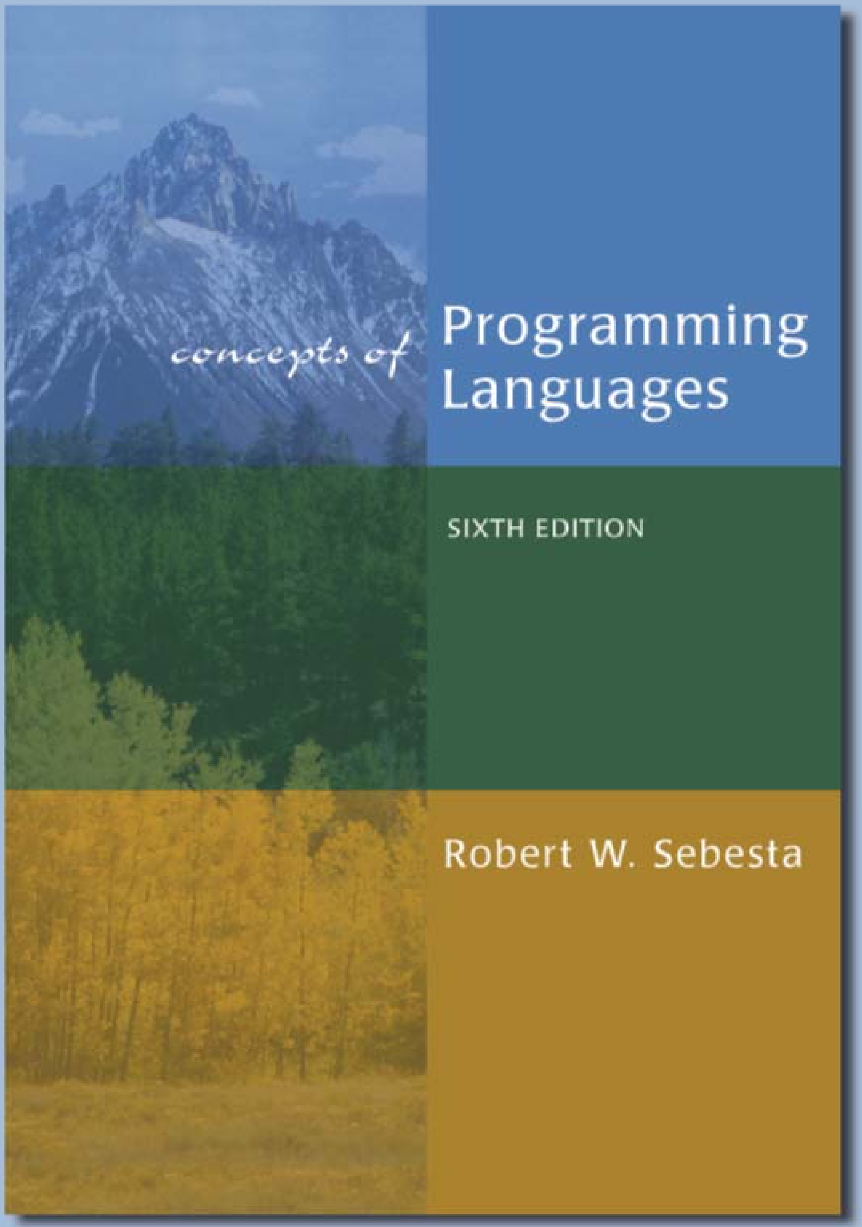
Chapters 1 & 2

Preliminaries



ISBN 0-321-19362-8

Chapter 1 Topics

- Motivation
- Programming Domains
- Language Evaluation Criteria
- Influences on Language Design
- Language Categories
- Language Design Trade-Offs
- Implementation Methods
- Programming Environments

Motivation
Why Study Programming Languages?

- Increased ability to express ideas
- Improved background for choosing appropriate languages
- Greater ability to learn new languages
- Understand significance of implementation
- Ability to design new languages
- Overall advancement of computing

Programming Domains

- Scientific applications
  - Large number of floating point computations
- Business applications
  - Produce reports, use decimal numbers and characters
- Artificial intelligence
  - Symbols rather than numbers manipulated. Code = Data.
- Systems programming
  - Need efficiency because of continuous use. Low-level control.
- Scripting languages
  - Put a list of commands in a file to be executed. Glue apps.
- Special-purpose languages
  - Simplest/fastest solution for a particular task.

Language Evaluation Criteria

- Readability
- Writability
- Reliability
- Cost
- Others

Language Evaluation Criteria Readability

- Overall simplicity
  - Too many features is bad
  - Multiplicity of features is bad
- Orthogonality
  - Makes the language easy to learn and read
  - Meaning is context independent
  - A relatively small set of primitive constructs can be combined in a relatively small number of ways
  - Every possible combination is legal
  - Lack of orthogonality leads to exceptions to rules

