Preliminaries

What we will discuss:

- Programming languages and the process of programming.
- Criteria for the design and evaluation of programming languages
- Basic ideas of programming language implementations.

Programming languages and the process of programming

Points to discuss:

- 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 have to *maintain* the program.
- Programming languages are used to instruct computers.
  - What do we communicate to computers?
  - How do computers talk back to us?

Programming means more than coding (2)

- How do programming languages differ from natural languages? Would talking to computers (instructing them) in human languages be preferable?
- What makes someone a good programmer?
- Should a good programmer know more than one programming language?

Why should we study programming languages?

- To have an increased capacity to express ideas
  - By learning new language constructs
  - develop an appreciation for valuable language features and constructs
    - e.g., simulating Perl's associative arrays in C

Why should we study programming languages? (2)

- To have a choice of programming tools that best match the task at hand:
  - people are fluent in multiple languages
  - identify subtasks and apply to each of them *the best* language, using the full expressive power of each language.
    - the Microsoft.NET case
    - libraries and community-developed extensions could be a differentiating factor

Why should we study programming languages? (3)

- To understand better the connection between algorithms and programs.
  - some programming languages are more suitable for solving particular types of 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
    - the Microsoft.NET example

Why should we study programming languages? (4)

- To learn new programming languages easily
  - familiarize with the building blocks of the language
  - able to recognize them later in other languages
    - e.g., OOP concepts -> Ruby

Why should we study programming languages? (5)

- To appreciate the workings of a computer equipped with a programming language—by knowing how languages are implemented.
  - improve software efficiency by knowing implementation issues
    - understanding the impact of the different choices (data structures, workflows) on the efficiency of the implementation
      - e.g., vectorized operations in MATLAB preferred over repetitive constructs

Why should we study programming languages? (6)

- To see how a language may influence the discipline of computing and strengthen good software engineering practice.
  - Some programming languages could have become very popular due to some appealing features
  - Yet their conceptual design was not well understood by the software engineering community
  - e.g., ALGOL 60 vs. Fortran

The many classes of programming languages: programming language paradigms

- Every programming language supports a slightly different or dramatically different style of problem solving.
- The same computation can be expressed in various languages, and then run on the same computer.

Programming languages classified by paradigm

- Imperative: *how* do we solve a problem (what steps does a solution have)?
  - Object-oriented: what *objects* play roles in a problem, what can they do, and how do they interact to solve the problem?
- Logic-based: *what* do we do to solve a problem? (The language decides *how* to do it.)
- Functional: what simple *operations* can be applied to solving a problem, how are they mutually related, and how can they be combined?

Classification by generality of use

- General-purpose programming languages (most of the known languages are in this category);
- Specialized programming languages
  - database languages
  - vector-processing languages
  - mobile languages
  - data management languages
  - web languages
  - and more

Classification by abstraction level

- Low-level (1GL-2GL) languages (machine languages, assembly languages).
- High-level (3GL) languages (most of the well-known languages belong to this category).
- Very high-level (4GL) languages (Prolog is sometimes listed in this category, and some specialized languages).
- Beyond programming languages:
  - Programming environments, software development tools and workbenches.

Classification by area of application

- Data processing (also known as "business applications").

  Now made largely unnecessary, since we have databases and other business-related packages, such as spreadsheets, and special-purpose software.

- Scientific computing (this includes engineering).

  Today this has been changed by new hardware designs such as supercomputers or vector computers, and specialized computing devices.

Classification by area of application (2)

- Artificial intelligence and other applications not in the computer science mainstream.
  - This might include educational software and games.
  - New hardware (so far mostly simulated) such as connection machines and neural networks.
    - e.g. Deep Learning (Theano, TensorFlow)
- "In-house" computing applications.
  - compiler construction, systems programming, GUI, API, and so on.

