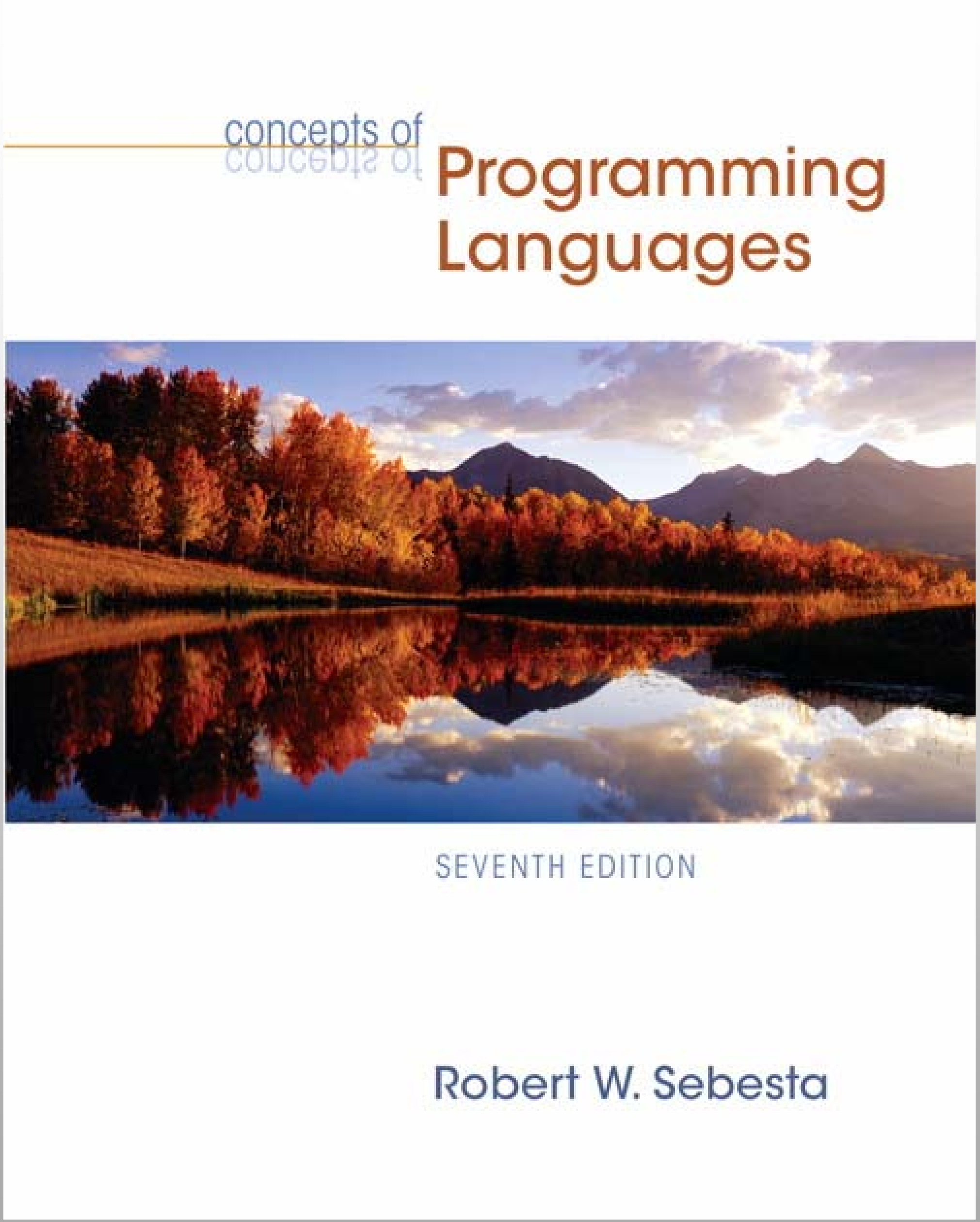
Chapter 1

Preliminaries



ISBN 0-321-33025-0

Chapter 1 Topics

- Reasons for Studying Concepts of Programming Languages
- Programming Domains
- Language Evaluation Criteria
- Influences on Language Design
- Language Categories
- Language Design Trade-Offs
- Implementation Methods
- Programming Environments

Programming is an unnatural act

- Alan Perlis
- 1922–1990
- First President of the ACM
- First Turing Award winner
- Member of the Algol–60 design team

An example of an early computer

- Harvard Mark I (IBM, Aiken, 1948)
  - electro-mechanical
    - ENIAC is an electronic copy of Mark I design
  - executed 3 operations each second (3 IPS)
  - remained in use until 1959
  - 51' long, 8' high, 3' deep
  - 730,000 parts (relays, switches, wheels, shafts), 530 miles of wiring, 18,000 vacuum tubes, ...
- How many programmers could one 'buy' with the price of one computer?

