A very brief history of programming languages

In the beginning….

- The very first programming language was Konrad Zuse's *Plankalkul*, designed for a hypothetical machine that was never built.
- He worked mostly alone, in Bavaria, late 1930s-1940s.
- His notes included 2's complement integers, floating point (with hidden bit), searching/sorting algorithms, graph connectivity & primality testing, and notes toward a chess program
- Because of the war, his work was mostly unknown until the 1970s.

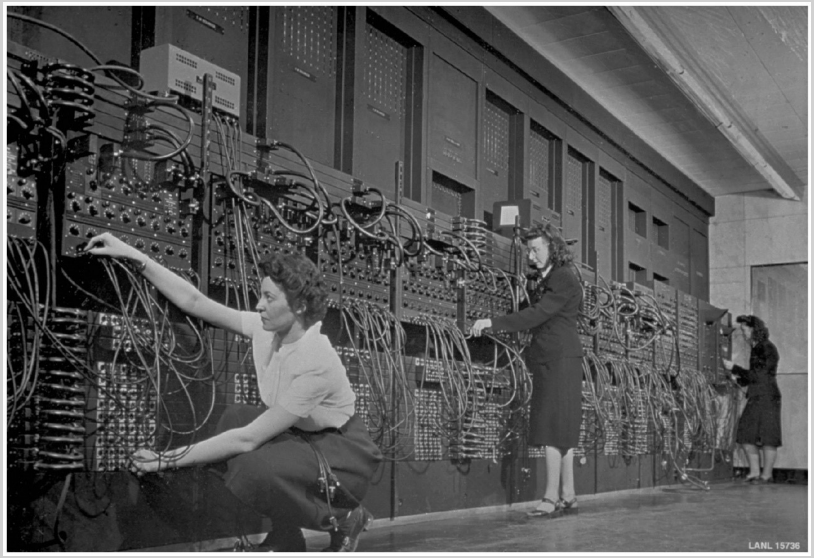
Meanwhile, in the US and UK…

- The Allies built large electromechanical devices to compute artillery tables.
- The Harvard Mark 1 was programmed on paper tape and had a 5 HP motor driving the mechanism.
  - Loops were made by literally looping the tape
  - FP division took over 2 minutes
  - Frequent breakdowns: Relays have moving parts, & wore out. Even if a given relay had a 10-yr expected life, there were 10,000+ of them in the Mark 1.
  - Grace Hopper worked on this project.
- In the UK, electrical engineer Tony Flowers designed & built the Colossus machine to decipher Enigma messages by brute force
  - Built using tubes, so no moving parts in core. Full redundancy for failover.

Further advances

- After the war, the ENIAC, the first fully programmable computer, was built.
- Entirely with tubes, programmed by plugboard
  - This meant changing programs took anywhere from several hours to 3 weeks.
  - Keeping a very expensive machine idle for that long every change of programs drove need for better programming capability
- Thus a need to program more efficiently

Programming the ENIAC



- The first six ENIAC programmers were women
- Every program step specified by making physical connections – the output of this circuit becomes the input over here; the output of that becomes the input over here; etc.

A plugboard

- A program for a quarterly sales report, IBM
- Swappable plugboards reduced the time to change programs to about 10 minutes

Stored Program (Von Neumann) architecture

- In a Von Neumann architecture, the program is loaded into memory & shares memory with the program's data.
- This allowed the program to be stored externally & loaded when needed
- So program development could proceed in parallel—writing one program can happen while another program is running.

Machine language

- At the very lowest level is machine language, which drive the path of data through the circuitry. So, for example, something like:
  - ADD 0010 the contents of
  - REG 4 0100 to the contents of
  - REG 1 0001 and then
  - STO 0011 the contents of
  - REG 4 0100 to memory address 440
  - 000110111000
- So the 3 12-bit ENIAC instructions would be
  `001001000001`
  `001101000000`
  `000110111000`
  (with 4 bits padding at the end of the STO instruction)

Assembler Programming

- The first step was to write a helper program (in machine language) that could read *mnemonics* (such as ADD, STO, BEQ, etc) and translate (assemble) them into the appropriate machine language.
- Then, rewrite the helper program in assembly language
- Then additional features could be added
  - On a branch instruction, it's necessary to specify how many bytes forward or back to jump. If more code is added between the branch and the target, the branch is off and has to be re-figured. This could be delegated to the assembler—the target could be labeled, and the assembly instruction set to jump to the label
  - Pseudo-instructions could be added that translated to named blocks of code in a stored library, breaking the 1:1 link between assembly & machine instructions

Short code

- John Mauchly developed Short code, in which specified commands (including math functions) were coded as numeric values
- No specification or samples survive, but a programming manual does
- Codes were byte-value pairs
- Mauchly claimed Short code programs could be written significantly faster than assembly programs.
- Ran about 50 times slower than machine code.