Criteria for the design and evaluation of programming languages

- Points to discuss:
  - Readability
  - Writability
  - Reliability
  - Cost

Criteria for the design and evaluation of programming languages

- **Readability**: the ease with which programs can be read and understood
- **Writability**: the ease with which a language can be used to create programs
- **Reliability**: conformance to specifications (i.e., performs according to its specifications)
- **Cost**: the ultimate total cost

Criteria for the design and evaluation of programming languages

Table 1.1 Language evaluation criteria and the characteristics that affect them

	CRITERIA		
Characteristic	READABILITY	WRITABILITY	RELIABILITY
Simplicity	•	•	•
Orthogonality	•	•	•
Data types	•	•	•
Syntax design	•	•	•
Support for abstraction		•	•
Expressivity		•	•
Type checking			•
Exception handling			•
Restricted aliasing			•

Readability

- This is subjective, but very important.
- Language readability is essential because of software engineering practices, and in particular the needs of software evolution and maintenance.
  - 1970s the **software lifecycle** concept
  - Abstraction—support for generality of programs: procedural abstraction, data abstraction.
  - Absence of ambiguity (and absence of too many coding choices, like having five different loop constructs).

Readability (2)

- Simplicity:
  - Number of basic constructs: keep it small
  - Minimal feature multiplicity

```
count = count + 1
count += 1
count++
++count
```

  - Minimal operator overloading
    - could be misleading if users do not employ it carefully
      - e.g., “+” to sum all elements in an array
      - “+” to sum the first two elements in an array

Readability (2)

- Orthogonality: there are no restrictions on combinations of primitive language concepts.
  - Orthogonality follows from a symmetry of relationships among primitives.
  - e.g., a pointer to any primitive data type
  - pointers to arrays too (type operators)
- More orthogonality = fewer special cases in the language.
- This may be carried too far (as in Algol 68).
- Simplicity is partially dictated by orthogonality.
- Functional languages are good at both.

Readability (3)

- Data types need to be adequately defined
  - `timeOut = 1`
  - `timeOut = true`
- Syntax design
  - Special words
    - `while, class, for…`
    - `ADD AX, BX` vs. `AR AX, BX`
  - Compound statements, e.g., `{ }` vs. `end if`
  - Using keywords as variable names (Fortran 95)

Writability

- Simplicity and orthogonality: once more, subjective.
  - Pascal has always been considered simple, Ada complicated
  - Basic is very simple
  - Prolog is conceptually simple, but may be difficult to learn.
  - is Java simple?
- Modularity and tools for modularization, support for integrated programming environments.
- Writability should be considered in the context of the target problem domain of a language.

Writability (2)

- Expressivity of control and data structures.
  - what is better (easier to read, maintain and so on):
    - a longer program made of simple elements?
    - a shorter program 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: instructions in machine or assembly languages.
- Appearance: syntax, including comments.

Reliability

- If a program performs according to its specifications under all conditions
  - avoiding errors as much as possible
    - More safety for the programmer
  - **Type checking**
    - Strongly typed languages are often more reliable
      - compile-time checking whenever possible
  - **Error and exception handling**
    - intercept runtime errors, take corrective measures and then continue
  - **Aliasing**: Accessing same memory locations via different names

Cost

- 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, C++ and Java have been notable successes).
  - Translation time and the quality of object code.
  - Maintenance
  - Portability and standardization.

In a nutshell…

- Designing a programming language requires the reconciliation and satisfaction of multiple, often conflictive, evaluation criteria
  - Reliability vs. cost of execution
    - Type checking increases run time
    - Java designers traded execution efficiency for reliability
  - Writability vs. readability
  - Writability vs. reliability
    - Pointers in C++ did not make it to Java
- Choosing constructs and features for a programming language requires many compromises and trade-offs.

Implementing programming languages

- Points to discuss:
  - Language processors, virtual computers
  - Models of implementation
  - Compilation and execution

Language processors

- A **processor** for language L is any device (hardware or software) that understands and can execute programs in language L.
- **Translation** is the process of mapping a program in the source language into the target language while preserving the meaning or function of the source program.
- The **target language** may be directly executable on the computer or (more often) may have to be translated again — into an even lower-level language.

