Early programming languages
Agenda

• Earliest *described* programming language: Konrad Zuse’s Plankalkül, 1933-48

• Predecessors to earliest practical programming languages
  • Sort-Merge generator, (Betty Holberton, 1951)
  • Compiling Routine A-0 (Grace Hopper, 1951-1952)

• Some early “real languages”
  • FLOW-MATIC (1955)
    … which begat COBOL…
  • FORTRAN (1956)
    (the oldest language still in use)
  • Lisp (1958)
    … which begat Scheme, Haskell, OCaml, Erlang….
  • ALGOL (1958)
    … which begat nearly everything else.
Let’s start at the start, with the first programming language ever designed: Plankalkül, meaning roughly “plan calculus”

It was designed by the guy on the right here, Konrad Zuse, a German civil engineer who built a series of computers (called Z1, Z2, Z3, and Z4, and so on) starting in 1933

His Z3, completed in 1941, was the world’s first programmable, fully automatic digital computer. It was never put in use, because the Nazi government considered it “not war-important.”
The Z1
Z1 replica in Berlin

https://www.youtube.com/watch?v=RG2WLDxi6wg

(sorry about the awful music)
Zuse’s company

https://www.youtube.com/watch?v=n8Yo-wD-QTo
Plankalkül

- Zuse (unlike the government that employed him in the 1930s) was remarkably forward-looking. He realized early on that it would be useful to have a formal system to describe algorithms.

- What he came up with was a notation scheme derived from the German logician Frege’s *Begriffsschrift*, or “concept writing.”

- Zuse never tried to actually implement Plankalkül on any of his Z-series machines – it’s mainly interesting because it was the first attempt to define a real formal system for describing algorithms.

- Zuse didn’t publish his work on Plankalkül, but he had spoken about it with many of the people who participated in the design of ALGOL (which we’ll talk about in a bit). He was quite salty indeed about not being cited by the ALGOL developers.
Fig. 4.

„Der weiße König kann einen Zug machen, ohne dabei in Schach zu kommen.“

\[ P 148 \quad R(V) \Rightarrow R 148 \quad (1) \]

\[
\begin{array}{c|cc}
V & 0 & 0 \\
A & 5 & 0 \\
\end{array}
\]

\[ \frac{\dot{x}}{V} \left[ (x \in V) \land (x = L0) \right] \Rightarrow Z \quad (2) \]

\[
\begin{array}{c|ccc}
V & 0 & 1 \\
A & 5 & 3 \\
\end{array}
\]

\[ (\exists x) \left[ (x \in V) \land R 17 (Z, x) \land (x = 0) \lor x \right] \quad (3) \]

\[
\begin{array}{c|cccc}
V & 0 & 0 & 1 & 1.3 \\
A & 4 & 4 & 5 & 2 & 3 & 0 \\
\end{array}
\]

\[ \land \overline{E y} \left[ (y \in V) \land y \land R 128 (v, y, x) \right] \quad (4) \]

\[
\begin{array}{c|ccc}
V & 1.3 & 0 & 0 \\
A & 4 & 5 & 0 \\
\end{array}
\]
Plankalkül

P1 \( \max_3 (V_0[:8.0], V_1[:8.0], V_2[:8.0]) \rightarrow R_0[:8.0] \)
\( \max(V_0[:8.0], V_1[:8.0]) \rightarrow Z_1[:8.0] \)
\( \max(Z_1[:8.0], V_2[:8.0]) \rightarrow R_0[:8.0] \)
END

P2 \( \max (V_0[:8.0], V_1[:8.0]) \rightarrow R_0[:8.0] \)
\( V_0[:8.0] \rightarrow Z_1[:8.0] \)
\( (Z_1[:8.0] < V_1[:8.0]) \rightarrow V_1[:8.0] \rightarrow Z_1[:8.0] \)
\( Z_1[:8.0] \rightarrow R_0[:8.0] \)
END
Plankalkül

You can find a decent overview of how Plankalkül, in its original notation, was meant to work here:

http://cl-informatik.uibk.ac.at/teaching/ss15/bob/reports/ss15-BB.pdf

(this is a seminar report prepared by a Bernhard Behr, a grad student at the Universität Innsbruck).
Last thoughts on Zuse

• After the war, Zuse successfully commercialized his computers. His company, Zuse KG, made and sold hundreds of computers between 1949 and 1967, when he sold the company to Siemens.

