Programming Languages: Introduction

What Is a Programming Language?

- A language (formal notational system)
  - For describing **computations** so that they can be executed on a computer (machine)
    - **Human-readable**
    - **Machine-readable**

Describing Solvable Computations

- A programming language must be **universal**.
  - **Any problem if it can be solved at all by a computer, must have a solution that can be programmed (expressed) in the language.**

Universal – All Solvable Computations

- A programming language is **universal** if
  - integer values and arithmetics
  - variables
  - assignment statement
  - selection statement
  - loop statement/go to statement

Not-computable Computations (Problems)?

- Is there a not-computable problem at all?
  - Yes
  - The halting problem

Human Readability

- A programming language must be **human-readable**
- How easy?
  - Readability
  - Writability
- How?
  - Abstractions
    - Data abstraction
    - Control abstraction

Machine Readability

- A programming language must be **implementable** on a computer.
  - Every well-formed program in the language must be **executable** on a computer.
- How efficient?
  - Time
  - Space

Not-executable?

- Is there an unimplementable language?
  - Yes
  - Specification languages

Two Aspects of Programming Languages

- **Syntax**
  - Structure or form
- **Semantics**
  - Meaning

How to Evaluate Programming Languages?

- Readability
- Writability
- Reliability
- Cost
- Others:
  - Portability, generality, well-definedness

Readability

- The ease with which programs can be read and understood
- Factors:
  - Overall simplicity
    - Too many features is bad.
    - Multiplicity of features is bad.
    - Operator overloading

Readability

  - **Orthogonality:** *A relatively small set of primitive constructs can be combined in a relatively small number of ways to build the control and data structures of the language.*
    - Lack of orthogonality leads to rule exceptions.
    - Makes the language easy to learn and read.
    - Too much orthogonality?
  - Sufficient control statements – No go-to's
  - Sufficient data types and structures

Writability

- A measure of how easily a language can be used to create programs for a chosen problem domain.
- Factors:
  - Simplicity and orthogonality
  - Support for abstraction – process and data abstraction
  - Expressivity

Reliability

- A program is reliable if it performs to its specifications under all conditions.
- Factors:
  - Type checking
  - Exception handling
  - Aliasing
  - Readability and writability

Cost

- Cost for
  - Programmer training
  - Software creation
  - Compilation
  - Execution
  - Compiler cost
  - Maintenance

Other Criteria

- 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 defining document

How to Implement Programming Languages?

- Compilation
- Pure interpretation
- Hybrid – compilation + interpretation

Compilation

- Translated into machine code by a program called a compiler.
- And then executed directly on the computer.
  - **Slow translation**
  - **Fast execution**

Phases in Compilation

- In a typical compiler, compilation proceeds through a series of well-defined phases.
- Each phase discovers information or transforms the program into a form of use to later phases.

Phases in Compilation

- Scanner (Lexical analysis)
- Parser (Syntax analysis)
- Semantic analysis
- Intermediate code generation
- Machine-independent code improvement (optimization)
- Target code generation
- Machine-specific code improvement (optimization)

Recognizing the Structure of the Program

- Scanner (Lexical Analysis)
  - A stream of characters
  - A stream of tokens
- Parser (Syntax Analysis)
  - A stream of tokens
  - A parse tree

Discovering the Meaning of the Program

- Semantic Analysis
  - A parse tree (concrete syntax tree)
  - An abstract syntax tree (syntax tree)
- Builds and maintains a symbol table.
  - Symbol table serves as a repository for information about identifiers throughout compilation.
- Intermediate Code Generation
  - An abstract syntax tree (syntax tree)
  - An intermediate code (form)

Translating Into Target Code

- Semantic-preserving translation
- Target Code Generation
  - A modified intermediate code (form)
  - A target (assembly or machine) language

Improving (Optimizing) Code

- Machine-independent code improvement
  - An intermediate code (form)
  - A modified intermediate code (form)
- Machine-dependent code improvement
  - A target (assembly or machine) language
  - A modified target language
- Optional!