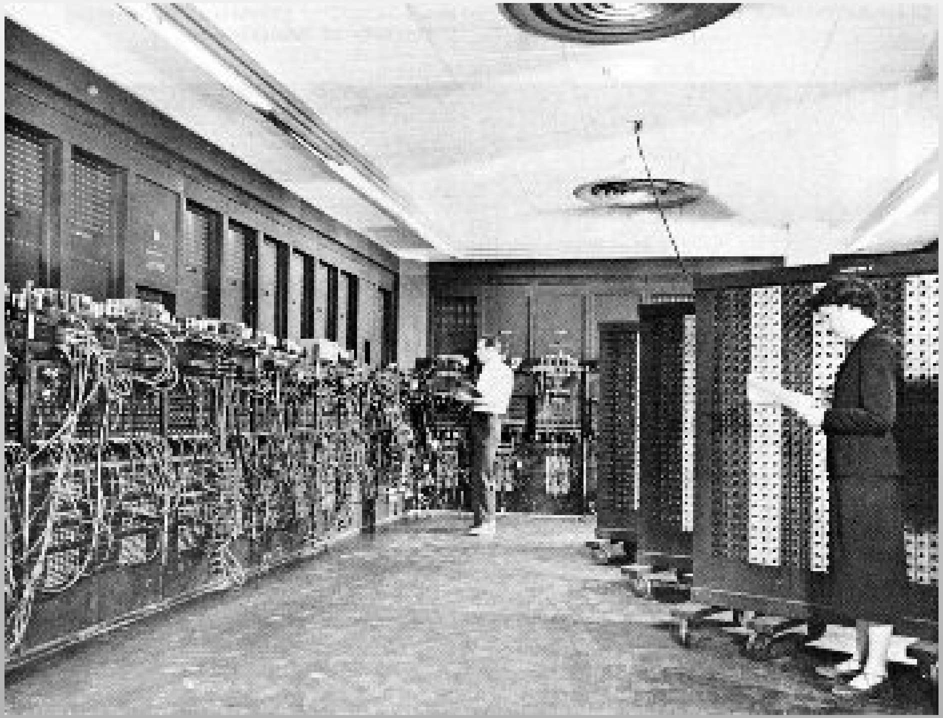
An example of a new computer

- Sun Fire 15K
  - 106 UltraSPARC III processors
    - 900 MHz to 1.2 GHz clock speed
    - 29 million transistors
    - supports 4 Gb of memory
  - 602,270 JBB operations per second
  - list price $3,739,230.00 (72 processors)

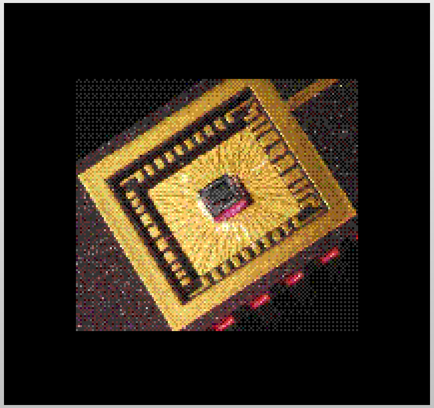
Picture of Mark I

Computer Size

ENIAC then…



ENIAC today…



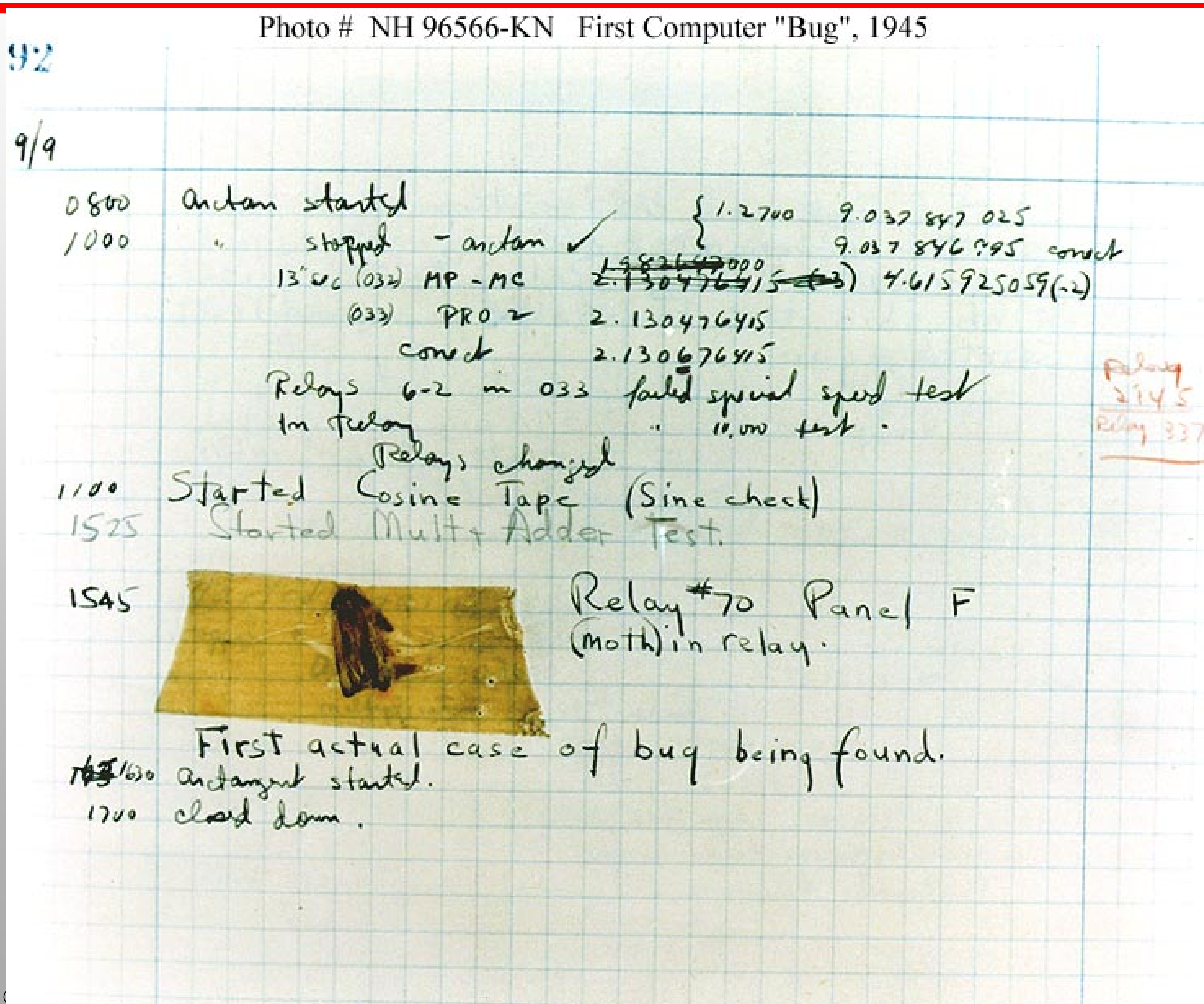
- With computers (small) size does matter!