Speedcoding

- 1954: John Backus at IBM develops Speedcoding for the IBM 701.
- Allowed the computer to function as a 3-address FP calculator
- Pseudoinstructions for arithmetic operations, sine, arctangent, exponent, logarithm
- Conditional and unconditional branching
- Each instruction took 4.2 ms to execute. Speedcoding interpreter left 700 words for main program.
- Automatically incremented address registers, which wouldn’t appear in hardware until 1962.
- Backus claimed programs taking 2 weeks to write in assembly could be done in a few hours in speedcode

AL-0, AL-1, etc

- Grace Hopper was doing similar work, developing Algorithmic Language 0 and 1. Again, no samples survive.
- She also wrote the first working compiler, translating from a higher-level language directly to blocks of machine code, often selected from a code library and compiled (combined) into a working program.
- She encountered resistance because of the belief computers couldn't "do" language, only numbers
  - She had it mansplained to her that it was an interesting theoretical idea but probably couldn't actually be done—she'd had one working for 2 years at the time

FORTRAN (1950s)

- John Backus at IBM heads team developing a new language for the new line of IBM computers under development
- Called FORTRAN for FORmula TRANslation, it focused on numeric computation
  - The IBM 704 had hardware support for floating point—programs are much faster, and interpretation no longer trivial overhead
  - Ambitious goals: that it would run as efficiently as hand-coded assembly programs, and would be so much easier to use that coding errors would be eliminated. FORTRAN programs wouldn't need debugging.
    - So little syntax-checking was provided….

Design constraints

- 1950s computers:
  - Small memory
  - Slow, relatively unreliable
  - Mostly used for scientific computation
  - No efficient way to program them
  - Computer time very expensive compared to programmer time
- Thus, primary goal is speed of the completed object code
  - Doesn’t have to be convenient for programmers
  - If it’s efficient, it has better chance of completing before hardware malfunctions

FORTRAN I (1956)

- Designed for IBM 704
- Most commands map directly to 704 commands or short blocks of assembly code
- i/o formatting
- 6-character variable names
- User-defined subroutines
- Commands for IF, DO
- No data typing: Variable names beginning with I, J, K, L, M, N assumed integers, all others floating point
- Within a year, half of all code written for the 704 was in FORTRAN
- Object code about half as efficient as hand-coded assembly

Later Versions

- FORTRAN II (1958)
  - Mostly bug fixes
  - Separate compilation of subroutines
    - At the time, 300-400 lines source code was about the upper limit, or hardware would probably crash before compilation finished.
    - Precompiled libraries were major step forward
- FORTRAN III developed but never widely distributed
- FORTRAN IV (1966)
  - Explicit type declaration
  - Ability to pass subprograms as parameters
- FORTRAN 77 (1977)
  - Added character strings, logical loop control, optional ELSE for IF

Modern Fortran

- Fortran 90 (1990)
  - Included list of features recommended for removal
  - Abandoned ALL CAPS variable names
  - Dropped fixed-format code requirement
- Fortran 95 (1995)
  - Added Forall to ease parallelization
- Fortran 2003
  - Added object oriented programming, procedure pointers, interoperability with C
- Fortran 2008
  - Added local scopes, co-arrays (parallel execution), DO CONCURRENT

Evaluation

- FORTRAN was an incredible success for its time
- Language design was secondary to getting the compiler right
- Originally only ran on IBM hardware
- Static typing, no dynamic allocation, so sacrificed flexibility for simplicity & efficiency
- Recursion & flexible data structures not possible, but given the typical programs of the time, that wasn't a big issue
- First widely-used high-level language, still in use today
- Most imperative languages trace their ancestry to FORTRAN

