Introduction

Contents

Programming languages and the process of programming

Points

- Programming means more than coding
- Why study programming languages?
- Programming language paradigms and applications

Programming means much more than coding in a programming language.

- Before coding begins, you analyze the problem, design (or borrow) an algorithm, analyze the cost of the solution.
- After all coding has been done, you maintain the program.

Programming languages are used to instruct computers.

What do we communicate to computers?

How do computers get back to us?

How do programming languages differ from natural languages? Would talking to computers in English be preferable?

What makes a programmer good?

Why should a good programmer know more than one programming language?

Why should we study programming languages?

- to understand better the connection between algorithms and programs;
- to be able to look for general, language-independent solutions;
- to have a choice of programming tools that best match the task at hand—for example, use more than one language and employ the full expressive power of each language;
- to appreciate the workings of a computer equipped in a programming language—by knowing how languages are implemented;
- to learn new languages easily and to know how to design new languages (for example, data formats or user interfaces!);
- to see how a language may influence the discipline of computing and strengthen good software engineering practice.

There are many classes of programming languages and programming language paradigms.

The same computation can be expressed in various languages, to be run on the same computers.

Every language supports a slightly or dramatically different style of problem solving.

Classification of programming languages by paradigm:

- Imperative: how to solve a problem (steps)?
- Logic-based: what to do to solve a problem ("how to do it" is standard)?
- Functional: what operations can be applied to solving a problem, and how they are mutually related?
- Object-oriented: what objects are involved in a problem, what they can do, and how they interact to solve the problem.

Classification by generality of use:

- general-purpose languages (a majority of languages are in this category);
- specialized languages (e.g., database languages, vector-processing languages).

Classification by sophistication, abstraction, level:

- low-level languages (machine languages, assemblers);
- high-level languages (a majority of languages are in this category);
- very high-level languages (Prolog is sometimes listed in this category).

Beyond programming languages:

- programmibng environments, software development tools and workbenches.

Classification by area of application:

- data processing (also known as "business applications"), now largely superseded by the existence of database systems and other business-related packages, e.g. spreadsheets;
- scientific computing (this includes engineering, too)—today it is changed by new hardware designs such as supercomputers or vector computers, and specialized computing devices;
- Artificial Intelligence and other applications not in the mainstream—this should include educational software and games (for real AI new hardware solutions, such as connection machines and neural networks, so far only simulated);
- "in-house" computing applications—compiler construction, systems programming, GUI, API, on so on.

Criteria for the design and evaluation of programming languages

Points
- Readability
- Writability
- Reliability
- Cost

Readability

This set of criteria is subjective, but very important. Software engineering and in particular the needs of software evolution and maintenance make concern for readability essential.

- Abstraction—support for generality: procedural abstraction, data abstraction.
- Absence of ambiguity (and of too much choice).
- Orthogonality: no restrictions on combinations of concepts. For example, can a procedure parameter have any type? Can everything be evaluated?

  (This is a property that may be carried too far!)

Readability (continued)

- Expressivity of control and data structures.

  Is longer code made of simple elements better (easier to read, maintain and so on)? Or is it better to have shorter code built out of complex, specialized constructions?

  Examples of high expressive power: recursion, built-in backtracking (as in Prolog), search in database languages.

  Examples of low expressive power: machine languages, assemblers.

- Appearance. Style of comments!

Writability

- Abstraction again, and simplicity. This is subjective (Pascal is considered simple, Ada—complicated; Prolog is conceptually simple, but may be difficult to learn).

- Expressivity again.

- Modularity and tools for modularization, support for integrated programmer's environments.

Reliability

- Safety for the programmer (type checking, error and exception handling, unambiguous naming).

Cost

- Development time (ease of programming, availability of shared code).
- Efficiency of implementation: how easy it is to build a language processor (Algol 68 is a known failure, Ada almost a failure; Pascal, C and Java are notable successes).
- Translation time and quality of object code.
- Portability and standardization.

