History of Programming Languages

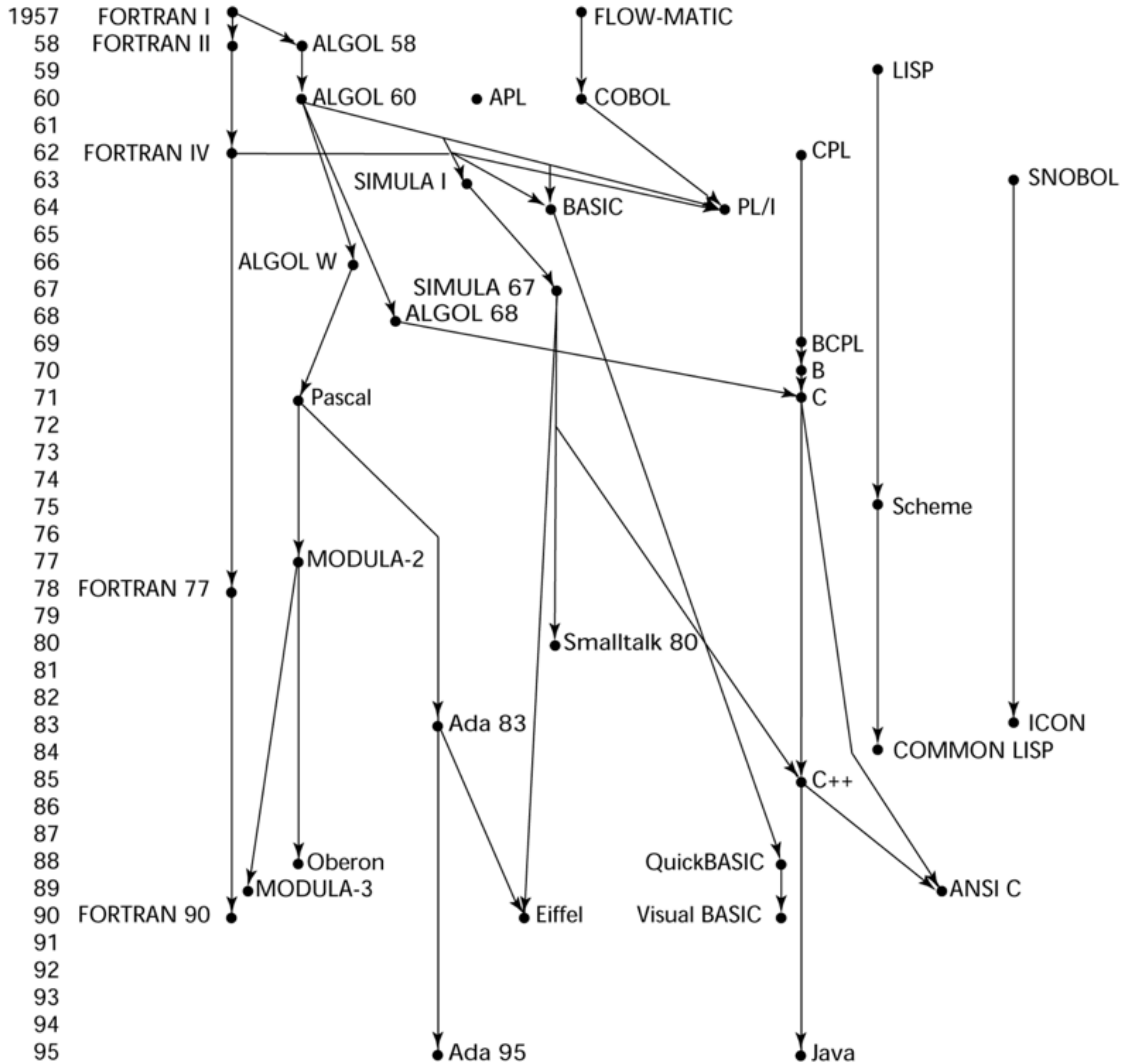
http://fm.zju.edu.cn

Programming Design Methodology

- 1950s and early 1960s: simple applications; worry about machine efficiency (FORTRAN)
- Late 1960s: people efficiency important; readability, better control structures (ALGOL)
  - Structured programming, free format lexical
  - Top-down design and step-wise refinement
- Late 1970s: process-oriented to data-oriented
  - Data abstraction
- Middle 1980s: object-oriented programming
  - Data abstraction + inheritance + dynamic binding