Functional programming: LISP

- Interest in AI coming from several sources – linguistics & natural language processing, psychologists modeling information storage, mathematicians interested in automating theorem proving
- Need to process symbolic data in lists, often recursively, rather than crunch numbers in arrays
- John McCarthy of MIT took summer position at IBM Research, working on symbolic differentiation
  - Came away with list of requirements: better control flow, recursion, conditional expressions. FORTRAN had none of those.
  - Also wanted automatic deallocation of elements no longer needed. Didn't like allocation & deallocation statements cluttering up his algorithms
- McCarthy & Marvin Minsky form MIT AI Project

LISP data structures

- LISP has only atoms (identifiers or numeric literals) and lists (ordered collections)
- '(A B C D) is a list with 4 elements (note the quote indicating this is data)
- '(A (BC) D (E (FG))) also has 4 elements: A, (B C), D, (E (F G))
- (A B C D) is a call to function A that has parameters B, C, and D
- Stored internally as linked lists.
  - Each node as 2 pointers: 1 to payload, the other to the next item in the list
  - List is referenced by pointer to its first element

Functional programming

- All computation is application of functions to arguments
- No assignment statements or variables
- Recursion instead of iteration
- Functions affect their environment only by returning values into them; functions do not have *side effects*
- We'll discuss in much more detail later

Evaluation

- Dominated AI for over 25 years
- First garbage collected language (1958)
- Reputation for slowness largely overcome
- Several versions developed independently
- Common Lisp devised as compromise, became “standard” version
  - Supports records, arrays, strings, packages
- Scheme introduced static scoping
- Racket optimized for domain-specific languages
- Other functional languages: ML, Miranda, Haskell, Caml, Ocaml, F#

ALGOL

- Result of years-long effort to develop a single universal language for scientific programming
  - Multiple languages, each around a single hardware, hindered progress
- First language developed internationally. Goals:
  - Syntax as close as possible to standard mathematical notation
  - Programs should be readable with little additional explanation
  - Should be possible to use the language to describe algorithms in printed publications
  - Must be translatable to machine language
- After many, many compromises, ALGOL 58 was released

ALGOL features

- Very similar to FORTRAN
  - Formalized concept of data type
  - Added concept of compound statements
  - Identifiers could be any length
  - Any number of array dimensions allowed
  - Lower bounds of arrays could be programmer-specified
  - Nested selection
- Originally used European style for assignments: value => variable
- But keyboard limitations led to > being replaced with : v =: var
- Then the Americans insisted on turning the whole thing around to do it like FORTRAN: variable := value

Adoption of ALGOL

- Intended as a design document, but the US Air Force's JOVIAL was widely adopted.
- IBM backed it but soon soured; promoting a language is tough, and this was competing with FORTRAN
- 1959 conference to discuss revisions, John Backus introduced a new notation for describing the syntax
  - Backus-Naur form was considered too abstract & esoteric at the time, but became the standard way of describing languages
- ALGOL 60 introduced block scope, pass by value, pass by name, recursion, stack-dynamic arrays

Evaluation

- ALGOL was a great success, influencing almost every language that came after it and quickly becoming THE method for describing algorithms in journals
  - Most imperative languages derived from it more-or-less directly
  - First internationally designed language
  - First machine-independent language
  - First language with syntax described with mathematical rigor
- ALGOL was a dismal failure, never achieving widespread use in the US and never adopted widely in Europe
  - Some features were *too* flexible, hard to understand
  - No I/O defined as part of the language, as it's hardware dependent, but that led to fragmentation
  - BNF seemed strange & complicated
  - Lack of support from IBM probably contributed

Computerizing Business

- ALGOL wasn't widely adopted but influenced a lot of what came later
- COBOL is the opposite: Widely adopted but not much influence on later languages
  - Partly it's because COBOL does business computing very well; not much pressure for something 'better'
  - Business computing driven mostly by small business buying off the shelf software rather than in-house development
- Several companies working on a business language in early 50s, including IBM's COMTRAN, still under development
- Grace Hopper: "Mathematical programs should be written in mathematical notation, data processing programs should be written in English statements."