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.
    - fetch/execute cycle

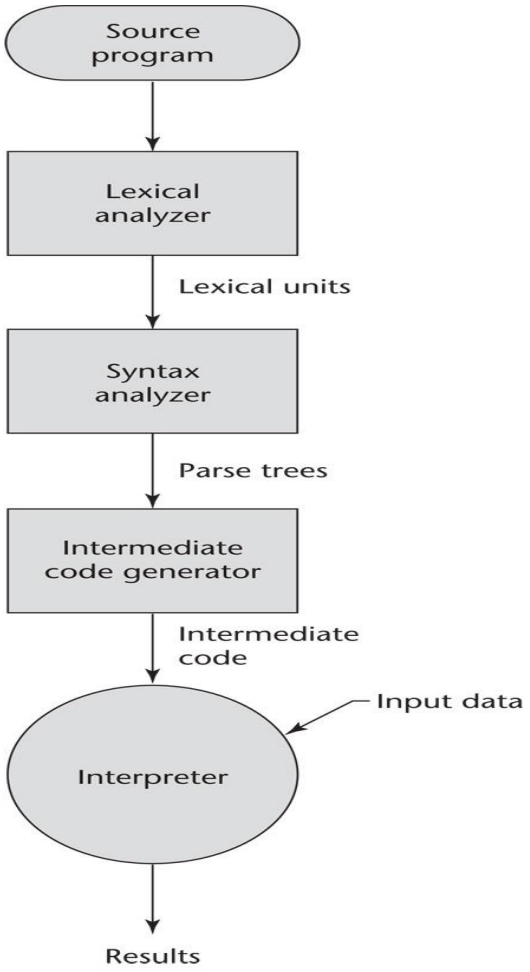
Models of implementation (2)

- Compilation:
  - Very fast program execution
    - directly using the native code of the target machine
  - opportunity to apply quite powerful optimizations during the compilation stage
- Interpretation:
  - easier to implement
  - no need to run a compilation stage: can execute code directly "on the fly“

Models of implementation (3)

- Pure compilation and pure interpretation are seldom used. Normally, an implementation employs a mix of these models.
  - For example: compile Java into bytecodes, then interpret bytecodes.
- We consider a language processor an interpreter if it has "more interpretation than compilation". We consider a language processor a compiler if there is more of compilation.

Models of implementation (4)

