“Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do.”

— Don Knuth, *Literate Programming*, 1984
A Brief History of Programming Languages

Initial focus: machine-centric programming

• Machine language — the exact bits that control a processor’s operations

• Assembly language — machine language with mnemonics (e.g. sw [store word] instead of 10101111101)

• Assembly language with macro expansion — support for programmer-specified abbreviations of frequent assembly constructs or parameters

Shift to machine independence

• Eliminate the need to rewrite the same program for multiple machines

• “High-level” languages (and their clever acronyms) — Fortran (Formula Translator), Lisp (List Processing), Algol (Algorithmic Language)

• The compiler is born — no more one-to-one correspondence between programmed symbols and machine instructions

• Originally, compiled programs were viewed as a slower alternative to directly-written assembly

• Advances in compiler technology and increased sophistication of software have reversed that
Programming Language Diversity

So many languages, so little time — why isn’t there “one true programming language?”

- **Evolution** — ongoing shifts in programming paradigms: structured programming, object-oriented programming, document-centric languages…

- **Special purposes** — many languages are (initially) designed for a specific domain

- **Personal preference** — individual tastes, aptitudes

Characteristics of “Successful” Languages

- **Expressive power** — though theoretically all equivalent, some languages can do more with less work

- **Learning curve** — or, the opposite of “barrier to entry”

- **Ease of implementation** — elimination of another “barrier to entry” — how easy is it to get a language onto your machine of choice?

- **Compiler quality** — can be extended to “development tool quality”

- **Non-technical factors** — “economics, patronage, inertia”
A Language for Everything

Programming languages
◇ declarative
  • functional: Lisp/Scheme, ML, Haskell
  • data flow: Id, Val
  • logic, constraint-based: Prolog, VisiCalc
◇ imperative
  • Von Neumann: Fortran, Pascal, Basic, C
  • object-oriented: Smalltalk, Eiffel, C++, Java
  • “dynamic” (a.k.a. scripting): Perl, JavaScript

“Other” languages
◇ What is “programming,” and what is not?
◇ document specification
  • “paper-like” documents: HTML, XML, LaTeX
  • graphs and diagrams: ER, UML
◇ information systems
  • querying: SQL, Query-by-Example
  • data/object definition: SQL, ER, UML
◇ focused applications
  • development: make, ant
  • game scripting: QuakeC, UnrealScript
  • multimedia: ActionScript (formerly known as Lingo)
Programming Workflows

• source code ➔ compiler ➔ target program
  input ➔ target program ➔ output

• source code, input ➔ interpreter ➔ output

• source code ➔ translator ➔ intermediate code
  intermediate code, input ➔ virtual machine ➔ output

Compiler Strategies, Variations

• Code reuse: libraries — break up compilation into compilation + linking

• Ease of debugging, machine abstraction — compile to assembly language, not direct to machine code

• Comments, macros, directives — preprocessor prior to compilation, resulting in intermediate source code

• High-level machine abstraction: the “virtual machine” — compile to an intermediate mode that is “interpreted” by the virtual machine
Compilation Overview

- Understanding the compilation process is a key cornerstone to many skills:
  - Error analysis and resolution
  - Mapping corresponding concepts across languages
  - Creating appropriate layers of abstraction

- Two major phases
  - Determine the meaning of a program — front end
  - Construct an equivalent target program — back end