WHAT BROKE NEXT

A Problem-Driven History of Programming Languages

Sultan Zavrak

April 2026

PREVIEW

WHAT BROKE NEXT

A PROBLEM-DRIVEN HISTORY OF
PROGRAMMING LANGUAGES

SULTAN ZAVRAK

FREE PREVIEW

Introduction + Part I (Chapters 1–4)

PREVIEW

Table of contents

Preface 1
How to Use This Book 1
What This Book Is Not 2
A Note on the Diagrams 2

I. Introduction (Preview) 4

1. Introduction — What This Book Is (and What It Is Not) 5
1.1. The Pressure Cycle 5
1.2. Why “Problem-Driven”? 7
1.3. The Five Questions 7
1.4. The Book Map 8
1.5. What Makes a Language Survive? 8

II. The Hardware Bottleneck (Preview) 11

2. Writing for the Machine 12
2.1. The Machine’s Native Language 12
2.2. Programming as Physical Labor 14
2.3. The First Error Budget 14
2.4. The Real Pressure: Cognitive Load at Scale 16
2.5. What Scale Broke First 17
2.6. The Hired Computers 17
2.7. The Setup for Everything That Follows 18

3. One Step Up from the Metal 19
3.1. Mnemonics: The First Relief 19
3.2. Two Assembly Dialects, Same Problem 20
3.3. The Business Case for Portability 22
3.4. What Assembly Got Right (and Why It Still Matters) . . . 23
3.5. The Cognitive Model Mismatch 23

PREVIEW

3.6. The Setup for FORTRAN . . . 24

4. Let the Machine Do the Translation 26

4.1. The Compiler as a Bet . . . 26
4.2. What FORTRAN Actually Looked Like . . . 28
4.3. The Skeptics and the Proof . . . 30
4.4. The IBM 704 and the First Ecosystem Lock-In . . . 31
4.5. What FORTRAN Could Not Do . . . 31
4.6. The Lasting Lesson of FORTRAN . . . 32

5. The Language That Refused to Die 34

5.1. Grace Hopper and the Business Computability Problem . . . 34
5.2. The Domain Fit . . . 35
5.3. The Verbose Syntax and Why It Was Right . . . 36
5.4. The Infrastructure Trap . . . 38
5.5. What COBOL Got Right That Nobody Admits . . . 40
5.6. The Y2K Moment . . . 41
5.7. The Lesson for Language Designers . . . 42

PREVIEW

Preface

Every programming language you have ever used was a response to a failure.

Not a personal failure. Not a bug someone forgot to fix. A *systemic* failure — a moment when the tools of the day cracked under strain and the people who cared most about programming decided to do something about it. Sometimes that something became FORTRAN. Sometimes it became C. Sometimes it became JavaScript, which is a story with a moral we will get to.

This book is about those moments of cracking: what became hard, why it became hard, who felt the pain most acutely, and what they built to relieve it. In that sense, it is not really a book about programming languages at all. It is a book about *pressure* — technical, human, economic, organizational — and about the artifacts those forces produce.

You do not need to be a programming language theorist to read this. You do not need to have written a compiler or designed a type system. You only need to have written enough code to know that some tools feel like they fit your hand and others feel unmistakably wrong for the job. That feeling is history. That feeling is data.

How to Use This Book

Each chapter follows a five-part structure, sometimes explicit, always present:

1. **What became hard?** — The pressure that accumulated until something had to give.
2. **Why were existing languages inadequate?** — Not because they were poorly designed, but because they were designed for a different set of constraints.

3. **What design move solved part of the pain?** — The key insight, the bet, the tradeoff the designers made.
4. **What new tradeoff appeared?** — Every solution is a new problem. Every relief introduces a new ache.
5. **What pressure led to the next wave?** — History does not end. The cycle continues.

You can read this linearly, as a narrative from punch cards to LLMs. You can also jump to the chapter that covers the language you care about most. The chapters are designed to be coherent on their own while building on each other.

What This Book Is Not

This is not a language comparison guide. You will not find a table scoring Python against Go against Rust on ten axes. That book exists in a hundred forms; this is not another one.

This is not a tutorial. We will look at code, sometimes quite a bit of it, but always to understand *why* — why that syntax, why that constraint, why that tradeoff — not to teach you to use the language.

This is not neutral. Languages have tradeoffs, and tradeoffs have opinions baked in. When a language makes a choice, it is making a statement about what problems matter and who deserves to have them solved. We will say so, plainly, when the evidence supports it.

A Note on the Diagrams

The figures in this book lean on text-defined diagrams: Mermaid for structural, process, and timeline-style figures, and tables where a comparison is clearer in rows and columns than in nodes and arrows. In all cases, the figure is a claim, not decoration. If you read only the prose and skip the figures, you will miss part of the argument.

What Broke Next began as a question I asked every time I learned a new language: not “what can this do?” but “what was wrong before this

PREVIEW

existed?" The answers turned out to be the best history of computing I had ever read. I hope they do the same for you.

PREVIEW

Part I.

Introduction (Preview)

PREVIEW

1. Introduction — What This Book Is (and What It Is Not)

There is a kind of question that sounds obvious until you actually try to answer it: *Why do programming languages exist?*

The answer you learned in school, if you learned one, was probably something like: “To let humans express computations in terms they can understand, rather than in terms the machine understands.” That answer is true. It is also incomplete in a way that makes it almost useless as an explanation for *why there are so many of them*, *why they look so different from each other*, and *why languages that seem objectively worse than their successors somehow refuse to die*.

This book proposes a different framing: **programming languages are not designed in a vacuum; they are designed in response to specific, felt pressures**. They are responses to pain. The designers of FORTRAN were not dreaming abstractly about what mathematical notation might look like in silicon; they were watching trained engineers waste months hand-translating algebraic formulas into assembly code and deciding that this was an unconscionable waste of talent. The designers of Rust were not theorizing about ownership semantics simply because ownership semantics are beautiful (though they are); they were watching the web’s most important browser engine ship memory safety vulnerabilities in production year after year and deciding that the tools themselves needed to change.

The language is the artifact. The pressure is the story.

1.1. The Pressure Cycle

Every chapter in this book traces a version of the same cycle. Pressure accumulates — on programmers, on organizations, on hardware — until the existing tools are clearly inadequate. Someone (or some team, or some institution) designs a new tool that relieves that bottleneck. Relief arrives,

PREVIEW

and for a while things are better. Then the new tool enables new things to be attempted. Those new attempts create new constraints. The cycle continues.

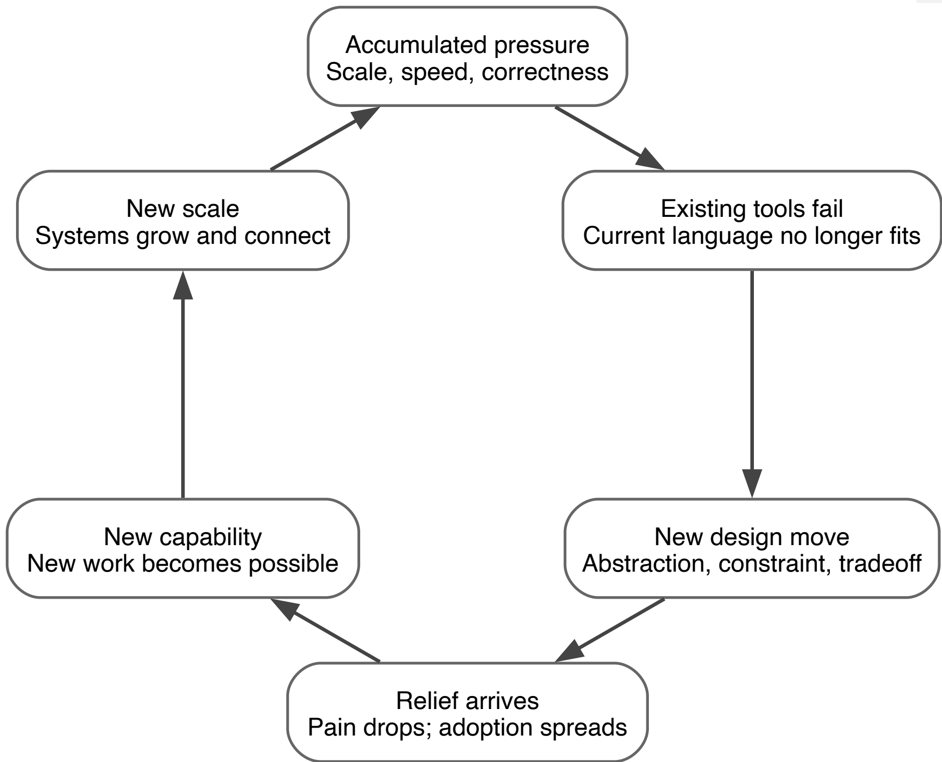

Figure 1.1.: The Pressure Cycle. Every language solves a pressure and introduces a tradeoff; that tradeoff, amplified by use, becomes the next pressure.

