A Brief History of Compilers

Asher Mancinelli

November 9, 2025, Revision: 31b3875

Contents

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. The history of compilers has only ever been told in a patchwork of isolated stories, but never told its own rite.

For example, there are wonderful retellings of the very first compilers on Konrad Zuse's Z4 computer at the ETH Zurichand 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-time 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.

There have been several monuments works on the history of computing and a few on the history of programming languages, but in the two and a half decades since the turn of the century there have been ample developments in compiler technology that no comprehensive work has covered thus far. This book aims to fill that gap to a small degree; it is not exhaustive, but contextualizes many of the most important recent developments in the larger narrative of compiler history. The introduction chapter contains a brief version of the entire book; the reader is encouraged to read it first. 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.

0.1 A Note on Structure

You might imagine two primary ways to organize this document. Firstly, you might imagine we pick a topic and narrate its history from start to finish. Each thread of history would be discussed exhaustively. Alternatively, we could take slices of history, examining developments in compiler technology in relation to other efforts going on at the same time. I chose the latter format so the connections between the compiler efforts are clearer, but the former would also be valid.

There are trade-offs to both approaches. I feel particularly limited when one thread of history is not cleanly broken into the time periods I've grouped the chapters and sections into. In these instances, I either extend the narrative outside the defined time period to preserve continuity, or attempt to cross-reference the content sufficiently such that readers that would rather continue with the particular history instead of continuing to the next topic are able to do so. In all cases, preserving continuity and the interactions between histories is prioritized.

1
What to Read Instead

This work is highly derivative. Titans of the computing and information sciences have written extensively about the history of computing and programming languages. While this collection has a unique focus on compilers, there are a number of fantastic related works that readers may want to consider instead.

John Backus's (John W. Backus, 1980) and Knuth and Pardo's (Knuth and Pardo, 1976) are particularly instructive about the very early days of programming languages. Backus has a unique voice–he "talks like a regular guy"–and he wrote many of his thoughts on the average programmer's experience in the 1950s. Similar works on the early days of programming languages tend to reference Knuth and Pardo's work extensively; I found that after reading their book, many of those other works covering the same time period felt derivative.

Jean Sammet's (Sammet, 1969) should not be overlooked. If this work is at all interesting to the reader, they will no doubt find tremendous value in Sammet's works, this one in particular.

If section 3.7 is especially interesting to you, I recommend reading Cardone and J Roger Hindley's (Cardone and J Roger Hindley, 2006). It thoroughly covers the pre-history of higher-order logic and the eventual interactions between combinatory logic, λcalculus, programming languages, and compiler theory.

In general, when discussing topics where I feel another work may interest or inform the reader more than this one, I attempt to point it out.

2
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 (Haigh, 2021):

> 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.

And Jean Sammet used a similar description:

> A compiler is a program, not a piece of hardware. A compiler is simply a program which translates a source program written in a particular programming language to an object program which is capable of being run on a particular computer. A compiler is therefore both language and machine dependent. The most important characteristic of a compiler is that its output iS a program in some form or another and not an answer of any kind. This contrasts with the interpreter...The first completed compiler seems to be the A-0 system developed by Dr. Grace Hopper and her staff at Remington Rand in 1952...(Sammet, 1969)

However, neither definition is 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 kind of program they are contrasted with is interpreters, but distinction is not always 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, targeting CPUs, GPUs, and other specialized hardware (Authors, 2024)(Lam, Pitrou, and Seibert, 2015)(NVIDIA Corporation, 2024)(Tillet, Kung, and Cox, 2019).

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 is a compiler that consumes Python source 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(Alfred V. Aho, 2022). 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 the establishment of a broad definition of compiler technology before embarking on the details of its development. We largely exclude theoretical works like Ada King Lovelace's notes on calculating 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 especially relevant.

3
Dawn, *1940-1960*

3.1 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 Raul Rojas and Hashagen note:

> [Expressions] 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. (Raul Rojas and Hashagen, 2002)

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. (Haigh, 2021, ch.5)

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. John Backus even pointed out how the ambiguity around the term *compiler* makes computing history circa 1950s especially difficult to untangle (John W. Backus, 1980):

> There is an obstacle to understanding, now, developments in programming in the early 1950s. There was a rapid change in the meaning of some important terms during the 1950s. We tend to assume that the modern meaning of a word is the same one it had in an early paper, but this is sometimes not the case. Let me illustrate this point with examples concerning the word ”compiler.”

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

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

- John Backus's FORTRAN I compiler at IBM

I attempt to discuss these in order, however their efforts overlap significantly in time. I try to tell their stories as a whole, though each story contains references to the others;if you find yourself confused by names introduced out of context, please finish the chapter or search for the content within this chapter before giving up.

3.2 The Z4

3.2.1 Konrad Zuse

The historians Paul Ceruzzi and Raul Rojas have written extensively about Zuse and his work on the Z3, Z4, and the special-purpose S1 and S2 machines– the reader is encouraged to consult their works for a more exhaustive account (P. E. Ceruzzi, 1981) (Raúl Rojas, 2000), as well as Konrad's son's account of his father's life(H. Zuse, 2009).

We will meet Konrad Zuse, a German civil engineer, during World War II when he began work on the Z4. 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; he rescued his Z4 by moving it himself to a remote alpine village called Hinterstein(Knuth and Pardo, 1976). While he was funded by the Third Reich at some point, he was primarily *denied* funding by the government. In a lecture in New Mexico in 1976, he claimed that his computers were never used to carry out their *final solution*.

At this time, computers were transitioning from *sequenced calculators* (capable only of executing straight-line arithmetic calculations) to "true" computers capable of control flow. The Z3 was unique in its ability to perform conditional jumps, qualifying it as a "true" computer, more advanced than the sequenced calculators. It also had floating-point arithmetic units, which were missing from other early computers like Commander Howard Aiken's Mark I at Harvard and IBM's 703.

After the war, he began looking for a market for some of his wartime inventions. His machines were unique, containing floating-point arithmetic units and a bespoke memory system. From 1943 to 1945, he worked on the *Plankalkül* (for *Plan Calculus*), an algebraic programming language and possibly the first high-level programming language, though it was not implemented until 1975 in Joachim Hohmann's PhD dissertation (Hohmann, 1979).

Towards the end of the war, Zuse and his pregnant wife fled Berlin. Thanks to some confusion about the purpose of Zuse's machine, he was able to leverage transport vehicles from the German army. At the time, the German military was working on a new generation of rocket, which went by the name *V2*. At the time, Konrad named his machines V1, V2, and so on, where V stood for *Versuchsmodell*, or *trial models*. Thus the German army believed his machine was somehow related to the rocket program, and he was able to commandeer transport vehicles from the army.

He was originally to take his machine to the underground factories in Nordhausen where the V2 rocket was under production. After being shocked by *twenty thousand* concentration camp workers toiling in miles of underground tunnels, he decided to take his wife and computer as far away as possible. Thus they fled to Hinterstein, Germany and then Hopferau, Bavaria, to hopefully restore the machine.

It is unclear how Professor Stiefel learned of Zuse's Z4, but it was in Hopferau where he met Konrad Zuse and first saw the Z4 in action.

3.2.2 The ETH's Acquisition 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. (Speiser, 2000)

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 floating point 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. (Raúl Rojas, 2021)

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. (Speiser, 2000)

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.

3.3 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 **??**, 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 1944as 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." (Beyer, 2009, 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.(Miller, 1947):

3.3.1 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:

> 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 punch card 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 readCharles 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. "Their coworkers in the Harvard lab would jokingly cast Aiken as Babbage and Hopper as Ada King. Aiken ran a rigidly hierarchical 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. In 1946, Commander Edmund Berkeley wrote a report on the conditions at theHarvard Computation Laboratory, which perhaps contextualizes her incapacity to cope.

> 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 ofPennsylvania. He named eleven talented people who had left or been dismissed byAiken 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.

3.3.2 Postwar Collaboration

Hopper was relieved from active service in 1946, but she joined the Aiken's lab to continue working on Aiken's Mark II (a paper-tape sequenced calculator) andMark III (an electronic computer with magnetic drum storage). As the war effort wound down, Aiken and his laboratory found themselves growing in stature. He had the authority to move military personele to his Harvard laboratory at his discretion, which he did. He also expanded the reach of his lab's influence by opening up the computing community. During the war, research and development of computers and programming was closely guarded, but after the war ended, cross-organization collaboration was possible. Aiken started the *Symposium on Large Scale Digital Computing Machinery*in 1947 to foster this collaboration. By this time there were numerous other organizations with computing projects underway in the United States, which Aiken was now permitted to collaborate with:

- Eckert-Mauchly Computer Corporation (EMCC), BINAC and UNIVAC
- Harvard, Mark II and Mark III
- IBM, SSEC
- MIT, Whirlwind
- Institute for Advanced Study, MANIAC
- Engineering Research Associates (ERA)

> 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.

Hopper's postwar fellowship with the laboratory ended in 1949; after a brief stint of unemployment, she 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 Aiken's 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. (Beyer, 2009)

Once it was obvious to everyone in the industry that Hopper was done at Harvard, she had a flurry of offers, but she chose EMCC because of her impression of John Mauchly:

> In 1949 when people knew I had run out the time at Harvard, (and I guess everyone in the industry knew it) practically everyone asked me to come for interviews, including IBM. I went to the IBM headquarters and they gave me a huge [offer]. I was one of the very few people who did not work for IBM. I went for interviews with practically every computer manufacturer that there was at the time. Honeywell, RCA was thinking about it, Burroughs was in it. But it was John Mauchly I just couldn't miss. Working for him was obviously going to be a great pleasure. He was a wonderful guy, one of the best that ever lived. (Grace Murray Hopper, 1980)

3.3.3 Hopper at the 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," (Beyer, 2009) 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; she felt that IBM was too close to Aiken's lab. 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 1950 with Remington Rand, a typewriter and office equipment manufacturer. In 1955, Remington Rand merged with Sperry Corporation to form Sperry Rand.

> Another Hopper programmer, Adele Mildred Koss, was assigned to Commonwealth Edison when the utility approached the Chicago sales office concerning a potential purchase of aUNIVAC 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 IBM701 when it became available. Koss recalled: "I remember GraceHopper'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.'" (Beyer, 2009, Adele Mildred Koss, interviewed by Kathy Kleiman)

Rand's team did not have nearly the programming expertise nor the personnel to support their customers. Compounding with these challenges was the resistance Hopper's team faced from the newRemington Rand management. Remington Rand was a typewriter company and their management was far more familiar with the mechanical punchcard technology of the previous generation of computers than theUNIVAC's magnetic-tape memory. Even Thomas Watson Sr. had similar inclinations about magnetic-tape memory:

> Having built his career on punch cards, Dad distrusted magnetic tape instinctively. On a punch card, you had a piece of information that was permanent. You could see it and hold it in your hand. Even the enormous files the insurance companies kept could always be sampled and hand-checked by clerks. But with magnetic tape, your data were stored invisibly on a medium that was designed to be erased and reused.(Beyer, 2009)

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. (Beyer, 2009)

Shortly after the fiasco with the utility company, 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. She began work on the A-0 in October 1951 in her spare time in order to address this 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 the sales team supported their customers as well as one might expect. Management was as probably as receptive to her ideas about compilation as they were to the UNIVAC I's magnetic-tape memory:

> 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. (Gürer, 2002)

As Hopper later recalled:

> 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. (Grace Hopper, 1978)

Hopper was not the only member of the programming group with ideas about programs generating other programs:

> [Betty Holberton]'s retired. I think she's still part time at the National Bureau of Standards. Everybody's forgotten that she wrote the first program that wrote a program. She wrote that sort-merge generator, and what she did was feed in the specs for the data you were handling and the keys and that sort of thing, and then it generated the sort program for that specific data. That's the first time to my knowledge that anyone used

> the computer to write a program. Betty did that. I don't think she's ever fully received the credit for what she did in that case…
>
> I'm not sure that I would necessarily have gotten done what I did get done if she hadn't been ahead of me, so to speak. Knowing that she had used a program to generate a program, I had a good deal more nerve to go ahead and build the first A-O compiler.

Hopper and the team had decided that the atom of a programming language ought to be the command imperative; verbs and nouns. All programmers, no matter their native language, should be able to understand the verbs acting on nouns. Thus they began working on their first mnemonics, the inputs to the A-0 compiler.

3.3.4 The A-0 Compiler

At this time we should note that the term *compiler* had not yet taken on its modern meaning. Hopper used the terms *automatic programming* and *compiler* to refer to programs that produce other programs, but they did not do the jobs that we associate with compilation today. Once Hopper and her team had developed an environment of collaborative programming, they ran into new problems with re-using each other's code.

> On each of these routines they started with zero, which when you put them into a new program you had to add every one of the addresses to position it in the new program. Programmers could not add. There sat that beautiful big machine whose sole job was to copy things and do addition. Why not make the computer do it? That's why I sat down and wrote the first compiler. It was very stupid. What I did was watch myself put together a program and make the computer do what I did. (Grace Murray Hopper, 1980)

John Backus had this to say about Hopper's A-2 compiler in 1976:

> The above items give some idea of what the word "compiler" meant to one group in early 1954. It may amuse us today to find "compiler" used for such a system, but it is difficult for us to imagine the constraints and difficulties under which its authors worked (John W. Backus, 1980)

Given that the A-2 was more sophisticated than the two prior iterations, this should tell us something about how far their notion of a compiler was from our present day understanding. Let us turn to the 1952 paper *The Education of a Computer* in which Hopper announced her A-0 compiler, which she presented at a Pittsburgh ACM meeting(Grace Murray Hopper, 1952).

In this paper, she dubbed her A-0 as a compiler because it was compiling subroutines from a library into a program, adjusting offsets as necessary. This is far closer to our modern notion of a linker; its role was to copy machine code from different locations into a single output program based on an input program *that was still mostly in machine code*, save for the references to library subroutines. Her notion of compilation was perhaps closer to the regular English meaning of compilation, like that of compiling research papers into a book. Her hope was that "the programmer may return to being a mathematician," though her A-0 did not lift the programmer away from machine code to nearly the same extent as her subsequent efforts would (Grace Murray Hopper, 1952). One may also consider this effort to be the first *standard library*, which would become a major feature

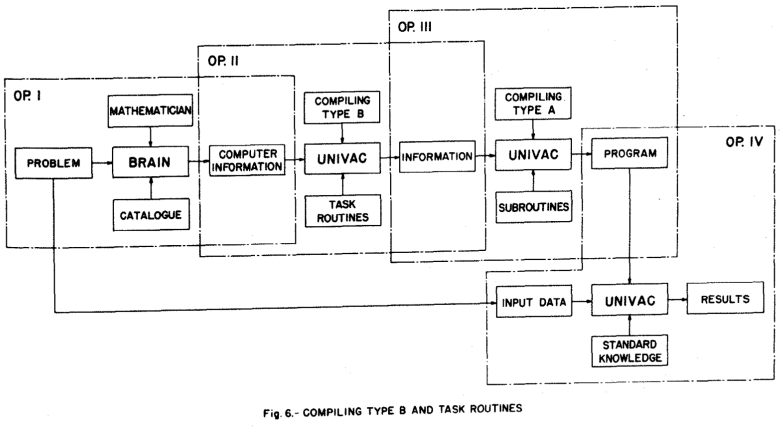


Fig. 6.- COMPILING TYPE B AND TASK ROUTINES

Page 245

Figure 3.1: Depiction of *type-B compilation* from Hopper's *The Education of a Computer*

of later compilers and programming languages. This set the stage for compilers and programming languages providing a default set of useful routines for programmers to pull from.

The cost of renting a computer remained high relative to the labor cost of hiring a programmer, thus it was not always economical for computing centers to use compilers at first. Richard Ridgway and several members of Hopper's team began testing the A-0 against hand-written programs in the summer of 1952 (Ridgway, 1952). For this study, Richard compared the computer and labor time spent to calculate a table of values for the equation:

$$y = e^{-x^2} \sin\left(\frac{x}{2}\right) \quad \text{for } |x| < 1, \Delta x = 0.01.$$

After an analysis of the same program written by hand and by the A-0, he concluded that the A-0 (already considered "antique" by 1952, not more than two years after Hopper began working on it) was cost-efficient to use for most workloads. Richard writes in *Compiling Routines*(Ridgway, 1952):

> Thus, while more UNIVAC time may be required for the numerical solution of a problem as programmed by UNIVAC, more UNIVAC time, in toto, is consumed by the conventional method. This remains true until the entire problem including tis self-contained repetitions is to be repeated, in this case at least eighteen times.
>
> If the total preparation time is considered, the problem must be repeated some 800 times before the conventional programming method overtakes the compiler method. In this case, the compiler used was the "antique," or A-0, the first to be constructed and the most inefficient.

3.3.5 The A-1 and A-2 Compilers

As Hopper and her team began work on the subsequent A-1 and A-2 compilers, their motivations shifted away from reducing the tedium of programming to the economic costs of programming (as is seen in Richard's report). While computing time was initially far more costly than human time, as the proportion of computing costs dedicated to human labor increased over time, the importance

of reducing human labor increased as well. Her team began working on the A-1 and A-2 in 1953, and the A-2 was available to UNIVAC customers by the end of the year. Hopper recruited Herbert Mitchell and Richard Woltman from Aiken's lab to lead the development of the A-1 and A-2 compilers. Margaret Harper, Frank Delaney, Mildred Koss, and Richard Ridgway were the primary developers of these compilers.

There were a number of significant innovations between the A-0 and A-2. The A-2 performed more of the jobs we associate with compilers today. Most significantly, by 1954, the A-2 compiler accepted source code in the form of *pseudocode*, which was a (slightly) more human-friendly format than plain machine code. [TODO: Some sources reference the A-2 compiler instruction manual, but I'm unable to find this source online.] John Backus described the A-2 compiler's 1954 May update as a significant improvement because of these pseudocode instructions (J. Backus, 1978c). He placed the A-2 with Laning and Zierler's algebraic compiler and his own FORTRAN I compiler as the primary compilers of significance in the mid 1950s.

Along with pseudocode instructions, this compiler also produced twelve-character error codes to inform the user why something went wrong during compilation, which must have been tremendously helpful when users had been accustomed to the alternative. The compiler worked in two phases, first constructing an index of the program and then producing the output machine code, informing the user as it did so. The a compiler should be friendly to its users was not necessarily a given at the time; modern compiler tools owe this to Hopper (to the extend that they are any better than the A-1). Another feature of pseudocode instructions is that programs written in such a manner may be translated to run on different machines, provided the compiler and standard library are available on those machines. As far as we know, Hopper never attempted this, however.

By winter of 1953, the A-2 compiler was already being used heavily by several institutions, including the US Census Bureau and Lawrence Livermore National Laboratory. Several of these provided feedback, bug reports, and occasionally features back to the Remington Rand team. Nora Moser of the Army Map Service sent Hopper a collection of comiler improvements, library enhancements, and altogether new libraries in the winter of 1954. One cannot overlook the fact that Hopper was a woman in a male dominated industry, and a number of other women in the industry were her early collaborators.

Hopper presented on the A-2 at the Pentagon in 1953 (Directorate of Management Analysis, Deputy Chief of Staff, Comptroller, Headquarters, U.S. Air Force, and Remington Rand, Inc., 1953), facilitating communication with and amongst her users. This was not the only time she sought to present her team's work to a wider audience. Non-UNIVAC customers also expressed interest; more government agencies and potential customers (with existing IBM machines) came to learn about their developments. This open collaboration and sharing of information would set a precedent for intellectual property in the budding software industry. She had built the compilers as a way to make programming accessible and facilitate the sharing of code, and this philosophy extended to all sorts of information. She had been successful in a more restricted environment at Harvard, and she was able to see clearly the merits of a more open industry.

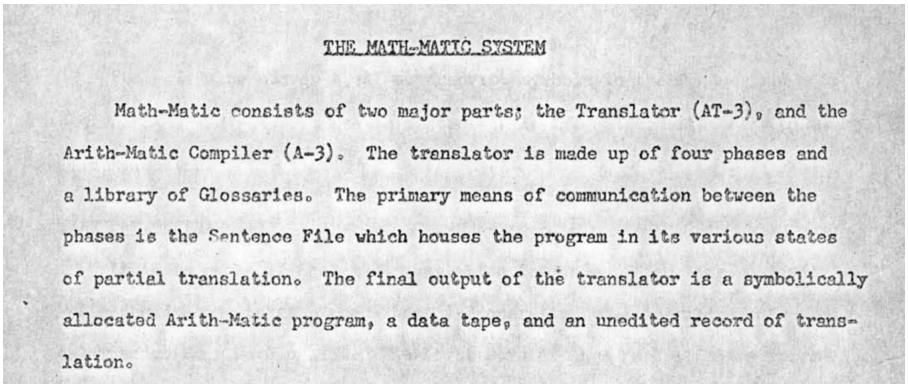
3.3.6 After the A-2: A-3, AT-3, MATH-MATIC

[TODO: Should make some mention of interpreters in here. Hopper made some observations that might be useful later.]

Not everyone was supportive of the advancements in compiler technology. There was a significant chunk of the computing industry that thought programming took too much creativity to be

fully automated. Hopper and John Backus found themselves on the same side of the debate, both firm in their beliefs that computing should be made accessible. Their shared beliefs would lead them to develop competing products in COBOL (at Remington Rand) and FORTRAN (at IBM), respectively. This debate is covered in Section 3.5.3; suffice it to say that the subset of the programming industry that opposed the development of compilers and programming languages was referred to as the "priesthood" of programming, and their rationale seemed more emotional than practical.

Nonetheless, compiler development marched on and the number of compiler developers continued to grow. In 1954, Remington Rand formed the Automatic Programming Department in support of Hopper's team. With Hopper's ability to teach and communicate along with her compassion for the programmer, the group thrived. [TODO: Adele Mildred Koss, interview by Kathy Kleiman, 1993, talking about how nice it was to work for her.] Just as Backus was inspired by Laning and Zierler's work at MIT, so too was Hopper. In preparation for the Office of Naval Research's 1954 Symposium, her new department was focused on extending the A-3 compiler, providing a compiler capable of generating more efficient machine code, but not much else past what the A-2 could do. Inspired by the work coming from MIT, they attempted to extend the A-3 to support source code that resembled equations (more so than the three-address pseudocode that preceded it, at least). This new compiler was called the AT-3 and formed one of the two major components of the MATH-MATIC. The AT-3 was the *Translator* and the A-3 was the *Arith-Matic Compiler* (Ash et al., 1957).



THE MATH-MATIC SYSTEM

Math-Matic consists of two major parts; the Translator (AT-3), and the Arith-Matic Compiler (A-3). The translator is made up of four phases and a library of Glossaries. The primary means of communication between the phases is the Sentence File which houses the program in its various states of partial translation. The final output of the translator is a symbolically allocated Arith-Matic program, a data tape, and an unedited record of translation.

Figure 3.2: Excerpt from the MATH-MATIC User's Manual, 1957

It is worth noting that *Grace Hopper and the Invention of the Information Age* (Beyer, 2009) appears to have a mistake regarding the A-3 and AT-3:

> Hopper and her colleagues explored the possibility of an equation-based programming language, and by 1956 they had modified A-3 to the point where it could support a user-friendly source code. The resultant AT-3 compiler was later named MATH-MATIC.

This led me to believe that the A-3 *became* the AT-3 which was renamed to the MATH-MATIC; however, the MATH-MATIC user manual (Ash et al., 1957) specifies that the MATH-MATIC is composed of the A-3 and the AT-3, which were apparently separate programs with separate names (the Translator and the Arith-Matic Compiler, respectively). See Figure 3.2. Knuth and Pardo describe the MATH-MATIC's development as: "The language was originally called AT-3; but it received the catchier name MATH-MATIC in April, 1957, when its preliminary manual was released." (Knuth and Pardo, 1976) Perhaps the *language* A-3 was extended to become AT-3 which was renamed MATH-MATIC, while the A-3 and AT-3 compiler programs remained distinct; because I cannot find sources

to clarify this nor the original programs, I cannot conclusively describe the MATH-MATIC and its precise relationship to the A-3 and AT-3 past what the user's manual describes.

Because the MATH-MATIC translated source code into A-3 pseudocode as an intermediate step, one might also consider this the first compiler to have an internal intermediate representation. The MATH-MATIC would last until about 1961, at which point UNIVAC users were already expecting FORTRAN compilers available on their UNIVAC systems, favoring IBM's language to Hopper's.

These layers of translation worked against them; the vast majority of codes at the time were dominated by floating point arithmetic, so any degradation of floating point performance would be catastrophic for the overall performance of the system. Around the same time, John Backus had convinced IBM leadership that their new machine, the IBM 704, should have index registers and floating point processing hardware, which is exactly what both UNIVAC and IBM 701 customers lacked and were spending all their time on. Now that the 704 accounted for these prior deficiencies in both machines, Backus's FORTRAN combined with the 704's hardware entirely outclassed the UNIVAC I and Hopper's MATH-MATIC.

> But the MATH-MATIC programmers did not share the FORTRAN group's enthusiasm for efficient machine code; they translated MATH-MATIC source language into A-3 (an extension of A-2), and this produced extremely inefficient programs, especially considering the fact that arithmetic was all done by floating-point subroutines. The UNIVAC computer was no match for an IBM 704 even when it was expertly programmed, so MATH-MATIC was of limited utility.
>
> Knuth and Pardo(Knuth and Pardo, 1976).

3.3.7 The B-0, FLOW-MATIC, and COBOL

In an interview for the Computer History Museum's Oral History series, Hopper explains her motivations for the compilers to follow the A-2 (Grace Murray Hopper, 1980):

> **Pantages:** At that point did you have a feeling for what was happening, in terms of what you were contributing?
>
> **Hopper:** No. I've always objected to doing anything over again if I had already done it once. That was building the compiler. Then I decided there were two kinds of people in the world who were trying to use these things. One was people who liked using symbols - mathematicians and people like that. There was another bunch of people who were in data processing who hated symbols, and wanted words, word-oriented people very definitely. And that was the reason I thought we needed two languages. The data processors did not like symbols, abbreviations that didn't convey anything to them. They were totally accustomed to writing things in words. So why not give them a word-oriented language? And that was part of what was behind Flow-Matic B-0, which became one of the ancestors of COBOL.

Hopper's interests in making programming accessible had not waned, however her target audience shifted. In January of 1955, she shared a her more radical ideas for a new compiler in a report titled *Preliminary Definitions: Data-Processing Compiler* (Grace Murray Hopper, 1955) . This compiler would take on several names: first, the data-processing compiler as outlined in the 1955 paper, then the B-0, the Procedure Translator, and finally the FLOW-MATIC.

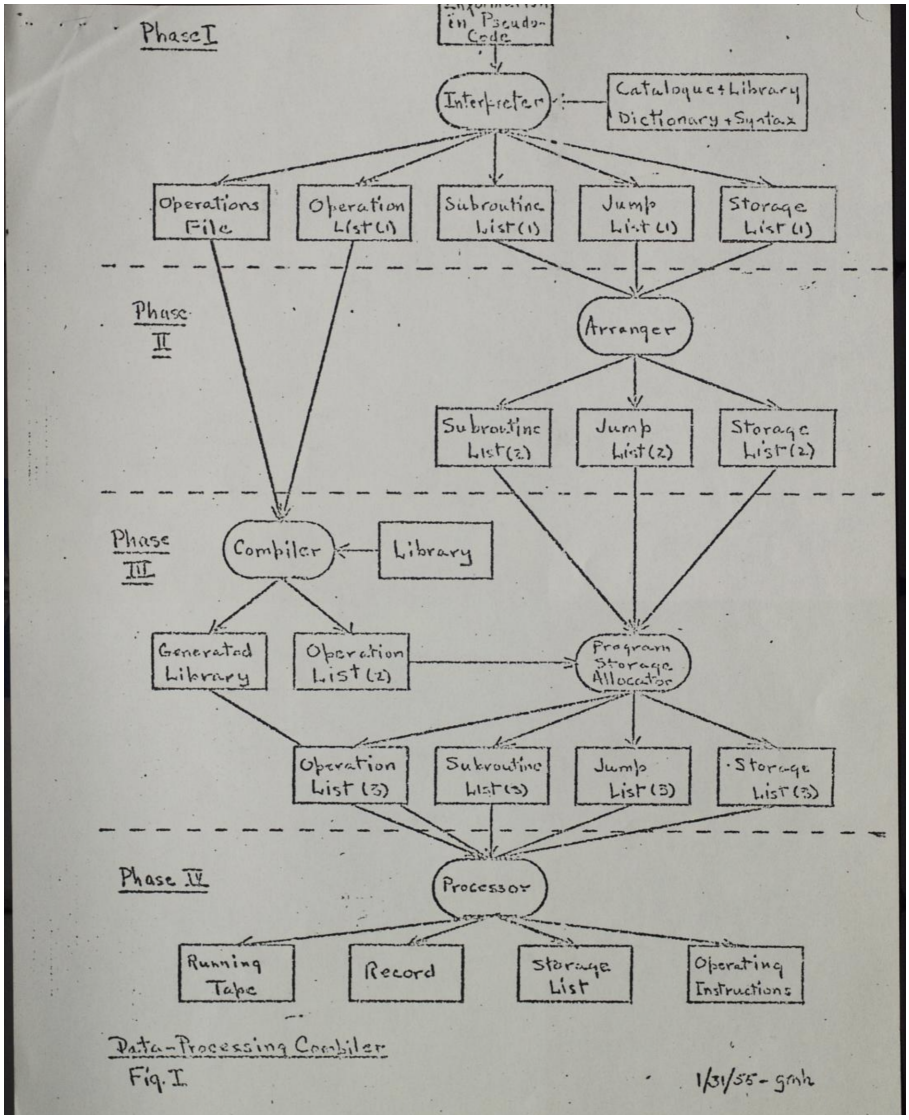


Figure 3.3: Depiction of Hopper's Data Processing Compiler(Grace Murray Hopper, 1955)

> 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", and finally as FLOW-MATIC, This language used English words, somewhat as MATH-MATIC did but more so, and its operations concentrated on business applications. (Howlett, Rota, and Metropolis, 1980)

```
(1)  COMPARE PART-NUMBER (A) TO PART-NUMBER (B) ; IF GREATER GO TO
     OPERATION 13 ; IF EQUAL GO TO OPERATION 4 ; OTHERWISE GO TO
     OPERATION 2 .
(2)  READ-ITEM B ; IF END OF DATA GO TO OPERATION 10 .
```

Figure 3.4: Example FLOW-MATIC code from Knuth and Trabb Pardo(Knuth and Pardo, 1976)

Always concerned with how intelligible her users would find her compilers and programming languages, she focused more on business applications and managers. Donald Knuth and Luis Trabb Pardo describe the FLOW-MATIC as "far more influential and successful, since it broke important new ground." (Knuth and Pardo, 1976) (see also Figure 3.4). Instead of catering her compilers to mathematicians and scientists by introducing mathematical symbols and notation, she sent members of her team to UNIVAC customers to learn about their *business* needs.

The business compiler B-0 was first made available to UNIVAC customers at the start of 1958. Shortly thereafter, Remington Rand merged with Sperry Gyroscope to form Sperry Rand. The marketing department of the newly formed company renamed the B-0 to FLOW-MATIC. Catering to business users, the FLOW-MATIC was programmed in English-like statements such as: `IF EQUAL GO TO OPERATION 5 ; OTHERWISE GO TO OPERATION 2 .` The group toyed with using abbreviations for common words, but after studying their customer's programs, they found that too many abbreviations could be mapped to different words based on the customer, so the abbreviated form was abandoned. Jean Sammet notes: "A preliminary manual for the running system was marked Company Confidential and dated July, 1957; it was available to me at that time since I was an employee of the Sperry Rand Corporation" (Sammet, 1969). The first generally available version of this document was available in early 1958 in *FLOW-MATIC Programming System, U1518 Rev. 1*(Sperry Rand Corporation, 1959). In this manual, one now finds advertisements for the UNIVAC II3.5.

In this document, one also finds a crude method for specifying the language, which the reader should note; Backus would propose a more formal method for specifying programming languages not long after this. For example, the `CLOSE-OUT` command is specified in the FLOW-MATIC manual as [1]:

$$\Delta\text{CLOSE-OUT}\Delta\left[\begin{array}{c}\text{file}\\\text{files}\end{array}\right]\Delta f_1\Delta\left[f_2\Delta f_3\Delta\ldots\Delta f_n\right]$$

There was no format method for specifying languages at the time. In 1959, one of the B-0 developers, Mary Hawes, contacted Sperry Rand about a meeting to discuss the direction of compilers for business applications. [TODO: In some places, Hawes is described as one of the B-0 developers which leads me to believe she worked at RR or SR, but Sammet points out that she worked for the ElectroData division of the Burroughs Corporation. Need to clarify.] Sammet states(Sammet,

[1]The Δ symbols were present in the original manual because it signified an empty space in the punchcards the FLOW-MATIC was written on

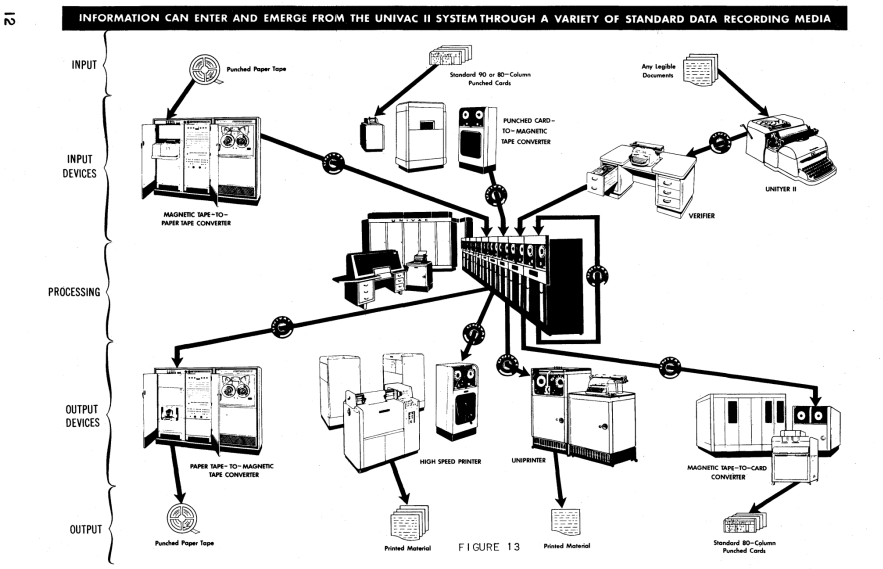


Figure 3.5: Advertisement for FLOW-MATIC and UNIVAC II

1978) that Block's 1959 *Report on Meeting held at University of Pennsylvania Computing Center* names Hawes as having called this meeting:

> ...meeting was the result of a request by Mary K. Hawes (ElectroData Division, Burroughs Corporation) to plan a formal meeting involving both users and manufacturers where plans could be prepared to develop the specifications for a common business language for automatic digital computers.

I am unable to retrieve this document; we will take Jean Sammet's word for it. [TODO: there is not much about Mary Hawes available online, I can't even find the papers Hawes originally wrote. Try to find *Mary Hawes, "Automatic Routines for Commercial Installations"*] Through a connection at the Department of Defense (Charles Phillips), this group asked the DoD to sponsor the meetings and recommended a list of attendees, including 7 representatives from computer manufacturers and 14 from user organizations. In an ACM talk on September 1, 1959, Phillips remarked that "embarrassed that the idea for such a common language had not had its origin by that time in Defense since we would benefit so greatly from the success of such a project." (Sammet, 1969, Phillips, as quoted by Sammet in). There was a great deal of discussion around a common business language that could be shared across different computer systems and reduce the overall cost of programming. *Three separate committees* were formed for this purpose; a short, medium, and long-term committee. Sammet remarks that "the importance of the notion of Short and Intermediate-Range Committees is absolutely crucial to an understanding of the development of COBOL"(Sammet, 1978). She went on to say:

> This wording is clearly ambiguous, and at the time, there was some discussion as to whether or not we were to try and create a language. It was by no means clear *then* that we were to do anything *except* try and *combine* the three known languages of the time. These were FLOW-MATIC, ...AIMACO, ...and COMTRAN

AIMACO was a language developed at the Air Force, and COMTRAN (renamed to Commercial Translator) existed only in an IBM manual whose implementation had never even started. Clearly, Hopper and Sperry Rand's FLOW-MATIC was the most mature of these languages. Indeed, Hopper pointed out that the language resulting from these committees was almost entirely FLOW-MATIC(Grace Murray Hopper, 1980), though the committee notes did not say as much. Hopper's lack of an ego and her diplomatic skills led her to give credit liberally in order to drive consensus.

> We'd written FLOW-MATIC before that, and if you take the FLOW-MATIC manual and compare it with COBOL 60 you'll find COBOL 60 is 95% FLOW-MATIC. So the influence of Commercial Translator in fact was extremely small. But I figured the thing to do was corral those people and when we had something to say, we'd say it was a compound of FLOW-MATIC and Commercial Translator and keep the other people happy and wouldn't try to knock us out. We'd give them some credit and they'd have to get on board with us. But if you compare the two manuals you'd find that it had hardly any influence at all. But if you gave them credit for it, why they'd go right along with you. If you didn't, they'd fight you. You can always give credit, you can always afford to.
>
> That again is the practical. Think about the other guy and his position and his interest. You are always trying to work with people rather than against them. You've got a new idea; give the boss credit for it. It doesn't cost you anything.