• His Z10 was the first working commercially-sold computer anywhere

• His Z22 was the first computer to use magnetic storage for memory

• In his 1969 book Rechnender Raum (in English, Calculating Space) he was among the first to suggest that the universe might be itself running on a computer.

• Zuse on the subject of AI: “The danger of computers becoming like humans is not as great as the danger of humans becoming like computers.”
Grace Hopper
(an aside on Hopper)
IF AND ONLY IF SUFFICIENT DEMAND BECOMES EVIDENT, SPERRY RAND WILL CONSIDER UNDERTAKING TRANSLATORS FROM THE LANGUAGE DEFINED BY THE COMMITTEE OR FROM LANGUAGES PROPOSED BY OTHER COMPUTER MANUFACTURERS. IT IS THE CLEAR AIM OF SPERRY RAND TO ULTIMATELY DESIGN COMPILERS WHICH ACCEPT GOOD ENGLISH AS THEIR INPUT. IT IS NOT THE INTENTION OF SPERRY RAND TO PROGRESS BACKWARDS BY INTRODUCING MATHEMATICAL SYMBOLS AND BANNING ADJECTIVES, ETC., ETC.

SPERRY RAND WOULD LIKE TO SUPPORT THE DEVELOPMENT OF A COMMON DATA-PROCESSING LANGUAGE, BUT SUGGESTS THAT SUCH A LANGUAGE WILL PROBABLY HAVE TO BE DEVELOPED BY PEOPLE FAMILIAR WITH BUSINESS DATA-PROCESSING PROBLEMS AND NOT BE MATHEMATICIANS AND PROGRAMMERS.
(an aside on Hopper)
Compiling Routine A-0

• Hopper’s first tool for using UNIVAC to program UNIVAC was what she called “Compiling Routine A-0” (1952), which has been called the first working programming language compiler ever.

• (though as we’ll see, it’s kind of more of a linker than a compiler)

• Remington Rand refused to give her a team to work on programming language design, because they did not believe it was possible.

• Inspirations: John Mauchly’s “Short Code,” which was a system for specifying arithmetic statements as arithmetic rather than UNIVAC machine code

• Betty Holberton’s “Sort-Merge Generator,” about which more in a second.
Holberton’s Sort-Merge Generator

“I think the first step to tell us that we could actually use a computer to write programs was Betty Holberton's "Sort-Merge Generator." You fed it the specifications of the files you were operating on, and the Sort-Merge Generator produced the program to do the sorting and merging, including not only carrying out the operations, but also managing all of the input and output in the various tape units, and it contained, I think, what I would define as the first version of a virtual memory in that it made use of overlays automatically without being told to by the programmer. I think that meant a great deal to me. It meant that I could do these things automatically; that you could make a computer write a program.” -- Grace Hopper
Compiling Routine A-0

1. Compiling routine A-0.
Compiling Routine A-0
Compiling Routine A-0

• So what did compiling routine A-0 do?

• UNIVAC programmers had already developed a set of subroutines to do common tasks.

• Quote Hopper: “If I needed a sine subroutine, angle less than $\pi/4$, I’d whistle at Dick and say ‘can I have your sine subroutine?’ and I’d copy it out of his notebook.

• In practice, there were a couple of problems with this method.
Compiling Routine A-0

- “We were using subroutines. We were copying routines from one program into an other. There were two things wrong with that technique: one was that the subroutines were all started at line 0 and went on sequentially from there. When you copied them into another program, you therefore had to add to all those addresses as you copied them into the new program—you had to add to all those addresses. And programmers are lousy adders!”