An example of an early program

```
27bdffd0 afbf0014 0c1002a8 00000000 0c1002a8 afa2001c 8fa4001c
00401825 10820008 0064082a 10200003 00000000 10000002 00832023
00641823 1483fffa 0064082a 0c1002b2 00000000 8fbf0014 27bd0020
03e00008 00001025
```

- Euclid's algorithm for GCD (greatest common divisor)
  - actually this is for a quite new computer (MIPS R4000)
- Writing programs in this way is very expensive and hard
  - but the early computers cost much much more
  - even *using* the computer cost more than programming it

With Mark II came the bugs



Problems of machine code

- Programming = *coding* in the true meaning of the word
- Code is not
  - reusable: monolithic 'structure'
  - relocatable: consider adding one instruction in the middle
  - readable (more important)
- Practically impossible to create large programs

Symbolic assembly language

- Assembler
  - *translator* from symbolic language to machine language (one-to-one mapping)
  - tool to *assemble* the symbolic program in the machine
- Advantages
  - relocatable & reusable (copy) programs
  - *macro expansion*
    - first step towards higher-level programming
  - larger programs (like operating systems) possible

Euclid's GCD program in MIPS assembly language

```
    addiu   sp,sp,-32
    sw      ra,20(sp)
    jal     getint
    nop
    jal     getint
    sw      v0,28(sp)
    lw      a0,28(sp)
    move    v1,v0
    beq     a0,v0,D
    slt     at,v1,a0
A:  beq     at,zero,B
    nop
    b       C
    subu    a0,a0,v1
B:  subu    v1,v1,a0
C:  bne     a0,v1,A
    slt     at,v1,a0
D:  jal     putint
    nop
    lw      ra,20(sp)
    addiu   sp,sp,32
    jr      ra
    move    v0,zero
```

Problems of assembler

- Each kind of computer has its own
- Programmers must learn to think like computers
- Maintenance of larger programs is difficult
- Higher-level languages
  - portability
  - natural notation (for anything)
  - support to software development

First high-level language

- Fortran (Backus, 1957)
  - IBM Mathematical **For**mula **Tran**slator
  - *compilation* instead of translation
  - language for scientific computing
    - most important task in those days
  - efficiency important to replace assemblers
  - introduced many important language concepts that are still in use

A Fortran program

```
C  FORTRAN PROGRAM
DIMENSION A(99)
REAL MEAN

READ(1,5) N
5 FORMAT(I2)

READ(1,10) (A(I), I=1,N)
10 FORMAT(6F10.5)

SUM = 0.0
DO 15 I=1,N
15 SUM = SUM + A(I)

MEAN = SUM/FLOAT(N)
NUMBER = 0
DO 20 I=1,N
IF(A(I) .LE. MEAN) GOTO 20
NUMBER = NUMBER + 1
20 CONTINUE

WRITE(2,25) MEAN, NUMBER
25 FORMAT(8H MEAN = , F10.5, 5X, 20H NUMBERS OVER MEAN =, I5)
STOP
END
```

What matters in programming?

- 1950s: cost and use of machines
- Nowadays
  - problems other than efficiency are often more important
    - performance gap between compiled and hand-tailored machine code has diminished
    - modern hardware is too complicated for humans
  - cost of labor has far surpassed the cost of machinery
    - standard PC costs like NT 20,000
  - software systems are getting more and more complex
  - problems to solve are getting difficult even to *define*

Why are there so many programming languages?

- Read the "Perlis quotes"
- Evolution
  - CS is constantly finding 'better' ways to do things
  - structured programming, modules, o-o, ...
- Special languages for special purposes
  - Scientific applications, e.g. MATLAB, Mathematica, Fortran, ALGO 60 etc.
  - Business applications, e.g. COBOL
  - Artificial intelligence (AI), e.g. LISP, Ada
  - Systems programming, e.g. PL/S, C/C++, Pascal
  - Web software, e.g. HTML, XML, PHP, .NET, Java
- Personal preference
  - We are not all driving a NISSON or TOYOTA!?

Why are some programming languages more successful?

