



The First Programming Languages: Crash Course Computer Science #11

Crash Course: Computer Science

<https://youtube.com/watch?v=RU1u-js7db8>

<https://nerdfighteria.info/v/RU1u-js7db8>

This episode is brought to you by CuriosityStream.

Hi, I'm Carrie Anne and welcome to CrashCourse Computer Science! So far, for most of this series, we've focused on hardware -- the physical components of computing -- things like: electricity and circuits, registers and RAM, ALUs and CPUs.

But programming at the hardware level is cumbersome and inflexible, so programmers wanted a more versatile way to program computers - what you might call a "softer" medium. That's right, we're going to talk about Software! INTRO In episode 8, we walked through a simple program for the CPU we designed.

The very first instruction to be executed, the one at memory address 0, was 0010 1110. As we discussed, the first four bits of an instruction is the operation code, or OPCODE for short. On our hypothetical CPU, 0010 indicated a LOAD_A instruction -- which moves a value from memory into Register A.

The second set of four bits defines the memory location, in this case, 1110, which is 14 in decimal. So what these eight numbers really mean is "LOAD Address 14 into Register A". We're just using two different languages.

You can think of it like English and Morse Code. "Hello" and "... . -.. -.. ---" mean the same thing -- hello! -- they're just encoded differently. English and Morse Code also have different levels of complexity. English has 26 different letters in its alphabet and way more possible sounds.

Morse only has dots and dashes. But, they can convey the same information, and computer languages are similar. As we've seen, computer hardware can only handle raw, binary instructions.

This is the "language" computer processors natively speak. In fact, it's the only language they're able to speak. It's called Machine Language or Machine Code.

In the early days of computing, people had to write entire programs in machine code. More specifically, they'd first write a high-level version of a program on paper, in English, for example "retrieve the next sale from memory, then add this to the running total for the day, week and year, then calculate any tax to be added" ...and so on. An informal, high-level description of a program like this is called Pseudo-Code.

Then, when the program was all figured out on paper, they'd painstakingly expand and translate it into binary machine code by hand, using things like opcode tables. After the translation was complete, the program could be fed into the computer and run. As you might imagine, people quickly got fed up with this process.

So, by the late 1940s and into the 50s, programmers had developed slightly higher-level languages that were more human-readable. Opcodes were given simple names, called mnemonics, which were followed by operands, to form instructions. So instead of having to write instructions as a bunch of 1's and 0's, programmers could write something like "LOAD_A 14".

We used this mnemonic in Episode 8 because it's so much easier to understand! Of course, a CPU has no idea what "LOAD_A 14" is. It doesn't understand text-based language, only binary.

And so programmers came up with a clever trick. They created reusable helper programs, in binary, that read in text-based instructions, and assemble them into the corresponding binary instructions automatically. This program is called -- you guessed it -- an Assembler.

It reads in a program written in an Assembly Language and converts it to native machine code. 3:07 "LOAD_A 14" is one example of an assembly instruction. Over time, Assemblers gained new features that made programming even easier. One nifty feature is automatically figuring out JUMP addresses.

This was an example program I used in episode 8: Notice how our JUMP NEGATIVE instruction jumps to address 5, and our regular JUMP goes to address 2. The problem is, if we add more code to the beginning of this program, all of the addresses would change. That's a huge pain if you ever want to update your program!

And so an assembler does away with raw jump addresses, and lets you insert little labels that can be jumped to. When this program is passed into the assembler, it does the work of figuring out all of the jump addresses. Now the programmer can focus more on programming and less on the underlying mechanics under the hood enabling more sophisticated things to be built by hiding unnecessary complexity.

As we've done many times in this series, we're once again moving up another level of abstraction. A NEW LEVEL OF ABSTRACTION! However, even with nifty assembler features like auto-linking JUMPs to labels, Assembly Languages are still a thin veneer over machine code.

In general, each assembly language instruction converts directly to a corresponding machine instruction -- a one-to-one mapping -- so it's inherently tied to the underlying hardware. And the assembler still forces programmers to think about which registers and memory locations they will use. If you suddenly needed an extra value, you might have to change a lot of code to fit it in.

Let's go to the Thought Bubble. This problem did not escape Dr. Grace Hopper.

As a US naval officer, she was one of the first programmers on the Harvard Mark 1 computer, which we talked about in Episode 2. This was a colossal, electro-mechanical beast completed in 1944 as part of the allied war effort. Programs were stored and fed into the computer on punched paper tape.

By the way, as you can see, they "patched" some bugs in this program by literally putting patches of paper over the holes on the punch tape. The Mark 1's instruction set was so primitive, there weren't even JUMP instructions. To create code that repeated the same operation multiple times, you'd tape the two ends of the punched tape together, creating a physical loop.

In other words, programming the Mark 1 was kind of a nightmare! After the war, Hopper continued to work at the forefront of computing. To unleash the potential of computers, she designed a high-level programming language called "Arithmetic Language Version 0", or A-0 for short.

Assembly languages have direct, one-to-one mapping to machine instructions. But, a single line of a high-level programming language might result in dozens of instructions being executed by the CPU. To perform this complex translation, Hopper built the first compiler in 1952.

This is a specialized program that transforms "source" code written in a programming language into a low-level language, like assembly or the binary "machine code" that the CPU can directly process. Thanks, Thought Bubble. So, despite the promise of easier programming, many people were skeptical of Hopper's idea.