- “The second thing that inhibited this was that programmers are lousy copyists! And it was amazing how many times a 4 would turn into a delta which was our space symbol, or into an A—and even Bs turned into 13s. All sorts of interesting things happened when programmers tried to copy subroutines. And there of course stood a gadget whose whole purpose was to copy things accurately and do addition. And it therefore seemed sensible, instead of having programmers copy the subroutines, to have the computer copy the subroutines. Out of that came the A-0 compiler.”
Compiling Routine A-0

- A-0 code doesn’t look like much – it only took numeric input, so code was just sets of three memory addresses, one referring to the memory location of the subroutine used, one referring to the memory location of the data to be operated on, one referring to the output location.
Compiling Routine A-0

• Why did it have to be a single-pass compiler?

• Input and output was stored on these tape drives, and unspooling the tape, rewinding the spool, then putting it back in the drive was time-consuming.
A-0 saved enough time for UNIVAC programmers that management gave Hopper her own team: the “Automatic Programming Group.”

Hopper despised this name, since they weren’t doing “automatic programming” – they were just developing tools for programmers to program more efficiently. Marketing loved it, though.

In December of 1953 Hopper put together a proposal for a new programming language that would allow programmers to give instructions using variable-length English language words strung together in sentences. Management told her that her team couldn’t do it, because it was impossible.
“That December 1953 report proposed to management that mathematical programs should be written in mathematical notation, data processing programs should be written in English statements, and we would be delighted to supply the two corresponding compilers to translate to machine code. I was promptly told that I could not do that. And this time the reason was that computers couldn't understand English words. Well, I allowed as to how I never expected any computer anywhere to understand anything; that all I intended to do was to compare bit patterns.

But it was not until January of 1955 that we were able to write a formal proposal for writing a data processing compiler. I keep it with me for a couple of reasons. One, to remind me that I always have to push into the future; and the other to remind me that any given moment in time there's always a line out here, and that line represents what your management will believe at that moment in time. And just you step over the line and you don't get the budget.”
FLOW-MATIC

• So, in response to being denied funding, Hopper did what she generally did: she wrote it anyway, or at least a demo version.

• We got our little compiler running. It wouldn't take more than 20 statements. And on the back of this report, we put a nice little program in English. And we said, "Dear Kind Management: If you come down to the machine room, we'll be delighted to run this program for you." And it read: INPUT INVENTORY FILE A; PRICE FILE B; OUTPUT PRICED INVENTORY FILE C. COMPARE PRODUCT #A WITH PRODUCT #B. IF GREATER, GO TO OPERATION 10; IF EQUAL, GO TO OPERATION 5; OTHERWISE GO TO OPERATION 2. TRANSFER A TO D; WRITE ITEM D; JUMP TO OPERATION 8. It ended up: REWIND B; CLOSE OUT FILE C AND D; and STOP.
Yes to English, no to French and German

- Management approved the English version on the spot. But when they demoed French and German, management was horrified…

- “Have you figured out what happened to that? That hit the fan!! It was absolutely obvious that a respectable American computer, built in Philadelphia, Pennsylvania, could not possibly understand French or German! And it took us four months to say no, no, no, no! We wouldn't think of programming it in anything but English.”
FORTRAN

• First proposed by IBM engineer John Backus in 1953
• IBM gave him a large skilled team right from the start
• Draft specification done by 1954
• First compiler completed 1957
• Main focus for Backus’s team: designing a compiler that could compete with hand-written assembly
Although UNIVAC was the industry leader in early 1952, IBM was far and away the biggest computer company by the end of the 1950s.

They had won the contract for SAGE (Semi-Autonomous Ground Environment), the system that correlated radar reports from across the world and warned NORAD of potential Soviet attacks.

SAGE was, by at least an order of magnitude, the largest programming project up to that date.
SAGE
FORTRAN – Need for Speed

• John Backus was designing a language for “scientific computing” – which meant doing hard mathematical manipulations to relatively small data sets.

• Practically speaking, “scientific computing” for IBM meant things like rapidly calculating trajectories – things like being able to tell, quickly, whether a radar trace observed over Greenland was a commercial airliner or an incoming Soviet bomber.