« ‡ ◊ ¤ Δ ¤ ◊ ‡ »

Implementing programming languages

Points

- Language processors, virtual machines
- Models of implementation
- Compilation and execution
- Optimization

A **language processor** is a device that understands and can execute programs in this language.

Translation is a process of mapping the source language into the target language.

The target language may be directly executable on the computer or (more often) may have to be translated again—into a still lower-level language.

A **virtual machine** is a software realization (simulation) of a language processor.

Programming directly for hardware is very difficult—we cover it in layers of software.

A layer may be shared by several language processors, each building its own virtual machine on top of this layer.

Examples of shared layers:

- all language processors require support for input/output;
- all language processors eventually must do computation.

There are normally hierarchies of virtual machines:

- at the bottom, hardware;
- at the top, languages close to the programmer's natural way of thinking.

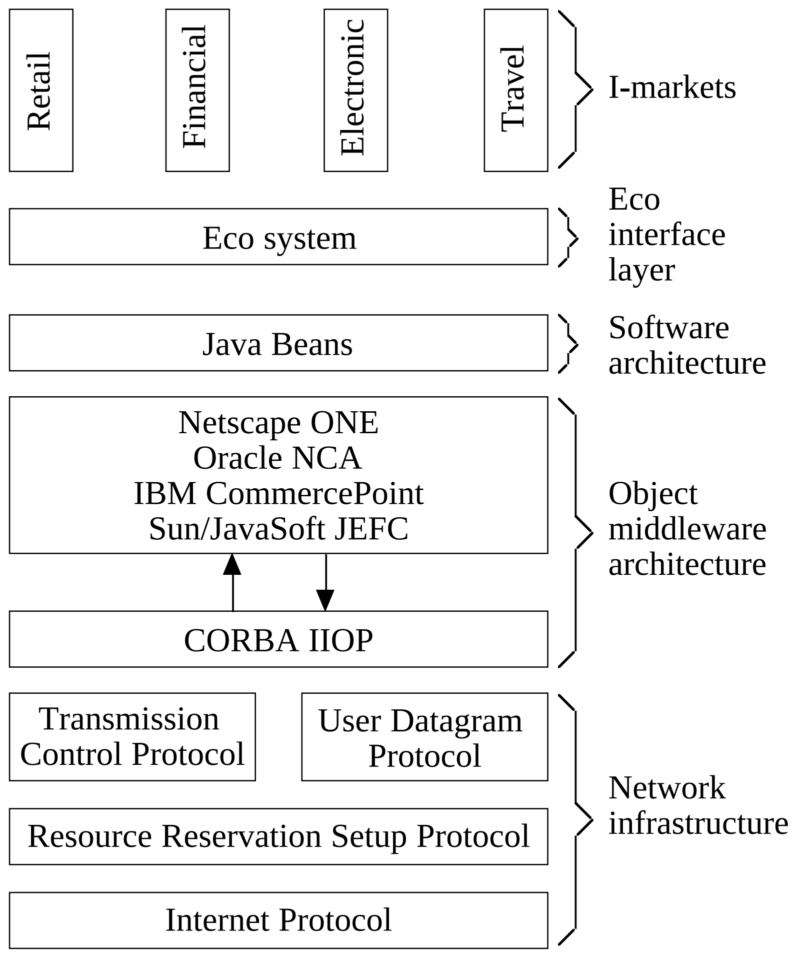
Each layer is expressed only in terms of the previous layer—this ensures proper abstraction.

☒

A typical rich hierarchy of virtual machines:

layer 0:	hardware
layer 1:	microcode
layer 2:	machine language
layer 3:	system routines
layer 4:	machine-independent code
layer 5:	high-level language (or assembler)
layer 6:	application program
layer 7:	input data [this is also a language!]

An example of layers



IEEE Computer May 1997, p. 52

Models of implementation

Compilation: translate the program into an equivalent form in a lower-layer virtual machine language; execute later.