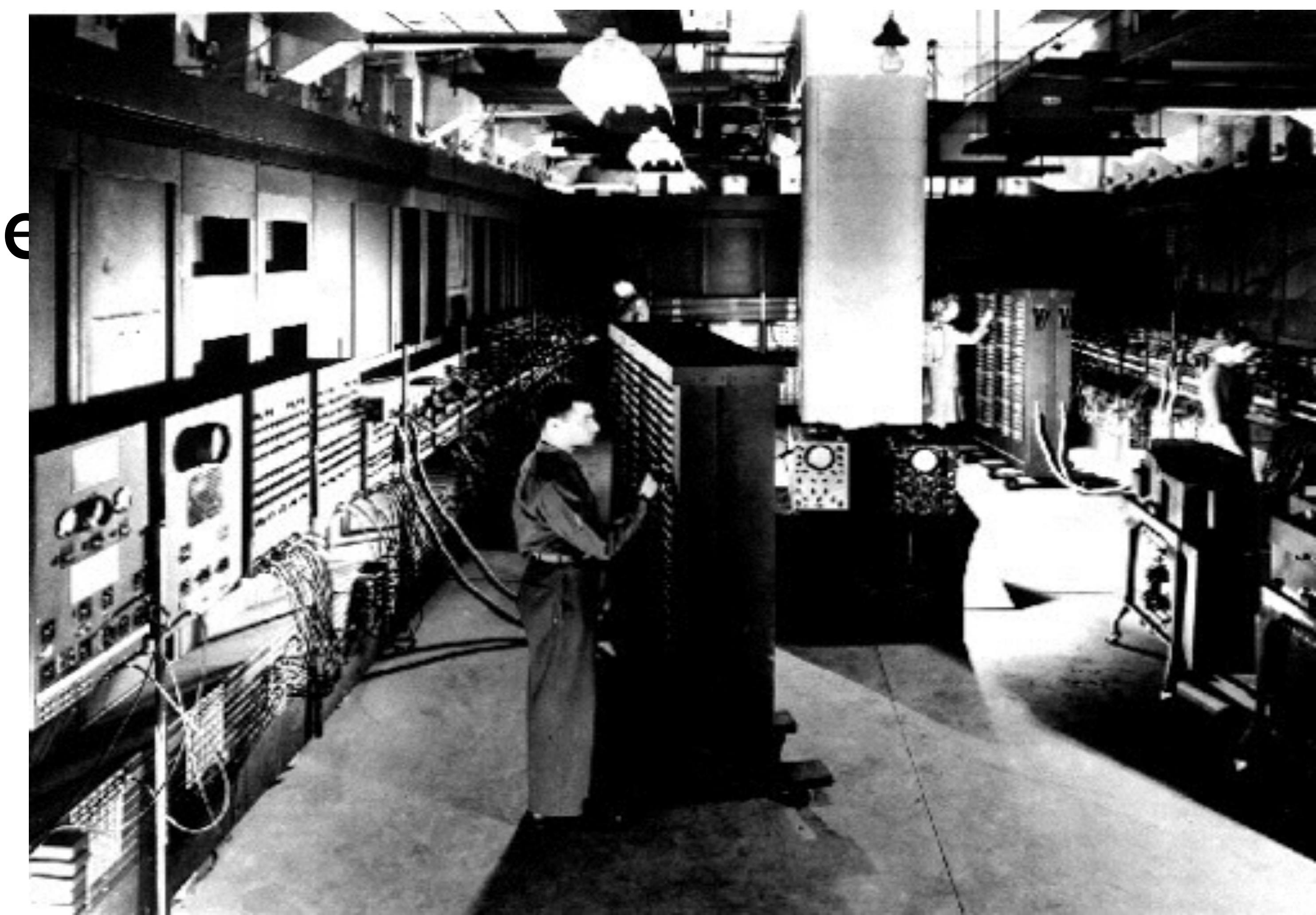
Genealogy o Common Languages

(Fig. 2.1)



The Dawn of Modern Computers

- Early computers (40's and early 50's) are programmed using machine code directly:
  - Limited hardware; no FP, indexing, system software
  - Computers more expe
    programmers/users
  - Poor readability, modifiability, expressiveness



