Chapter 1
Preliminaries

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

What impacts Programming Language Design?
- Application domain
- Evaluation Criteria
- Computer architecture
- Programming methodologies

Evaluation Criteria: Writability
- Writability describes the ease with which a language can be used to create programs for a given domain.
  - Be careful not to compare things which should not be.
  - Most of the features that affect readability affects also writability.

Language Evaluation Criteria

<table>
<thead>
<tr>
<th>Characteristics</th>
<th>Readability</th>
<th>Writability</th>
<th>Reliability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simplicity and Orthogonality</td>
<td>★</td>
<td>★</td>
<td>★</td>
</tr>
<tr>
<td>Control structure</td>
<td>★★</td>
<td>★★</td>
<td>★★</td>
</tr>
<tr>
<td>Data type and structures</td>
<td>★★</td>
<td>★★</td>
<td>★★</td>
</tr>
<tr>
<td>Syntax design</td>
<td>★</td>
<td>★</td>
<td>★</td>
</tr>
<tr>
<td>Support for abstraction</td>
<td>★</td>
<td>★</td>
<td>★</td>
</tr>
<tr>
<td>Expressivity</td>
<td>★</td>
<td>★</td>
<td>★</td>
</tr>
<tr>
<td>Type-checking</td>
<td>★</td>
<td>★</td>
<td>★</td>
</tr>
<tr>
<td>Exception handling</td>
<td>★</td>
<td>★</td>
<td>★</td>
</tr>
<tr>
<td>Restricted aliasing</td>
<td>★</td>
<td>★</td>
<td>★</td>
</tr>
</tbody>
</table>

Evaluation Criteria: Writability
- Simplicity and Orthogonality
  - Lack of familiarity with some features leads to misuse and disuse of those features.
  - Misuse could cause bizarre results.
  - Too much orthogonality may produce undetected errors.
- Any combination of primitive is legal.
Chapter 1: Preliminaries

Evaluation Criteria: Writability

Factors

Support for abstraction
- Ability to define and use complicated structures or operations ignoring all the details.
- Important for modular programming.
- Two forms of abstraction
  - Process: subprograms
    - e.g. using a subprogram to implement a search or sort algorithm.
  - Data: data types
    - e.g. trees, arrays, etc.

Expressivity
- Aids writability by make it convenient and easy to specify things.
  - e.g. count++ vs. count = count + 1

Evaluation Criteria: Reliability

Factors

Reliable programs work (according to specifications) under all conditions.
- Type checking
  - Earlier error detection is less expensive to repair
  - Compile-time checking is preferred.
- Exception handling
  - The ability of a program to intercept runtime errors, take corrective measures, and then continue (e.g. C++, Java, Ada).

Aliasing
- Having to or more distinct referencing methods, or names, for the same memory cell.
  - e.g. using pointer in C++, reference in Java

Readability and Writability
- The easiest a program is to write, the more likely it is to be correct.
- Programs that are difficult to read are difficult to both to write and modify.

Evaluation Criteria: Cost

Factors

Cost of learning/teaching a language (programmer training)
Cost of writing/developing a program (software creation)
Cost of compiling the program (fast)
Cost of running the program (fast)
Cost of the compiler (free e.g. Java)
Cost of poor reliability
Cost of maintaining the program (corrections and modifications to add new capabilities)

Evaluation Criteria: Other

Factors

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 a language’s official defining document.
Language Design Trade-Offs

“There are so many important but conflicting criteria, that their reconciliation and satisfaction is a major engineering task.”  
(Tony Hoare 1973)

<table>
<thead>
<tr>
<th>Reliability</th>
<th>vs. Cost (execution)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expressivity</td>
<td>vs. Readability</td>
</tr>
<tr>
<td>Writability</td>
<td>vs. Readability</td>
</tr>
<tr>
<td>Reliability</td>
<td>vs. Writability(flexibility)</td>
</tr>
</tbody>
</table>

Most criteria cannot be defined nor measured precisely. The way a language is evaluated is heavily influenced by the point of view and background of the evaluator.

- Language designer
- Language implementor
- Language user

A real designer understands trade-offs and make decisions rather than skirt them.

What impacts Programming Language Design?

- Application domain
- Evaluation Criteria
- Computer architecture
- Programming methodologies

The programming language should map well to the hardware (computer architecture).

