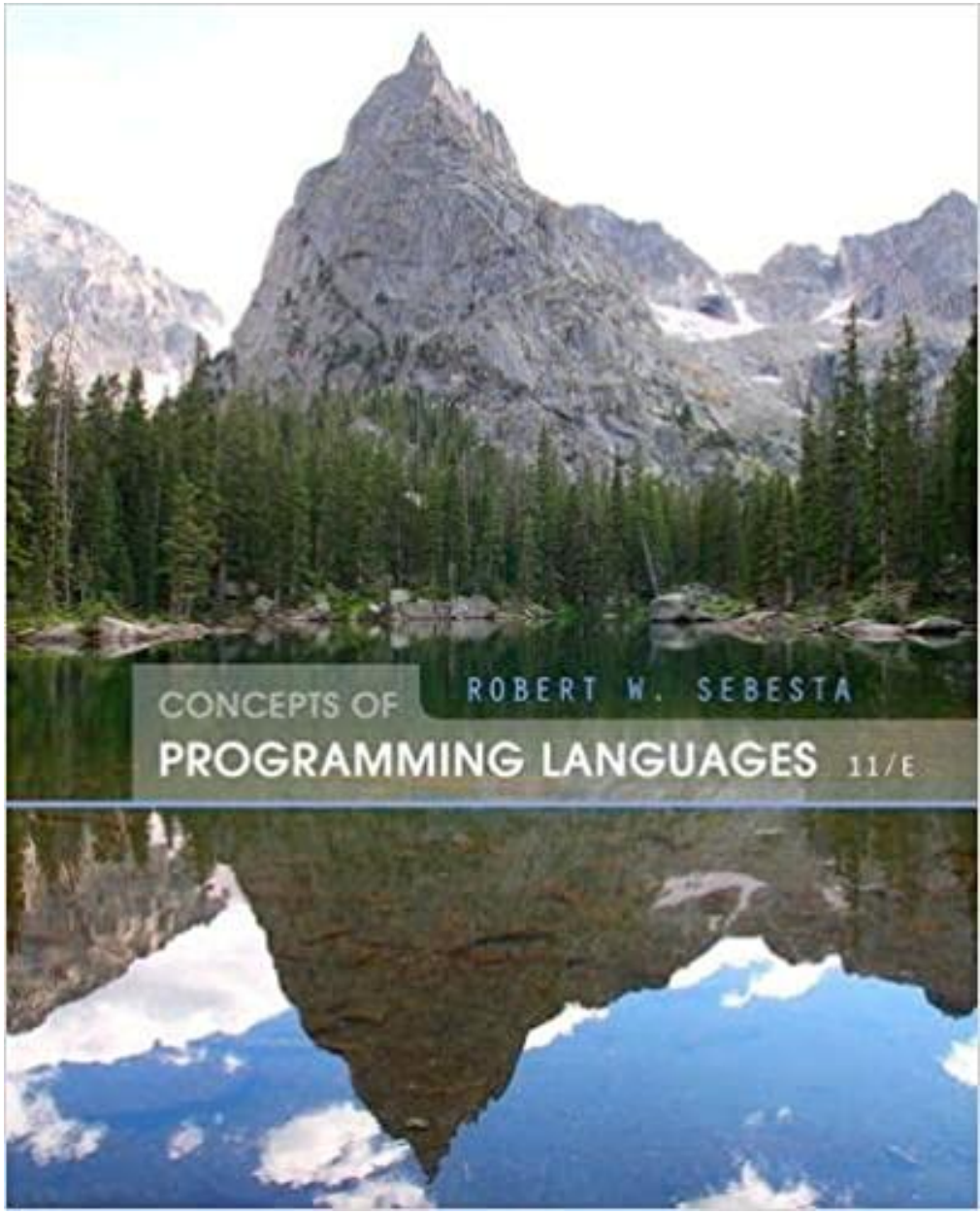
# Chapter 1

COMP333: Concepts of Programming Languages

## Preliminaries

Kamran Zamanifar, PhD.



# Chapter 1 Topics

- Reasons for Studying Concepts of Programming Languages
- Programming Domains
- The Top Programming Languages 2024
- Language Evaluation Criteria
- Influences on Language Design
- Language Categories
- Language Design Trade-Offs
- Implementation Methods
- Programming Environments

# Reasons for Studying Concepts of Programming Languages

- Increased ability to express ideas
- Improved background for choosing appropriate languages
- Increased ability to learn new languages
- Better understanding of significance of implementation
- Better use of languages that are already known
- Overall advancement of computing

# Programming Domains

- Scientific applications
  - Large numbers of floating points computations; use of arrays
  - Fortran, Fortran90, HPF, ALGOL 60, MATLAB
- Business applications
  - Produce reports, use decimal numbers and characters
  - Cobol
- Artificial intelligence
  - Symbols rather than numbers manipulated; use of linked lists
  - Lisp, Common Lisp, Prolog
  - Python for implementing LLM (Large Language Model)
- Systems programming
  - Need efficiency because of continuous use
  - C, C++, PL/1, BLISS, ALGOL.
- Web Software
  - Collection of languages: markup (e.g., HTML), scripting (e.g., JS, PHP), general-purpose (e.g., Java)

# Programming Domains

- Web Development:
» Front-end (what user sees & interact): **HTML, CSS, JS**
» Back-end (Behind-the scenes logic, databases, servers): **JS, Java, Python, C#, php**

- Mobile Development:
» Native: for Android Apps: **Java, KATLIN**
» for iOS Apps: **Swift, OBJ-C**
» Cross platform (use a single code base for both Android and iOS): **JS, Dart**

- Game Development:
» Unity Engine (small and medium games): **C#**
» UNREAL engine (advanced games): **C++**

- Embedded Systems (Applications): **C, C++, RUST**
» **Arduino** Language or platform for education kits

# The Top Programming Languages 2024

- **Python:** easy to learn, OOP & Procedural, it is very slow languages (run time speed is slow), general purpose language, AI Engines, machine learning, Desktop Apps, Web developing, Data sciences, it is interpreted, it is suitable for small and large companies, good documentations, *Spotify*, *intel*, *IBM*
- **Java Script (JS):** easy learning, Web Development (back-end & front-end, Netscape browsers), used in small and very large companies. It is interpreted language . Dynamic language (type checking), *PayPal*, *NETFLIX*, *Bigbasket*, *Google*
- **C:** medium(low) level compiled language (closer to CPU), procedural language, System programming (OS), Game development (Engines), Core AI Engines, Embedded applications, supper efficient, fast, many different company use this language.
- **C++:** Object Oriented programming version of C, Desktop Apps, Systems programming, memory issues (memory allocation & deallocation), fast, memory issues. *Google*, *Evernote*, *Linked**in***, *Microsoft*, *Opera*, *NASA*
- **JAVA:** Different from **JS,** open source**,** a universal language, it is hybrid (compiled and interpreted) using JVM,Web Apps, Desk top Apps, Mobile development (Android Apps), is used in medium to large companies. Many documentation, *Uber*, *Airbnb*, *Instagram*