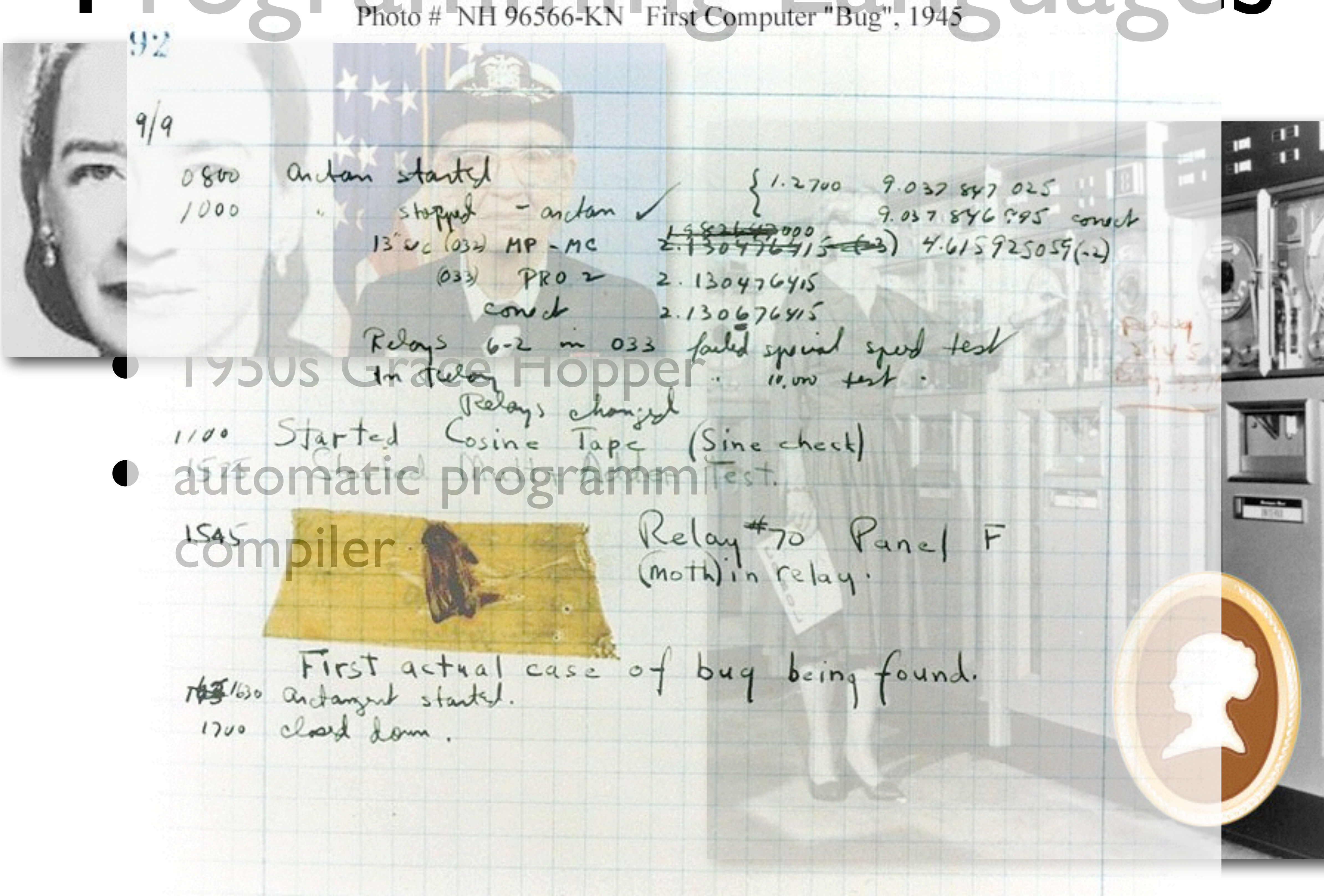
Early Programming

- Programmers have to enter machine code to program the computer
  - Floating point: coders had to keep track of the exponent manually
  - Relative addressing: codes had to be adjusted by hand for the absolute addresses
  - Array subscripting needed
  - Something easier to remember than octal opcodes
- Early aids:
  - Assembly languages and assemblers: English-like phrases 1-to-1 representation of machine instructions
- Saving programmer time became important …

1948 Cambridge

- 111000000000100110010
- A 25 S
- 1948 David Wheeler: translate strings into bin code
- 1950: *The Preparation of Programs for an Electronic Digital Computer*

Programming Languages



- 1950s Grace Hopper
- automatic programmi
  compiler

Fortran

- First popular high-level programming language
  - Computers were small and unreliable
    → machine efficiency was most important
  - Applications were scientific
    → need good array handling and counting loops
- "The IBM Mathematical FORmula TRANslating System: FORTRAN", 1954: (John Backus at IBM)
  - To generate code comparable with hand-written code using simple, primitive compilers
  - Closely tied to the IBM 704 architecture, which had index registers and floating point hardware