Computer Architecture Influence

- Imperative languages have been designed around the von Neumann architecture
  - Data and programs are stored in memory
  - Central processing unit (CPU) executed the instructions
  - Instructions/data must be transmitted from memory to CPU
  - Results from operations are transmitted back to memory
- Imperative languages map well to this architecture
  - Variables are memory locations
  - Assignments move data back and forth between CPU and memory
  - Iteration for repetition

The von Neumann Architecture

Central processing unit

Input and output devices

Programming languages respond to different ways of thinking.
Programming Methodologies

Influence

People’s needs affect the design of programming languages and paradigms.

- 1950’s and early 1960’s
  - Worry about machine efficiency
  - Simple applications

- Late 1960’s
  - Worry about people efficiency
  - Better control structures and improved readability
    - Structured programming
    - Top-down design and step-wise refinement

Programming Paradigms

Paradigms are programming styles (a special way to express an idea or algorithm) that embody programming design technology.

<table>
<thead>
<tr>
<th>Paradigm</th>
<th>Imperative</th>
<th>Declarative</th>
<th>Functional</th>
<th>Logical</th>
<th>Object-Oriented</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pascal</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Algol</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Smalltalk</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Objects</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>With blocks</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scheme</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lisp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Java</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C++</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Python</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Smalltalk</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Eiffel</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Haskell</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Prolog</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Occam</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C++</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fortran</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Database</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Functional</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Logical</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Object-Oriented</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Imperative</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Declarative</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Functional</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Logical</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Object-Oriented</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Programming Paradigms: Imperative

Central features are variables, assignment statements, and iterative form of repetition.

- Specific order of execution of the instruction
  - Program = order series of steps
  - Separation of data and algorithm
  - C, Pascal, Cobol, Fortran

Programming Paradigms: Functional

Central features are functions (applied to given parameters)

- Program = a set of mathematical functions each with an input (domain) and an output (range)
- No assignments, tons of recursion, and less focus on order
- Lazy evaluation: postpone operand evaluation until operation.
- Lisp, Scheme, Haskell, ML
Programming Paradigms: Logic

- What vs. How
- Rule-based language
- Rules are specified in no particular order
- Program = collection of logical declarations that describe the problem to be solved
  - An inference engine then finds the solution
- It is also called declarative
  - Declare or make assertions
  - No sequence
- Prolog

Example

Programming Example

Greatest Common Denominator (gcd)

- C
  ```c
  (define (gcd u v)
    (if (= v 0) u
      (gcd v (modulo u v))))
  ```

- Scheme
  ```scheme
  gcd ( U, V, U ) :- V=0
  gcd ( U, V, X ) :- not( V=0 )
  Y is U mod V,
  gcd ( V, Y, X )
  ```

- Prolog
  ```prolog
  public class IntGcd
  {
    private int value;
    public IntGcd ( int val ) {
      value = val; }
    public int GetValue() {
      return value; }
    public int gcd ( int v ) {
      int z = value;
      int y = v;
      while ( y != 0 ) {
        int t = y;
        y = z%y;
        z = t; }
      return z; }
  }
  ```

- Java
  ```java
  Compilation
 
  Translation high-level program to machine code
  - Slow translation
  - Fast execution
  - Optimization (improve program by making it smaller or faster)
  - Slow for development
  - Difficult dealing with runtime errors

  Interpretation

  - No translation
  - Easier implementation
  - Slower execution
  - Often requires more space
  - Easy run-time error handling
  - Becoming rare on high-level languages
  - Significant comeback with some Web scripting languages (e.g. JavaScript)
Hybrid

- A compromise between compilers and pure interpreters
- Faster than pure interpretation (medium execution speed)
- A high-level language program is translated to an intermediate language that allows easy interpretation (small translation cost)

Language Implementation: Comparison

<table>
<thead>
<tr>
<th></th>
<th>Compiler</th>
<th>Interpreter</th>
<th>Hybrid</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speed (runtime)</td>
<td>simple instructions</td>
<td>complex instructions</td>
<td></td>
</tr>
<tr>
<td>Memory needed</td>
<td>simple</td>
<td>reusable backend</td>
<td>less</td>
</tr>
<tr>
<td>Reliability</td>
<td>intermediate language</td>
<td>reusable backend</td>
<td>less</td>
</tr>
<tr>
<td>Memory needed</td>
<td>simple</td>
<td>simple</td>
<td>less</td>
</tr>
</tbody>
</table>

Summary

- Reasons to study concepts of PLs
  - Increase our capacity to use different constructs
  - Enables us to choose languages more intelligently
  - Makes learning new languages easier
- Most important criteria for evaluating PLs
  - Readability, writability, reliability, and cost
- Major influences on language design
  - Machine architecture and software development methodologies
- Major methods of implementing languages
  - Compilation, pure interpretation, and hybrid implementation