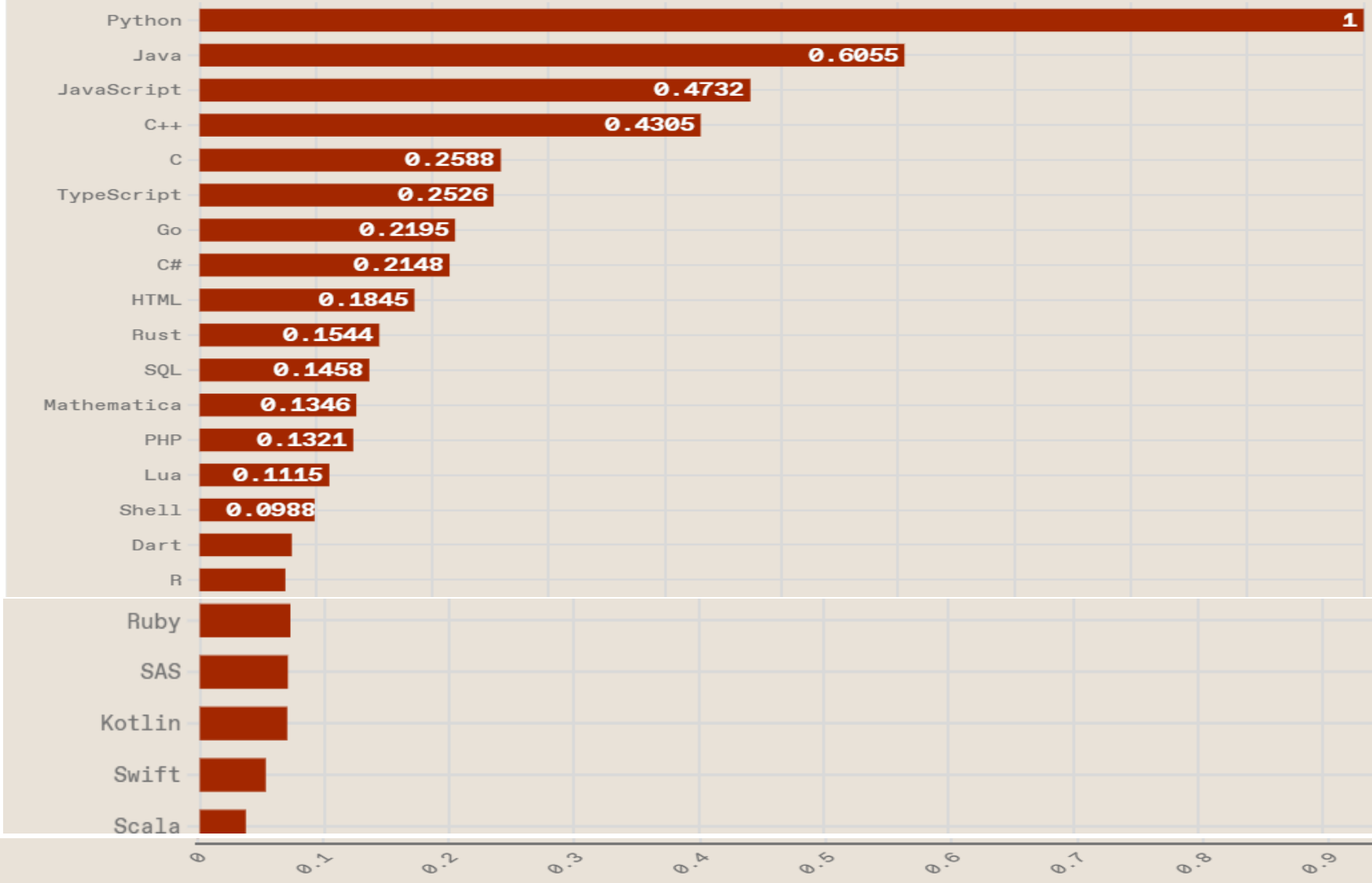
# The Top Programming Languages 2024

- **C#:** OOP, Web Apps, Desktop Apps, Game development, Systems programming, Mobile development, Server-side programming. Supported by .NET(Microsoft) platform. Many documentation. *Microsoft*, *LIBABA TRAVELS*, *Facebook (Meta)*, *Accenture* supports the *C#*
- **php:** Personal Home Page or Hypertext Preprocessor (HTML), Open source, Interpreted language, Web development, small companies, start up, contain management, good documentation, *Facebook*, *Tumblr.*, *WordPress*
- **Swift:** OOP, Used by Apple, iOS (mobile development, OS) for iPhone, iPadOS, MacOS, runs very fast, similar to KATLIN
- **RUST:** alternative for C++, fast, modern language, hard using, good documentation
- **CSS:** Web development(front-end), styling web pages
- **HTML:** Web development
- **ASM(assembly**): low level language, fast, programming on chip, Embedded Apps
- **GO:** new language, open source, easy learning, supports pointers, fast, cloud native language, Google
- **KATLIN:** replacement for Java, general purpose, Web development, Mobile Apps, Android Apps, good documentation, *NETFLIX*, *Coursera*, *Pinterest*, *Postmates*
- **MySQL:** Data base management, good documentation, *Oracle*
- **Ruby:** OOP, dynamic, easy learning and flexible, extremely popular, Web development, good documentation, it is interpreted

