Contents

1.1 Principles
1.2 Paradigms
1.3 Special Topics
1.4 A Brief History
1.5 On Language Design
   1.5.1 Design Constraints
   1.5.2 Outcomes and Goals
1.6 Compilers and Virtual Machines
1.1 Principles

Programming languages have four properties:

- Syntax
- Names
- Types
- Semantics

For any language:

- Its designers must define these properties
- Its programmers must master these properties
Syntax

The *syntax* of a programming language is a precise description of all its grammatically correct programs. When studying syntax, we ask questions like:

- *What is the grammar for the language?*
- *What is the basic vocabulary?*
- *How are syntax errors detected?*
Names

Various kinds of entities in a program have names:

variables, types, functions, parameters, classes, objects, ...

Named entities are bound in a running program to:

– Scope
– Visibility
– Type
– Lifetime
Types

A *type* is a collection of values and a collection of operations on those values.

- **Simple types**
  - *numbers, characters, booleans, ...*

- **Structured types**
  - *Strings, lists, trees, hash tables, ...*

- **A language’s type system** can help to:
  - *Determine legal operations*
  - *Detect type errors*
  - *Optimize certain operations*
Semantics

The meaning of a program is called its *semantics*. In studying semantics, we ask questions like:

– *When a program is running, what happens to the values of the variables?*

– *What does each statement mean?*

– *What underlying model governs run-time behavior, such as function call?*

– *How are objects allocated to memory at run-time?*
1.2 Paradigms

A programming paradigm is a pattern of problem-solving thought that underlies a particular genre of programs and languages.

There are several main programming paradigms:

- **Imperative**
- **Object-oriented**
- **Functional**
- **Logic**
- **Dataflow**

Focus of this course
Imperative Paradigm

Follows the classic von Neumann-Eckert model:

- *Program and data are indistinguishable in memory*
- *Program = sequence of commands modifying current state*
- *State = values of all variables when program runs*
- *Large programs use procedural abstraction*

Example imperative languages:

- *Cobol, Fortran, C, Ada, Perl, ...*
The von Neumann-Eckert Model

Figure 1.1: The von Neumann-Eckert Computer Model
Object-oriented (OO) Paradigm

An OO Program is a collection of objects that interact by passing messages that transform the state.

When studying OO, we learn about:

– Encapsulated State
– Sending Messages
– Inheritance
– Subtype Polymorphism

Example OO languages:

Smalltalk, Java, C++, C#, and Python
Functional Paradigm

Functional programming models a computation as a collection of mathematical functions.

- *Input = domain*
- *Output = range*

Functional languages are characterized by:

- *Functional composition*
- *Recursion*

Example functional languages:

- *Lisp, Scheme, ML, Haskell, OCaml,...*
Functional Paradigm

Functional programming models a computation as a collection of mathematical functions.

- Input = domain
- Output = range

Notable features of modern functional languages:

- Functions as values
- Symbolic data types
- Pattern matching
- Sophisticated type system and module system
Logic Paradigm

Logic programming declares what outcome the program should accomplish, rather than how it should be accomplished.

When studying logic programming we see:

- Programs as sets of constraints on a problem
- Programs that achieve all possible solutions
- Programs that are nondeterministic

Example logic programming languages:

- Prolog
1.3 Special Topics

- Event handling
  - E.g., GUIs, home security systems

- Concurrency
  - E.g., Client-server programs

- Correctness
  - How can we prove that a program does what it is supposed to do under all circumstances?
  - Why is this important?
1.4 A Brief History

How and when did programming languages evolve?
What communities have developed and used them?

– *Artificial Intelligence*
– *Computer Science Education*
– *Science and Engineering*
– *Information Systems*
– *Systems and Networks*
– *World Wide Web*
Figure 1.2: A Snapshot of Programming Language History
1.5 On Language Design

Design Constraints

- Computer architecture
- Technical setting
- Standards
- Legacy systems

Design Outcomes and Goals
What makes a successful language?

Key characteristics:

– *Simplicity and readability*
– *Clarity about binding*
– *Reliability*
– *Support*
– *Abstraction*
– *Orthogonality*
– *Efficient implementation*
Simplicity and Readability

• Small instruction set
  – E.g., Java vs Scheme

• Simple syntax
  – E.g., C/C++/Java vs Python

• Benefits:
  – Ease of learning
  – Ease of programming
Clarity about Binding

A language element is bound to a property at the time that property is defined for it.

So a binding is the association between an object and a property of that object

- *Examples:*
  - a variable and its type
  - a variable and its value
- *Early binding* takes place at compile-time
- *Late binding* takes place at run time
Reliability

A language is *reliable* if:

- *Program behavior is the same on different platforms*
  - E.g., early versions of Fortran
- *Type errors are detected*
  - E.g., C vs Haskell
- *Semantic errors are properly trapped*
  - E.g., C vs C++
- *Memory leaks are prevented*
  - E.g., C vs Java
Language Support

- Accessible (public domain) compilers/interpreters
- Good texts and tutorials
- Wide community of users
- Integrated with development environments (IDEs)
Abstraction in Programming

- **Data**
  - *Programmer-defined types/classes*
  - *Class libraries*

- **Procedural**
  - *Programmer-defined functions*
  - *Standard function libraries*
Orthogonality

A language is *orthogonal* if its features are built upon a small, mutually independent set of primitive operations.

- Fewer exceptional rules = conceptual simplicity
  - *E.g.*, restricting types of arguments to a function
- Tradeoffs with efficiency
Efficient implementation

• Embedded systems
  – Real-time responsiveness (e.g., navigation)
  – Failures of early Ada implementations

• Web applications
  – Responsiveness to users (e.g., Google search)

• Corporate database applications
  – Efficient search and updating

• AI applications
  – Modeling human behaviors
1.6 Compilers and Virtual Machines

Compiler – produces machine code
Interpreter – executes instructions on a virtual machine

• Example compiled languages:
  – *Fortran, Cobol, C, C++*

• Example interpreted languages:
  – *Scheme, Haskell, Python*

• Hybrid compilation/interpretation
  – *The Java Virtual Machine (JVM)*
  – *OCaml*
The Compiling Process

Figure 1.4: The Compile-and-Run Process

Diagram showing the process:
- Source Program
  - Lexical Analyzer
  - Syntactic Analyzer
  - Type Checker
  - Code Optimizer
  - Code Generator
  - Machine Code
  - Input
  - Computer
  - Output
The Interpreting Process

Figure 1.5: Virtual Machines and Interpreters
Discussion Questions

1. Comment on the following quotation:

   *It is practically impossible to teach good programming to students that have had a prior exposure to BASIC; as potential programmers they are mentally mutilated beyond hope of regeneration.* – E. Dijkstra

2. Give an example statement in your favorite language that is particularly unreadable. E.g., what does the C expression `while (*p++ = *q++)` mean?