- Expressive power
  - in principle, all languages are Turing-complete
  - has a huge effect on programmer's ability to
    - write, read, and maintain
    - understand and analyze
  - abstraction facilities (for computation & data)
- Ease of use
  - low learning curve (Basic, Logo, Pascal)
- Ease of implementation
  - Pascal & p-code (forefather of Java VM) made it easy to port compilers
  - free availability in general

More reasons for success

- Excellent compilers and tools
  - fast compiled code (Fortran)
  - debugging tools
  - project management tools
  - teamwork tools
- Economics, inertia
  - 10000000 lines of Cobol is hard to rewrite
  - 100000 Cobol programmers are hard to re-train
- Patronage
  - many languages have powerful 'sponsors'
    - Cobol, PL/I, Ada, Visual Basic, C#

Reasons for Studying Concepts of Programming Languages

- Increased ability to express ideas
- Improved background for choosing appropriate languages
- Increased ability to learn new languages
- Better understanding of significance of implementation
- Overall advancement of computing

Programming Domains

- Scientific applications
  - Large number of floating point computations
  - Fortran
- Business applications
  - Produce reports, use decimal numbers and characters
  - COBOL
- Artificial intelligence
  - Symbols rather than numbers manipulated
  - LISP
- Systems programming
  - Need efficiency because of continuous use
  - C
- Web Software
  - Eclectic collection of languages: markup (e.g., XHTML), scripting (e.g., PHP), general-purpose (e.g., Java)

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**: conformance to specifications (i.e., performs to its specifications)
- **Cost**: the ultimate total cost

Evaluation Criteria: Readability

- Overall simplicity
  - A manageable set of features and constructs
  - Few feature multiplicity (means of doing the same operation)
  - Minimal operator overloading
- Orthogonality
  - A relatively small set of primitive constructs can be combined in a relatively small number of ways
  - Every possible combination is legal
- Control statements
  - The presence of well-known control structures (e.g., `while` statement)
- Data types and structures
  - The presence of adequate facilities for defining data structures
- Syntax considerations
  - Identifier forms: flexible composition
  - Special words and methods of forming compound statements
  - Form and meaning: self-descriptive constructs, meaningful keywords

Evaluation Criteria: Writability

- Simplicity and orthogonality
  - Few constructs, a small number of primitives, a small set of rules for combining them
- Support for abstraction
  - The ability to define and use complex structures or operations in ways that allow details to be ignored
- Expressivity
  - A set of relatively convenient ways of specifying operations
  - Example: the inclusion of `for` statement in many modern languages

Evaluation Criteria: Reliability

- Type checking
  - Testing for type errors
- Exception handling
  - Intercept run-time errors and 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

