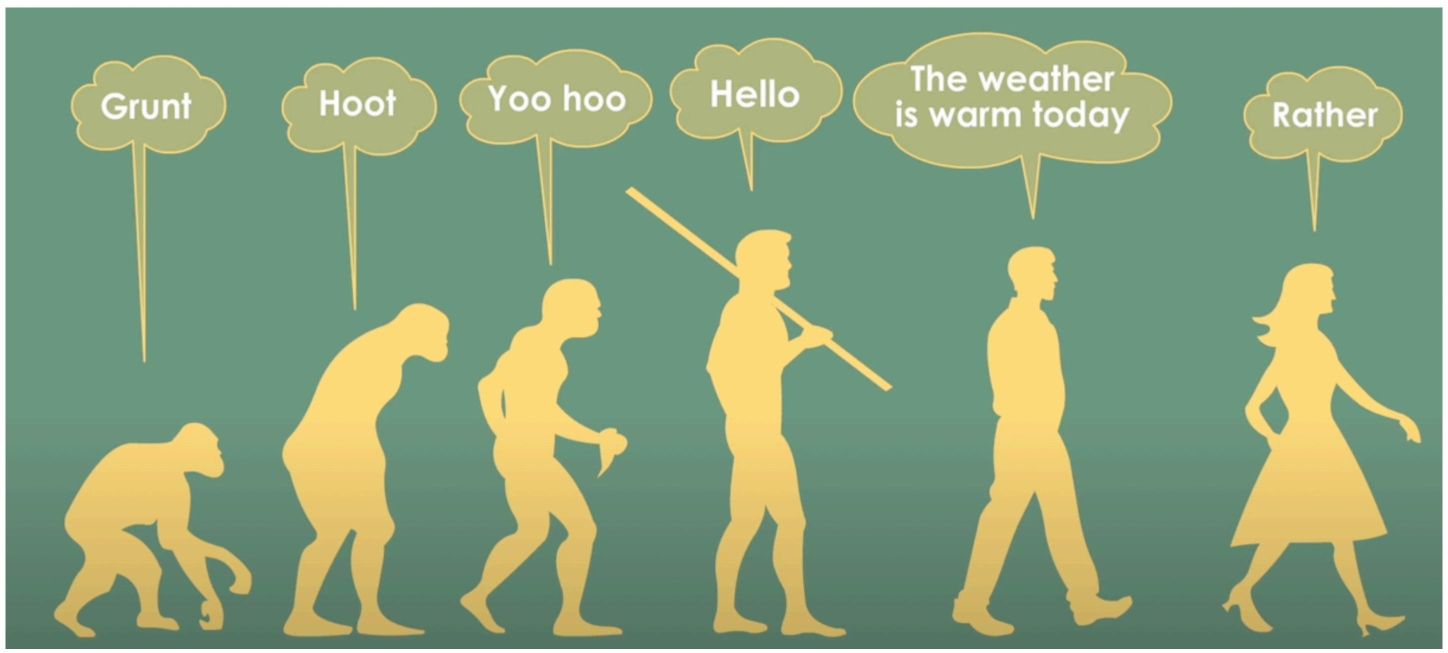
# Programming Paradigms: An Introduction

Summer Semester 2023

Dr. Abhishek Tiwari, Prof. Dr. Christian Hammer

- Schedule
  - Lecture: Tuesday 12-14,
  - Recitation (As required): Tuesday 14-16, 16-18, Thursday 14-16
- Office Hours
  - arranged via email

# Natural Languages




# Programming Languages (PLs)

- Enables humans to convey their ideas/algorithms to computers
- Various forms of requirements — various categories of PLs