This cycle is not a criticism. It is not a lament about the inadequacy of human planning. It is simply what happens when a tool becomes successful enough to be used on problems its designers never imagined. LISP was designed for symbolic manipulation in AI research; its ideas about garbage collection showed up sixty years later in Java, Go, and Swift. C was designed to write a portable operating system; it became the lingua franca of all systems programming for five decades and counting. JavaScript was designed to add form validation to web pages; it now runs data centers.

The cycle does not respect its own initial conditions.

1.2. Why "Problem-Driven"?

Programming language textbooks often organize themselves around *features*: type systems, scoping rules, evaluation strategies, concurrency models. These are real and important. But features do not explain *choices*. Why does C have no array bounds checking? Not because Dennis Ritchie forgot, but because the constraint he was responding to — the need to write a portable operating system on hardware with kilobytes of RAM — made runtime bounds checking an unacceptable tax. Why does Java have garbage collection? Not because James Gosling didn't trust programmers, but because he was targeting embedded devices running consumer software and had watched C++ programmers write the same memory management bugs into every non-trivial program. The features follow from those constraints. Without that context, the features are arbitrary.

A problem-driven history asks: what was the *actual complaint*? What was hard in a way that mattered to real programs on real machines? Once you have that, the language design stops looking like a set of arbitrary choices and starts looking like what it actually is: a set of tradeoffs, some wise and some disastrous, made under specific constraints by people who had to ship something.

1.3. The Five Questions

Every chapter in this book can be read as an attempt to answer five questions about a particular moment in programming language history:

1. **What became hard?** — What pressure accumulated until the existing tools were clearly inadequate? Scale, speed, correctness, collaboration, deployment, hardware change — the problem takes many forms across different eras.
2. **Why were existing languages inadequate?** — Not because they were poorly designed, but because they were designed for different constraints. Every "inadequate" language in this book was, at some point, a breakthrough. Understanding why it stopped being adequate requires understanding what it was designed for.
3. **What design move solved part of the pain?** — The key insight. The bet. The tradeoff that the designers made when they decided to take

one kind of pain seriously and accept a different kind. This is usually the most interesting part.

4. **What new tradeoff appeared?** — Every solution is a new problem. Garbage collection solves memory leaks and introduces latency spikes. Static typing solves a class of runtime bugs and introduces a friction that slows prototyping. The tradeoff is not a failure of design; it is the fundamental nature of tradeoffs.
5. **What pressure led to the next wave?** — The chapter's ending is the next chapter's beginning. The pressure cycle continues.

1.4. The Book Map

Here is how the five parts relate to each other and to the problems they address:

The parts are roughly chronological, but the pattern does not respect chronology. Memory management, which shows up formally in Part III, was already a crisis in the LISP community in the 1960s. Gradual typing, which shows up in Part IV, is recapitulating arguments about type theory that ALGOL theorists were having in 1960. The history of programming languages is not a clean forward march; it is a spiral, revisiting the same fundamental tensions — control vs. safety, abstraction vs. performance, expressiveness vs. analyzability — at each new scale.

1.5. What Makes a Language Survive?

One more observation before we begin, because it is a question that will echo through the whole book: what makes a programming language survive?

The obvious answer — quality of design — is wrong. Or rather, it is incomplete. ALGOL was, by any serious measure, a better-designed language than COBOL. It had a cleaner syntax, a more rigorous semantics, a more elegant treatment of scope and block structure. ALGOL also failed to achieve industrial adoption and is used by almost nobody today. COBOL, which reads like a parody of English prose and whose type system would make a modern programmer wince, runs an estimated 95 billion transactions per day in the financial industry.

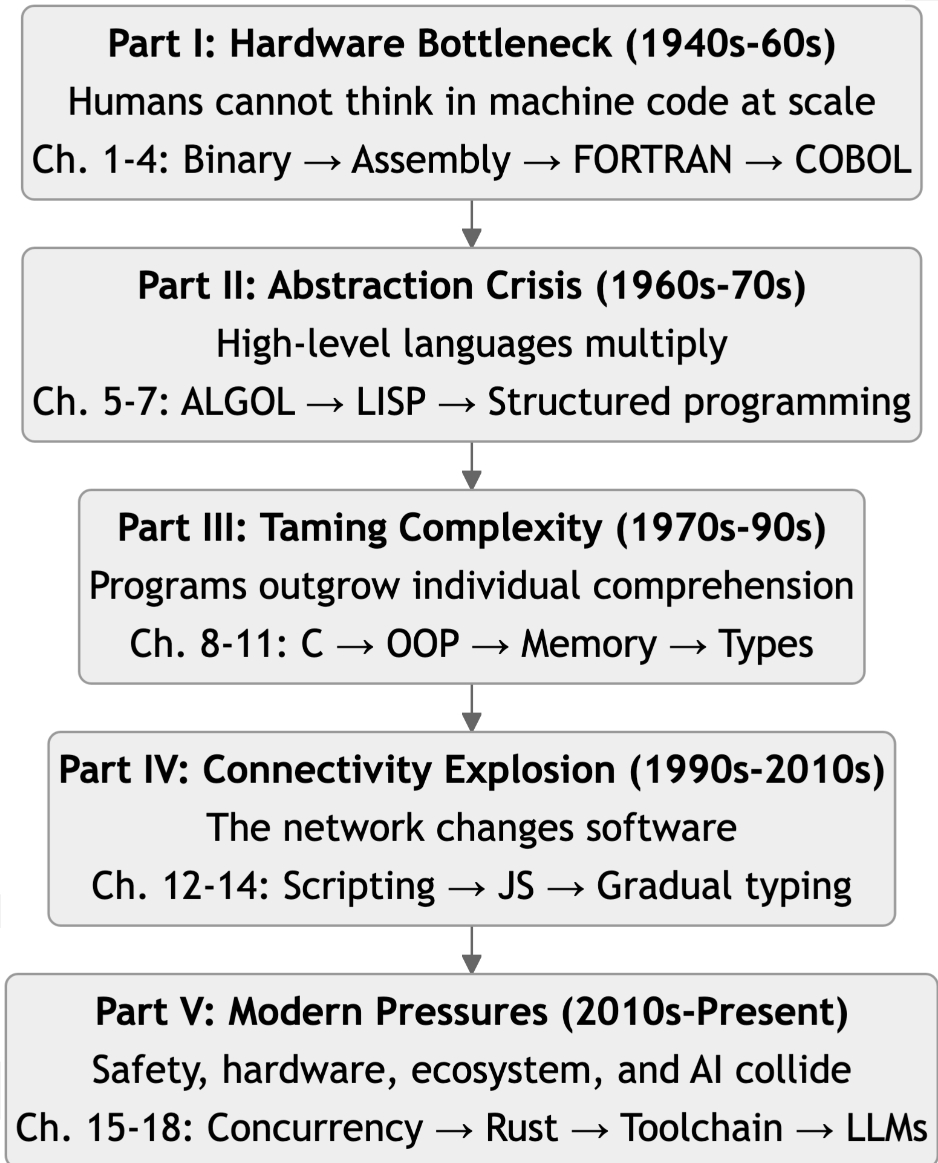

Figure 1.2.: Book Map. Five parts, five dominant pressures, roughly chronological but not strictly — pressures accumulate and recur.

A better answer is this: a language survives when it solves a *stable* problem *adequately* at the *moment the problem is most urgent*, and when the cost of replacing it exceeds the cost of maintaining it. Languages do not win by being great; they win by being present, cheap, and good enough when it counts. And once they win, they become infrastructure, and infrastructure does not get replaced just because something better exists.

This is uncomfortable news if you are hoping to design the perfect language. But it is clarifying news if you are trying to understand why the world looks the way it does. COBOL is not a mistake. COBOL is a perfect explanation of how software history actually works.

Let us begin.

PREVIEW

Part II.

The Hardware Bottleneck (Preview)

PREVIEW

2. Writing for the Machine

Ch 1 — Wiring, punch cards, raw binary. Programming as physical labor. Pressure: humans cannot reliably think in machine code at scale.

Before we had programming languages, we had programming. And programming, in the early days, was a form of craft labor that had more in common with typesetting or electrical wiring than with anything we would recognize today.

This is not a metaphor. In 1945, programming the ENIAC — the first programmable general-purpose electronic digital computer — literally meant rewiring it. The machine had no memory in the modern sense, no instruction set you could load and swap out. Its program *was* its physical configuration: patch cables connecting function units, switches set to encode constants, accumulators wired to receive and forward results. To change the computation, you changed the wiring. Programming was plumbing.