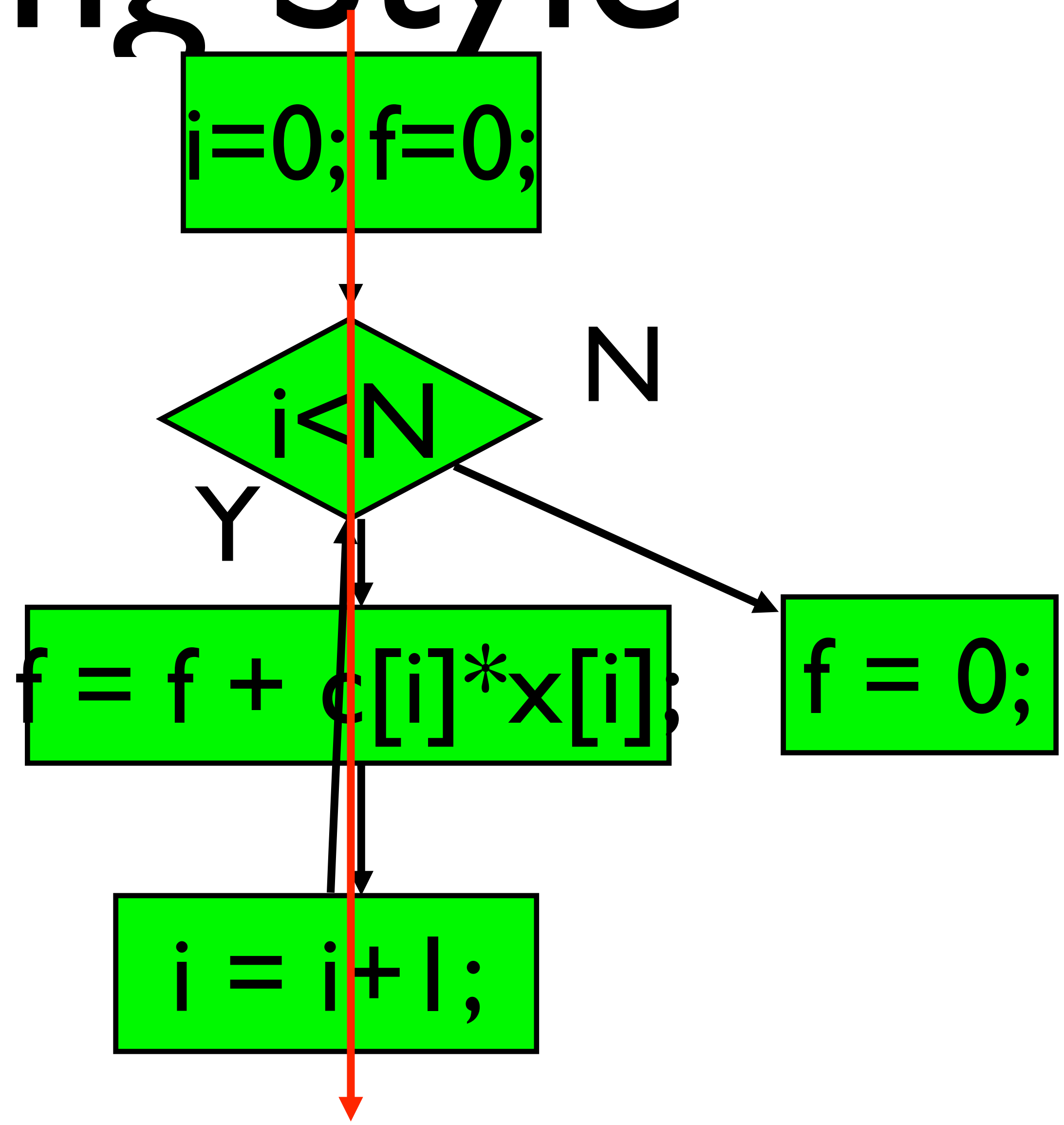
Fortran

- Fortran I (1957)
  - Names could have up to six characters, formatted I/O, user-defined subprograms, no data typing
  - No separate compilation (compiling "large" programs – a few hundred lines – approached 704 MTTF)
  - Highly optimize code with static types and storage
- Later versions evolved with more features and platform independence
  - Almost all designers learned from Fortran and Fortran team pioneered things such as scanning, parsing, register allocation, code generation, optimization

FORTRAN and von Neumann Arch.

- FORTRAN, and all other imperative languages, which dominate programming, mimic von Neumann architecture
  - Variables ←→ memory cells
  - Assignment statements ←→ data piping between memory and CPU
  - Operations and expressions ←→ CPU executions
  - Explicit control of execution flows
  - Efficient mapping between language and HW
→ efficient execution performance, but limited by von Neumann bottleneck

FORTRAN Programming Style



- Global view, top down
- Program starts from first executable statement and follow a sequential flow with go-to
  - Conceptually, a large main() including everything but without main() declaration, though FORTRAN has functions
  - Match a flow chart with traces

Problems: developing large programs, making errors, being inflexible, managing storage by programmers, …

Functional Programming: LISP

- AI research needed a language to
  - Process data in lists (rather than arrays)
  - Symbolic computation (rather than numeric)
- John McCarthy of MIT developed LISP (LISt Processing language) in 1958
- A LISP program is a list:
  ```
  (+ a (* b c))
  ```
  - List form both for input and for function
  - Only two data types: atoms and lists

LISP

- Pioneered functional programming
  - Computations by applying functions to parameters
  - No concept of variables (storage) or assignment
    - Single-valued variables: no assignment, not storage
  - Control via recursion and conditional expressions
    - Branches → conditional expressions
    - Iterations → recursion
  - Dynamically allocated linked lists

First Step Towards Sophistication

- Environment (1957-1958):
  - FORTRAN had (barely) arrived for IBM 70x
  - Many other languages, but all for specific machines → no universal lang. for communicating algorithms
  - Programmer productivity became important
- ALGOL: universal, international, machine-independent (imperative) language for expressing scientific algorithms
  - Eventually, 3 major designs: ALGOL 58, 60, and 68
  - Developed by increasingly large international committees

Issues to Address (I)

- Early languages used label-oriented control:

```
GO TO 27
30 IF (A-B) 5,6,7
```

- ALGOL supports sufficient phrase-level control, such as **`if`**, **`while, switch`**, **`for`**, **`until`**
  →structured programming

- Programming style:
  - Programs consist of blocks of code: blocks → functions → files → directories
  - Bottom-up development possible
  - Easy to develop, read, maintain; make fewer errors

Issues to Address (II)

- ALGOL designs avoided special cases:
  - Free-format lexical structure
  - No arbitrary limits:
    - Any number of characters in a name
    - Any number of dimensions for an array
  - Orthogonality: every meaningful combination of primitive concepts is legal—no special forbidden combinations to remember
    - Each combination not permitted is a special case that must be remembered by the programmer

Example of Orthogonality

	Integers	Arrays	Procedures
Passing as a parameter			
Storing in a variable			
Storing in an array			
Returning from a procedure			