Evaluation Criteria: Cost

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

Evaluation Criteria: 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

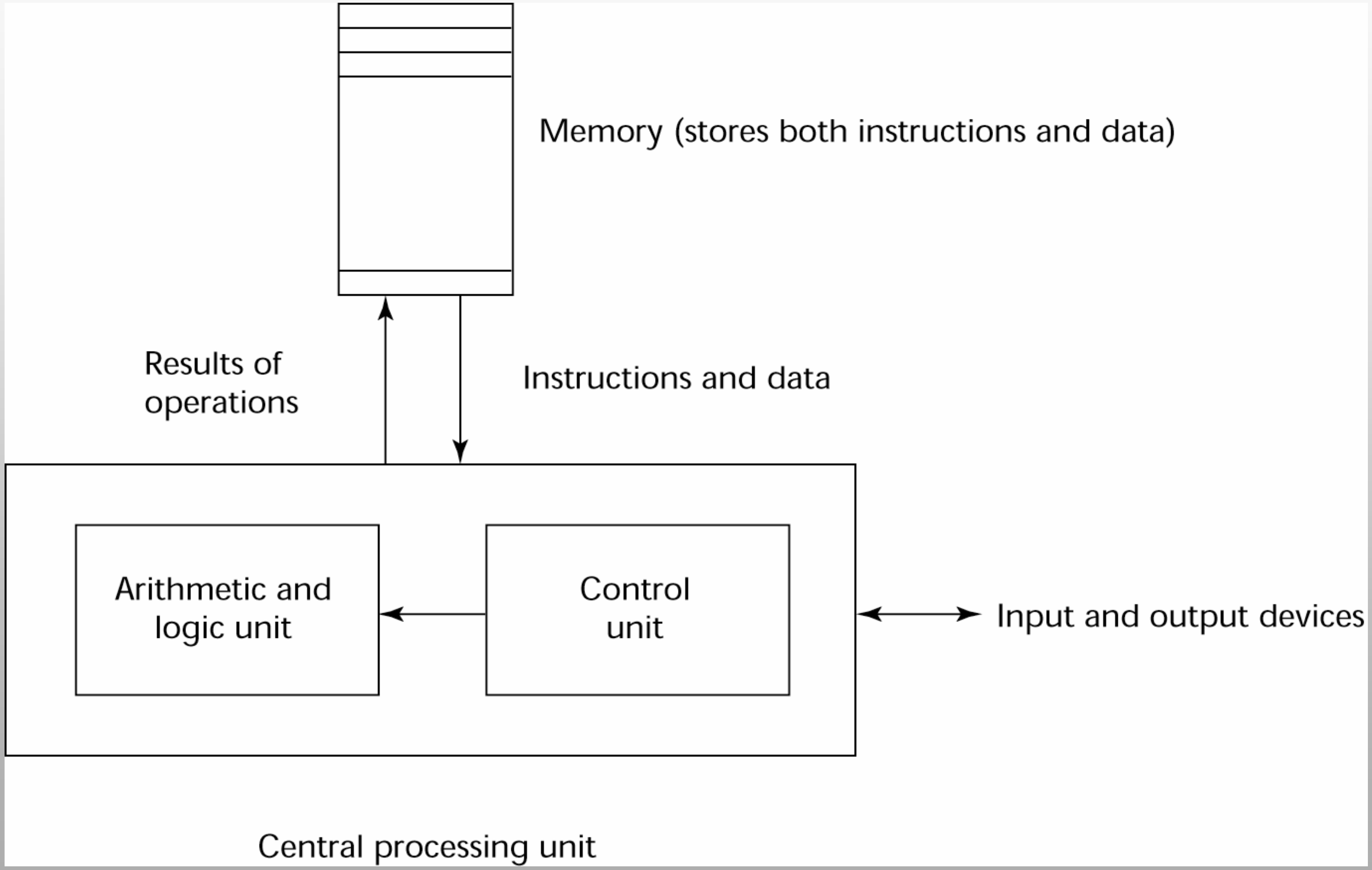
Influences on Language Design

- Computer Architecture
  - Languages are developed around the prevalent computer architecture, known as the *von Neumann* architecture
- Programming Methodologies
  - New software development methodologies (e.g., object-oriented software development) led to new programming paradigms and by extension, new programming languages

Computer Architecture Influence

- Well-known computer architecture: Von Neumann
- Imperative languages, most dominant, because of von Neumann computers
  - Data and programs stored in 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

The von Neumann Architecture



Programming Methodologies Influences

- 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
  - Data abstraction + inheritance + polymorphism

Language Categories

- Imperative
  - Central features are variables, assignment statements, and iteration
  - Examples: C, Pascal
- Functional
  - Main means of making computations is by applying functions to given parameters
  - Examples: LISP, Scheme
- Logic
  - Rule-based (rules are specified in no particular order)
  - Example: Prolog
- Object-oriented
  - Data abstraction, inheritance, late binding
  - Examples: Java, C++
- Markup
  - New; not a programming per se, but used to specify the layout of information in Web documents
  - Examples: XHTML, XML

Language Design Trade-Offs

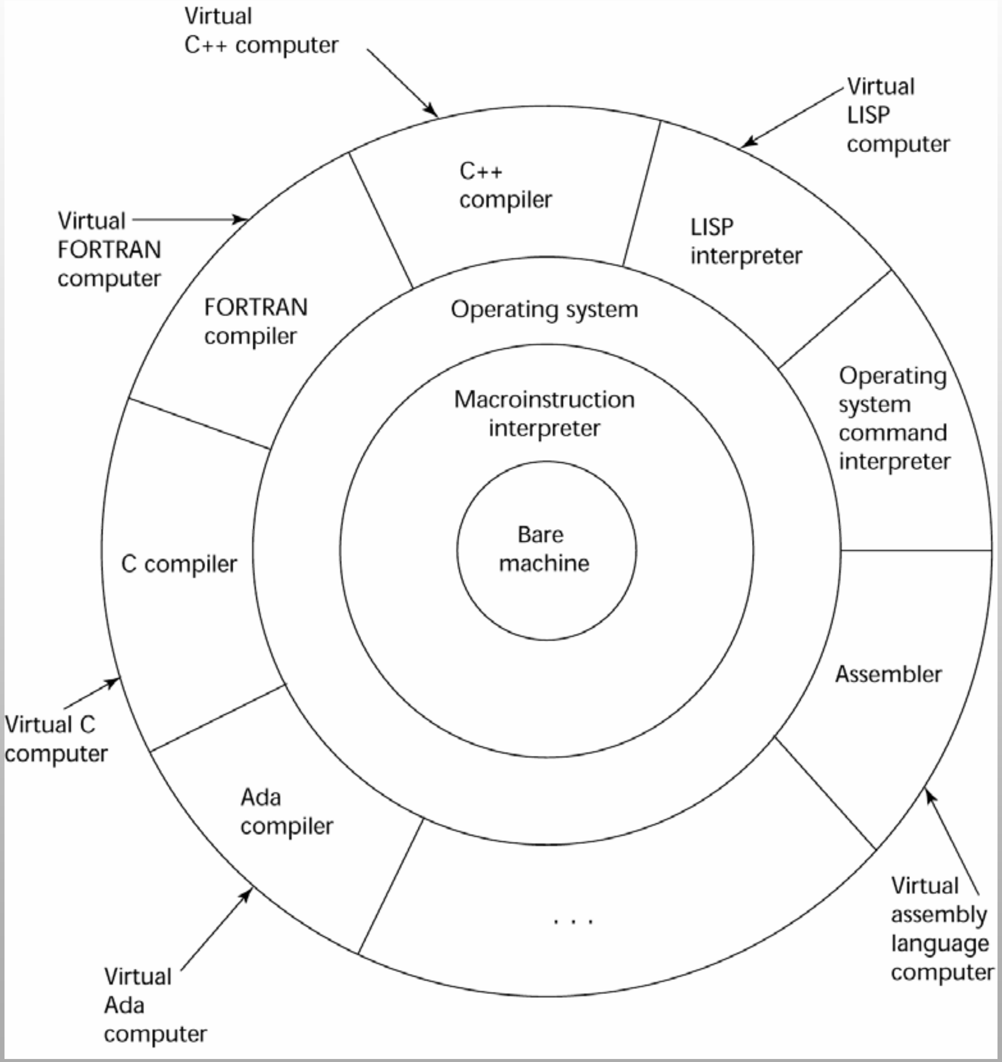
- Reliability vs. cost of execution
  - Conflicting criteria
  - Example: Java demands all references to array elements be checked for proper indexing but that leads to increased execution costs
- Readability vs. writability
  - Another conflicting criteria
  - Example: APL provides many powerful operators (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
  - Another conflicting criteria
  - Example: C++ pointers are powerful and very flexible but not reliably used