2.1. The Machine's Native Language

Subsequent machines stored their programs in memory, which was a revolution. You no longer had to rewire the hardware every time you wanted to solve a different problem. But "stored in memory" did not mean "stored in anything a human could read without effort." Memory was binary: a sequence of bits, each either on or off, stored in media like mercury delay lines, cathode-ray tubes, and later magnetic cores. The processor read these bits as instructions. It knew that a certain bit pattern meant "add the contents of register 3 to the contents of register 4." It did not know, and did not care, that the human who wrote that bit pattern had to look up 0110 0011 in a manual to figure out what it meant.

PREVIEW

Here is what programming looked like in raw machine code on an early stored-program computer. Each instruction is a number, and the number's bit pattern encodes both the operation and the operands:

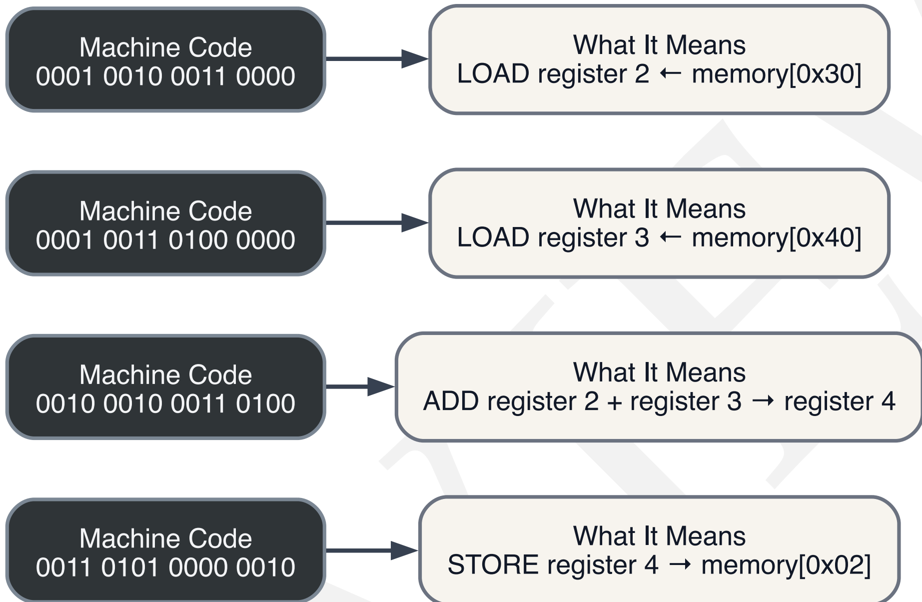

Figure 2.1.: Machine code versus its meaning. The same simple operation (add two numbers, store result) expressed as bit patterns versus human-readable description. The machine needs one; the human needs the other.

Notice what the programmer has to hold in their head to write even this simple four-instruction program: the binary encoding of each operation (not the name — the number), the binary address of each memory location, the register numbering scheme, and the mapping from their intent to this specific sequence. There is no "add these two variables." There is only the specific, concrete, hardware-determined instruction for that specific processor.

Now imagine doing this for a ballistic trajectory calculation. Or a payroll computation. Or a sorting routine over a thousand records. Early programmers did. And they made errors. Constantly.

2.2. Programming as Physical Labor

As stored-program machines spread, punch cards became the primary medium of programming for roughly three decades. This overlapped with the era of raw machine code and early assembly: you did not type your program into a terminal; you *punched* it into physical cardboard cards, one instruction (or one line of data) per card. A program was a deck of cards: physical objects you carried, dropped, sorted, rubber-banded together, and stored in labeled boxes.

A dropped deck — cards shuffled out of order — was a small disaster. If your cards were not numbered (and early cards often weren't), you might spend hours reassembling the program. If they were numbered but the numbering was wrong, the machine would execute instructions in the wrong sequence and produce nonsense. Physical entropy was a legitimate source of program bugs.

2.3. The First Error Budget

We need to understand something about this era that is easy to underestimate from the present: the cost of computing time was enormous. An hour on an early mainframe cost, in inflation-adjusted terms, tens of thousands of dollars. Access was rationed. Programs were submitted in batches, and you might wait overnight — or longer — to find out whether your program ran correctly.

This created an extraordinary demand for correctness *before* submission. If you sent a deck with a bug, you did not get an interactive debugger. You got a printout — sometimes just a core dump, a raw hexadecimal listing of memory at the moment of failure — and you went back to your paper notes and figured out what went wrong.

The economics forced a particular kind of programmer. The best early programmers were people who could *simulate the machine in their head* — who could trace through a sequence of binary instructions mentally, keeping track of register values and memory state, and predict what the machine would do. This was not a skill everyone had. It was not a skill that scaled.

Here is the thing about skills that don't scale: they become bottlenecks. A factory that can only run when a specific master craftsman is present

PREVIEW

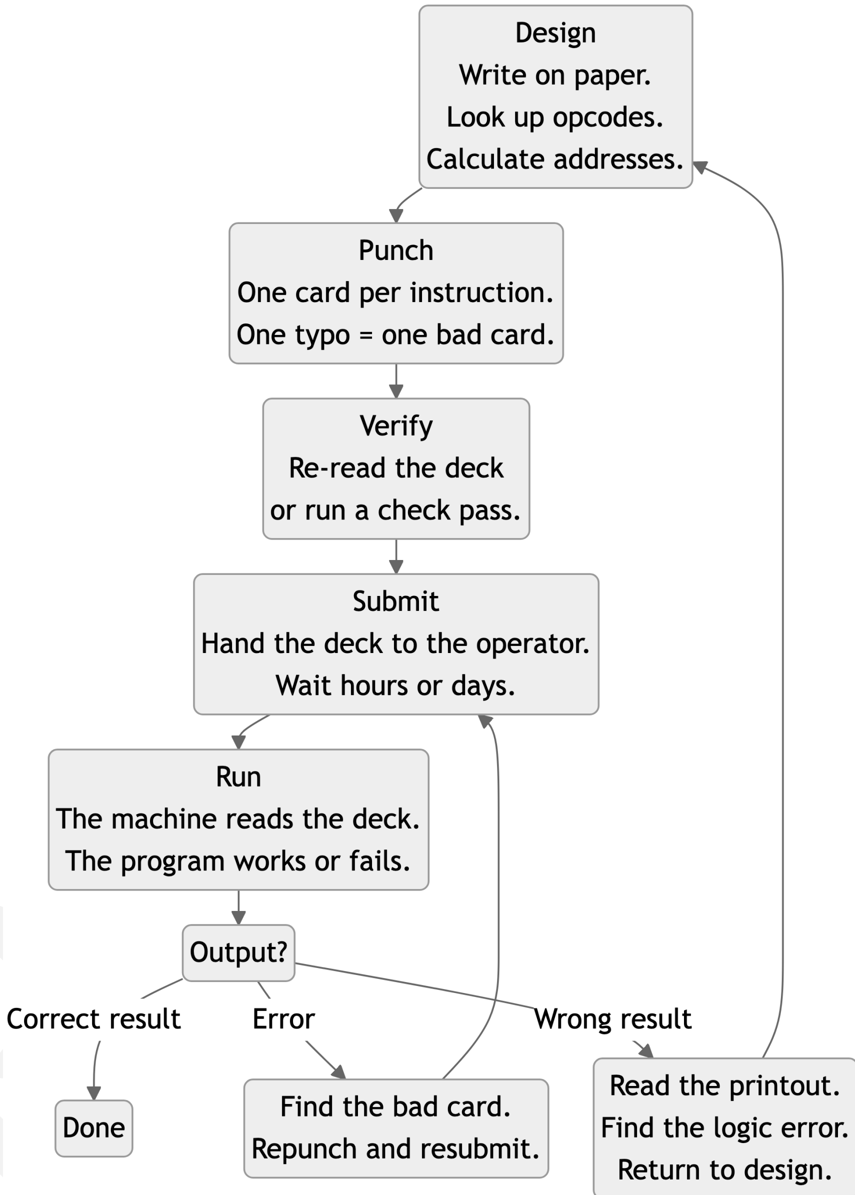

Figure 2.2.: The workflow of punch-card programming. Each stage is a physical action, each error requires restarting from the stage where the error occurred.

has a serious problem. The early computing industry had that problem in spades.

2.4. The Real Pressure: Cognitive Load at Scale

By the late 1940s, computers were already being deployed on genuinely important problems: calculating artillery firing tables and solving differential equations for physics. By the early 1950s, that range had widened to large administrative and data-processing tasks as well. These were not toy problems. They required programs that were hundreds or thousands of instructions long.