https://www.watelectronics.com/types-of-programming-languages-with-differences/

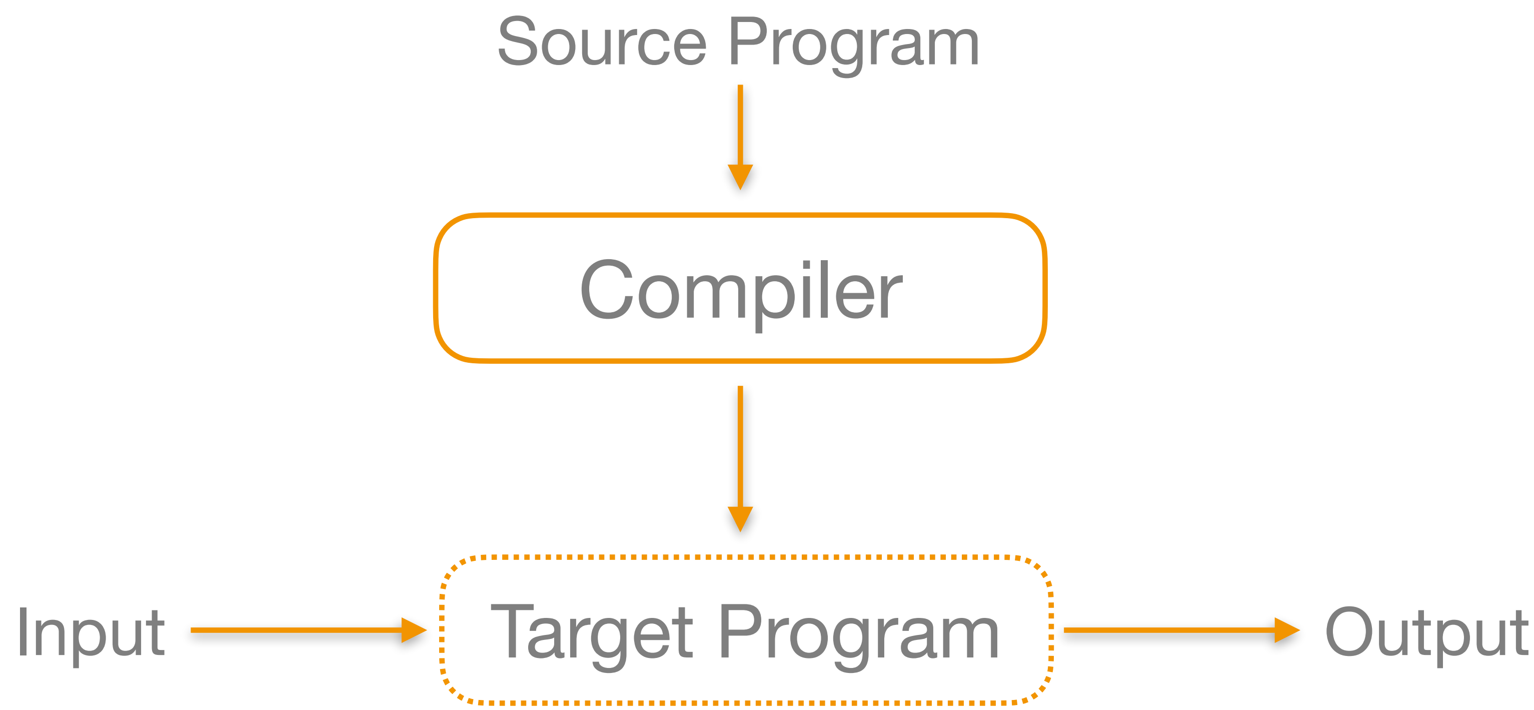
- First computers were huge in size
  - filled several rooms
  - costed millions of dollars (in 1940s)
- Programmers believed that computer's time was more valuable
- Programs were written in machine language — sequence of bits
- Greatest common divisor of two integer in machine language

```
55 89 e5 53   83 ec 04 83   e4 10 e8 31   00 00 00 89   c3 e8 2a 00
00 00 39 C3   74 10 8d b6   00 00 00 00   39 c3 7e 13   29 c3 39 c3
75 16 89 1c   24 e8 6e 00   00 00 8b 5d   fc c9 c3 29   d8 eb eb 90
```

- Scalability and correctness issues for large programs
- Assembly language was invented
  - operations to be expressed with mnemonic
- Machine-centric development
- Developer still think in terms of machine-level instructions
- Frustrating to have to rewrite programs for every new machine

- Desirable features
  - machine independent
  - computation more resemble mathematical formulas
- Idea is to translate a language to assembly or machine language — compilers
- Fortran was designed considering such features
  - also considered as (arguably) the first high-level language