Here the reader should note Sammet's observation of the differing goals of the three committees:

> I am certainly convinced in my own mind that had the Short-Range Committee realized at the outset that the language it created (i.e., COBOL) was going to be in use for such a long period of time, it would have gone about the task quite differently and produced a rather different result. But I believe that most of us viewed our work as a stopgap measure–a very important stopgap indeed, but not something intended for longevity.

Most of the disdain for COBOL may indeed come from the fact that it was designed by the *short term* committee, which was solely trying to combine the existing compiler and programming language practices of the time into one language to prevent further bifurcation. If COBOL seems like a mishmash of different ideas designed by committee without proper foresight or thoughtful design, well, it more or less was. Yet another committee was formed on top of these three, the "Committee on Data Systems Languages" or CODASYL, which would be the exeucutive committee steering the direction of the other three.

3.3.8 COBOL Comes Into Focus

Throughout September and October of 1959, the short-term committee met regularly to build consensus around this new language. There were a number of similar names considered before settling on COBOL:

BUSY	Business System
BUSYL	Business System Language
INFOSYL	Information System Language
DATASYL	Data System Language
COSYL	Common Systems Language
COCOSYL	Common Computer Systems Language

On September 18th of that year, the committee settled on COBOL, according to Jean Sammet's personal notes(Sammet, 1978). I spare the reader of the full details of the committee's meetings, and encourage interested readers to consult Sammet's detailed notes on the proceedings. Perhaps the most important conclusion from these meetings was that there existed a *Basic COBOL*, and that no computer manufacturer could claim to support COBOL unless they supported this core subset of the language. This set the stage for numerous modern programming languages that contain a standard language specification that all compilers must support to claim to be a compiler for that language, key examples being C and C++.

The first correct and complete compilation of a COBOL program was performed in August of 1960 on an RCA 501. In December of that year Sperry Rand and RCA each wrote COBOL programs, ran them on their own machines, exchanged programs, and verified that they ran on the other manufacturer's machine as well (a UNIVAC II and an RCA 501). Thus was the first programming language standardized to the point that different compilers running on different machines could compile the same program correctly.

While Hopper's legacy tends to be tied to COBOL, I believe her true legacy lies in her passion for open collaboration and making programming accessible. She thought deeply about the needs of her users; she sent employees to study them and learn what their pain points were and how best the compiler developers could serve them. She considered how to tailor her technology to her users, how to make her compilers more friendly speakers of other languages, and how to make compilers produce understandable error message and make them more debuggable. She facilitated communication between users and developers and advocated for a liberal notion of intellectual property to facilitate the sharing of ideas and programs; this outlook would lead to a cambrian explosion of compiler technology in the open-source era decades after her work on COBOL. This is where I think Grace Hopper's true legacy lies, not solely in the language cobbled together by a potpourri of committees with a short-term view.

3.4 Laning and Zierler at MIT

Of the early compiler efforts, Laning and Zierler's team is perhaps the most overshadowed. Their contemporaries were very impressed by their work, and they inspired a number of innovations at Remington Rand and IBM. Backus described the pseudocode compilers of the time (such as the A-2) as merely providing an instruction set slightly different from the machine's actual code, but not providing any real abstraction; writing the pseudocode still tedious and unproductive. Laning and Zierler's work was different.

John Backus was hugely supportive of their work, though he was not directly invluenced by it prior to starting work on Fortran. In Backus's 1980 paper on the programming landscape of the 1950s (John W. Backus, 1980), he recalls:

> Very early in the 1950s, J. Halcombe Laning, Jr., recognized that programming using algebraic expressions would be an important improvement. As a result of that insight he and Neal Zierler had the first algebraic compiler running on WHIRLWIND at MIT in January 1954. (A private communication from the Charles Stark Draper Laboratory indicates that they had demonstrated algebraic compiling sometime in 1952!) The priesthood ignored Laning's insight for a long time. A 1954 article by Charles W. Adams and Laning (presented by Adams at the ONR symposium) devotes less than 3 out of 28 pages

> to Laning's algebraic system; the rest are devoted to other MIT systems. The complete description of the system's method of operation as given there is the following

In this quote, the *priesthood* Backus is referring to is the group of programmers who rejected the idea that programming should or could be done in a higher level language, and that programming in machine code is an art that would need to be preserved. On his own side of the argument, Backus found Laning and Zierler's work on compiling algebraic expressions into programs to be more accessible and efficient. Grace Hopper was on this side of the argument as well. See Section 3.5.3 for more details about this debate.

Donald Knuth and Trabb Pardo also recall the importance of their work in the development of programming languages and compilers in (Knuth and Pardo, 1976):

> In retrospect, the biggest event of the 1954 symposium on automatic programming was the announcement of a system that J. Halcombe Laning, Jr. and Niel Zierler had recently implemented for the Whirlwind computer at M.I.T. However, the significance of that announcement is not especially evident from the published proceedings [NA 524-], 97% of which are devoted to enthusiastic description s of assemblers, interpreters, and 1954-style "compilers".

Clearly, their work was influential. The compiler system they described is often referred to as simply *Laning and Zierler's algebraic compiler* because they never gave it a name. They described its use in their paper (Laning and Zierler, 1954), which describes problems that Backus and Hopper were also facing, such as syntactic ambiguities, the likes of which would not be worked out until Alfred Aho's work at Bell Labs 4.5.8.

Their user manual gives the following example of an iterative algorithm for calculating the power series of $cosine(x)$:

```
    x = 0,
1   y = 10,
    z = 1,
2   z = 1 - z x^2/y (y - 1),
    y = y - 2,
    e = 1 - y,
    CP 2,
    PRINT x, Z.
    x = x + 0.1,
    a = x - 1.05,
    CP 1,
    STOP
```

While this might seem trivial today, it was a significant deviation from the best-known methods for computing at the time. To write a program with their algebraic compiler, one needn't know anything about the machine the calculation was being performed on–the programmer was free to consider only the mathematical results they were interested in.

They also provided a set of subroutines to their users which was, though quite inconvenient, an early standard library. In (Laning and Zierler, 1954, Section 11. Function Subroutines), they provide a table of supported subroutines:

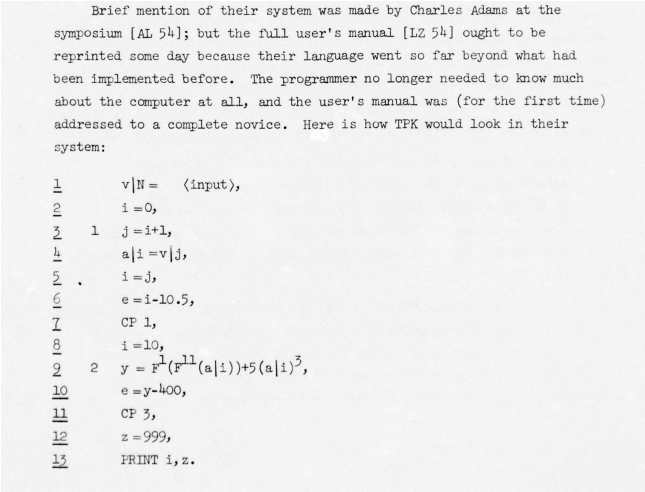
> It is clearly desirable to build into the computer subroutines that compute automatically certain frequently used functions. This process has been begun for our computer and at present the following functions are available:
>
> | Designation | Function |
> |---|---|
> | F^1 | Square Root |
> | F^2 | sine |
> | F^3 | cosine |
> | F^4 | tangent |
> | F^5 | $sine^{-1}$ |
>
> The sine of x may be obtained by writing simply $F^2(x)$ and such an expression may be treated arithmetically in the same way as a variable in an equation.

Inconvenient as it would have been to keep track of the function of each function by number rather than by name, Laning and Zierler were primarily concerned with how readily a programmer could convert their mathematics into a useful program. This higher level of abstraction set them apart from their peers and inspired similar developments.

> 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 (Haigh, 2021)

[TODO: Backus programming in the 50s has lots to say about this.]

> 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 specifications 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 firsthand the efficiency dilemma of compiler-based language design.



Brief mention of their system was made by Charles Adams at the symposium [AL 54]; but the full user's manual [LZ 54] ought to be reprinted some day because their language went so far beyond what had been implemented before. The programmer no longer needed to know much about the computer at all, and the user's manual was (for the first time) addressed to a complete novice. Here is how TPK would look in their system:

```
1        v|N =   ⟨input⟩,
2        i =0,
3    1   j =i+1,
4        a|i =v|j,
5        i =j,
6        e =i-10.5,
7        CP 1,
8        i =10,
9    2   y = F[1](F[11](a|i))+5(a|i)^3,
10       e =y-400,
11       CP 3,
12       z =999,
13       PRINT i,z.
```

Figure 3.6: Knuth and Pardo on Laning and Zierler's Algebraic Compiler

> 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 compiler that maximized efficiency 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 sufficiently economical program at a relatively low cost.13 (Beyer, 2009)

3.5 IBM's Compilers

There were several compiler efforts going on at IBM in this time period; the primary topic of this section is John Backus and his work on FORTRAN. FORTRAN was IBM's answer for scientific computing, but given their namesake (*International Business Machines*), they also had business users who did not necessarily have the same interests as members of the scientific computing community did.

3.5.1 John Backus and Fortran

IBM did not feel that Aiken and the Harvard Computation Laboratory had given them sufficient credit 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 (John W. Backus, 2006).

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 pre-medical 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.

3.5.2 Speedcoding to FORTRAN

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(John W. Backus, 2006):

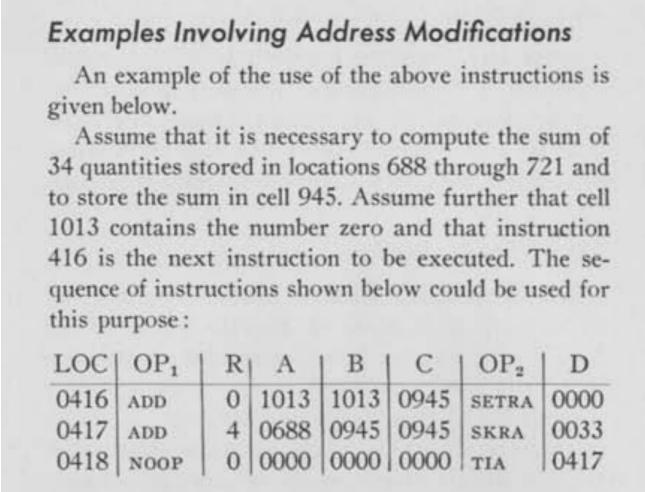
> **Grady Booch:** So then from your experience with the SSEC, you then went on to produce Speedcoding, the Speedcoder...What were sort of the things that influenced you to create that in the first place?
>
> **John Backus:** Well, programming in machine code was a pretty lousy business to engage in, 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?" (John W. Backus, 2006) He credits himself with getting index registers and floating point into the 704.

Backus did not think all that highly of Speedcoding in retrospect, though it gained traction in large part due to IBM's marketing power and the number of users of the 701 relative to the size of the computer market at the time (John W. Backus, 1980). It is unclear whether Backus's assessment of his own code is accurate or if it's born of humility.

> The success of some programming systems depended on the number of machines they would run on. Thus, an elegant system for a one-of-a-kind machine might remain obscure while a less-than-elegant one for a production computer achieved popularity.
>
> This point is illustrated by two papers at the 1954 ONR symposium One, by David E. Muller, describes a floating point interpretive system for the ILLIAC designed by



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 3.7: Excerpt from *IBM Speedcoding for the Type 701 Electronic Data Processing Machine* (International Business Machines Corporation, 1954)

> D. J. Wheeler. The other, by Harlan Herrick and myself, describes a similar kind of system for the IBM 701 called Speedcoding. Even today Wheeler's 1954 design looks spare, elegant, and powerful, whereas the design of Speedcoding now appears to be a curious jumble of compromises. Nevertheless, Wheeler's elegant system remained relatively obscure (since only ILLIAC users could use it) while Speedcoding provided enough conveniences, however clumsily, to achieve rather widespread use in many of the eighteen 701 installations.

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 proposal. He formed a ten-person team of his own choosing based out of IBM's Manhattan headquarters, including Irving Ziller, [TODO: list other members] . They released (International Business Machines Corporation (IBM), 1954) after about one year of working together. Roughly two years after its first conception, FORTRAN was released for the first time. It would go on to ship with every IBM 704 and become the primary means of programming in the scientific community.

Backus could not stand how slow programming was without a higher-level language, and machines were expensive; leasing a machine and spending time programming in machine code wasted money compared to a compiler capable of generating reasonable machine code (though at the time it usually underperformed hand-written code). Backus and his team would continue to develop and stabilize this compiler for several years, though.

> FORTRAN did not really grow out of some brainstorm about the beauty of programming in mathematical notation; instead it began with the recognition of a basic problem of economics: programming and debugging costs already exceeded the cost of running a program, and as computers became faster and cheaper this imbalance would become more and more intolerable. This prosaic economic insight, plus experience with the drudgery of coding, plus an unusually lazy nature led to my continuing interest in making programming easier. This interest led directly to work on Speedcoding for the 701 and to efforts to have floating point as well as indexing built into the 704. (John W. Backus, 1980)

When Backus was forming his team in January of 1954, he was moved from the Pure Science department at IBM into the Applied Science department because his boss Rex Seeber wanted nothing to do with the project. There he found Irving Ziller, who became his first teammate. By April, they had been joined by Harlan Herrick who co-authored the Speedcoding paper with Backus at the ONR symposium "IBM 701 Speedcoding and Other Automatic Programming Systems" in which they observe:

> The question is, can a machine translate a sufficiently rich mathematical language into a sufficiently economical program at a sufficiently low cost to make the whole affair feasible? consider the advantages of being able to state the calculations...for a problem solution in a concise, fairly natural mathematical language.

The reader should note that it is often incorrectly asserted (at times even by Backus himself(John W. Backus, 1980)) that this came *after* Backus and Ziller had been given a demonstration of Laning and Zierler's algebraic compiler for the Whirlwind at MIT at the ONR symposium of 1954. When they received this demonstration, there were already four members of the FORTRAN team, Irving Ziller, Robert Nelson, Harlan Herrick, and Backus himself. In Backus's words(John W. Backus, 1980):

> The article and the letter therefore show that, much to my surprise, the FORTRAN effort was well under way before the ONR symposium and that, independently of Laning (but later), we had already formulated more ambitious plans for algebraic notation (e.g., Gail bjk) than we were later to find in Laning and Zierler's report and see demonstrated at MIT. It is therefore unclear what we learned from seeing their pioneering work, despite my mistaken assumption over the years that we had gotten our basic ideas from them

Indeed, even Grace Hopper made the same assertion at a 1956 symposium:

> A description of Laming and Zierler' s system of algebraic pseudocoding for the Whirlwind computer led to the development of Boeing 's BACAIC for the 701, FORTRAN for the 704, AT-3 for the Univac, and the Purdue System for the Datotron and indicated the need for far more effort in the area of algebraic translators. (Knuth and Pardo, 1976)

Donald Knuth and Trabb Pardo quote her in *The Early Development of Programming Languages* as well. I am not sure who to believe either!

With the support of his new boss Cuthbert Hurd, his family, friends, and his team, the first report of FORTRAN was released externally to 704 users. This brought interest from a variety of users, many of whom offered up members of their teams to help.

The original FORTRAN report had some peculiarities and made some large claims, some of which came true and some of which *very much* did not. Variables had one or two characters, function names of three or more characters, and array variables with up to three subscripts. The document used the term *arithmetic formulas* to describe what we would today call *assignment statements*.

DO formulas were particularly interesting, and not necessarily the construct that we today call *loops*. Any *formula* could have an associated number with it, allowing that formula to be invoked from elsewhere in the program. For example, in the following snippet taken from the original report, the formulas starting with 10 and ending with 14 would be executed 9 times in total, starting with i taking the value of 4 and incrementing by 2 until reaching 20; once completed, control resumes at formula 30:

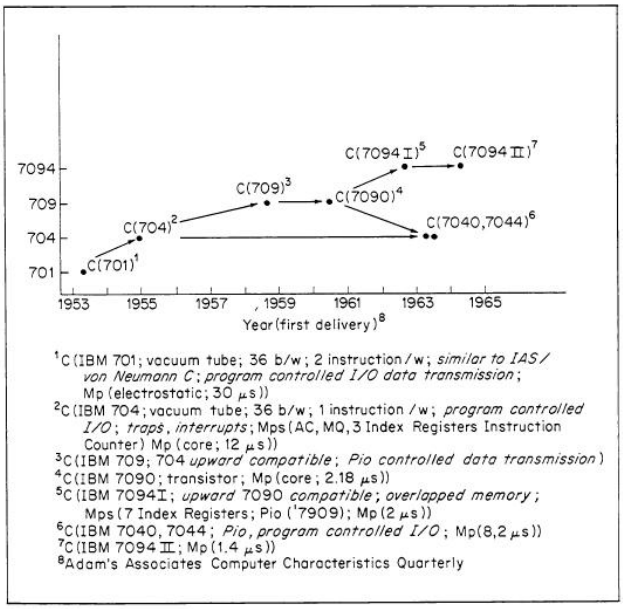


Figure 3.8: Excerpt from *The IBM 701–7094 II Sequence: A Family by Evolution* (Hamming and Feigenbaum, 1971), illustrating the instruction structure for summing quantities.

```
      do 10 14 30 i=4,20,2
```

Whew! For such a fundamental concept, the CFG is potentially very complex. I suspect that in most cases, only one or two formulas were specified. When the third formula number is omitted, execution resumes at the instruction immediately following the final formula of the DO construct, and if only one formula is specified, then that is the last formula to execute, and the first is the one immediately following the DO statement.

Another peculiar feature was the RELABEL formula, which was intended to make matrix rotation operations simpler. For example, for a 4x4 matrix `b`, the formula `RELABEL(3,1)` would result in `b(1,i)` taking the same meaning as `b(i,1)` had prior to the relabel. There soon came better ways to express the same idea, and I am unable to find any example programs using this construct, even in the original report.

The final section of the report notes that "no special provisions have been included in the FORTRAN system for locating errors in formulas…Since FORTRAN formulas are fairly readable, it should be possible to check their correctness by independently recreating the specifications for the problem from its FORTRAN formulation." (International Business Machines Corporation (IBM), 1954, C. DEBUGGING) It is remarkable that they *did* consider how to debug FORTRAN programs, and then *thought better* of providing error messages.

Again, they were tragically optimistic: in the second page of the report, they claimed that "FORTRAN should virtually eliminate coding and debugging."

3.5.3 The Priesthood

At this time, it was not obvious to everyone in the industry that programming languages were needed at all–a large consortium of programmers thought that machine code was all anyone would

ever need. Backus would later describe this group in "Programming in America in the 1950s: Some Personal Impressions":

> 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. (John W. Backus, 1980)

In the Office of Naval Research's 1954 symposium, J. H. Brown and Carr III summarize the recent developments of *automatic programming* techniques like *pre-translation* and *running-translation*, or compilers and interpreters. Their depiction also described the opposition to the development of more friendly input formats for computers, not yet even discussing programming languages:

> Many "professional" machine users strongly opposed the use of decimal numbers for computer inputs or outputs as being unnecessary. Often they felt that direct binary (or associated octal or hexadecimal) coding of instructions was the only possible method. To this group, the process of machine instruction was one that could not be turned over to the uninitiated. (J. H. Brown and Carr III, 1954)

At each step in the development of compilers and programming languages, a non-negligible subset of the programming community has resisted the move towards user-friendliness, and the sentiment J. H. Brown and Carr III described above is perhaps the first. [2]

Backus recalled that this sentiment disuaded efforts towards accessibility: "[t]he priesthood wanted and got simple mechanical aids for the clerical drudgery which burdened them, but they regarded with hostility and derision more ambitious plans to make programming accessible to a larger population." He and Bill Heising would later recall (emphasis added):

> At that time, most programmers wrote symbolic machine instructions exclusively (some even used absolute octal or decimal machine instructions). Almost to a man, they firmly believed that any mechanical coding method would fail to apply that versatile ingenuity which each programmer felt he possessed and constantly needed in his work. Therefore, it was agreed, **compilers could only turn out code which would be intolerably less efficient than human coding**...

Setting aside the emotional arguments against programming languages and making computing more accessible, Backus had very practical and economic reasons as well. He noted that the most costly part of computing was becoming the development of software. In the early days of computing, the cost of the hardware was so astronomically high and the standard for software was so low that the development of software was an afterthought–the organization leasing the computer could easily find programmers to write whatever programs they cared about, and the cost would be a tiny fraction of the cost of the hardware.

As the machines themselves became cheaper and more powerful and the scale of software increased, the cost of software development became more and more important. This will be the focus of Chapter 4. The key players on the side of the debate that favored accessibility and programming

[2] Perhaps engineers working on the ENIAC resisted accessibility as well: "real engineers plug the phone lines of the computer! Punch-card programs are for sissies!"

languages were John Backus, Grace Hopper, J. Halcombe Laning, and Neil Zierler, along with their respective teams.

The sentiment of the priesthood may seem antiquated, but it seems to be present in one form or another in every chapter of this history of programming languages and compilers, and it had a notably positive influence on the development of FORTRAN: because FORTRAN would face significant resistance if it could not produce efficient machine code, Backus, Ziller, and their team devoted a huge amount of effort towards optimization. Their primary fear was that after spending much effort building a FORTRAN compiler, an important application would be written in FORTRAN and turn out to run at half the speed of the machine-coded version, and the sceptics would be right(J. W. Backus and Heising, 1964).

Their work would be foundational for modern optimization techniques, and FORTRAN compilers continue to produce extremely efficient machine code to this day, in part because the language's design allows for a high degree of optimization, and in part because compiler engineers have been optimizing FORTRAN programs for longer than any other programming language.

3.5.4 FORTRAN I

In 1956, they released the reference manual (J. W. Backus, Beeber, Best, Goldberg, H. L. Herrick, et al., 1956) with a FORTRAN that looked very similar to the original, with a few deletions and minor changes. The ill-fated RELABEL formula was removed and the DO statement was simplified. Instead of taking potentially three formulas (the first, last, and continuation formulas), one now only specified the final formula of the loop, or the back-edge.

In the following loop, I begins at 1 and increments by 1 until it reaches 5, and for each iteration, N(I) is accessed, multiplied with I and stored in A(I):

```
10      DO 20 I=1,5
20      A(I) = I*N(I)
30
```

IF-statements were still quite unweildy; they took three formulas, the first if the condition yielded a negative value, the second if it was zero, and the third if it was positive:

```
10      IF(A(I)) 20,30,40
20      ! ... Jump to here if A(I) < 0
30      ! ... Jump to here if A(I) = 0
40      ! ... Jump to here if A(I) > 0
```

To be fair to John Backus, he is quite self-deprecating when recalling these details about the language design process.

> Since our entire attitude about language design had always been a very casual one, the changes we felt to be desirable during the course of writing the compiler were made equally casually...
>
> In our naive unawareness of language design problems–of course, we knew nothing of many issues that were later thought to be important, e.g., block structure, conditional expressions, and type declarations.

At this time, Backus, Herrick, Ziller, and their team at IBM were shopping around their compiler and programming language to potential customers of the IBM 704. Their primary purpose was to

collect aspects of the system that their customers found confusing or objectionable, or aspects they felt were missing. In late 1954 and early 1955, they gave talks to these customers and recieved next to no helpful feedback. It seemed that too frequently, customers had been systems with lots of promise, only to finally recieve an unweildy or peculiar system with restrictions that were not mentioned during the sales process.

The first significant payoff from these talks came in January 1955 when they visited the United Aircraft Corporation. Roy Nutt was leased out by his boss Walter Ramshaw to IBM to work on the input/output libraries for FORTRAN, and he would visit New York several times a week to collaborate with the team until 1957. Charles Adams from MIT allowed Sheldon Best to join the team, and Sidney Fernback from Livermore Radiation Laboratory joined as well.

Backus remarks that while his team could have been better salespeople and that Laning and Zierler's compiler was more influential and compelling at the time, the early FORTRAN efforts likely influenced at least the development of Hopper's Math-Matic compiler at Sperry Rand. If they did not convince users of the utility of their new compiler and programming language, they at least picked up new contributors that would go on to make large contributions.

3.5.5 Their First Compiler

In early 1955 when development on their compiler began, it was only referred to as their *translator*. Herrick wrote lots of test programs to feel out the language, and the team had an ad-hoc understanding of the sort of code they expected to get from various source programs, though their language lacked a rigorous specification. The team broke into smaller teams of one to three people, each responsible for a section of the compiler. Each group agreed on the input and output formats to be used on the boundaries of their component and what came before and after.

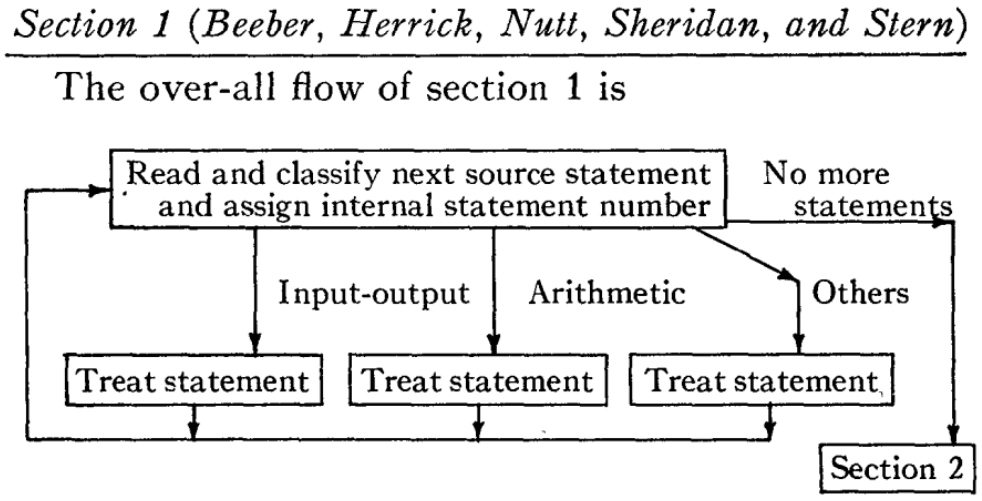


Figure 3.9: Section 1 of the Original FORTRAN compiler(J. W. Backus, Beeber, Best, Goldberg, Haibt, et al., 1957)

Their *translator* was a single-pass compiler. The first section (see Figure 3.9) of the compiler consumed the entire program, produced machine code for what it could, and left the rest of the information about the source program in lookup tables for the other sections of the compiler. We

would likely call this the front-end of the compiler as it composed parsing and a bit of semantic analysis.

In section 2, some parts of the user's program (particularly I/O statements) would be transformed into DO-statements, which would then still need to be compiled. This section also performed some optimization steps, including loop-invariant code motion and register allocation. The optimizations in section 2 were primarily invented by Robert Nelson and Irving Ziller. These optimizations originally included register allocation *for the entire program*, until Backus proposed that the earlier phases of the compiler should assume the target machine (at this point only the IBM 704) has unlimited index registers, and later phases of the compiler would finish the job of register allocation.

This proposal neccesitated sections 4 and 5, and section 3 to translate the output of the first two sections into the format section 4 expected. Lois Mitchell Haibt joined the group to design and program section 4, which was responsible for some significant analyses. Section 2 would be responsible for constructing and analyzing the CFG of the user's program, breaking the program into basic blocks, and *performing a Monte-Carlo simulation* of the user's program based on DO-statements and optimization hints in the form of FREQUENCY statements to track how often each basic block was used, and to collect utilization statistics on the program's index registers.

Wow! That must have been a massive undertaking. Aside from inventing new compiler optimization techniques, this is the first instance I can find of profile-guided optimization. They quickly ran into tradeoffs between compile-time spent in analyses and optimizations and the actual differences in performance these analyses brought to the user's program. In once instance, a subcomponent of the register allocation analysis took ten minutes of the twenty total minutes in compile time, even though the program used no index registers–thus half the compile time went towards analyses that made no difference in the final program (J. W. Backus and Heising, 1964).

Section 5 was then responsible for leveraging the analyses from section 4 to perform register allocation for the 704's three index registers. This section was designed and implemented by Sheldon Best from MIT, who was on loan as a temporary IBM employee. Best's methods would become the basis for register allocation research for many years to come.

It was not obvious to the team until later in 1955 that a section in between 2 and 4 would be needed; Richard Goldberg joined in November of 1955, and he designed and implemented this section, and took over section 5 from Best after he returned to MIT.

Section 6 was naturally designed by Nutt, as he wrote the I/O runtime libraries for the original compiler, and section 6 assembled, linked, and loaded the final program. For such an I/O-heavy portion of the compiler, he was the natural choice. Thanks to the efficient I/O features Nutt added to the language, this section was many times faster than competing assemblers and linkers.

From mid 1956 to early 1957, the focus was on bringing the compiler to market. They would rent out a hotel near the headquarters to sleep during the day so they could debug their new compiler all night long while it sat unused by other teams. The team started surprising themselves with the transformations their compiler was capable of; the optimization gurus Nelson and Ziller would always be able to figure out how the compiler determined the transformation was correct and beneficial, but it was still a surprise to see all the pieces come together.

In 1957 they published a small addendum to the Programmer's Reference Manual, documenting *functions statements*, which have persisted through to modern Fortran. They allow users to define functions in a single statement, like so:

```
FIRSTF(X) = A*X+B
SECONDF(X, B) = A*X+B
```

```
THIRDF(D) = FIRSTF(E)/ D
FOURTHF(F,G) = SECONDF(F, THIRDF(G))
```

There were numerous language features specific to the 704 as well, such as the SENSE LIGHT statement, which toggled specific sensor lights:

GENERAL FORM	EXAMPLES
”SENSE LIGHT i” where i is 0, 1, 2, 3, or 4.	SENSE LIGHT 3

The PUNCH statement was also present in the handbook, which allowed users to directly use the card-punching attachment on the 704.

Among the optimization efforts, Nelson and Ziller optimized array indexing expressions and wrote loop analysis passes. Backus specifies that they could ”could move computations from the object program to the compiler” which appears to be the first instance of constant folding in a compiler.

In the spring of 1957, they published “The FORTRAN automatic coding system” and they distributed it to every IBM 704 customer. After lots of debugging in the field and difficulties punching out copies of the compiler, it was not long before most of the installations were using it. By fall of 1958, more than half of the machine code produced at the roughly 70 installations with an IBM 703 or 704 were being produced by their FORTRAN compiler.

3.5.6 Optimizing FORTRAN I

Driven by the fear of the sceptics’ fears coming true and delivering a FORTRAN compiler that produced programs significantly slower than their hand-coded equivalents, they tried to evaluate which parts of a program would be most subject to inefficiencies. One of the first considerations was the calculation of *addressing expressions* (J. W. Backus and Heising, 1964) (which continues to be a challenging optimization problem). This will be explained further below.

When a program iterates over the elements of a matrix $A_{m,n}$ with m rows and n columns in a loop, the element of $A_{i,j}$ is accessed in column-major order with the following calculation:

$$A_{i,j} = \text{base-address}(A) + ((i - \mathit{lbound}(A, 1)) + m \times (j - \mathit{lbound}(A, 2))) \times s$$

where s is the number of bytes in an element of A, and *lbound* gives the lower bound of the array for a given dimension. In many cases, the lower bound of all dimensions is 1, which gives a more straightforward calculation:

$$A_{i,j} = \text{base-address}(A) + ((i - 1) + n \times (j - 1)) \times s$$

And even more simply, the offset of an index from the base address of a matrix with all-one lower bounds measured in units of the size of an element is simply:

$$A_{i,j} = (i - 1) + n \times (j - 1)$$

If the loop accesses the elements of A such that each element resides immediately next to the previous element in the innermost loop, then the compiler can re-use the address calculation from the previous iteration of the innermost loop.

More concretely, here we see the elements of our matrix A:

$$A = \begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,N} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,N} \\ \vdots & \vdots & \ddots & \vdots \\ a_{M,1} & a_{M,2} & \cdots & a_{M,N} \end{bmatrix}$$

Thus the elements are laid out in memory as a contiguous block of memory:

$$(a_{1,1},\ a_{2,1},\ \ldots,\ a_{M,1},\ a_{1,2},\ a_{2,2},\ \ldots,\ a_{M,2},\ \ldots,\ a_{1,N},\ \ldots,\ a_{M,N})$$

So, a FORTRAN program accessing elements of A linearly as they are stored in memory (also known as stride 1) would look like this:

```
      real A(3,3)
      integer i,j
c     Innermost loop is the fastest-moving index
      do 10 j = 1, 3
         do 10 i = 1, 3
            A(i,j) = real(i)/real(j)
            print *, i, j
   10 continue
      end

c     Program prints:
c     1           1
c     2           1
c     3           1
c     1           2
c     2           2
c     3           2
c     1           3
c     2           3
c     3           3
```

Now, the FORTRAN team's concern was that their compiler would not be able to recognize that a program like the one above was accessing memory in a way consistent with the way arrays are stored in memory, and would miss the fact that in calculating the address of an array element $A_{i,j}$ for $i > 1$, parts of the address calculation for index $A_{i-1,j}$ could be re-used and $A_{i-1,j}$ and $A_{i,j}$ reside next to each other in memory.

Because programmers writing machine code would naturally do this and FORTRAN programs would spend much of their time in loops iterating over matrices, it was critical that their compiler be able to recognize this idiom. After scoping out the problem, they found the space of possible address calculation expressions to be very large indeed.

For example, in the loops above, it is obvious that the matrix A is accessed with stride 1 because the innermost loop iterates over the columns of the matrix, and the induction variable i is incremented by one via the DO loop, and is not modified in the loop body. Consider the following program, where it is not *quite* so clear:

```
      real A(3,3)
      integer i,j
c     Innermost loop is the fastest-moving index
      do 10 j = 0, 2
         do 10 i = 1, 3
            i = i + 1
            A(i,j) = real(i)/real(j)
            print *, i, j
   10 continue
      end
```

Does this loop access A with stride 1? Yes, it does, but I hope the potential problems are becoming clear. What if i is not simply a constant offset of the innermost dimension of A? What if the first index of A is the result of a function call which takes i as a parameter and performs some calculation based on it? This sort of analysis would go on to be a rich subset of compiler optimization research, and the answers are not always straightforward. Their work on FORTRAN would become the basis for loop analyses attempting to prove that address expressions are linear functions of induction variables, permitting impactful compiler optimizations.

To match or exceed the efficiency of hand-coded loops, the group settled on a few language restrictions that would need to be adhered to (J. W. Backus and Heising, 1964):

- DO-loop induction variables must be incremented by a positive, constant integer.
- Array subscripts must be linear (or *affine*) functions of induction variables.
- The number of subscripts must not exceed three.
- Control should not jump into a loop body from elsewhere in the program.

Outside of these restrictions, other were imposed for the performance of the *compiler* and not necessarily for the user's program. For example, identifiers could not exceed six characters.

3.5.7 FORTRAN II

FORTRAN II was developed primarily to address the restrictions required for the optimization of FORTRAN programs discussed in the last section and quality of life enhancements to address common issues programmers reported. After debugging in the field, the team noted the need for improved compile time, proper diagnostics, and subroutines. FORTRAN II was mostly designed by Irv Ziller, John Backus and Robert Nelson, and it was implemented by Lois Mitchell, and the first report to contain these additions was *Proposed Specifications for FORTRAN II for the IBM 704* in 1957.

Some statements in this report did not make it to the final language, but it nonetheless set out the design for more structured programming. Subroutines were defined with `SUBROUTINE DEFINITION, NAME(ARGUMENT1,ARGUMENT2)` which is not *too* far off of the language as it would come to be. The *END* and *RETURN* statements were also introduced.

Separable compilation features were introduced as well. In the original FORTRAN, the entire program was a single FORTRAN program that had to be compiled as one translation unit, thus

any changes to any part of the program required recompilation of everything. This became quite expensive as the team continued to add optimizations and analyses.

In the new separable compilation model, information from the symbol table of the original program could be kept in the object code so that a loader could combine references between subroutines compiled in different source files into one program. The loader program, the *Binary Symbolic Subroutine (BSS) Loader*, would load the separately compiled subroutines together into one program using the symbolic information kept in the *transfer vectors* in object code. [3] The transfer vectors were placed at the start of the compiled subroutine and contained the symbolic information about the subroutine to follow.

This allowed the final deck of punchcards to be placed in the card reader with each independently compiled subroutine in any order, and the symbol information allowed the BSS loader to build a table of the available subroutines and replace symbolic references to external subroutines with actual addresses. The BSS loader was a two-pass program: the first pass traversed all the compiled subroutines and scanned their transfer vectors, storing the actual addresses of the referenced subroutines in a symbol table. The second pass replaces occurrences of symbolic references to subroutines with their actual respective addresses. The proposal suggests that programs of mixed FORTRAN and machine code could also be loaded in this way, but did this was merely a suggestion.

The biggest gripe users had with this version of FORTRAN was likely still compilation time. While this edition of the language allowed for separable compilation and users did not need to recompile the entire program after every single change, the compiler still spent significant time during optimization. When writing and debugging new programs, this optimization effort was wasted, and there were no tuning options to allow users to opt-out, as modern compilers do.

Users adopted FORTRAN II widely and immediately, and the use of the language accellerated greatly in this period. Jean Sammet remarked that a significant portion of the resources on FORTRAN from the early 1960s did not explicitly mention *FORTRAN II* anywhere even though they covered material specific to that version of the language, so some sources that appear to cover FORTRAN I may really cover the second version(Sammet, 1969): "In other words, FORTRAN II was issued so relatively soon after FORTRAN I that the distinction rapidly became blurred and to some extent was even dropped, although it was clear in Program Library references."