# Top Programming Languages 2024

Click a button to see a differently weighted ranking

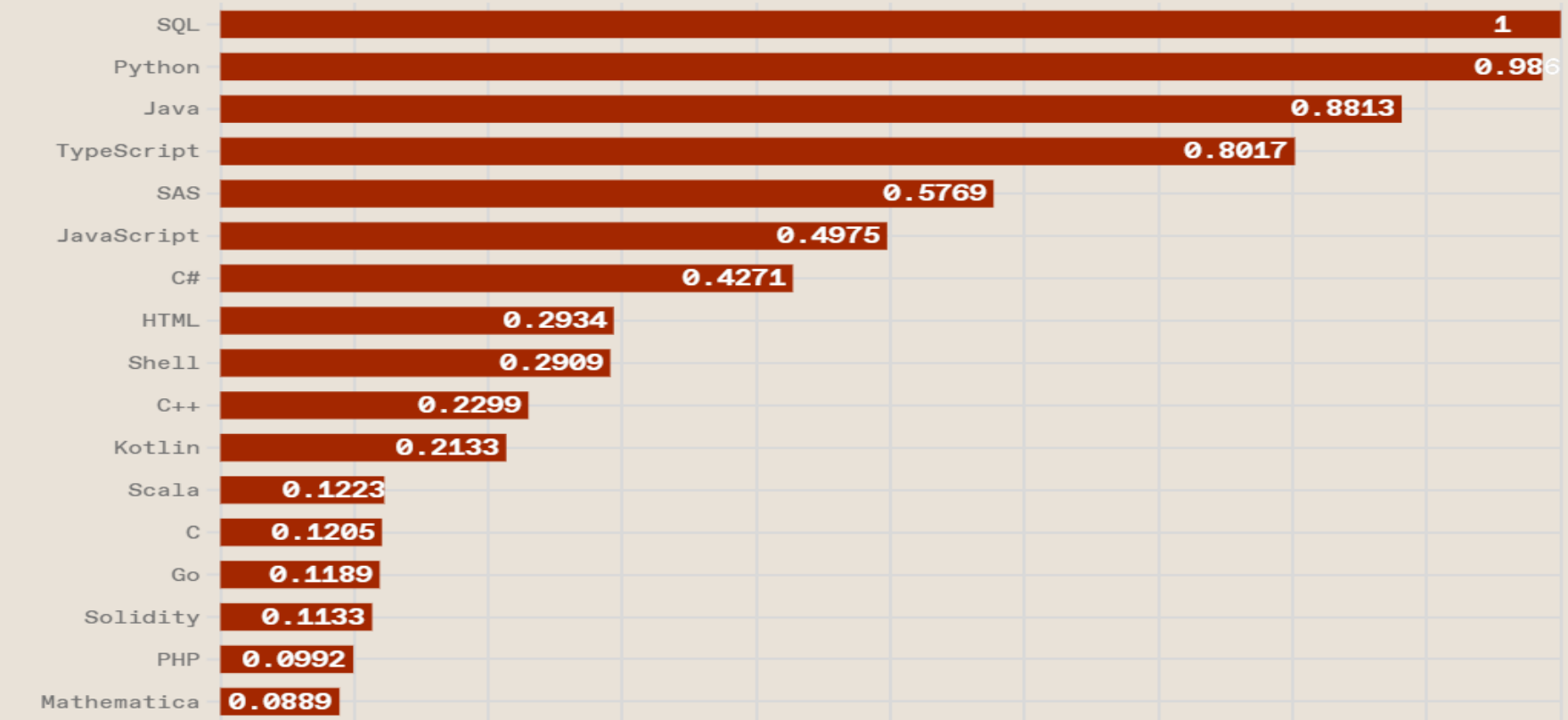
Spectrum | Trending | Jobs



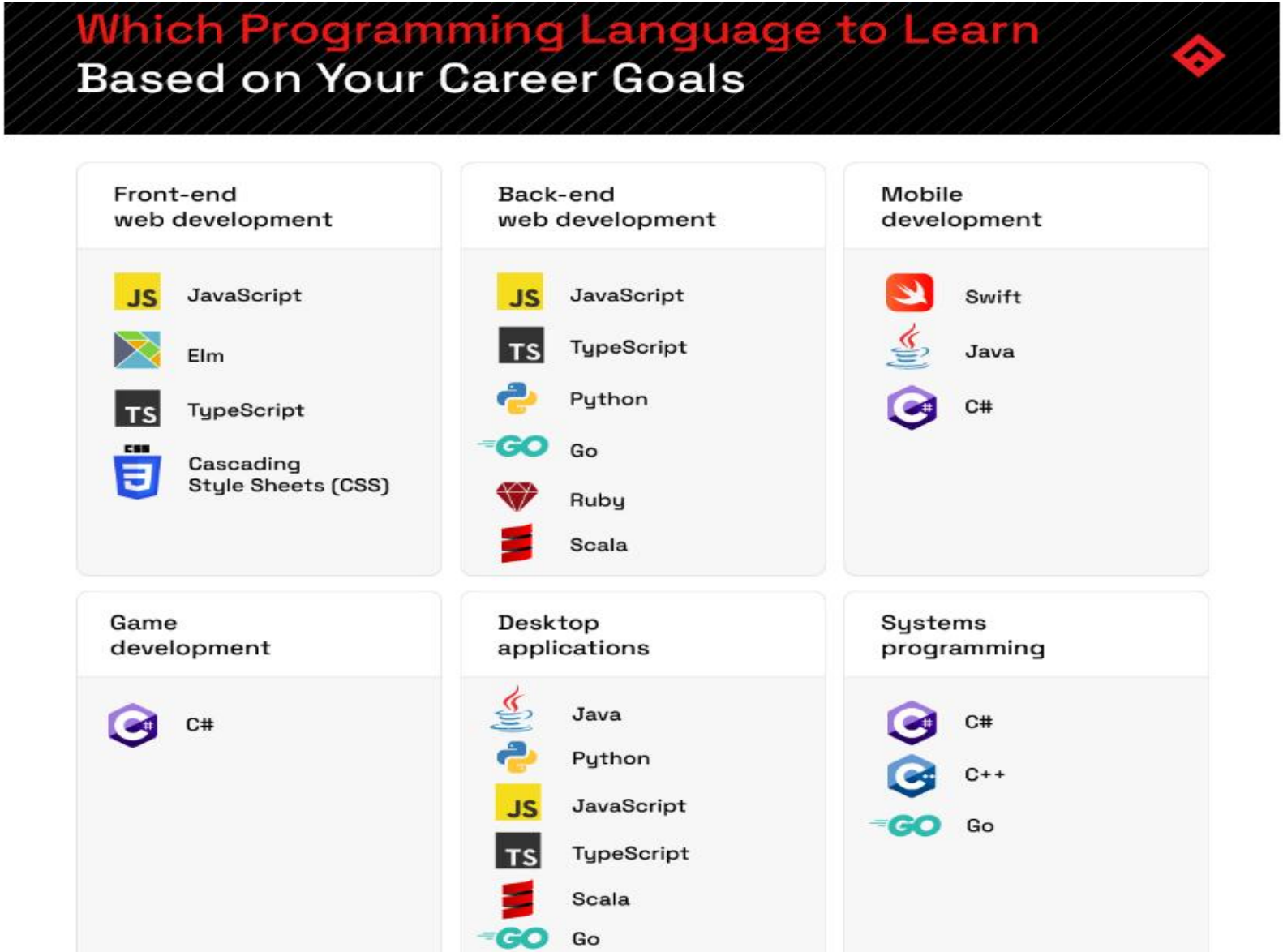
# Top Programming Languages 2024

Click a button to see a differently weighted ranking

Spectrum | Trending | Jobs

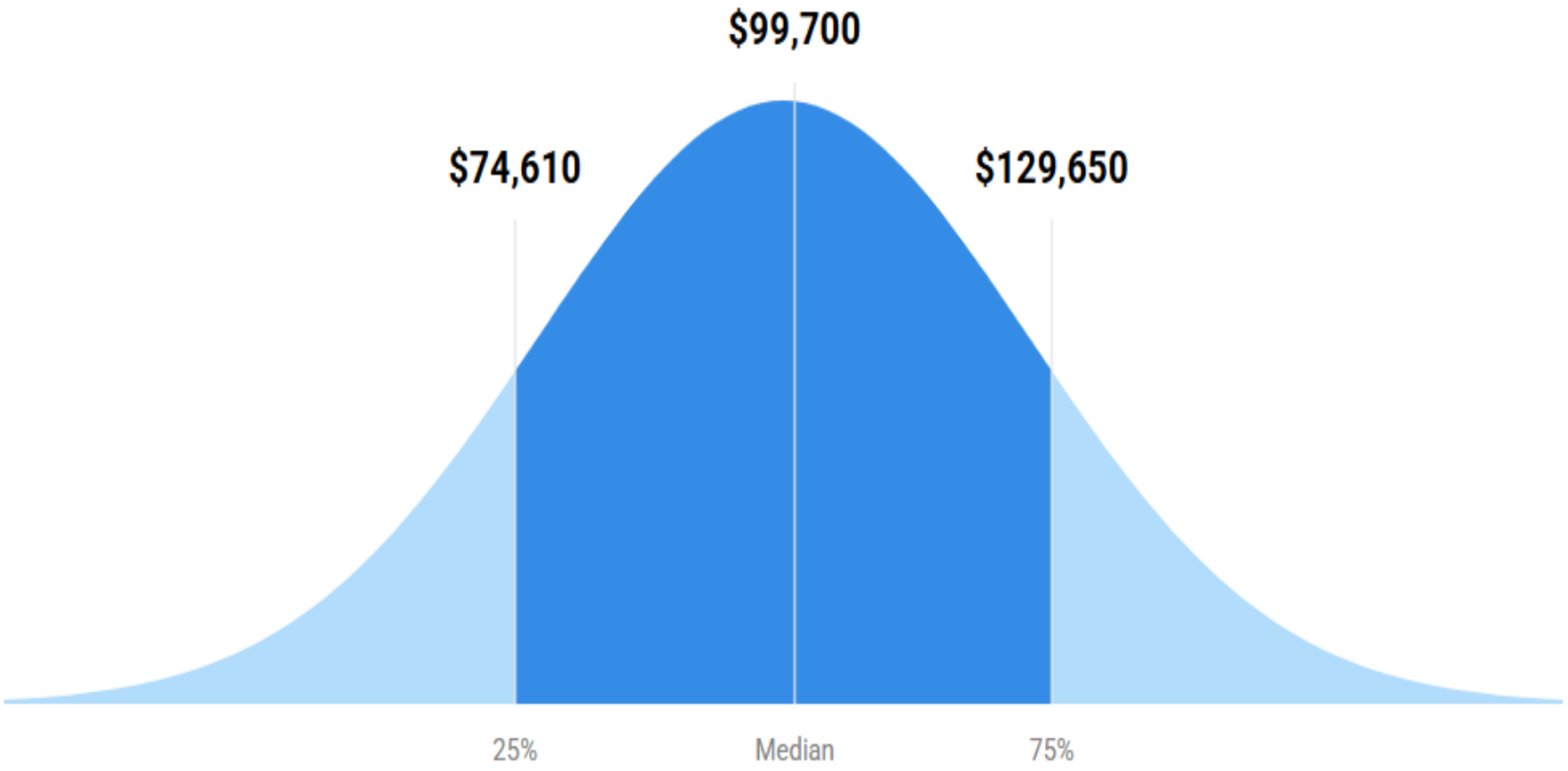


# The Top Programming Languages 2024



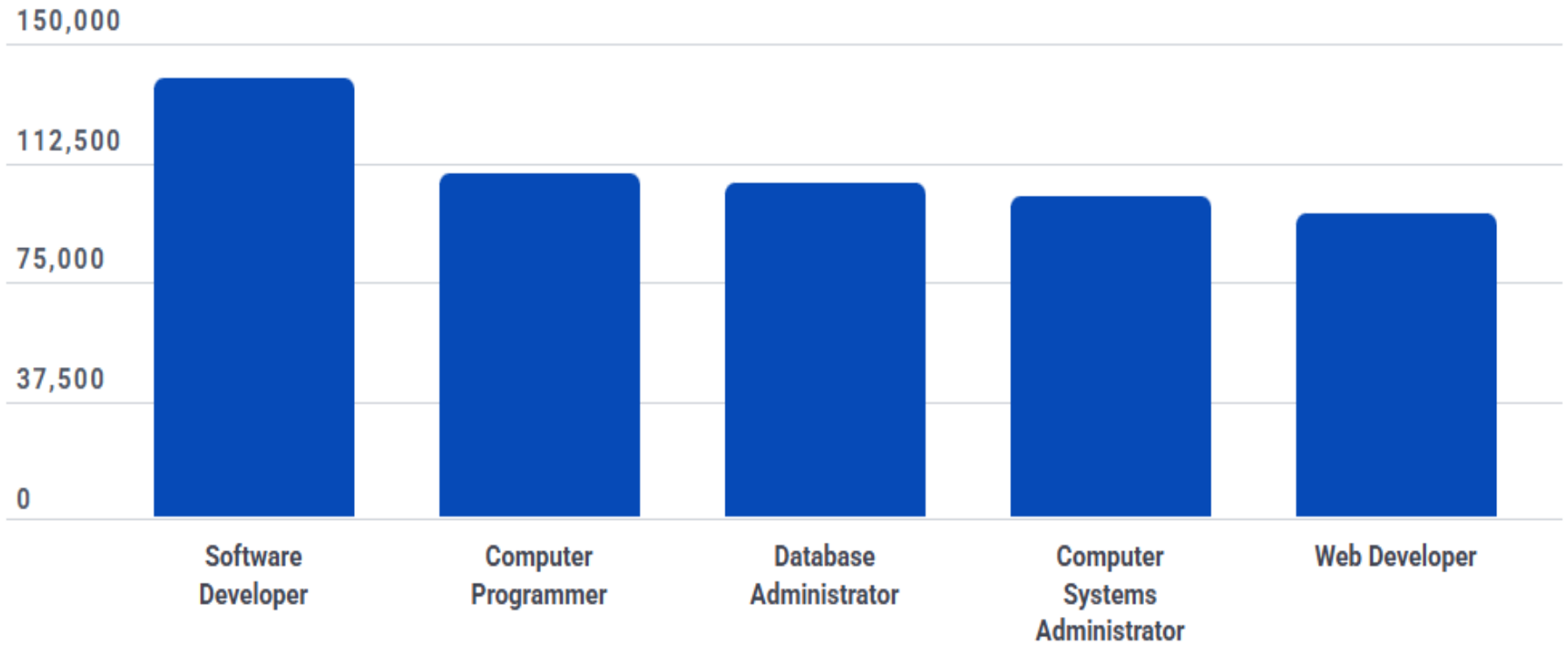
# How Much Does a Computer Programmer Make?