- By ALGOL 68, all combinations above are legal
- Modern languages seldom take this principle as far as ALGOL → expressiveness vs efficiency

Influences

- Virtually all languages after 1958 used ideas pioneered by the ALGOL designs:
  - Free-format lexical structure
  - No limit to length of names and array dimension
  - BNF definition of syntax
  - Concept of type
  - Block structure (local scope)
  - Compound stmt (**begin end**), nested if-then-else
  - Stack-dynamic arrays
  - Call by value (and call by name)
  - Recursive subroutines and conditional expressions

Beginning of Timesharing: BASIC

- BASIC (Beginner's All-purpose Symbolic Instruction Code)
  - Kemeny & Kurtz at Dartmouth, 1963
- Design goals:
  - Easy to learn and use for non-science students
  - Must be "pleasant and friendly"
  - Fast turnaround for homework
  - Free and private access
  - User time is more important than computer time
- First widely used language with time sharing
  - Simultaneous individual access through terminals

keyword: interactive

Everything for Everybody: PL/I

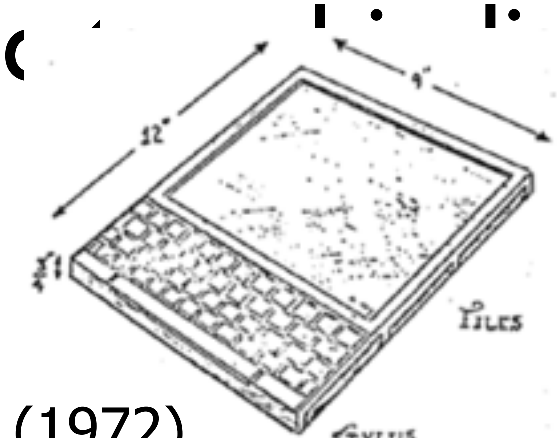
- IBM at 1963-1964:
  - Scientific computing: IBM 1620 and 7090, FORTRAN
  - Business computing: IBM 1401 and 7080, COBOL
  - Scientific users began to need elaborate I/O, like in COBOL; business users began to need FP and arrays
- The obvious solution
  - New computer to do both → IBM System/360
  - Design a new language to do both → PL/I
- Results:
  - Unit-level concurrency, exception handling, pointer
  - But, too many and too complex

Beginning of Data Abstraction

- SIMULA
  - Designed primarily for system simulation in University of Oslo, Norway, by Nygaard and Dahl
- Starting 1961: SIMULA I, SIMULA 67
- Primary contributions
  - Co-routines: a kind of subprogram
  - Implemented in a structure called a class, which include both local data and functionality and are the basis for data abstraction

Object-Oriented Programming

- Smalltalk: Alan Kay, Xerox PARC, 1972
- First full implementation of an object-oriented language
  - Everything is an object: variables, constants, activation records, classes, etc.
  - All computation is performed by objects sending and receiving messages
  - Data abstraction, inheritance, dynami[illegible]g
- Also pioneered graphical user interface design



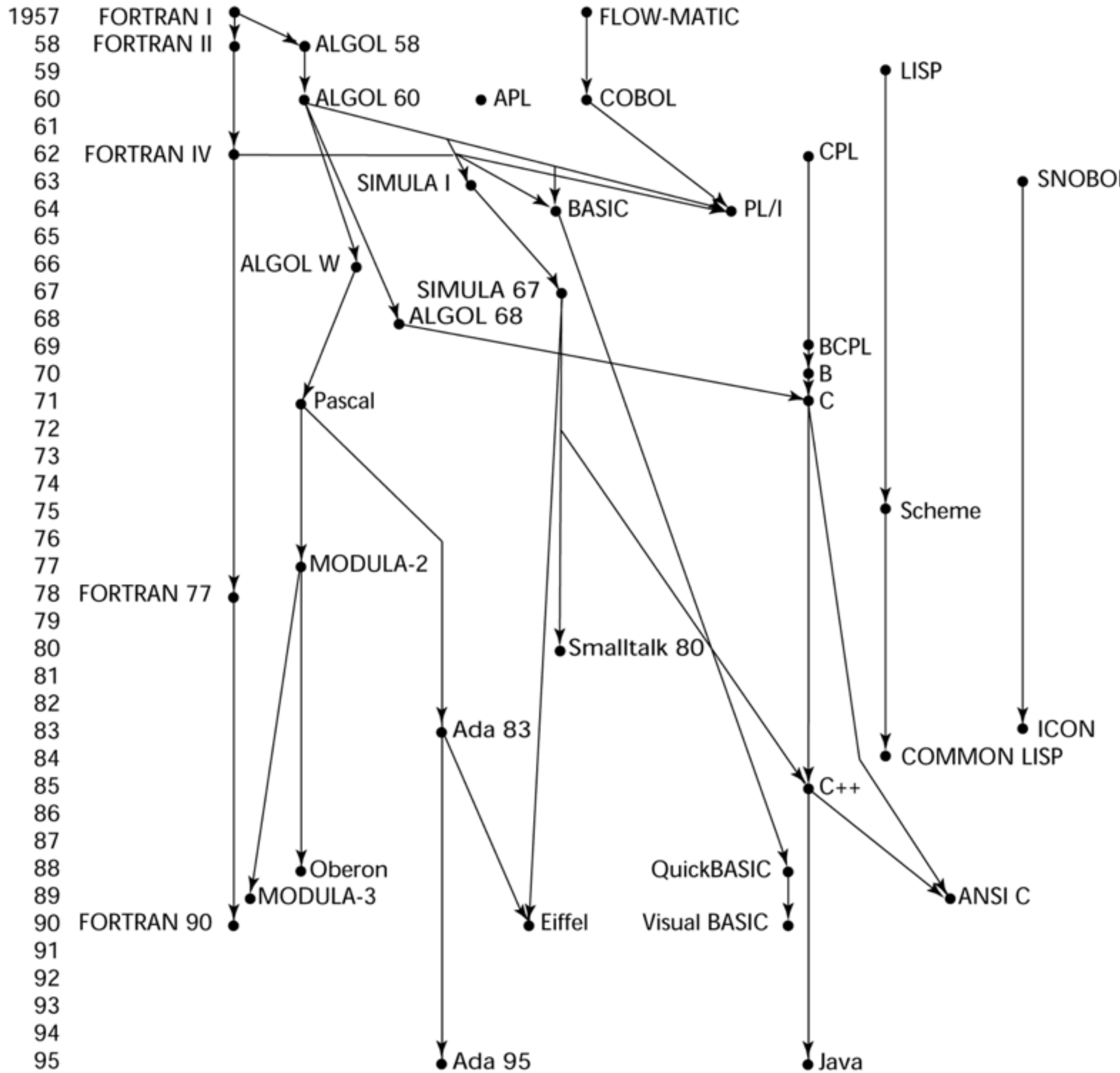
Dynabook (1972)

