

CS 345: Programming Languages

Course Objectives

After completing this course you should be able to demonstrate your understanding of a number of important programming-language concepts by

- predicting the results of programs that use them
- developing small programs or program fragments that make essential use of them
- explaining how they are typically implemented

This course...

- *is not* a “grand tour” of programming languages.
- *is not* intended to provide expertise in any specific language
- *is* a survey of programming-language *concepts*, using portions of numerous programming languages to illustrate them.

The Goal:

To help you become a sophisticated “consumer” of programming languages, able to understand their strengths and limitations and why they have them.

E.W. Dijkstra: “A Programming Language is a tool that has profound influence on our thinking habits.”

The Organizing Theme: Problems and Solutions

- Each programming-language concept and feature is intended as a solution of some computation-specification problem.
- We try to identify the purpose behind the concepts and features that we examine.

What is a programming language?

- a formal notation for specifying computations
- “notation” vs. “language”

Two views of what programs are for:

- original view: The purpose of software is to *control computers*
- modern view: The purpose of software is to *specify computations* (and the purpose of computers is to execute the software). A program’s most important audience is not machines but people.

Why are there so many programming languages?

- many different machines (old view)
- many different applications
- need for better tools (yielding software that’s more trustworthy, faster, and less expensive)

How should programming languages be judged?

Some criteria:

0. extent of support for program correctness
1. cost of program development and modification
2. cost of program use (time, space)

Total cost: always some combination of 1 and 2.
⇒ there’s a trade-off: cost vs. speed

Early programming languages

0. machine languages (1940’s) and assembly languages (1950’s)
 - machine-specific (nonportable)
 - verbose
 - modularity: zero
1. FORTRAN, COBOL, ALGOL (1960’s)—the first “high-level” languages— brought improved
 - portability (reusability)
 - redundancy (⇒ consistency checks)
 - conciseness, readability
 - modularity (subroutines, procedures)

The transition 0→1 brought about

- **small increase** in the cost of **use** as software became slightly bigger and slower
- **large reduction** in the cost of **development** (reportedly a factor of 10) as software became much cheaper, more portable, and more reliable.

For most purposes, level-1 (“higher-level”) languages have replaced level-0 languages.

The quest for another similar increase in *programmer productivity* continues to this day— so far, with no clear success (although industrial use of Erlang is delivering 4× improvements over C++/Java).

Programming paradigms

Most programming languages belong to one of two distinct paradigms:

- **Imperative**: focus on machine, execution via *state changes* (bottom-up, original-view)
Central operation: **assignment**
Examples: FORTRAN, COBOL, ALGOL-60
Variations and extensions:
 - ★ **structured control**: Pascal, C
 - ★ **concurrency** : Ada
 - ★ **encapsulation** (ADTs): Modula-2, C++
 - ★ **object-orientation**: C++, Smalltalk, Java

- **Declarative**: focus on computation in the abstract (top-down, modern-view)

Functional: execution via *evaluation*

Central operation: **function application**

Variations:

- ★ “eager” evaluation (impure):
ML, Scheme, Erlang
- ★ “lazy” evaluation (pure):
Miranda, Haskell

Logic: execution via *deduction*

Central operation: **unification**

Example: Prolog

A transportation analogy:

assembler	spelunking
C	hiking
C++	driving a car
Java	riding the bus
Haskell	flying

Each has its uses— there are places you can't get to except by spelunking, etc.

This course spends a lot of time on Haskell— why?

- Haskell is our representative of the functional paradigm.
- Most students are already familiar with the imperative paradigm
- To understand a new paradigm, you can't just learn a few facts— you must learn enough of a language to write real programs in it.
- In many ways —especially its type system— Haskell is a state-of-the-art language regardless of paradigm; some of its features are beginning to be copied in more conventional languages.

According to The Pragmatic Programmers' Language of the Year Project (2002)

<http://www.pragmaticprogrammer.com/loty>,

"This year's language is Haskell. We've chosen Haskell because it's a popular functional programming language, and we're interested in learning more about the functional programming paradigm."

Programming's biggest problem:

Unmastered Complexity

The sheer scale of digital computations—

- megabits of information
- millions of operations per second

— requires us to focus our small brains on **one part at a time**.

A programming language helps if ...

- ☛ it is as *small* and *simple* as possible (but no simpler)
 - it suppresses unnecessary detail
 - it consists of a few parts that fit together in simple ways
- ☛ it supports *modularization*
 - programs are readily built up from separate subprograms (modules)
 - modules' interfaces are clear and simple
- ☛ it supports *sharing* and *reuse* of program parts

Modularity and Structure

Structure: the relationship between the *whole* and the *parts*.

A time-honored technique for solving complex problems:

divide and conquer:

0. *divide* the problem into (easier) subproblems
1. solve each subproblem *separately*
2. *combine* the separate solutions (modules)

The source of divide-and-conquer's power is

recursivity

i.e., the same technique can be applied to each subproblem.

Among the most significant advances in programming languages are discoveries of new ways...

- to **decompose** (divide) problems
- to **combine** solutions

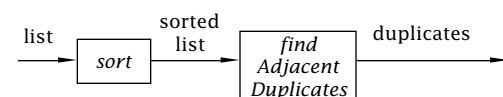
Problem decompositions are seldom unique.

Example problem:

To output the elements of a list that occur more than once — i.e., the list's duplicates.

1. **functional decomposition**

- a. sort the list
- b. in the sorted list, find *adjacent* duplicates



duplicates = *findAdjacentDuplicates* (*sort* (*list*))

Advantages: *sorting* and *finding adjacent duplicates*

- can be programmed independently
- can be executed concurrently (pipeline)