3.5.8 SHARE

At this point in FORTRAN's history, there was such significant adoption of many of IBM's innovations that a volunteer user's group, SHARE, had grown to significant membership. SHARE was founded in 1955 and gained traction shortly after (Akera, 2001), made up of IBM customers managing installations of the 704, and collectively aimed to deduplicate the effort of managing the system (Armer, 1980).

This group was an early precursor to the open-source movement. Prior to this period, most machines were used for accounting and were for highly specialized machines that constituted hardly more than a calculator. Once more advanced machines were introduced, the task of writing software and maintaining the hardware became increasingly difficult.

Out of this difficulty, organized user groups like SHARE emerged alongside ad-hoc collaboration, like Grace Hopper's collaboration with [TODO: some air force group I think? where they would send

[3] The symbolic information kept in object programs that the BSS loader used is not the same as the BSS section in ELF programs on modern computers– the latter is used for program-scoped uninitialized or zero-initialized data.

her fixes and enhancements to the compilers.]

3.5.9 FORTRAN III

While Lois was carrying out Ziller, Backus and Nelson's design for FORTRAN II, Ziller was already working on a more advanced version of FORTRAN, FORTRAN III. Ziller's design incorporated a form of inline assembly that allowed 704 instructions to take the addresses of FORTRAN variables as arguments. Modern forms of inline assembly allow the user to write assembly as *text* in the host program, which the compiler then embeds in the corresponding part of the surrounding program–Ziller made the unfortunate decision of making IBM 704 instructions available *at the language level*, which, as Backus remarked (J. Backus, 1978c), doomed FORTRAN III to die roughly as soon as the IBM 704 was replaced. Some subsequent versions of IBM's scientific computing machines (like the 709) were partially backwards-compatible and capable of running 704 instructions, but Backus's evaluation of the situation was more true than it was not.

Ziller and the team also introduced boolean expressions and the capability to pass functions and subroutines as arguments and FORMAT codes for printing alphanumeric strings. This version of FORTRAN was never widely used and was only distributed to 20 or so installations in the winter of 1958. It was in operation until the 1960s using the IBM 709's compatibility mode, which kept FORTRAN III's machine-specific IBM 704 features alive for a bit longer than the 704 itself.

3.5.10 FORTRAN IV

By 1958, most of the original FORTRAN team had moved on to other work and some members of the original team, especially John Backus, were unhappy with the direction they took. The following edition, FORTRAN IV, was more of a successor to FORTRAN II than it was FORTRAN III given the latter's machine-specific features and lack of adoption.

The IBM user's group SHARE in 1961 declared their desire for a new FORTRAN with new features and lacking all the bagage of FORTRAN II. This appeard in the 1963 edition of *Datamation* where the SHARE group discussed their automatic translator, discussed above in section 3.5.8.

FORTRAN's adoption had grown so large and the set of features so restrictive that in 1961 the team at IBM decided to create a new compiler conceptually based on the FORTRAN II compiler, but completely rewritten, as SHARE had requested. They used the opportunity not only to introduce new features but also to correct some design decisions made in the FORTRAN II compiler. This compiler would go on to run on the IBM 7090/94. Among the larger features they introduced, there were many small changes that cleaned up the language and made it suitable for future machines, such as the deprecation of the `PUNCH` and `TAPE` instructions in favor of a proper I/O library and runtime system.

To address the long compilation times that users of FORTRAN II dealt with, the team Programming Research Department responsible for FORTRAN IV tried to deliver the best of both worlds by introducing a new compiler that was both faster and generated more efficient code, instead of adding different compilation modes. Modern compilers let the user choose to enable optimizations via command-line flags. For example, most C, C++ and FORTRAN compilers provide the `-ON` flag, where `N` is a number between 0 and 3 specifying how aggressive the compiler ought to be.

As a result, IBM's FORTRAN IV compiler was slower than other fast FORTRAN compilers like the *WATFOR* and *WATFIV* compilers developed at the University of Waterloo (the **WAT**erloo **FOR**tran

compiler(Cress, Dirksen, and J. W. Graham, 1970)), and in general it did not produce machine code as fast as IBM's FORTRAN II compiler.

The team introduced the labeled COMMON block in FORTRAN IV, which allowed separably compiled modules to share data with each other, and required that each module agree on the size and layout of the data. [4] Programmers found this useful because they could share data between modules without needing to recompile the entire program after every change (unless the layout of the common block changed!), but there are lots of mistakes that are hard to avoid with COMMON blocks.

For example, take the following subroutines, and assume they are defined in different files (and thus different translation units):

```
! f.F
subroutine f()
  real n
  real A(10)
  common /lab/ A, n
  ! uses of A and n...
end subroutine

! g.F
subroutine g()
  real n
  real A(5)
  common /lab/ n, A
  ! uses of A and n...
end subroutine
```

Notice how the declarations of A and n differ between the two subroutines. The array A is declared with a different size in each subroutine, and the order of the variables is different! If the programmer of the subroutine g assigns to n, they will be assigning to the same memory referred to by the first element of A in the subroutine f.

The team went to great lengths to make FORTRAN IV a superset of FORTRAN II so all the existing users could keep compiling their code with the newest compilers, but there were some features they just couldn't support. The IBM user's group *SHARE*section 3.5.8 came up with a translation program *SIFT*, or *S*HARE *I*nternal *F*ORTRAN *T*ranslator, to help users upgrade their FORTRAN II code to FORTRAN IV. [5] The term sift grew to become a more general term for this sort of translation–there were many efforts in this direction in the Python 2 to 3 upgrading process.

The first instance of this process was before the term sift existed or was used to mean translation more generally, with the ALTAC to FORTRAN II translator (Olsen, 1965).

[4] Sammet notes that COMMON blocks were introduced in FORTRAN II (Sammet, 1969, Section IV.3.1.), so this might need a fact-check. Or was it only *labeled* COMMON blocks? I specify *labeled* COMMON blocks here since I'm not sure.

[5] Jean Sammet discusses this in *Programming Languages: History and Fundamentals* and cites the article *SHARE Internal FORTRAN Translator* in Volume 9 of *Datamation* in 1963, but I'm unable to track this edition down myself, so we take Sammet's word for it.

3.5.11 Standardization of FORTRAN IV

FORTRAN IV was unique among the early versions of the language in that several teams outside IBM developed compilers for it. J. W. Backus and Heising noted that while a large number of computer manufacturers distributed compilers for FORTRAN IV with their hardware by 1963, a majority of the FORTRAN code in existence was probably still written in FORTRAN II(J. W. Backus and Heising, 1964).

A few companies wrote FORTRAN compilers in the early 1960s, thought not always under the name *FORTRAN*. In 1960, [TODO: who?] developed a compiler for an altered version of FORTRAN II called the ALTAC system for the Philco 2000 machine(Sammet, 1969)– this was the first FORTRAN compiler outside IBM.

In 1961, as detailed by Rosen, the team responsible for Philco 2000 successfully automatically ported an existing FORTRAN program from the IBM 704 to their Philco 2000 machine, and claimed the performance improvements were entirely consistent with the differences between the two machines: "there is an improvement in speed which is consistent with the extent to which the 2000 is faster than the 704"(Rosen, 1961). Now, roughly a year earlier, a single COBOL program ran on two different machines as well, as discussed in section 3.3.8, but the purpose in that case was to demonstrate the portability of the COBOL language and not necessarily to facilitate the needs of a programmer with a real existing program–the work with ALTAC as described by Rosen appears to be the first such case.

The first one *to call itself a FORTRAN compiler* was for FORTRAN I on the UNIVAC Solid State 80, which was running by January 1961 (Sammet, 1969), and later that year Remington Rand developed a compiler for a slightly altered FORTRAN II.

In the Aug. 1964 edition of *Datamation*, Oswald constructed a nearly-exhaustive table comparing the features and (mis)behaviors of 16 different FORTRAN compilers of the 43 he was aware of at the time. He also provided readers with some of the potential reasons for the differences between the compilers (Oswald, 1964):

> One is tempted to ask: Why these differences? There is no single cause. Certainly the basic hardware differences make their contribution. Some compilers differ from others because of misinterpretation or a desire to improve upon what exists.
>
> Noncompatibility is also a potential sales tool. Given a class of machines, one competitor can write language specifications that would permit the proper compilation of programs from the competing machines but effectively prevent reverse compatibility. This can be done by adding one or more features that the competitors do not have...
>
> The "universal" programming language, FORTRAN, is indeed universal but not all dialects are the same. We have presented some of the differences in the language and their effect on compatibility of FORTRAN programs. To those engaged in writing codes intended to be compatible with another machine now or in the future, caution is suggested. Such "compatible" programming requires great care and foresight.

As Oswald pointed out, the value of a new FORTRAN compiler was that the different compiler would allow organizations to buy that company's computers if they were used to using IBM equipment (and most were). So, if the competing company provided a compiler capable of handling all their customers' existing code, they could take IBM's customers. Once they had already taken IBM's customers however, the temptation would be to introduce new *features* that their customers would like and adopt, thereby locking them into their platforms.

The gamut of new compilers for the language prompted its standardization. If the language is handled by several different compilers with potentially divergent interests and customers, who is to determine the proper behavior of FORTRAN programs where the language's specification is ambiguous? Much of the value of the language is derived from its user's ability to take their programs and run them on another device rather than rewriting them–and rewriting them for a vendor's bespoke FORTRAN compiler is hardly better than rewriting them in a new instruction set for a new machine. Thus FORTRAN IV served as the basis of the first FORTRAN standards, which the American Standards Association (ASA) began developing in 1962 in the first FORTRAN standards committee, entitled the ASA X3.4.3.

The ASA's development of the first FORTRAN standard constituted the first standardization of a programming language by the ASA, and in the end *two* FORTRAN standards were produced: standard FORTRAN and standard Basic FORTRAN. These roughly corresponded to FORTRAN IV and FORTRAN II, though Basic FORTRAN *actually was* a perfect subset of FORTRAN, while IBM's actual FORTRAN IV compiler did not support a proper superset of their FORTRAN II compiler's notion of FORTRAN.

3.5.12 PL/1

PL/1 used in the development of the Amber OS (Frankston, 1984).

> PL/I was chosen as the programming language in 1964. Other possibilities were a port of MAD (the Michigan Algorithm Display) or a port of AED-0 (an MIT display). The full PL/I language was harder to implement than expected. A contract was awarded to an outside firm to produce a PL/I compiler, and BTL administered the contract. The contractor assigned two people and had produced no compiler a year later. Bob Morris and Doug Mcllroy (at BTL) created a back-up plan for a PL/I compiler, using McClure's TMG language on the 7094. This language was called EPL (Early PL/I). (Salus, 1994).

3.5.13 COMTRAN

3.6 ALGOL

3.6.1 The IAL and the ALGOrithmic Language

Before Hopper, Mayes and Phillips pulled together their committees for a common business language, design-by-committee was already in the academic milieu, especially in Europe. While not commercially successful, ALGOL introduced a number of important concepts like block scopes and the declaration of the types of variables, and it would go on to be the standard language for describing algorithms in academia. Originally called IAL (International Algebraic Language), it came to be called ALGOL, or ALGOrithmic Language, and was designed by an international committee with representatives from different organizations with the goal of a truly machine-independent language.

After deliberation in numerous committees, representatives from the German Association for Applied Mathematics and Machinery (GAMM) and the ACM met in Zurich, Switzerland in the summer of 1958. They had both produced similar reports and wanted to meet and agree on a unified language. John Backus, Charles Katz, Alan Perlis and Joseph Wegstein from the ACM attended this

meeting. [TODO: who were these people? Add some narrative. Maybe of Naur, then tie in with Backus.] They arrived at the following objectives (Perlis and Samelson, 1958):

1. The new language should be as close as possible to standard mathematical notation and be readable with little further explanation.

2. It should be possible to use it for the description of computing processes in publications.

3. The new language should be mechanically translatable into machine programs.

Shortly thereafter, a large number of dialects and partial implementations sprung up around Europe and the US such as BALGOL from Burroughs Corporation in Detroit, Michigan for the Burroughs 220. Manufacturers such as Burroughs found the standard to be insufficient for their users: "BAC-220 provides additions for the ALGOL reference language which are essential to the operation of data-processing systems: input-output facilities, conventions for inclusion of segments of machine-language coding, and diagnostic features" (Burroughs Corporation, 1963). This was intentional; the specifications of ALGOL (both the 1958 and the 1960 versions) was solely for the purposes of describing computation; no I/O or system libraries were specified. Other dialects included CLIP, JOVIAL, MAD, and NELIAC. The first issue of the *ALGOL Bulletin* was issued in March of 1959 out of Copenhagen with Peter Naur as the editor.

Jean Sammet describes the impact of ALGOL 58 (Sammet, 1969):

> Among the more intriguing technical features of ALGOL 58 were its essential simplicity; the introduction of the concept of three levels of language, namely a reference language, a publication language, and hardware representations; the ***begin...end*** delimiters for creating a single (compound) statement from simpler ones; the flexibility of the procedure declaration and the ***do*** statement for copying procedures with data name replacement allowed; and the provision for empty parameter positions in procedure declarations. While ALGOL 58 is not an exact subset of ALGOL 60, the only items of significance which are in the former but not the latter are the ***do*** which was removed as a concept (although the word was used for something else) and the empty parameter positions. Because of this major carry-over, specific technical description of ALGOL 58 is not necessary.

3.6.2 ALGOL 60

[TODO: backstory of Peter Naur]

The International Conference on Information Processing was held in Paris in June, 1959, where there were several key developments in ALGOL. Firstly, John Backus presented the now-famous paper on Backus-Naur Form for formal specification of programming languages, (John Warner Backus, 1959). Although other accounts describe this paper as having garnered significant attention in the IAL proceedings, Backus would recount that his paper "was received with a silence that made it seem that precise syntax description was an idea whose time had not yet come. As far as I know that paper had only one reader, Peter Naur" (John W. Backus, 1980).

Backus was involved in two programming languages at the time, FORTRAN and IAL, having already moved on from Speedcoding. Backus had been made aware of the logician Emil Post and her work on computability in a course given by Martin Davis, in particular, Post's notion of a "production." It was this concept that inspired Backus to write the 1959 paper, and Naur would go on to

extend it. This is part of Backus's specification of arithmetic expressions in the paper before Naur's changes(John Warner Backus, 1959):

$$\langle factor \rangle :\equiv \langle number \rangle \overline{\text{or}} \langle function \rangle \overline{\text{or}} \langle variable \rangle \overline{\text{or}} \ \ldots$$

Naur would make the notation more readable by replacing $\bar{or}$ with | and $:\equiv$ with $::=$ and by fully spelling out names that backus had abbreviated. Naur's edits and extensions would bring interest from more readers than Backus's original paper had. This form would come to be known as Backus-Naur Form, or BNF. Naur would submit this in a paper to the subsequent ALGOL meeting in 1960, titled "Report on the algorithmic language ALGOL 60"(J. W. Backus, F. L. Bauer, Green, Katz, J. McCarthy, Perlis, Rutishauser, Samelson, Vauquois, Wegstein, A. van Wijngaarden, Woodger, and Peter Naur, 1960). Various commitees had established some shortfalls of the original design of the AIL, and the 1960 meeting set out to address them with the help of Backus and Naur's specification. This report "represents the union of the Committee's concepts and the intersection of its agreements." The IAL had been renamed ALGOL 58 and then finally ALGOL 60 in Naur's report.

The final report specified expressions to have the following syntax:

$$\langle expression \rangle ::= \langle arithmetic\ expression \rangle | \langle Boolean\ expression \rangle | \langle designational\ expression \rangle$$

The attempts to formalize ALGOL 60 were varied; in (P. J. Landin, 1965), P. J. Landin attempted to formalize the semantics of ALGOL 60 by translating them into the λ^6 . The inclination to formalize the semantics of a programming language before working on a compiler or interpreter was emblematic of the British (and more broadly European) approach to programming language design.

3.6.3 Adoption of ALGOL

The *ACM Communications* started publishing an *Algorithms* section; ALGOL 58 was initially used, but ALGOL 60 was adopted once the updated report was in wide circulation. FORTRANwas not considered acceptable for publication until 1966, and even then it did not see widespread use in the journal. Because ALGOL had been designed from scratch by committee, there were no reference compilers for people to use; the only way to verify the correctness of an algorithm described in ALGOL was to rewrite it in another language and run test against test cases.

In early 1960, the *ACM ALGOL Maintenance Group* formed: an informal working group for the purpose of discussing the implementation of an ALGOL compiler. They discussed ambiguities in the specification and potential changes to the ALGOL 60 report. They mostly corresponded by mail and reports of their discussions were synthesized in the *ALGOL Bulletin*(Sammet, 1969). The 14 issue of the bulletin in 1962 contained a questionnaire from Peter Naur about some ambiguities in the 1960 report, and the philosophy of some proposed changes and enhancements to the language. [7] This questionnaire was used as a guide for another committee meeting in Rome in April of 1962 the primary result of this meeting is detailed in Naur's "Revised report on the algorithmic language ALGOL 60".

[6] This is more thoroughly discussed in section 3.7.3

[7] The reader may find it interesting that in this edition of the bulletin, Jean Sammet submitted a paper on *A Method of Combining ALGOL and COBOL*; [TODO: follow up on this?]

3.6.4 Peculiarities of ALGOL

The reader may find it interesting that ALGOL had a number of peculiar features. Namely, the distinction between *call by name* and *call by value*. Naur's 1962 revised report explains it best, I think(J. W. Backus, F. L. Bauer, Green, Katz, J. McCarthy, Perlis, Rutishauser, Samelson, Vauquois, Wegstein, A. van Wijngaarden, Woodger, and P. Naur, 1963, Section 4.7.3):

> **4.7.3.1 Value assignment (call by value)**
> of the corresponding actual parameters...the effect is as though an additional block embracing the procedure body were created in which these assignments were made to variables local to this fictitious block...
> **4.7.3.2 Name replacement (call by name)**
> Any formal parameter not quoted in the value list is replaced, throughout the procedure body, bu the corresponding actual parameter, after enclosing this latter in parentheses wherever syntactically possible.

While neither Naur's revised report nor Edgar Dijkstra's *Primer of Algol 60 Programming* point out the implications of this distinction, it is important to note the subtle implications. Call-by-name is not common in modern programming languages, and readers may find it more similar to macro expansion (though it is not, in fact, simply textual replacement). The following example illustrates the difference:

```
def get_5():
    print("Hello!")
    return 5

def double(x):
    return x + x

# Prints "Hello!" once with call-by-value
# and twice with call-by-name
double(get_5())
```

Most implementations of call-by-name use zero-argument thunks to delay the evaluation of arguments until they are used in the body. The arguments to function calls become zero-argument closure functions returning the value of the argument expression and capable of being *l-values*, or values in the left-hand side of an assignment expression. Then, each instance of the argument in the callee's body is replaced with an invocation of this function such that it is evaluated anew each time it appears. The prior Python example may be rewritten to emulate call-by-name as follows:

```
def get_5():
    print("Hello!")
    return 5

def double(x):
    # argument thunk is evaluated twice
```

```
    return x() + x()

double(lambda: get_5())
```

Call-by-value semantics demand that the argument to `double` be evaluated before entering the function, so `get_5` is called once and *the result* is passed to `double`. With call-by-name semantics, `get_5` is passed the expression itself, which is the substituted into the body of the callee *before* being evaluated.

Donald Knuth proposed the *man or boy* test in the *ALGOL Bulletin* to distinguish between conformant ALGOL compilers capable of handling recursion and non-local references using call-by-name semantics(Knuth, 1964).

> There are quite a few ALGOL60 translators in existence which have been designed to handle recursion and non-local references properly, and I thought perhaps a little test-program may be of value. Hence I have written the following simple routine, which may separate the man-compilers from the boy-compilers.
>
> This uses nothing known to be tricky or ambiguous. My question is: What should the answer be? Unfortunately, I don't have access to a "man-compiler" myself, asn do I was forced to try hand calculations. My conjecture (probably wrong) is that the answer will be
>
> $$73 - 119 - 177 + 102 = -121.$$
>
> I'd be very glad to know the right answer.

The program is as follows:

```
begin
  real procedure A(k, x1, x2, x3, x4, x5);
  value k; integer k;
  real x1, x2, x3, x4, x5;
  begin
    real procedure B;
    begin k := k - 1;
          B := A := A(k, B, x1, x2, x3, x4)
    end;
    if k ≤ 0 then A := x4 + x5 else B
  end;
  outreal(1, A(10, 1, -1, -1, 1, 0))
end
```

There are some useful applications of call-by-name semantics as well; it is not *solely* an avenue for confusion. Jensen's Device, named in honor of Mr. J. Jensen of Regnecentralen, Copenhagen, is a well-known application of call-by-name semantics. This allows the user of a function to describe an expression to be evaluated in a loop body inside the function, permitting a very general summation function. There is not an especially elegant way to do this in a call-by-value language, but as Dijkstra points out(Dijkstra, 1961), it can be implemented with this ALGOL program:

```
begin
    real procedure Sum(k, l, u, ak)
        value l, u;
        integer k, l, u;
        real ak;
    begin
        real s;
        s := 0;
        for k := l step 1 until u do
            s := s + ak;
        Sum := s
    end;
end
```

To simply sum the values of the array `V` from 1 to 100, one would call: `Sum(i, 1, 100, V[i])`. The real power of this device is its generality; if a user were to need to compute $\sum_{i=1}^{100} V_i \times i$, they could call `Sum(i, 1, 100, V[i]*i)`. The first and final arguments to this function are not evaluated until they are used in the body of the loop in `Sum`. Others have also pointed out numerous easy-to-make mistakes that arise because of this feature. For example, a naive implementation of a swap function may easily cause unintended side effects:

```
real procedure Swap(a, b)
    real a, b;
begin
    integer temp;
    temp := a;
    a := b;
    b := temp;
end;
```

While calling `Swap(i, j)` may work correctly, calling `Swap(i+1, j+1)` or `Swap(A[i], A[j])` would not.

Among the now-unusual argument evaluation semantics, ALGOL also lacked any I/O facilities as we have already discussed. The language also made string variables available, but provided no means of manipulating them. ALGOL permitted *own variables*, which are akin to static variables in C or SAVE variables in FORTRAN; they retain their values between entrances to the block. Dijkstra also points out nowhere is the evaluation order of subexpressions specified (Dijkstra, 1961).

While the declared intent of defining a language detached from any particular manufacturer or machine was laudable (and profitable in the case of portable languages to follow), ALGOL was so far removed from the machine to be useless without supplementary libraries (for I/O, string processing, etc).

3.6.5 ALGOL 68 (and X, Y, W, ...)

For years after the 1962 Rome meeting, there were discussions amongst the ALGOL community about potential changes and improvements to the language. The 1960 and 1962 meetings restricted themselves to clarifying ambiguities and rectifying true errors in the prior reports, but not extensions or significant changes. In the March of 1964 as follow up to Rome meeting, they finally agreed to start on ALGOL X and ALGOL Y, drafts of the language that would consider significant changes and additions to the language. ALGOL X would be intended to solve the issues of ALGOL 60 in the short-term, compared to the "radically reconstructed future ALGOL Y" (Duncan and A. van Wijngaarden, 1964).

To summarize the following section: three draft reports were proposed, Adriaan Van Wijngaarden was asked to revise his draft report until the other two authors were satisfied, the draft report that all three were satisfied with became canonized as ALGOL 68, many people withdrew from the language community (formally or informally) in reaction to the new language, and the language was never widely adopted.

1964:	**ALGOL X**
1965–1969:	**Development** 1965: Wirth, Seegmüller and Van Wijngaarden 1966: Van Wijngaarden commissioned to make the Report 1967: Draft [MR88] 1968: Drafts [MR93] and [MR95]. Accepted Report [MR100] 1969: Final Report [MR101]
1970–1974:	**Revision** 1970: First Implementation: ALGOL68-R 1974: Revised Report [RR]
1975–1979:	**Active Support** 1976: Standard Hardware Representation, ALGOL 68S 1978: Modules and Separable compilation

Table 3.1: Timeline of ALGOL 68 development, based on(Marchesi, 2025).

In 1965, the first three drafts for ALGOL X were proposed by Niklaus Wirth (who benefitted from numerous comments from Tony Hoare), Gerhard Seegmüller, and Adriaan van Wijngaarden. Seegmüller's proposal was not too different from Wirth and Hoare's, which was called ALGOL W, but Wijngaarden's proposal was dramatically different. Wijngaarden described his ALGOL X language in an entirely new formal grammar which came to be known as a *W-Grammar* (after his own namesake).

The impact of Wijngaarden's proposal on the development of ALGOL cannot be overstated. In the spring of 1968, Adriaan van Wijngaarden released a draft report for ALGOL X (which was, mind you, supposed to be the *incremental* improvement to ALGOL 60) based on Wirth and Hoare's ALGOL W but using his own W-Grammar. In this time period, Peter Naur and Brian Randall of IBM even suggested that the incremental ALGOL X be dropped entirely in favor of the more ambitious ALGOL Y. At nearly every step, the committee ran significantly past their schedules: "Throughout the whole

project, the WG in general, and Van Wijngaarden in particular, consistently underestimated the time it would take by substantial factors. Recall that ALGOL 60 was put together in six days"(Lindsey, 1993a).

Wijngaarden's ALGOL W proposal was distinct for at least three reasons:

1. A two-level *W Grammar*.
2. "The combination of a minimal number of language concepts in an orthogonal way" (Lindsey, 1993a).
3. Expression orientation; everything is an expression and has a value.

The computer and aerospace historian Paul Ceruzzi describes the result of the ALGOL 68 report as follows:

> Whereas ALGOL-60 was based on a formal structure and was very lean, ALGOL-68 was burdened by an attempt to do too much, with the effects that some features interfered with the clean implementation of others. It was hard to understand. In an attempt to satisfy a broad range of users worldwide, the committee produced something that satisfied few.

Charles Lindsey, British computer scientist and editor of (A. van Wijngaarden, B. J. Mailloux, J. E. L. Peck, C. H. A. Koster, et al., 1976) (and numerous other texts on ALGOL 68) describes popular sentiment about the report(Lindsey, 1993a):

> The world seems to have a rather negative perception of ALGOL 68. The language has been said to be "too big," to be defined by an "unreadable Report" produced by a committee which "broke up in disarray," to have no implementations, and to have no users.

This was not his opinion, however(Lindsey, 1993a):

> I should point out that my own involvement with the project came after the basic design of the language, and of its original Report, were complete…It is only now, in the course of studying the minutes and other documents from that time, that I have come to see what the real fuss was about, and I hope that all this has enabled me to take a dispassionate view of the events. The reader of this paper will certainly see discord, but I believe he will see also how good design can win through in the end.

Lindsey was appointed Lecturer in Computer Science at Manchester University in 1967. Aside from his standardization efforts, he wrote a research implementations of ALGOL 68 for the MU5, one of the Manchester computers, and he maintained an implementation of a subset of ALGOL 68, *ALGOL 68S*. It is difficult to take Lindsey's opinions about ALGOL 68 entirely at face value because so much of his career was dedicated to the language. If we look at those involved with the design of ALGOL 68 who had programming language design and compiler experience elsewhere, we find his depiction above to be more accurate than not.

Another meeting in May 1967 in Zandvoort, Netherlands was intended to develop a direction for ALGOL Y, however nearly all the discussion was about ALGOL X. From the outset, the only language aspect the designers knew they wanted from ALGOL Y was the ability for an ALGOL Y program to modify itself. Lindsey remarks:

> They even spent an afternoon on ALGOL Y, from which it emerged that no one had very concrete ideas of what it was about, and that the only role model was LISP, on account of the fact that that language could construct its own programs and then *eval* them.

The largest single outpouring of criticism of ALGOL 68 came from (A. van Wijngaarden, B. J. Mailloux, J. E. L. Peck, and C. H. A. Koster, 1968), also known as [MR93]. Several members of the committee (including Peter Naur) resigned after this report's circulation. It was first circulated to the *ALGOL Bulletin* in February 1968, and "was the cause of much shock, horror, and dissent, even (perhaps especially) among the membership of WG 2.1"(Lindsey, 1993a). Numerous papers were written in response to MR93, including Peter Naur's *scathing* critique "Successes and failures of the ALGOL effort". His report was especially critical of the International Federation for Information Processing (IFIP), which had formed a working-group (WG 2.1) responsible for specifying the language.

Also in response to MR93, Lindsey joined the language effort and distributed his (Lindsey, 1968) in which he attempted to synthesize the nicer language hidden within the earlier reports [8]. He clarified a number of terms which appear to be specific to ALGOL, and have well-understood meanings under different names in other languages. For example, what ALGOL calls a *name* is more commonly known as a reference or pointer; numerous differences in nomenclature contributed to the difficulty of reading the original reports.

Van Wijngaarden had circulated another report(B. Mailloux, J. Peck, and C. Koster, 1968) in October of 1968 as he had earlier promised the committee; at the same time, other members who disagreed with the direction but had not resigned were also developing several minority reports, include Dijkstra, Hoare, and Randell. Sparing the reader the full details and political history of the ensuing committee meetings, the final report "Report on the Algorithmic Language ALGOL 68" was published in *Numerische Mathematik* and *Kybernetika* in 1969, thus specifying what we know today to be ALGOL 68.

3.6.6 The W-Grammar

Before analyzing the technical details of ALGOL 68, we must first understand what made it so offensive and uninterpretable to so many people: its grammar.

The primary reason for this grammar's complexity is that it encompasses far more than compiler engineers typically associate with a grammar in order to be *formally context-sensitive*. It is not merely a specification of the syntax of the language; it also specifies a great deal of the *semantics* of the language, because that is required for the language to be context-sensitive. When the grammar is described as *two-level*, it communicates that the grammar consists of two levels: the meta-grammar and the concrete grammar, and the meta-grammar produces the concrete grammar. Readers of [MR93] would need to keep track of both levels in order to understand the document; meta-rules would need to be kept conceptually distinct in the reader's mind. The *format* of the grammar was also problematic and difficult to read, but that may not have been so problematic had they not fit so many semantic concepts into the grammar.

When Wijngaarden first proposed his grammar, nobody had seen anything like it. The authors attempted to formalize object lifetimes, variable scoping rules, and relatively complicated type systems (even by today's standards), all without the benefit of decades of research into type theory and

[8] For those unaware, this title is an homage to the English textbook *French without Tears*.

formal semantics which we have today. Let the reader give the authors the benefit of the doubt; keep in mind they attempted to formalize concepts for the first time, and subsequent research benefitted greatly from their ideas (and mistakes). The language specification was, effectively, a compiler front-end specification.

Lindsey argues that the grammar is relatively close to Prolog, where goals and sub-goals are specified and solved for by unification [9].

Lindsey uses the following Prolog example to introduce a rule in ALGOL 68's W-grammar, which asks whether the input program contains an assignation, and to determine this, we must test if we have a destination, followed by some language construct that becomes a symbol, followed by a source.

```
assignation(ref(MODE)) :-
      destination(ref(MODE)), becomes_symbol, source(MODE).
```

The corresponding rule in the [MR93] draft report is as follows (Van Wijngaarden et al., 1968):

```
MODE assignation :
    reference to MODE destination, becomes symbol, MODE source.
```

Substituting ***int*** for our ***MODE***, we have:

```
int assignation :
    reference to int destination, becomes symbol, int source.
```

If you squint your eyes, maybe you can see how *i := 5* fits the mold for this grammar, where the ***MODE*** is ***int***, the ***destination*** is the location of ***i*** of mode ***ref int***, and the source is the integer literal ***5*** of ***MODE int***. This may feel familiar, like a looser version of the Backus-Naur syntax used by compiler-compilers. The problem is, nearly anything else is also expressible in this grammar, making it nearly impossible to write a parser based on an unrestricted W-grammar.

A more nebulous component of the original grammar from [MR93] can be found in *Section 4.4 Context Conditions*, which specified conditions for a program to be *semantically valid* based on the surrounding context. Modern readers may find this strange; we are discussing the language *syntax*, why is the report discussing the semantic validity of, for example, the use of a particular identifier and its relationship to its declaration? As we have already established, this is an example of language *semantics* being specified in the grammar.

Most modern compilers are broken into three parts: the front-end, middle-end [10], and back-end. The front-end typically consumes the input program, produces data structures representing the program, and performs *semantic* analysis on those data structures to ensure the program is valid. The consumption of the input program is usually determined by the grammar specified by the language, and semantic validity is determined by the language's type system and other rules, but not typically by the grammar. Because Wijngaarden attempted to specify all of this in the grammar itself, the grammar needed to specify these semantic conditions as well.

See this excerpt on ***NEST***s from (A. van Wijngaarden, B. J. Mailloux, J. E. L. Peck, C. H. A. Koster, et al., 1976, Section 3.0.2, Semantics):

[9] The term *unification* may be loaded; Milner did not publish his work on SML and type inference until later, though his work was inspired by ALGOL 68 to some degree.

[10] Only compiler engineers could produce a term as idiosyncratic as *"middle" "end"*.

> A "nest" is a 'NEST'. The nest "of" a construct is the "NEST" enveloped by the original of that construct, but not by any 'defining LAYER' contained in that original.
>
> The nest of a construct carries a record of all the declarations forming the environment in which that construct is to be interpreted.
>
> Those constructs which are contained in a range R, but not in any smaller range contained within R, may be said to comprise a "reach". All constructs in a given reach have the same nest, which is that of the immediately surrounding reach with the addition of one extra "LAYER". The syntax ensures (3.2.1.b, 3.4.1.i,j,k, 3.5.1.e, 5.4.1.1.b) that each 'PROP' (4.8.1.E) or "property" in the extra 'LAYER' is matched by a defining. indicator (4.8.1.a) contained in a definition in that reach.

Today, ***NEST***s might exist concretely as data structures inside the semantic analysis phase of a compiler, but in ALGOL 68, they were specified as part of the grammar itself.

ALGOL 68's type system was also remarkably complex and, again, fully embedded in the grammar. Union modes are full commutative and accumalative types, meaning that the following types are equivalent: ***union(int, real, bool)***, ***union(real, int, bool)***, and ***union(bool, union(real, int))***.

Lindsey pointed out a particularly tricky aspect of the type system in structural equivalence. Structure types are equivalent if their members are equivalent, as opposed to name equivalence, which would have implied two structures with the same members but different names are not equivalent. The following two types are equivalent by structural comparison:

```
mode a = struct (int val,
                 ref a next);
mode b = struct (int val,
                 ref struct (int val, ref b next) next);
```

3.6.7 Concepts of ALGOL 68

Having covered the political machinery that resulted in the final report and the grammar that caused the community so much grief, we will now look to more technical details of the language.

Firstly, ALGOL 68 is an expression-oriented language; there is fundamentally no distinction between statements and expressions, allowing for constructs like the following:

```
x := (real a = p*q;
      real b = p/q;
      if a>b then a else b fi)
      + (y := 2*z);
```

One of the key goals of the committee was to design language features that would be *orthogonal*–M. L. Scott put it well in (M. L. Scott, 2009):

> One of the principal design goals of Algol 68 was to make the various features of the language as orthogonal as possible. Orthogonality means that features can be used in any combination, the combinations all make sense, and the meaning of a given feature is consistent, regardless of the other features with which it is combined. The name is meant to draw an explicit analogy to orthogonal vectors in linear algebra: none of the

> vectors in an orthogonal set depends on (or can be expressed in terms of) the others, and all are needed in order to describe the vector space as a whole.

There were numerous extensions to arrays, including higher-dimensional arrays, complicated slicing mechanisms, and flexible arrays, such as the following:

```
loc[1:4, 1:5] int a45;
a45[2  ,    ] # row 2 #
a45[   , 3  ] # column 3 #
a45[2:3, 3  ] # part of column 3 #
a45[2:3, 2:4] # a little square in the middle.  #

loc flex [1:0] int array; # initially empty #
array := (1, 2, 3);       # now it has bounds [1:3] #
array := (4, 5, 6, 7, 8); # now it has bounds [1:5] #
```

Call-by-name as it was known in ALGOL 60 was removed in favor of call-by-value and call-by-reference; some of the other proposals (such as Seegmüller's) preserved the two cases of call-by-name under different terms, but this was not adopted. One may argue that *proceduring* is a form of call-by-name, but this feature was removed in the revised report of 1973.

Call-by-name is really two different concepts: call-by-reference (where the actual parameter is a named variable to be assigned to) *or* call-by-full-name (where the actual parameter is an expression to be placed in a thunk and re-evaluated each time it is used in the body of the callee). This meant that Jensen's Device and other tricks made possible by call-by-name were no longer possible in ALGOL 68. Wirth's proposal included call-by-name untouched from ALGOL 60 alongside a *new* parameter passing mechanism called *parameterless* ***procedure*** parameters, meaning the parameters were thunks to be evaluated each time they were used in the body of the callee(Lindsey, 1993a). Lindsey uses an inner-product algorithm to illustrate this:

```
begin
    procedure innerproduct
        (real a, real procedure b,
         integer value k, integer p, real result y);
    begin y := 0; p := 1;
        while p<=k do begin y := y + a * b; p := p + 1 end
    end;
    real array [1:n] x1, y1; integer j; real z;
    innerproduct(x1[j], y1[j], n, j, z);
end
```

He notes that the first and second parameters are effectively both called by-name. As far as I can tell, the only difference is that call-by-name parameters can be l-values or r-values while parameterless procedures are always r-values.