Language Evaluation Criteria Writability

- Simplicity and orthogonality
- Support for abstraction
- Expressiveness

Language Evaluation Criteria Reliability

- Type checking
- Exception handling
- Aliasing
- Readability and writability

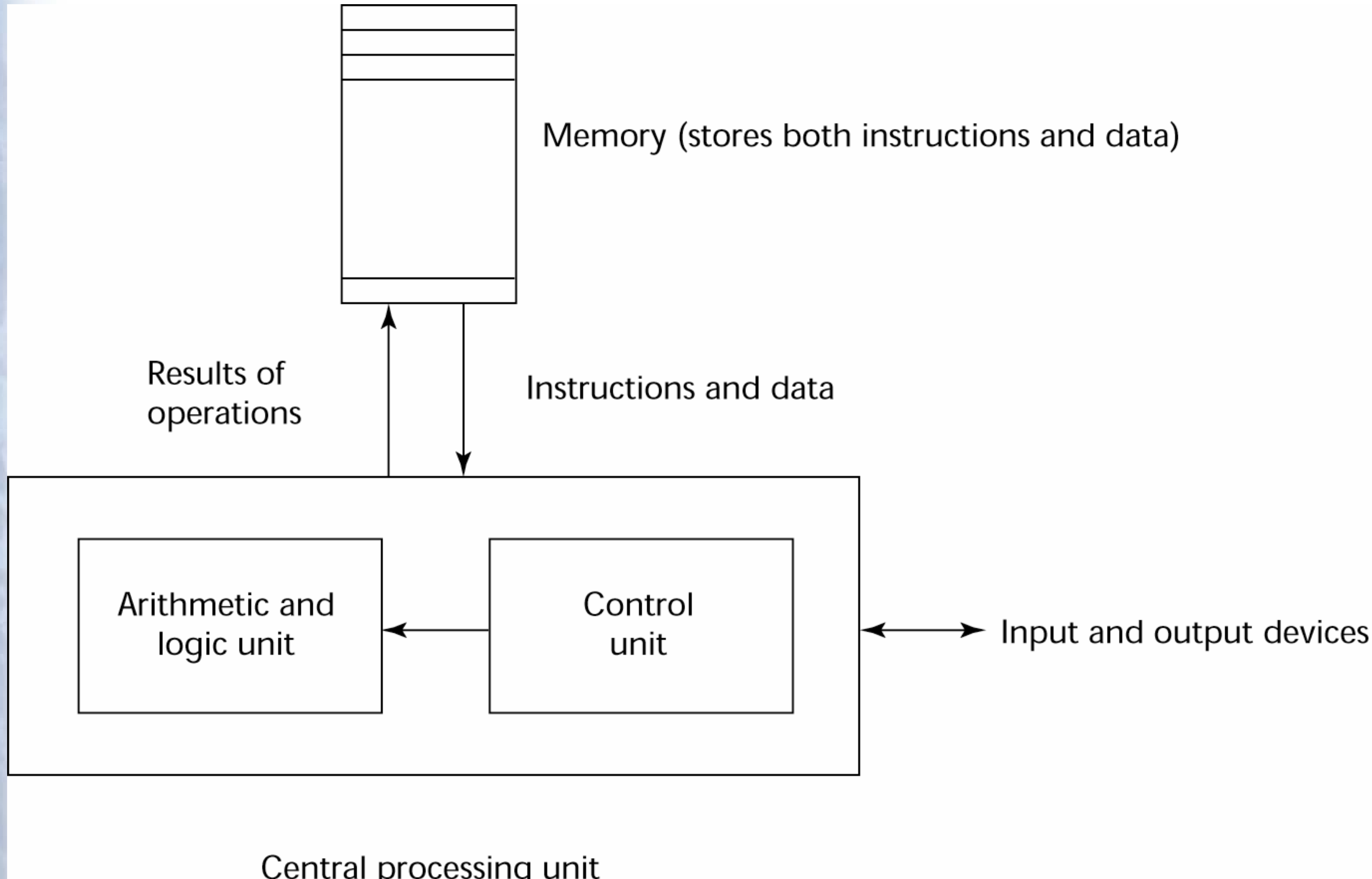
Language Evaluation Criteria

- Cost
  - Categories
    - Training programmers to use language
    - Writing programs
    - Compiling programs
    - Executing programs
    - Language implementation system
    - Reliability
    - Maintaining programs
- Others: portability, generality, well-definedness

Influences on Language Design

- Computer architecture: Von Neumann
- We use imperative languages, at least in part, because we use von Neumann machines
  - Data and programs stored in same memory
  - Memory is separate from CPU
  - Instructions and data are piped from memory to CPU
  - Basis for imperative languages
    - Variables model memory cells
    - Assignment statements model piping
    - Iteration is efficient

Von Neumann Architecture



Influences on Language Design

- Programming methodologies
  - 1950s and early 1960s: Simple applications; worry about machine efficiency
  - Late 1960s: People efficiency became important; readability, better control structures
    - Structured programming
    - Top-down design and step-wise refinement
  - Late 1970s: Process-oriented to data-oriented
    - data abstraction
  - Middle 1980s: Object-oriented programming

Language Categories

- Imperative
  - Central features are variables, assignment statements, and iteration
  - FORTRAN, C, Pascal
- Functional
  - Main means of making computations is by applying functions to given parameters
  - LISP, Scheme