Computer Programmers made a median salary of $99,700 in 2023. The best-paid 25% made $129,650 that year, while the lowest-paid 25% made $74,610.



# Average Computer Programmer Pay vs. Other Best Jobs

Computer Programmers earned an average salary of $107,750 in 2023. Comparable jobs earned the following average salary in 2023: Software Developers made $138,110, Database Administrators made $104,810, Computer Systems Administrators made $100,580, and Web Developers made $95,570.



# Language Evaluation Criteria

- **Readability:** One of the most important criteria for judging a programming language is the ease with which programs can be read and understood.
- **Writability:** Writability is a measure of how easily a language can be used to create programs for a *chosen problem domain.*
- **Reliability:** A program is said to be reliable if it performs to its specifications under all conditions
- **Cost**: The ultimate total cost which is a function of the cost of training programmers to use the language, the cost of writing programs in the language, and the cost of executing programs written in a language

# Language Evaluation Criteria

**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 | | | • |

# Evaluation Criteria: Readability

- One of the most important criteria for judging a programming language is the ease with which programs can be **read and understood**.
- Before 1970, software development was largely thought of in terms of writing code.
- Primary positive characteristic of programming languages was efficiency.
- Language constructs were designed more from the point of view of the computer than of the computer users. But today focus is on human orientation.
- Readability must be considered **in the context of the problem domain**.
- The **overall simplicity** of a programming language strongly affects its readability.
- A language with a large number of basic constructs is more difficult to learn than one with a smaller number.

# Evaluation Criteria: Readability

- A second complicating characteristic of a programming language is the feature **multiplicity**—that is, having more than one way to accomplish a particular operation.
- Another potential problem is **operator overloading**, **in which a single operator symbol has more than one meaning**. Although this is often useful, some times it can lead to reduced readability.
- **Orthogonality** means that a small set of primitive constructs can be combined in a small number of ways to build the control and data structures of the language. *C/C++ is not orthogonal because arithmetic operators do not consistently work on pointers*.
- **Orthogonality is closely related to simplicity: The more orthogonal the design of a language, the fewer exceptions the language rules require.** Fewer exceptions mean a higher degree of regularity in the design, which makes the language easier to learn, read, and understand.

# Evaluation Criteria: Readability

- The word orthogonal comes from the mathematical concept of orthogonal vectors, which are independent of each other.
- Orthogonality follows from a symmetry of relationships among primitives
- For example, in IBM mainframes there are 2 instructions for adding the contents of *memory cells* and *registers*, but in VAX minicomputers there is a single instruction for memory cells and registers.
- VAX instruction design is  orthogonal.

# Evaluation Criteria: Readability

- **Overall simplicity in a language is supported by**:
  - A manageable set of features and constructs
  - Minimal feature multiplicity
  - Minimal operator overloading

- **Orthogonality**
  - A small set of primitive constructs can be combined in a small number of ways which supports readability
  - Every possible combination should legal

- **Data types**
  - Adequate predefined data types and data structures is another significant aid to support readability. The meaning of **timeout = true** is perfectly clear.

- **The syntax, or structure of a language**
  - Special words (for example, *while, class, and for*) and methods of forming compound statements, i.e.{ }, *end, end if, end loop*
  - Identifier forms (naming): flexible composition
  - Form and meaning: self-descriptive constructs, meaningful keywords

# Evaluation Criteria: Writability

- Writability is a measure of how easily a language can be used to create programs for a *chosen problem domain*.

- Most of the language characteristics that affect readability also affect writability.
- As is the case with readability, writability must be considered in the **context of the target problem domain of a language**.

- It is simply not reasonable to compare the writability of two languages in the realm of a particular application when one was designed for that application and the other was not.

# Evaluation Criteria: Writability

- **Simplicity and orthogonality**
  - Few constructs, a small number of primitives, a small set of rules for combining them supports writability
- **Support for abstraction**
  - Abstraction means the ability to define and use complex structures or operations in ways that allow details to be ignored
  - Two distinct categories of abstraction, process and data is supported
  - Process abstraction is the use of a subprogram to implement a sort algorithm that is required several times in a program (reusing)
- **Support for abstraction is an important factor in the writability of a language**
- **Expressivity** means that a language has relatively convenient ways of specifying computations. For example, in C, the notation *count*++ is more convenient and shorter than *count = count + 1*.
- **Expressivity supports writability by:**
  - A set of relatively convenient ways of specifying operations
  - Strength and number of operators and predefined functions
  - The inclusion of the **for statement** in Java makes writing loops easier than with the use of **while.**

# Evaluation Criteria: Reliability

- **Type checking**
  - Type checking is testing for type errors in a given program, either by the compiler or during program execution
  - Type checking is an important factor in language reliability
- **Exception handling**
  - The ability of a program to intercept run-time errors, take corrective measures, and then continue is an obvious aid to reliability
- **Aliasing**
  - Aliasing is a dangerous feature in a programming
- **Readability and writability**

— language that does not support "natural" ways of expressing an algorithm will require the use of "unnatural" approaches and hence reduced reliability

— The easier a program is to write, the more likely it is to be correct

— Readability affects reliability in both the writing and maintenance phases of the life cycle. Programs that are difficult to read are difficult both to write and to modify.

# Evaluation Criteria: Reliability

- A program is said to be **reliable** if it performs to its specifications under all conditions.
- **Type checking** is simply testing for type errors in a given program, either by the compiler or during program execution. Type checking is an important factor in language reliability

- The ability of a program to intercept (follow up) run-time errors (as well as other unusual conditions detectable by the program), take corrective measures, and then continue is an obvious aid to reliability. This language facility is called **exception handling**.

- **Aliasing** is having two or more distinct names that can be used to access the same memory cell. It is now widely accepted that aliasing is a dangerous feature in a programming language

# Evaluation Criteria: Cost

• The total cost of a programming language is a function of many of its characteristics.

1. **Cost of training programmers** to use the language, which is a function of the simplicity and orthogonality of the language and the experience of the programmers.
2. **Cost of writing programs** in the language. This is a function of the writability of the language (closeness to particular applications)
3. **Cost of compiling programs** in the language.
4. **Cost of executing programs** written in a language is greatly influenced by that language's design.
5. **Cost of the language implementation system**.
6. **Cost of poor reliability**. If the software fails in a critical system, such as a nuclear power plant or an X-ray machine for medical use, the cost could be very high.
7. **Cost of maintaining programs,** which includes both corrections and modifications to add new functionality. The **software maintainability** is so important. It has been estimated that for large software systems with relatively long lifetimes**, maintenance costs can be as high as two to four times as much as development cost.**

# Evaluation Criteria: Cost

**Continuted**

- A simple **trade-off** can be made between compilation cost and execution speed of the compiled code.

- **Optimization** is the name given to the collection of techniques that compilers may use to decrease the size and/or increase the execution speed of the code (using concurrency) they produce.
- **Using refactoring the code, do concurrency on loop iterations** (whenever there is no dependency between iterations), etc.

# Evaluation Criteria: Others

- **Portability**
  - The ease with which programs can be moved from one implementation to another
  - Portability is most strongly influenced by the degree of **standardization** of the language.
  - The **applicability** to a wide range of applications
  - Java is a portable language
- **Generality**

  **--** the applicability to a wide range of applications

  -- Java is general purpose language
- **Well - definedness**

  **--** The **completeness and precision** of the language's official defining document is so important in evaluation of a language.

# A final note on evaluation criteria

- language design criteria are weighed differently from different perspectives.
- **Language designers** are likely to emphasize elegance and the ability to attract widespread use (generality, reliability, well defined ness, etc.). Involved with design issues
- **Language implementors** are concerned primarily with the difficulty of implementing the constructs and features of the language. The implementation should be free of any errors and bugs. Involvded with optimization techniques.

- **Language users** are worried about writability first and readability later.
- These characteristics often conflict with one another.