There were at least eight changes to the automatic type conversion rules between ALGOL 60 and ALGOL 68; Lindsey had this remark about the automatic type conversion features:

> Although coercions had existed in previous programming languages, it was ALGOL 68 that introduced the term and endeavoured to make them a systematic feature (although it is often accused of a huge overkill in this regard).

Many of these coercion rules were uncontroversial; for example, a ***real*** may be assigned to a ***union(real, int)***. [TODO: Discuss *rowing*? relates to rank polymorphism a bit...] One particularly problematic coercion rule was *proceduring*. This allowed users to force call-by-name semantics by coercing an expression into a parameterless procedure, like so:

```
      PROC x plus 1 = INT : x + 1;
```

The right hand side of this expression is a *cast* (a term originally coined for the specification of ALGOL) of an integer expression into a procedure taking no arguments. In "ALGOL68 with fewer tears" [11], Lindsey argues that references obviate the need for call-by-name semantics, but that call-by-substitution is what ALGOL 60's call-by-name effectively was, and this was still possible with ALGOL 68 via this *proceduring* coercion:

```
proc series = (int k, ref int i, proc real term) real expr
    begin
    real sum(0);
    for j to k do
        begin i := j;
              sum plus term
        end;
    sum
    end

x := series (100, i, real expr(1/i));

# Or, via proceduring: #
x := series (100, i, 1/i);
```

This facility was abandoned in the ALGOL 68-R implementation which led to its removal in the revised report of 1973, as we will see in the next section. There were still other mechanisms to achieve similar results, as tricks like Jensen's Device had become important to the community. One such example is the extensions to procedures.

Naur and Wirth both proposed block-expressions, meaning the final value in a block is the value of the entire block in the context of an expression, and Naur's 1966 proposal (which was accepted) merged the formal parameters into the same line as the parameters themselves, meaning one need not restate the types of parameters at the beginning of the procedure body. These two features allowed for convenient use of anonymous procedures capturing parts of their surrounding context and procedures taking other procedures, relatively advanced features even for today. The revised report contains this example (A. van Wijngaarden, B. J. Mailloux, J. E. L. Peck, C. H. A. Koster, et al., 1976, Section 11.2, Innerproduct 1):

```
proc innerproduct 1 = (int n, proc (int) real x, y) real:
# the innerproduct of two vectors, each with 'n'
components, x (i), y (i), i = 1, ... , n, where 'x' and 'y'
```

[11] This paper was, in fact, a valid ALGOL 68 program in and of itself

```
are arbitrary mappings from integer to real number #
begin long real s := long 0;
    for i to n do
        s +:= leng x (i) \times leng y (i)
    od;
    shorten s
end

# Real-calls using innerproduct 1: #
innerproduct 1 (m, (int j) real: x1[j], (int j) real: y1[i])
innerproduct 1 (n, nsin, ncos)
```

ALGOL 60 was the first programming language to permit nested functions, a feature ALGOL 68 retained. This meant that nested procedures had to be capable of capturing their environments, dramatically complicating the work of the compiler engineer (as did many of the language's features). The compiler had to somehow pass the captured environment along with the procedure so it had access to the state of the outer scope. In C++ for example, lambda functions are really classes which capture the enclosing scope as a member variable - no such facility was specified in the language standard, so compiler authors were left to sort it out themselves.

3.6.8 ALGOL 68-R and the Revised Report

In 1973, the *Revised Report on the Algorithmic Language ALGOL 68* was published, codifying some of the changes and rejected features from the ALGOL 68-R team.

[TODO: look at peck's (John E. Peck, 1978)]

His work on ALGOL 68S was not an outlier amongst would-be implementers of the language; because the report was so large and complex, most implementers ended up restricting themselves to a subset of the language.

3.6.9 Legacy of ALGOL

It is difficult to overstate the impact of the design decisions of both ALGOL 60 and 68, even though neither saw widespread adoption in the industry outside depictions of algorithms in academic papers. When Dennis Ritchie extended Ken Thompson's B compiler with a type system, he drew heavy inspiration from ALGOL 68:

> The scheme of type composition adopted by C owes considerable debt to Algol 68, although it did not, perhaps, emerge in a form that Algol's adherents would approve of. The central notion I captured from Algol was a type structure based on atomic types (including structures), composed into arrays, pointers (references), and functions (procedures). Algol 68's concept of unions and casts also had an influence that appeared later. (Ritchie, 1996)

Lindsey points out a few others:

> The type system of ALGOL 68 has been adopted, more or less faithfully, in many subsequent languages. In particular, the structs, the unions, the pointer types, and the parameter passing of C were influenced by ALGOL 68 [Ritchie 1993], although the syntactic sugar is bizarre and C is not so strongly typed. Another language with a related type system is SML [Milner 1990], particularly with regard to its use of ref types as its means of realizing variables, and C++ has also benefitted from the reftypes [Stroustrup 1996]. "A history of ALGOL 68"

ALGOL 68 also had a notable influence in the Soviet Union, details of which can be found in Andrey Terekhov's 2014 paper (Terekhov, 2014).

[TODO: Pascal, Ada] [TODO: Lindsey: "So here are my recommendations to people who essay to design programming languages." (Lindsey, 1993a)]

The influence of ALGOL was so wide that it is hard to point to compilers or programming languages that are *not* heavily influenced by it. A few languages stand out as exceptions because their authors were involved in the design of ALGOL and went on to develop new languages in light of those. Among others, the most notable were Ada, Pascal, and SIMULA. We will treat these more thoroughly in the following chapter, but we briefly introduce their connections to ALGOL here.

3.6.9.1 Pascal

Nicolas Writh wrote the language based on ALGOL 60. He was heavily involved in the development of ALGOL X, or ALGOL 68 as it came to be known. His proposal for ALGOL X was not taken up by the committee after they determined that [TODO: what's his name again?] would move forward with his W-grammar, and the ALGOL 68 process divulged into complexity and political back-and-forths. Wirth picked up his proposal and continued developing it until it became Pascal (Wirth, 2021). See section 4.4 for the full discussion.

3.6.9.2 SIMULA

SIMULA is unique among programming languages: it was developed as a direct extension of ALGOL 60. The *SIMU*lation *LA*nguage was developed by [TODO: who?] for the UNIVAC 1107 in early 1965 by tacking on features related to parallelism. "50 years of Pascal".

[TODO: Defined in Wijngaarden Grammar by Adriaan van Wijngaarden. Contains parsing and things which in other langauges are called semantics.]

[TODO: '68 critcized by Hoare and Dijkstra for abandoning simplicity of '60. In 1970, ALGOL 68-R became the first working compiler for ALGOL 68.]

3.7 The λ-Calculus

A surprising number of the foundations for programming language theory were laid before many familiar programming concepts were developed. Higher-order logic, combinatory logic, and λ-calculus were all being developed prior to their uses in programming languages with the aim of providing a firm foundation for *mathematics*, and would only cross-polinate with programming languages later on.

Let me digress for a moment before we begin. Students of computer science often feel a sense of bewilderment when they are first introduced to many foundational concepts in a way that is unique

to the field [12]. I believe the reason for this is that the field is directly built upon information theory and higher-order logic, fields which were only recently developed in the history of mathematics, and contain some unsettling and bewildering concepts.

In the early 20th, Russell's paradox [13] was recently developed and Gödel's incompleteness theorems [14] were not. For those unfamiliar, they call into question the validity of the foundations of mathematics and point out the limits of provability. These led to several efforts to establish a firm foundation for the entire field of mathematics, some of which were based on sets and other functions and the two that we concern ourselves with here are *combinatory logic* (abbreviated *CL* from here) and λ(occasionally abbreviated λ), which were attempts to use functions. They are closely related and share many concepts. These efforts directly led to functional programming and type theory, and underpin many other concepts in programming languages and compilers.

I am no expert in this field and it does not entirely concern the history of compilers, so we will digress only insofar as it informs our topics of focus.

3.7.1 Combinatory Logic and the λ-Calculus

(Cardone and J Roger Hindley, 2006) summarizes the timeline of the development of λand CL as follows:

> the history of λ and CL splits into three main periods: first, several years of intensive and very fruitful study in the 1920s and '30s; next, a middle period of nearly 30 years of relative quiet; then in the late 1960s an upsurge of activity stimulated by developments in higher-order function theory, by connections with programming languages, and by new technical discoveries.

In this section, we discuss primarily the first and second periods, and pick up with the *applications* to programming languages and compilers in section 4.6.

Both the λ and CL were invented in the 1920s– CL first by the Ukrainian mathematician Moses Schönfinkel at the University of Göttingen (likely the most advanced mathematics institution at the time). He introduced the concepts informally in a talk in 1920, and the ideas were first published in (Schönfinkel, 1967). These ideas would later be discovered by John von Neumann in (Neumann, 1925) and then re-invented by Haskell Curry in 1926-1927. After Curry found Schönfinkel's work, he set out to get a PhD at the University of Göttingen researching combinators, and his doctoral thesis was published in (Curry, 1930). Just prior to Curry's thesis, Alonzo Church invented the λ-calculus in roughly 1928, and first published this work in (Church, 1932).

In broad strokes, all these efforts were attempts to use functions to build a foundation for mathematics. In the λ, Church sought to build such a foundation out of functions, with $\lambda x.E$ meaning *the function that takes argument x and yields the expression E*, and Fx meaning *the function F applied to argument x*. From functions, Church built up the concept of numbers, similar to the way numbers are formed by nesting sets in set-theoretical foundations for math. In his second paper on the subject (Church, 1933), Church formulates the natural numbers as functions with N levels of nesting.

[12] You will have to take my word on this. I provide no evidence, but I believe readers will come to agree.

[13] Essentially, set-theory constructions of mathematics lead to contradictions.

[14] Essentially, every formal logical system contains true statements that it is unable to prove.

Various components of the λwere developed independently, including Alan Turing in (Turing et al., 1936). Turing would go on to take a doctorate under Church, completing in 1938. The applications to compilers were not fully developed; John McCarthy would adopt a small number of concepts from the λin the 1950s, the discusssion of which is continued in section 3.7.2.

3.7.2 Lisp

Lisp was developed primarily by John McCarthy in two bouts: most of the key ideas were developed in 1956-1958, and in 1958-1962 the language was actually implemented and applied to artificial intelligence.

In terms of programming language design, Lisp's fundamental data structure is the *list*– hence the name (***LIS**t **P**rocessing*). Symbolic computing (which was still novel at the time) was implemented using lists and basic operations on them. Lisp *programs* were also represented as data; one of the key innovations of the language the ability manipulate Lisp programs as any other data structure, with the `eval` function serving as the bridge between the code as data and code as actions to be performed by a Lisp program. This concept of self-modifying programs was to serve as the basis for ALGOL Y, the more ambitious successor to ALGOL 60, which was abandoned due to the eventual scope and controversy of the ALGOL X project, or ALGOL 68 as it came to be known.

3.7.2.1 FORTRAN Lisp Processing Language

Lisp was born out of John McCarthy's desire for a list-oriented language for work on an IBM 704 at Dartmouth College during a summer research project in 1956, which was the first organized study of artificial intelligence (John McCarthy, 1978). Around this time, McCarthy was presented the list-processing programming language *IPL 2* (*Information Processing Language*), written for the RAND Corporation's JOHNNIAC computer. [15] Dartmouth was soon to get access to an IBM 704 thanks to the New England Computation Center at MIT, which IBM was in the midst of establishing. McCarthy was to consult with a team at IBM developing a theorem-proving program for plane geometry, and it was not clear at the time whether IBM's FORTRAN would be suitable for list-processing.

McCarthy was also independently working on artificial intelligence, publishing his first paper in the field in 1958, "Programs with common sense". This paper, also called the *Advice Taker* proposal, involved representing information in the form of sentances in a formal language, and an accompanying program that would make inferences based on that information. These sentences were to be structured as lists, so he naturally needed a list-processing language to process these sentences.

McCarthy started by considering how list structures would be represented in memory. The IBM 704 had an addressable word size of 36 bits with a 15-bit address space, so the pointers would need to be 15 bits, which allowed for two pointers in each word, plus 6 extra bits. A list in Lisp was to be represented in a word like so [16]:

Nathaniel Rochester, Herbert Gelernter and Carl Gerberich at IBM took on the task of writing implementing this list-processing language in FORTRAN, called FLPL for *FORTRAN-Compiled List Processing Language*. They considered using the IPL, but decided against it, persuaded by McCarthy's

[15] The RAND Corporation was not the same company as Remington Rand/Sperry Rand, where Grace Hopper worked around this time.

[16] This is the layout as McCarthy describes in "History of LISP", but Gelernter, Hansen, and Gerberich include a sign bit as the uppermost bit of the tag.

	Tag	Decrement	Prefix	Address
Width	3 Bits	15 Bits	3 Bits	15 Bits
Purpose	Type Code	Address of Rest of List	Type	Data/Pointer to Data
Lisp primitive	`ctr`	`cdr`	`cpr`	`car`

Table 3.2: Layout of a 36-bit word on the IBM 704 as used for Lisp list structures.

suggestion to instead adapt a FORTRAN compiler. This primitive version of the language lacked conditional expressions, recursion, and other fundamental features.

They note in "A Fortran-Compiled List-Processing Language" that while IPL's interpretation was prohibatively costly, their adaptation of FORTRAN was able to run the same programs in a fraction of the time. They also pointed out the utility of FORTRAN III's introduction of separable compilation and linking of hand-written machine code and regular FORTRAN code, which allowed them to hand-write certain performance-critical sections of their list-processing programming environment(Gelernter, Hansen, and Gerberich, 1960):

> It must be emphasized that FORTRAN is in itself an information processing language of great versatility and sophistication. Our list-processing functions merely serve to increase the "vocabulary" of the language so that list manipulation processes may be described within the FORTRAN framework as are ordinary computer processes. We are thus able to enjoy the same ease of programming, ease of modification, and estensive debugging aids available to the programmers of standard numerical problems.

Thus FLPL was not necessarily a compiler or interpreter, but a set of libraries and tools that made list-processing within FORTRAN programming environment easier. The numerical features of FORTRAN and the runtime libraries for I/O and formatting were available in FLPL, but the programming environment also made available operations to evaluate symbolic expressions and manipulate lists. Notably, FLPL lacked the ability to treat machine code as data, a feature that IPL had and Lisp would come to be famous for. Gelernter, Hansen, and Gerberich conclude their paper by pointing this out:

> One feature of IPL V is excluded from FLPL by the nature of a compiler. Sequences of IPL instructions to be interpreted are stored in the computer as NSS lists, just as are the data. Although this property has been largely irrelevant to all programs written to date, it is conceivable that one might want to write a program in which the symbolic entities that are processed are IPL instructions themselves, and in which transfers of control take place between the metaprogram and the machine-generated one. The fact that the transformation of FLPL expressions into computer activity is a two-stage, irreversible process places this kind of behavior beyond the range of our language, even though it is quite feasible to manipulate FLPL expressions within FLPL.

3.7.2.2 McCarthy's Contributions

Nathaniel Rochester invited McCarthy to join the IBM Information Research Department for the summer of 1958 to implement differentiation of algebraic expressions in FLPL, where he would go on to develop many more foundational concepts. In fact, the differentiation program would never be completed due to limitations in FLPL, and his time there was demarcated by pushing the boundaries of what was possible in the language.

McCarthy developed the conditional expressions in 1957-1958 while developing a chess program in FORTRAN, though it was not the conventional notion of an if-statement since both arms of the conditional expression were always evaluated; `XIF(C, E1, E2)` would return `E1` if `C` was equal to 1 and `E2` otherwise, which we would now call a *merge* or *select* operation. This was motivated by the clumsy syntax and semantics of the `IF` statement in FORTRAN I and II, and something similar to McCarthy's version would soon be added to the language.

Because the process of differentiation is inherently recursive, McCarthy naturally found the need for recursion in FLPL, and went about implementing it. For differentiating an arbitrary list of symbol expressions, he added `maplist` to apply a function to each element of a list, the first his many steps towards higher-order programming. To pass functions as arguments to `maplist` and other higher-order functions, he would need a notation for functions, which he introduced with `lambda` (John McCarthy, 1978):

> use functions as arguments, one needs a notation for functions, and it seemed natural to use the λ-notation of Church (1941). I didn't understand the rest of his book, so I wasn't tempted to try to implement his more general mechanism for defining functions. Church used higher order functionals instead of using conditional expressions. Conditional expressions are much more readily implemented on computers.

Again, we see how far Alonzo Church's λ-calculuswas ahead of the actual implementation of programming languages capable of representing its concepts. McCarthy also ran into the problem of deallocation of these list structures. While list-processing made some problems far more convenient, it also introduced new problems and some old problems more difficult. Memory management was one of the latter.

Techniques for automatic memory management had not yet been developed, and McCarthy would not solve this problem before his summer at IBM came to an end. The teams at IBM were quite pleased with the directions they were taking FORTRAN and FLPL, dismissing recursive functions and proper conditional expressions as not necessary. The culmination of these limitations led McCarthy to conclude that a new language was needed. FORTRAN could not easily be extended because of both the complexity of the language and existing compilers, and because of the political difficulties that would arise from suggesting to IBM leadership that their wildly successful compiler and programming language would need to be significantly changed. Thus he began working on Lisp.

3.7.2.3 Lisp is Born

The Fall of the same year, in 1958, McCarthy became an assistant professor at MIT, where he started the MIT Artificial Intelligence Project with Marvin Minsky. Their original plan was to produce a compiler, but it was in the zeitgeist that compilers were extremely time intensive to develop; John Backus's team noted in "The FORTRAN automatic coding system" that it took them 18 person-years to complete the first FORTRAN compiler, and that was the definitive compiler for the most popular programming language of the time.

They instead began developing a Lisp *environment*, at this point a set of functions compiled by hand for reading and printing list structures. These programs were written in *M-expressions*, an informal language resembling FORTRAN with assignments, `goto`s, the core Lisp functions, and a few of the features they had wanted from FORTRAN but couldn't get, namely recursive functions.

Beginning with "Recursive functions of symbolic expressions and their computation by machine, Part I", McCarthy and the team at MIT began writing papers describing theories in Lisp. The 1960 paper was for using Lisp as both a programming language and formal language for recursive function theory, and several subsequent papers represented Lisp programs in first-order logic. A more impactful application was the attempt to prove that Lisp was more elegant than turing machines by formally describing a *universal Lisp function*, and showing that it was more elegant than the Turing machine.

This universal function would be a Lisp program capable of evaluating any other possible Lisp program. McCarthy described it formally as $eval[e, a]$ where the function *eval* takes a Lisp expression e and evaluates it in the context of a list of assignments to variable names a. Steve Russel noticed that an implementation of this function could serve as an interpreter for the language–he wrote this function in assembly, and the team had an interpreter.

Similar to Backus's experience with FORTRAN, many of the decisions they made in designing the original Lisp without much forethought became solidified by the availability of a programming environment, and the expectation that the code people had written in the language would continue to work. The number 0 representing both the empty list `NIL` and the boolean value `FALSE` is one such case.

The prefix-notation and use of parentheses (also known as *S-expressions*) are another case; they had initially set out to write Lisp programs in their M-expressions and the prefix notation syntax was simply the internal representation of the list structures in the language. The tasks of formally defining M-expressions and translating them into S-expressions was never completed, and the first users of Lisp did not mind writing programs in the compiler's internal representation, so it stuck. The team dubbed this version of Lisp, with all its warts, Lisp 1.

With this version, the team showed that with a few primitive functions, `quote`, `atom`, `eq`, `car`, `cdr`, `cons`, and `cond`, they could implement a *total language*, or a language capable of expressing any Lisp function. By providing those primitives in assembly, they had bootstrapped a programming language, at this time just an interpreter. It had very few features (truly, only the primitives above were available), but they would soon extend it to produce a far more usable programming language.

3.7.2.4 Lisp 1.5

Many of the obvious missing features of Lisp 1 were added in Lisp 1.5 (including numbers(!)). We will not cover them exhaustively here, but McCarthy's reccolection in (John McCarthy, 1978) seems to be the authoratative source and the list is short. The one we will cover is *lexical scoping*. McCarthy points to example below as the onus for introducing lexical scoping. James Slagle of the Lisp team wrote the following function to find a subexpression in `x` such that the predicate `p(x)` is true, and to return `f(x)` if so. If the search did not find such an `x`, then the function containing the remainder of the computation taking no arguments `u` should be called and its value returned.

```
(define (testr x p f u)
  (cond
    [(p x) (f x)]
    [(not (pair? x)) (u)]
    [else (testr (car x) p f (lambda () (testr (car x) p f u)))]))
    ;                          Which x is this?  ^^^^^
```

The issue, pointed out in a comment in the code block, is that the `(car x)` component of the continuation function actually used the value from the scope *where it was called* and not the scope *where it was defined*. James had expected the latter, but got the former. McCarthy simply thought this to be a bug and assumed that Steve Russel would fix it.

Today, these two semantics for variable scoping are called *lexical scoping* and *dynamic scoping*. Lexical scoping is the semantic James wanted, and dynamic scoping is the one he got in the original Lisp implementation. They are so-called because in lexical scoping, the value of a variable is determined by the literal code surrounding the variable's use–in this case, the function body of `testr`. Dynamic scoping is so-called because the variable is looked up from the environment *where the variable is used*. In this case, `(car x)` is looked up in the environment where the continuation function is called, instead of where the continuation function is defined.

This distinction was important in the evolution of ALGOL as well– the final ALGOL 60 report does not mention these terms for scoping by name, but their concepts are seen in the revised report, particularly (A. van Wijngaarden, B. J. Mailloux, J. E. L. Peck, C. H. A. Koster, et al., 1976, Section 4.7.3.3. Body replacement and execution). All of the early compilers were faced similar challenges– languages like ALGOL that were designed from the top-down by a committee and languages like Lisp that were designed from the bottom-up for a particular purpose still had to grapple with the same issues eventually.

Russell eventually did implement a solution for James through the FUNARG construct, which allowed a function definition to take with it a pointer to the environment where it was defined, thereby permitting lexical scoping in a language with dynamic scoping. Members of the team at MIT discussed solutions to this problem in (Weizenbaum, 1968) and (Moses, 1970) a few years later in 1968 and 1970, respectively.

Along with the problem of lexical and dynamic scoping, the team also provided a means for call-by-name, very similar to the concept in ALGOL. The "functions" FEXPRS and FSUBRS are not given regular arguments, but are given expressions that will evaluate to their arguments. The function bodies can use Lisp's `eval` function to actually evaluate their arguments in the body of the function as they see fit. While this is similar to ALGOL's call-by-name which treated all unquoted arguments this way, the Lisp versions allowed users to decided exactly when and how their arguments were evaluated. Users could pass the code that comprised their arguments to further calls to FEXPRS and FSUBRS, or they could evaluate some of them at the start and some of them later, and so on.

3.7.2.5 The First Lisp Compilers

J. Allen collected the early histories of Lisp compilers in (J. Allen, 1978) from a few primary sources that are difficult to find, namely the paper "The structure of the LISP compiler" by Blair, which was apparently never published. Essentially, a few different teams set out to develop compilers for Lisp to varying degrees of success and completion.

The first was Robert Brayton at MIT (Blair, 1971), who started working on the first Lisp compiler in 1957 on an IBM 704, though he was not successful until 1960, after which he left MIT. He wrote this compiler in assembly, and so it was thought that his compiler would be more performant than the competing Lisp compilers, as they were all written in Lisp themselves. Evidently, nobody was able (or willing, perhaps) to maintain this compiler after Brayton left MIT.

Klim Maling started work on a Lisp compiler written in Lisp in the time after Brayton began work on his compiler, but before the compiler was functional, though the project was dropped before it was functional itself.

Timothy Hart and Michael Levin developed the first complete Lisp compiler, also written in Lisp, which they claimed was the first bootstrapped compiler, or the first compiler to be written in the language being compiled. This compiler was distributed with the Lisp 1.5 system for the IBM 704, and was the basis for much of the future work on Lisp compilers. While future versions of Lisp were being worked on, most users in this period were stuck with this Lisp 1.5 developed at MIT, which fixed only the most glaring of the issues with the original language.

As Blair recalls in "The structure of the LISP compiler", the first Lisp compilers were quite straightforward compared to the contemporary compilation efforts with ALGOL and Fortran. Blair points out that because Lisp is comprised only of untyped data objects, the compiler's primary concern is that of variable binding and variable evaluation. Blair cites Peter Landin as influential in his understanding of expression evaluation semantics, and prefers the model he describes in "The mechanical evaluation of expressions" over the typical understanding of a Lisp interpreter *eval*[*e*, *a*], which we discussed above.

3.7.2.6 After Lisp 1.5

In the early 1960s, prior to Lisp 2, the language and all its implementations were still extremely limited. Numerical programs were painfully slow compared to other compiled languages at the time (*numbers* were only just added in the first version), the I/O facilities were limited, and the memory model was not yet robust. The garbage collector was implemented, but slow and limited. Lisp objects could not be represented in registers and garbage-collected.

Two companies, Systems Development Corporation and Information International Inc., collaborated on the second version of Lisp with a new compiler built for the military version of IBM's 7090 called the Q32, but system's the memory constraints proved too restrictive. The breakdown of the collaboration between SDC and III and MIT and Stanford along with practical, political, and economic decisions that went into deciding which machine the new compiler should be developed for, all added up to the abandonment of the project. There were few subsequent efforts to radically change Lisp in the decades to follow after the failure of this more ambitious effort. Suffice it to say that the dominant system for AI work would become the DEC PDP-10, the successor to the PDP-6, and the language would not look radically different from version 1.5.

Outside of MIT, the adoption of Lisp was greatest at DEC. Their implementations of Lisp on the PDP-6 and PDP-10 were widely used and their instruction set was defined with Lisp in mind(John McCarthy, 1978). The IBM 704 Lisp was extended to the 7090, 360, and 370. The subsequent years of Lisp development were eventful; we will resume this story in the next section.

[TODO: Continue here: "The evolution of Lisp", 6.2.1 From Lisp 1.5 to PDP-6 Lisp: 1960-1965. First cross-compiler to bring up a language. Just had to port/cross compile the "subconscious" (J. Allen, 1978, pp 289). Teams were generally isolated and had a few people that knew lisp really well and everyone else just used the language. Maclisp and Interlisp compilers. Richard Stallman worked on MIT Lisp Machine project in 1974. The DWIM/Do what I mean lisp that would spell-correct or fix syntax errors and go ahead and do what the user intended(!!), and had an ast-based editor(!!). NLAMBDA vs LAMBDA, more calling conventions. PSL, portable standard lisp. Mostly in the 70s and 80s now. "The Portable Lisp Compiler (PLC) compiled Lisp code into an abstract assembly language. This language was then converted to a machine-dependent LAP (Lisp Assembly Program) format by pattern-matching" wow!]

[TODO: mention Lisp machines so we can refer back to them in section 5.2] . [TODO: mention (Kessler, 1984) compiler optimizations originally deployed to lisp compilers.]

3.7.3 Peter Landin on the Lambda Calculus

Exemplary of the varied approaches to formalizing the design of ALGOL 60, Peter Landin sought to express the language's semantics in the λin (P. J. Landin, 1965) [17]. A few years earlier, McCarthy adopted some components of the λin Lisp, however McCarthy's language broke with λin a few key areas, namely dynamic binding [18]. ALGOL's semantics provided a much cleaner relationship to λ, thanks to its block structure and lexical scoping rules, thus Landin made it possible to look at λas a programming language in and of itself in a more complete way than McCarthy had done with Lisp.

In parallel with his translation of ALGOL, Landin developed in (Peter J Landin, 1964) an abstract machine for λcalled the *SECD-machine*.

3.7.4 From the Theoretical to the Practical

At this point in our history, λwas primarily a theoretical tool used in academia, the exception being McCarthy's work on Lisp, which was still not nearly as pragmatic a programming language as FORTRAN, for instance. In the section 4.6, we will explore how these theories made their way out of academia and strict mathematics into the more practical realm of programming languages and compilers.

3.7.5 todo: notes on history of λ-Calculus

"History of lambda-calculus and combinatory logic" *Lambda-Calculus and Combinators: An Introduction* Cardone and J Roger Hindley

history of lambda calc: Both lambdacalc and combinatory logic were invented in the 1920s. Describe most basic properties of functions. logical foundation of math was real fluid in early 19thc. Russell's paradox was recent Gödel's theorems not known yet, still developing the foundations of math. Some were based on sets, some based on functions, all up in the air. lambda and CL were developed here, build higher concepts of functions based only on these two foundations. <cardone> describe as chassis of bus in prog langs. lambda/cl gain purpose in these systems, like chassis; underpins things and isnt seen, doesn't have much value on its own. function-based higher-order logic is the real context, but we're already so far off topic and I'd like to get back to compilers soon as I can. Lots was developed in the 20s and 30s, then not much till the 60s largely cuz of connections with prog langs. notations for higher-order thinking about functions dates earlier, at least 1889 Giuseppe Peano on axioms for arithmetic. was not until Moses Schönfinkel developed basic combinators for functions (1924 sec 1) that we start to get combinatory logic. at gottingen germany, prob top math research group of the period. Pointed out f(x,y) could be (f(x))(y) where f returns func, now known as "currying." curry attributed to Scho.. many times but the name stuck. that was the last thing scho published on combinators, and by 1927 he was said to be mentally ill and institutionalized.

his ideas cropped up again in jv neumann doctoral thesis on foundations of set theory (neumann 1025) but it was really function-based not set-based, and his axioms contained combinator-like operators (p225). not sure if JVN got this from scho or not, he didn't mention him and his ideas looked really different. next step in CL: haskell curry re-invented combinators.

6.1 lisp 1956-60 mccarthy lisp w function-abstraction. didn't have numbers, but just like lambda calc. substitution was not like lambdacalc bc dynamic binding, which helped interpreters but made

[17] See section 3.6.2 for more details on the development of ALGOL 60.

[18] For a more complete treatment of Lisp, see section 3.7.2.

programming really complicated. mccarthy contributed some back, including cond-exprs in functional formalism.

6.2 peter landin early 60s PL proposed lambda terms to make constructs of a programming language algol 60 (landin 1965). algol block structure/identifier semantics matched lambdacalc, allowed ppl to look at lambda as a programming langauge. in parallel with this algol stuff, landin 1963 abstract machine for reducing lambda terms SECD-machine (1964, 1966a) of 4 components: stack for intermediate results, environment, control (code) driving the process and dump for showing the state of the program. the rules of his machine used call-by-value. would be followed up by further considerations of lambdacalc as a prog lang (Plotkin 1975). For the most part, other than McCarthy's papers, CL/lambda contributed to proglangs but not the other way around, until Corrado Bohm from 1960s on. bohm phd thesis w first description of complete compiler written in its own language [bohm 1954].

encoding of operations of iversons APL marisa venturini zilli 1962-64. gotta summarize bohm. james morris MIT 1968 thesis, untyped and simply typed lambda calculus. substitution had to be reconsidered in lambda calculus and CL cuz Lisp necessitated the efficient implementation of these algorithms, not just theoretical.

8 types: russel whitehead's principia mathematica. Types were there before they came up in programming languages bigtime, but they werent fully deployed until later. so we start with lambdaclac and CL and then we'll do type theory in Software section. 1967 los angeles set theory symposium [Lawvere, 1969a].

3.8 Timeline

Date	Event
1942	Zuse conceptualizes first high-level language *Plankalkül*
1944	Aiken and IBM's Automatic Sequence Control Calculator (*Mark I*) started working
1946	Hopper's Mark I manuscript published
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
1951	Böhm develops practical compiler for PhD thesis at ETH
	Hopper begins work on the A-0 compiler for the UNIVAC I
1952	Hopper presents *Automatic Programming* at ACM
	Alick Glennie develops compiler for Mark I
1953	Hopper's team completes the A-1
	Hopper's team completes the A-2 (featuring *pseudocode*, 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 Backus at IBM begins work on FORTRAN
1954	Nora Moser of US Army sends Hopper modifications to the A-2
1957	John Backus at IBM released the first FORTRAN compiler, the first commercial compiler

Table 3.3: Timeline of early developments in programming languages and compilers.

4
Software, *1960-1980*

4.1 FORTRAN

FORTRAN carried computing into the age of software, and it would continue to dominate for decades.

4.1.1 Fortran V/66

4.1.2 Fortran 77

4.2 The Software Crisis

Note that we will discuss some giants of computing history in this chapter who cast long shadows over the field; however, we will primarily focus on those involved in the development of compilers and programming languages. In particular, we will not do justice to key figures from Bell Labs, such as Brian Kernighan, Ken Thompson, Dennis Ritchie, Claude Shannon, and Doug McIlroy. I can recommend *The Idea Factory* for a more comprehensive account.

As Hopper and Backus had put it numerous times, programming was still exceedingly painful at this point in time, and the relative cost of programming kept increasing. As machines became cheaper and more powerful, the share of the cost of building and running a program devoted to programming increased, and thus the importance of ease of programming increased as well.

Michael Mahoney, doctor history and history of science, goes so far as to describe programming and computer design in this way as early as 1945 (Raul Rojas and Hashagen, 2002, 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.

While this description was correct about the *direction* compiler engineering and programming language design were going, the statement was far too strong to make about the year 1945. Automatic programming was still scarcely more than synthetic machine code so the programmer did not have to concern themselves with the details of the particular machine's instruction set to the same extent, but they were still very much writing programs by writing specific instructions instead of concerning themselves with the problem they were trying to solve and allowing the compiler to turn that into proper machine code.

American historian of computing and aerospace Paul Ceruzzi describes the situation in 1968:

> 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 [1]. 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. (P. E. Ceruzzi, 2003)

Backus describes this as part of the motivation for beginning work on `Fortran` (J. Backus, 1978c, *The Economics of Programming*):

> Another factor that influenced the development of Fortran was the economics of programming in 1954. The cost of programmers associated with a computer center was usually at least as great as the cost of the computer itself...Thus, programming an debugging accounted for as much as three quarters of the cost of operating a computer; and obviously, as computers got cheaper, this situation would get worse.

Thus this chapter is about the efforts to make programming easier and more productive to address this crisis. Software rose in importance in industry, it became a field of study in academia, and true research on compilers and programming languages began.

4.3 Structured Programming

(O. J. Dahl, Dijkstra, and Hoare, 1972) in 1972 from O. J. Dahl, Dijkstra, and Hoare. This surveyed several diverse techniques for structuring programs. It was the notion that control flow jumping around programs (as with `goto`) was generally a bad thing that connected these papers and concepts together.

[TODO: simula]

4.4 Pascal

Runtime checking. tagged pointers and objects had tags such that if the tags didn't match you would get a runtime error (like the lisp machines). If your program compiled and didn't get any runtime errors, your program was probably correct.

pascal started as teaching, couldn't do anything with it. then they made modula to make it usable. omsi pascal named after our own omsi in oregon.

[1] Readers may find it entertaining that Ceruzzi describes COBOL and FORTRAN as obsolete *by 1968*; both are still heavily used in some industries, and one of the very first compilers to adopt MLIR (to be discussed in a later chapter) is a Fortran compiler(Spickett, 2025).

4.5 Bell Lab's Computing Science Research Center

4.5.1 Multics and Unix at Bell Labs

There is no avoiding Multics and UNIX if we are to discuss the development of compilers at Bell Labs. We will cover the history of the American Telephone and Telegraph Company (AT&T) in *very* broad strokes to contextualize the development of the programming languages and compiler tools developed in this time period.

In very broad strokes, American Telephone and Telegraph Company (AT&T) was a government-sanctioned monopoly that provided telephone service to the United States. They became a monopoly by buying up smaller telephone companies across the country. Over time, they found they needed quite a bit of fundamental science in order to keep up with the requirements of servicing the entire country, so in 1925 they created Bell Telephone Laboratories in New York. If they did not continue providing quality service, the government could (and very often did) threaten to break up the company, so this need was existential.

During the second world war, they expanded to New Jersey, early 1960s they moved to Murray Hill, in suburban New Jersey, where they had a few thousand employees. The laboratory was responsible solely for research, and thousands of other employees worked on applying the research and developing the process knowledge needed to bring the research into the phone system. This research led to numerous fundamental discoveries and inventions in physics, materials science and many more domains; these discoveries include the transistor, lasers, fiber-optics and solar cells, to name a small subset. Patents were the laboratory's primary measure of success, and they produced many. At its peak, AT&T was the nation's largest private employer with over 1 million employees. If you are interested in an account of Bell Labs in anything close to the proper detail, I recommend *The Idea Factory* by Jon Gertner; our focus if far too narrow to do it justice here.

In the early 1960s, the labs spun off a new department, the Computing Sciences Research Center, off of the mathematics group. This group had about 25 people tasked with software and computing research, which included operating systems. In 1961, Fernando Corbató at MIT created a timesharing system called CTSS, or the Compatible Time Sharing System, with relative success. In 1965 MIT set out to make a successor system with all kinds of enhancements; they partnered with Bell Labs and General Electric to create the operating system called *Multics*. The machine was developed at GE (Multics required a special machine), Multics was designed at MIT, and Bell Labs was responsible for large parts of the software development, which was primarily done in IBM's PL/1 programming language. Ken Thompson described it as "monstrously over-engineered" and "typical second-system syndrome" (Thompson, 2019); while they had a wonderful time-sharing system in CTSS, they got too ambitious with thier sophomore project. Ultimately, this project ran long past deadlines and was slow to run and altogether not what the labs was hoping for, so in 1969 they pulled out of the project. Honeywell took over Bell Labs' share of the project.

This left a bad taste in their mouths, and Bell Labs leadership sought to avoid operating systems work in the wake of this venture. Some of the members of the Multics team had other ideas; they had gotten a taste for operating systems work, and now they had not operating system to work *on*. These included Ken Thompson, Dennis Ritchie, and Doug McIlroy.