• Therefore, machine code generated by FORTRAN had to be FAST – ideally, as fast as hand-written assembly.
“It was our belief that if FORTRAN, during its first months, were to translate any reasonable "scientific" source program into an object program only half as fast as its hand coded counterpart, then acceptance of our system would be in serious danger. [...] To this day I believe that our emphasis on object program efficiency rather than on language design was basically correct. I believe that had we failed to produce efficient programs, the widespread use of languages like FORTRAN would have been seriously delayed.” – John Backus
Skepticism about “automatic programming”

• Another reason it had to be fast was that software developers had gotten burned about “automatic programming” before.

“Experience with slow "automatic programming" systems, plus their own experience with the problems of organizing loops and address modification, had convinced programmers that efficient programming was something that could not be automated. Another reason that "automatic programming" was not taken seriously by the computing community came from the energetic public relations efforts of some visionaries to spread the word that their "automatic programming" systems had almost human abilities to understand the language and needs of the user; whereas closer inspection of these same systems would often reveal a complex, exception-ridden performer of clerical tasks which was both difficult to use and inefficient.” – John Backus
Skepticism about “automatic programming”
FORTRAN – 1st optimizing compiler!

• https://www.youtube.com/watch?v=KohboWwrsXg&t=11m30s
FORTRAN

SDS Fortran II Coding Form

Prepared By: ________________________________

Problem Identification: ______________________

Date: ________________________________

Page: ______ of ______

1. CALCULATE AND TYPE THE ROOTS OF A SET OF QUADRATIC EQUATIONS

2. DIMENSION A(I:100), B(I:100), C(I:100)

3. ACCEPT TAPE 1, N, (A(I)), B(I), C(I), J=1, N

4. FORMAT(I3/(1P3E12.4))

5. DO 2 I=1,N

6. DISCR=B(I)*& 2-4.*A(I)*C(I)

7. C IF ROOTS ARE COMPLEX GOTO 3 IF NOT GOTO 4

8. IF(DISCR)<3.9,4

9. ROOT1RL=-B(I)/(2.*A(I))

10. ROOT2RL=ROOT1RL

11. ROOT1IM=SQRRTF(-DISCR)/(2.*A(I))

12. ROOT2IM=-ROOT1IM

13. GOTO 2

14. ROOT1RL=(-B(I)+SQRRTF(DISCR))/(2.*A(I))

15. ROOT2RL=(-B(I)-SQRRTF(DISCR))/(2.*A(I))

16. ROOT1IM=0.

17. ROOT2IM=0.

18. TYPE S,A(I),B(I),C(I),ROOT1RL,ROOT1IM,ROOT2RL,ROOT2IM

19. FORMAT(3H A=PE11.4, 3H B=PE11.4, 3H C=PE11.4/6X, 7H RROOT1= (+1E11.4,4H)+I(1E11.4,4H) ROOT2=(1E11.4,4H)+I(1E11.4,4H))

20. STOP

21. END
Surprising optimizations

“When later on we began to get fragments of compiled programs out of the system, we were often astonished at the surprising transformations in the indexing operations and in the arrangement of the computation which the compiler made, changes which made the object program efficient but which we would not have thought to make as programmers ourselves. Transfers of control appeared which corresponded to no source statement, expressions were radically rearranged, and the same DO statement might produce no instructions in the object program in one context, and in another it would produce many instructions in different places in the program.” – John Backus
Some basic compiler optimizations

• Strength reduction: Say you’ve entered the statement $x = x \times 2$. We know that multiplication is expensive and some other operations are relatively cheap. What command should the compiler actually generate?

• Loop unrolling: The programmer types up this for loop:

  ```c
  for (int i = 0; i < 6; i++){
      printf("hello!\n");
  }
  ```

What code might we generate?
Some basic compiler optimizations

• Function inlining: Say you have a short function that’s reused many times in a given program (maybe a function to calculate cosines). There’s a little bit of overhead each time it’s called – you’ve got to save the return address and the values in any registers that the function might change, then you have to jump back to the return address, then you have to reload the saved registers. If you just put the body of the function everywhere the function call appears, you can gain a little bit of speed.