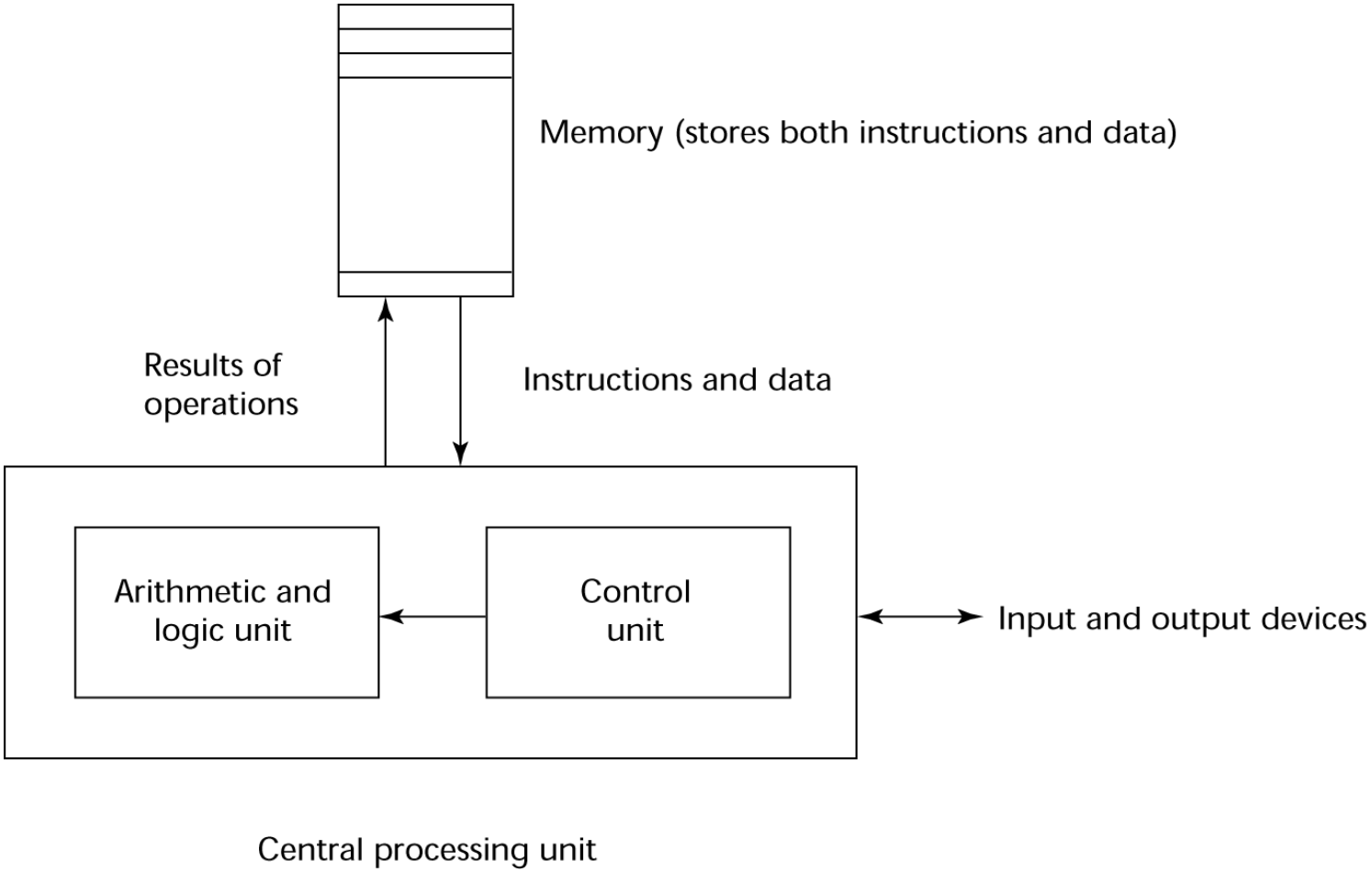
# Influences on Language Design

- **Computer Architecture**
  - Languages are developed around the prevalent eternal computer architecture, known as the ***von Neumann* architecture**
- **Program Design Methodologies** ( Software development)
  - New software development methodologies (e.g., object-oriented software development) led to new programming paradigms (new programming languages)

# Computer Architecture Influence

- Well-known computer architecture: Von Neumann
- Imperative languages and procedural languages, most dominant, inspired by von Neumann computers
  - Data and programs stored in memory
  - Memory is separate from CPU
  - Instructions and data are piped from memory to CPU registers
  - Basis for imperative languages
    - Variables model memory cells
    - Assignment statements model piping
    - Iteration is performed efficiently

# The von Neumann Architecture



# The von Neumann Architecture

- The execution of a machine code program on a von Neumann architecture computer occurs in a process called the **fetch-execute cycle**
- The **fetch-execute cycle** can be simply described by the following algorithm(on a von Neumann architecture computer)

```
initialize the program counter
repeat forever
   fetch the instruction pointed by the counter
   increment the counter
   decode the instruction
   execute the instruction
end repeat
```

# Programming Methodologies Influences

- 1950s and early 1960s: Focus on **Simple applications** ; worry about **machine efficiency**

  — The shift in the major cost of computing from hardware to software,

  — Increases in **programmer productivity**
- Late 1960s: programs were being written for large and **complex tasks**, such as controlling large petroleum-refining facilities and providing worldwide airline reservation systems.
- **Readability and better control structures** supported
  - **structured programming**
  - **top-down design** and step-wise refinement
- Late 1970s: A shift from **Process-oriented** to **data-oriented** began
  - **data abstraction (encapsulation)**
- Middle 1980s: **Object-oriented methodology** begins with *data abstraction*, which *encapsulates* processing with *data objects* and controls access to data, and adds *inheritance* and *dynamic method binding*.
  - *Data abstraction + inheritance + polymorphism*

# Language Categories

- Imperative & Object Oriented
  - Central features are variables, assignment statements, and iteration, procedures
  - Include languages that support object-oriented programming
  - Include scripting languages
  - Include the visual languages
  - C, Java, Perl, JavaScript, Visual BASIC .NET, C++
- Functional
  - Main means of making computations is by applying functions to given parameters
  - LISP, Scheme, ML, F#
- Logic
  - Rule-based (rules are specified in no particular order)
  - Prolog
- Markup/programming hybrid
  - Markup languages extended to making contents in web pages
  - XML(Extensible Markup Language), HTML, php
  - JSTL, XSLT

# Language Design Trade-Offs

- **The programming language evaluation criteria provide a framework for language design**. That framework unfortunately is **self-contradictory**. There are so many important but conflicting criteria, that their reconciliation and satisfaction is a major engineering task

- **Reliability vs. cost of execution**
  - Java demands all references to array elements be checked for proper indexing, which leads to increased execution costs
  - C does not require index range checking, so C programs execute faster than semantically equivalent Java programs, although Java programs are more reliable.
- **Readability vs. writability**
  APL provides many powerful operators (and a large number of new symbols), allowing complex computations to be written in a compact program but at the cost of poor readability

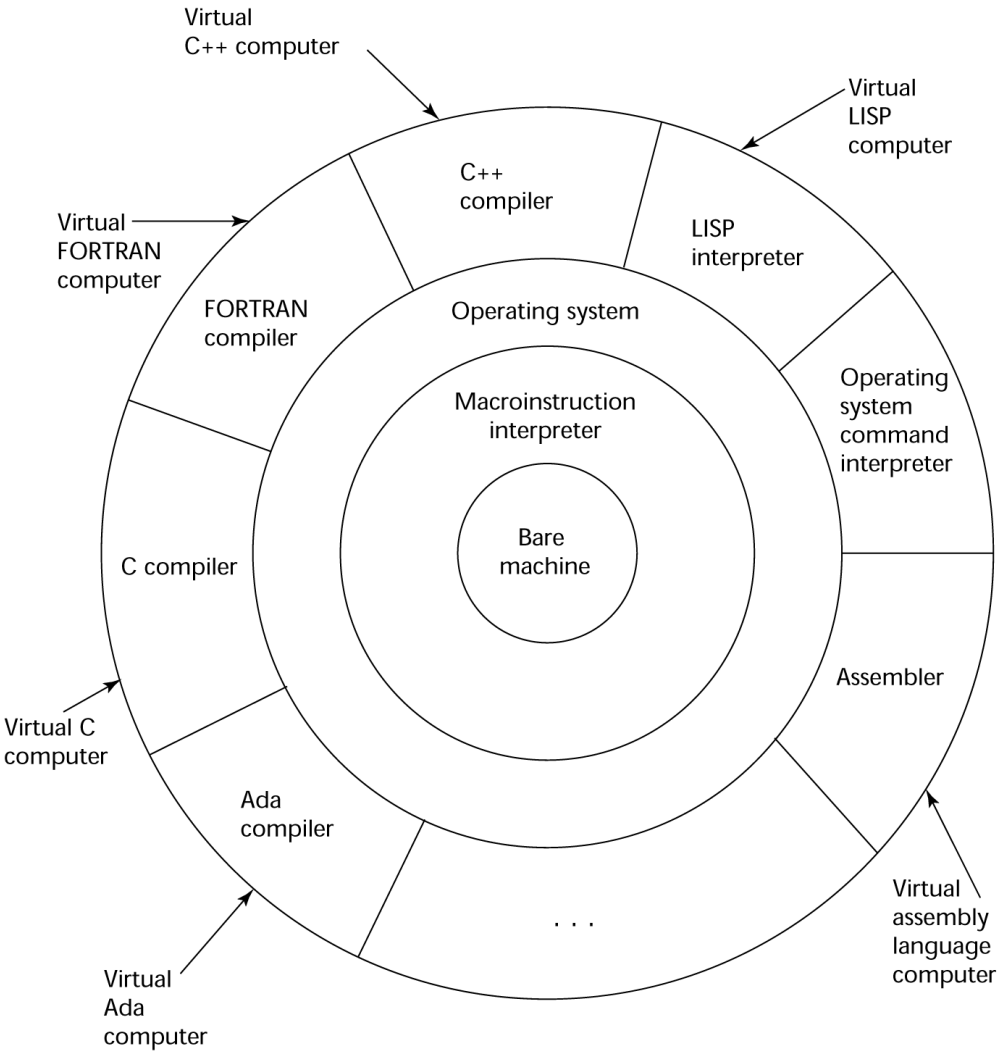
- **Writability vs. reliability**
  - C++ pointers are powerful and very flexible but are unreliable