The Compilation Process

- The compilation process
  - **See fig 1.3**

Pure Interpretation

- Executed directly by a program called an interpreter.
  - No translation
  - Slow execution
  - More space
  - Source-level debugging

The Pure Interpretation Process

- The pure interpretation process
  - **See Fig. 1.4**

Compilation + Interpretation

- Compiled first and then interpreted
  - Hybrid implementation
  - **Small translation cost**
  - **Medium execution speed**

The Compilation + Interpretation Process

- The compilation & interpretation process
  - **See Fig. 1.5**

Programming Language Design

- Primary influences on programming language design:
  - Computer architecture
    - Machine efficiency
  - Programming methodologies
    - Human efficiency

Computer Architecture

- The basic architecture of computers
  - The von Neumann machines
  - A sequential machine
  - **See fig 1.1**
  - Imperative programming languages – based on variables and assignments

Programming Methodologies/Paradigms

- Imperative programming
- Functional programming
- Object-oriented programming
- Logic programming
- Concurrent programming
- …

Categories of Programming Languages

- Imperative languages
  - Procedure-oriented
- Functional languages
  - Function-oriented
- Logic languages
  - Rule-based
- Object-oriented languages
  - Closely related to imperative
- Scripting languages
  - Glue
- Domain-specific languages
  - Special-purpose

Programming Languages vs. Programming Paradigms

- Imperative languages for Imperative Programming
- Functional languages for Functional Programming
- Object-oriented languages for Object-oriented programming
- Logic languages for Logic Programming

High-level Programming Languages

- How many languages?
  - ?

Major High-level Programming Languages So Far

- **FORTRAN**
- **LISP**
- **ALGOL**
- **COBOL**
- **BASIC**
- **PL/I**
- **APL**
- **SNOBOL**
- **SIMULA**
- **PASCAL**
- **C**
- **SCHEME**
- **MODULA**
- **PROLOG**
- **ADA**
- **SMALLTALK**
- **ICON**
- **COMMON LISP**
- **C++**
- **ML/ SML**
- **Quick BASIC**
- **Visual BASIC**
- **HASKELL**
- **Eiffel**
- **JAVA**
- **JavaScript**
- **C#**
- **Perl**
- **PhP**
- **Phyton**
- **Ruby**
- **…**

The First High-level Language: FORTRAN

- The first high-level programming language
  - **FORTRAN** (**FOR**mula **TRAN**slator) I
  - FORTRAN II
  - FORTRAN 77
  - FORTRAN 90
  - For scientific applications
- First implemented version of FORTRAN
  - Names could have up to six characters
  - User-defined subprograms
  - Three-way selection statement (arithmetic IF)

The First Functional Language: LISP

- The first functional language
- **LISP** (**LIS**t **P**rocessing language) – 1959
  - For list processing and AI applications
- Pioneered functional programming
  - No need for variables or assignment
  - Control via recursion and conditional expressions

Functional Languages: Descendants of LISP

- **Scheme**
- **COMMON LISP**
- **ML** (**M**eta**L**anguage)
- **SML** (**S**tandard **ML**)
- **Miranda**
- **Haskell**

The First Step Toward Sophistication: ALGOL

- **ALGOL** (**ALGO**rithmic **L**anguage) **58** - 1958
- Language Features:
  - Concept of types was formalized
  - Names could have any length
  - Arrays could have any number of subscripts
  - Parameters were separated by mode (in & out)
  - Subscripts were placed in brackets
  - Compound statements (begin ... end)
  - Semicolon as a statement separator
  - Assignment operator was :=
  - if had an else-if clause

ALGOL60

- New Features:
  - **Block structure (local scope)**
  - **Two parameter passing methods: pass by value & pass by name**
  - **Subprogram recursion**
  - **Stack-dynamic arrays**
  - First language whose syntax was formally defined (using BNF).
- **All subsequent imperative languages are based on it.**
  - **"Algol-like" programming languages**

The First Language For Business Application: COBOL

- **COBOL** (**CO**mmon **B**usiness **O**riented **L**anguage) - 1960
  - Designed for business applications.
- Contributions:
  - First macro facility in a high-level language
  - Hierarchical data structures (records)
  - Nested selection statements
  - Long names (up to 30 characters), with hyphens
  - Data Division
- Still the most widely used business applications language