Design goals

- First design conference sponsored by DoD in 1959
- CBL (common business language) should:
  - Use English as much as possible (a few argued for more mathematical notation)
  - Must be easy to use, even at the expense of being less powerful, to broaden the base of people who could become programmers
    - Managers should be able to understand programs
  - Design must not be overly restricted by implementation details
- Early decision to separate executable statements from data description
- Initial report published 1960

Evaluation

- Several new concepts
  - DEFINE verb: first high-level language construct for macros
  - Hierarchical records
  - Names could be 30 characters *and* have connectors (hyphens)
  - Every variable defined in detail:
    - Format
    - Number of decimal places
    - Location of implied decimal
  - Files defined in detail
  - Lines output to printer defined in detail (ideal for printing accounting reports)
- Division of code & data strongest feature
- Weakest is subprograms: didn't have subprograms with parameters until 1974
- Poor compiler made language expensive to use
- First language mandated by DoD

Timesharing: BASIC

- Widely successful, but gets no respect
- Early versions limited, inelegant
- Popular on early home computers of 1980s
  - Easy for beginners to learn
  - Smaller dialects can be implemented very compactly—Commodore 64 & Apple II had interpreters on 4K ROM chips
  - Waned as computers got more powerful, but resurgence in 1990s with Visual Basic & VBA
- Developed at Dartmouth by 2 mathematics professors who'd written compilers for Fortran & Algol 60
- Science & engineering students generally had little problem with those languages, but that was only about 25% of students; liberal-arts students struggled

Timeline

- 1963: Decision to design a new language for liberal-arts students
- Would use time-sharing terminals, just being installed, to access
  - This was new & cutting edge technology at the time—many universities would not have terminals for another 20 years
- Goals:
  - Must be easy for nonscience students to learn & use
  - Must be pleasant & friendly
  - Must provide a fast turnaround for homework
  - Must allow free & private access
  - Must consider user time more important than computer time (a new, and revolutionary, idea at the time)
- 1964, May 1: First program using timeshared BASIC typed in & run
  - By fall, there were 20 terminals on campus
- Original version non-interactive; couldn't get input from user
- 14 statement types, 1 variable type (FP)
- Design came mostly from FORTRAN
- Developed in several ways, no effort to standardize it

Evaluation

- BASIC programs tend to be poorly structured
  - Dijkstra's letter to CACM ("Goto statement considered harmful") argued that those learning BASIC as their first language were "damaged beyond repair"
- Readability & reliability are both weaknesses
- Early versions weren't meant for serious programs of any significant size; later versions did better
- First language written/used on timesharing terminals
- First language to prioritize user time over computer time
- Visual Basic led to a resurgence
  - Easy way to write GUIs
  - Microsoft sponsorship—VBA became interface language to MS Office
  - VB.NET quickly supplanted the original Visual Basic
- Sometimes reported that it stood for *Beginner's All-purpose Symbolic Instruction Code*. This is a back-formation; the original authors said they called it BASIC "because it's a basic programming language."

State of computing from IBM's view, 1960s

- Scientific computing was mostly FORTRAN with a little assembly
  - Floating-point, arrays – ran on IBM 7090 or smaller IBM 1620
- Business applications mostly assembly; COBOL just starting
  - Fixed-decimal, character strings, elaborate I/O
  - Ran mostly on IBM 7080 or smaller IBM 1401
- Language tied to hardware, and vice-versa
- But things were changing:
  - Scientists using larger data sets, needed better I/O
  - Business beginning to use regression & statistics, needed FP & arrays
- Few installations would be interested (or able to afford) 2 installations, with separate technical staff, supporting different languages/hardware
- Thus needed a single universal language, capable of all of these
- This was the origin of the System/360—a range of computers from small minicomputers to large mainframes, all capable of running the same software

Something for Everyone: PL/I

- Needed a language to support scientific *and* business applications
- Just for good measure, threw in systems-level programming and list processing.
- Thus the new language could replace FORTRAN, COBOL, and LISP.
- Initially called FORTRAN IV, but compatibility dropped as a design goal quickly

PL/I's contributions and firsts

- Took best parts of
  - ALGOL 60: recursion, block structure
  - FORTRAN: Separate compilation, communication through global data
  - COBOL 60: data structures, good I/O & report generation
- First language to have:
  - Concurrently executing subprograms
  - Exception handling – 23 types of runtime errors detected
  - Recursive subprograms allowed, but recursion could be disabled, making nonrecursive programs link more efficiently
  - Pointers as a data type
  - Array slicing – 3rd row of matrix could be addressed as if it were a one-dimensional array

So what went wrong?