# Implementation Methods

- The **machine language** of the computer is its set of instructions. In the absence of other supporting software, its own machine language is the only language that most hardware computers "understand."
- A **language implementation system** (compiler, interpreter) cannot be the only software on a computer. Also required is a large collection of programs, called the operating system, which supplies higher-level primitives than those of the machine language.
- The operating system and language implementations are layered over the machine language interface of a computer. These layers can be thought of as **virtual computers**, providing interfaces to the user at higher levels.
- For example, an operating system and a C compiler provide a virtual C computer
- **User programs** form another layer over the top of the layer of virtual computers
- The layered view of a computer is shown in Figure 1.2

# Layered View of Computer

The operating system and language implementation are layered over machine interface of a computer



# Implementation Methods

- **Compilation**
  - Programs are translated into machine language; includes JIT (Just In Time) systems
  - Use: Large commercial applications
  - Slow translation, fast execution
- **Pure Interpretation**
  - Programs are interpreted by another program known as an interpreter : fast translation, slow execution
  - Use: Small programs or when efficiency is not an issue
- **Hybrid Implementation Systems**
  - A compromise between compilers and pure interpreters
  - Use: Small and medium systems when efficiency is not the first concern

# Compilation (Compiler Implementation)

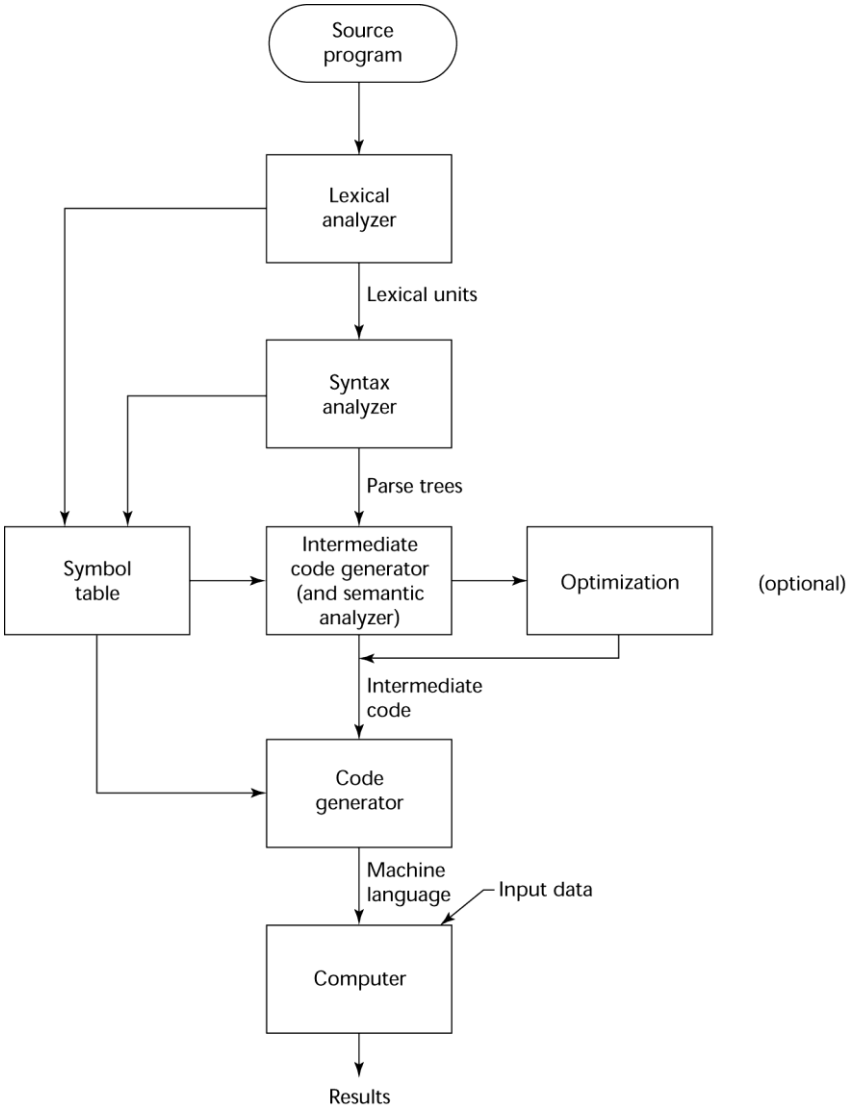
- Translate high-level program (source language) into machine code (machine language) which can be executed directly on the computer
- Slow translation, fast execution
- C, COBOL, C++, and Ada, are supported by compilers, or compiled based languages.

# Compilation (Compiler Implementation)

- **lexical analysis**: converts characters in the source program into lexical units (called token). The lexical units of a program are identifiers, special words, operators, and punctuation symbols.
- **syntax analysis**: The syntax analyzer takes the lexical units from the lexical analyzer and uses them to construct *hierarchical structures called parse trees. These parse trees represent the syntactic structure of the program*
- **Semantic analysis**: intermediate code generator and semantic analyzer *generate intermediate code*
- **Intermediate code generation**: The intermediate code generator produces a program (intermediate code) in a different language, called intermediate language which looks very much like assembly languages.
- **Semantic analyzer** is an integral part of the intermediate code generator. The semantic analyzer checks for errors, such as type errors, that are difficult, if not impossible, to detect during syntax analysis.
- **Code generation:** *machine code is generated* (using features of the machine)

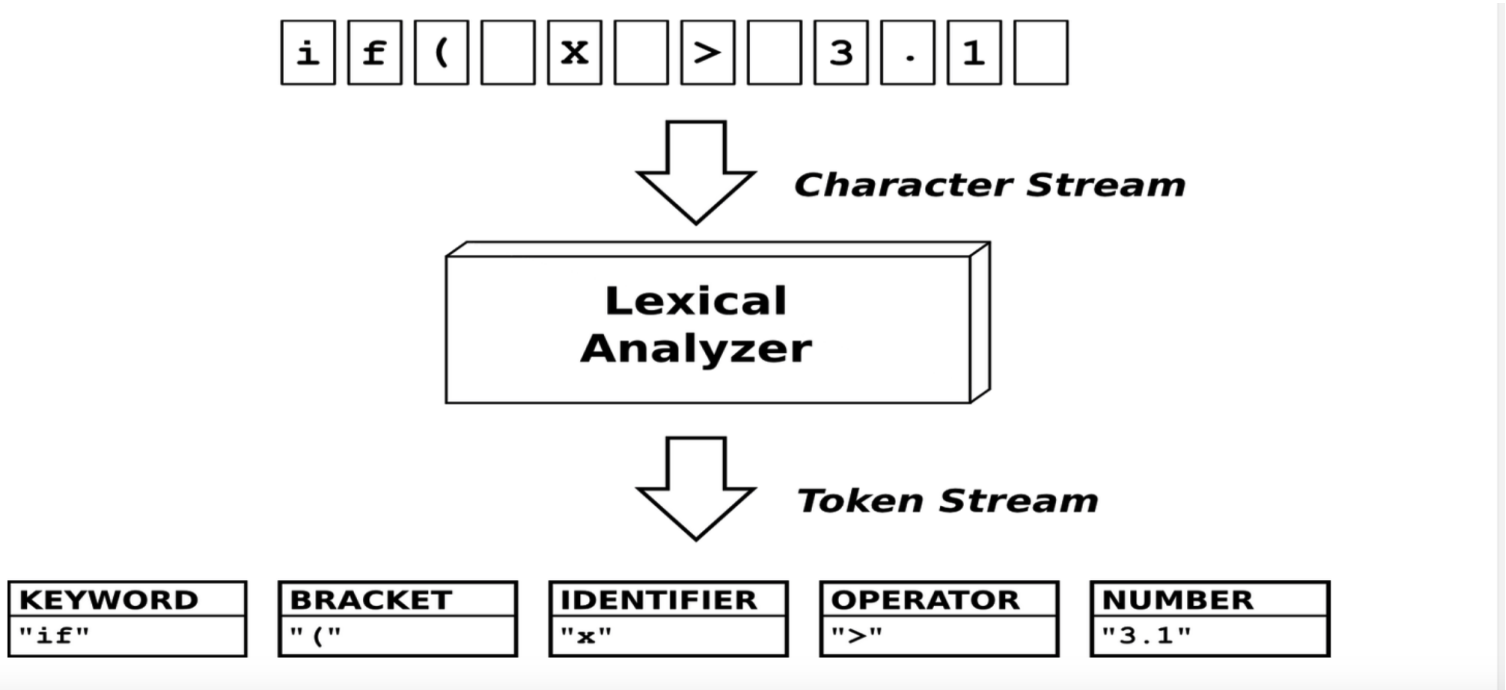
# The Compilation Process

## Figure 1.3



# Lexical Analyzer

- Lexical analysis is the process of breaking down the source code of the program into smaller parts, called **tokens**, such that a computer can easily understand.
- These tokens can be individual words or symbols in a sentence, such as keywords, variable names, numbers, and punctuation.
- It is also known as a **scanner**.
- The output generated from Lexical Analysis are a sequence of tokens sent to the parser for syntax analysis.