She once said, "I had a running compiler and nobody would touch it... they carefully told me, computers could only do arithmetic; they



The First Programming Languages: Crash Course Computer Science #11

Crash Course: Computer Science

<https://youtube.com/watch?v=RU1u-js7db8>

<https://nerdfighteria.info/v/RU1u-js7db8>

could not do programs." But the idea was a good one, and soon many efforts were underway to craft new programming languages -- today there are hundreds! Sadly, there are no surviving examples of A-0 code, so we'll use Python, a modern programming language, as an example. Let's say we want to add two numbers and save that value.

Remember, in assembly code, we had to fetch values from memory, deal with registers, and other low-level details. But this same program can be written in python like so: Notice how there are no registers or memory locations to deal with -- the compiler takes care of that stuff, abstracting away a lot of low-level and unnecessary complexity. The programmer just creates abstractions for needed memory locations, known as variables, and gives them names.

So now we can just take our two numbers, store them in variables we give names to -- in this case, I picked a and b but those variables could be anything - and then add those together, saving the result in c, another variable I created. It might be that the compiler assigns Register A under the hood to store the value in a, but I don't need to know about it! Out of sight, out of mind!

It was an important historical milestone, but A-0 and its later variants weren't widely used. FORTRAN, derived from "Formula Translation", was released by IBM a few years later, in 1957, and came to dominate early computer programming. John Backus, the FORTRAN project director, said: "Much of my work has come from being lazy.

I didn't like writing programs, and so ... I started work on a programming system to make it easier to write programs." You know, typical lazy person. They're always creating their own programming systems.

Anyway, on average, programs written in FORTRAN were 20 times shorter than equivalent handwritten assembly code. Then the FORTRAN Compiler would translate and expand that into native machine code. The community was skeptical that the performance would be as good as hand written code, but the fact that programmers could write more code more quickly, made it an easy choice economically: trading a small increase in computation time for a significant decrease in programmer time.

Of course, IBM was in the business of selling computers, and so initially, FORTRAN code could only be compiled and run on IBM computers. And most programming languages and compilers of the 1950s could only run on a single type of computer. So, if you upgraded your computer, you'd often have to re-write all the code too!

In response, computer experts from industry, academia and government formed a consortium in 1959 -- the Committee on Data Systems Languages, advised by our friend Grace Hopper -- to guide the development of a common programming language that could be used across different machines. The result was the high-level, easy to use, Common Business-Oriented Language, or COBOL for short. To deal with different underlying hardware, each computing architecture needed its own COBOL compiler.

But critically, these compilers could all accept the same COBOL source code, no matter what computer it was run on. This notion is called write once, run anywhere. It's true of most programming languages today, a benefit of moving away from assembly and machine code, which is still CPU specific.

The biggest impact of all this was reducing computing's barrier to entry. Before high level programming languages existed, it was a realm exclusive to computer experts and enthusiasts. And it was

often their full time profession.

But now, scientists, engineers, doctors, economists, teachers, and many others could incorporate computation into their work. Thanks to these languages, computing went from a cumbersome and esoteric discipline to a general purpose and accessible tool. At the same time, abstraction in programming allowed those computer experts -- now "professional programmers" -- to create increasingly sophisticated programs, which would have taken millions, tens of millions, or even more lines of assembly code.

Now, this history didn't end in 1959. In fact, a golden era in programming language design jump started, evolving in lockstep with dramatic advances in computer hardware. In the 1960s, we had languages like ALGOL, LISP and BASIC.

In the 70's: Pascal, C and Smalltalk were released. The 80s gave us C++, Objective-C, and Perl. And the 90's: python, ruby, and Java.

And the new millennium has seen the rise of Swift, C#, and Go - not to be confused with Let it Go and Pokemon Go. Anyway, some of these might sound familiar -- many are still around today. It's extremely likely that the web browser you're using right now was written in C++ or Objective-C.

That list I just gave is the tip of the iceberg. And languages with fancy, new features are proposed all the time. Each new language attempts to leverage new and clever abstractions to make some aspect of programming easier or more powerful, or take advantage of emerging technologies and platforms, so that more people can do more amazing things, more quickly.

Many consider the holy grail of programming to be the use of "plain ol' English", where you can literally just speak what you want the computer to do, it figures it out, and executes it. This kind of intelligent system is science fiction... for now. And fans of 2001: A Space Odyssey may be okay with that.

Now that you know all about programming languages, we're going to deep dive for the next couple of episodes, and we'll continue to build your understanding of how programming languages, and the software they create, are used to do cool and unbelievable things. See you next week. Hey guys, this week's episode was brought to you by CuriosityStream which is a streaming service full of documentaries and nonfiction titles from some really great filmmakers, including exclusive originals.

I just watched a great series called "Digits" hosted by our friend Derek Muller. It's all about the Internet - from its origins, to the proliferation of the Internet of Things, to ethical, or white hat, hacking. And it even includes some special guest appearances... like that John Green guy you keep mentioning in the comments.

And Curiosity Stream offers unlimited access starting at \$2.99 a month, and for you guys, the first two months are free if you sign up at curiositystream.com/crashcourse and use the promo code "crash course" during the sign-up process.