And here is where the real bottleneck emerged. A single programmer working in raw machine code, on a single problem, over a period of months, could stay mentally organized. They knew their program. They remembered what memory location 0x0043 held, because they had assigned it themselves three weeks ago and had written it in their notebook.

But what happened when: - The program grew to 5,000 instructions? - A second programmer had to maintain it? - The hardware changed and all the memory addresses shifted? - A new hire joined the team who had not been present for the original design?

What happened was that the program became opaque — not to the machine, but to the humans who needed to maintain it. The machine never had trouble reading binary. The machine *only* reads binary. The trouble was that humans are not machines, and treating them as though they were was producing catastrophic outcomes.

Maurice Wilkes, who built one of the first stored-program computers (the EDSAC at Cambridge), wrote candidly about this. He described programming as being in a “state of sin” — permanently guilty of the errors you were about to commit, but not yet knowing which ones. You were always wrong; the question was merely *how* wrong. This is not a description of a sustainable engineering discipline. This is a description of a craft that needed to be industrialized.

2.5. What Scale Broke First

The first thing that cracked was **address calculation**. In a machine code program of any length, you are constantly computing memory addresses. If your data lives at address 0x0040 and you want to access the fifth element of an array, you need to add 5 to 0x0040 and load from 0x0045. Fine for one access. When you have a routine that accesses a table of 200 elements, and that table's base address changes because you reorganized your program, you have to find every single reference to every address in the entire program and update it by hand. Miss one and you have a bug that produces silently wrong output.

The second thing that cracked was **subroutine linkage**. If you wrote a routine to multiply two numbers (early hardware often had no multiply instruction), you needed to call it from many places. But "calling" in raw machine code meant setting up the right registers, jumping to the right address, and then jumping back to the right return address when done. Each call required knowing concrete addresses. Every time you reorganized your program, every call site had to be updated.

The third thing that cracked was **debugging**. When your program produced a wrong answer, you were handed a printed listing of memory — thousands of hexadecimal numbers — and asked to figure out what went wrong. There was no variable name "sample_value" to inspect. There was only memory address 0x01B4 and the value 0x00FF, and you needed to remember — or look up in your paper notes — that 0x01B4 was where you had stored the sample value, and that 0x00FF was the wrong value.

Each of these failures is a failure of *translation*: the cost of constantly mapping between the human's representation of the problem and the machine's representation of the computation. The computer was asking humans to speak its language fluently and at scale. Humans were failing to do so, and the failures were expensive.

2.6. The Hired Computers

There is one more piece of context that matters for understanding this era: the human computers.

Before electronic computers, "computer" was a job title. Human computers were people — disproportionately women, particularly during World

War II — who performed calculation as their professional occupation. They sat at mechanical adding machines and desk calculators, working through mathematical procedures described in instruction sheets, checking each other's work. The ENIAC itself was initially "programmed" (in the original sense: given a procedure to follow) by six women who had been human computers: Jean Jennings Bartik, Frances Bilas Spence, Betty Holberton, Marlyn Wescoff Meltzer, Ruth Lichterman Teitelbaum, and Kathleen McNulty Mauchly Antonelli. They became the first programmers of an electronic computer because they already understood the work of mechanical computation.

This context matters because it tells you what the early computer operators and programmers were actually trying to do: they were automating human calculation work, and they were automating it in the same terms that human calculators used. The gap between "this is how a human calculator follows a procedure" and "this is how to encode that procedure in binary" was the original translation problem of computing. Everything that followed — every assembler, every compiler, every programming language ever designed — is in some sense an attempt to close that gap further.

2.7. The Setup for Everything That Follows

The bottleneck established in this era never really went away. It transformed. It got pushed up a level of abstraction each time a new tool made the previous level manageable. But the fundamental problem — that humans cannot reliably think about computation in the same terms that machines execute computation — is the founding problem of programming language design. Every language in this book is a response to some version of it.

The first response — the most direct one, the one that bought us a decade of relief before revealing its own inadequacies — was assembly language. And that is where we go next.

The pressure: humans cannot reliably translate mathematical intent into machine binary at the scale required by serious computation. Addresses are manual, subroutine linkage is manual, debugging is a binary archaeology exercise. The relief: give the instructions human-readable names. The next pressure: the names help, but the code still dies when the hardware changes.

3. One Step Up from the Metal

Ch 2 — Assembly gives mnemonics but zero portability. Pressure: code dies when hardware changes.

The first thing programmers did when they got access to a stored-program computer was write a program to help them write programs.

This sounds recursive to the point of absurdity, but it was the most natural thing in the world. The problem was not that programmers lacked cleverness or discipline. The problem was that the machine's native language — the binary encoding we traced in the last chapter — imposed a tax so high that much of the available human intelligence was being consumed just paying the tax, with nothing left for actual problem-solving. The first tool to reduce that tax was the assembler, and the language it processed was assembly.

3.1. Mnemonics: The First Relief

An assembler is a program that reads a text file full of symbolic instructions and translates them into the binary patterns the machine expects. Instead of writing `0001 0010 0011 0000` to load a value from memory, you write `LOAD R2, 0x30`. Instead of tracking that `0x30` is the address where you stored the sample value, you write `LOAD R2, sample_value` and let the assembler figure out the address.

The relief was immediate and substantial. Programs became readable — not *comfortable* to read, not *pleasant*, but readable. A programmer looking at assembly code a month later had some chance of understanding what it did. A programmer looking at binary had effectively no chance.

The key innovations of assembly language, from the programmer's perspective:

PREVIEW

Mnemonics for opcodes. `ADD`, `SUB`, `MOV`, `JMP`. Every CPU has its own set — these are not standardized across hardware — but within a given architecture, you learned the names once and used them forever. The names aligned with human intention. `MOV A, B` moves a value from B to A. The binary equivalent was a number you had to look up.

Symbolic labels. Instead of calculating "the address of the next instruction minus the current address to compute this branch offset," you wrote `JMP loop_start` and the assembler did the arithmetic. When you reorganized your program and the addresses shifted, you re-ran the assembler and it recalculated everything. The manual address catastrophe from Chapter 1 was largely solved.

Named constants and storage locations. You could write `sample_value DB 0.05` — "define a byte, call it sample_value, initialize to 0.05" — and reference it by name throughout your program. The assembler assigned the actual memory address.

For programmers in the late 1940s and early 1950s, this was genuinely transformative. The cognitive tax dropped sharply. Programs that would have taken weeks to write in raw binary could be written in days. Programs that would have been too complex to debug could now be debugged, because you had names to think with.

3.2. Two Assembly Dialects, Same Problem

Here is where we hit the first wall. Assembly is not one language. It is a family of languages, one per CPU architecture. When you wrote assembly for the IBM 704, you wrote it in 704 assembly. When IBM released the 709, it had a different instruction set — subtly different, but different — and porting 704 assembly to it still meant real work. When the 7090 came out (a transistorized version of the 709), the same basic problem remained: even within a machine family, assembly code was entangled with the details of the hardware.

This is the portability problem in its sharpest form: the language you write in is the language of a specific piece of hardware, and hardware changes. The figure below uses later architectures to make the point vividly, but the problem itself was already visible in the 1950s.

