Organization of Programming Languages

Why study programming languages?

It’s part of the very core of computer science

To avoid re-inventing the wheel

To better apply Programming Languages you already know

Understand the underlying design decisions

To be able to effectively communicate

Useful Job Skills

Make educated decisions when choosing a language for a project.

- Java is great for writing applications
- C is great for systems programming

Make better use of language features

- Obscure features
- Cost of features
- Simulate useful features
- Which features is your chosen language missing?
  - Can they be emulated in a library?

Learn new PLs more quickly.

- Evolution => Similarities

What is a programming language?

Donald Knuth-

> Programming is the art of telling another human being what one wants the computer to do

What is a programming language?

A language used to express instructions to a computer.

Used to provide a definition of legal Programs (Syntax)

Used to provide the meaning of a Program (Semantics)

Style of Programming (Pragmatics)

Lots and lots of programming languages

Why is the number of programming languages so large?

- Evolution
- Special Purpose
- Personal Preference

A programming language is a way of thinking

- Different people think in a different way

What makes a “good” programming language?

Expressive power

Ease of use for the novice

Ease of implementation

Excellent compilers

Economics, patronage, and inertia

Programming language classifications

Imperative

- von Neumann languages
- Developed around the computer architecture
  - Data and programs are stored in memory
  - Memory is separate from the CPU
  - Instructions and data are sent from memory to the CPU
  - Fetch-execute-cycle - Von Neumann bottleneck

Object-oriented

- Data abstraction, Inheritance, Polymorphism

Programming language classifications

Functional

- Only function calls
- General lack of variables/storage
- Referential transparency

Logic

- Rule based
- Predicate logic

Primary influences on language design

Computer architecture

- We use imperative languages, at least in part, because we use von Neumann machines

Programming methodologies

- 1950s and early 1960s: Simple applications; worry about machine efficiency
- Late 1960s: People efficiency became important; readability, better control structures
- Late 1970s: Data abstraction
- Middle 1980s: Object-oriented programming

Language Trade-offs

Reliability versus cost of execution

Writability versus readability

Flexibility versus safety

Programming Domains

Scientific applications

- Large numbers of floating point computations; use of arrays

Business applications

- Produce reports, use decimal numbers and characters

Artificial intelligence

- Symbols rather than numbers manipulated; use of linked lists

Systems programming

- Need efficiency because of continuous use

Web Software

- Markup, scripting, general-purpose

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

Readability

Simplicity

- Manageable set of features
- Minimal overloading

Orthogonality

- A relatively small set of primitive constructs can be combined in a relatively small number of ways
- Every possible combination is legal
- Avoid special rules or exceptions

Adequate data types

Syntax

Writability

Simplicity

Orthogonality

Support Abstraction

- Ability to define and use complex structures

Expressiveness

- Ability to adequately express the needs of the program

Reliability

Type checking

Exception handling

Aliasing

- Unrestricted memory referencing can be problematic

Robustness without excess

Languages not supporting a “normal” functionality will require missing features to be implemented through other means

Cost

Training/education

Reusability

Maintenance

Required software

Compilation

Execution

Other

Portability

Documentation

Flexibility

- Can it be used for a wide range of applications

Implementation of languages

Compilation

Interpretation

Hybrid

Compilation

Source code → Compiler → “Executable” file
Input → “Executable” file → Output

Translate high level source code into machine code

- Several steps in the process
  - Lexical analysis
  - Syntax analysis
  - Semantic analysis
  - Code generation
- Issues with von Neumann bottleneck

Slow translation, fast execution

Interpretation

Source code →

Input → Interpreter → Output

No translation prior to execution

Slower execution speed

Improved debugging

Likely to require more memory

Improved portability

Hybrid

Compromise between compilers and interpreters

High level language is “compiled” into an intermediate language

Intermediate language is easily translated into a machine language