• Register optimization (this is a big one). You have a fixed number $n$ of registers available. Accessing values in registers is thousands of times faster than accessing values in memory. If you can rearrange or reduce the code you generate to never use more than $n$ variables at a time, you can get huge speed gains, because you never have to access anything but registers.
I’d like to talk really quickly about ALGOL. This language never saw much widespread use, but there are few things about it that make it important:

It was the first language defined by a formal metalanguage – the so-called Backus-Naur form. In every previous language, what commands in the language meant were what the compiler came out with when you entered that command. ALGOL was the first language where we used a rigorous formalism to describe what the language did.

The syntax of every language in common use today – C, C++, Java, Python – is largely inspired by ALGOL.

It was a famously well-designed language; unlike FORTRAN and (especially) COBOL. ALGOL was designed with the aim of making it not just useful, but also provably correct. Here’s C.A.R. Hoare, the designer of quicksort, on ALGOL:

“Here is a language so far ahead of its time, that it was not only an improvement on its predecessors, but also on nearly all its successors.”
Sample ALGOL-60 code.

Example 1

procedure euler (fct, sum, eps, tim); value eps, tim; integer tim; real procedure fct; real sum, eps;

comment euler computes the sum of fct(i) for i from zero up to infinity by means of a suitably refined euler transformation. The summation is stopped as soon as tim times in succession the absolute value of the terms of the transformed series are found to be less than eps. Hence, one should provide a function fct with one integer argument, an upper bound eps, and an integer tim. The output is the sum sum. euler is particularly efficient in the case of a slowly convergent or divergent alternating series;

begin integer i, k, n, t; array m[0:15]; real mn, mp, ds;
i := n := t := 0; m[0] := fct(0); sum := m[0]/2;
nextterm: i := i + 1; mn := fct(i);

for k := 0 step 1 until n do
begin mp := (mn + m[k])/2; m[k] := mn; mn := mp end means;

if (abs(mn) < abs(m[n])) ∧ (n < 15) then
begin ds := mn/2; n := n + 1; m[n] := mn end accept
else ds := mn;
sum := sum + ds;
if abs(ds) < eps then t := t + 1 else t := 0;
if t < tim then go to nextterm

end euler
Backus on FORTRAN: A mistake

“My own opinion as to the effect of FORTRAN on later languages and the collective impact of such languages on programming generally is not a popular opinion. That viewpoint is the subject of a long paper (Backus, 1978). I now regard all conventional languages (e.g., the FORTRANs, the ALGOLs, their successors and derivatives) as increasingly complex elaborations of the style of programming dictated by the von Neumann computer. These "von Neumann languages" create enormous, unnecessary intellectual roadblocks in thinking about programs and in creating the higher level combining forms required in a really powerful programming methodology. Von Neumann languages constantly keep our noses pressed in the dirt of address computation and the separate computation of single words, whereas we should be focusing on the form and content of the overall result we are trying to produce. We have come to regard the DO, FOR, WHILE statements and the like as powerful tools, whereas they are in fact weak palliatives that are necessary to make the primitive von Neuman style of programming viable at all.”
LISP

• John Backus thought FORTRAN- and ALGOL-style languages were a mistake. LISP was the first language from the other major programming language paradigm: functional programming.

• How does functional programming work? There is another mathematical formulation of computing, completely unlike Turing’s Turing Machine, that is provably exactly as powerful as the Turing Machine. I’ll let the computerphile guys explain:

• https://www.youtube.com/watch?v=eis11j_iGMs
LISP – accidentally implemented

• We saw earlier that it took a large team of top-notch computer programmers three years to implement FORTRAN. Here’s the story of how LISP was implemented almost by mistake
“[A] way to show that LISP was neater than Turing machines was to write a universal LISP function and show that it is briefer and more comprehensible than the description of a universal Turing machine. This was the LISP function eval(e, a), which computes the value of a LISP expression e, the second argument a being a list of assignments of values to variables, (a is needed to make the recursion work.) Writing eval required inventing a notation representing LISP functions as LISP data, and such a notation was devised for the purposes of the paper with no thought that it would be used to express LISP programs in practice..

[...] Steve Russell noticed that eval could serve as an interpreter for LISP, promptly hand coded it, and we now had a programming language with an interpreter.”