- Concurrency was poorly implemented, hard to use
- Pointers were poorly implemented, hard to use
- Exception handling was poorly implemented, hard to use
- At the time, no one had much experience designing programming languages
  - Design was based on the idea that if a feature is useful, it should be included…
  - …but not enough thought to how a programmer could understand & make use of it
  - Dijkstra, 1972 Turing Award Lecture: “I absolutely fail to see how we can keep our growing programs firmly within our intellectual grip when by its sheer baroqueness the programming language—our basic tool, mind you!—already escapes our intellectual control.”
  - Programmers tend to learn favorite techniques & commands; programmers working in same application space were writing mutually incomprehensible code

Evaluation

- Backed by IBM, it did achieve a fair amount of use in business & scientific applications
- Widely used as a teaching language
- But generally regarded today as a beautiful, glorious, ambitious failure

APL

- Developed 1960 by Kenneth Iverson at IBM
- Originally intended as a language to describe computer architecture, not a programming language
- Described in the book A Programming Language
- Specialized operators for matrix manipulations.
  - Most commands are 1 character.
  - Language has many specialized symbols. (matrix transpose is a 1-character command)
  - Initially implemented on IBM printing terminals, which could have special print balls with extra characters
- Very expressive; most programs only a few lines
- Very orthogonal: operators can be put in almost any order and still be a valid program
- Very LOW readability, maintainability; generally considered a write-once language, best used for throwaway programs
  - “I realized there was a problem when it took me all morning to work out what a 4-line program was doing.” –Kenneth Iverson, inventor of APL

Sample APL Code

- SD←((+/((X - AV←(T←+/X)÷⍴X)*2))÷⍴X)*0.5
- (~R∈R∘.×R)/R←1↓⍳R
- life←{↑1 ⍵∨.∧3 4=+/,¯1 0 1∘.⊖¯1 0 1∘.⌽⊂⍵}
- What the heck are those?
  - Standard deviation of array (T = total, AV = average value)
  - Prime numbers from 1 to R
  - Conway's Game of Life
    - All samples from Wikipedia

SNOBOL

- Designed in 1960s at Bell Labs
- Designed for text processing, has powerful operations for pattern matching in text
  - Early application was text editors
- Still in use today, most in text-processing applications
- AWK partially based on SNOBOL

Data Abstraction: SIMULA 67

- Designed 1964 at Norwegian Computing Center in Oslo
- Improvement efforts culminated in SIMULA 67
- Extension of ALGOL, mostly focusing on subprograms
  - Simulations often need co-routines: subprograms that can return a value, then on the next call resume execution where they left off
- First idea of a class

Orthogonal design: ALGOL 68

- Incorporated several new ideas:
  - User-defined data types built up from primitive types
  - Implicit heap-dynamic arrays, called flex arrays
    - Specification didn't need to include subscript bounds
    - Arrays expanded as new items added
- Heavy emphasis on orthogonality—little distinction between statements & expressions, making it very writeable
- BUT:
  - Described in van Wijngaarden grammar, far more complex than BNF
  - Designers invented new terms: keywords were 'indicants,' substring extraction was 'trimming,' subprogram execution was "coercion of deproceduring," which could be 'meek,' 'firm,' or something else
  - PL/I had the backing of IBM; ALGOL 68 had no one

Other ALGOL variants

- ALGOL-W invented at Stanford, used mostly for teaching
  - Pass by value/result
  - Introduction of case statement for multiway selection
- ALGOL-W formed the basis of 1971's Pascal, designed as a teaching language
  - User defined data types; case statement; records
  - Because it was designed for teaching, it was limited:
    - Can't write a subprogram that takes a variable-length array as a parameter
    - No separate compilation
    - Random-access file i/o mostly undefined
  - Vendors did their own version of these; Borland's Turbo Pascal was most popular