Interpretation: divide the program up into small (syntactically meaningful) fragments; in a loop, translate and execute each fragment immediately.

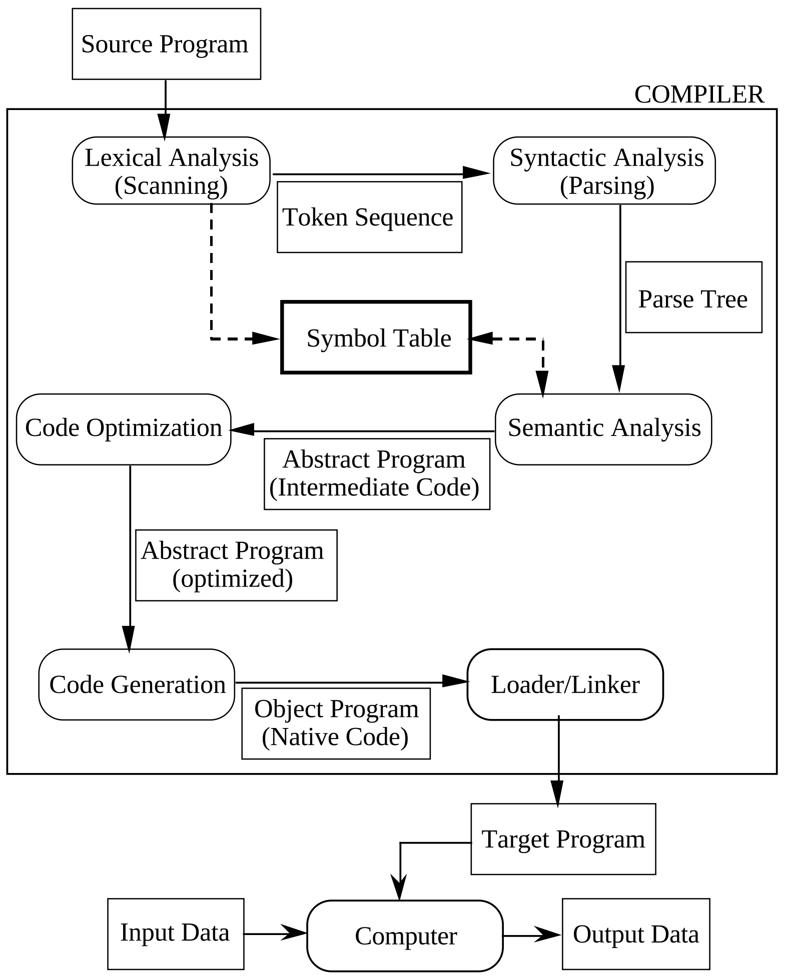
Pure compilation and pure interpretation are seldom use. Normally, an implementation employs a mix of these models. For example: compile Pascal into P-code, then interpret P-code.

We consider a language processor an interpreter if there is more of interpretation, a compiler if there is more of compilation.

Some languages are better interpreted: e.g., interactively used Prolog or Lisp. Some languages are better compiled: e.g. Pascal, C, Ada.

There can also be compiled Prolog or Lisp:
an interpretive top-level loop handles user interaction; predicates are compiled into an optimized form which is then interpreted.

Compilation and execution



☒ Intermediate code for a hypothetical machine with arithmetic registers, generated by a simple-minded compiler:

```
   reg1   :=   I
   reg2   :=   N
   reg3   :=   reg2 + 1
   reg4   :=   reg1 - reg3
   if reg4 ≥ 0 goto L1
   reg5   :=   I
   K      :=   reg5
L1 ...
```

This code is very inefficient. Its optimized form (with the same results) could be this:

```
   reg2   :=   N
   reg2   :=   reg2 + 1
   reg1   :=   I
   reg2   :=   reg1 - reg2
   if reg2 ≥ 0 goto L1
   K      :=   reg1
L1 ...
```

Summary

....

....

....

....

....

....

....

....

....

....

....

....

....

....

....

....