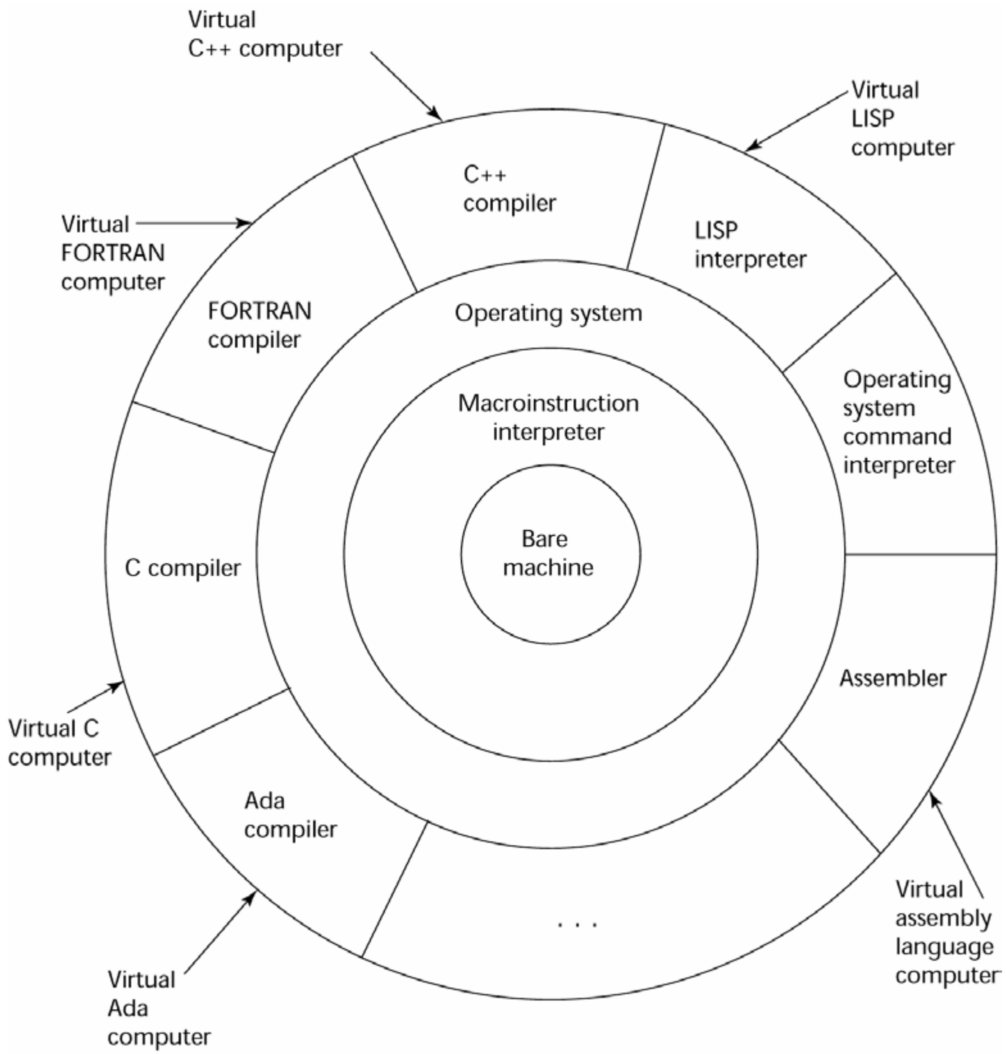
Language Categories

- Logic
  - Rule-based
  - Rules are specified in no special order
  - Prolog
- Object-oriented
  - Encapsulate data objects with processing
  - Inheritance and dynamic type binding
  - Grew out of imperative languages
  - C++, Java

Some Language Design Trade-Offs

- Reliability vs. cost of execution
- Readability vs. writability
- Flexibility vs. safety