At the time, Ken Thompson was researching file systems on an outdated and underused Digital Equipment Corporation (DEC) PDP-7 machine from the acoustics group. [2] He originally just used

[2]Because acoustics research was so core to Bell Labs' and AT&T's mission, that department got just about anything

it for writing video games, until finally getting around to his research on file systems. He developed disc scheduling algorithms to maximize throughput on the PDP-7's disc drive. To adequately test the performance of his file system algorithms, he needed some test programs to stress the file system. At some point in this process, he realized he was not far off from an operating system of his own(Thompson, 2019):

> At some point, I realized with out knowing it up until that point, that I was three weeks from operating system, with three programs, one a week. I needed an editor to write code, I needed an assembler to turn the code into language I could run, and I needed a kernel overlay, call it an operating system. Luckily, right at that moment, my wife went on a three-week vacation to take my one-year-old to visit her parents in California. One week, one week, one week, and we had Unix.

Thompson's assembler was not a complete toolchain; there were no libraries, linkers, or loaders. The entire assembly program had to be presented to the assembler all at once, and the output was directly executable. Thus the `a.out` naming convention for the default executable output of modern compilers was born (*a*ssembly *out*put). Even after the system gained more toolchain components, the name remained.

Very early on, Thompson's system picked up very impressive users. Two at a time, Dennis Ritchie, Doug McIlroy, Morris McMahon, and Brain Kernighan all became users on Thompson's new system. They sent a proposal to get a PDP-10, which was top-of-the-line at the time, to port Unix to a more powerful machine. This was rejected as soon as Labs leadership saw it was related to operating systems, in spite of the fact that their request was well within budget. Joe Ossanna (another colleague) came up with a proposal that positioned their need for a PDP-10 as a research project to develop better typesetting software for filing patent applications (notably missing any mention of operating systems). Ossanna developed the Nroff and Troff typesetting programs to fullfil their claimed goal of helping the patent office. The patent office loved their software, and they ended up buying the computing group a PDP-11[3] sometime in the early 1970s. Unix came up on the PDP-11 almost immediately. Thus we have the basis for discussing the subsequent development of compilers and programming languages at Bell Labs.

4.5.2 Personal Histories

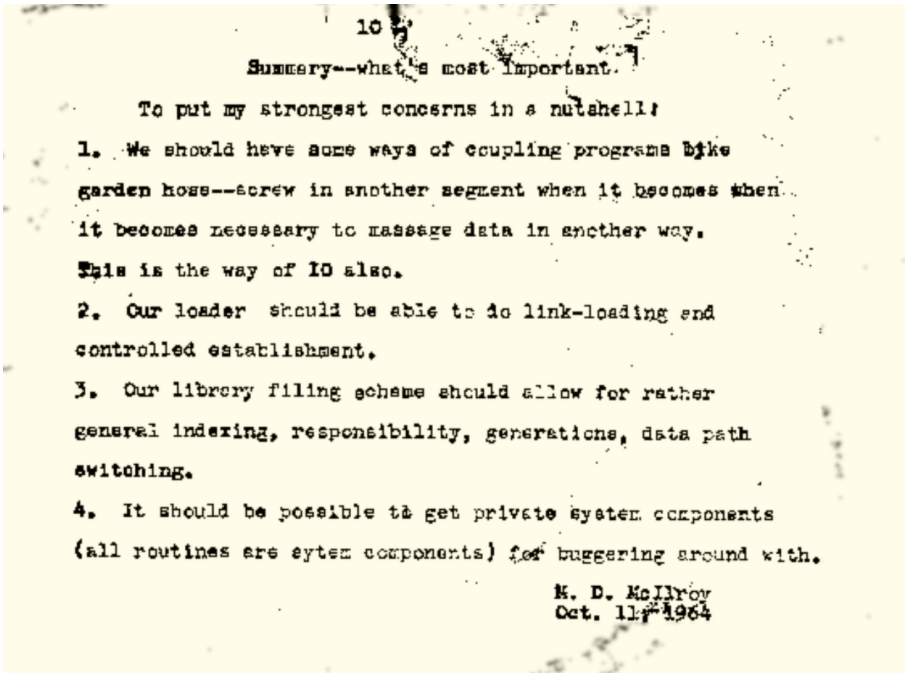
The following sections will feature many of the same figures; here we will introduce them so their interleaving stories may be told uninterrupted in the following sections. Readers unfamiliar with the history of Unix may find unfamiliar terms in these personal accounts; these will be discussed in the following sections after our characters have been introduced.

4.5.2.1 Doug McIlroy, Joined 1958

Douglas McIlroy was born in Fishkill, New York. His father worked in electrical engineering and spent time at MIT and ended his career at Cornel, having contributed to the RADAR effort in World

they asked for in terms of computing resources, hence the outdated PDP-7 sat unused in their offices–they already had more powerful machines.

[3]While many resources will mention the PDP-11, they usually refer to the PDP-11/*22*; very few of original PDP-11s (which the computing center worked on) were ever produced, but the refreshed models had many users.



10

Summary--what's most important.

To put my strongest concerns in a nutshell:

1. We should have some ways of coupling programs like garden hose--screw in another segment when it becomes then it becomes necessary to massage data in another way. This is the way of IO also.

2. Our loader should be able to do link-loading and controlled establishment.

3. Our library filing scheme should allow for rather general indexing, responsibility, generations, data path switching.

4. It should be possible to get private system components (all routines are sytem components) for buggering around with.

M. D. McIlroy
Oct. 11, 1964

Figure 4.1: Doug McIlroy's 1964 memo proposing Unix pipes(M. D. McIlroy, 1964).

War II. His father collected maps, and Doug grew an interest in maps himself. One day, while ill in bed with chicken pox, his father assigned him the task of drawing a dam on a USGS map, and showing which regions would be inundated by the dam (Malcolm D. McIlroy, 2019). His mother, too, had a master's degree in physics from the University of Rochester, which was very unusual at the time. She was forced to audit some of her classes there, because "women can't take this class! But you can sit in on it, if you want." Thus he was raised in an engineering-minded household.

Barbara, his wife, also studied mathematics and faced discouragement similar to his mother's. They ended up meeting at the laboratory, where Doug recalled that some figures at the labs sought to improve the situation for women, specifically Bernie Holbrook.

Doug joined Bell Labs in 1958 and shortly thereafter earned his PhD in applied mathematics from MIT. McIlroy became head of the Computing Techniques Research department in 1965, and he's regarded as one of the most brilliant members of the staff by key members that readers may be more familiar with.

He is described by nearly everyone at the Bell Labs Computing Research Center (who wrote or spoke about their time there) as the most brilliant member of that team that no one has heard of (Thompson, 2019); Ken Thompson described him as "the smartest one of all of us and the least remembered or written down of all of us."

He was relatively hands-off as a manager, preferring to peek into his employees' offices with suggestions or interesting problems and wait for the employees to let *him* know what needed to be done, rather than assigning tasks directly. His employees respected his tasted in technical and personal matters, as Brian Kernighan describes(Kernighan, 2020):

> Unix might not have existed, and certainly would not have been as successful, without Doug's good taste and sound judgment of both technical matters and people.

In 1964, he circulated a memo which led to Unix pipes(M. D. McIlroy, 1964) (see 4.1) though it

took him quite a while to convince Ken Thompson that he really ought to implement them. Aside from pipes, he wrote many common Unix utilites we still use today, like `diff`, `sort`, `join`, `tr`, `echo`, `tee`, and `spell`.

He hired Alfred Aho in 1967(Alfred V. Aho, 2022):

> 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 co-invented macros for programming languages and he's also in this class of one of the smartest people I've ever met.

4.5.2.2 Ken Thompson, Joined 1966

[TODO: Ken's oral history is rich and includes fun stories about how he was recruited to the labs.]

4.5.2.3 Dennis Ritchie, Joined 1967

4.5.2.4 Jeffrey Ullman, Joined 1967

[TODO: This section can be shorter since Ullman went back to Princeton and only consulted at the lab part-time.]

4.5.2.5 Alfred Aho, Joined 1967

Alfred grew up in a Finnish household, and when he showed up to Kindergarten, he couldn't speak any English. In his first report card, the teacher told his parents that "Al can't speak any English!", and in the next report card, the teacher said reported that "Al speaks *too much* English!"; so he learned it relatively late and became quite a social kid. Alfred played music all throughout his childhood, and the schools he went to as a kid had quite good musical programs. He went to University of Toronto for undergraduate school and continued playing the violin all through his years at Bell Labs. He was an only child. He had a proclivity not only for music but for mathematics and reading science fiction.

After finishing his undergraduate degree in Toronto in engineering physics, he got a masters and then PhD from Princeton University in electrical engineering and computer science, completing the latter in 1967. At Princeton took a course from John Hopcroft on computer science theory with a heavy emphasis on automata and language theory, which sparked his long-lasting interest in formal languages. His PhD thesis was on extending the theory of context-free grammars, an area of expertise that would serve him well at the Labs.

One of the fist people he met at Princeton was Jeffrey Ullman, who had also recently started there, beginning a long-term friendship and collaboration between the two (Alfred V. Aho, 2022):

> 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.

His PhD advisor was John Hopcroft, whom he would also continue to work with for a long time. He told Alfred to "find his own research problem," and he turned to his interest in programming languages and compilers. He was always very keen on precision; he wanted to use precise terms and he would press even his own friends to be very precise in their informal discussions, and this acuity for precision extended to his thesis, which was on the very precise understanding of the *syntax* and the *semantics* of programming languages. Alfred would contribute his deep understanding of the theory of automata and programming languages to the Labs, where there were many other brilliant people who were more practical and lacked familiarity with the literature. In 1967 his good friend Jeffrey Ullman had started working at Bell Labs, and a few months after starting there, Alfred interviewed there. He was interviewed by Doug McIlroy, who hired him. For some time, he and Jeffrey worked together at the Labs, but Jeffrey had wanted to spend more time in academia than Alfred, so he returned to Princeton's electrical engineering department, but continued to consult at Bell Labs one day a week. Thus the two continued to work together. Alfred described their collaboration at Bell Labs:

> 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, 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. (Alfred V. Aho, 2022)

Ken Thompson joined the labs several months before the two of them to start working on Multics, and had developed a plethora of tools based on regular expressions, such as `grep`. This combined with the ED and QED editors that came from MIT and were shipped with Multics sparked Alfred's interest in regular expressions.

Aho is probably best known for the textbooks he co-authored with his friend Jeffrey, colloquially known as *The Dragon Book* ((Alfred V. Aho, Sethi, and Jeffrey D. Ullman, 1986)) and his book with John Hopcroft on algorithms, (Alfred V. Aho and Hopcroft, 1974).

4.5.2.6 Brian Kernighan, Joined 1969

4.5.2.7 Bjarne Stroustrup, Joined 1979

Two people working on compilers at Bell Labs in the late 70s and early 80s were on different wavelengths from the C and Unix developers (and also from each other)– Bjarne Stroustrup and David MacQueen.

Bjarne was a Danish computer scientist who came to the labs from [TODO: where?] with a background in using SIMULA for event-driven simulations, and wanted to bring that experience to C.

4.5.2.8 David MacQueen, Joined 1981

David didn't start working at the labs until much later than the others in this section and he primarily worked on his own, unlike the others mentioned in this section. Because of his work on Meta Language and type theory, we include some of his personal history here as well. University of Edinburgh, working on Standard ML.

4.5.3 The First Unix Compilers

Alfred Aho described this era of compiler history as as cambrian explosion of programming languages and compiler tools (Alfred V. Aho, 2019), and there were several reasons for that. Tools were developed that made the description of compiler front-end tools extremely simple, the computing center got access to machines with more memory, and the theory of parsing context-free grammars advanced. The Labs employees worked all of these together to produce many compiler tools.

While Doug was the department head of the Computing Sciences Research Center, he would regularly stop into the offices of his employees and drop ideas and requests to them. In one case, he mentioned to Ken how convenient it would be to have a tool for searching text files for patterns. Ken already had a rough version of such a tool in his home directory, so the next day he showed Doug his program and thus was born `grep`; many such tools came to be like this.

On another similar occasion, Doug walked into Ken's office to discuss the TMG language (standing for TransMoGrifier), designed by Robert McClure(McClure, 1965), one of his friends at the lab. McClure had gotten defensive about this language, and when he left the lab, he claimed it was proprietary and nobody else could use it. TMG was a top-down recursive-descent compiler-compiler for context-free grammars and procedural elements similar to Yacc, a tool to help write compiler front-ends. See this example from Doug's TMG manual (M. D. McIlroy, 1972) and notice the similarities to modern parser generators and Backus-Naur Form:

> This simple program defines the translation of fully parenthesized infix expressions to Polish postfix for a stack machine.

```
1       expr: <(>/exp1 expr operator expr <)> = { 3 1 2 };
2       exp1: ident = { < LOAD > 1 };
3       operator:
4       op0: <+>/op1 = { < ADD > };
5       op1: <->/op2 = { < SUB > };
6       op2: <*>/op3 = { < MPY > };
7       op3: </>     = { < DIV > };
```

Ken recounts(Thompson, 2019) the story of how Doug brought a full TMG compiler *written in TMG, on paper* to Ken's office. He then decided to feed the TMG program into his TMG compiler *by hand* to produce an assembly listing of the TMG program. He then went over to Ken's keyboard and typed in the program that his TMG-TMG compiler had produced, and with astonishingly few errors, they had a working TMG compiler on the PDP-7.

Ken's logical next step was to use this new TMG compiler to write a `Fortran` compiler for the PDP-7 because "no computer was complete without `Fortran`." Now, the PDP-7 was only 8k of 18-bit words of memory and Ken's Unix system needed about half of that just to run, leaving only 4k for user programs. Once completed, the new `Fortran` compiler took far more memory than

was available for user programs, so he had to cut down the TMG program to get a program small enough to fit on the machine but capable enough to facilitate a usable programming language. Once he finally cut down his compiler to 4k, he called that programming language *B*. His ill-fated attempt at producing a `Fortran` compiler ended up being a very different language, based more so on BCPL than on `Fortran` [4]. BCPL had been designed by Martin Richards at MIT in the mid-1960s, and thus had found its way onto MIT's CTSS system, which Bell Labs employees were familiar with.

4.5.4 BCPL and B

BCPL, B and C may differ syntactically, but semantically they are very similar and operate at a similar level of abstraction. The BCPL compiler was allowed to use more memory than Thompson had on the PDP-7, so it allowed for more complicated expression-oriented language features. Dennis Ritchie points out the following two examples as only being possible in BCPL and not in B or C due to memory constraints (Ritchie, 1996):

```
let P1 be procedure
and P2 be procedure
and P3 be procedure

E1 := valof ( declarations ;
              commands ;
              resultis E2 ) + 1
```

Because the entire program was held in memory, the BCPL compiler could make procedure ***P3*** available to expressions inside procedure ***P1***, but the B compiler was limited to a one-pass technique and could not provide such features. On the other hand, some BCPL features were purposefully left out of B; for example, to share data between separately compiled source files, BCPL programs had to use a global buffer. To make a procedure or variable available to other source files, the programmer had to manually associate the name with an offset into this buffer, similar to the `COMMON` block in `Fortran` programs. In more mature B compilers and later in C, the linker handles this automatically.

The B compiler did not emit machine code directly, but instead used *threaded code*(Bell, 1973), an early form of bytecode operating on a simple stack machine, which was then interpreted. James Bell claimed that this threaded code needed no interpreter, but really, the execution model for threaded code is just a specification for an interpreter without outside the routines responsible for executing a given bytecode instruction. Today this would be considered a bytecode interpreter, but in the day when interpreters were only known to interpret the source language directly, his ideas may have seemed distinct from other forms of interpretation. See Bell's depiction of the execution of threaded code 4.2.

Neither B nor BCPL had types; only the addressable size of data for the particular machine. Since pointers were just integers representing offsets into memory, pointer arithmetic was natural in both languages, and both had syntax sugar to facilitate pointer arithmetic and dereferencing of offsets into arrays.

[4] The origins are really unclear. Ken himself mentions `Fortran` as his only inspiration for B in a live interview(Thompson, 2019), however Dennis Ritchie describes it as a "language of his own" and "BCPL squeezed into 8K bytes of memory and filtered through Thompson's brain" (Ritchie, 1996). He describes the name as either a contraction of BCPL or (less likely) a contraction of Bon, a language Thompson developed for Multics.

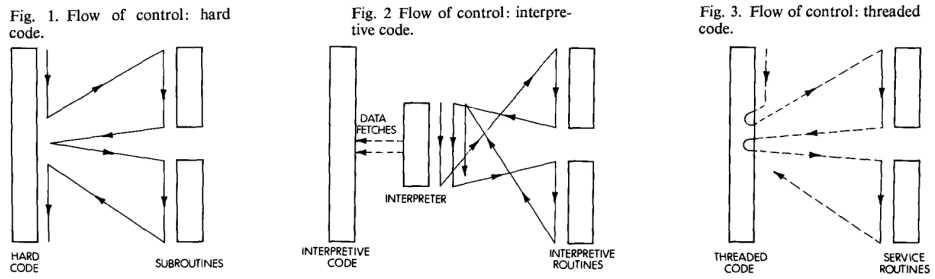


Figure 4.2: Execution of Threaded Code(Bell, 1973)

```
1 /* Declaring an array of 10 integers */
2 let V = vec 10 In BCPL
3 auto V[10];    In B
4
5 /* Accessing element i of array V */
6 V!i            In BCPL
7 *(V+i)         In B
8 V[i]           Also in B
```

After Ken had the TMG version of B working, he bootstrapped the compiler by implemented it in B. Dennis Ritchie recalls(Ritchie, 1996):

> During development, he continually struggled against memory limitations: each language addition inflated the compiler so it could barely fit, but each rewrite taking advantage of the feature reduced its size. For example, B introduced generalized assignment operators, using x=+y to add y to x. The notation came from Algol 68 via McIlroy, who had incorporated it into his version of TMG.

Dennis also recalls this about their language design and implementation decisions:

> Although we entertained occasional thoughts about implementing one of the major languages of the time like Fortran, PL/1, or Algol 68, such a project seemed hopelessly large for our resources: much simpler and smaller tools were called for. All these languages influenced our work, but it was more fun to do things on our own.

By 1970, the team had been able to acquire the PDP-11, which they promptly ported Unix to, along with the B bytecode interpreter and a small assembler written in B. As the number of PDP-11 users grew, so too did their B codebase, which now included quite a few programs and subroutines, as a result of the tedium of writing assembly, which was the only alternative. B proved useful– however, the new machine revealed some serious inadequacies. The only data type in BCPL and B was the `cell`, roughly equivalent to the hardware's machine word. The 18-bit PDP-7[5] did not map

[5]18-bit might seem like an odd choice to the modern reader; most mainframe computers had 36-bit words, and the PDP-7 was marketed as a smaller system. Thus at 18-bits, it would be marketed as roughly half a mainframe.

cleanly to the 16-bit PDP-11. BCPL on the PDP-7 had support for floating-point operations through special operators, but this only worked because the floats could fit in a single word and this was not possible on the PDP-11.

> For all these reasons, it seemed that a typing scheme was necessary to cope with characters and byte addressing, and to prepare for the coming floating-point hardware. Other issues, particularly type safety and interface checking, did not seem as important then as they became later. Aside from the problems with the language itself, the B compiler's threaded-code technique yielded programs so much slower than their assembly-language counterparts that we discounted the possibility of recoding the operating system or its central utilities in B. (Ritchie, 1996)

In 1971, Dennis Ritchie began extending B by adding a character type and directly generating PDP-11 machine code. Thus the creation of a compiler capable of generating efficient machine code on a system with very little memory and the transition from B to C were simultaneous. Dennis renamed the B compiler to NB, or *New-B*, and then *C* shortly thereafter.

4.5.5 C

Dennis's new language now had types and new addressing schemes, and it evolved quickly. *New-B* existed so briefly that no full description of its syntax or semantics was ever written.

The run-time overhead associated with BCPL and B's vector indexing semantics were addressed in C:

> Values stored in the cells bound to array and pointer names were the machine addresses, measured in bytes, of the corresponding storage area. Therefore, indirection through a pointer implied no run-time overhead to scale the pointer from word to byte offset. On the other hand, the machine code for array subscripting and pointer arithmetic now depended on the type of the array or the pointer: to compute `iarray[i]` or `ipointer+i` implied scaling the addend `i` by the size of the object referred to. (Ritchie, 1996)

C gained a type system, overlaying semantics onto the raw bits that BCPL and B users were working with. The type system was heavily inspired by ALGOL 68, though not necessarily in a way the authors of the ALGOL language would have been happy with, as Dennis himself admits (Ritchie, 1996). Many features were added to the language in a short time; some of the warts that persist in even modern C were because of Dennis's desire to make porting programs from B to C as simple as possible.

For example, the precedence of `&` and `==` were made equal in C, thereby requiring parentheses in the second expression below:

```
if (a == b & c) { /* ... */ }
if ((c & b) == a) { /* ... */ }
/*  ^ Parentheses required here */
```

While many features were added to C in 1972-1973, the preprocessor was likely the most significant. Dennis implemented it partly at the request of Alan Snyder (who had also suggested that Dennis add the `&&` and `||` to make boolean logic more explicit) at MIT in *A Portable Compiler For The Language C*, and partly inspired by the textual inclusion mechanisms in PL/1 and BCPL. The first version could only `#include` other files and do simple textual substitution, though it was soon extended by Mike Lesk and John Reiser to handle macros with arguments and support conditional compilation.

In the summer of 1973, the language and compiler were mature enough that the team was able to rewrite the Unix kernel in C. Ken Thompson had tried once before to port Unix to C, but the language did not support structures at the time, making the task too onerous to continue. The compiler was also ported to the Honeywell 635 and IBM 360 and 370, and commonly used procedures coalesced into what eventually became the standard library. The first of these major library components was Mike Lesk's input/output library, which became the basis of the C *standard I/O*. Dennis and Brian Kernighan wrote the seminal (Kernighan and Ritchie, 1989). Brian wrote most of the exposition and Dennis wrote most of the programs. This would come to be common; Brian's gift for technical writing that was easy to understand made him a common author of documentation and publications.

The subsequent period was filled with language features focusing on type safety and portability as Unix and all the tools they had rewritten in C had to be ported to more and more systems, and much of the codebase remained untyped, harkening back to its B and assembly heritage. In the same vein, pointers and integers were mutually assignable in C, absent of any syntactic indication that the programmer's intent had changed. This practice fell out of style, and *casts* were introduced to the language, inspired by a similar *but distinct* concept in ALGOL 68. Similarly, the member-dereference operation did not care much about the type of the pointer being dereferenced; `ptr->field` could be used regardless of the type of `ptr`, and `field` was taken to mean an offset from some pointer into a structure. [6]

In 1978, Steve Johnson started work on a C compiler that was easy to retarget to other machines while he, Dennis and Thompson worked on porting Unix to the Interdata 8/32. While the language continued to evolve and some older practices fell out of style, Johnson came up with the `lint` program, adapted from his *Portable C Compiler* `pcc`, to search a set of source files for coding practices that might need to be updated. This is one of the first (if not *the* first) static-analysis tool, its namesake lives on in today's tools; one still talks of a *linter* remarking on a dubious construct in one's code.

The third version of Unix, System III, was the first one to be primarily written in C. As Unix matured and its System III and System V versions were distributed, institutions outside AT&T developed Unix distributions (particularly at the University of California at Berkeley, yielding the Berkeley Software Distribution of Unix), and C compilers were ported to many new machines, C became a mainstay of the software world, as it remains today.

[6] Perhaps they should have preserved more of ALGOL 68's type system; these pointer rules plague optimizers working with C programs to this very day. The modern Linux kernel is *not* compiled with a flag usually spelled `-fstrict-aliasing`, which allows the optimizer to take advantage of the C language's aliasing rules. These aliasing rules specify that a pointer cannot refer to multiple incompatible types, and programs that use such behavior are undefined behavior. This flag would theoretically allow the compiler to better optimize the Linux kernel, however, strict aliasing rules are broken so often in the kernel's source that the flag results in a buggy kernel, so it cannot be enabled. Programs that make use of casting to and from `void*` or type-pun through unions are also common in C, and in general the aliasing rules are not very friendly to optimization when compared to Fortran, for example.

4.5.6 Standardization of C

Using pcc as a reference compiler became impractical. The X3J11 committee formed in 1983 to standardize C. Continue with "The Development of the C Language". [TODO: includes early criticism of C.]

4.5.7 SNOBOL

Before we continue with regular expressions and string-processing languages, we must first discuss one of the first languages developed at Bell Labs, and one which most other string-processing languages were inspired by.

SNOBOL would soon be eclipsed by Awk, Perl, Lex, Yacc, and other languages filling similar roles–it gained traction in some universities, but was not used much outside of Bell Labs.

The first publication mentioning SNOBOL was (Farber, Griswold, and Polonsky, 1964); the authors also recieved help from Doug McIlroy, though he is not listed as an author [7] . McIlroy's prior works (like his unpublished paper *String Manipulation System for FAP Programs*) and influence on the development of string processing languages were significant.

SNOBOL was developed to address the limitations of an earlier string-processing language called *COMIT*, the only language with a syntax similar to that of SNOBOL. COMIT did not support named variables or arithmetic, which largely motivated the development of SNOBOL. (Farber, Griswold, and Polonsky, 1964) uses the following program do describe the syntax of SNOBOL:

```
START     VOWEL = "A,E,I,O,U"
V1 VOWEL *V* "," = /F(END)
V2 TEXT   V      = /S(V2)F(V1)
END       START
```

This program removes all the vowels from the string defined by TEXT.

While SNOBOL never gained significant traction, it's important to note the language's legacy and influence on the far more popular string-processing compiler technologies that Bell Labs would produce in the following years. One notable application of SNOBOL outside of Bell Labs was the University of Arizona's PO, an object-code optimizing compiler, discussed in section 5.2.2.2.

4.5.8 Regular Expressions: Grep, Yacc, Lex

Regular expressions were a large part in the explosion of new programming languages and compiler tools at the labs in the early 1970s.

One such use of regular expressions was Ken Thompson's s program ("s" for "search") to do pattern-based searching in his home directory sometime after working on Unix on the PDP-11. In one of those casual drop-ins, his boss McIlroy asked him to come up with a program for finding patterns in files and searching through directories and these things, to which Ken responded "ah, let me think about it overnight" knowing he already had the perfect program. After a night of cleaning up his program, he gave it to Doug who loved the program and moved it from Ken's home directory into the shared binary directory. It was at this time renamed `grep` after the command in the ed

[7] The authors make this remark in the conclusion: "The enthusiasm and numerous suggestions of M. D. McIlroy have been particularly helpful. In addition, his string macro package greatly facilitated the implementation."

editor which performed the same function: `g/re/p`, for global, regular-expression, print. Many small programs like this formed over time at the Labs.

On another occasion, Steven Johnson came into Alfred Aho's office to ask about parsing algorithms for his Portable C Compiler. He was having trouble getting the C parser working properly, so he went to Aho for his theoretical background in parsing algorithms and automata theory and asked how he would go about constructing a parser. Aho replied that first he would construct a grammar for the language, and then apply Knuth's LAR parsing algorithm–Aho got some sheets of cardboard from the stockroom, and did the sets-of-items construction for the grammar that Johnson had given him over the weekend.

On Monday, he would give Johnson the constructed parsing table, Johnson would implement it, and find issues with Aho's cardboard construction. After a few iterations of this, Johnson asked Aho to explain the theory, which he implemented. This program went on to become the `yacc` program, standing for *Yet Another Compiler-Compiler*, which is used for generating parsers from a context-free grammar. Aho and Johnson continued to improve the performance of the program and it vastly decreased the effort required to create a parser for a new language. Johnson's Yacc program was especially useful because it would inform the user of shift-reduce and reduce-reduce conflicts, so they knew if they had any difficult-to-parse constructs in their grammar. Yacc became one of the most widely used tools for creating parsers, even outside Bell Labs in computer science courses. [TODO: Aho's egrep program that constructed the regexes lazily.]

In a mailing list, Johnson recalled that:

> The name cames as a result of a comment by Jeff Ullman – Al was telling him about the program, and he said "What? Yet Another Parser Generator?" So I started calling it Yacc.
>
> The first implementation was really slow. We were all sharing a single PDP 11 with Unix on it, and Yacc could take 20 minutes to generate a 50-rule grammar table. Everyone would groan when i started the program: "Ohhhh. Johnson's running Yacc again".

After Al and Johnson's optimization efforts, the group was eventually able to parse Fortran 77.

One of the fast regular expression algorithms that Aho had come up with also saw use in the `lex` program, which is used for generating lexical analyzers from regular expressions. Eric Schmidt, summer intern at Bell Labs but perhaps better known today as the CEO and co-founder of Google, used one of Aho's algorithms and worked with Mike Lesk of Bell Labs to produce the first version of `lex`. Using these tools in tandem allowed for very tight iteration cycles for designing and developing new programming languages and compilers. With Lex for the lexer and Yacc for the parser, one could quickly develop the front-end of a compiler based only on the grammar for the language.

4.5.9 Applications of Yacc and Lex: Eqn and AWK

In 1974, Brian Kernighan and Linda Cherry developed one of the first programs using Lex and Yacc to develop compilers. It was called `eqn` and was used for typesetting mathematical expressions, usually as a part of *troff* documents. At the time, troff was the state-of-the-art typesetting tool, and it got lots of use at Bell Labs (recall that the Lab's primary output was publications).

Troff documents were marked by beginning and ending characters with a leading perdiod. An Eqn equation in a troff document would start with `.EQ` and end with `.EN`(Kernighan and Cherry, 1975):

```
.EQ
left [ x + y over 2a right ]~=~1
.EN
```

The grammar for Eqn was defined by Yacc rules like so:

```
eqn : box  eqn box
box : text
     { eqn }
     box SUB box  box SUP box
     [ROMAN  BOLD | ITALIC] box
```

This would embed the equation $[\frac{x+y}{2a}] \approx 1$ in the document. This was incredibly useful! Prior to typesetting programs, an author would need to deliver their hand-written notes to a typesetter to produce a draft before sending the typeset drafts to publishers and editors. Placing the typesetting programs in the hands of the authors themselves allowed them to see more or less the final product as they wrote.

Usually, papers would not be typeset until they were already accepted and published. When the Bell Labs employees started writing papers with Eqn and Troff, their papers would be rejected by converences and editors because they "didn't want to publish anything that had already been published elsewhere." Of course these papers had not already been published, but they were so refined that the reviewers assumed they had been.

Kernighan and Cherry wanted to be able to typeset equations in publications by typing something similar to plain-English descriptions, similar to how one mathematician would describe an equation to another over the phone. Al and Steve helped Brian and Linda develop Eqn, and were their first users, along with Richard Hamming and Doug McIlroy. Many of Eqn's features were eventually incorporated into Donald Knuth's TEX typesetting program.

Al recalls that the first Fortran compiler took 18 staff-years to create at IBM, while the folks at Bell Labs were able to produce programming languages and compilers left and right thanks to his new tools and the developments in parsing theory they built upon.

Another significant use of Lex and Yacc that perhaps sees more use today than Eqn is the AWK language, named after its creators Alfred Aho, Brian Kernighan, and Peter Weinberger. At the time, they considered it a throwaway tool and nobody else would be interested in it. It was designed to be a pattern-action language; programs describe a pattern and a corresponding action to be performed on every instance of that pattern in the input.

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

4.5.10 The Dragon Book

Jeffrey Ullman and Alfred Aho continued working together on developing compiler theory and parsing algorithms. Other folks at the Labs would impress on Al that it was important to write about what you were working on to build your reputation in the scientific community, an idea that Jeff also bought into. They decided they ought to write a book together about their parsing techniques in light of how much interest the development of Unix and C had generated. Lots of people were newly

Figure 4.3: Cover of the first edition of the Dragon Book

interested in developing new programming languages and compilers, and they were well-positioned to write about it; thus was *The Dragon Book* born.

> 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. (Alfred V. Aho, 2022)

They published updates to the dragon book over time as well; they co-authored the second edition with Ravi Sethi, another Bell Labs researcher, in the 1980s. The second edition had grown to over 800 pages, and the third edition was published in 2007 at close to a thousand pages. At this point none of the original authors wanted to work on a fourth edition because of how much the field of compiler design had grown since then.

The first edition (with the iconic green dragon in Figure 4.3) had a very heavy focus on lexing, parsing, and semantic analysis, all but omitting optimization techniques. This makes sense because parsing algorithms were Al and Jeff's specialties, but as this book went on to be the seminal compiler textbook for a generation, many student's primary exposure to compiler design was almost entirely focused on front-end design. Of the 600 or so pages in the first edition, only about 100 were dedicated to optimization and data-flow analysis (Alfred V. Aho and Jeffrey D. Ullman, 1977, Chapters 12, 13, and 14).

4.5.11 Trusting Trust

"Reflections on trusting trust".

4.5.12 Standard ML at Bell Labs

Just as Aho and Ullman were collaborating on compiler technologies at Bell Labs and Princeton respectively, so too were David MacQueen and Andrew Appel working on Standard ML at the same respective institutions.

4.5.13 C++

Bjarne wanted simula 67 stuff in C. Started C with classes, built on the C preprocessor, which allowed all the other Bell people to use his stuff since it dropped right into C. Eventually needed to do more than the C preprocessor could handle, so he built a new preprocessor *cfront*, which grew to become a complete and distinct compiler. (Stroustrup, 1995). David Macqueen interview had interesting perspectives on this.

4.6 Type Theory

(Cardone and J Roger Hindley, 2006) summarizes the importance of λand CL to compilers and programming langauges as follows:

> λ and CL are used extensively in higher-order logic and computing. Rather like the chassis of a bus, which supports the vehicle but is unseen by its users, versions of λ or CL underpin several important logical systems and programming languages. Further, λ and CL gain most of their purpose at second hand from such systems, just as an isolated chassis has little purpose in itself.

In section 3.7, we discussed the early foundations, and in this section we discuss the applications.

Building on the foundations laid by John McCarthy, Alonzo Church and Haskell Curry, functional programming came into its own with the development of Meta Langauge. In chapter 3, we covered John Backus's work on Fortran extensively; what we didn't cover was his famous 1977 ACM Turing Award lecture, "Can programming be liberated from the von Neumann style? a functional style and its algebra of programs", in which he tears down his own language's contributions to the field of programming languages, favouring instead a functional and expression-oriented approach to writing programs. He contrasts the way one must reason about programs written in procedural languages like ALGOL and Fortran with those written in functional languages like APL, which we will explore in this section. This paper contributed to the eventual zeitgeist of functional programming which would dominate some areas of the field for many years to come, especially in academia.

4.6.1 Economic Model of Developments in Functional Programming

One framework for understanding trends in programming language design goes like this:

1. Something happens in the world that results in lots of money being spent on programming languages and software.

2. Lots of academics use this money to think about how to design programming languages.

3. They converge on mathematical (in particular, algebraic) approaches to programming languages design, and these ideas gain traction.

4. That money dries up, and software developers generally move away from algebraic approaches to programming languages design, and sometimes go in the *opposite* direction towards practicality at all costs.

I don't necessarily believe this and macro trends are hard to prove one way or another, but sometimes it's a useful lens.

Perhaps the first instance of this trend was Laning and Zierler's algebraic compiler which formed during and immediately after the Second World War, when computing was in its infancy and there was lots of government funding for research into programming languages. This culminated in the development of ALGOL with it's principle of orthogonality of language features. This line of thinking eventually gave way to more practical and less principled approaches to language design found in Fortran and C.

Another example might be the development of ML... [TODO: ...]

4.6.2 Peter Landin's ISWIM

In section 3.7 we established the λand its early applications to compilers in Lisp, and in section 3.6.2 we discussed Landin's use of λto formalize the semantics of ALGOL 60. Like many of the prominent British computer scientists of the time, he was heavily involved in the development of ALGOL, and then took that experience to further programming language design efforts. Landin's work continued to be highly influential, in particular his (P. J. Landin, 1966).

In this paper, he describes a general framework for programming languages called *ISWIM*, standing for *If You See What I Mean* [8], which conveyed the semantics of λwith a particularly elegant syntax over λconstructs. This language represented his vision for the future of programming languages with an emphasis on expressing the programmer's intent uncluttered by the details of the machine running the program.

> Most programming languages are partly a way of expressing things in terms of other things and partly a basic set of given things. The ISWIM (If you See What I Mean) system is a byproduct of an attempt to disentangle these two aspects in some current languages.
>
> ISWIM is an attempt at a general purpose system for describing things in terms of other things, that can be problem-oriented by appropriate choice of "primitives." So it is not a language so much as a family of languages, of which each member is the result of choosing a set of primitives.

Landin described the grammar of ISWIM in informal English, which was perhaps a step backwards from John Backus and Peter Naur's formal grammars (J. W. Backus, F. L. Bauer, Green, Katz, J. McCarthy, Perlis, Rutishauser, Samelson, Vauquois, Wegstein, A. van Wijngaarden, Woodger, and Peter Naur, 1960) which were also being developed for the specification of ALGOL, though the strict evaluation semantics and emphasis on expressivity made the conception of λas a programming language more concrete.

[8] ISWIM is sometimes pronounced *eye-swim* (MacQueen, 2025).

While ISWIM was not statically typed, Landin did describe an informal way to describe data types in ISWIM, which he used for describing the data structures used to represent the syntax of the language. The only true implementation of ISWIM as it was in the paper (aside from the prototype Landin *mentioned* in (P. J. Landin, 1966)) was (Evans, 1968) developed by Evans at MIT.