Programming Based on Logic: Prolog

- Developed by Comerauer and Roussel (University of Aix-Marseille) in 1972, with help from Kowalski (University of Edinburgh)
- Based on formal logic
- Non-procedural
  - Only supply relevant facts (predicate calculus) and inference rules (resolutions)
  - System then infer the truth of given queries/goals
- Highly inefficient, small application areas (database, AI)

Genealogy o Common Languages

(Fig. 2.1)



Summary: Application Domain

- Application domains have distinctive (and conflicting) needs and affect prog. lang.
  - Scientific applications: high performance with a large number of floating point computations, e.g., Fortran
  - Business applications: report generation that use decimal numbers and characters, e.g., COBOL
  - Artificial intelligence: symbols rather than numbers manipulated, e.g., LISP
  - Systems programming: low-level access and efficiency for SW interface to devices, e.g., C
  - Web software: diff. kinds of lang. markup (XHTML), scripting (PHP), general-purpose (Java)

Summary: Programming Methodology in Perspective

- 1950s and early 1960s: simple applications; worry about machine efficiency (FORTRAN)
- Late 1960s: people efficiency important; readability, better control structures (ALGOL)
  - Structured programming
  - Top-down design and step-wise refinement
- Late 1970s: process-oriented to data-oriented
  - Data abstraction
- Middle 1980s: object-oriented programming
  - Data abstraction + inheritance + dynamic binding

Theory of PL: Turing Equivalence

- Languages have different strengths, but fundamentally they all have the same power

  {problems solvable in Java}
  = {problems solvable in Fortran}
  = …

- And all have the same power as various mathematical models of computation

  = {problems solvable by Turing machine}
  = {problems solvable by lambda calculus}
  = …

- Church-Turing thesis: this is what "computability" means

What Make a Good PL?

Language evaluation criteria:

- **Readability**: the ease with which programs can be read and understood
- **Writability**: the ease with which a language can be used to create programs
- **Reliability**: a program performs to its specifications under all conditions
- **Cost**

Features Related to Readability

- Overall simplicity: lang. is more readable if
  - Fewer features and basic constructs
    - Readability problems occur whenever program's author uses a subset different from that familiar to reader
  - Fewer feature multiplicity (i.e., doing the same operation with different ways)
  - Minimal operator overloading
- Orthogonality
  - A relatively small set of primitive constructs can be combined in a relatively small number of ways
  - Every combination is legal, independent of context
    ➔ Few exceptions, irregularities

Features Related to Readability

- Control statements
  - Sufficient control statements for structured prog.
    → can read program from top to bottom w/o jump
- Data types and structures
  - Adequate facilities for defining data type & structure
- Syntax considerations
  - Identifier forms: flexible composition
  - Special words and methods of forming compound statements
  - Form and meaning: self-descriptive constructs, meaningful keywords

Writability

- Simplicity and orthogonality
  - But, too orthogonal may cause errors undetected
- Support for abstraction
  - Ability to define and use complex structures or operations in ways that allow details to be ignored
  - Abstraction in process (e.g. subprogram), data
- Expressivity
  - A set of relatively convenient ways of specifying operations
  - Example: the inclusion of **for** statement in many modern languages

Reliability

- Type checking
  - Testing for type errors, e.g. subprogram parameters
- Exception handling
  - Intercept run-time errors & take corrective measures
- Aliasing
  - Presence of two or more distinct referencing methods for the same memory location
- Readability and writability
  - A language that does not support "natural" ways of expressing an algorithm will necessarily use "unnatural" approaches, and hence reduced reliability

Cost

- Training programmers to use language
- Writing programs (closeness to particular applications)
- Compiling programs
- Executing programs: run-time type checking
- Language implementation system: availability of free compilers
- Reliability: poor reliability leads to high costs
- Maintaining programs

Others

- Portability
  - The ease with which programs can be moved from one implementation to another
- Generality
  - The applicability to a wide range of applications
- Well-definedness
  - The completeness and precision of the language's official definition
- Power efficiency?