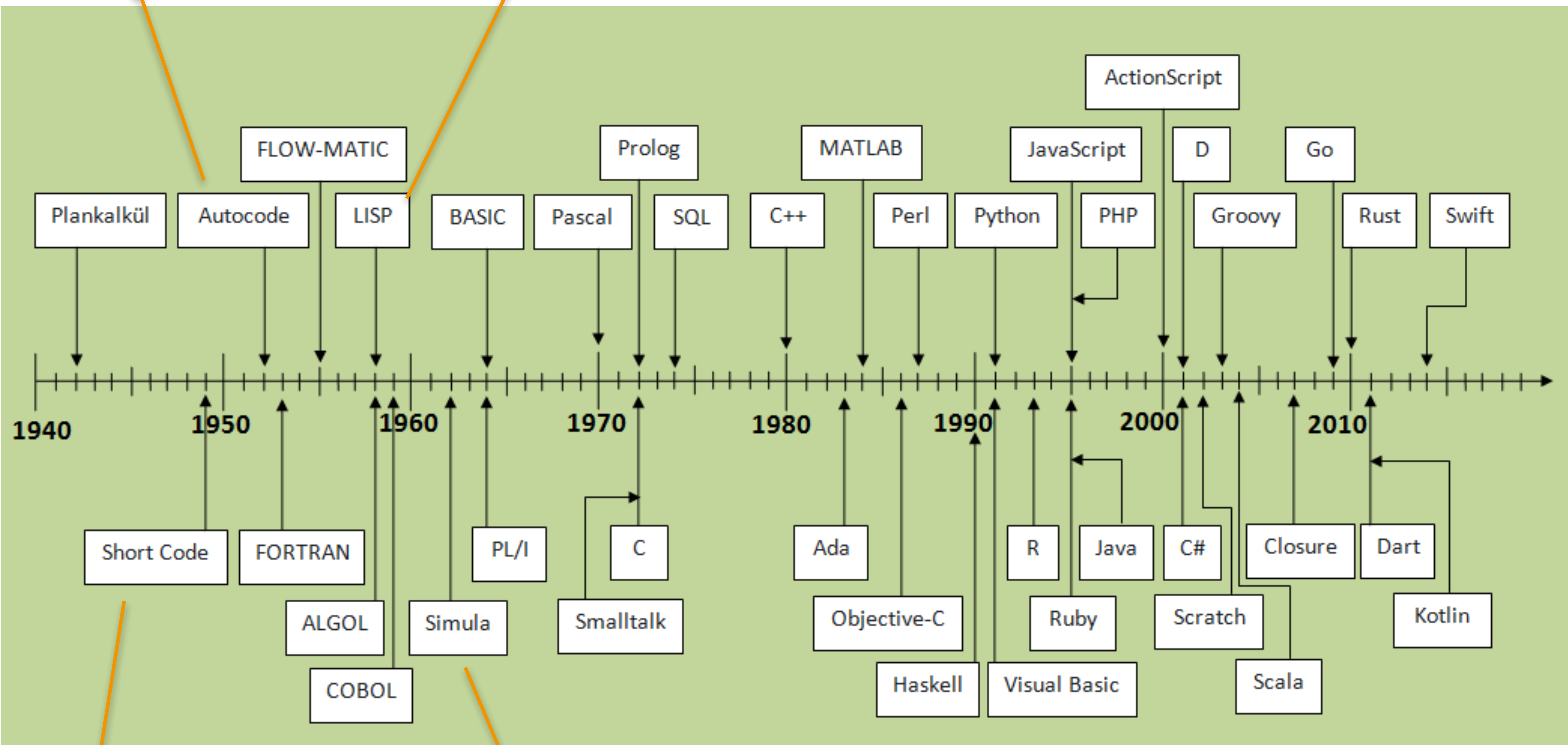
# *Compilation and Interpretation*




# *History of Programming Languages*

first compiled PL

introduced essential concepts such as tree data structure, dynamic typing, recursion etc