[TODO: was dynamically typed, much like lisp but with semantics closer to λ.]

4.6.3 Christopher Strachey

Christopher Strachey, another British computer scientist, played a significant role in the development of compiler and programming language theory. In a series of lectures in 1967 (Christopher Strachey, 2000), he introduced the concepts of l-values and r-values.

When one reads of the history of ALGOL, the temptation is the anachronistically assume the concepts modern students of compilers are familiar with were also clear to the developers of the ALGOL standard. Strachey points out how ill-defined many of these concepts were, and attempts to define many of them more precisely in the aforementioned lectures.

> The difficulty is that although we all use words such as 'name', 'value', 'program', 'expression' or 'command' which we think we understand, it often turns out on closer investigation that in point of fact we all mean different things by these words, so that communication is at best precarious.

CPL, standing for *Combined Programming Language*, was like many of the programming languages developed in the wake of ALGOL: it was heavily based on the concepts therein, but sought to extend those concepts and make the language more practical, in light of ALGOL's lack of adoption and lack of a useful compiler or programming environment.

> CPL is based on, and contains the concepts of, ALGOL 60...However, CPL is not just another proposal for the extension of ALGOL 60, but has been designed from first principles and has a logically coherent structure. (Barron et al., 1963)

(Barron et al., 1963). (Richards, 2000). (Christopher Strachey, 2000). (Peter J Landin, 2000). (D. Scott and C. Strachey, 1971).

(MacQueen, Harper, and Reppy, 2020) notes how most of the characters in this story were in part brought together by a shared interest and an interested amateur who noticed them all reading similar materials in the library. In this way, many of the early meetings formalizing the language were informal and poorly recorded.

> It is interesting to note that most of the central personalities first met through an unofficial reading group formed by an enthusiastic amateur named Mervin Pragnell, who recruited people he found reading about topics like logic at bookstores or libraries. The group included Strachey, Landin, Rod Burstall, and Milner, and they would read about topics like combinatory logic, category theory, and Markov algorithms at a borrowed seminar room at Birkbeck College, London. All were self-taught amateurs, although Burstall would later get a PhD in operations research at Birmingham University. Rod Burstall was introduced to the lambda calculus by Landin and would work for Strachey briefly before moving to Edinburgh in 1965. Milner had a position at Swansea University before spending time at Stanford and taking a position in Edinburgh in 1973.

4.6.4 Strachey and Landin Together

(Peter J Landin, 2000). (INRIA, 2019).

4.6.5 LCF at Stanford

LCF, standing for *Logic for Computable Functions*, was a proof assistant program which attempted to translate the formalized logic for computable functions from (D. S. Scott, 1993) into λ, originally developed by Milner at Stanford (Milner, 1972b). This program, originally implemented in Lisp on the PDP-10, would continue to be developed at the University of Edinburgh when Milner moved there to study in

4.6.6 The History of Standard ML

(MacQueen, Harper, and Reppy, 2020) described the origins of the Meta Language in Landin's paper like so:

> The basic framework of the language design was inspired by Peter Landin's ISWIM language from the mid 1960s, which was, in turn, a sugared syntax for Church's lambda calculus. The ISWIM framework provided higher-order functions, which were used to express proof tactics and their composition. To ISWIM were added a static type system that could guarantee that programs that purport to produce theorems in the object language actually produce logically valid theorems and an exception mechanism for working with proof tactics that could fail. ML followed ISWIM in using strict evaluation and allowing impure features, such as assignment and exceptions.

4.6.7 Meta Language

"If you had to choose between a surgeon that was theoretically proven to be perfect, or a surgeon that successfully performed 10,000 operations without making a mistake, which one would you choose?"

4.6.8 Standard Meta Language of New Jersey

University of Endinburgh; David MacQueen. *Bell Labs.*

4.6.9 Caml

Inria

4.7 APL

APL, standing for *A Programming Language*, was developed by Kenneth E. Iverson in 1962 and partially implemented on the IBM System/360 as APL/360. While this partial implementation was only an interpreter for a subset of the language Iverson designed, it would spur on the development of a family of programming languages called *array programming languages* or *Iversonian languages*. *Programming Languages: History and Fundamentals*, Section X.4.

[TODO: dig into ML languages (MacQueen, Harper, and Reppy, 2020)] . [TODO: type systems, type inference, Hindley Milner, SML.] [TODO: similar vein to Laning and Zierler's algebraic compiler.]

4.8 Programming Languages and Their Compilers

probably not worth a full section here. [TODO: *Programming languages and their compilers: Preliminary notes*] .

4.9 Seymore Cray

> The CDC 160 and the Origins of the Minicomputer
> The Whirlwind (a computer prototype built at MIT) had a word length of only 16 bits, but the story of commercial minicomputers really begins with an inventor associated with very large computers: Seymour Cray. While at UNIVAC Cray worked on the Navy Tactical Data System (NTDS), a computer designed for navy ships and one of the first transistorized machines produced in quantity. Around 1960 Control Data, the company founded in 1957 that Cray joined, introduced its model 1604, a large computer intended for scientific customers. Shortly thereafter CDC introduced the 160, designed (P. Ceruzzi, 2000)

4.10 The DEC VAX and the IBM System/360

> Through the 1980s the dominant mainframe architecture continues to be a descendent of the IBM System/360, while the dominant mini was the DEC VAX, which evolved as a 32 bit extension of the 16-bit PDP-11. (P. Ceruzzi, 2000)

4.11 Commercialization

Once IBM split off a new group to sell software, it started to become evident that entire business models could be built off of software. Though few understood it at the time, the software business model would become among the most lucrative in the world due to the nature of software.

If a carpenter designs and crafts a chair, then they can sell that one chair, and one person gets value from it. When a customer goes to a resturant and pays the waitstaff and cooks for their time and the ingredients, they get one meal's value out of their labor. Software is fundamentally different. Rather than providing a good or a service, the process of writing software is far closer to writing sheet music. A programmer can write out a blueprint, or a script, which defines actions from which someone may derive value, and that blueprint may be replicated once, twice, or a trillion times.

Of course there are maintenance costs associated with software, and at some point, someone has to manage the hardware as well. But fundamentally, the potential value someone can derive from software has a very high upside relative to the labor it takes a programmer to produce it. It is in this time period that a few companies started to discover the commercial value of selling software.

4.11.1 Microsoft

Microsoft would come to be so astronimically large that it is almost easy to lose sight of their impact. Because they constitute an entire developer ecosystem, their influence sometimes does not escape the walls of their garden.

[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;]

4.11.2 Sun Microsystems

Sun had three core folks that made their business special: a brilliant engineer, software architect, and marketing... Bill Joy Spark and Solaris.

[TODO: oxide and friends episode: Supercomputers, Cray, and How Sun Picked SGI's Pocket] (Salus and Reed, 2008).

4.12 Compilers in the US National Laboratories

(Smotherman, 2023).

4.13 Timeline

[TODO:]

5
Collaboration, *1980-2000*

5.1 Short History of Open Source

The most significant feature of this time period is the collaboration brought about by the advent of the free software movement. In the beginning of software development, much of it *was already* given away to customers of hardware companies like IBM. When the software was already in machine code, there was not much of an alternative to giving your source code away to your customers. You had a deck of punchcards with your software, and to get that to your customers, you had to punch out a new copy and mail it to them; other than the accompanying documentation, you *had* to ship the code for your programs, and there was nothing in between your programs and the runnable program.

Hardware companies would give the software away for free with the hardware because the cost of the software was negligible relative to the cost of the hardware. The turning point in this model of software distribution came in 1969 when IBM broke some of their software off to be sold separately. Prior to this, perhaps the first instance of free software was in Grace Hopper's development of the A-2 compiler (see Section 3.3.5), for which the Remington Rand company accepted changes and suggestions their customers sent them. When software demanded a market of its own, businesses were able to support themselves solely by selling software, and it became the core business value of the company. This worked against the free distribution of software to a large extent.

Shortly after Unix was developed at Bell Labs in the early 1970s, its source code was distributed under a license that would today be considered similar to FOSS. While the source code of Unix was only given to universities for academic use and only users covered by the license were technically allowed to see the source, large communities began to form around it, namely USENIX. Some entirely new distributions and rewrites of Unix formed, most notably the Berkeley Software Distribution (BSD), which was developed at the University of California, Berkeley.

[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.(P. E. Ceruzzi, 2003)

5.2 GNU

In the 1980s, business models centered on software had grown substantially and the Unix programming environment began permeating the industry, bringing the C programming language along with it. The commercialization of software has obvious advantages and disadvantages: it is convenient that people can make a living off of writing software, but it often stifles the collaborative environment that compilers and programming languages grew out of. The reaction to the commercialization of software was strong and wiped out many companies and teams within companies dedicated to compiler development while centering the industry around a standard, free set of tools.

5.2.1 Richard Stallman

Richard Stallman attended Harvard in 1970, pursuing a bachelor's degree in physics. At the end of his first year there, he began attending the MIT Artificial Intelligence Lab, where Lisp was developed (section 3.7.2). After graduating from Harvard in 1974, he joined the MIT lab as a graduate student. This only lasted for a year, at which point he dropped out and began working at the lab full-time (Gross, 1999).

Shortly after he began working full-time, many of the original hackers that drew him to MIT began starting their own companies and hiring away the other hackers at MIT. The two most significant companies were Symbolics and *Lisp Machines, Inc.* (LMI), both set out to build Lisp machines and Lisp-based operating systems based on their founder's experience working with Lisp at MIT.

Both companies attempted some level of collaboration with MIT, to the extent that Symbolics and LMI both *continued to use MIT's machines and source code*, which led to intellectual property disagreements. Symbolics eventually reached an agreement with MIT such that MIT would continue to benefit from Symbolic's Lisp machine developments *only as users*, and members of MIT were not allowed to view Symbolic's source code.

In an effort to maintain MIT's hacking culture, Stallman sought to replicate [1] Symbolics' Lisp machine developments as they came out, such that the software available to MIT would keep feature parity with that of the commercial products. This benefited LMI, since they were allowed to see and use MIT's Lisp code. Stallman [2] kept LMI in business out of spite for Symbolics, because he percieved them to have taken away his community.

5.2.2 Birth of GNU and GCC

Stallman tired of attempting to punish Symbolics and began considering what he should do next. By mid-1983, Stallman had considered building a new community around a free operating system similar to Unix, GNU, for *GNU's Not Unix*. He no longer worked for MIT, but he was still able to use their systems.

While the first and primary program he worked on for GNU was the GNU Emacs editor (which he rewrote after developing the original Emacs at MIT, which he took over from Guy Steele (R. Stallman, 2002, footnote 1)), the GNU C compiler followed quickly.

[1] Initially, Stallman read Symbolics' source code instead of replicating their features from scratch; Weinreb understood Stallman's prior statements to mean that he did not read Symbolics' source code, which he pointed out in (Weinreb, 2020), and Stallman left a footnote in (R. Stallman, 2002, footnote 7) noting that Symbolics' source code was made available to MIT and he re-implemented their features before eventually deciding against even reading it.

[2] Again, this is Stallman's word against Weinreb's. There are many conflicting accounts.

Von Hagen notes that the idea for GCC actually predates the wider GNU project:

> In late 1983, just before he started the GNU Project, Richard M. Stallman, president of the Free Software Foundation and originator of the GNU Project, heard about a compiler named the Free University Compiler Kit (known as VUCK) that was designed to compile multiple languages, including C, and to support multiple target CPUs. Stallman realized that he needed to be able to bootstrap the GNU system and that a compiler was the first strap he needed to boot. So he wrote to VUCK's author asking if GNU could use it. Evidently, VUCK's developer was uncooperative, responding that the university was free but that the compiler was not. As a result, Stallman concluded that his first program for the GNU Project would be a multilanguage, cross-platform compiler. (Von Hagen, 2011)

GCC originally stood for the GNU C Compiler, but as the GNU project grew, new frontends were added for other languages, thus GCC now stands for the GNU Compiler Collection. There were two primary software projects that Stallman based the original GCC on: the Pastel compiler, and the Portable Optimizer.

5.2.2.1 The Pastel Compiler

Stallman recalls in (R. M. Stallman, 2025) that he thereafter obtained the code for a cross-platform compiler from Lawrence Livermore National Laboratory (LLNL) called *Pastel*, because it compiled (and was written in) an "off-color Pascal," as the authors described it to Stallman (R. M. Stallman, 1986).

Pastel was the compiler and programming language of choice for the Amber operating system (a competitor to Multics) being developed at LLNL for the S-1 computer (a competitor to the Cray-1). This OS was originally written in PL/1, but the language was found to be inadequate; the Amber developers originally extended PL/1 with macros to improve the type definition system, but after struggling for about 6 months with the limitations of PL/1's type system and module system, and the high financial cost of the G PL/1 compiler, LLNL decided to use another language (Frankston, 1984).

Jeff Broughton, the leader of the Amber project, wrote two compilers for Pascal, extending the language as he saw fit. The first of these compilers was completed in a few months, and targetted PL/1 instead of S-1 machine code so the compiler could be used while the native backend was still being developed. The existing PL/1 codebase was ported to Pastel over the course of a few months. It was around this time that Stallman visited LLNL and learned of the Pastel compiler. A historical note from 1998 in Frankston's Bachelor's thesis makes note of this interaction (Frankston, 1984):

> [A]t one point in the project Richard Stallman visited, and had the Pastel compiler explained to him. He left with a copy of the source, and used it to produce the Gnu C compiler. Most of the techniques that gave the Gnu C compiler its reputation for good code generation came from the Amber Pastel compiler.

Pastel was an interesting language and compiler in its own right; the compiler supported parameterized types and complicated features that would be handled by template metaprogramming in the equivalent C++ program. Stallman points to the example of strings: in Pastel, a programmer could specify a `string` if they wanted a dynamically sized string, or `string(n)` if they knew the

length ahead of time. The compiler could then store the string in static memory and reuse it with each function call, saving an allocation and deallocation.

He began to add a C frontend and a Motorola 68000 backend to this compiler, but he ran into some issues stemming from the extended Pascal it was written in. This version of Pascal did not require users to forward-declare functions, meaning the compiler had to process the entire file to find the declarations for every function all at once, consuming an amount of memory proportional to the size of the file being compiled. Stallman was working on what he called "a horrible version of Unix" (R. M. Stallman, 1986) with needlessly limited stack space, which further complicated the use of this compiler. When he attempted to perform liveness analysis of temporary values (where the compiler determines which values are used by the program in a given region) he needed a quadratic matrix of bits, which took up to several hundred kilobytes for large functions, which ran up against the onerous memory limits imposed by this version of Unix.

5.2.2.2 The Portable Optimizer

Stallman also took inspiration from the University of Arizona Portable Optimizer especially in store-to-load forwarding and strength reduction. This optimizer was also called PO, which stands for *peephole optimizer* and not *portable optimizer*.

This optimizer was interesting in large part because it operated on *object code*, and was designed to be portable across different architectures. The machine description for each architecture supported by the compiler was a Lex program that recognized instructions and their properties, and the rest of the compiler would use that information in a relatively machine-independent way. As of 1980, PO was written in only five pages of SNOBOL–a curious language discussed in section 4.5.7.

In this USENET discussion, Christopher W. Fraser responds to a question about why GCC uses register transfer language (RTL) for an intermediate representation instead of simple Lisp tuples, connecting GCC to its history with PO:

> GCC is based in part on PO, a retargetable peephole optimizer that Jack Davidson and I developed at the University of Arizona starting in 1978…
>
> Most peephole optimizers operate on machine instructions, and they need to know what the instructions do. A machine-specific peephole optimizer can use a machine-specific representation for instructions, and the programmer can burn into the optimizer machine-specific information about the effects of instructions. A retargetable peephole optimizer, however, needs a machine-independent way to represent the effects of machine-specific instructions. (If that sounds like a contradiction, consider C: one can write machine-specific C programs, but the language itself is machine independent.) Register transfers are such a representation. (Christopher W. Fraser, 1990)

RTL is discussed at length in (J. W. Davidson and Christopher W Fraser, 1984). GCC continues to use RTL in its backend, as well as machine-specific peephole optimizations initially developed by Jack Davidson and Chris Fraser. PO represented instructions symbolically; for example, `r[4]=r[4]+1` increments the value in register 4 by 1. Stallman used s-expressions to represent the same information. The previous example would be `(set (reg 4) (+ (reg 4) (int 1)))` in Stallman's representation.

This new representation also had an infinite number of pseudo-registers which get converted to real registers and stack slots by a legalization phase. The part of the compiler generating the

IR does not have to concern itself with register allocation. This legalization phase also performs liveness analysis to determine which pseudo-registers are in use in the same region, implying that they cannot go in the same physical register.

Many other issues needed to be addressed in cleanup passes too, especially when dealing with machine-specific details. For example, on the 68000, you cannot make the output of an operation a memory location, so to add two values stored in memory, you must first store one of them to a register, then perform the addition, and finally store the result back to memory.

Many of these issues were solved or being solved by other teams in the same time period [TODO: graph coloring algorithm, ask Sanjin] .

PO also performed some rudimentary control-flow graph simplification and dead code elimination by identifying and simplifying redundant nodes in the CFG, particularly branch chains [TODO: could expand on this...] . These optimizations are critical in modern compilers because many optimizations rely on the IR being in a normal form to be maximally effective.

5.2.3 The Kernel

Linux was started by Linus Torvalds after Stallman began work on GNU, so as far as Stallman knew, he still needed to come up with a kernel to provide a complete, free operating system capable of competing with Unix. The GNU community considered two existing projects for the GNU operating system kernel before arriving at Linux: in 1986, Stallman planned on using Trix, which was a research project from MIT; by 1990, he planned to build a Unix-compatible OS layer called GNU Hurd [3] on top of the Mach microkernel, designed at Carnegie Mellon University (and later at the University of Utah) (R. M. Stallman, 2025).

Linux Torvalds fortunately started developing Linux in 1991, and released it as free software in 1992, permitting the combination of GNU and Linux. Linux, compiled with GCC and paired with the other GNU utilities formed the Unix-compatible free operating system (and the community around it) that Stallman had sought after.

5.2.4 Cygnus and GCC

In 1986, Stallman finally decided to throw out this Pastel compiler and start over, using his experience with Pastel and the University of Arizona's optimizer to build a new C compiler (though he did re-use the C frontend from his modified Pastel compiler).

Throughout the 1980s, progress on GCC was relatively slow because Stallman prioritized developing Emacs. In the 90s though, the GNU project piqued the interest of more developers who began contributing to the compiler. The company Cygnus Solutions made significant contributions and built their entire organization around the GNU project. Their business model was so innovative and impactful to the market for compilers and compiler engineers that the company's history deserves its own section.

At this point, there were loads of proprietary compilers from virtually every hardware company. The embedded systems market supported many small compiler companies since the larger companies were primarily focused on compilers for their flagship. These smaller companies charged high markups for their compilers and expensive "seats" that companies had to purchase for every single

[3] Initially called *Alix* after Stallman's then-girlfriend, a system administrator at MIT.

developer that used their compilers (sometimes up to $10,000 in 1990s dollars). This environment was ripe for disruption–and that was exactly what Cygnus' intended to do.

Cygnus Solutions was founded in 1989 by Gilmore, Tiemann, and Henkel-Wallace in Palo Alto, California, at which point GCC compiled itself and produced pretty good machine code for the DEC Vax and the Motorola 68000. Linux would not be created and joined with the GNU project until years later. The founders thought they might be able to replace the compiler departments at larger companies like Sun, SGI, and DEC in favor of paying Cygnus for support contracts. While the larger companies did not adopt GCC or purchase Cygnus' support contracts, they did find product-market fit with companies working on embedded systems.

Cygnus put significant efforts into making GCC a robust cross-compiler to better support these embedded systems. Sony and Cisco were their first large customers. For Sony, their support contract with Cygnus for cross-compilers and emulators for their game consoles meant that game developers could start developing games for an upcoming console long before the game hit the market, allowing their customers to get more and better games, earlier than the competition.

For the customers, Cygnus' value proposition was favorable compared to the proprietary compilers they could buy from the smaller compiler companies; for instance, if Sony didn't like Cygnus' support after a couple years, they could always take GCC and modify it to their liking. Cygnus had to earn their keep every single development cycle. Additionally, it lowered the bringup cost for new chips, since they didn't have to throw away the compiler every time they wanted to significantly change the architecture. They could simply send patches to the GNU project to add support for their new chips, and the compilers and debuggers they were used to would continue to work.

With this business model, Cygnus obliterated many small compiler companies. Many of them were bought out by their customers, and many of their customers moved to open-source compilers even if they didn't purchase support contracts from Cygnus. This pattern was primarily restricted to the embedded market since the large companies could afford to keep many compiler engineers in-house and produce compilers that genuinely out-performed their open source alternatives, but this trend would continue to grow.

Cygnus employees continued to contribute to the GNU project and put small proprietary compiler companies out of business, and their influence in the GNU project continued to grow. They pioneered many features in GCC, including more complete support for C++ than existed in any other compiler at the time, including Bjarne's own at Bell Labs. In (Gilmore, Tiemann, and Henkel-Wallace, 2006), Gilmore claimed that Michael Tiemann's C++ compiler was the first true C++ compiler:

> Our C++ compiler, g++, was the first true compiler for C++, producing assembler code rather than translating C++ into C. It implements all the features of AT&T C++ 2.0. It runs on any machine that gcc supports, and does the same optimizations. It also provides additional C++ specific optimizations...[Michael Tiemann] is the author of GNU C++, the first available native code C++ compiler.

In the 1990s, a subset of the GCC contributors (most of whom were employed by Cygnus) began releasing *EGCS*, or the Experimental/Enhanced GNU Compiler Suite. This was the first and last major fork of the GCC project to date [4]:

[4]The Pentium Compiler Group also had a fork, but while EGCS was separately maintained, PGCC (not to be confused with the *Portland* Group C Compiler) closely tracked the EGCS code base, and development of the project seemed to have died out once EGCS was merged back into GCC.

> EGCS was intended to be a more actively developed and more efficient compiler than GCC, but was otherwise effectively the same compiler because it closely tracked the GCC code base and EGCS enhancements were fed back into the GCC code base maintained by the GNU Project. Nonetheless, the two code bases were separately maintained. In April 1999, GCC's maintainers, the GNU Project, and the EGCS steering committee formally merged. At the same time, GCC's name was changed to the GNU Compiler Collection and the separately maintained (but, as noted, closely synchronized) code trees were formally combined, ending a long fork and incorporating the many bug fixes and enhancements made in EGCS into GCC. This is why EGCS is often mentioned, though it is officially defunct. (Von Hagen, 2011)

The timing could not have been much better for Cygnus and GCC, as just after the company was started, Sun decided to break their compiler software out of the their existing software bundle (thereby increasing the price). This 1989 marketing document (Sun Microsystems, Inc., 1989) tries to put a positive spin on this decision:

> Sun Offers New Unbundled C Compiler In addition, Sun introduced a new product, Sun C 1.0 — its first C compiler sold separately from SunOS , Sun's UNIX operating system. By unbundling the compiler, Sun can provide more frequent updates and enhancements independent of operating system releases. A version of the C compiler will continue to be bundled and supported with SunOS, but feature enhancements will be made to the unbundled version only.

At the time, Sun sold compilers for C, FORTRAN, Pascal, Modula-2 and C++, but only C was sold separately from SunOS. This left their users with an easy decision–start paying more (roughly $2000 for each user of each compiler), or start using and contributing to the new GCC compilers. All that was needed was a Sparc backend for GCC–which it already had.

Peter Salus recalls in (Salus and Reed, 2008) that the FSF saw far more orders of CD copies of GCC after Sun's decision. Their compilers cost only $45 each.

[TODO: sun had to sell their products internally too, that's why they charged for it bryan cantril.]

[TODO: In 1986, Stallman's compiler still generated debugging information in the DBX debugger's format, but could also use GDB's internal format. There were numerous compiler tools developed in this era, beginning with the *lint* tool (discussed in section 4.5.5); we discuss these compiler tools in detail in section 5.4.]

5.3 Fortran

We have already discussed Fortran in detail in the previous section; we will continue to discuss Fortran's impact in every section. Of the first programming languages, Fortran and Lisp have had the most longevity, and Fortran programs tend to do more work today than Lisp programs, in general. Many critical applications in science and engineering are written in Fortran, and Lisp, while it is still around and used in some contexts, is not nearly as widely used.

5.3.1 Fortran V (or 66)

5.3.2 Fortran 77

5.4 Tools

Starting with the *lint* program for...valgrind, purify, DBX, GDB.

5.5 Ada

ADA was a cool language because it was designed from the ground-up to solve a particular problem– it wasn't designed by people thinking big thoughts in a committee, but by people hired to do a job. So it was really well-specified. One manual for the spec, one manual for the interpretation.

"The rise, fall and persistence of Ada".

5.6 Should Your Language Be Typed?

When Lamport et al. ("Should your specification language be typed") presented on these concepts, people were so offended - Backus's position in "Can programming be liberated from the von Neumann style? a functional style and its algebra of programs" had gained lots of influence in some circles, so the suggestion that we should *regress* into a less structured programming language was antithetical.

If you look at the programming languages that would reign for the decades to come however, you might end up agreeing with Lamport. Python and C and JavaScript have been seeing more and more adoption in spite of their (lack of) type systems.

5.7 Timeline

[TODO:]

6
Codesign, *2000-2025*

In each age, contemporaries have attempted to place their work in the greater 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 this task will be much easier at a future time looking back on the past. Nonetheless, this is what I will attempt to do in this chapter.

> 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. (Alfred V. Aho, 2022)

6.1 What Does Codesign Mean?

I primarily focus on two issues in this period, and how compiler design aims to address them:

1. More so than in previous periods, software and hardware must be designed together to give users the best performance.

2. There is a diverse ecosystem of compiler tools and programming languages that do not interoperate.

Prior to multicore CPUs, software did not have to change much to get the best performance from newer CPUs–programmers could rely almost entirely on hardware manufacturers to deliver them the best performance simply by making the chips faster. With the advent of multicore CPUs and SIMD instructions in the early 2000s, this stopped being the case.

For a programmer to get the best possible performance from a CPU that supports SIMD instructions, they must either rewrite and possibly restructure their application to explicitly use SIMD instructions, or they must rely on automatic vectorization, where the compiler infers parallelism by analyzing the user's program and generating SIMD instructions automatically.

There are two issues with SIMD, both stemming from the lack of cooperation between the design of software and hardware. Explicit use of SIMD instructions is difficult for the programmer, and generally represents a step *backwards* in the evolution of compilers and programming languages. Programming languages were developed *so that* users no longer needed to write (and rewrite) their applications for each new machine they wanted to target. John Backus through users should not

need to worry about index registers and floating-point units (or the lack thereof), but should instead program in a language with a compiler that took care of it for them–and here we are in the 21st century asking users to *go back* to writing assembly to get the best performance from their CPUs.

Automatic vectorization was (and sometimes still is) unreliable, though lots and lots of programs benefit greatly from it. When auto-vectorization works, it can seem almost magical–the user simply recompiles thier code with a better compiler, and suddenly they can take advantage of all the performance of the hardware without changing their code at all. The problems come when it fails.

Compilers tend to be opaque black-boxes to users, making it difficult for users to understand why their code was not vectorized, or even whether or not it was vectorized. Compilers do often have debugging tools or ways for users to inspect the compiler's analysis of their program, but even for compiler engineers, this investigation can be difficult and time-consuming.

Users have to grapple with not only the *feasability analyses* in the compiler (which the compiler uses to determine *if* the program has implicit parallelism that can be exploited with SIMD instructions), but also *profitability analyses*, which the compiler uses to determine if the SIMD version of the user's program is likely to run any faster than the non-SIMD version. If a user wants to debug the failure modes of auto-vectorization, they have to understand the intricacies of the compiler's analyses, know how to get those analysis results from the compiler, and how to change their code to better take advantage of automatic vectorization. This is a leaky abstraction, because the users suddenly need to understand the compiler at a deep level, instead of relying on the compiler to do its job. [TODO: compilers are not debuggable, macros, special flags, leaky abstractions. sometimes its good for certain users though.]

(C. Lattner, 2021). (C. Lattner and Minsky, 2025).

6.2 Chris Lattner

Chris Lattner's impact on the landscape of compiler technology in the 21st century can't be overstated. His work on LLVM has more or less beat out every other compiler technology. We will now discuss the stories of the technologies that Chris developed contiguously, with soliloquies for uses of those technologies that Chris was not involved in.

6.2.1 Low-Level Virtual Machine

In December of 2000, Chris began work on LLVM with his advisor Vikram Adve as part of his PhD research at the University of Illinois at Urbana-Champaign. LLVM stood for Low-Level Virtual Machine at the time but is no longer an acronym and is simply the name of the project. At the time of the publication of his thesis (C. Lattner, 2005), the umbrella project initially contained only an intermediate representation, an optimizer for that IR, IR-level linking, and both offline compilation and online compilation for code generation. Today, these components are all part of the LLVM *subproject* within the LLVM *umbrella project*, which contains other compiler libraries and tools.

> This chapter describes LLVM — Low-Level Virtual Machine — a compiler framework that aims to make lifelong program analysis and transformation available for arbitrary software, and in a manner that is transparent to programmers. LLVM achieves this through two parts: (a) a code representation with several novel features that serves as a common representation for analysis, transformation, and code distribution; and (b) a

> compiler design that exploits this representation to provide a combination of capabilities that is not available in any previous compilation approach we know of. (C. Lattner, 2005)

While LLVM is perhaps best-known for some of the specific technologies contained in the umbrella project, its most novel features lie in the compiler architecture. The IR was more flexible and language-agnostic than the other contemporary IRs, but the nature of LLVM as a set of *libraries* that can roughly be used independently of each other. The tools developers know LLVM for (like Clang, LLD, and LLDB) are really thin main programs that simply call into the libraries for parsing C code, optimizing a chunk of LLVM IR, or generating machine code for that LLVM IR. No prior art provided this level of flexibility, and LLVM's IR, terminology, and interfaces have become the lingua-franca of the compiler world.

> While LLVM provides some unique capabilities, and is known for some of its great tools (e.g., the Clang compiler, 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. (A. Brown and Wilson, 2011, Section 11. LLVM)

Here we continue discussion of Chris's career and discuss LLVM itself in greater detail in section 6.3.

6.2.2 Chris and LLVM at Apple

6.2.2.1 Clang

6.2.2.2 Swift

Date	Organization	Notes
2022–	Modular AI	CEO/co-founder, Mojo programming language and compiler.
2020–2022	SiFive	Started LLVM CIRCT project applying MLIR to chip design.
2017–2020	Google	Started the MLIR project, worked on TPU compilers including XLA.
2017	Tesla	
2005–2017	Apple	Major expansion of the LLVM project. See Section 6.2.2.
2000–2005	UIUC	Designed foundations of LLVM.

Table 6.1: Timeline of Chris Lattner's career (C. Lattner, 2025)

6.2.3 Chris and MLIR at Google

6.2.4 Mojo

6.3 LLVM

LLVM was introduced in section 6.2.1 as part of Chris Lattner's history, but LLVM is a large and diverse project and it deserves discussion outside of the context of Chris Lattner.

Subproject	Date Added	Notes
LLVM Core	2000	Project inception at UIUC, see Section 6.2.1.
Clang	2006	Chris starts work on Clang, see Section 6.2.2.1.
compiler-rt	~2009	Compiler runtime libraries, drop-in replacement for `libgcc`
Clang	~2009	Clang supports C and Objective-C.
LLDB	~2010	LLDB shipped as part of Xcode 4.
Clang++, libc++	~2010	Clang now supports C++; C++ standard library added as LLVM project.
lld	~2011–2015	LLVM linker
libclc	~2012	OpenCL standard library
OpenMP runtime	~2013–2014	Added as LLVM's OpenMP runtime
MLIR	2019	[TODO: see section on MLIR]
Flang	2020	[TODO: see section on Flang]
llvm-libc	2020s	

Table 6.2: Approximate dates major LLVM subprojects joined the umbrella project.

6.3.1 Architecture of the Project

LLVM is an *umbrella project* containing many *subprojects* that can be used independently.

The first (and maybe only) interaction most non-compiler-engineers have with LLVM is through the frontends, like Clang for C, C++, Objective-C, or Swift. This betrays the complexity and elegance with which the pieces of LLVM fit together. These frontends are typically wrappers around the LLVM command-line parsing library, a library to perform the lexing, parsing, semantic analysis and whatever else the frontend for that specific language is expected to do (like module dependency analysis in Swift, for example), and then various other LLVM subprojects.

Most of the time, these other subprojects include pieces from the LLVM subproject, which does optimization and code generation (among other things).

[TODO: refer to (A. Brown and Wilson, 2011, 11.4.2. LLVM is a Collection of Libraries)]

6.3.2 The IR

Lattner et al refer to LLVM's IR as *C with vectors* in (C. Lattner, Amini, et al., 2021). Type annotations are needed in more places than with C and many constructs are lower-level than C (like control flow), but the approximations is accurate. The IR was designed to preserve high-level information from diverse programming languages, but it is apparent that it is designed especially for C and C++.

There are no physical registers in LLVM IR. It is an static single assignment (SSA) ir, meaning there are infinite "registers" and each can be assigned to only once. Typically, SSA IRs use a *phi* or φinstruction to merge values from different paths, like the value `x` in the C expression `if (cond) x = 1; else x = 2;`. The control-flow graph (CFG) is also explicit; functions are made up of basic blocks, basic blocks are made up of instructions and terminated by branches or terminators, and functions end in exactly one terminator (like a `return`):

```
; Perhaps not the most efficient way to add two numbers.
; unsigned add2(unsigned a, unsigned b) {
```

```
;   if (a == 0) return b;
;   return add2(a-1, b+1);
; }

define i32 @add2(i32 %a, i32 %b) {
entry:
  %tmp1 = icmp eq i32 %a, 0
  br i1 %tmp1, label %done, label %recurse

recurse:
  %tmp2 = sub i32 %a, 1
  %tmp3 = add i32 %b, 1
  %tmp4 = call i32 @add2(i32 %tmp2, i32 %tmp3)
  ret i32 %tmp4

done:
  ret i32 %b
}
```

There are few instructions and some of them are overloaded, making the IR look very similar to RISC assembly languages. The IR as it was when Lattner published his PhD thesis had only 31 opcodes (C. Lattner, 2005).

Structs are represented in an unsurprising way:

```
; struct RT {char A; int B[10][20]; char C;};
; struct ST {int X; double Y; struct RT Z;};
%struct.ST = type { i32, double, %struct.RT }
%struct.RT = type { i8, [10 x [20 x i32]], i8 }
```

Memory operations are represented by the `load` and `store` instructions. The access `s[1].Z.B[5][13];` in the snippet above looks roughly like this in modern LLVM IR:

```
%0 = ; pointer to value of type ST
%1 = getelementptr i8, ptr %0, i64 1296
%2 = load i32, ptr %2
```

LLVM IR was initially more strongly typed than it is today:

> One of the fundamental design features of LLVM is the inclusion of a language-independent type system. Every SSA register and explicit memory object has an associated type, and all operations obey strict type rules. (C. Lattner, 2005)

This strictness has been eased over time. The `getelementptr` instruction was used to compute address offsets on base pointers, and explicit types were required. Pointer types were specified with `i8*`, just as one would write in C. Now, with the shift to *opaque pointers* (or *untyped pointers*),

there is a simple `ptr` type that does not specify the pointee type, and `ptradd` is used instead of `getelementptr` with no required type annotation.

[TODO: review https://discourse.llvm.org/t/rfc-de-type-ification-of-llvm-ir-why/88257.]

6.3.3 The Interface

As already discussed, a huge part of LLVM's value proposition is its ability to be consumed by other applications by virtue of its modular design. Another project can link against LLVM's libraries and use its interfaces to procedurally build IR, or they can just write LLVM IR to a file and pass it to the LLVM tools. This means LLVM's interfaces for constructing IR are more important than in other compiler projects; other developers *outside of LLVM* depend on those interfaces. The users of LLVM's interfaces far outnumber the developers contributing to LLVM directly on a regular basis.

```
static LLVMContext *Context;
static IRBuilder<> *Builder;
static Module *LLVMModule;

Function *createAddFunction() {
    Type *i32 = Builder->getInt32Ty();
    auto *FT = FunctionType::get(i32, {i32, i32}, /*isVarArg=*/false);
    auto *Func = Function::Create(FT, Function::ExternalLinkage,
                                  "add_ints", LLVMModule);
    auto *BB = BasicBlock::Create(*Context, "entry", Func);
    Builder->SetInsertPoint(BB);
    auto LHS = Func->getArg(0);
    auto RHS = Func->getArg(1);
    auto Result = Builder->CreateAdd(LHS, RHS);
    Builder->CreateRet(Result);
    return Func;
}
```

The above snippet of C++ constructs the function in LLVM IR below:

```
define i32 @add_ints(i32 %0, i32 %1) {
entry:
  %2 = add i32 %0, %1
  ret i32 %2
}
```

The modular interfaces for working with IR do not stop with the *construction* of IR; there are also rich interfaces for constructing compiler passes and pattern-matching. The example instruction-simplification pass below from (A. Brown and Wilson, 2011) demonstrates how the pattern-matching interfaces might be used in a peephole optimization optimization:

```
// X - 0 -> X
if (match(Op1, m_Zero()))
  return Op0;
// X - X -> 0
if (Op0 == Op1)
  return Constant::getNullValue(Op0->getType());
// (X*2) - X -> X
if (match(Op0, m_Mul(m_Specific(Op1), m_ConstantInt<2>())))
  return Op1;
// ...
return nullptr;  // Nothing matched, return null to indicate no transformation.
```

This function might be called in a simplification pass that traverses all the instructions in a basic block, and if a simpler version of the instruction is found, the simpler version replaces the original instruction like so:

```
for (auto &I : BB)
  if (auto *NewValue = simplifyInstruction(I))
    I.replaceAllUsesWith(NewValue);
```

6.3.3.1 Aside: Influence of ML on LLVM

As a brief (and opinionated) aside: in (A. Brown and Wilson, 2011), Lattner makes this remark about the pattern matching features in LLVM:

> LLVM is implemented in C++, which isn't well known for its pattern matching capabilities (compared to functional languages like Objective Caml), but it does offer a very general template system that allows us to implement something similar. The match function and the `m_` functions allow us to perform declarative pattern matching operations on LLVM IR code. For example, the `m_Specific` predicate only matches if the left hand side of the multiplication is the same as `Op1`.