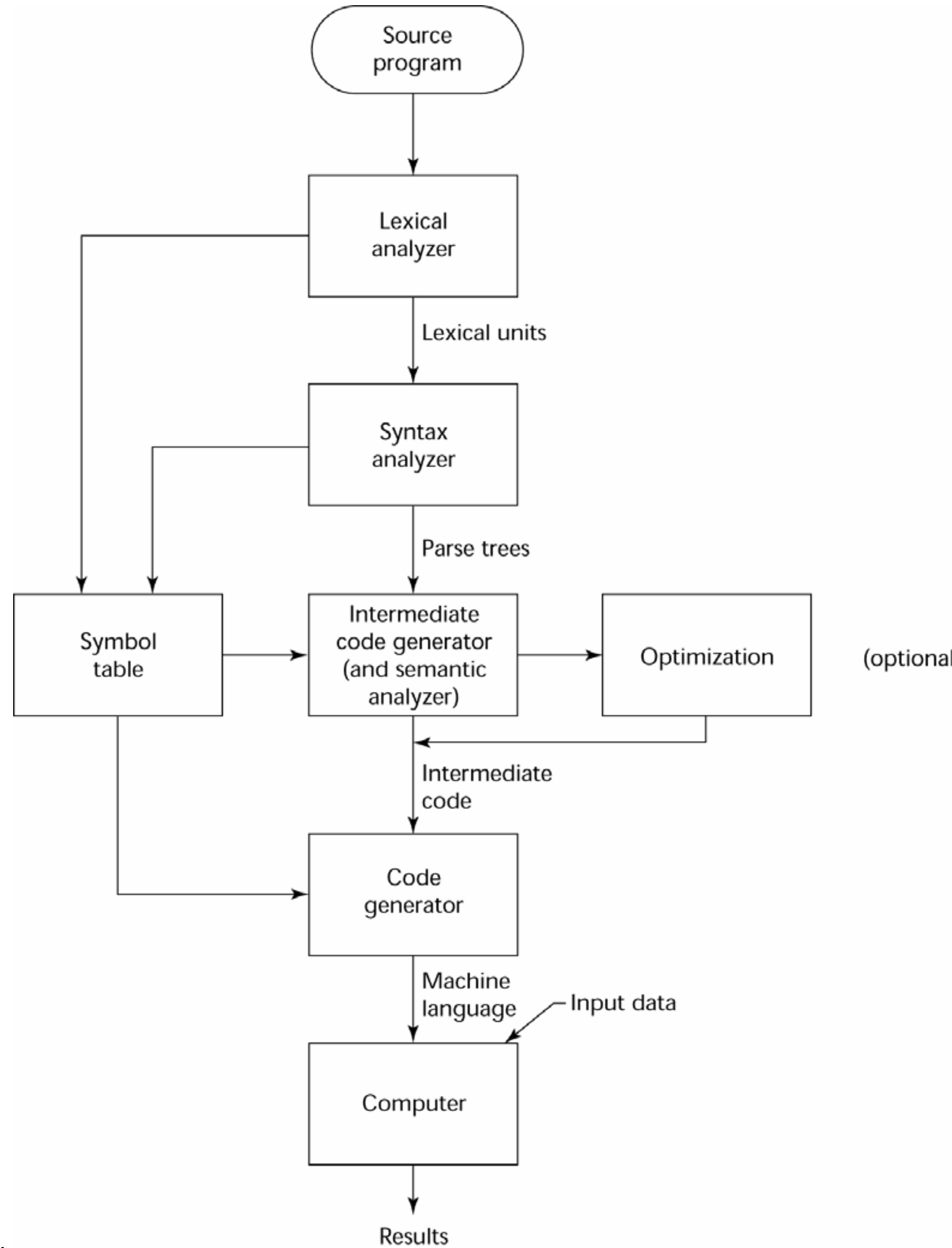
Layered View of Computer



Implementation Methods

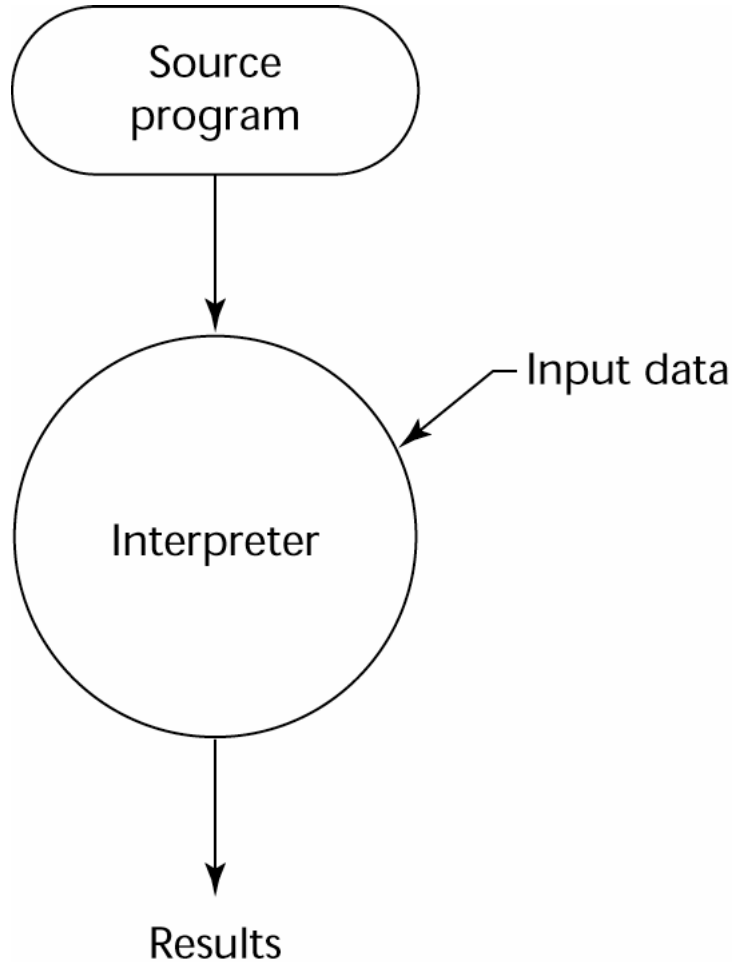
- Compilation
  - Translate high-level program to machine code
  - Slow translation
  - Fast execution