implemented on a computer

first object oriented PL

https://javaconceptoftheday.com/history-of-programming-languages/

# History of Programming Languages

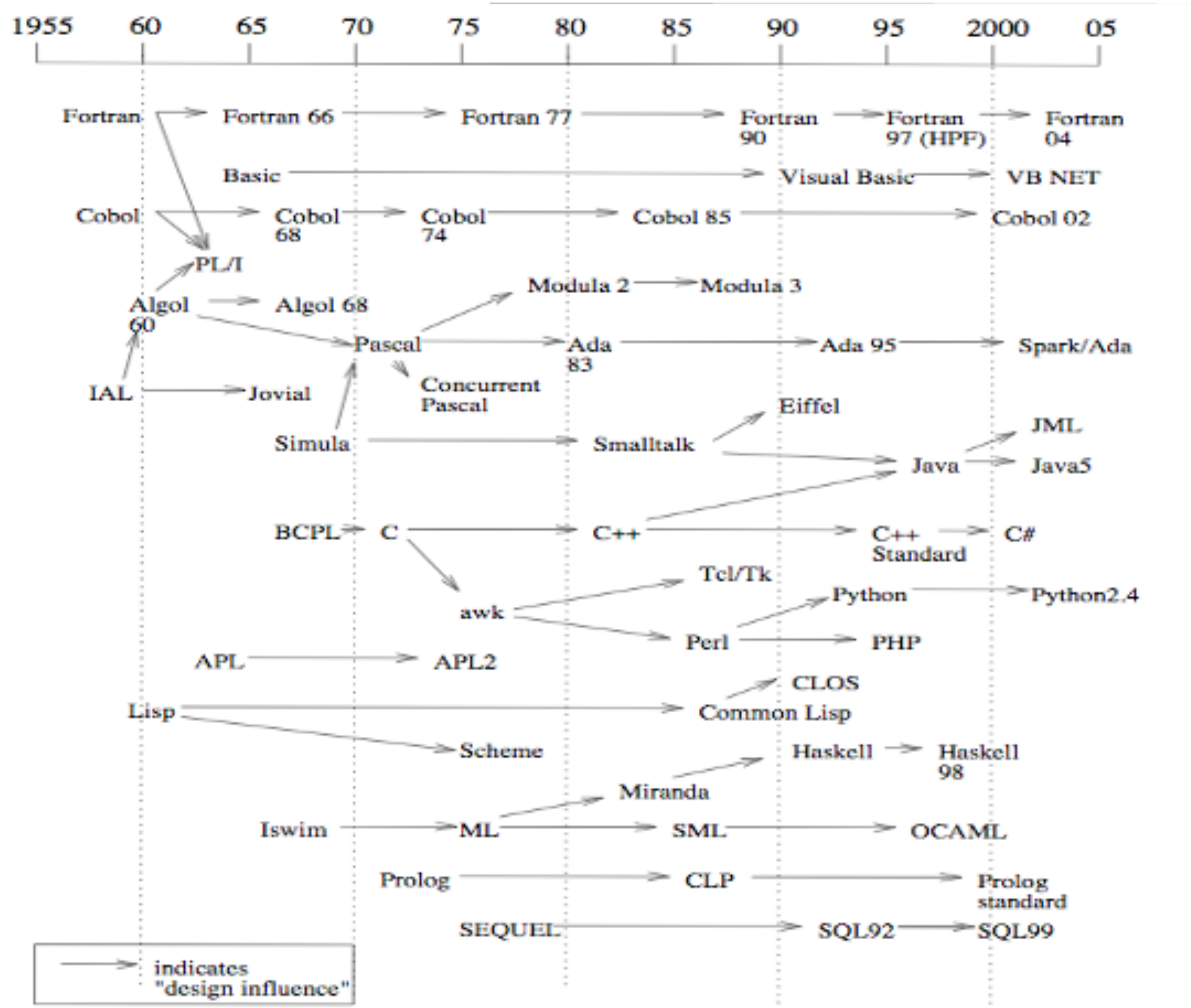



Figure 1.2: A Snapshot of Programming Language History

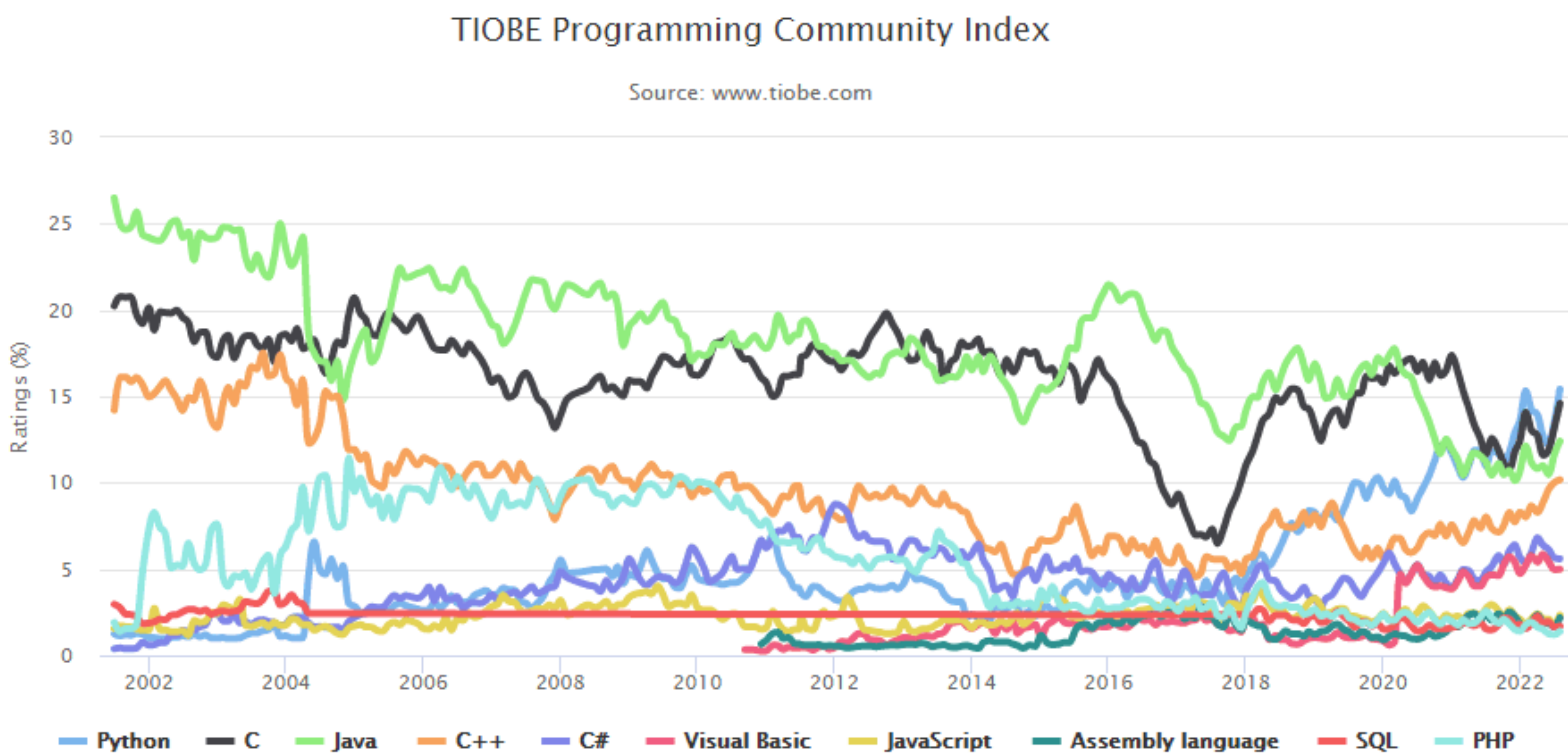
# *Why Take this Course?*

- This course will enable you to
  - understand the goals and objectives of different PLs
  - understand the differences between workings of different PLs
  - choose effective PLs for specific tasks
- This course is not about
  - one particular programming language
  - semantics of one or two PLs

# Learning Objectives

- Understand the workings of programming languages
- Learn to classify programming languages based on their features
- Understand the language creation process
- Identify the language choices based on various scenarios

# Widely Used Programming Languages by Years




https://de.wikipedia.org/wiki/TIOBE-Index

- Why not use one programming language?
- Why does the community adds new PLs regularly?