Implementation Methods

- Compilation
  - Programs are translated into machine language
- Pure Interpretation
  - Programs are interpreted by another program known as an interpreter
- Hybrid Implementation Systems
  - A compromise between compilers and pure interpreters

Layered View of Computer

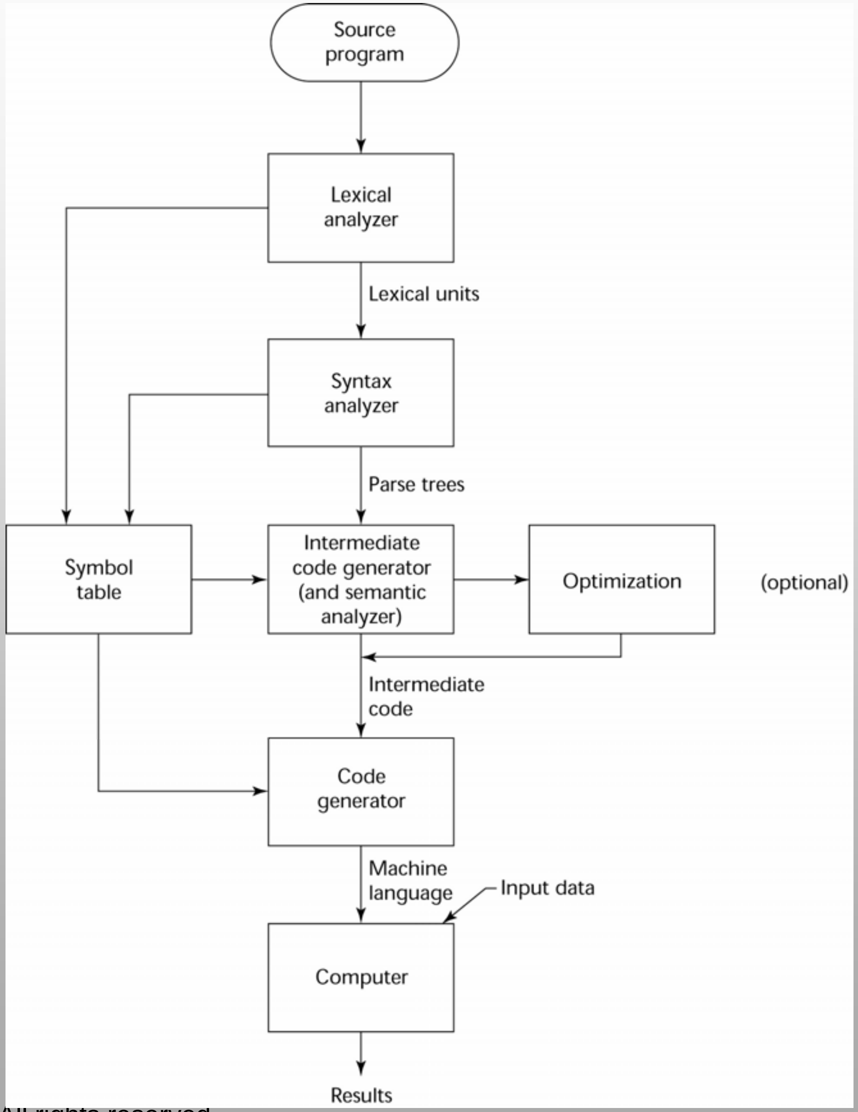
The operating system and language implementation are layered over Machine interface of a computer



Compilation

- Translate high-level program (source language) into machine code (machine language)
- Slow translation, fast execution
- 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 the syntactic structure of program
  - Semantics analysis: generate intermediate code
  - code generation: machine code is generated

The Compilation Process



Additional Compilation Terminologies

- **Load module** (executable image): the user and system code together
- **Linking and loading**: the process of collecting system program and linking them to user program

Execution of Machine Code

- Fetch-execute-cycle (on a von Neumann architecture)

```
initialize the program counter
repeat forever
  fetch the instruction pointed by the counter
  increment the counter
  decode the instruction
  execute the instruction
end repeat
```