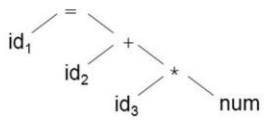
# Syntax Analyzer

## Example: Syntactic Analysis

- The tokens

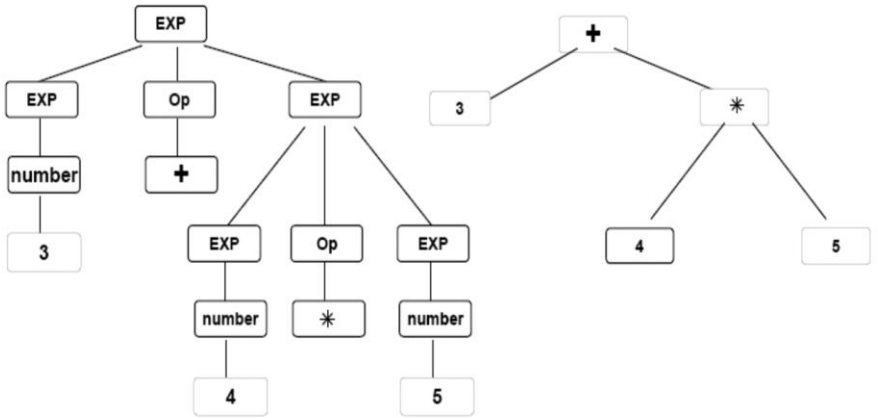
$$id_1 = id_2 + id_3 * num ;$$

may be represented by the following tree.




**Syntax Trees**

Parse trees are often converted into a simplified form known as a syntax tree that eliminates wasteful information from the parse tree.



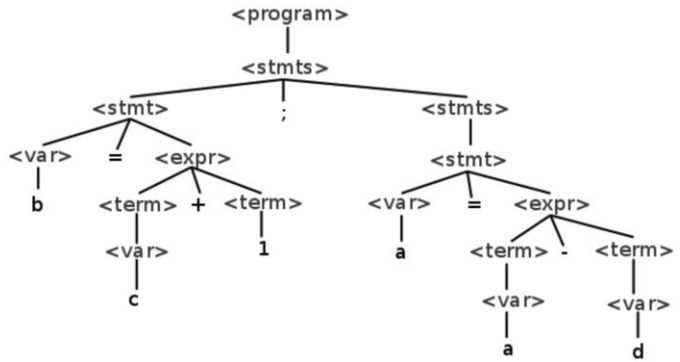

## Syntax Analysis

### Overview

Syntax analysis, or parsing, is the problem of translating code in language into a parse tree using a given grammar.

b = c + 1;
a = a - d

→



# Semantic Analyzer

- A semantic analyzer is a component within a compiler or natural language processing system that **analyzes the meaning of code** or text by examining the **relationships between words and phrases within a context**
- Ensuring that the syntax is not only **correct** but also makes **logical sense** based on the **language rules and data types** involved
- Essentially, it **interprets the meaning behind the code** or text beyond just its structure

**Semantic Errors:**

Errors recognized by semantic analyzer are as follows:

•Type mismatch

•Undeclared variables

•Reserved identifier misuse

# Semantic Analyzer

**Functions of Semantic Analysis:**

**1.Type Checking –**
Ensures that data types are used in a way consistent with their definition.

**2.Label Checking –**
A program should contain labels references.

**3.Flow Control Check –**
Keeps a check that control structures are used in a proper manner.

-- Example: no break statement outside a loop

# Additional Compilation Features

- **Optimization,** which improves programs (usually in their intermediate code version) by making them **smaller** or **faster** or both. It is often an optional part
- The **symbol table** serves as a database for the compilation process. The primary contents of the symbol table are the type and attribute information of each user-defined name in the program.
- The information is placed in the symbol table by the lexical and syntax analyzers and is used by the semantic analyzer and the code generator.
- Most user programs also require programs from the operating system (library functions). **Linking** operation (program) connects the user program to the system programs
- **Linking and loading**: The process of collecting system program units (library functions) and linking them to a user program
- The user and system code together are sometimes called a **load module,** or **executable image**.

# Von Neumann Bottleneck

- Connection speed between a computer's memory and its processor determines the speed of a computer

- **The connection speed (**access time to data in main memory**)** results in a ***bottleneck.*** This is known as the ***von Neumann bottleneck***; it is the primary limiting factor in the speed of computers.

- Using **Cache memory** alleviates this problem.

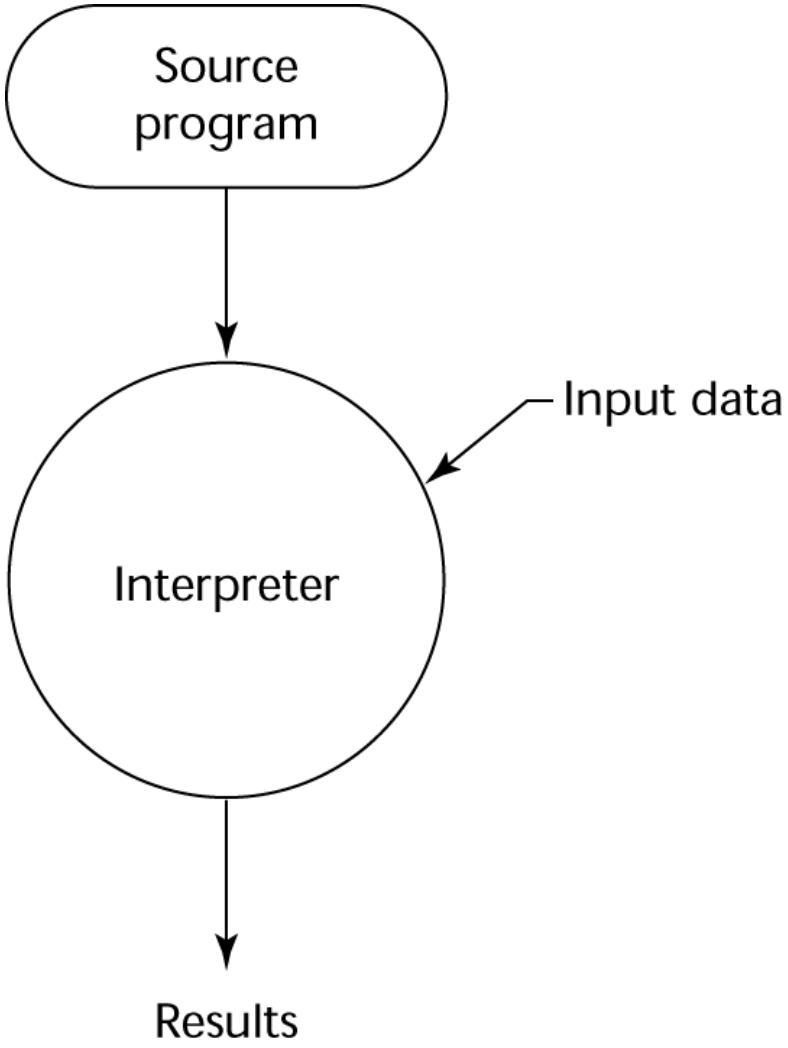
- Program instructions often can be executed much faster than the speed of the connection.

- The von Neumann bottleneck has been one of the primary motivations for the research and development of concurrency.

# Pure Interpretation

- Using pure interpretation, programs are interpreted (translated) by another program called an **interpreter.**
- An interpreter translates source code into object code, and then executes the object code immediately
- The interpreter program acts as a **software simulation** of a machine whose fetch-execute cycle deals with high-level language program statements rather than machine instructions.
- This software simulation obviously provides a **virtual machine** for the language. It mimics (learns) the behavior of a machine, how it executes an instruction.
- Pure interpretation has the advantage of allowing easy implementation of many source-level debugging operations, because all run-time error messages can refer to source-level units
- Interpreted code can run slower than compiled code (10 to 100 times slower)
- Often requires **more space**. In addition to the source program, the symbol table must be present during interpretation
- In recent years, pure interpretation has made a significant comeback with some general purpose and Web scripting languages, such as Java, Ruby, JavaScript and php.