And in this interview (C. Lattner and Minsky, 2025), Chris elaborates further:

> **Chris** And so Clang has some really cool stuff that allowed it to scale and things like that, but I was also burned out. We had just shipped it. It was amazing. I'm like, there has to be something better. And so, Swift really came starting in 2010. It was a nights and weekends project. Turns out, programming languages are a mature space. It's not like you need to invent pattern matching at this point. It's embarrassing that C++ doesn't have good pattern matching.
>
> **Ron** We should just pause for a second, because I think this is like a small but really essential thing. I think the single best feature coming out of language like ML in the mid-seventies…having this pattern matching facility that lets you basically in a reliable way do the case analysis so you can break down what the possibilities are—is just incredibly

useful. And very few mainstream languages have picked it up. I mean Swift again is an example, but languages like ML, SML, and Haskell, and OCaml.

Chris I mean pattern matching, it is not an exotic feature. Here we're talking about 2010. And so pattern matching, when I learned OCaml, it's so beautiful. It makes it so easy and expressive to build very simple things.

It appears to me that *every* compiler is written in ML to some degree–for whatever reason, the language features that Landin envisioned in (P. J. Landin, 1966) and were expanded upon in LCF/ML [TODO: needs citation, refer to the ML section] happen to be particularly useful when designing compilers.

This quote is often attributed to Tony Hoare: "I don't know what the language of the year 2000 will look like, but I know it will be called Fortran." Well, I don't know what the language of choice for writing compilers will be in the year 2100, but I know it will look like ML.

6.3.4 Tablegen

Perhaps the part of LLVM that is least-known among *users* of LLVM is its *Tablegen* system. Tablegen is a program for declarative metaprogramming inside LLVM.

(A. Brown and Wilson, 2011, 11.6.2. Unit Testing the Optimizer).

6.3.5 Testing

[TODO: lit, filecheck, generate-test-case.py, other tools.]

6.4 MLIR

6.4.1 An IR In Tension

In modern compilers, the IR must be extremely flexible and generic because every component of the compiler needs something different from it.

For example, the compiler's frontend typically cares very much about the grammar of the source language, the semantic correctness of the AST, and perhaps some early optimizations that might be performed at a very high level. However, after that, the frontend really only cares about communicating the information it has to the optimizer. It does not necessarily care about the details of the optimizer, and simply wants a textual representation of the program. The frontend would ideally emit an IR that is simple with a level of abstraction roughly matching that of the source language, minimizing the amount of work required to generate it. The frontend may record that the user requested a particular region of code to be inlined, unrolled, or offloaded, but writ large, the frontend does not want to deal with the details of actually performing those transformations.

The optimizer, on the other hand, cares very much about the semantics the IR because it is searching for patterns in the program that can be optimized. The optimizer will want *normal forms* of programs that make pattern matching more straightforward, and it needs to be able to represent the artifacts of optimization. For example, after vectorization, the IR produced by the optimizer may look dramatically different from the user's source code, with loops versioned for different vector lengths based on aliasing information, inlined function calls, dead code eliminated, and operations

with operands known at compile time folded away. For this phase of the compiler (and for certain optimizations more than others), an IR capable of representing *optimized* code is necessary, including operations that may not be representable in the source language. The optimizer does not care about how many registers the target machine has nearly as much as the backend does, and it does not mind making drastic changes to the representation of the program.

The backend has another set of desires; it must have information about the machine being targeted, and it needs to map the semantics represented by the IR to the machine's capabilities. It cares about the number of registers available, the legality of operations on a particular machine, and how to perform those operations efficiently. The optimizer may have produced an IR using very wide vectors, but the backend will have to decide how to perform those operations on a specific machine that may or may not have hardware that corresponds to vector operations of that length.

For these reasons, the different components of a compiler are constantly pulling the IR in different directions, because all of their needs and wants must be representable in their common format. Now, imagine we have an IR that allows each component of the compiler to express the semantics in a format and with semantics that *they dictate*. The frontend can define very high-level operations that are roughly equivalent to the source language so the work it needs to do when converting the AST into the IR is minimal. The vectorizer may define new operations on vectors, the semantics they describe, and the process for converting them into another format. The backend may define different operations for each machine it targets, and it can progressively convert constructs from the frontend and optimizer into operations suitable for the target machine, that it can convert into machine code. This is one of the key benefits of MLIR; each component can define its own operations and semantics, the IR can be *progressively* lowered from one phase to the next, and each phase of the compiler can coexist with the others.

Prior to MLIR, most IRs primarily took a *one size fits all* approach, which, as we have just outlined, puts different parts of the compiler in tension with each other. There have been numerous benefits though; LLVM's IR is roughly "C with vectors," which, for obvious reasons, is very easy for C and C++ compilers to target, but other compilers have been able to adopt it as well. Rust, Swift, Julia, Zig, some Fortran compilers, and many other complete programming languages emit LLVM IR. Their respective projects focus on the language's frontend and on high-level optimizations, and once the IR has been sufficient optimized, those language's compilers simply convert *their* IRs into LLVM IR and let the LLVM toolchain handle lower-level optimizations and code generation.

LLVM IR does impose some serious constraints, however. High-level and domain-specific analyses and optimizations remain difficult.

6.4.2 Explaining MLIR

MLIR(C. Lattner, Amini, et al., 2021) is a *multi-level* IR; similar to the *nanopass* architecture(Keep and Dybvig, 2013b), it allows for very flexible construction of *dialects*, which is the MLIR project's notion of an IR. The key feature of these dialects is that they can be composed together–in the representation of a program, many dialects can be used together to represent different aspects of the program at different points in the compilation process. This orthogonality reminds me of the ALGOL committee's orthogonality design principle. They aimed to design programming language features that all composed with each other on different axes. Similarly, MLIR dialects can be composed together without knowledge of each other.

The original MLIR paper, "MLIR: Scaling Compiler Infrastructure for Domain Specific Computation", points out the difficulty of higher-level analysis of C++ programs on LLVM IR; the IR simply

cannot represent all the complicated semantics of C++ programs, and thus optimizations that take advantage of them must either occur in the frontend, try to recover the high-level information from the IR itself, or find a way to stuff that information into the IR in the form of metadata, and perform the optimizations and analyses early enough in the compiler that the information is not obfuscated by other optimizations first.

In the previous section, I pointed out a few compilers and interpreters that target LLVM IR, such as Rust and Swift. It is a testament to LLVM IR's flexibility that all those projects were able to leverage it, but they also had to maintain a level of abstraction higher than that of LLVM IR to represent constructs specific to their languages. MLIR seeks to fill this gap as well.

One of the primary goals of MLIR as listed in "MLIR: Scaling Compiler Infrastructure for Domain Specific Computation" was that of *progressivity*:

> Premature lowering is the root of all evil. Beyond representation layers, allow multiple transformation paths that lower individual regions on demand. Together with abstraction-independent principles and interfaces, this enables reuse across multiple domains.

MLIR allows each phase of the compiler to build an IR on its own terms, and the transition from one phase to the next is *continuous* instead of *discrete*. The compiler may perform optimizations that are suitable for *just* after code generation, when the frontend has passed the program to the optimizer, or it may perform low level optimizations towards the end of the compilation pipeline using dialects with machine-specific operations. The progressive nature of the transition between phases of an MLIR-based compiler allows all the phases of the compiler to coexist and interact seamlessly.

One of the programming languages we mentioned above was Fortran. While Fortran was one of the first programming languages we discussed in this book, we will come to find that the Flang Fortran compiler, also part of the LLVM project, was one of the first projects to adopt MLIR, and one of the only compilers for a general-purpose programming language to do so.

6.4.3 Flang

The figure given in fig. 6.1 depicts a simplified version of Flang's process lowering the AST to MLIR. There are numerous optimizations better suited to different stages of this process, and each is able to perform at the optimal level of abstraction.

LLVM Fortran Levels Up: Goodbye `flang-new`, Hello `flang`!

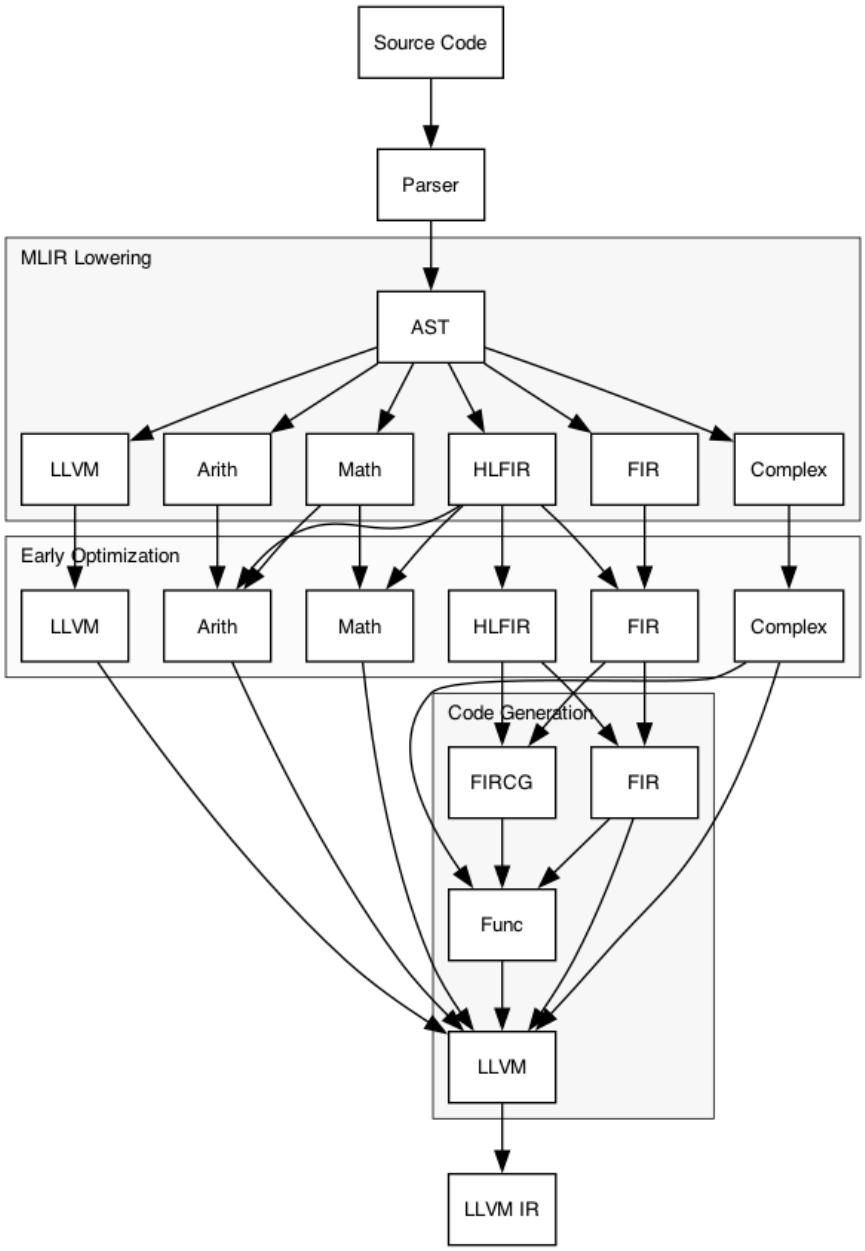


Figure 6.1: MLIR Lowering in Flang

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.

6.5 *A Catalogue of Optimizing Transformations*

F. Allen and Cocke (1971) 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.”

6.6 “Keynote Address”

Grace Hopper (1978)

> Mark I had built-in programs for sine, cosine, exponential, arctangent, .-... On theother 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 alwaysknew 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 Ihave your sine subroutine?” and I’d copy it out of his notebook. We soon found that weneeded just a generalized format of these if we were going to copy them, and I found ageneralized subroutine for Mark I. With substitution of certain numbers, it can be copiedinto any given program. So as early as 1944 we started putting together things which wouldmake it easier to write more accurate programs and get them written faster. I think we’ve forgotten to some extent how early that started.

6.7 *The First Computers: History and Architectures*

Raul Rojas and Hashagen (2002)

> 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 than 45 degrees to only ten digits. Rather than use the slow sine unit built into themachine, 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 bymagnetic core memory, to which access was slow relative to processor operations. Thus it made sense tospecify in great detail what one wanted to do with a piece of data before going off to memory to get it. Theinstruction sets also reflected the state of compiler technology. If the processor could perform a lot ofarithmetic on data with only one instruction, then the compiler would have that much less work to do. A richinstruction set would reduce the "semantic gap" between the English-like commands of a high-levelprogramming language and the primitive and tedious commands of machine code. Cheap read-only memorychips meant that the designer could create these rich instruction sets at low cost if the computer was microprogrammed.

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

Beyer (2009)

Though it is sometimes difficult to identify the motivation behind particularinventions, it appears that a dearth of talented programmers, a personalfrustration with the monotony of existing programming techniques, and the lack of resourcesmade available by senior management at Remington Rand to support computerclients 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 reasoning in regard to the invention changed. Compilers became lessabout relievingprogrammers 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 taskto the computer itself.

First andforemost, the central motivations for automatic programmingwere far more personal in the 1952 paper. With the constructionof a functioning compiler, Hopper hoped, "the programmermay return to being a mathematician." Though Hopper hadsincerely enjoyed the challenge of coding since first being introduced to computers 8 years earlier, she wrote, "the novelty ofinventing programs wears off and degenerates into the dull laborof writing and checking programs. The duty now looms as animposition on the human brain." By teaching computers toprogram themselves, Hopper would be free to explore otherintellectual pursuits.

These computing pioneers, according toHopper, created machines and methods that removed the arithmetical chore from the mathematician. This chore, however, wasreplaced by the new burden of writing code, thus turning mathematicians into programmers. Hopper's paper boldly offers thenext step in the history of computing: shifting the human-machine interface once again so as to free the mathematicianfrom this new burden, and making "the compiling routine bethe programmer and perform all those services necessary to theproduction of a finished program."

He [the mathematician] is supplied with a catalogue of subroutines. Nolonger does he need to have available formulas or tables of elementaryfunctions. He does not even need to know the particular instructioncode used by the computer. He needs only to be able to use the catalogue to supply information to the computer about his problem.20The "catalog of subroutines" was a menu that listed all theinput information needed by the compiler to look up subroutines in the library, assemble them in the proper order, manageaddress assignments, allocate memory, transcribe code, and createa final program in the computer's specific machine code.21 Asubroutine entry in the catalogue consisted of a subroutine"call-number" and the order in which arguments, controls, andresults were to be stated. The call-number identifi ed the type ofsubroutine (t for trigonometric, x for exponential, etc.), specifiedtransfer of control (entrance and exit points in each subroutine),and set operating and memory requirements. In fact, languagesuch as "call-number" and "library" compelled Hopper to nameher program generator a "compiler," for it compiled subroutinesinto a program in much the same way that historians compilebooks into an organized bibliography.22

A program generated by a compiler could not only berun as a stand alone program whenever desired; it also "may itselfbe placed in the library as a more advanced subroutine." Thissuggested that subroutine libraries could increase in size andcomplexity at an exponential rate, thus enabling mathematiciansto solve problems once deemed impossible or impractical.23

Hopper ends the paper by establishing a short-term road mapfor the future development of compilers. She describes a "type B" compiler, which, by means of multiple passes, could supplement computer information provided by the programmer withself-generated information. Such a compiler, she imagines, wouldbe able to automate the process of solving complex differentialequations. 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 byrepeated application of the type-B compiler.24Hopper also admits that the current version of her compilerdid 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 programcompute them simultaneously. Hopper's compiler would embed botha sine subroutine and a cosine subroutine in sequence, thuswasting valuable memory and processing time. Hopper statesboldly 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 endorsementof the A-0 compiler, Ridgway dedicates a substantial amount ofhis paper to the inefficiency of run-programs. (A "run-program"was the final product of the compiler process. Today, such aprogram is called object or machine code.) During the 5 monthssince Hopper had introduced compilers, critics had pointed outthat run-programs generated by compilers were less efficient thanthose 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 morecomputer time, both as a result of compiling a program and asa consequence of inefficient code. But "in this case," he argued,"the compiler used was the 'antique,' or A-0, the first to beconstructed and the most inefficient." Ridgway was confidentthat Hopper and her team at the Computation Analysis Laboratory would construct new compilers that "squeezed" codinginto "neat, efficient, and compact little packages of potentialcomputation."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 thesignifi cant improvements over the A-0 compiler, automatic programming had itslimitations. Hopper's vision of intuitive, user-friendly, hardware-independentpseudo-codes generating efficient running programs was far from realization.The A-2 provided a three-address "pseudocode" specifically designed for theUNIVAC I 12-character standards. The manual defined "pseudo-code" as "computerwords other than the machine (C-10) code, designed with regard to facilitatingcommunications between programmer and computer."32 Today we refer to it assource code. Since pseudo-code could not be directly executed by the UNIVAC I,the A-2 compiler included a translator routine which converted the pseudocodeinto 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 towrite."33

The most groundbreaking change was the A-2's ability to debugpseudo-code and flag errors automatically. The compiler generated twelve-character error codes that captured the nature of theerror, a miraculous innovation for any programmer who hadexperienced the pain and monotony of debugging computercode. (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 ofthe pseudo-code.

Not only would it be far easier to learnthan machine code; its intuitive logic would help users debugtheir work. "I felt," Hopper recalled, "that sooner or later...our attitude should be not that people should have to learn howto code for the computer but rather the computer should learnhow to respond to people because I figured we weren't going toteach the whole population of the United States how to writecomputer code, and that therefore there had to be an interfacebuilt that would accept things which were people-oriented andthen use the computer to translate to machine code."38She mentioned that

compilerscould be designed to program the machine code of any computer. "A problem stated in a basic pseudo-code can thus beprepared 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 programin machine language, pseudo-code was now liberated from a specific 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 benefit was theoretical, insofar as her laboratory had tried it only once, with inconclusive results.39Though 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 efficiently. In her final 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 field. 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 non-UNIVAC 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.

Reflecting on the negative reactions of some of her fellow programmers, Hopper expressed the belief that arguments focusing on "efficiency" 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 themore the likes of Backus and Hopper preached the benefits ofautomatic programming, the more concerned the programming priesthood became about the spreading technology.

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

Hiltzik (1999) 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 programmersa 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 beforereplacing 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 research programmers, like those 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 chair."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 that casual. 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.

6.10 *A New History of Modern Computing*

Haigh (2021)

> 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 togenerate code in the new machines language, and some work to tweak the Unix kernel andstandard libraries to accommodate its quirks.

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

Speiser (2000)

> [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 atthe ETH. We spent most of the year 1949 with Howard Aiken at Harvard and John von Neumann atPrinceton, but we also looked at other installations, among them the ENIAC at Aberdeen and the Mark II atDahlgren. We gratefully acknowledge the hospitality with which we were received and the openness withwhich we were given information.Despite the fact that the Z4 was only barely operational, he decided that the idea of transferring it to Zurichshould 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 decadelater, 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 ofletting the Z4 run as a compiler, a mode of operation which Zuse had never intended. For this purpose, thenecessary instructions were interpreted as numbers and stored in the memory. Then, a compiler programcalculated the program and punched it out on a tape. All this required certain hardware changes. Rutishausercompiled a program with as many as 4000 instructions. Zuse was quite impressed when we showed him thisachievement.

Bibliography

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

Aho, Alfred V. (Feb. 2019). *Bell Labs' Role in Programming Languages and Algorithms*. Lecture page, Simons Foundation. Lecture description; accessed October 18, 2025. URL: https://www.simonsfoundation.org/event/bell-labs-role-in-programming-languages-and-algorithms/.

— (June 2022). *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. URL: https://www.computerhistory.org/collections/catalog/102792704/.

Aho, Alfred V. and John E. Hopcroft (1974). *The Design and Analysis of Computer Algorithms*. 1st. USA: Addison-Wesley Longman Publishing Co., Inc. ISBN: 0201000296.

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

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

— (1977). *Principles of Compiler Design (Addison-Wesley series in computer science and information processing)*. USA: Addison-Wesley Longman Publishing Co., Inc. ISBN: 0201000229.

Aiken, Howard (Feb. 1973). *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. URL: https://mads.si.edu/mads/id/NMAH-AC0196_aike73027.

Akera, Atsushi (2001). "Voluntarism and the Fruits of Collaboration: The IBM User Group, Share". In: *Technology and Culture* 42.4, pp. 710–736. ISSN: 0040165X, 10973729. URL: http://www.jstor.org/stable/25147801 (visited on 10/30/2025).

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

Allen, J. J., D. P. Moore, and H. P. Rogoway (1963). "SHARE Internal FORTRAN Translator". In: *Datamation* 9.3, pp. 43–46. URL: https://bitsavers.org/magazines/Datamation/196303.pdf.

Allen, John (1978). *Anatomy of LISP*. USA: McGraw-Hill, Inc. ISBN: 007001115X.

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

Appel, Andrew W. (1997). *Modern Compiler Implementation in ML: Basic Techniques*. USA: Cambridge University Press. ISBN: 052158275X.

Armer, Paul (Apr. 1980). "SHARE-A Eulogy to Cooperative Effort". In: *IEEE Ann. Hist. Comput.* 2.2, pp. 122–129. ISSN: 1058-6180. DOI: 10.1109/MAHC.1980.10013. URL: https://doi.org/10.1109/MAHC.1980.10013.

Ash, R. et al. (Apr. 1957). *Preliminary Manual for MATH-MATIC and ARITH-MATIC Systems for Algebraic Translation and Compilation for UNIVAC I and II*. Tech. rep. Prepared by the Automatic Programming Development group at Remington Rand UNIVAC. Philadelphia, Pennsylvania: Automatic Programming Development, Remington Rand UNIVAC. URL: https://archive.computerhistory.org/resources/access/text/2016/06/102724614-05-01-acc.pdf.

Aspray, William (1985). *Proceedings of a Symposium on Large Scale Digital Calculating Machinery 1948*. Cambridge, MA, USA: MIT Press. ISBN: 0262081520. URL: http://www.bitsavers.org/pdf/harvard/Proceedings_of_a_Second_Symposium_on_Large-Scale_Digital_Calculating_Machinery_Sep49.pdf.

Authors, JAX (2024). *JAX: High performance array computing*. https://github.com/jax-ml/jax.

Babbage (May 5, 2024). *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). URL: https://thechipletter.substack.com/p/a-history-of-c-compilers-part-1-performance.

Backus, J. W., F. L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, M. Woodger, and P. Naur (Jan. 1963). "Revised report on the algorithmic language ALGOL 60". In: *Commun. ACM* 6.1, pp. 1–17. ISSN: 0001-0782. DOI: 10.1145/366193.366201. URL: https://doi.org/10.1145/366193.366201.

Backus, J. W., F. L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, M. Woodger, and Peter Naur (May 1960). "Report on the algorithmic language ALGOL 60". In: *Commun. ACM* 3.5, pp. 299–311. ISSN: 0001-0782. DOI: 10.1145/367236.367262. URL: https://doi.org/10.1145/367236.367262.

Backus, J. W., R. J. Beeber, S. Best, R. Goldberg, L. M. Haibt, et al. (1957). "The FORTRAN automatic coding system". In: *Papers Presented at the February 26-28, 1957, Western Joint Computer Conference: Techniques for Reliability*. IRE-AIEE-ACM '57 (Western). Los Angeles, California: Association for Computing Machinery, pp. 188–198. ISBN: 9781450378611. DOI: 10.1145/1455567.1455599. URL: https://doi.org/10.1145/1455567.1455599.

Backus, J. W., R. J. Beeber, S. Best, R. Goldberg, H. L. Herrick, et al. (1956). *Fortran: Automatic Coding System for the IBM 704 EDPM*. The addenda were published in 1957. URL: http://bitsavers.informatik.uni-stuttgart.de/pdf/ibm/704/704_FortranProgRefMan_Oct56.pdf.

Backus, J. W. and W. P. Heising (1964). "Fortran". In: *IEEE Transactions on Electronic Computers* EC-13.4, pp. 382–385. DOI: 10.1109/PGEC.1964.263818.

Backus, J. W., H. Stern, et al. (1957). "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, pp. 188–198. DOI: 10.1145/1455567.1455599. URL: http://portal.acm.org/citation.cfm?doid=1455567.1455599 (visited on 10/04/2025).

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

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

— (1978c). “The history of Fortran I, II, and III”. In: *History of Programming Languages*. New York, NY, USA: Association for Computing Machinery, pp. 25–74. ISBN: 0127450408. URL: https://www.cs.toronto.edu/~bor/199y08/backus-fortran-copy.pdf.

Backus, John W. (1980). “Programming in America in the 1950s: Some Personal Impressions”. English. In: *A History of Computing in the Twentieth Century: A Collection of Essays*. Ed. by N. Metropolis, J. Howlett, and Gian-Carlo Rota. Originally written in 1976. Reprinted in *A History of Computing in the Twentieth Century*. New York, USA: Academic Press, pp. 125–135. URL: https://www.softwarepreservation.org/projects/FORTRAN/paper/Backus-ProgrammingInAmerica-1976.pdf.

— (Sept. 5, 2006). *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. URL: https://www.computerhistory.org/collections/catalog/102657970/.

Backus, John W. and Harlan Herrick (May 1954). “IBM 701 Speedcoding and Other Automatic Programming Systems”. English. In: *Proceedings of the Symposium on Automatic Programming for Digital Computers*. Navy Mathematical Computing Advisory Panel, 13–14 May 1954. Computer History Museum Lot X2677.2004, Box 3 of 6. Donated by J.A.N. Lee. Office of Naval Research, Department of the Navy. Washington, D.C., USA, pp. 106–113. URL: https://www.softwarepreservation.org/projects/FORTRAN/paper/Backus%20and%20Herrick%20-%20Speedcoding%20-%20ONR%201954.pdf.

Backus, John Warner (1959). “The Syntax and Semantics of the Proposed International Algebraic Language of the Zurich ACM-GAMM Conference”. In: *Proceedings of the First International Conference on Information Processing*. UNESCO. Paris, France: R. Oldenbourg, Munich and Butterworth, London, pp. 125–132. URL: https://www.softwarepreservation.org/projects/ALGOL/paper/Backus-Syntax_and_Semantics_of_Proposed_IAL.pdf.

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

Barron, D. W. et al. (Aug. 1963). “The Main Features of CPL”. In: *The Computer Journal* 6.2, pp. 134–143. ISSN: 0010-4620. DOI: 10.1093/comjnl/6.2.134. eprint: https://academic.oup.com/comjnl/article-pdf/6/2/134/1041447/6-2-134.pdf. URL: https://doi.org/10.1093/comjnl/6.2.134.

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

Bauer, Friedrick, Peter Naur, and Brian Randell (Oct. 1968). “SOFTWARE ENGINEERING”. In: *SOFTWARE ENGINEERING*. Garmisch, Germany: NATO SCIENCE COMMITTEE. URL: https://www.scrummanager.com/files/nato1968e.pdf.

Bell, James R. (June 1973). “Threaded code”. In: *Commun. ACM* 16.6, pp. 370–372. ISSN: 0001-0782. DOI: 10.1145/362248.362270. URL: https://doi.org/10.1145/362248.362270.

Bellia, M. (Sept. 1981). “Hierarchical development of programming languages”. en. In: *Calcolo* 18.3, 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).

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

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

Bibliographie Der Programmiersprachen: Bibliography of Programming Languages. Bibliographie Des Langages De Programmation (1973). First Edition. ISBN-13: 9783794036516, Language: English, Hardcover. It has been impossible to track down this book, please reach out if you have a copy. Pullach/München: Verlag Dokumentation. ISBN: 3794036514.

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

Blair, Fred (1971). "The structure of the LISP compiler". In: *Unpublished paper*. URL: https://softwarepreservation.computerhistory.org/LISP/ibm/Blair-StructureOfLispCompiler.pdf.

Böhm, Corrado and Peter Sestoft (May 2016). *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.

Brown, Amy and Greg Wilson, eds. (2011). *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. ISBN: 978-1-257-63804-3. URL: https://aosabook.org/en/index.html.

Brown, J. H. and John W. Carr III (1954). "Automatic Programming and Its Development on the MIDAC". In: *Symposium on Automatic Programming for Digital Computers, Office of Naval Research, Department of the Navy, Washington, D.C., 13-14 May 1954*. Washington, D.C.: U.S. Dept. of Commerce, Office of Technical Services, pp. 84–97.

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

Burroughs Corporation (Mar. 1963). *BAC-220: Burroughs Algebraic Compiler (Revised Edition)*. Document No. 220-21017. Detroit 32, Michigan. URL: https://bitsavers.org/pdf/burroughs/Electrodata/220/220-21017_B220_BALGOL_196303.pdf.

C. H., A. Koster (1994). *A SHORTER HISTORY OF ALGOL68*. Department of Computer Science, University of Nijmegen, The Netherlands. 180994 (identifier), email: kees@cs.kun.nl. URL: https://web.archive.org/web/20071217203826/http://npt.cc.rsu.ru/user/wanderer/ODP/ALGOL68.txt.

Cameron, Robert D. (Mar. 2002). *Pass-By-Name Parameter Passing*. CMPT 383 Lecture Notes. Accessed <add access date here>. URL: https://www.cs.sfu.ca/~cameron/Teaching/383/PassByName.html.

Cardone, Felice and J Roger Hindley (2006). "History of lambda-calculus and combinatory logic". In: *Handbook of the History of Logic* 5, pp. 723–817. URL: https://api.semanticscholar.org/CorpusID:123879682.

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

Carruth, Chandler (2023). *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. URL: https://www.youtube.com/watch?v=ZI198eFghJk.

Ceruzzi, Paul (2000). ""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, pp. 195–217. ISBN: 0262181975.

— (2025). "Review of Konrad Zuse's Early Computers: The Quest for the Computer in Germany, by Raúl Rojas". In: *IEEE Annals of the History of Computing* 47.2, pp. 60–61. DOI: 10.1109/MAHC.2025.3560208.

Ceruzzi, Paul E. (July 1981). "The Early Computers of Konrad Zuse, 1935 to 1945". In: *IEEE Ann. Hist. Comput.* 3.3, pp. 241–262. ISSN: 1058-6180. DOI: 10.1109/MAHC.1981.10034. URL: https://doi.org/10.1109/MAHC.1981.10034.

— (2003). *A History of Modern Computing*. 2nd ed. Cambridge, MA, USA: MIT Press. ISBN: 0262532034.

Church, Alonzo (1932). "A Set of Postulates for the Foundation of Logic". In: *Annals of Mathematics* 33.2, pp. 346–366. ISSN: 0003486X, 19398980. URL: http://www.jstor.org/stable/1968337 (visited on 11/02/2025).

— (1933). "A Set of Postulates For the Foundation of Logic". In: *Annals of Mathematics* 34.4, pp. 839–864. ISSN: 0003486X. URL: http://www.jstor.org/stable/1968702 (visited on 11/02/2025).

— (1940). "A Formulation of the Simple Theory of Types". In: *The Journal of Symbolic Logic* 5.2, pp. 56–68. ISSN: 00224812. URL: http://www.jstor.org/stable/2266170 (visited on 10/27/2025).

Cocke, John (1969a). *Programming languages and their compilers: Preliminary notes*. USA: New York University. ISBN: B0007F4UOA.

— (1969b). *Programming languages and their compilers: Preliminary notes*. USA: New York University. ISBN: B0007F4UOA. URL: https://www.softwarepreservation.org/projects/FORTRAN/CockeSchwartz_ProgLangCompilers.pdf.

Cohen, I. Bernard (2000). "Howard Aiken and the dawn of the computer age". In: *The First Computers: History and Architectures*. Cambridge, MA, USA: MIT Press, pp. 107–120. ISBN: 0262181975.

Collections (2025). en. URL: https://computerhistory.org/collections/ (visited on 10/05/2025).

Computer History Collection (2025). URL: https://americanhistory.si.edu/comphist/ (visited on 10/05/2025).

Corporation, IBM (Oct. 9, 2025). *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. URL: https://www.ibm.com/history/john-backus.

Cress, Paul, Paul Dirksen, and J. Wesley Graham (1970). *FORTRAN IV with WATFOR and WATFIV*. USA: Prentice Hall Press. ISBN: 0133294331.

Curry, H. B. (1930). "Grundlagen der Kombinatorischen Logik". In: *American Journal of Mathematics* 52.4, pp. 789–834. ISSN: 00029327, 10806377. URL: http://www.jstor.org/stable/2370716 (visited on 11/02/2025).

— (1934). "Functionality in Combinatory Logic". In: *Proceedings of the National Academy of Sciences of the United States of America* 20.11, pp. 584–590. ISSN: 00278424, 10916490. URL: http://www.jstor.org/stable/86796 (visited on 10/27/2025).

Dahl, O. J., E. W. Dijkstra, and C. A. R. Hoare, eds. (1972). *Structured programming*. GBR: Academic Press Ltd. ISBN: 0122005503.

Davidson, Jack and Christopher Fraser (Apr. 1980). "The Design and Application of a Retargetable Peephole Optimizer". In: *ACM Trans. Program. Lang. Syst.* 2, pp. 191–202. DOI: 10.1145/357121.357129.

Davidson, Jack W and Christopher W Fraser (1984). "Register allocation and exhaustive peephole optimization". In: *Software: Practice and Experience* 14.9, pp. 857–865.

Deutsch, Harry and Oliver Marshall (2025). "Alonzo Church". In: *The Stanford Encyclopedia of Philosophy*. Ed. by Edward N. Zalta and Uri Nodelman. Spring 2025. Metaphysics Research Lab, Stanford University.

Dijkstra, E. W. (Nov. 1961). "Letter to the editor: defense of ALGOL 60". In: *Commun. ACM* 4.11, pp. 502–503. ISSN: 0001-0782. DOI: 10.1145/366813.366844. URL: https://doi.org/10.1145/366813.366844.

— (1962). *Primer of Algol 60 Programming*. USA: Academic Press, Inc. ISBN: 0122162501.

Directorate of Management Analysis, Deputy Chief of Staff, Comptroller, Headquarters, U.S. Air Force, and Remington Rand, Inc. (Dec. 1953). *Second Workshop on UNIVAC Automatic Programming (Exhibit F)*. Archival material, Box 5, Folder 1, Grace Murray Hopper Collection, Series 6: Compiling Routines. Record ID: ebl-1562729477047-1562729477111-2. GUID: https://n2t.net/ark:/65665/ep8407e928-4c4a-a0fe-7010232d7b9b. Usage conditions apply. The Pentagon. URL: https://americanhistory.si.edu/fr/collections/archival-item/sova-nmah-ac-0324-ref334.

Duncan, F. G. and A. van Wijngaarden (May 1964). "Cleaning up ALGOL60". In: *ALGOL Bull.* 16, pp. 24–32. ISSN: 0084-6198.

Evans, Arthur (1968). "PAL—a language designed for teaching programming linguistics". In: *Proceedings of the 1968 23rd ACM National Conference*. ACM '68. New York, NY, USA: Association for Computing Machinery, pp. 395–403. ISBN: 9781450374866. DOI: 10.1145/800186.810604. URL: https://doi.org/10.1145/800186.810604.

Farber, David J, Ralph E Griswold, and Ivan P Polonsky (1964). "SNOBOL, a string manipulation language". In: *Journal of the ACM (JACM)* 11.1, pp. 21–30.

Frankston, Charles (May 1984). "The Amber Operating System". Bachelor's thesis. Thesis supervisor: F. J. Corbató. MA thesis. Cambridge, MA: Massachusetts Institute of Technology. URL: https://www.mit.edu/~cbf/thesis.htm.

Fraser, Christopher W. (Dec. 1990). *RTL (Register Transfer Language): Discussion on GCC*. Usenet posting. comp.compilers. URL: https://sites.cs.ucsb.edu/~ckrintz/papers/90-fraser-on-PO-and-gcc.html.

Gelernter, H., J. R. Hansen, and C. L. Gerberich (Apr. 1960). "A Fortran-Compiled List-Processing Language". In: *J. ACM* 7.2, pp. 87–101. ISSN: 0004-5411. DOI: 10.1145/321021.321022. URL: https://doi.org/10.1145/321021.321022.

Gertner, Jon (n.d.). *The Idea Factory: Bell Labs and the Great Age of American Innovation*. Penguin Press. ISBN: 9781594203282.

Gilmore, John, Mike Tiemann, and David Henkel-Wallace (2006). *Marketing Cygnus Support — Free Software History*. Cygnus Support (Cygnus Solutions). Last updated Wed Sep 27 17:51:01 PDT 2006. URL: http://www.toad.com/gnu/cygnus/.