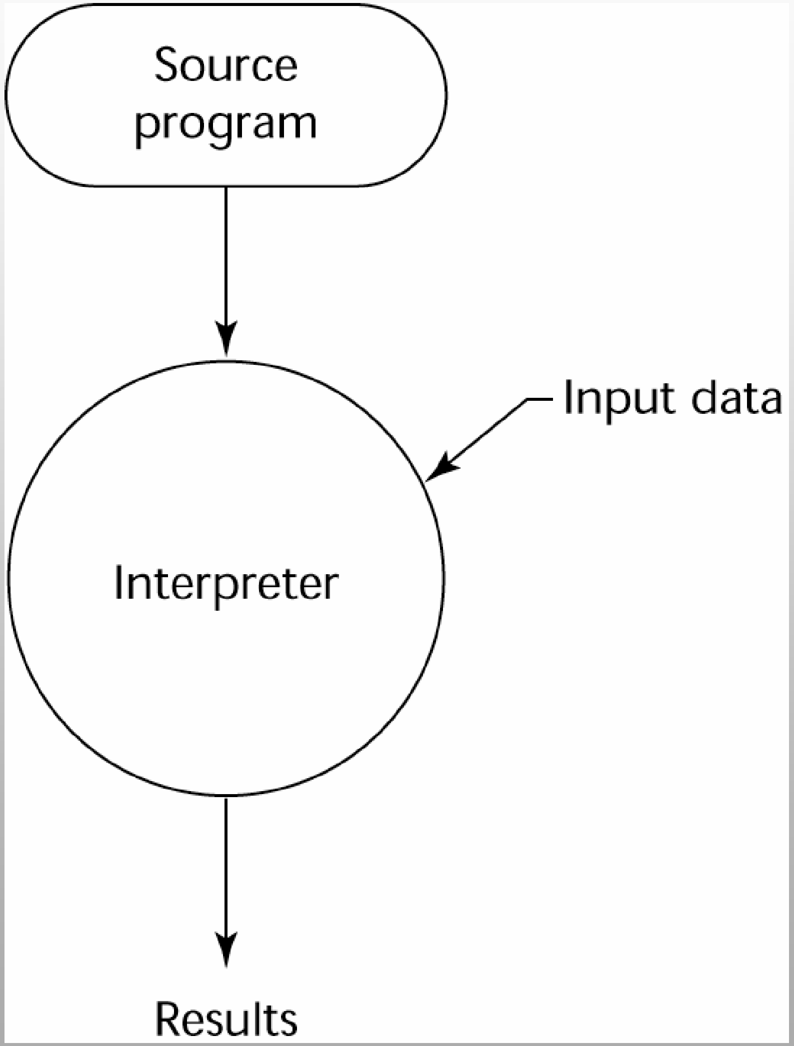
Von Neumann Bottleneck

- Connection speed between a computer's memory and its processor determines the speed of a computer
- Program instructions often can be executed a lot faster than the above connection speed; the connection speed thus results in a *bottleneck*
- Known as von Neumann bottleneck; it is the primary limiting factor in the speed of computers

Pure Interpretation

- No translation
- 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
- Becoming rare on high-level languages
- Significant comeback with some Web scripting languages (e.g., JavaScript)

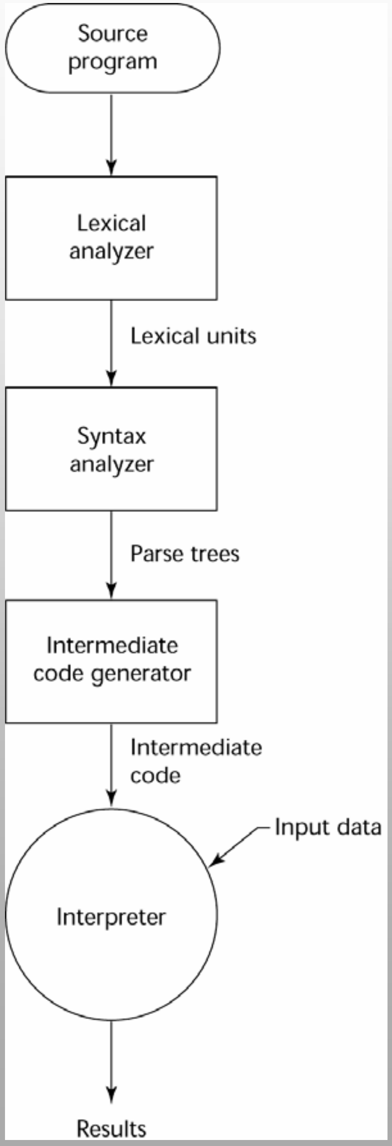
Pure Interpretation Process



Hybrid Implementation Systems

- A compromise between compilers and pure interpreters
- A high-level language program is translated to an intermediate language that allows easy interpretation
- Faster than pure interpretation
- Examples
  - Perl programs are partially compiled to detect errors before interpretation
  - Initial implementations of Java were hybrid; the intermediate form, *byte code*, provides portability to any machine that has a byte code interpreter and a run-time system (together, these are called *Java Virtual Machine*)