Compilation Process



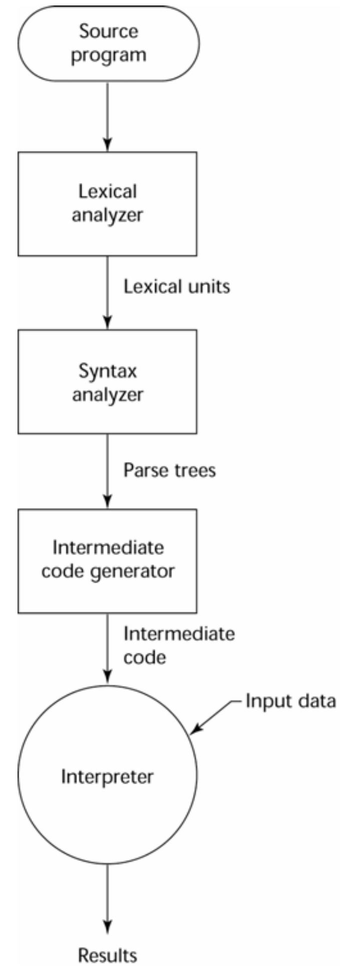
Implementation Methods

- Pure interpretation
  - No translation
  - Slow execution
  - Becoming rare



Implementation Methods

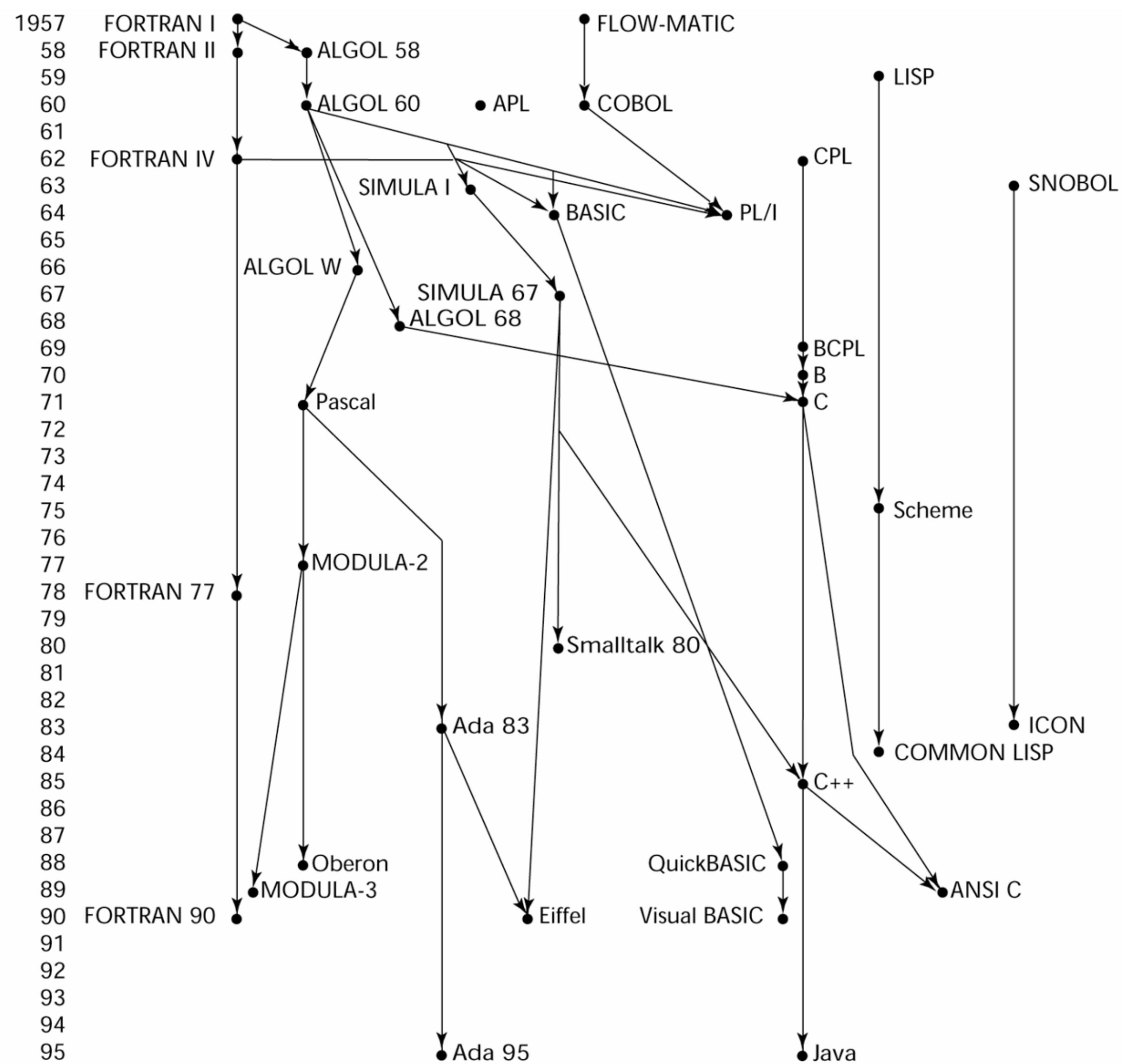
- Hybrid implementation systems
  - Small translation cost
  - Medium execution speed
  - Portable



Programming Environments

- The collection of tools used in software development
- UNIX
  - An older operating system and tool collection
- Borland JBuilder
  - An integrated development environment for Java
- Microsoft Visual Studio.NET
  - A large, complex visual environment
  - Used to program in C#, Visual BASIC.NET, Jscript, J#, or C++

Genealogy of Common Languages



Zuse's Plankalkül - 1945

- Never implemented
- Advanced data structures
  - floating point, arrays, records
- Invariants

Plankalkül

- Notation:

A[7] = 5 * B[6]

```
     |  5  *  B  =>  A
  V  |  6            7         (subscripts)
  S  |  1.n          1.n       (data types)
```

Pseudocodes - 1949

- What was wrong with using machine code?
  - Poor readability
  - Poor modifiability
  - Expression coding was tedious
  - Machine deficiencies--no indexing or floating point

Pseudocodes

- Short code; 1949; BINAC; Mauchly
  - Expressions were coded, left to right
  - Some operations:

    1n => (n+2)nd power

    2n => (n+2)nd root

    07 => addition

Pseudocodes

- Speedcoding; 1954; IBM 701, Backus
  - Pseudo ops for arithmetic and math functions
  - Conditional and unconditional branching
  - Autoincrement registers for array access
  - Slow!
  - Only 700 words left for user program

Pseudocodes

- Laning and Zierler System - 1953
  - Implemented on the MIT Whirlwind computer
  - First "algebraic" compiler system
  - Subscripted variables, function calls, expression translation
  - Never ported to any other machine

IBM 704 and FORTRAN

- FORTRAN I - 1957

  (FORTRAN 0 - 1954 - not implemented)
  - Designed for the new IBM 704, which had index registers and floating point hardware
  - Environment of development:
    - Computers were small and unreliable
    - Applications were scientific
    - No programming methodology or tools
    - Machine efficiency was most important

IBM 704 and FORTRAN

- Impact of environment on design of FORTRAN I
  - No need for dynamic storage
  - Need good array handling and counting loops
  - No string handling, decimal arithmetic, or powerful input/output (commercial stuff)

IBM 704 and FORTRAN

- First implemented version of FORTRAN
  - Names could have up to six characters
  - Post-test counting loop (`DO`)
  - Formatted I/O
  - User-defined subprograms
  - Three-way selection statement (arithmetic `IF`)
  - No data typing statements

IBM 704 and FORTRAN

- First implemented version of FORTRAN
  - No separate compilation
  - Compiler released in April 1957, after 18 worker-years of effort
  - Programs larger than 400 lines rarely compiled correctly, mainly due to poor reliability of the 704
  - Code was very fast
  - Quickly became widely used

IBM 704 and FORTRAN

- FORTRAN II - 1958
  - Independent compilation
  - Fix the bugs

IBM 704 and FORTRAN

- FORTRAN IV - 1960-62
  - Explicit type declarations
  - Logical selection statement
  - Subprogram names could be parameters
  - ANSI standard in 1966

IBM 704 and FORTRAN

- FORTRAN 77 - 1978
  - Character string handling
  - Logical loop control statement
  - **`IF-THEN-ELSE`** statement