- Programming languages have four properties
  - Syntax
  - Names
  - Types
  - Semantics
- For any languages
  - Its designer must define these properties
  - Its programmers must master these properties

- 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?

# Example: A Rudimentary imperative language

- expression $e ::= \quad x \mid n \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1/e_2 \mid e_1 * e_2$

  Commands $c ::= \quad x := e \mid x := input() \mid$

  $skip \mid$

  $\text{if } e \text{ then } c_1 \text{ else } c_2 \mid$

  $\text{while } e \text{ do } c \mid$

  $c_1; c_2$

- Various kinds of entities in a program have names:
  - variables, types, functions, parameters, classes…
- Named entities are bound in a running program to:
  - Scope
  - Visibility
  - Type
  - Lifetime

- 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

- The meaning of a program is called its semantics
- Some questions that semantics addresses:
    - During a program's execution, 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?

- Formal description of a *program execution*
- A program $c$ is executed under a memory $\mu$, which maps identifiers to values
- Expressions are evaluated atomically, letting $\mu(e)$ denote the value of $e$ in memory $\mu$
- The structural operational semantics is defined in terms of a transition relation between configurations. A configuration is either a pair $(c, \mu)$ or a memory $\mu$ (yielded by programs finishing their computation)

# Example: Operational Semantics

$$\textsc{Update}\quad \frac{x \in dom(\mu)}{(x := e, \mu) \to \mu[x \mapsto \mu(e)]}$$

Memory update

$$\textsc{No-Op}\quad (\mathsf{skip}, \mu) \to \mu$$

$$\textsc{Branch True}\quad \frac{\mu(e) \neq 0}{(\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2, \mu) \to (c_1, \mu)}$$

$$\textsc{Branch False}\quad \frac{\mu(e) = 0}{(\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2, \mu) \to (c_2, \mu)}$$

$$\textsc{Loop True}\quad \frac{\mu(e) \neq 0}{(\mathsf{while}\ e\ \mathsf{do}\ c, \mu) \to (c; \mathsf{while}\ e\ \mathsf{do}\ c, \mu)}$$

$$\textsc{Loop False}\quad \frac{\mu(e) = 0}{(\mathsf{while}\ e\ \mathsf{do}\ c, \mu) \to \mu}$$

$$\textsc{Sequence 1}\quad \frac{(c_1, \mu) \to \mu'}{(c_1; c_2, \mu) \to (c_2, \mu')}$$

$$\textsc{Sequence}\quad \frac{(c_1, \mu) \to (c_1', \mu')}{(c_1; c_2, \mu) \to (c_1'; c_2, \mu')}$$

- Paradigms are a way to classify programming languages based on their features
- Each paradigm consists of certain structures, features, and opinions about how common programming problems should be tackled
- Four main types:
  - Imperative
  - Object-oriented
  - Functional
  - Logic

- Follows the classic von Neumann-Eckert model:
  - Programs and data are indistinguishable in memory
  - Program = a sequence of commands
  - State = values of all variables when the program runs
  - Large programs use procedural abstraction
- Example imperative languages:
  - Cobol, Fortran, C, Ada, Perl, …