Language Design Trade-Offs

- Reliability vs. cost of execution
  - e.g., Java demands all references to array elements be checked for proper indexing but that leads to increased execution costs
- Readability vs. writability
  - e.g., APL provides many powerful operators (and a large number of new symbols), allowing complex computations to be written in a compact program but at the cost of poor readability
- Writability (flexibility) vs. reliability
  - e.g., C++ pointers are powerful and very flexible but not reliably used

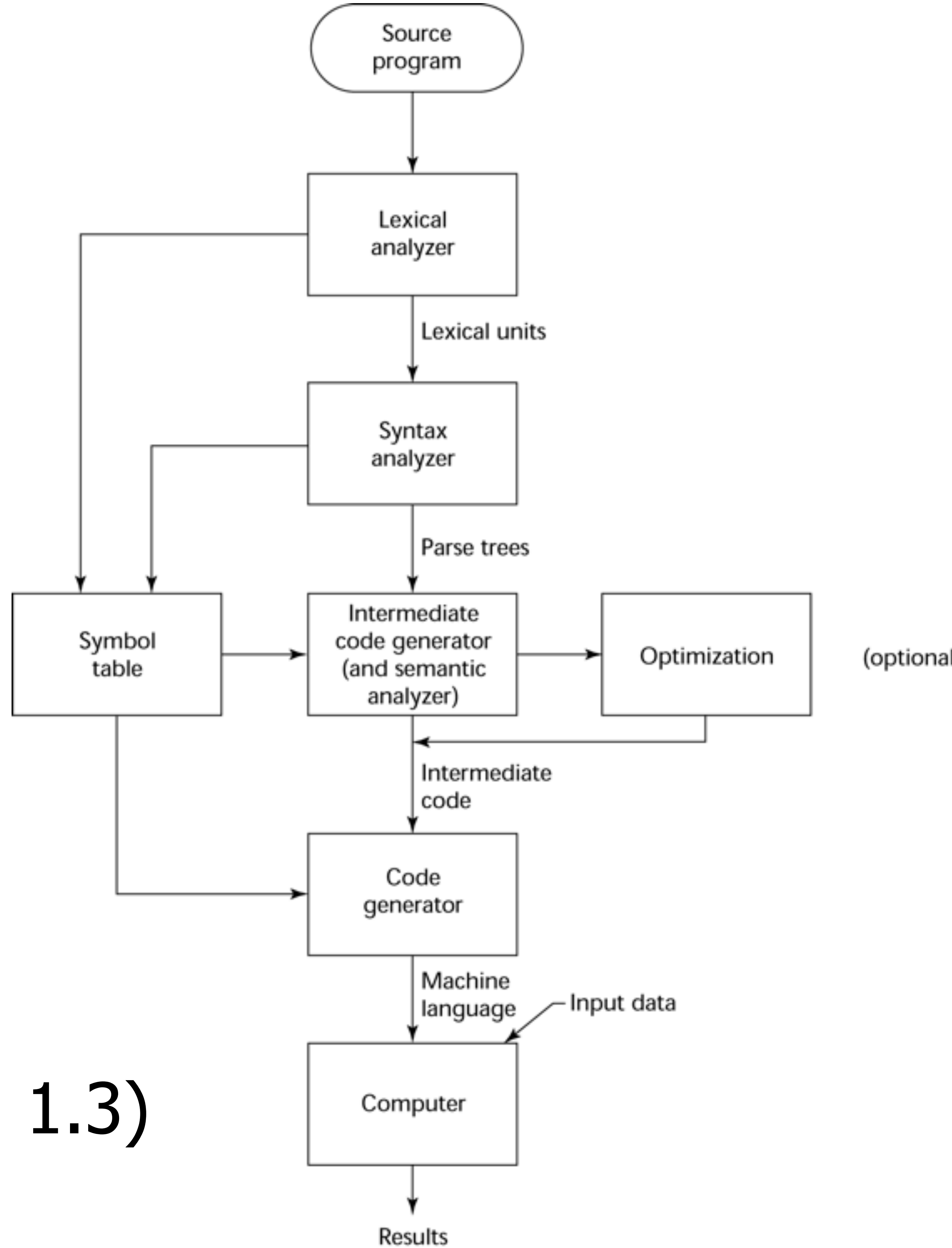
Implementations of PL

- It is important to understand how features and constructs of a programming language, e.g., subroutine calls, are implemented
  - Implementation of a PL construct means its realization in a lower-level language, e.g. assembly
    → mapping/translation from a high-level language to a low-level language
  - Why the need to know implementations?
    Understand whether a construct may be implemented efficiently, know different implementation methods and their tradeoffs, etc.