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

Goldstine, Herman H. (Mar. 1980). "*Annals of the History of Computing* . Bernard A. Galler". en. In: *Isis* 71.1, 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).

Gonzalez-Barahona, Jesus M. (Feb. 2021). " A Brief History of Free, Open Source Software and Its Communities ". In: *Computer* 54.02, pp. 75–79. ISSN: 1558-0814. DOI: 10.1109/MC.2020.3041887. URL: https://doi.ieeecomputersociety.org/10.1109/MC.2020.3041887.

Graham, Paul (Jan. 2002). "The Roots of Lisp". Draft. URL: https://languagelog.ldc.upenn.edu/myl/llog/jmc.pdf.

Gross, Michael (1999). *Richard Stallman: High School Misfit, Symbol of Free Software, MacArthur-Certified Genius. Interview transcript in *The More Things Change**. URL: https://www.mgross.com/books/my-generation/my-generation-bonus-chapters/richard-stallman-high-school-misfit-symbol-of-free-software-macarthur-certified-genius/ (visited on 04/09/2014).

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

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

Haigh, Thomas and ACM Staff (2020). *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. URL: https://amturing.acm.org/award_winners/aho_1046358.cfm.

Hamming, Richard W. and Edward A. Feigenbaum (1971). "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. URL: https://tcm.computerhistory.org/ComputerTimeline/Chap41_ibm7094_CS1.pdf.

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

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

Hindley, J. Roger (1997). *Basic simple type theory*. USA: Cambridge University Press. ISBN: 0521465184.

Hindley, J.R. and J.P. Seldin (2008). *Lambda-Calculus and Combinators: An Introduction*. Cambridge University Press. ISBN: 9781139473248. URL: https://books.google.com/books?id=9fhujocrM7wC.

Hohmann, J. (1979). *Der Plankalkül im Vergleich mit algorithmischen Sprachen*. Informatik und Operations Research. Toeche-Mittler. ISBN: 9783878200284. URL: https://books.google.com/books?id=f_dLAAAACAAJ.

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

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

Hopper, Grace M. (1956). "The Interlude 1954-1956". English. In: *Symposium on Advanced Programming Methods for Digital Computers: Washington, D.C., June 28, 29, 1956*. Washington, D.C.: Office of Naval Research, Dept. of the Navy, pp. 1–2.

Hopper, Grace Murray (1952). "The education of a computer". In: *Proceedings of the 1952 ACM National Meeting (Pittsburgh)*. ACM '52. Pittsburgh, Pennsylvania: Association for Computing Ma-

chinery, pp. 243–249. ISBN: 9781450373623. DOI: 10.1145/609784.609818. URL: https://doi.org/10.1145/609784.609818.

— (Jan. 1955). *Preliminary Definitions: Data-Processing Compiler*. Archival material, Box 4, Folder 12, Grace Murray Hopper Collection, Series 5: Eckert-Mauchly Corporation. Record ID: ebl-1562729477047-1562729477108-3. GUID: https://n2t.net/ark:/65665/ep8441f23a6-5a7e-40dd-a2c4-d7060145f911. Usage conditions apply. URL: https://americanhistory.si.edu/collections/archival-item/sova-nmah-ac-0324-ref311.

— (Dec. 1980). *Grace Hopper Oral History*. English. Ed. by Angeline Pantages. Transcript, Computer History Museum Oral History Collection. Catalogue number 102702026. Gift of Angela Pantages Doyne. Acquisition number X5142.2009. Naval Data Automation Command, Maryland, USA. URL: https://www.computerhistory.org/collections/catalog/102702026/.

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

INRIA (2019). *A History of OCaml*. URL: https://ocaml.org/learn/history.html (visited on 11/06/2019).

International Business Machines Corporation (1954). *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. URL: https://archive.computerhistory.org/resources/access/text/2018/02/102678975-05-01-acc.pdf (visited on 07/04/2022).

International Business Machines Corporation (IBM) (Nov. 1954). *Preliminary Report: Specifications for the IBM Mathematical FORmula TRANslating System (FORTRAN)*. English. Technical Report 102679231. Gift of J.A.N. Lee. Fortran Archive, Computer History Museum. Acquisition number X2677.2004. New York, USA: International Business Machines Corporation. URL: https://archive.computerhistory.org/resources/text/Fortran/102679231.05.01.acc.pdf.

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

Johnson, Stephen C. and Ravi Sethi (1990). "Yacc: a parser generator". In: *UNIX Vol. II: Research System (10th Ed.)* USA: W. B. Saunders Company, pp. 347–374. ISBN: 0030475295.

Johnson, Steve (Feb. 2019). *Re: What year was YACC born?* Email message to Eric S. Raymond, archived on the TUHS mailing list. Quoted in message from Eric S. Raymond titled *Steve Johnson's reply*, posted to the TUHS mailing list. Provides a first-hand account of the origins and early development of Yacc at Bell Labs. URL: https://minnie.tuhs.org/pipermail/tuhs/2019-February/017065.html.

Keep, Andrew W. and R. Kent Dybvig (2013a). "A nanopass framework for commercial compiler development". In: *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*. ICFP '13. Boston, Massachusetts, USA: Association for Computing Machinery, pp. 343–350. ISBN: 9781450323260. DOI: 10.1145/2500365.2500618. URL: https://doi.org/10.1145/2500365.2500618.

— (Sept. 2013b). "A nanopass framework for commercial compiler development". In: *SIGPLAN Not.* 48.9, pp. 343–350. ISSN: 0362-1340. DOI: 10.1145/2544174.2500618. URL: https://doi.org/10.1145/2544174.2500618.

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

Kernighan, Brian W. and Lorinda L. Cherry (Mar. 1975). "A system for typesetting mathematics". In: *Commun. ACM* 18.3, pp. 151–157. ISSN: 0001-0782. DOI: 10.1145/360680.360684. URL: https://doi.org/10.1145/360680.360684.

Kernighan, Brian W. and Dennis M. Ritchie (1989). *The C programming language*. USA: Prentice Hall Press. ISBN: 0131103628.

Kessler, Robert R (1984). "Peep: an architectural description driven peephole optimizer". In: *ACM SIGPLAN Notices* 19.6, pp. 106–110.

Knuth, Donald E. (July 1964). "Man or Boy?" In: *ALGOL Bulletin* 17. AB17.2.4 Donald Knuth: Man or Boy?, page 7. See also: *Algol Bulletin. Computing at Chilton: 1961–2000*. Retrieved December 25, 2009, p. 7. URL: https://archive.computerhistory.org/resources/text/algol/algol_bulletin/A17/P24.HTM.

Knuth, Donald E. and Luis Trabb Pardo (July 1976). *The Early Development of Programming Languages*. English. Technical Report ADA032123. Commissioned by the *Encyclopedia of Computer Science and Technology*, ed. Jack Belzer, Albert G. Holzman, and Allen Kent. Approved for public release. Stanford, CA, USA: Department of Computer Science, Stanford University, p. 112. URL: https://apps.dtic.mil/sti/citations/ADA032123.

Lam, Siu Kwan, Antoine Pitrou, and Stanley Seibert (Nov. 2015). "Numba: a LLVM-based Python JIT compiler". en. In: *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*. Austin Texas: ACM, 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).

Lamport, Leslie and Lawrence C. Paulson (May 1999). "Should your specification language be typed". In: *ACM Trans. Program. Lang. Syst.* 21.3, pp. 502–526. ISSN: 0164-0925. DOI: 10.1145/319301.319317. URL: https://doi.org/10.1145/319301.319317.

Landin, P. J. (Feb. 1965). "Correspondence between ALGOL 60 and Church's Lambda-notation: part I". In: *Commun. ACM* 8.2, pp. 89–101. ISSN: 0001-0782. DOI: 10.1145/363744.363749. URL: https://doi.org/10.1145/363744.363749.

— (Mar. 1966). "The next 700 programming languages". en. In: *Communications of the ACM* 9.3, 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).

Landin, Peter J (1964). "The mechanical evaluation of expressions". In: *The computer journal* 6.4, pp. 308–320.

— (2000). "My years with Strachey". In: *Higher-Order and Symbolic Computation* 13.1, pp. 75–76.

Laning, J Halcombe and Neal Zierler (1954). *A program for translation of mathematical equations for Whirlwind I*. Instrumentation Laboratory, Massachusetts Institute of Technology. URL: https://archive.computerhistory.org/resources/text/Fortran/102653982.05.01.acc.pdf.

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

Lattner, Chris (May 2005). "Macroscopic Data Structure Analysis and Optimization". *See* http://llvm.cs.uiuc.ed PhD thesis. Urbana, IL: Computer Science Dept., University of Illinois at Urbana-Champaign. URL: https://hdl.handle.net/2142/10994.

— (2021). *The Golden Age of Compiler Design in an Era of HW/SW Co-design*. Keynote presented at ASPLOS 2021, April 20, 2021. Available at https://youtu.be/4HgShra-KnY?si=yh2LZdeUpnPNCv-p. ASPLOS 2021 Keynote, 48,383 views as of Apr 20, 2021. ASPLOS 2021 Conference. URL: https://youtu.be/4HgShra-KnY?si=yh2LZdeUpnPNCv-p.

— (2025). *Chris Lattner's Resumé and Work History*. https://www.nondot.org/sabre/Resume.html. Last updated June 2025. URL: https://www.nondot.org/sabre/Resume.html.

Lattner, Chris and Vikram Adve (2004). "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, p. 75. ISBN: 0769521029. URL: https://llvm.org/pubs/2003-09-30-LifelongOptimizationTR.pdf.

Lattner, Chris, Mehdi Amini, et al. (2021). "MLIR: Scaling Compiler Infrastructure for Domain Specific Computation". In: *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pp. 2–14. DOI: 10.1109/CGO51591.2021.9370308. URL: https://ieeexplore.ieee.org/document/9370308.

Lattner, Chris and Ron Minsky (Sept. 3, 2025). *Why ML Needs a New Programming Language*. https://signalsandthreads.com/why-ml-needs-a-new-programming-language/. Podcast interview on Signals and Threads, Jane Street. URL: https://signalsandthreads.com/why-ml-needs-a-new-programming-language/.

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

Lindsey, C. H. (July 1968). "ALGOL68 with fewer tears". In: *ALGOL Bull.* 28, pp. 9–49. ISSN: 0084-6198. URL: https://dl.acm.org/doi/pdf/10.5555/1061112.1061116.

— (1993a). "A history of ALGOL 68". In: *The Second ACM SIGPLAN Conference on History of Programming Languages*. HOPL-II. Cambridge, Massachusetts, USA: Association for Computing Machinery, pp. 97–132. ISBN: 0897915704. DOI: 10.1145/154766.155365. URL: https://doi.org/10.1145/154766.155365.

— (Mar. 1993b). "A history of ALGOL 68". In: *SIGPLAN Not.* 28.3, pp. 97–132. ISSN: 0362-1340. DOI: 10.1145/155360.155365. URL: https://doi.org/10.1145/155360.155365.

Losh, Warner (Feb. 2020). *The Early History of Unix*. Talk presented at FOSDEM 2020, Brussels, Belgium. Room Janson, scheduled start 13:00. Video available on YouTube and FOSDEM archives. URL: https://www.youtube.com/watch?v=XuzeagzQwRs&t=2024s.

MacQueen, David (Jan. 7, 2025). *The History of LCF, ML and HOPE*. Podcast episode. URL: https://www.typetheoryforall.com/episodes/the-history-of-lcf-ml-and-hope.

MacQueen, David, Robert Harper, and John Reppy (June 2020). "The History of Standard ML". In: *Proc. ACM Program. Lang.* 4.HOPL. DOI: 10.1145/3386336. URL: https://doi.org/10.1145/3386336.

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

Mailloux, B. J., J. E. Peck, and C. H. Koster (Dec. 1969). "Report on the Algorithmic Language ALGOL 68". In: *Numer. Math.* 14.2, pp. 79–218. ISSN: 0029-599X. DOI: 10.1007/BF02163002. URL: https://doi.org/10.1007/BF02163002.

Mailloux, B.J., J.E.L. Peck, and C.H.A. Koster (Jan. 1968). "Penultimate draft report on the algorithmic language Algol 68, 1 : chapters 1-9".

Marchesi, José (Oct. 2025). *GA68: The GNU ALGOL 68 Compiler*. YouTube video, GNU Tools Cauldron 2025 Conference, Porto, Portugal, September 26–28, 2025. Published on the GNU Tools Cauldron YouTube channel, retrieved October 14, 2025. URL: https://www.youtube.com/watch?v=3aMwC24EcJk&list=RQe2HRCqomoBtsS-ud3OO8RBgWKMo&t=390s.

McCarthy, John (1959). "Programs with common sense". In: *Computation & Intelligence: Collected Readings*. USA: American Association for Artificial Intelligence, pp. 479–492. ISBN: 0262621010.

McCarthy, John (Apr. 1960). "Recursive functions of symbolic expressions and their computation by machine, Part I". In: *Commun. ACM* 3.4, pp. 184–195. ISSN: 0001-0782. DOI: 10.1145/367177.367199. URL: https://doi.org/10.1145/367177.367199.

— (1978). "History of LISP". In: *History of Programming Languages*. New York, NY, USA: Association for Computing Machinery, pp. 173–185. ISBN: 0127450408. URL: https://doi.org/10.1145/800025.1198360.

McClure, R. M. (1965). "Programming languages for non-numeric processing—1: TMG—a syntax directed compiler". In: *Proceedings of the 1965 20th National Conference*. ACM '65. Cleveland, Ohio, USA: Association for Computing Machinery, pp. 262–274. ISBN: 9781450374958. DOI: 10.1145/800197.806050. URL: https://doi.org/10.1145/800197.806050.

McIlroy, M. D. (Oct. 1964). "The Origin of Unix Pipes". Internal Bell Labs memorandum describing the concept of pipes prior to Unix. Murray Hill, New Jersey. URL: https://doc.cat-v.org/unix/pipes/.

— (Sept. 1972). *A Manual for the Tmg Compiler-writing Language*. Technical Memorandum. Originally a Bell Laboratories internal report; scanned version accessed <insert date>. Bell Laboratories, Murray Hill, New Jersey. URL: https://amakukha.github.io/tmg/TMG_Manual_McIlroy_1972.html.

McIlroy, Malcolm D. (Sept. 2019). *Oral History of Malcolm D. McIlroy*. English. Interview transcript. Interviewed by David C. Brock. Catalogue number 102795421. CHM Oral History Collection, Etna, New Hampshire, USA. 59 pages. Etna, NH, USA. URL: https://www.computerhistory.org/collections/catalog/102795421/.

McJones, Paul (July 2025). *History of FORTRAN and FORTRAN II*. English. Ed. by Paul McJones. Last modified 2025-07-21. Project to preserve source code, design documents, and other materials concerning the original IBM 704 FORTRAN/FORTRAN II compiler. Computer History Museum, Software Preservation Group. URL: https://mcjones.org/dustydecks.

Miller, J. C. P. (1947). "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, pp. 178–181. DOI: 10.2307/3610522. URL: https://chimera.roma1.infn.it/SP/COMMON/MarkI_operMan_1946.pdf.

Milner, Robin (1972a). "Implementation and applications of Scott's logic for computable functions". In: *Proceedings of ACM Conference on Proving Assertions about Programs*. Las Cruces, New Mexico, USA: Association for Computing Machinery, pp. 1–6. ISBN: 9781450378918. DOI: 10.1145/800235.807067. URL: https://doi.org/10.1145/800235.807067.

— (Jan. 1972b). "Implementation and applications of Scott's logic for computable functions". In: *SIGACT News* 14, pp. 1–6. ISSN: 0163-5700. DOI: 10.1145/942580.807067. URL: https://doi.org/10.1145/942580.807067.

— (Jan. 1972c). "Implementation and applications of Scott's logic for computable functions". In: *SIGPLAN Not.* 7.1, pp. 1–6. ISSN: 0362-1340. DOI: 10.1145/942578.807067. URL: https://doi.org/10.1145/942578.807067.

Moses, Joel (July 1970). "The function of FUNCTION in LISP or why the FUNARG problem should be called the environment problem". In: *SIGSAM Bull.* 15, pp. 13–27. ISSN: 0163-5824. DOI: 10.1145/1093410.1093411. URL: https://doi.org/10.1145/1093410.1093411.

Naur, Peter (1962). "ALGOL bulletin no. 14.1, The Questionnaire". In: *ACM SIGPLAN Notices* 14.1.

— (July 1968). "Successes and failures of the ALGOL effort". In: *ALGOL Bull.* 28, pp. 58–62. ISSN: 0084-6198.

Neumann, J von (1925). "Eine Axiomatisierung der Mengenlehre." In.

Norman, Jeremy (1959). *Grace Hopper and Colleagues Introduce COBOL*. URL: https://www.historyofinformatic com/detail.php?id=778.

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

Nygaard, Kristen and Ole-Johan Dahl (1978). "The development of the SIMULA languages". In: *History of Programming Languages*. New York, NY, USA: Association for Computing Machinery, pp. 439–480. ISBN: 0127450408. URL: https://doi.org/10.1145/800025.1198392.

Olsen, Thomas M. (Dec. 1965). "Philco/IBM translation at problem-oriented, symbolic and binary levels". In: *Commun. ACM* 8.12, pp. 762–768. ISSN: 0001-0782. DOI: 10.1145/365691.365933. URL: https://doi.org/10.1145/365691.365933.

Orosz, Gergely (Nov. 5, 2025). *From Swift to Mojo and high-performance AI Engineering with Chris Lattner*. Podcast episode; duration 1h 32m. URL: https://newsletter.pragmaticengineer.com/p/from-swift-to-mojo-and-high-performance.

Oswald, H. (Aug. 1964). "The Various FORTRANS". In: *Datamation* 10.8, pp. 25–29. URL: https://bitsavers.org/magazines/Datamation/196408.pdf.

Peck, John E. (Nov. 1978). "The ALGOL 68 Story: A personal account by a member of the design team". In: *ACS Bulletin* -.November 1978. Accessed 10-13-2025, pp. 4–6. URL: https://www.softwarepreservation.org/projects/ALGOL/paper/The%20Algol%2068%20Story.pdf.

Perlis, A. J. and K. Samelson (Dec. 1958). "Preliminary report: international algebraic language". In: *Commun. ACM* 1.12, pp. 8–22. ISSN: 0001-0782. DOI: 10.1145/377924.594925. URL: https://doi.org/10.1145/377924.594925.

Plotkin, Gordon, Colin P. Stirling, and Mads Tofte (May 2000). *Proof, Language, and Interaction: Essays in Honour of Robin Milner*. The MIT Press. ISBN: 9780262281676. DOI: 10.7551/mitpress/5641.001.0001. URL: https://doi.org/10.7551/mitpress/5641.001.0001.

Programming Research Department, International Business Machines Corporation (Sept. 1957). *Proposed Specifications for FORTRAN II for the IBM 704*. Internal report. Internal technical specification; IBM 704 FORTRAN II proposal. 590 Madison Avenue, New York, NY 10022 (NY 22): International Business Machines Corporation. URL: https://archive.computerhistory.org/resources/text/Fortran/102663106.05.01.acc.pdf.

Richards, Martin (Apr. 2000). "Christopher Strachey and the Cambridge CPL Compiler". In: *Higher Order Symbol. Comput.* 13.1–2, pp. 85–88. ISSN: 1388-3690. DOI: 10.1023/A:1010014110806. URL: https://doi.org/10.1023/A:1010014110806.

Ridgway, Richard K. (1952). "Compiling routines". In: *Proceedings of the 1952 ACM National Meeting (Toronto)*. ACM '52. Toronto, Ontario, Canada: Association for Computing Machinery, pp. 1–5. ISBN: 9781450379250. DOI: 10.1145/800259.808980. URL: https://doi.org/10.1145/800259.808980.

Ritchie, Dennis M. (1996). "The Development of the C Language". In: *History of Programming Languages II*. Ed. by Thomas J. Bergin Jr. and Richard G. Gibson Jr. Originally presented at the Second History of Programming Languages Conference (HOPL-II), Cambridge, MA, April 1993. Copyright © 1993 ACM. Electronic reprint courtesy of the author. New York, NY and Reading, MA: ACM Press and Addison-Wesley, pp. 671–698. ISBN: 0-201-89502-1. URL: https://www.bell-labs.com/usr/dmr/www/chist.html.

Rojas, Raúl (2000). “The architecture of Konrad Zuse’s early computing machines”. In: *The First Computers: History and Architectures*. Cambridge, MA, USA: MIT Press, pp. 237–261. ISBN: 0262181975.

— (2021). “The Architecture of Konrad Zuse’s Z4 Computer”. In: *2021 7th IEEE History of Electrotechnology Conference (HISTELCON)*, pp. 43–47. DOI: 10.1109/HISTELCON52394.2021.9787324. URL: https://ieeexplore.ieee.org/document/9787324.

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

Rosen, Saul (1961). “ALTAC, FORTRAN, and compatibility”. In: *Proceedings of the 1961 16th ACM National Meeting*. ACM ’61. New York, NY, USA: Association for Computing Machinery, pp. 22.201–22.204. ISBN: 9781450373883. DOI: 10.1145/800029.808498. URL: https://doi.org/10.1145/800029.808498.

Salus, Peter H. (1994). *A quarter century of UNIX*. USA: ACM Press/Addison-Wesley Publishing Co. ISBN: 0201547775. URL: https://www.fsf.net/~adam/qcu.pdf.

Salus, Peter H. and Jeremy C. Reed (2008). *The Daemon, the Gnu, and the Penguin*. Reed Media Services. ISBN: 097903423X. URL: https://web.archive.org/web/20220620020435/http://www.groklaw.net/article.php?story=20050525231654621.

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

— (1978). “The early history of COBOL”. In: *History of Programming Languages*. New York, NY, USA: Association for Computing Machinery, pp. 199–243. ISBN: 0127450408. URL: https://doi.org/10.1145/800025.1198367.

Schönfinkel, Moses (1967). “On the building blocks of mathematical logic”. In: *From Frege to Gödel*, pp. 355–366.

Scott, Dana and C. Strachey (Jan. 1971). “Towards a Mathematical Semantics for Computer Languages”. In: *Proceedings of the Symposium on Computers and Automata* 21.

Scott, Dana S (1993). “A type-theoretical alternative to ISWIM, CUCH, OWHY”. In: *Theoretical computer science* 121.1-2, pp. 411–440.

Scott, Michael L. (2009). In: *Programming Language Pragmatics (Third Edition)*. Ed. by Michael L. Scott. Third Edition. Boston: Morgan Kaufmann, pp. 111–173. ISBN: 978-0-12-374514-9. DOI: https://doi.org/10.1016/B978-0-12-374514-9.00012-4. URL: https://www.sciencedirect.com/science/article/pii/B9780123745149000124.

Smotherman, Mark (Sept. 2023). *S-1 Supercomputer (1975–1988)*. Last update: September 2023. Clemson University, School of Computing. URL: https://people.computing.clemson.edu/~mark/s1.html (visited on 11/08/2025).

Snyder, A. (1975). *A Portable Compiler For The Language C*. Tech. rep. USA.

Speiser, Ambros P. (2000). “Konrad Zuse’s Z4: architecture, programming, and modifications at the ETH Zurich”. In: *The First Computers: History and Architectures*. Cambridge, MA, USA: MIT Press, pp. 263–276. ISBN: 0262181975.

Sperry Rand Corporation (1959). *FLOW-MATIC Programming System, U1518 Rev. 1*. © 1958, 1959. UNIVAC Division. Sperry Rand Corporation. Philadelphia, Pennsylvania. URL: https://bitsavers.trailing-edge.com/pdf/univac/flow-matic/U1518_FLOW-MATIC_Programming_System_1958.pdf.

Spickett, David (Mar. 11, 2025). *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. URL: https://blog.llvm.org/posts/2025-03-11-flang-new/.

Stallman, Richard (2002). *My Lisp Experiences and the Development of GNU Emacs*. Transcript of Richard Stallman's speech at the International Lisp Conference, 28 Oct 2002. URL: https://www.gnu.org/gnu/rms-lisp.html (visited on 11/08/2025).

Stallman, Richard M. (Oct. 30, 1986). *RMS lecture at KTH (Sweden), 1986: Transcript*. Transcript of a speech at the Kungliga Tekniska Högskolan (Royal Institute of Technology), Stockholm, arranged by the student society Datorföreningen Stacken on 30 October 1986. GNU Project / Free Software Foundation. URL: https://www.gnu.org/philosophy/stallman-kth.en.html (visited on 11/08/2025).

— (1990). "The GNU manifesto". In: *Computers, Ethics, & Society*. USA: Oxford University Press, Inc., pp. 308–317. ISBN: 019505850X.

— (n.d.). *Biographies*. https://stallman.org/biographies.html. Accessed: 2025-11-08.

— (2025). *The GNU Project*. Essay page credited to Richard Stallman on GNU.org. GNU Project / Free Software Foundation. URL: https://www.gnu.org/gnu/thegnuproject.en.html (visited on 11/08/2025).

Steele, Guy L. and Richard P. Gabriel (1996). "The evolution of Lisp". In: *History of Programming Languages—II*. New York, NY, USA: Association for Computing Machinery, pp. 233–330. ISBN: 0201895021. URL: https://doi.org/10.1145/234286.1057818.

Strachey, Christopher (Apr. 2000). "Fundamental Concepts in Programming Languages". In: *Higher Order Symbol. Comput.* 13.1–2, pp. 11–49. ISSN: 1388-3690. DOI: 10.1023/A:1010000313106. URL: https://doi.org/10.1023/A:1010000313106.

Stroustrup, Bjarne (1995). *The design and evolution of C++*. USA: ACM Press/Addison-Wesley Publishing Co. ISBN: 0201543303.

Sun Microsystems, Inc. (Apr. 1989). *SUN Enhances Compilers: New SPARCompiler Optimization Technology Boosts Hardware Performance*. https://simson.net/ref/free_software/compiler_clips.pdf. Press release. 2550 Garcia Avenue, Mountain View, CA 94043.

Sward, Ricky E. (2010a). "The rise, fall and persistence of Ada". In: *Proceedings of the ACM SIGAda Annual International Conference on SIGAda*. SIGAda '10. Fairfax, Virginia, USA: Association for Computing Machinery, pp. 71–74. ISBN: 9781450300278. DOI: 10.1145/1879063.1879081. URL: https://doi.org/10.1145/1879063.1879081.

— (Oct. 2010b). "The rise, fall and persistence of Ada". In: *Ada Lett.* 30.3, pp. 71–74. ISSN: 1094-3641. DOI: 10.1145/1879097.1879081. URL: https://doi.org/10.1145/1879097.1879081.

Terekhov, Andrey (2014). "ALGOL 68 and Its Impact on the USSR and Russian Programming". In: *2014 Third International Conference on Computer Technology in Russia and in the Former Soviet Union*, pp. 97–106. DOI: 10.1109/SoRuCom.2014.29.

Thompson, Ken (Aug. 1984). "Reflections on trusting trust". In: *Commun. ACM* 27.8, pp. 761–763. ISSN: 0001-0782. DOI: 10.1145/358198.358210. URL: https://doi.org/10.1145/358198.358210.

— (May 2019). *VCF East 2019 – Brian Kernighan Interviews Ken Thompson*. YouTube video. Retrieved October 14, 2025. URL: https://www.youtube.com/watch?v=EY6q5dv_B-o.

Tillet, Philippe, H. T. Kung, and David Cox (June 2019). "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, 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).

Turing, Alan Mathison et al. (1936). "On computable numbers, with an application to the Entscheidungsproblem". In: *J. of Math* 58.345-363, p. 5.

Type Theory For All (podcast) (Jan. 2025). *Bell Labs*. Podcast episode (interview with David MacQueen). Accessed 2025-10-29. Summary: David MacQueen (Emeritus, University of Chicago) discusses his 20 years at Bell Labs — its technology, people, and management during the institution's golden age. URL: https://www.typetheoryforall.com/episodes/bell-labs.

Ullman, Jeffrey D. (1998). *Elements of ML programming (ML97 ed.)* USA: Prentice-Hall, Inc. ISBN: 0137903871.

Van Wijngaarden, A. et al. (Mar. 1968). "Draft Report on the Algorithmic Language ALGOL 68". In: *ALGOL Bull.* Sup 26, pp. 1–84. ISSN: 0084-6198.

Von Hagen, William (2011). *The definitive guide to GCC*. Apress. URL: https://sensperiodit.wordpress.com/wp-content/uploads/2011/04/hagen-the-definitive-guide-to-gcc-2e-apress-2006.pdf.

Weinreb, Dan (2020). *History of Symbolics Lisp Machines: Rebuttal to Stallman's Story About the Formation of Symbolics and LMI*. Archive of Dan Weinreb's comments on Symbolics and Lisp machines. URL: https://danluu.com/symbolics-lisp-machines/ (visited on 11/08/2025).

Weizenbaum, Joseph (Mar. 1968). "The FUNARG Problem Explained". Massachusetts Institute of Technology, Cambridge, MA. URL: https://softwarepreservation.computerhistory.org/LISP/MIT/Weizenbaum-FUNARG_Problem_Explained-1968.pdf.

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

Wijngaarden, A. van, B. J. Mailloux, J. E. L. Peck, and C. H. A. Koster (Mar. 1968). *Draft Report on the Algorithmic Language ALGOL 68*. Tech. rep. Commissioned by Working Group 2.1 on ALGOL of the International Federation for Information Processing. Supplement to ALGOL Bulletin 26. Amsterdam: Mathematisch Centrum. URL: https://algol68-lang.org/docs/algol68-draft-report.pdf.

Wijngaarden, A. van, B. J. Mailloux, J. E. L. Peck, C. H. A. Koster, et al., eds. (1976). *Revised Report on the Algorithmic Language ALGOL 68*. Retrieved October 14, 2025. Springer-Verlag. ISBN: 978-0-387-07592-1. URL: https://web.archive.org/web/20190419223929/http://web.eah-jena.de/~kleine/history/languages/algol68-revisedreport.pdf.

Wijngaarden, Adriaan van (Jan. 1965). "Orthogonal design and description of a formal language". URL: https://ir.cwi.nl/pub/9208/9208D.pdf.

Wirth, Niklaus (Feb. 2021). "50 years of Pascal". In: *Commun. ACM* 64.3, pp. 39–41. ISSN: 0001-0782. DOI: 10.1145/3447525. URL: https://doi.org/10.1145/3447525.

Woods, C. Geoffrey and Mary K. Hawes (Sept. 1971). "Optimized code generation from extended-entry decision tables". In: *SIGPLAN Not.* 6.8, pp. 74–80. ISSN: 0362-1340. DOI: 10.1145/953368.953378. URL: https://doi.org/10.1145/953368.953378.

Zuse, Horst (2009). *The Life and Work of Konrad Zuse*. Archived from the original on 29 June 2009. Retrieved 18 April 2010 via the Internet Archive. URL: https://web.archive.org/web/20100418164050/http://www.epemag.com/zuse/.

Zuse, Konrad (1976). *Lecture by Konrad Zuse*. Moving image (Betacam SP). Catalogue number 102639685. Gift of FIC. Extent: 00:12:22. Recorded in Los Alamos, NM, USA. Los Alamos, NM, USA: Computer History Museum. URL: https://www.computerhistory.org/collections/catalog/102639685.

Acronyms

abstract syntax tree (AST) A tree representation of the program after being parsed. 106, 107

combinatory logic (CL) . 59, 85

control-flow graph (CFG) In compiler theory, the control-flow graph is a directed graph representing the possible paths of control flow through a program. Nodes of the graph are usually basic blocks. 32, 36, 95, 102

dead code elimination (DCE) A compiler optimization that removes code that is never executed. 95

Free Software Foundation (FSF) The software foundation started by Richard Stallman with the goal of supporting free software. 97

intermediate representation (IR) The internal format of a program as it exists inside the compiler. 95, 100–102, 106

reduced instruction set computer (RISC) todo. 103

register transfer language (RTL) A low-level intermediate representation used to symbolically represent machine instructions in a target-independent format. 94

single instruction, multiple data (SIMD) A methodology for performing the same operation on multiple pieces of data at the same time. With SIMD operations on a CPU data is usually stored in *vectors* and entire vectors are processed at the same time via vector instructions. Often achieved with automatic vectorization or explicit use of SIMD instructions with compiler intrinsics or inline assembly. 99, 100

static single assignment (SSA) A format for IRs where there are infinite registers, and each can be assigned to only once. 102

Glossary

automatic vectorization A process for leveraging SIMD instructions where the compiler *infers* the opportunities to exploit parallelism in the user's program. The user does not *necessarily* have to change their program, but they can often benefit from it or provide compiler flags or preprocessor directives to explicitly request it. 99, 100

back-edge A back-edge is an edge in the CFG that returns execution from the end of a loop body back to the beginning of the loop, or the loop's header. 34

basic block In compiler theory, a basic block is typically a sequence of instructions that are executed in order and containing a single entry point and exit point. 36, 102, 105

bootstrap The process of writing a compiler in the language that it compiles, such that an older version of the compiler can be used to compile a newer version of itself. 65, 78

bytecode A compiler intermediate representation for the purpose of interpretation or execution instead of optimization. 7, 77, 78

call-by-name Function calling semantics where the argument's expression is evaluated *each time it is used* in the body of the function. Contrast this with call-by-value and dynamic binding. 47, 55, 64

call-by-reference Function calling semantics where the argument's address is passed in place of its value such that expressions and statements where the parameter occurs as an l-value result in assignements to the variable's storage in the caller's scope. 55

call-by-value Function calling semantics where the argument's expression is evaluated *once, before it is passed* to the body of the function. Contrast this with call-by-name and dynamic binding. This is how most programming languages work. 48, 55

constant folding Optimization that replaces an expression with a constant value if the expression's value can be determined at compile time. For example, the expression `2 + 3` can be replaced with `5` by the compiler so it need not be calculated by the final program. 37

dynamic binding Variable scoping semantics where the value of a variable is determined by the value the variable name corresponds to in the program's environment when the value is used.. 66

Fortran The Fortran programming language, standing for *FORmula TRANslator*. Most renditions of the name of the programming language have only the first letter capitalized (*Fortran*), however early versions were rendered as *FORTRAN*. We attempt to use the proper name for each time

period. *Fortran* came to be used after the 1990 edition of the standard, while the 1977 standard and all prior versions were rendered as *FORTRAN*. 25

Fortran 77 The 1977 version of the Fortran programming language's standard. 82

forward A compiler optimization where a load of a memory reference is replaced by the value last stored to it. Sometimes called forward substitution. 94

FOSS Free and open-source software. This software is typically distributed under a license that allows users to modify and redistribute it freely, though different licenses imply different permissions. The two primary categories of open-source licenses are *permissive* and *copyleft*. 91

induction variable A variable changes by some constant in each iteration of a loop. In `for (int i=0; i < n; i++)`, the variable `i` is an induction variable. 38, 39

inline assembly A programming language feature that allows a programmer to write assembly code directly within another programming language, reading from and writing to variables in the host programming language. 41

l-value An addressable value that another value can be stored to. Read as "a value that may occur on the left-hand side of an assignment". 47, 55, 87

Lisp machine A type of computer designed to run Lisp as the primary programming language, usually with some level of hardware support. 65, 92

loop-invariant code motion Loop-invariant code motion is an optimization that moves code outside of a loop if it does not depend in any way on the loop's induction variables. 36

normal form Many optimizations rely on the program being in the simplest possible form. Just as we expect fractions to be in their simplest form (it would be unusual to see $\frac{5}{10}^{\text{ths}}$), optimizations often expect or require their input programs to be as reduced as possible form to be maximally effective. for an example. 95

offline compilation todo. 100

online compilation todo. 100

peephole optimization Sometimes called a *peep*, this kind of transformation typically traverses the entire program searching for a small pattern that can be simplified or otherwise transformed in a beneficial way. 104

profile-guided optimization Compiler optimizations that take advantage of statistics from the execution of a program to improve the compiler's heuristics. A user might compile and run a very program with special compiler flags such that loop hotness and trip counts are recorded, and then re-compile their program such that the compiler can use those statistics to drive its optimization decisions. 36

r-value An value that can be stored to an address. Read as "a value that may occur on the right-hand side of an assignment." In most programming languages, l-values may act as r-values, but not all r-values may act as l-values. The variable `foo` may occur on either side of an assignment, but there is little sense in the number `5` being assigned a value. 55, 87

s-expression A data structure used to represent arbitrary lists in Lisp, such as *(a b c)*. Some tools and programming languages outside of Lisp use s-expressions to represent lists and list-like data structures. 94

separable compilation A programming paradigm where a program is divided into multiple modules that can be compiled independently. For example, C source files can be compiled independently, and then linked together to form an executable.. 39

sift The process of automatically translating code in one high-level language to another, preserving semantics and pointing out components of the source code that must be manually translated. 42

strength reduction A compiler optimization [TODO: todo] . 94

stride 1 An array access pattern where the innermost loop iterates over the array elements in contiguous memory locations. 38, 39

translation unit A component of a program being compiled such that the compiler has access to all the information contained in the component during compilation. For C, C++ and Fortran code, a translation unit typically corresponds to a single source file which is compiled to an object file before being linked into a program or library. Some compilers always treat the entire program as a single translation unit, like the original FORTRAN compiler or the Zig compiler. 39, 42

undefined behavior Source code constructs that are illegal as per the language's specification. Typically, undefined behavior assumed never to happen in a well-formed program, is used by optimizers when certain compiler flags are enabled (such as `-fstrict-aliasing`). 80