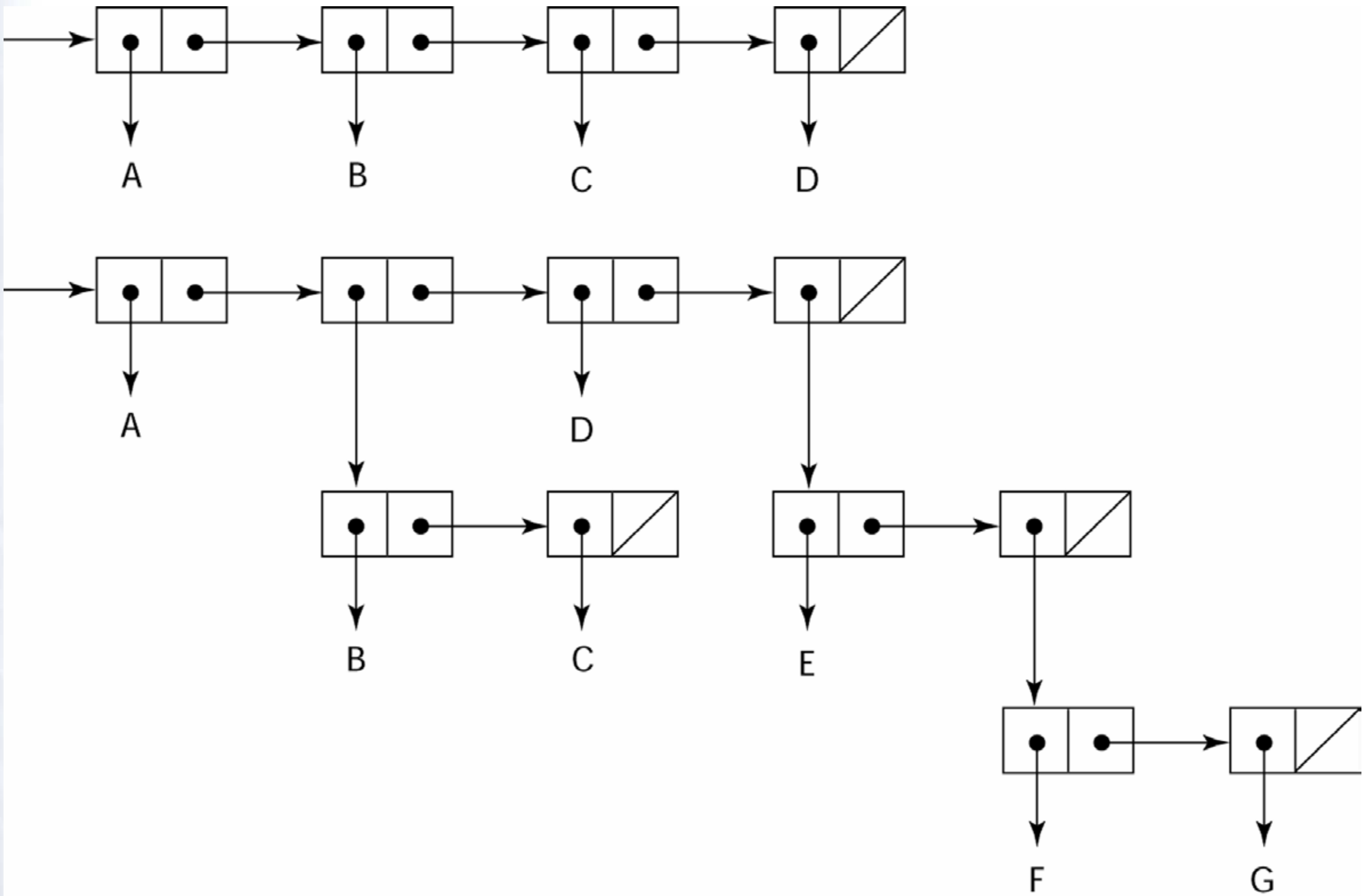
IBM 704 and FORTRAN

- FORTRAN 90 - 1990
  - Modules
  - Dynamic arrays
  - Pointers
  - Recursion
  - **`CASE`** statement
  - Parameter type checking

LISP - 1959

- LISt Processing language
  (Designed at MIT by McCarthy)
- AI research needed a language that:
  - Process data in lists (rather than arrays)
  - Symbolic computation (rather than numeric)
- Only two data types: atoms and lists
- Syntax is based on lambda calculus
- Same syntax for data and code

Representation of Two LISP Lists



LISP

- Pioneered functional programming
  - No need for variables or assignment
  - Control via recursion and conditional expressions
- Still the dominant language for AI
- COMMON LISP and Scheme are contemporary dialects of LISP
- ML, Miranda, and Haskell are related languages

ALGOL 58 and 60

- Environment of development:
  - FORTRAN had (barely) arrived for IBM 70x
  - Many other languages were being developed, all for specific machines
  - No portable language; all were machine-dependent
  - No universal language for communicating algorithms

ALGOL 58 and 60

- ACM and GAMM met for four days for design
- Goals of the language:
  - Close to mathematical notation
  - Good for describing algorithms
  - Must be translatable to machine code

ALGOL 58 and 60

- ALGOL 58 Language Features:
  - Concept of type was formalized
  - Names could have any length
  - Arrays could have any number of subscripts
  - Parameters were separated by mode (in & out)
  - Subscripts were placed in brackets
  - Compound statements (**`begin ... end`**)
  - Semicolon as a statement separator
  - Assignment operator was :=
  - **`if`** had an **`else-if`** clause
  - No I/O - "would make it machine dependent"

ALGOL 58

- Comments:
  - Not meant to be implemented, but variations of it were (MAD, JOVIAL)
  - Although IBM was initially enthusiastic, all support was dropped by mid-1959

ALGOL 58 and 60

- ALGOL 60
  - Modified ALGOL 58 at 6-day meeting in Paris
  - New features:
    - Block structure (local scope)
    - Two parameter passing methods
    - Subprogram recursion
    - Stack-dynamic arrays
    - Still no I/O and no string handling

ALGOL 60

- Successes:
  - It was the standard way to publish algorithms for over 20 years
  - All subsequent imperative languages are based on it
  - First machine-independent language
  - First language whose syntax was formally defined (BNF)

ALGOL 60

- Failure:
  - Never widely used, especially in U.S.
- Reasons:
  - No I/O and the character set made programs non-portable
  - Too flexible--hard to implement
  - Entrenchment of FORTRAN
  - Formal syntax description
  - Lack of support of IBM

COBOL - 1960

- Environment of development:
  - UNIVAC was beginning to use FLOW-MATIC
  - USAF was beginning to use AIMACO
  - IBM was developing COMTRAN

COBOL

- Based on FLOW-MATIC
- FLOW-MATIC features:
  - Names up to 12 characters, with embedded hyphens
  - English names for arithmetic operators (no arithmetic expressions)
  - Data and code were completely separate
  - Verbs were first word in every statement

COBOL

- First Design Meeting (Pentagon) - May 1959
- Design goals:
  - Must look like simple English
  - Must be easy to use, even if that means it will be less powerful
  - Must broaden the base of computer users
  - Must not be biased by current compiler problems
- Design committee members were all from computer manufacturers and DoD branches
- Design Problems: arithmetic expressions? subscripts? Fights among manufacturers

COBOL

- Contributions:
  - First macro facility in a high-level language
  - Hierarchical data structures (records)
  - Nested selection statements
  - Long names (up to 30 characters), with hyphens
  - Separate data division

COBOL

- Comments:
  - First language required by DoD; would have failed without DoD
  - Still the most widely used business applications language

BASIC - 1964

- Designed by Kemeny & Kurtz at Dartmouth
- 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
- Current popular dialect: Visual BASIC
- First widely used language with time sharing

PL/I - 1965

- Designed by IBM and SHARE
- Computing situation in 1964 (IBM's point of view)
  – Scientific computing
    - IBM 1620 and 7090 computers
    - FORTRAN
    - SHARE user group
  – Business computing
    - IBM 1401, 7080 computers
    - COBOL
    - GUIDE user group

PL/I

- By 1963, however,
  - Scientific users began to need more elaborate I/O, like COBOL had; Business users began to need floating point and arrays (MIS)
  - It looked like many shops would begin to need two kinds of computers, languages, and support staff--too costly
- The obvious solution:
  - Build a new computer to do both kinds of applications
  - Design a new language to do both kinds of applications