Implementation by Compilation

- Translate a high-level program into equivalent machine code automatically by another program (compiler)
- Compilation process has several phases:
  - Lexical analysis: converts characters in the source program into lexical units
  - Syntax analysis: transforms lexical units into parse trees which represent syntactic structure of program
  - Semantics analysis: generate intermediate code
  - Code generation: machine code is generated
  - Link and load

Compilation Process



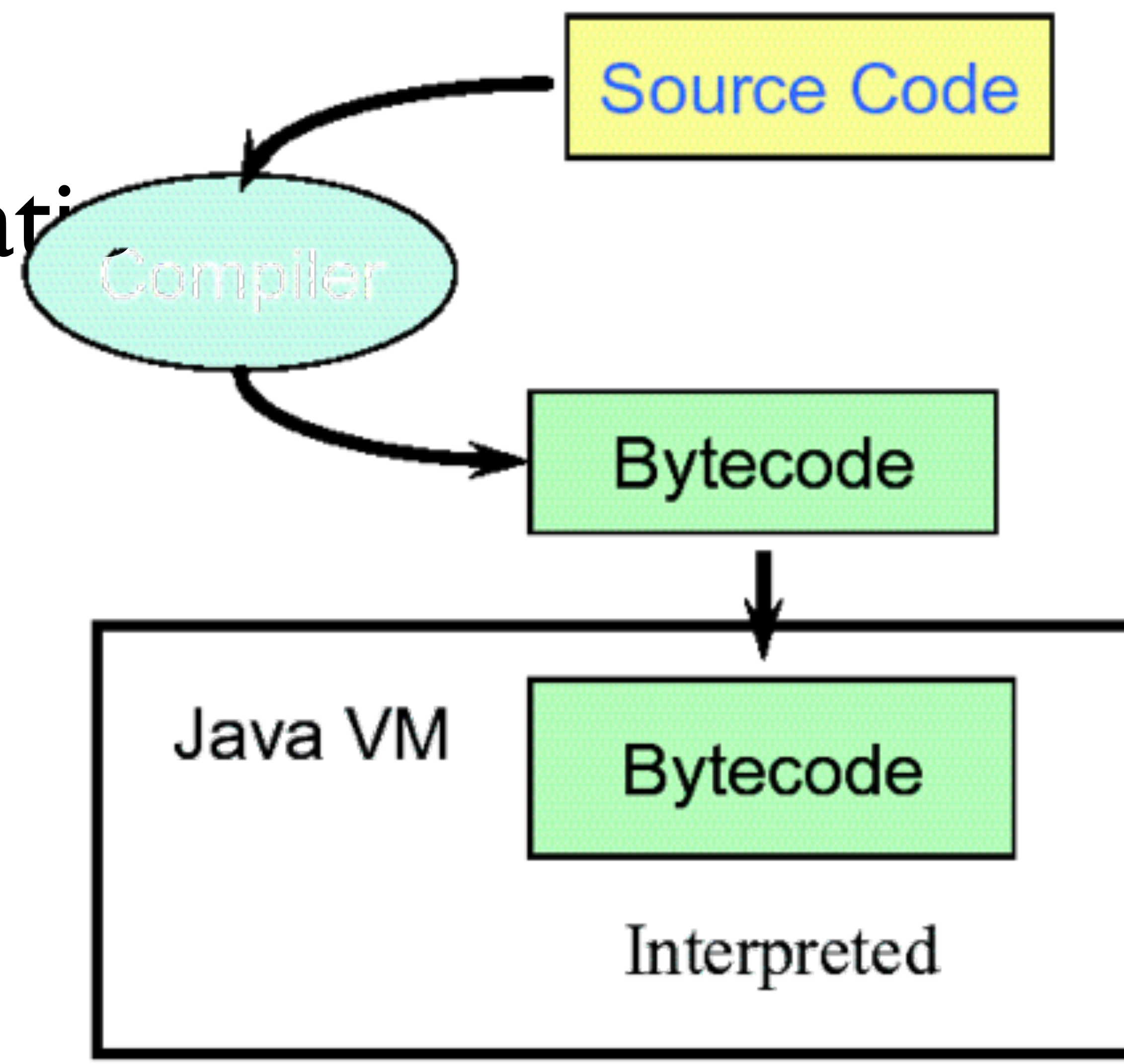
(Fig. 1.3)

Implementation by Interpretation

- Program interpreted by another program (interpreter) without translation
  - Interpreter acts a simulator or virtual machine
- Easier implementation of programs (run-time errors can easily and immediately displayed)
- Slower execution (10 to 100 times slower than compiled programs)
- Often requires more space
- Popular with some Web scripting languages (e.g., JavaScript)

Hybrid Implementation Systems

- A high-level language program is translated to an intermediate language that allows easy interpretation
  - Faster than pure interpretati



Summary

- Most important criteria for evaluating programming languages include:
  - Readability, writability, reliability, cost
- Major influences on language design have been application domains, machine architecture and software development methodologies
- The major methods of implementing programming languages are: compilation, pure interpretation, and hybrid implementation