# Pure Interpretation Process



# Hybrid Implementation Systems
## Java compiler

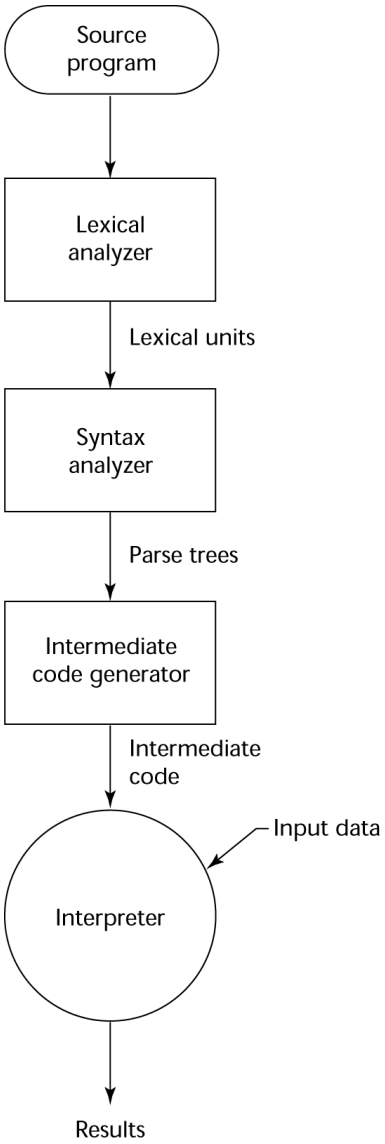
- A compromise between compilers and pure interpreters
- A high-level language program is translated to an intermediate language that allows easy interpretation
- Faster than pure interpretation, because the whole source language statements (program) are decoded into an intermediate code only once by compilation, and then the intermediated code interpreted to machine code.
- **Initial implementations of Java were hybrid;** the intermediate code or ***bytecode*** is changed (translated) to machine code. This is done by a run time system called ***Java Virtual Machine (JVM)***. This provides portability to any machine that has a byte code interpreter and a run-time system.
- **JVM** reads and executes the bytecode line by line. Every time a method is invoked, the JVM re-interprets the bytecode, which can be slow since the same method may be re-executed multiple times.

# Hybrid Implementation Systems

- Sometimes an implementor may provide both compiled and interpreted implementations for a language. In these cases, the interpreter is used to develop and debug programs. Then, after a bug-free state is reached, the programs are compiled to increase their execution speed.

# Hybrid Implementation Process

## Figure 1.5

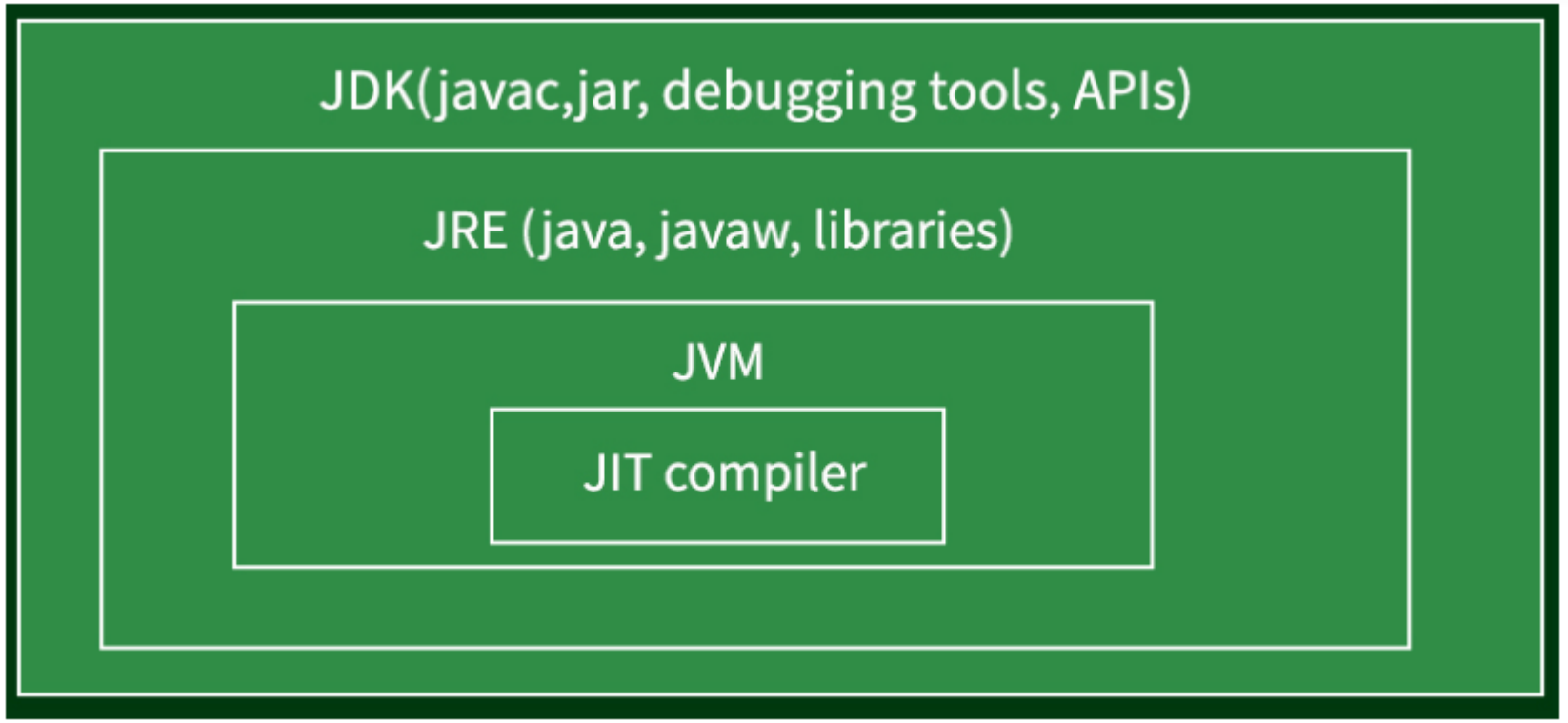


# Just-in-Time Implementation Systems

- The Just-In-Time (JIT) compiler in Java is a key component of the Java Runtime Environment (JRE) that enhances the performance of Java applications.
- The JVM identifying methods or code blocks that are executed frequently. This blocks called hot spot
- Once a hot spot is detected, the JIT compiler translates the corresponding bytecode into native (target) machine code. The generated native code is stored in a **cache for future use**.
- When the JVM encounters the same hot spot again, it directly executes the cached native code instead of interpreting the bytecode, resulting in significant performance improvements.
- **So, JIT compilation optimizes performance by reducing the overhead of bytecode interpretation for frequently executed code**.

# Just-in-Time Implementation Systems

- JIT compilation is a form of dynamic compilation ( run-time compilation)
- **JIT is an integral component of JVM**. It supports the JVM to execute Java byte code faster and improves the performance of Java application programs at run time.
- JIT systems are widely used for Java programs
- .NET languages are implemented with a JIT system



# Preprocessors & Macro Instruction

- A macro, or “macro instruction”, is a series of instructions in code that when executed performs repetitive tasks automatically.
- Macros are used extensively in programming and gaming.
- Macros provide a convenient way to substitute code snippets or values throughout your program, **improving both readability and maintainability.**
- A macro in C is essentially a piece of code or a value that is associated with an identifier. In C, the identifier of a macro is the name you give to the macro when you define it by #define directive.
- **#define** and **#include** are directives that declare a macro**.**
- #define PI 3.14159, PI is the identifier of the macro

# Preprocessors

- Preprocessor which is a program, manipulates the text of a source code file and performs several preprocessing operations before it is compiled by the compiler.
- Preprocessor instructions are embedded in programs.
- The preprocessor is essentially a macro expander.
- Preprocessor instructions are commonly used to specify the code from another file is to be included
- The C preprocessor instruction is called directive. Such as **#include, #define**
- **#include "myLib.h"** causes the preprocessor to copy the contents of **myLib.h** into the program at the position of the #include.

# Programming Environments or Integrated Development Environment (IDE)

- A collection of tools used in software development
- **UNIX**
  - An older operating system and tool collection
  - Nowadays often used through a GUI (e.g., CDE, KDE, or GNOME) that runs on top of UNIX
- **Borland JBuilder** is a programming environment that provides an *integrated editor*, *compiler*, *debugger*, *and file system for Java* development, where all four are accessed through a graphical interface.
- **Visual Studio .NET**
  - A large, complex visual environment
  - Used to build Web applications and non-Web applications in any .NET language
  - .NET languages: C#, Visual BASIC, JScript (Microsoft's version of JavaScript), F# (a functional language), C++/CLI, Open MP, and MPI for parallel programmin
- **NetBeans**
  - Related to Visual Studio .NET, except for applications in Java

# Summary

- The study of programming languages is valuable for a number of reasons:
  - Enable us to choose languages more intelligently
  - Increase our capacity to use different constructs
  - Makes learning new languages easier
  - To be able to design new languages

- Most important criteria for evaluating programming languages include:
  **Readability, writability, reliability, cost**

- Major influences on language design have been:
  **machine architecture and software development methodologies**

- The major methods of implementing programming languages are:
  **compilation, pure interpretation, and hybrid implementation**