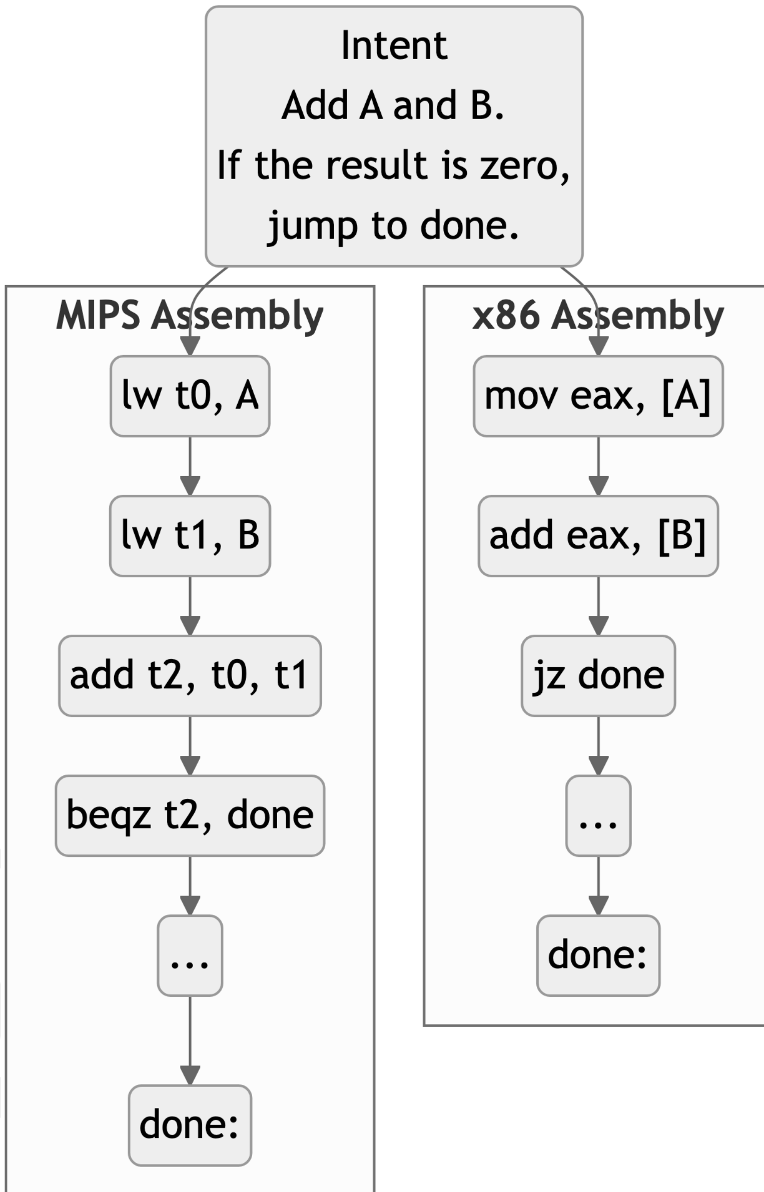

Figure 3.1.: A modern illustration of the assembly portability problem. The same logic (add two numbers and branch if result is zero) expressed in two different assembly languages: x86 and MIPS. Different instruction names, different register conventions, different branch syntax — but identical intent.

The divergence goes deeper than syntax. Each architecture has different registers (different numbers, different widths, different purposes), different memory addressing modes, different conventions for how subroutines are called and return values passed, and different ways of handling conditions and branches. Writing assembly for one architecture teaches you almost nothing about writing assembly for another, except the general mindset.

For a programmer at an organization that owned one machine, this was manageable. You learned your machine's assembly; you wrote for it; you didn't think about other machines. But the industry was changing fast. New machines were appearing every few years, each with a different architecture. Government agencies and universities that had bought one machine were being sold the next generation. Scientific programs written at great expense had to be rewritten from scratch — not because the algorithms changed, but because the hardware did.

3.3. The Business Case for Portability

In the mid-1950s, this was not just an inconvenience. It was a business crisis.

Consider the situation of a large organization — say, the U.S. Air Force, which was among the heaviest early users of digital computers for logistics and weapons calculations. You have invested years of programmer time in assembly code for a machine that is now being replaced by a newer model with better performance. Your code is your intellectual capital. It represents real work. And it is *completely useless* on the new machine.

You have two options: rewrite everything (enormously expensive), or keep running the old machine even after the vendor stops supporting it (which creates its own escalating costs). Neither option is good. The demand for a third option — *write once, run anywhere*, as a later generation would put it — was intense, real, and economic.

The organizations feeling this demand were also the organizations with the budget to fund research. Which is why, when a team at IBM proposed an experiment that seemed nearly impossible — write a compiler that could translate mathematical formulas directly into machine code as efficient as what expert assembly programmers would write by hand — there were people willing to fund it and time willing to try it.

3.4. What Assembly Got Right (and Why It Still Matters)

Before we move on, we should be fair to assembly. The jump from raw machine code to assembly was a genuine leap, and assembly's virtues are not just historical.

Assembly language gives you *total transparency*. There is no abstraction layer between your code and the machine's behavior. Every instruction you write maps to exactly one machine instruction. You know, with certainty, how long your program will take to run (if you know your hardware's timing specifications). You know exactly how much memory it uses. You know exactly what registers are in play at any moment. For certain classes of problems — device drivers, interrupt handlers, performance-critical inner loops, embedded systems with no runtime to speak of — this transparency is not a luxury. It is a necessity.

That is why assembly never died. You can write a modern iOS app without writing a single line of assembly. You can also write a modern x86 processor firmware in almost nothing *but* assembly. The hardware interface — the place where software makes contact with the physical machine — has always been, and remains, a place where assembly is not just relevant but often required.

The point is not that assembly was inadequate. The point is that assembly was adequate for a set of problems that was becoming a smaller and smaller fraction of the total problem space. As computers moved from scientific calculation to business data processing to general purpose use, the fraction of programs that needed total hardware transparency shrank, while the fraction of programs that needed portability, maintainability, and expressibility at a higher level grew.

Assembly solved the wrong half of the problem.

3.5. The Cognitive Model Mismatch

There is a deeper issue with assembly that goes beyond portability, and understanding it matters for everything that comes after.

Assembly language is organized around the machine's model of computation: registers, memory cells, and operations on small chunks of data. The *programmer's* model of computation — especially for the scientists and

engineers who were the primary users of early computers — was organized around algebra: equations with named variables, functions of multiple arguments, expressions that could be nested and composed.

When you write a ballistic trajectory calculation, you think:

$$x(t) = x_0 + v_{0x} \cdot t + ½ \cdot a_x \cdot t^2$$

When you write it in assembly, you are forced to decompose this into a sequence of low-level operations: - Load `x0` into a register - Load `v0x` into another register - Load `t` into another register - Multiply `v0x` by `t`, store somewhere - Load `ax` into a register - Multiply by `t`, store somewhere - Multiply that by `t`, store somewhere - Multiply by the constant for ½…

…and so on, for fifteen to twenty instructions, carefully managing which registers hold which intermediate results, making sure you don't clobber a value you still need, and tracking the accumulated floating-point precision loss across all these operations.

The gap between "the physicist's equation" and "the assembly sequence that computes it" is enormous, and every instruction you write in that gap is an opportunity for an error that produces a silently wrong result.

This mismatch — between the *shape of the programmer's thinking* and the *shape of the machine's execution* — is what FORTRAN was designed to close. Not perfectly. Not in all domains. But enough to change everything.

3.6. The Setup for FORTRAN

In 1954, a team at IBM led by John Backus began working on a system that would allow programmers to write their calculations in something approaching normal mathematical notation and have a program — a compiler — translate that notation into efficient machine code automatically. The proposal was greeted with deep skepticism, for a reason that will seem familiar: expert assembly programmers doubted that any automatic system could match what they could do by hand.

They were wrong. But they were wrong in an interesting way, and understanding exactly what Backus's team got right, and exactly what they could not yet do, is the story of the next chapter.

The pressure: assembly programs die when hardware changes. The human investment in machine-specific code is destroyed by hardware upgrades. The cognitive model of algebra doesn't fit the cognitive model of register operations. The relief: give the programmer a language that looks like their problem, not like the machine. The next pressure: one high-level language cannot serve all domains.

PREVIEW

4. Let the Machine Do the Translation

Ch 3 — FORTRAN bets a compiler can match hand-written assembly — and wins. Pressure: one high-level language cannot serve all domains.

The skepticism was reasonable. It was almost unanimous. And it was wrong.

In 1954, John Backus submitted the internal IBM proposal that launched the FORTRAN project: "Preliminary Report, Specifications for the IBM Mathematical FORmula TRANslating System." The claim at the heart of that proposal was audacious: a program — a compiler — could automatically translate mathematical formulas written in a notation resembling algebra into machine code that was, for most practical purposes, as efficient as what an expert programmer would write by hand.

The experts did not believe it. They had spent years developing the skills required to coax good performance out of assembly code. They understood, in their bones, every trick the machine offered: which memory access patterns were fast, which register combinations avoided conflicts, how to unroll loops to amortize overhead. They were certain that no automatic translation system could match their craftsmanship.

They were right about their craftsmanship. They were wrong about the compiler.

4.1. The Compiler as a Bet

What Backus and his team were actually proposing was a *bet* about the nature of programming inefficiency. The skeptics assumed that programming bottlenecks were about machine efficiency — about the gap between naive code and optimally hand-tuned code. Backus's insight was that the

bottleneck was increasingly about *programmer* efficiency — about the enormous tax imposed on scientific talent by the requirement to manually map algebra to assembly.