PL/I

- Designed in five months by the 3 X 3 Committee
- PL/I contributions:
  - First unit-level concurrency
  - First exception handling
  - Switch-selectable recursion
  - First pointer data type
  - First array cross sections

PL/I

- Comments:
  - Many new features were poorly designed
  - Too large and too complex
  - Was (and still is) actually used for both scientific and business applications

APL and SNOBOL

- Characterized by dynamic typing and dynamic storage allocation
- APL (A Programming Language) 1962
  - Designed as a hardware description language (at IBM by Ken Iverson)
  - Highly expressive (many operators, for both scalars and arrays of various dimensions)
  - Programs are very difficult to read

APL and SNOBOL

- SNOBOL(1964)
  - Designed as a string manipulation language (at Bell Labs by Farber, Griswold, and Polensky)
  - Powerful operators for string pattern matching

SIMULA 67 - 1967

- Designed primarily for system simulation (in Norway by Nygaard and Dahl)
- Based on ALGOL 60 and SIMULA I
- Primary Contribution:
  – Co-routines - a kind of subprogram
  – Implemented in a structure called a class
  – Classes are the basis for data abstraction
  – Classes are structures that include both local data and functionality
  – Objects and inheritance

ALGOL 68 - 1968

- From the continued development of ALGOL 60, but it is not a superset of that language
- Design is based on the concept of orthogonality
- Contributions:
  - User-defined data structures
  - Reference types
  - Dynamic arrays (called flex arrays)

ALGOL 68

- Comments:
  - Had even less usage than ALGOL 60
  - Had strong influence on subsequent languages, especially Pascal, C, and Ada

Important ALGOL Descendants

- Pascal - 1971
  - Designed by Wirth, who quit the ALGOL 68 committee (didn't like the direction of that work)
  - Designed for teaching structured programming
  - Small, simple, nothing really new
  - From mid-1970s until the late 1990s, it was the most widely used language for teaching programming in colleges

Important ALGOL Descendants

- C - 1972
  - Designed for systems programming (at Bell Labs by Dennis Richie)
  - Evolved primarily from B, but also ALGOL 68
  - Powerful set of operators, but poor type checking
  - Initially spread through UNIX

Important ALGOL Descendants

- Modula-2 - mid-1970s (Wirth)
  - Pascal plus modules and some low-level features designed for systems programming
- Modula-3 - late 1980s (Digital & Olivetti)
  - Modula-2 plus classes, exception handling, garbage collection, and concurrency

Important ALGOL Descendants

- Oberon - late 1980s (Wirth)
  - Adds support for OOP to Modula-2
  - Many Modula-2 features were deleted (e.g., **`for`** statement, enumeration types, **`with`** statement, noninteger array indices)

Prolog - 1972

- Developed at the University of Aix-Marseille, by Comerauer and Roussel, with some help from Kowalski at the University of Edinburgh
- Based on formal logic
- Non-procedural
- Can be summarized as being an intelligent database system that uses an inferencing process to infer the truth of given queries

Ada - 1983 (began in mid-1970s)

- Huge design effort, involving hundreds of people, much money, and about eight years
- Environment: More than 450 different languages being used for DOD embedded systems (no software reuse and no development tools)
- Contributions:
  - Packages - support for data abstraction
  - Exception handling - elaborate
  - Generic program units
  - Concurrency - through the tasking model

Ada

- Comments:
  - Competitive design
  - Included all that was then known about software engineering and language design
  - First compilers were very difficult; the first really usable compiler came nearly five years after the language design was completed

Ada

- Ada 95 (began in 1988)
  - Support for OOP through type derivation
  - Better control mechanisms for shared data (new concurrency features)
  - More flexible libraries

Smalltalk - 1972-1980

- Developed at Xerox PARC, initially by Alan Kay, later by Adele Goldberg
- First full implementation of an object-oriented language (data abstraction, inheritance, and dynamic type binding)
- Pioneered the graphical user interface everyone now uses

C++ - 1985

- Developed at Bell Labs by Stroustrup
- Evolved from C and SIMULA 67
- Facilities for object-oriented programming, taken partially from SIMULA 67, were added to C
- Also has exception handling
- A large and complex language, in part because it supports both procedural and OO programming
- Rapidly grew in popularity, along with OOP
- ANSI standard approved in November, 1997

C++ Related Languages

- Eiffel - a related language that supports OOP
  - (Designed by Bertrand Meyer - 1992)
  - Not directly derived from any other language
  - Smaller and simpler than C++, but still has most of the power
- Delphi (Borland)
  - Pascal plus features to support OOP
  - More elegant and safer than C++

Java (1995)

- Developed at Sun in the early 1990s
- Based on C++
  - Significantly simplified (does not include **`struct, union, enum`**, pointer arithmetic, and half of the assignment coercions of C++)
  - Supports *only* OOP
  - Has references, but not pointers
  - Includes support for applets and a form of concurrency

Scripting Languages for the Web

- JavaScript
  - Used in Web programming (client-side) to create dynamic HTML documents
  - Related to Java only through similar syntax
- PHP
  - Used for Web applications (server-side); produces HTML code as output

C#

- Part of the .NET development platform
- Based on C++ and Java
- Provides a language for component-based software development
- All .NET languages (C#, Visual BASIC.NET, Managed C++, J#.NET, and Jscript.NET) use Common Type System (CTS), which provides a common class library
- Likely to become widely used