The Beginning of Timesharing: BASIC

- **BASIC** (**B**eginner's **A**ll-purpose **S**ymbolic **I**nstruction **C**ode) - 1964
- For:
  - Easy to learn and use for non-science students
  - Extremely simple syntax and semantics
- Current popular dialects:
  - **QuickBASIC**
  - **Visual BASIC**

The First Language For "Everything For Everybody": PL/I

- **PL/I** (**P**rogramming **L**anguage/**I**)- 1965
- Contributions:
  - First unit-level **concurrency**
  - First **exception handling**
  - Switch-selectable recursion
  - First **pointer** data type

The Beginnings of Data Abstraction: SIMULA67

- **SIMULA** 67 - 1967
  - For system simulation
  - Based on ALGOL 60
- Contributions:
  - **Coroutines** - a kind of subprogram
  - Implemented in a structure called a **class**
  - Classes are the basis for data abstraction
  - Classes are structures that include both local data and functionality

Orthogonal Design: ALGOL68

- **ALGOL** 68 - 1968
  - Based on the concept of **orthogonality**
- Contributions:
  - **User-defined data structures**
  - Reference types
  - **Dynamic arrays**
- Comments:
  - Had even less usage than ALGOL 60.
  - Had strong influence on subsequent languages, especially Pascal, C, and Ada.

Simplicity by Design: PASCAL

- **Pascal** - 1971
  - For teaching **structured programming**
  - Small, simple, nothing really new
  - Most widely used language for teaching programming in colleges

A Portable System Language: C

- **C** - 1972
  - For **systems programming**
  - Evolved primarily from B, but also ALGOL 68
  - Powerful set of operators, but **poor type checking**.
  - Initially spread through UNIX.

The First Language Based on Logic: PROLOG

- **PROLOG** (**PRO**gramming in **LOG**ic)- 1972
  - Based on formal logic
  - Non-procedural
  - Being an intelligent database system that uses an inferencing process to infer the truth of given queries

History's Largest Design Effort: ADA

- **Ada** - 1983
  - Huge design effort, involving hundreds of people, much money, and about eight years
- Contributions:
  - **Packages** - support for data abstraction
  - Exception handling - elaborate
  - **Generic** program units
  - Concurrency - through the tasking model
- Included all that was then known about software engineering and language design.

Object-Oriented Language: SMALLTALK

- **Smalltalk** – 1980
- First full implementation of an object-oriented language
  - data abstraction, inheritance, and dynamic type binding
- Pioneered the **graphical user interface** everyone now uses

Combining Imperative and OO Features: C++

- Developed at Bell Labs by Stroustrup in 1985
- Facilities for object-oriented programming, taken partially from SIMULA 67, were added to C.
- A large and complex language
- Rapidly grew in popularity, along with OOP.
- ANSI standard approved in November, 1997.

Programming the WWW: JAVA

- Programming the World Wide Web.
- Developed at Sun in the early 1990s.
- Based on C++
  - Significantly simplified.
  - Supports only OOP.
  - Has references, but not pointers.
  - Includes support for applets and a form of concurrency.

Programming Future: Library & Scripting

- Growing importance on libraries
  - Interface with OS and Hardware
  - A rich library
  - Integrated with the programming languages
- Scripting languages
  - Ties utilities together

Summary: High-level Programming Languages

- A big picture:
- **See Fig. 2.1**!

Why Study Concepts of PLs?

- To increase capacity to express programming concepts.
- To improve background for choosing appropriate languages.
- To increase ability to learn new languages.
- To understand the significance of implementation.
- To increase ability to design new languages.

In This Course

- Foundations (syntax + semantics)
- Fundamental Concepts
- Implementations
- Paradigms
- Programming
  - **SML**
  - **Prolog**

QUIZ

- **A PL?**
- **Conditions for a PL?**
- **A universal min PL?**
- **How implement a PL?**
- **Evaluate PLs?**
- **Paradigms vs. PLs?**
- **Major PLs?**