# The von Neumann-Eckert Model

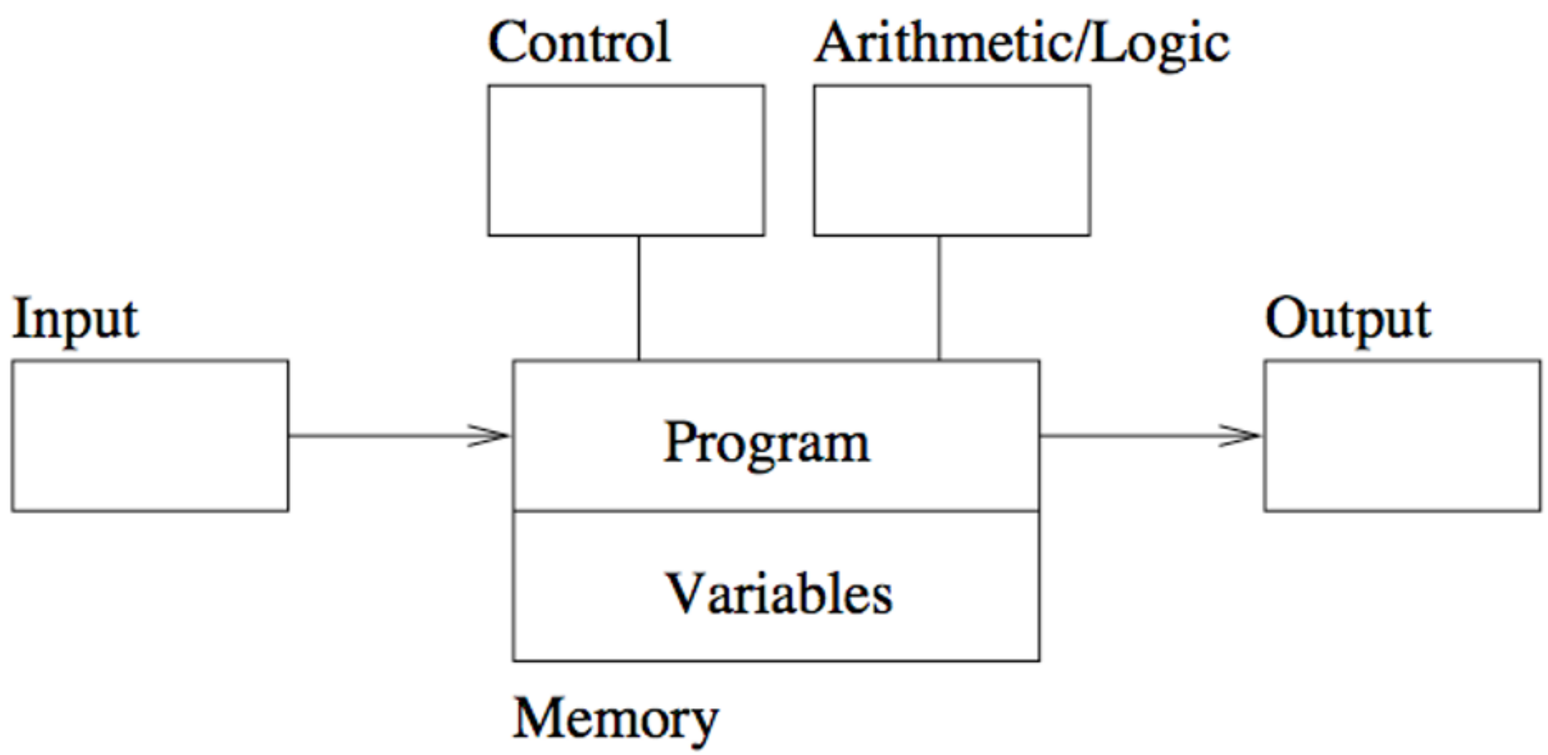



Figure 1.1: The von Neumann-Eckert Computer Model

```c
int search(int array[], int size, int x)
{
    int index;
    for (index = 0; index < size; index++)
    {
        if (array[index] == x)
            return index;
    }
    return -1;
}
```

Control-flow guiding statements

- A collection of objects containing data and code
  - data as properties
  - code as methods
- Some popular features:
  - Inheritance, Data abstraction, Encapsulation, polymorphism ..

- 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, …

```
data List a = Nil | Cons a (List a) deriving (Show, Eq)

search :: (Eq a) => a -> List a -> Bool
search n Nil = False
search n (Cons n' l') = (n' == n) || search n l'
```