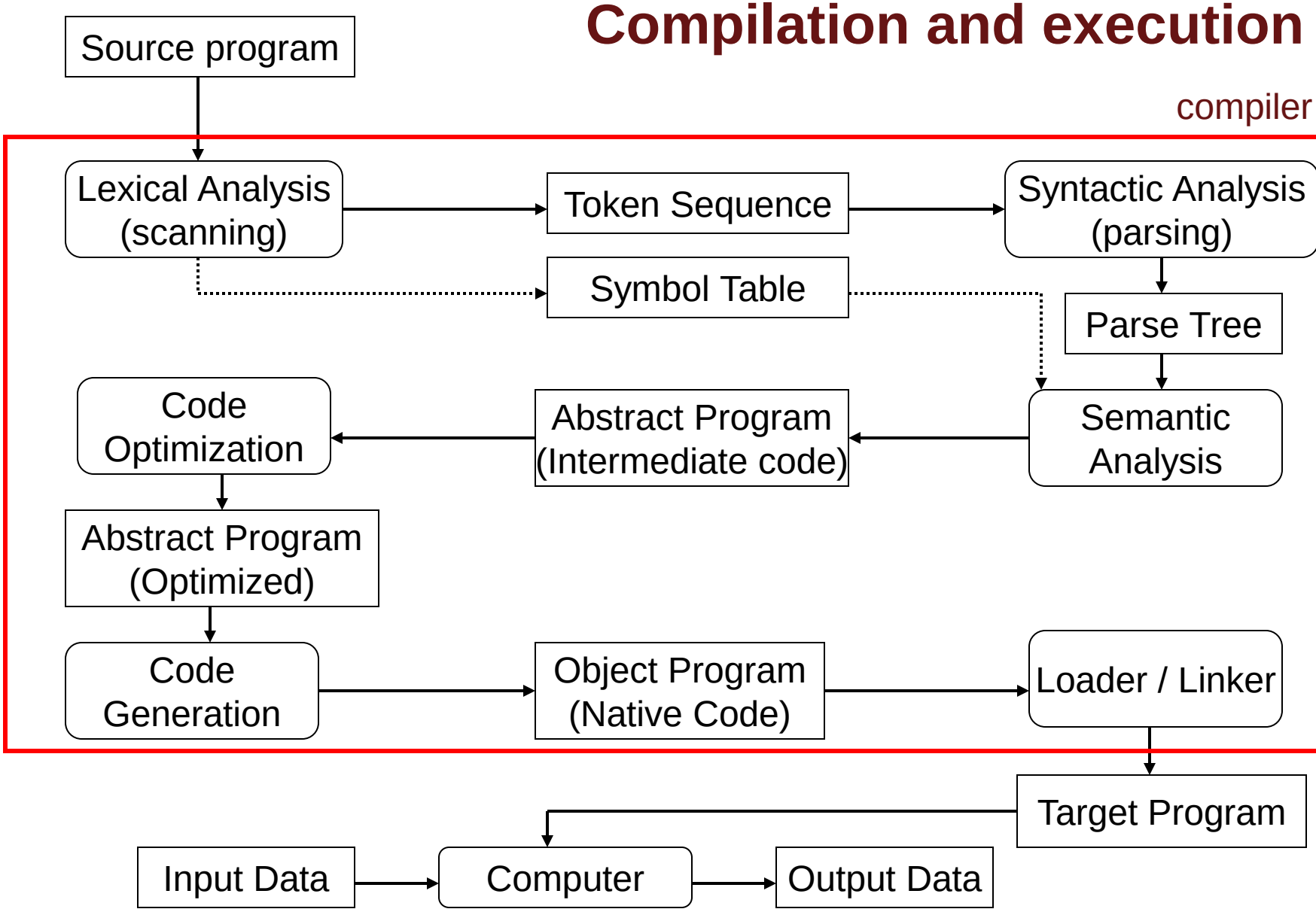


Copyright ©2016 Pearson Education, All Rights Reserved

Models of implementation (5)

- Some languages are better interpreted, for example interactively used Prolog or Lisp.
- Some languages are better compiled, for example, C++, Java.
- There can also be compiled Prolog or Lisp:
  - an interpretive top-level loop handles user interaction.
  - Predicates of functions are compiled into an optimized form which is then interpreted.

Compilation and execution



Virtual computers

- A **virtual computer** is a software realization (simulation) of a language processor.
- Programming directly for hardware is very difficult — we usually "cover" hardware with layers of software.
- A layer may be shared by several language processors, each building its own virtual computer on top of this layer.

Examples of shared layers

- All language processors require support for input/output.
- All language processors eventually must do some calculations, that is, use the CPU.

