bell Labs and the Great Age of American Innovation.* Penguin Press. ISBN: 9781594203282.

[33] Matt Godbolt. *Happy 10th Birthday Compiler Explorer!* Blog post celebrating the 10th anniversary of Compiler Explorer. Matt Godbolt’s Blog. June 22, 2022. URL: `https://xania.org/202206/happy-birthday-ce`.

[34] Herman H. Goldstine. “*Annals of the History of Computing* . Bernard A. Galler”. en. In: *Isis* 71.1 (Mar. 1980), pp. 160–160. ISSN: 0021-1753, 1545-6994. DOI: `10.1086/352427`. URL: `https://www.journals.uchicago.edu/doi/10.1086/352427` (visited on 10/05/2025).

[35] Denise Gürer. “Women in computing history”. In: *SIGCSE Bull.* 34.2 (June 2002), pp. 116–120. ISSN: 0097-8418. DOI: `10.1145/543812.543843`. URL: `https://doi.org/10.1145/543812.543843`.

[36] Thomas Haigh. *A New History of Modern Computing.* eng. History of Computing. Cambridge [Massachusetts]: The MIT Press, 2021. ISBN: 9780262542906.

[37] Thomas Haigh and ACM Staff. *Alfred Vaino Aho — A.M. Turing Award Laureate 2020.* English. Citation: “For fundamental algorithms and theory underlying programming language implementation and for synthesizing these results and those of others in their highly influential books, which educated generations of computer scientists.” Association for Computing Machinery. 2020. URL: `https://amturing.acm.org/award_winners/aho_1046358.cfm`.

[38] Richard W. Hamming and Edward A. Feigenbaum. "The IBM 701–7094 II Sequence: A Family by Evolution". In: *Computer Structures: Readings and Examples*. Ed. by C. Gordon Bell and Allen Newell. McGraw-Hill Computer Science Series. Reprinted by the Computer History Museum, Section 1 of *Computer Structures: Readings and Examples*. New York, USA: McGraw-Hill Book Company, 1971. URL: `https://tcm.computerhistory.org/ComputerTimeline/Chap41_ibm7094_CS1.pdf`.

[39] Spencer Henry. *Comp.compilers: Re: History and evolution of compilers*. 1997. URL: `https://compilers.iecc.com/comparch/article/97-10-017` (visited on 10/04/2025).

[40] Michael A. Hiltzik. *Dealers of Lightning: Xerox PARC and the Dawn of the Computer Age*. New York: HarperBusiness, 1999. ISBN: 9780887308918.

[41] G.M. Hopper and J.W. Mauchly. "Influence of programming techniques on the design of computers". In: *Proceedings of the IEEE* 85.3 (1997), pp. 470–474. DOI: `10.1109/5.558722`.

[42] Grace Hopper. "Keynote Address". In: *History of Programming Languages*. New York, NY, USA: Association for Computing Machinery, 1978, pp. 25–74. ISBN: 0127450408. URL: `https://dl.acm.org/doi/pdf/10.1145/800025.1198341`.

[43] J. Howlett, Gian Carlo Rota, and Nicholas Metropolis. *History of Computing in the Twentieth Century*. USA: Academic Press, Inc., 1980. ISBN: 0124916503.

[44] International Business Machines Corporation. *IBM Speedcoding System for the Type 701 Electronic Data Processing Machines*. Technical Manual Form 24-6059-0 (5-54:2M-W). [1953-09-10]. New York, USA: International Business Machines Corporation, 1954. URL: `https://archive.computerhistory.org/resources/access/text/2018/02/102678975-05-01-acc.pdf` (visited on 07/04/2022).

[45] Kenneth E. Iverson. "Notation as a tool of thought". In: *ACM Turing Award Lectures*. New York, NY, USA: Association for Computing Machinery, 2007, p. 1979. ISBN: 9781450310499. URL: `https://doi.org/10.1145/1283920.1283935`.

[46] Brian W. Kernighan. *UNIX: a history and a memoir*. eng. s. l.: Kindle Direct Publishing, 2020. ISBN: 9781695978553.

[47] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. "Numba: a LLVM-based Python JIT compiler". en. In: *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*. Austin Texas: ACM, Nov. 2015, pp. 1–6. ISBN: 9781450340052. DOI: `10.1145/2833157.2833162`. URL: `https://dl.acm.org/doi/10.1145/2833157.2833162` (visited on 10/04/2025).

[48] P. J. Landin. "The next 700 programming languages". en. In: *Communications of the ACM* 9.3 (Mar. 1966), pp. 157–166. ISSN: 0001-0782, 1557-7317. DOI: `10.1145/365230.365257`. URL: `https://dl.acm.org/doi/10.1145/365230.365257` (visited on 10/04/2025).