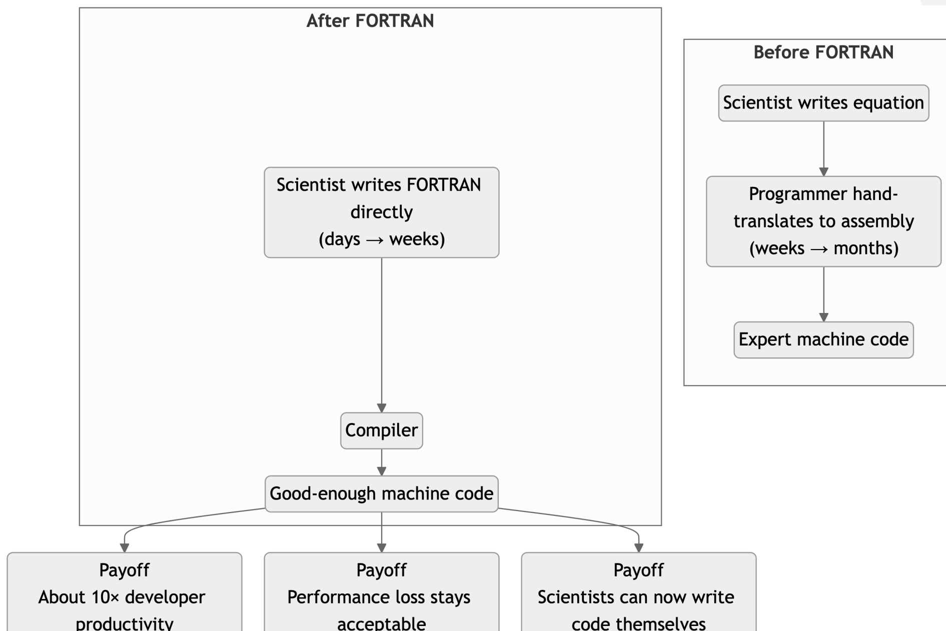

Figure 4.1.: The FORTRAN bet. The compiler need not produce perfect assembly; it needs to produce assembly that is good enough that the productivity gain from writing in FORTRAN outweighs any residual performance loss.

The bet worked because it was calibrated correctly. The FORTRAN compiler did *not* produce assembly as good as the best hand-written code. But it produced assembly that was good enough — often close enough to hand-tuned performance for the productivity gain to dominate — while allowing the scientist or engineer to write the program in algebraic notation, at a fraction of the time required for assembly.

When you are calculating whether your compiler is good enough, you are not just measuring the runtime of the generated code. You are measuring the *total cost* of getting a working program: the time to write it, the time to debug it, the time to maintain it, the time to modify it when the problem changes. Assembly wins on raw machine efficiency. FORTRAN won on total cost.

4.2. What FORTRAN Actually Looked Like

The first FORTRAN programs look surprisingly modern to a contemporary programmer — more modern, in many ways, than assembly code from the same era. This is because they were written in terms of the programmer's problem, not the machine's structure.

Here is what FORTRAN code for a simple computation looked like:

```
C  COMPUTE TRAJECTORY AT TIME T
   DT = 0.01
   X = X0
   V = V0
   DO 10 I = 1, 1000
      F = MASS * GRAV - DRAG * V * ABS(V)
      A = F / MASS
      V = V + A * DT
      X = X + V * DT
   10 CONTINUE
```

And here is roughly what that same code looks like in IBM 704 assembly — the machine FORTRAN was first implemented for:

Several things are worth noticing about the FORTRAN version:

Named variables. `MASS`, `GRAV`, `DRAG`, `V`, `X` — the physicist's variable names, directly in the code. No manual mapping from name to address. The compiler manages address assignment.

Infix arithmetic. `MASS * GRAV - DRAG * V * ABS(V)` is closer to how the physicist wrote the equation on their blackboard than anything assembly could offer. You write the formula; the compiler figures out the instruction sequence.

The DO loop. This was one of FORTRAN's killer features. The assembly programmer's loop required manual counter management: load the counter, test it, conditionally branch, increment, store. The FORTRAN `DO 10 I = 1, 1000` said: "execute the following block 1000 times, with I going from 1 to 1000." The compiler generated the loop machinery.

No registers in sight. The programmer wrote nothing about which registers to use. The compiler's register allocator figured it out. This was the part that most worried the skeptics — register allocation was considered

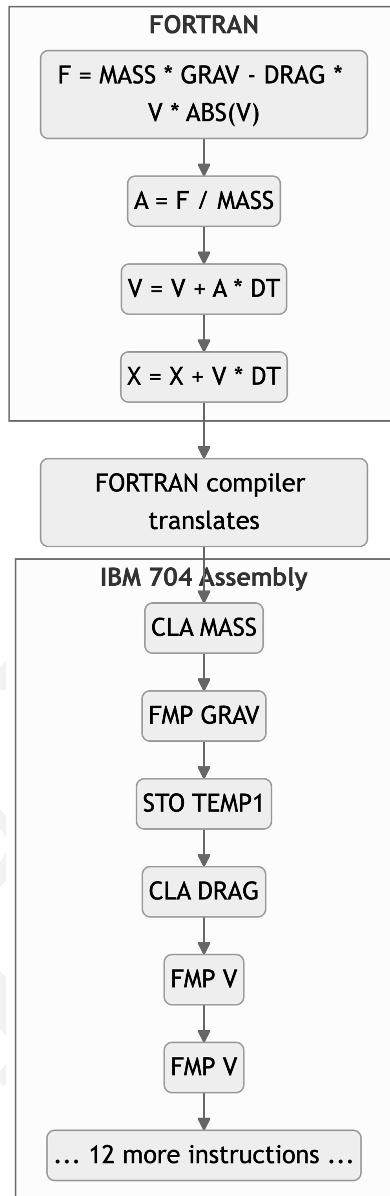

Figure 4.2.: FORTRAN versus assembly for the same computation. The FORTRAN is not just shorter; it maps to the physicist's mental model. The assembly requires a separate mental model of register allocation and memory access.

PREVIEW

one of the highest arts of assembly programming — and it was one of the parts the FORTRAN team got surprisingly right.

4.3. The Skeptics and the Proof

John Backus tells a revealing story about the early reception of FORTRAN. When his team demonstrated the compiler to a group of experienced IBM programmers, the programmers were polite. They acknowledged the results. And then they went home and waited for the catch — for the class of program that would reveal the compiler's inadequacy.

The most common test was to write a program in both FORTRAN and hand-optimized assembly and compare performance. For simple computations, the FORTRAN-generated code could come surprisingly close to hand assembly. For more complex programs with irregular memory access patterns, the gap was often larger. But here is the thing: the FORTRAN version took a fraction of the time to write, and it was orders of magnitude easier to debug.

The FORTRAN I compiler, released in 1957, shipped with a manual that contained the phrase "FORTRAN programs will generally be as efficient as those written in assembly language." This was marketing optimism; it was not always true. But it was true enough, often enough, that it changed the industry.

What changed first was who could program. Before FORTRAN, programming a scientific computation required both domain knowledge *and* assembly programming skill. These were separate skills; having one did not imply having the other. Scientists who needed computations done had to communicate their algorithms to programmers who could translate them to assembly — a process fraught with translation errors, because neither party fully understood the other's domain. After FORTRAN, a physicist or engineer could, with some training, write their own programs. The translation step was replaced by compilation. The domain expert became the programmer.

4.4. The IBM 704 and the First Ecosystem Lock-In

FORTRAN was initially implemented only for the IBM 704. This was practical necessity — you have to start somewhere — but it created the first instance of an ecosystem dynamic that would recur throughout programming language history: the language was tied to a specific platform, and the platform's adoption drove the language's adoption.

The IBM 704 was one of the dominant scientific computers of its era. If you did serious numerical computation in the late 1950s, you probably did it on a 704 or wanted to. FORTRAN, therefore, spread with the 704 into universities, national laboratories, aerospace companies, and government agencies. By the time IBM released the 709 and 7090, the installed base of FORTRAN code was large enough that IBM made sure the new compilers were compatible. The language had become infrastructure.

Here we see the first clear example of a pattern that will recur throughout the book: a language's adoption curve is shaped not just by its quality but by the platform it ships with. The best language that ships on the wrong platform is less successful than a mediocre language that ships on the dominant one.

4.5. What FORTRAN Could Not Do

Having won the bet on numerical computation, FORTRAN encountered its limits almost immediately, and those limits were domain-shaped.

FORTRAN was designed by and for people who thought in mathematical notation. Arrays of numbers. Algebraic formulas. Loops over indices. It was very good at this. It was also very good at getting fast code for this, because its structure mapped cleanly to what 1950s hardware was good at: sequential arithmetic on arrays.

What FORTRAN was not good at:

Business data processing. Banks, insurance companies, and retailers needed to process records — accounts, policies, transactions. A record is not an array of numbers. It is a heterogeneous collection of fields: a name, an account number, a balance, a date, a transaction type. FORTRAN's type system and data structures were not designed for this. You could *force* it into FORTRAN, but it was awkward.

Text manipulation. FORTRAN treated text as an afterthought. Strings were stored in integer arrays, manipulated with bit operations. The programs were possible to write; they were terrible to read and maintain.

Database-style operations. Sorting records by multiple fields, joining tables, generating formatted reports — these operations had structure that FORTRAN's abstractions did not naturally capture.