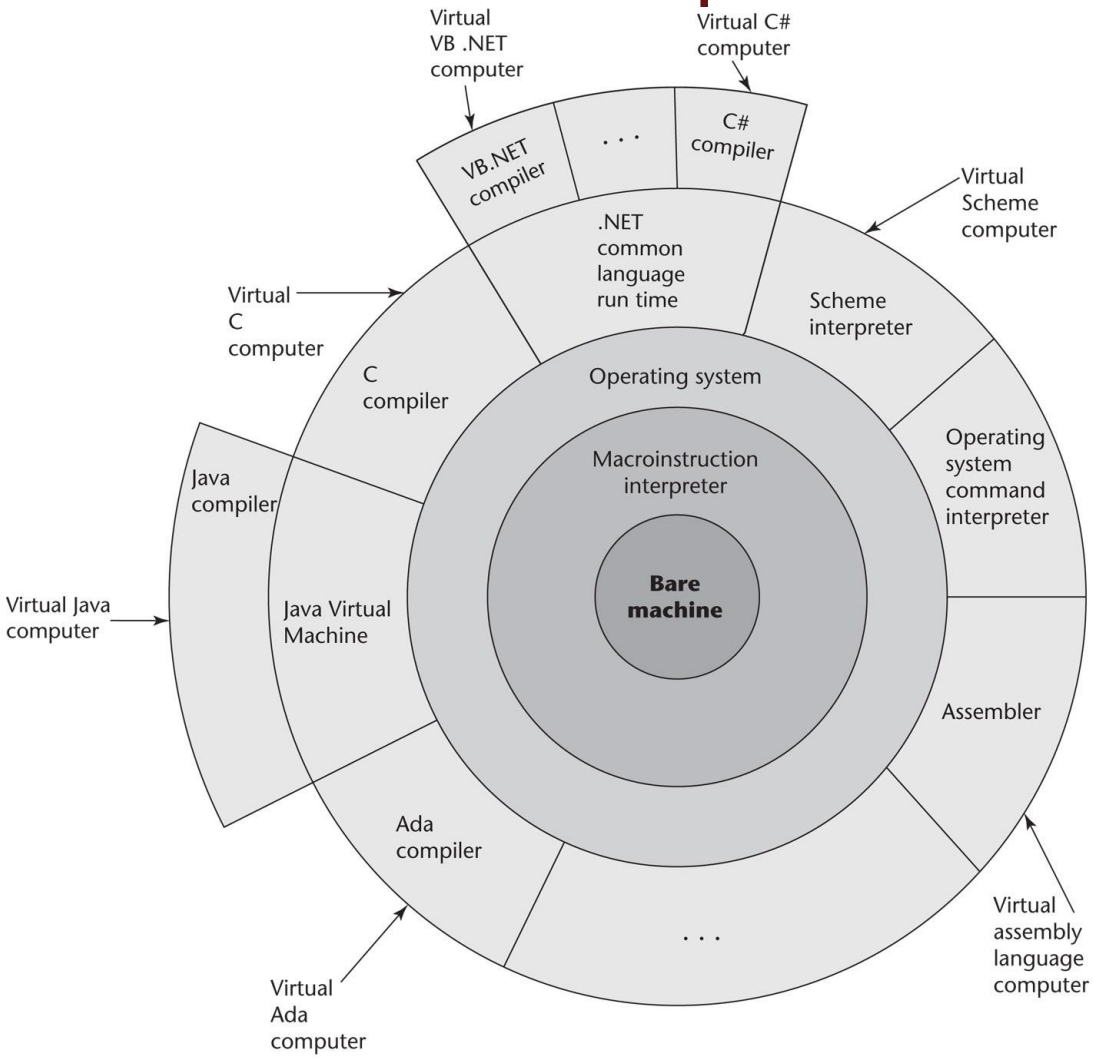
Virtual computers

- We normally have 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.

Language processors and the operating system

- OS mediates between a language processor and the macroinstruction / microinstruction processors.
  - e.g. an OS and a C compiler provide a virtual C computer.

Virtual computers



Copyright ©2016 Pearson Education, All Rights Reserved

A generic hierarchy of virtual computers

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!]

