Language Evaluation Criteria

- **Readability:**
  - the ease with which programs can be read and understood
  - (as maintenance became the major part of the life cycle)
- **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
  - Minimal feature multiplicity
  - 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
  - Too much orthogonality can also cause problems
- **Data types & control statements**
  - Adequate predefined data types
- **Syntax considerations**
  - Identifier forms: flexible composition
  - Special words and methods of forming compound statements
  - Form and meaning: self-descriptive constructs, meaningful keywords

Functional Lang.

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
  - Strength and number of operators and predefined functions
  - Examples:
    - In C++: `i++;`
    - In Ada: `and then`
Evaluation Criteria: Reliability

- **Type checking**
  - Testing for type errors (as early as possible)
- **Exception handling**
  - Intercept run-time errors and take corrective measures, and then continue
  - E.g. in Ada, C++, Java, etc.
- **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 require the use of "unnatural" approaches, and hence reduced reliability

Evaluation Criteria: Cost

- **Training programmers to use the language**
  - A function of simplicity & orthogonality
- **Writing programs**
  - A function of closeness to particular applications
- **Compiling programs**
  - 1st generation Ada Impediment
- **Executing programs**
  - A function of language design
  - A concern in the design of early PL's
- **Language implementation system**
  - Availability of free compilers; less expensive hardware
  - Reliability
  - Poor reliability leads to high costs
- **Maintaining programs**
  - A function of readability

Evaluation Criteria: Others

- **Portability**
  - The ease with which programs can be moved from one implementation to another
  - A function of degree of "standardization"
- **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: 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
The von Neumann Architecture

• 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

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

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

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
  – Include languages that support object-oriented programming
  – Include scripting languages
  – Include the visual languages
  – Examples: C, Java, Perl, JavaScript, Visual BASIC .NET, C++

• 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

• Markup/programming hybrid
  – Markup languages extended to support some programming
  – Examples: JSTL, XSLT

Language Design Trade-Offs

• Reliability vs. cost of execution
  – Example: Java demands all references to array elements be checked for proper indexing, which leads to increased execution costs

• Readability vs. writability
  – 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
  Simple examples:
  \[X ← 5 \ 8 \ 10 \ 15\]
  \[X + 6\]
  \[X(\text{[5]}←X4?100)\]

• Writability (flexibility) vs. reliability
  – Example: C++ pointers are powerful and very flexible but are unreliable

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
The Compilation Process

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 e.g.: identifiers, keywords, operators, punctuation...
  - 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

Additional Compilation Terminologies

- **Load module** (executable image):
  - the user and system code together
- **Linking and loading**:
  - the process of collecting system program units and linking them to a user program
  - As well as linking other user programs to it

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 much faster than the speed of the connection; the connection speed thus results in a bottleneck
- Known as the **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 be displayed)
- Slow execution
  - (10 to 100 times slower than compiled programs)
- Often requires more space
- Now rare for traditional high-level languages
- Significant comeback with some Web scripting languages (e.g., JavaScript, PHP)

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)
Just-in-Time Implementation Systems

- Initially translate programs to an intermediate language
- Then compile the intermediate language of the subprograms into machine code when they are called
- 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

- A 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 runs on top of UNIX
- Microsoft Visual Studio.NET
  - A large, complex visual environment
  - Used to build Web (and non-Web) applications in any .NET language
- NetBeans
  - An integrated development environment (IDE)
  - Similar to Visual Studio .NET, except for Web applications in Java, …

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