Mathematical Relation between I/O and O/P

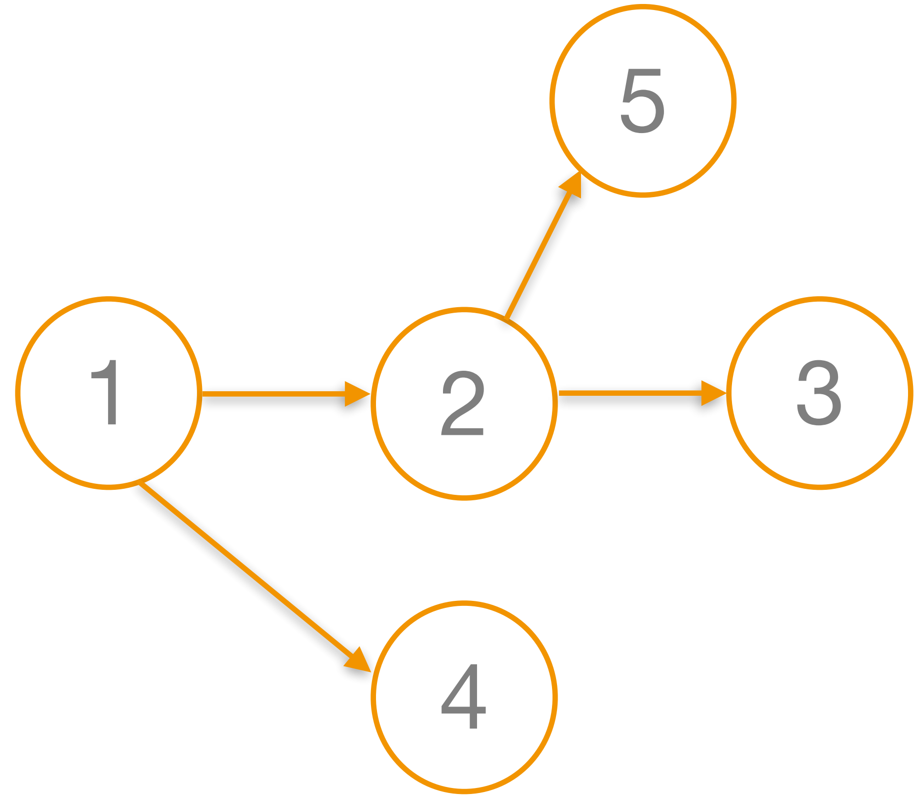
```
l = Cons 1 (Cons 2 (Cons 3 Nil))
search 5 , l

search 5 , Cons 1 (Cons 2 (Cons 3 Nil))
1 == 5 || search 5 (Cons 2 (Cons 3 Nil))
   .
   .
   .
search 5 Nil = False
```

- Based on formal logic
- Logic programming declares what outcome the program should accomplish, rather than how it should be accomplished
- Attributes of logic programming:
    - Programs as sets of constraints on a problem
    - Programs that achieve all possible solutions
    - Programs that are nondeterministic
- Example logic programming languages:
    - Prolog, Datalog, …

# Reachability in Prolog

Facts

```
edge(1,2)
edge(2,3)
edge(1,4)
edge(2,5)
```




```
reach(x, y) = edge(x, y)
reach(x,z) = edge(x,y) reach(y,z)
```

Rules

$$\dfrac{\dfrac{\texttt{edge(2,5)}}{\texttt{edge(1,2)}\quad\texttt{reach(2, 5)}}}{\texttt{reach(1,5)}}$$

- Design Constraints
  - Computer architecture
  - Technical setting
  - Standards
  - Legacy systems
- Design Outcomes and Goals

- 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

- A language element is bound to a property at the time that property is defined for it
  - a binding is the association between an object and a property of that object, e.g., a variable and its type 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 same on the different platforms
  - Type errors are detected
  - Semantics errors are properly trapped
  - Memory leaks are prevented

- Accesible compilers/interpreters
- Good texts and tutorials
- Wide community of users
- Integration with 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
- Tradeoff with efficiency

# *Efficient Implementation*

- Embedded systems
  - Real-time responsiveness (e.g., navigation)
- Web applications
  - Responsiveness to users (e.g., Google Search)
- Corporate database applications
  - Efficient search and updating
- AI applications
  - Modeling human behaviors

- Homework will be posted when required
- Two presentations - One at the mid of the semester and one at the end
- Contribute to 50% of total points
- Must be passed

- An individual project at the end of the semester
- A project report — summary of the techniques used, simple code documentation, etc.
- Contribute to 50% of total points
- Must be passed

- No book contains all resources
- Some lectures are created referencing the following books:
  - Programming Languages: Principles and Paradigms by Allen B. Tucker and Robert E. Noonan
  - Programming Language Pragmatics by Micheal L. Scott
- Additional material, such as research papers, will be suggested during the lectures