FORTH

- Developed by Charles Moore, first released 1970
- Stack-based, uses postfix notation
- Core language is very small; a few ‘words’ implemented in machine code
- Words can be combined into other words, extending the language toward the desired application – a meta-application language used to create problem-oriented languages
- No syntax as such. The interpreter parses input looking for a whitespace-delimited label. Once identified, it looks in a table for that label & executes the corresponding code
- Fit into very small systems such as 8-bit systems, embedded hardware
- Originally produced threaded code, current versions produce optimized machine code
- First native language environment for Intel 8086 chip & Macintosh 128K
- Electronic Arts published several games in Forth
- Still used in astronomical applications, embedded systems

Systems Programming: C

- UNIX operating system developed at Bell Labs in 1960s
- Originally written in a mix of assembly, BCPL, and a successor language called B
  - Both BCPL and B are untyped; data are machine words, leading to various complications & insecurities
- So a new typed language was developed. Originally called NB (new B), later changed to C.
- For the first 15 years or so, the only ‘standard’ was Kernighan & Ritchie’s book. ANSI produced official description in 1989, including some features already included by other implementers.
- Early versions didn’t type-check parameters; better flexibility, less security
- Basic compiler was quite good, producing fast compact object code; compiler was included with OS
  - This was partly because C was designed around the PDP-11, a standard mainframe of the day. Many C commands map directly to PDP-11 machine code or short blocks of assembly

Programming in Logic: Prolog

- Logic programming uses formal reasoning, usually based on resolution proof techniques or predicate calculus.
- They are nonprocedural; they do not specify how a solution is to be found
- Prolog has:
  - Facts
    - Mother(joanne, jake)
  - Rules
    - Grandparent(X Y) -: Parent(X,Y), Parent(Y,Z)
  - Goals
    - Grandparent(bob, jake)?
  - If the goal can be derived from the knowledge base, Prolog responds true, otherwise displays false
- Logic programs are very useful for some specific types of problems, less so for general-purpose computing. Also, most logic languages are relatively inefficient.

Programming for Reliability: Ada

- Most extensive (and expensive) design effort ever
- By 1974, over half of DoD's computer applications were embedded systems
- These included more than 450 programming languages, none of them standardized; every contractor could define a new and different language for every contract
  - So application software was rarely re-used, and development tools were seldom created
- DoD's standardization on COBOL had been successful
- The Army, Navy, & Air Force independently proposed developing a single high-level language for embedded systems

Design Process

- Jan 1975: High-Order Language Working Group formed
  - Representatives from all military services, liaison with UK, France, West Germany
  - Identify requirements for new DoD high-level language
  - Evaluate existing languages to see if a viable candidate exists
  - Recommend adoption or implementation of a minimal set of programming languages meeting all requirements
- Multiple rounds of design, roughly annually, each slightly ‘firmer’ than the one before (fewer large changes, more fine-tuning)
  - Progressed from April ’75 ‘Strawman’ through Woodenman, Tinman, Ironman, Steelman
  - 4 finalists chosen, all based on Pascal
  - 1979 name of Ada proposed & adopted. Design & rationale published in ACM SIGPLAN Notices
  - Public test/evaluation conference Oct 1979, 100+ organizations sent representatives
  - Stoneman released Feb 1980. MIL-STD 1815 (chosen to commemorate year of Ada Lovelace’s birth) adopted July 1980. Revised version published 1982, ANSI standard released 1983, design then frozen for 5 years

Features

- Packages allow encapsulating data objects, specifications for data types, and procedures
- Elaborate exception handling
- Introduction of generic (template) code, e.g. a sorting method that can sort any data with a < operator.
  - Still has to be instantiated for a specific type, but obviously encourages code re-use
- Concurrent execution of subprograms

Evaluation

- Open design process led to wide participation
- Embodies most ideas about programming languages from that era
- Some saw it as too large or complex. Hoare argued it shouldn't be used in situations where reliability was important—but that's what it was designed for.
- Compiler development was difficult. First working compilers didn't appear until 1985.
- Ada 95 not as widely used, because C++ was becoming popular for OO programming
- Ada 2005 added interfaces, better scheduling, synchronization control
- Motto of the Ada Foundation: In Strong Typing We Trust