[49] Phillip A. Laplante. *Encyclopedia of computer science and technology. Volume II, Fuzzy-XML*. eng. Second edition. OCLC: 1032027866. BOCA RATON: CRC Press, 2017. ISBN: 9781315115887.

[50] Chris Lattner and Vikram Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation". In: *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*. CGO '04. Palo Alto, California: IEEE Computer Society, 2004, p. 75. ISBN: 0769521029.

[51] Chris Lattner et al. "MLIR: Scaling Compiler Infrastructure for Domain Specific Computation". In: *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2021, pp. 2–14. DOI: `10.1109/CGO51591.2021.9370308`.

[52] Chris Arthur Lattner. "LLVM: An infrastructure for multi-stage optimization". PhD thesis. University of Illinois at Urbana-Champaign, 2002.

[53] Michael Sean Mahoney. "5 Software: The Self-Programming Machine". In: *Histories of Computing*. Cambridge, MA and London, England: Harvard University Press, 2011, pp. 77–85. ISBN: 9780674274983. DOI: `doi:10.4159/9780674274983-007`. URL: `https://doi.org/10.4159/9780674274983-007`.

[54] J. C. P. Miller. "The Annals of the Computation Laboratory of Harvard University — 1. Vol. I. A Manual of Operation for the Automatic Sequence Controlled Calculator. Pp. xvi + 561 + 17 plates. 1946. 2. Vol. II. Tables of the Modified Hankel Functions of Order One-third and of their Derivatives. Pp. xxxvi + 235. 1945. 3. Vol. III. Tables of the Bessel Functions of the First Kind of Orders Zero and One. Pp. xxxviii + [652]. 1947. 4. Vol. IV. Tables of the Bessel Functions of the First Kind of Orders Two and Three. Pp. x + [652]. 1947." In: *The Mathematical Gazette* 31.295 (1947), pp. 178–181. DOI: `10.2307/3610522`. URL: `https://chimera.roma1.infn.it/SP/COMMON/MarkI_operMan_1946.pdf`.

[55] Jeremy Norman. *Grace Hopper and Colleagues Introduce COBOL*. 1959. URL: `https://www.historyofinformation.com/detail.php?id=778`.

[56] NVIDIA Corporation. *Numba CUDA*. `https://github.com/NVIDIA/numba-cuda/`. The CUDA target for Numba. 2024. (Visited on 10/04/2025).

[57] Raúl Rojas. "The Architecture of Konrad Zuse's Z4 Computer". In: *2021 7th IEEE History of Electrotechnology Conference (HISTELCON)*. 2021, pp. 43–47. DOI: `10.1109/HISTELCON52394.2021.9787324`. URL: `https://ieeexplore.ieee.org/document/9787324`.

[58] Raul Rojas and Ulf Hashagen. *The First Computers: History and Architectures*. Cambridge, MA, USA: MIT Press, 2002. ISBN: 0262681374.

[59] Saul Rosen. "ALTAC, FORTRAN, and compatibility". In: *Proceedings of the 1961 16th ACM National Meeting*. ACM '61. New York, NY, USA: Association for Computing Machinery, 1961, pp. 22.201–22.204. ISBN: 9781450373883. DOI: `10.1145/800029.808498`. URL: `https://doi.org/10.1145/800029.808498`.

[60] Jean E. Sammet. *Programming Languages: History and Fundamentals*. USA: Prentice-Hall, Inc., 1969. ISBN: 0137299885.

[61] Ambros P. Speiser. "Konrad Zuse's Z4: architecture, programming, and modifications at the ETH Zurich". In: *The First Computers: History and Architectures*. Cambridge, MA, USA: MIT Press, 2000, pp. 263–276. ISBN: 0262181975.

[62] David Spickett. *LLVM Fortran Levels Up: Goodbye `flang-new`, Hello `flang`!* 33 minute read. Blog post on the LLVM Project site discussing the renaming of Flang and its development history. LLVM Project Blog. Mar. 11, 2025. URL: `https://blog.llvm.org/posts/2025-03-11-flang-new/`.

[63] Philippe Tillet, H. T. Kung, and David Cox. “Triton: an intermediate language and compiler for tiled neural network computations”. en. In: *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. Phoenix AZ USA: ACM, June 2019, pp. 10–19. ISBN: 9781450367196. DOI: 10.1145/3315508.3329973. URL: https://dl.acm.org/doi/10.1145/3315508.3329973 (visited on 10/04/2025).

[64] D. Whitfield and M. L. Soffa. “An approach to ordering optimizing transformations”. In: *SIGPLAN Not.* 25.3 (Feb. 1990), pp. 137–146. ISSN: 0362-1340. DOI: 10.1145/99164.99179. URL: https://doi.org/10.1145/99164.99179.