The organizations doing business data processing were just as interested in computers as the scientific computing community. And their problems had a different shape. What they needed was a language designed around records, files, and business operations — not around vectors, matrices, and algebraic formulas.

What they needed, in other words, was COBOL. And COBOL was being built in the same period, for exactly these reasons, by a committee that understood the failure mode of domain-specific design.

4.6. The Lasting Lesson of FORTRAN

FORTRAN's legacy is not primarily the language itself — though modern Fortran (capital F, standards-track, still actively maintained) is genuinely good at numerical computation and still used in scientific computing. FORTRAN's legacy is the *proof of concept it established*.

The bet Backus made was this: *a programmer who writes in a higher-level language, trading some machine efficiency for cognitive efficiency, will in most cases produce better outcomes than a programmer writing assembly*. That bet was never really in dispute after 1957. Every high-level language that followed — every compiler, interpreter, and runtime — rests on FORTRAN's proof that the bet was correct.

The second lesson of FORTRAN is more uncomfortable: *domain specificity is a feature with a cost*. By being perfectly suited to scientific computation, FORTRAN was poorly suited to everything else. That is not a design failure. It is a fundamental tradeoff. The question of whether to design a universal language (and accept that it will be mediocre everywhere) or a domain-specific language (and accept that it will leave other domains to other tools) has never been fully resolved, and we will watch programming language designers argue about it for the next seven decades.

The pressure: one high-level language cannot serve all domains. FORTRAN solved scientific computation; business data processing had a completely different shape. The relief: design a language for business records and operations. The next pressure: when a language solves a stable problem perfectly, replacing it becomes harder than maintaining it.

PREVIEW

5. The Language That Refused to Die

Ch 4 — COBOL solves business constraints so well it becomes infrastructure. Pressure: when a language solves a stable problem perfectly, replacing it becomes harder than maintaining it.

COBOL is the most successful programming language nobody wants to talk about.

You will not find COBOL in the "trending" section of any developer survey. Nobody posts excitedly about their new COBOL side project. There are no COBOL influencers on YouTube, no hot takes about COBOL on Hacker News (well, occasionally, but not in the way you'd want). The language is, to most programmers under fifty, a punchline — a relic, the butt of jokes about ancient mainframes and dusty government systems.

It is also, depending on how you count, running more active code than almost anything else on earth. The most cited estimate puts it at 95 billion COBOL transactions per day. ATM withdrawals. Payroll calculations. Insurance claims. Social Security payments. The financial plumbing of every developed economy runs, to a remarkable degree, on COBOL. When you tap your debit card, there is a non-trivial probability that COBOL code is involved in clearing the transaction before the receipt prints.

To understand why, you need to understand what COBOL was designed to solve, why it solved it so well, and why that very success created the trap it cannot escape.

5.1. Grace Hopper and the Business Computability Problem

COBOL did not emerge from academic computer science. It emerged from a military-industrial committee, with major influence from Grace Hopper's

PREVIEW

work at UNIVAC, in direct response to a specific complaint: the computer industry was balkanizing, and businesses were suffering for it.

By 1959, computers from different manufacturers were incompatible with each other. Programs written for one vendor's machine could not run on another's. This was fine for scientific computing, where each institution typically owned one machine and ran programs written for it. It was a disaster for business computing, where organizations needed to share data, transfer systems between machines as they upgraded, and write programs that would outlast the hardware they first ran on.

The U.S. Department of Defense backed the 1959 effort that led to the Conference on Data Systems Languages (CODASYL) and the first COBOL specification. The committee included representatives from government, academia, and industry. Grace Hopper, who had already built FLOW-MATIC — an early business-oriented programming language whose influence on COBOL was direct — was a central figure in the intellectual background of the effort. The explicit goal was a language that would be:

1. **Machine-independent** — programs written in it should run on any machine that had a COBOL compiler
2. **Business-oriented** — suited to record processing, arithmetic on decimal numbers, report generation
3. **Readable by non-programmers** — business managers should be able to read and verify COBOL code

That third goal is the strange one, and it shaped COBOL's syntax in ways that technical programmers have been mocking for sixty years and that turned out to be unexpectedly important.

5.2. The Domain Fit

Before we get to the syntax, let us understand what COBOL was designed to compute, because the domain determines almost everything.

Business data processing in 1960 was primarily:

File processing. You have a magnetic tape (or, later, disk) full of records — employee records, account records, transaction records. Each record has a fixed structure: a set of fields with defined positions and types. You read through the file, process each record, possibly update it, possibly write to an output file. The core loop is: read record, process record, write record.

Decimal arithmetic on money. Financial calculations must be exact to the cent. Floating-point arithmetic, which is how FORTRAN (and essentially every scientific language) handled numbers, introduces rounding errors that are unacceptable in financial contexts. If your payroll system uses floating-point arithmetic, someone's paycheck might be off by a penny due to floating-point rounding. This is not a theoretical concern; it is a regulatory and business concern.

Report generation. The output of most business programs was printed reports: ledgers, summaries, invoices, statements. These reports had precise formatting requirements — column alignment, subtotals, page headers, running totals.

Control breaks. A "control break" is what happens when you are processing a sorted file and the key field changes. You were processing accounts for customer 1001; now you have hit account 1002. Before processing the new customer's records, you need to produce a subtotal for customer 1001. This pattern — hierarchical grouping with summary operations at each level — is extremely common in business reporting and essentially absent from scientific computing.

COBOL was designed from the ground up for these operations. Let us look at what that actually means in the language:

The `PIC 9(7)V99` is a COBOL picture clause. It means: a number with 7 digits before the decimal point and 2 digits after, stored as an exact decimal (not floating-point). This is not a curiosity. It is a precise encoding of what business arithmetic actually requires: exact decimal representation of monetary values.

5.3. The Verbose Syntax and Why It Was Right

COBOL code is famously verbose. Here is a fragment that adds two numbers:

```
ADD EMPLOYEE-COUNT TO DEPARTMENT-TOTAL
    GIVING GRAND-TOTAL
```

Compare to FORTRAN:

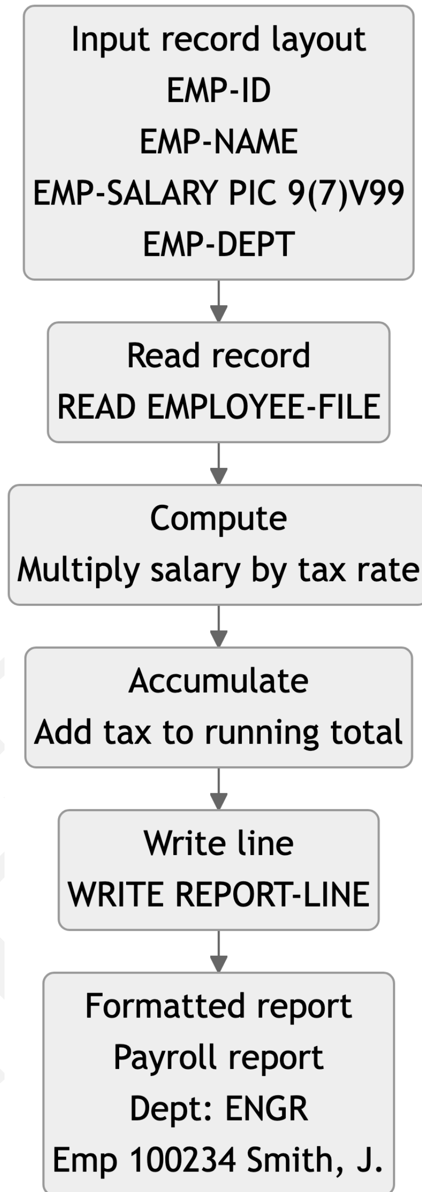

Figure 5.1.: COBOL's domain fit. The language maps directly onto the structure of business data processing: input files, record layouts, processing logic, and formatted output reports.

PREVIEW

```
GRAND_TOTAL = EMPLOYEE_COUNT + DEPARTMENT_TOTAL
```

The COBOL version is longer, uses English-ish verbs, and looks like a business memo rather than an algebraic expression. This drove mathematically-trained programmers absolutely crazy. It still does.

But recall the third design goal: business managers should be able to read and verify the code. This goal was not sentimental. In the business context of 1960, the people who understood the business logic — the accounting rules, the regulatory requirements, the correct formula for calculating overtime pay — were not programmers. They were accountants and business managers. The programmers who wrote the code were technical specialists who often did not fully understand the business domain they were encoding.