Erlang

- Designed to build fault-tolerant, distributed, scalable systems
- Developed for telecom applications, particularly phone infrastructure
  - Lightweight pre-emptive concurrency
  - Processes isolated from each other so if one crashes it doesn’t bring down everything
  - Supervision trees provide ways to specify how to recover from errors
  - Hot-swapping code—code can be replaced in the ‘live’ system without interruption
  - Functional programming features built in: pattern matching, higher-order functions, immutable data
  - Built-in support for distributed computing & scalability

Smalltalk

- Alan Kay's dissertation at the University of Utah had a remarkable insight for its time (1969): Powerful desktop computers would be available someday, far more powerful than the mainframes of that era.
  - These would be used by non-programmers, and thus would need a powerful human interface capability.
  - Therefore it would need to be very interactive and use sophisticated graphics in the UI.
- His research led to Dynabook, the first system to use a desktop metaphor, containing pages, some partially covered (out of focus), pages selectable by touchscreen or keyboard
- Kay went to Xerox PARC, which developed a language to support this (Smalltalk 72), later improved to Smalltalk 80.
- First fully OO language—everything is an object, control flow via message passing
- First language with a GUI as a fundamental part of the language
- It was tied to a particular hardware, and so was never widely adopted itself, but had a massive influence on the OO languages that followed

Effects of Smalltalk (and a well-timed magazine article)

- Byte Magazine ran a cover story on Smalltalk
- Bjarne Stroustrup at Bell Labs had used Smalltalk on another project, and started working on extensions to C in 1980
  - Function parameter type checking (finally!)
  - Classes, public/private methods, constructor/destructor methods, friend classes
  - 1981: Inline functions, default parameters, overloaded assignment operator
  - 1983: C with Classes published
- Goal was that C with Classes could be used anyplace C could be used, so no features were removed from C, even ones known to be unsafe
- Moderately successful, but community wasn't growing, and Stroustrup was doing all the maintenance work
  - 1984: Virtual methods, method & operator overloading, reference types. Released as C++.

Evaluation

- The additional features boosted popularity; C++ was first widely adopted OO language
  - Several good, cheap compilers available
  - Mostly backwards-compatible with C
  - Only OO language available for large projects when OO programming became popular
- Drawbacks
  - Very large, complex language
  - Inherited most of the insecurities of C
- 1989: Multiple inheritance, abstract classes added
- 1990: templates, exception handling
- Standardized 1998
- 2002: Microsoft releases Managed C++ for .NET platform

Objective-C

- Brian Cox, trying to manage large C projects that were ‘like soup,’ read the same article on Smalltalk.
- Designed Objective-C, consisting of C plus classes & message-passing of Smalltalk
- Steve Jobs left Apple and founded NeXT, licensed Objective-C to write the system software
- Apple bought NeXT, used Objective-C to write MAC OS software
- Early iPhone development all in Objective C
- Objective C is a strict superset of C, so retains all of its vulnerabilities

Other C-Based Languages

- Delphi: Added objects to Pascal
  - Pascal is more elegant and safer than C; Delphi is more elegant and safer than C++
  - Pascal is less powerful than C; Delphi is less powerful than C++
- Go
  - Designed 2007-2009, mostly in response to slow compilation of C++
  - Goal is fast compilation. Does not support inheritance or generics
  - Includes goto statement, pointers, associative arrays (hash tables), support for concurrency

Java

- Sun Microsystems needed a language for programming embedded consumer electronics—toasters, microwave ovens, interactive TV systems, etc.
- Reliability was primary goal—if a flaw is discovered after a product ships, recalling 100K microwave ovens would be very expensive if not impossible
- C was compact but didn't provide OO code. C++ too large & complex. Neither is particularly safe.
- BUT--None of the products it was designed for were ever marketed. Starting in 1993, with the advent of graphical browsers, Java quickly became a popular web programming language, through applets

Java

- Based on C++ but designed to be smaller, simpler, more reliable
  - No pointers, but reference types provide some of the same functionality
  - Garbage collection removes need to manage pointers
  - Arithmetic expressions can't be control expressions: `if (a*b)` is true in C++ if the expression is nonzero; in Java it's a syntax error
  - All Java programs are methods defined in classes, so only supports OO programming
  - Does not support multiple inheritance
  - Simple concurrency through `synchronized` modifier
  - Type coercion only allowed going from 'smaller' to 'larger' type
- Removing half the assignment coercions improves reliability
- Index range checking adds safety
- Initial JVM about 10 times slower than compiled C; modern versions use JIT compiler to translate to machine code