Virtual computers — example

Layer 0: IBM Netvista with Intel Pentium 4, 2GHz

Layer 1: IBM Intel machine language

Layer 2: Windows NT 4.0

Layer 3: Java byte-code

Layer 4: JDK 1.2

Layer 5: a Java implementation of Prolog

Layer 6: a Prolog implementation of mySQL

Layer 7: a database schema defined and created

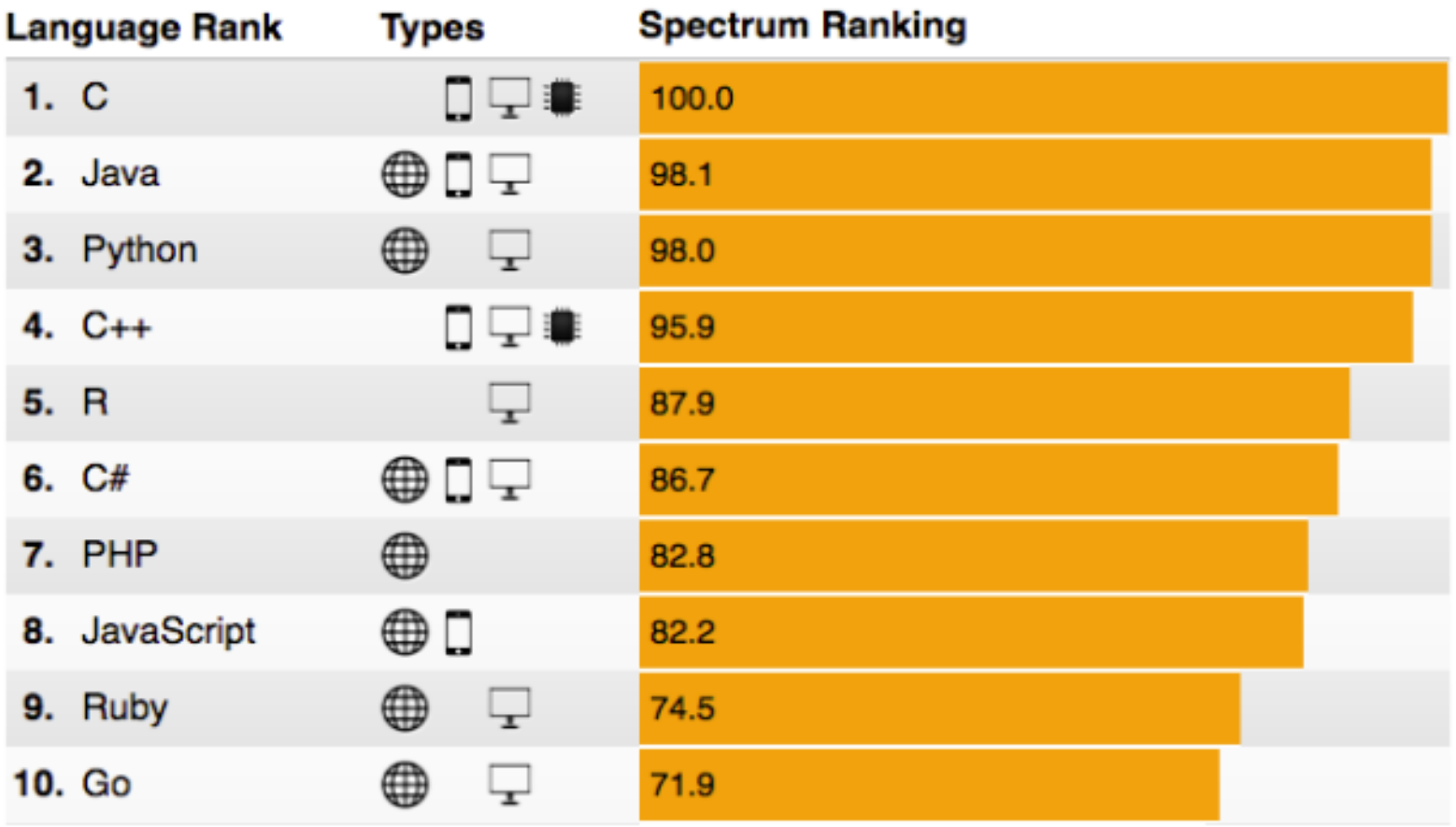
Layer 8: records for insertion into the database

August 2016
TIOBE Programming Languages Index

Aug 2016	Aug 2015	Change	Programming Language	Ratings	Change
1	1		Java	19.010%	-0.26%
2	2		C	11.303%	-3.43%
3	3		C++	5.800%	-1.94%
4	4		C#	4.907%	+0.07%
5	5		Python	4.404%	+0.34%
6	7	^	PHP	3.173%	+0.44%
7	9	^	JavaScript	2.705%	+0.54%
8	8		Visual Basic .NET	2.518%	-0.19%
9	10	^	Perl	2.511%	+0.39%
10	12	^	Assembly language	2.364%	+0.60%

more details here

2016 IEEE Programming Languages Ranking



more details here