The verbose, English-like syntax of COBOL was a form of documentation that non-programmers could, in principle, audit. `MULTIPLY EMP-SALARY BY TAX-RATE GIVING TAX-AMOUNT ROUNDED` is something an accountant can read and verify as correct (or incorrect). The algebraic equivalent `TAX_AMOUNT = EMP_SALARY * TAX_RATE` is comprehensible to a programmer but requires translation for the accountant.

Whether this goal was ever fully achieved is debatable. Business managers reading and auditing COBOL in practice was always more aspiration than reality. But the *side effect* was significant: COBOL programs, even old ones, tend to be more self-documenting than equivalent code in more terse languages. A COBOL program from 1975 is often more readable, in isolation, than a C program from the same era.

5.4. The Infrastructure Trap

COBOL's success created a trap so perfect that it is still springing sixty years later.

Here is how the trap works:

Notice the pattern of replacement waves. Every decade or so, a new candidate appears that is going to replace COBOL: PL/I in the 1960s (IBM's ambitious unified language), 4GLs in the 1970s, C++ in the late 1980s, Java Enterprise Edition in the late 1990s. Each wave fails — not because the

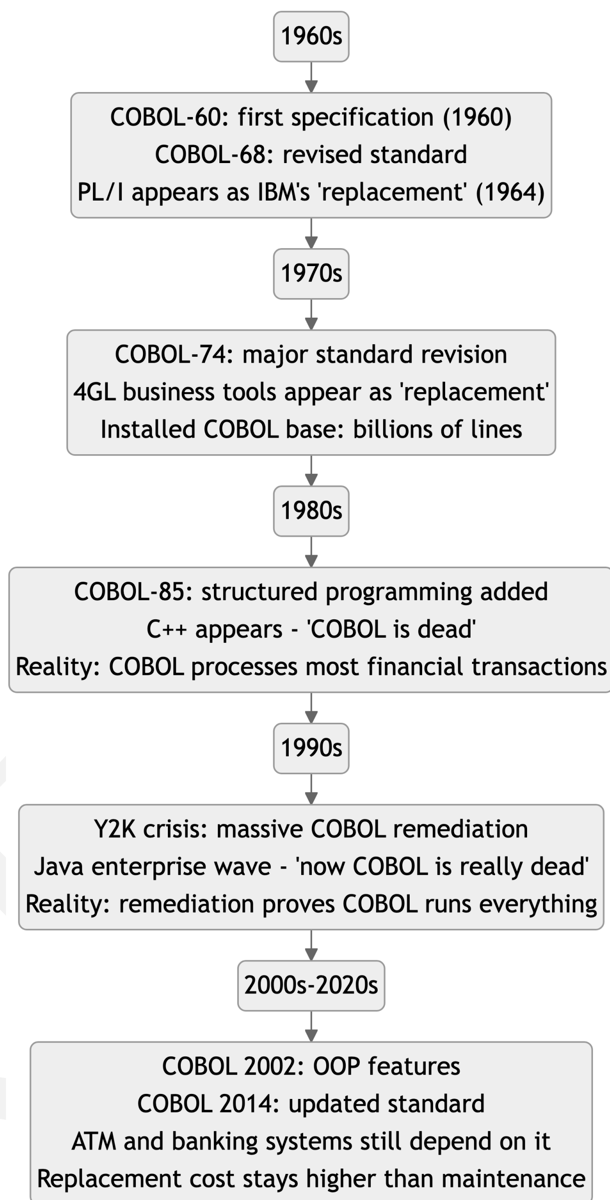

Figure 5.2.: The COBOL survival timeline. Each decade, the argument for replacement weakens as the cost of the installed base grows. The language outlives every 'successor.'

challenger is a bad language, but because the economics of replacement don't work.

Here is the arithmetic that kills every replacement attempt. A large bank or insurance company might have 40 to 100 million lines of COBOL. Rewriting that code from scratch is not a software project; it is a multi-year program involving hundreds of developers, at a cost of tens or hundreds of millions of dollars. The rewrite must produce software that is *exactly* as correct as the original — because the original is handling payroll, insurance claims, and financial transactions, and a correctness error can mean regulatory penalties, lawsuits, or material harm to people. And the rewrite must be done while the old system continues to operate, because you cannot simply turn off the payroll system for eighteen months while you rebuild it.

The UK government tried to modernize its child support payment system in 2003. Twelve years and £500 million later, the project was abandoned. The replacement system was never deployed. The COBOL it was meant to replace is still running.

This is not just a COBOL story. It is a story about **the cost of replacing working software at scale**. COBOL is simply the clearest example because the gap between "the language we would choose today" and "the language running our infrastructure" is so wide, and the infrastructure so deep, that the gap is visible even to non-programmers.

5.5. What COBOL Got Right That Nobody Admits

There is a version of the COBOL story that presents it as a failure — a dead end, a warning about language design. This version is wrong, or at least incomplete.

COBOL got several things right that the industry took decades to rediscover:

Exact decimal arithmetic. COBOL's fixed-point decimal arithmetic, which was mocked by mathematically sophisticated programmers, is the correct approach for financial computation. IEEE 754 floating-point, which virtually every other language uses, is wrong for money. The banking industry has had to implement decimal arithmetic libraries in Java, Python, and C++ to correct this. COBOL had it built in from the beginning.

Structured data records. The COBOL data division, with its hierarchical record structures and picture clauses, is a form of schema definition. You define the structure of your data — field names, types, widths, nested structures — before you write any processing code. Modern database schemas, JSON Schema definitions, and Protocol Buffer definitions are all doing something similar. COBOL did it first.

Self-documenting programs. COBOL's verbose English-like syntax produces programs that are often readable by domain experts, not just programmers. This is unusual among programming languages, and it turns out to have significant value when programs need to be maintained for decades by people who did not write them.

Batch processing performance. COBOL compilers are extremely good at sequential file processing. On mainframe hardware, a COBOL program processing millions of records sequentially can be competitive with anything written in more modern languages, because the problem maps so cleanly to the language's idioms.

5.6. The Y2K Moment

In the late 1990s, the world suddenly cared very much about COBOL. The Year 2000 problem — the risk that systems storing years as two digits would malfunction when 00 arrived — was, at its core, a COBOL problem. Not entirely — other languages had the same issue — but predominantly, because the systems that most relied on two-digit year fields were old business systems written in COBOL.

The Y2K remediation effort cost an estimated $300 to $600 billion worldwide. Most of it was COBOL: finding every date field, analyzing every date comparison, updating the logic, testing the fixes. It required going into systems that had been written in 1968 and not substantially touched since, understanding their logic, and making targeted changes.

The Y2K effort accomplished two things simultaneously. It was a massive and mostly successful exercise in COBOL maintenance, which demonstrated that the language and its codebase could be worked with. And it introduced a generation of programmers to COBOL who had never expected to encounter it — revealing, often to their surprise, that COBOL code was not as incomprehensible as its reputation suggested.

After Y2K, the COBOL conversation shifted from "when do we replace it?" to a more honest "do we actually replace it?" The more people looked at what replacement would entail — the business logic embedded in decades of COBOL, the edge cases accumulated through years of production use, the regulatory requirements encoded in subroutine calls — the more the cost of replacement looked less like an engineering problem and more like an institutional memory problem. The COBOL program is not just code. It is an accumulation of decisions about how the business works. Replacing it requires not just rewriting the code but rediscovering all those decisions.

5.7. The Lesson for Language Designers

COBOL teaches a lesson that language designers find uncomfortable: **the merit of a language is not the primary determinant of its long-term adoption or survival**.

COBOL survived because it was in the right place at the right time (the business computing boom of the 1960s and 1970s), it solved its domain's problems well enough (not perfectly, but well enough), and it became infrastructure before any successor could establish itself. Once it was infrastructure, the economics of replacement made it nearly impossible to displace, regardless of what successor languages offered.

This is not special pleading for COBOL. It is an observation about how infrastructure works. The lesson applies equally to C (still the dominant language for systems programming despite languages with much better safety properties), to SQL (still the dominant language for relational data despite thirty years of NoSQL alternatives), and to JavaScript (still the only client-side web language despite its well-documented deficiencies).

The early chapters of this book are about escaping the machine. FORTRAN escaped assembly. COBOL escaped FORTRAN's domain limitations. But as we will see throughout the rest of the book, escape never fully succeeds. What you escape from becomes what you inherit. And what you inherit is what the next generation of programmers has to maintain, replace, or work around.

COBOL is still here. It will be here when this book goes out of print.

PREVIEW

The pressure: when a language solves a stable problem perfectly and accumulates enough installed base, replacing it becomes harder than maintaining it. The infrastructure trap is real and it is waiting for every successful language. The relief: acceptance that maintenance is engineering, not failure. The next pressure: the theoretical foundations of programming languages themselves are unclear, and without agreed foundations, the discipline cannot mature.

PREVIEW