Scripting Languages

- Early systems allowed putting a **script**, a list of commands, into a file to be interpreted.
- Originally these were calls to system programs, executed in a straight-line fashion
- The first full scripting language, sh, added variables, control flow, functions, resulting in a complete programming language
- David Korn at Bell Labs developed the Korn shell. The Bourne shell and its open-source Bourne Again Shell (bash) are also popular
- Aho, Kernighan, and Weinberger at Bell Labs developed awk, originally as a report generator but becoming more general purpose

Perl

- Originally a combination of sh and awk
- Known as a scripting language but more similar to an imperative language
- Variables are statically typed, declared implicitly
- Arrays are dynamic length and can have missing elements. Hashes are built-in data type
- Scalars can be strings or numbers, and are converted back & forth automatically
- If a string is used in a numeric context & can't be converted, 0 is used, with no warning or error
- No set subscript range, therefore no range checking.
- Very writeable— "there's more than one way to do it" – but this hurts maintainability
- In 2019, Perl 6 renamed to Raku, following several modifications that helped performance but broke backwards compatibility

Javascript

- Explosion of the world wide web showed need for computation associated with HTML documents
- Original name was Mocha, later renamed to LiveScript due to trademark issues. Changed to JavaScript when Netscape entered joint venture with Sun
- Primary use is form validation & creating dynamic HTML documents
  - Dynamically typed
  - Strings & arrays have dynamic length, therefore no range checking
  - No inheritance or late binding of method calls

PHP

- Developed at Apache in 1994, originally to track visitors to personal website
- 1995: released Personal Home Page tools. Later renamed to PHP Hypertext Processor
- Open source, present on most web servers
- Server-side embedded scripting
- Usually produces HTML document as output, which replaces the PHP code in the document; the browser never sees the PHP
- Mostly for building programs that need web access to databases

Python

- 1990s, Guido von Rossum, currently maintained by Python Software Foundation
- Dynamically typed
- Includes lists, tuples, hashes, list comprehensions
- Mostly object oriented
- Pattern matching capabilities match Perl
- Exception handling, garbage collection
- Supports functional programming, more or less
- Widely adopted by non-experts

Ruby

- Designed by Yukihiro Matsumoto, released 1996, designed to overcome what he saw as shortcomings of Python
- Pure OO language. Operators are just method calls, and so can be redefined; all classes can be subclassed
- Classes and objects are dynamic—methods can be dynamically assigned to either
- Scope defined by variable name: begins with letter = local, begins with @ = instance, begins with $ = global
- First language designed in Japan to achieve widespread use in US, mostly through Ruby on Rails web framework

Lua

- Developed at Pontifical University of Rio de Janeiro
- Supports procedural & functional programming; extensibility is primary design goal
  - Only 21 reserved words
  - Basic data structure is the table
  - Functions are first-class values
  - Popular in gaming industry

Flagship .NET: C#

- Based on C++ but includes ideas from Delphi & VB
- Designed for component-based software development
  - Components can be written in any .NET language
  - C#, VB.NET, Managed C++, F#, Jscript.NET all use Common Type System, compiled into Common Intermediate Language, which in turn is handed over to a JIT compiler
- Doesn’t support multiple inheritance but does support pointers, structs, enumerated types, operator overloading, and goto

Julia

- Introduced 2012, designed for high-speed technical computing & data science
  - JIT compiler, type inference, multiple dispatch lead to high performance
  - Dynamically typed
  - Syntax based on Python, designed for interoperability with other languages

Swift

- Introduced by Apple as a replacement for Objective C
- Cleaner syntax than Objective-C, adds automatic memory management, type inference
- Supports procedural, OO, functional, protocol-based computing

Rust

- Systems programming language developed by Mozilla.
- Designed to provide safe concurrency and memory safety without sacrificing performance
- Some functional features & data types built in
- Gaining popularity in several areas—systems programming, games, embedded programming
- First language other than C to be allowed into the Linux kernel

Wow, that’s a lot…

- And that’s certainly not every language there is!
- Focus on:
  - Approximate dates
  - Ancestry
  - Purposes, goals, main features
  - How did it do?
  - What followed it?