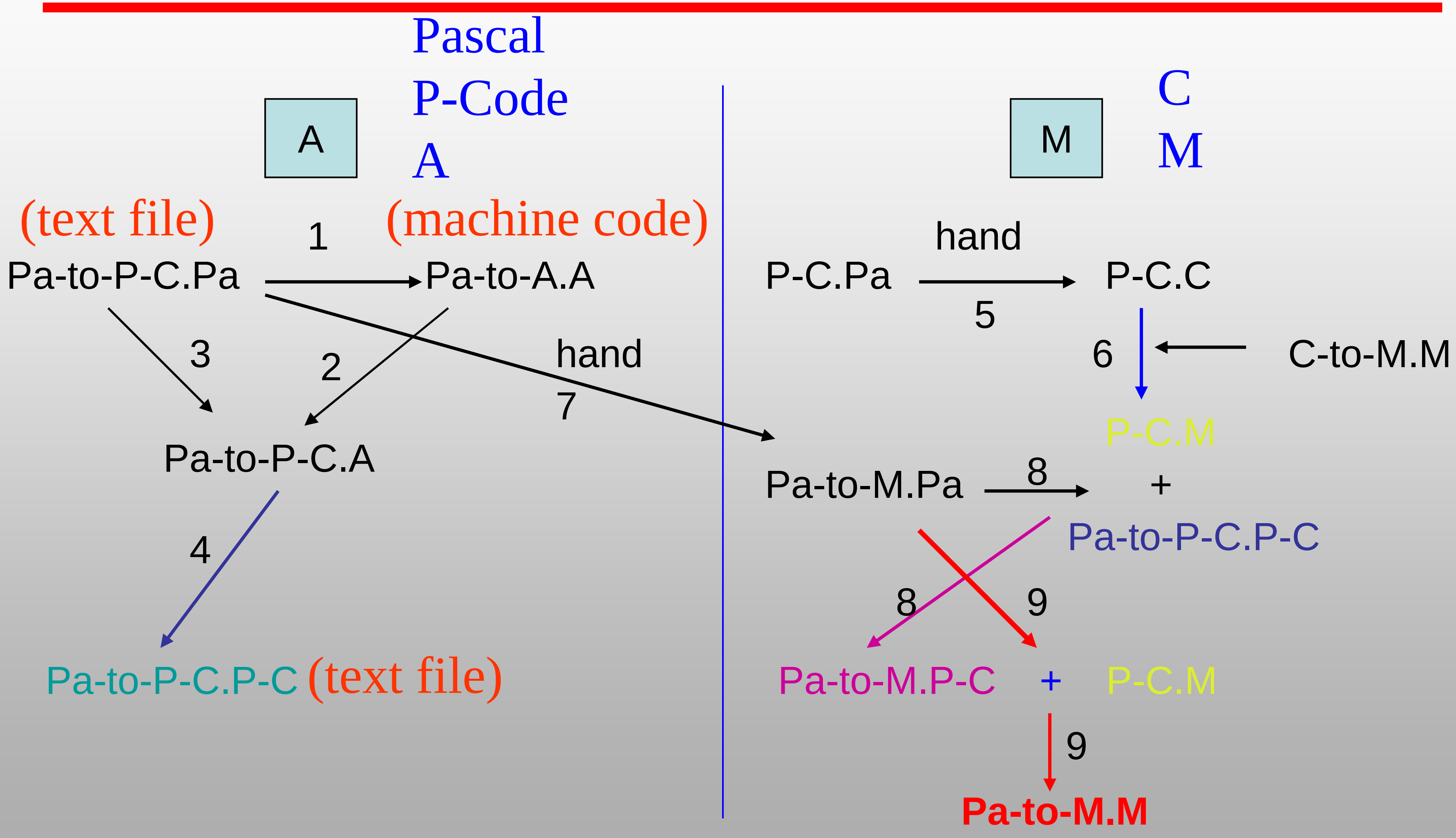
Hybrid Implementation Process



Pascal, P–code & bootstrapping

- Wirth tools (1972) for porting Pascal
  - Pascal compiler PaToP–C.Pa
    - written in Pascal, generating P–code
  - PaToP–C.P–C
    - i.e. PaToP–C.Pa compiled with itself on some computer
  - P–C.Pa: P–code interpreter written in Pascal
- Porting the compiler to machine M (bootstrapping)
  - translate P–C.Pa by hand to a local language, say C
  - compile the result, say P–C.C, obtain an interpreter P–C.M
  - modify (by hand) PaToP–C.Pa to PaToM.Pa
  - compile PaToM.Pa (run PaToP–C.P–C on P–C.M) to PaToM.P–C
  - compile PaToM.Pa (run PaToM.P–C on P–C.M) to PaToM.M

Porting a Pascal Compiler to M



Just-in-Time Implementation Systems

- Initially translate programs to an intermediate language
- Then compile intermediate language into machine code
- Machine code version is kept for subsequent calls
- JIT systems are widely used for Java programs
- .NET languages are implemented with a JIT system

Preprocessors

- Preprocessor macros (instructions) are commonly used to specify that code from another file is to be included
- A preprocessor processes a program immediately before the program is compiled to expand embedded preprocessor macros
- A well-known example: C preprocessor
  - expands `#include`, `#define`, and similar macros

Programming Environments

- The collection of tools used in software development
- UNIX
  - An older operating system and tool collection
  - Nowadays often used through a GUI (e.g., CDE, KDE, or GNOME) that run on top of UNIX
- 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++

Summary

- The study of programming languages is valuable for a number of reasons:
  - Increase our capacity to use different constructs
  - Enable us to choose languages more intelligently
  - Makes learning new languages easier
- Most important criteria for evaluating programming languages include:
  - Readability, writability, reliability, cost
- Major influences on language design have been machine architecture and software development methodologies
- The major methods of implementing programming languages are: compilation, pure interpretation, and hybrid implementation