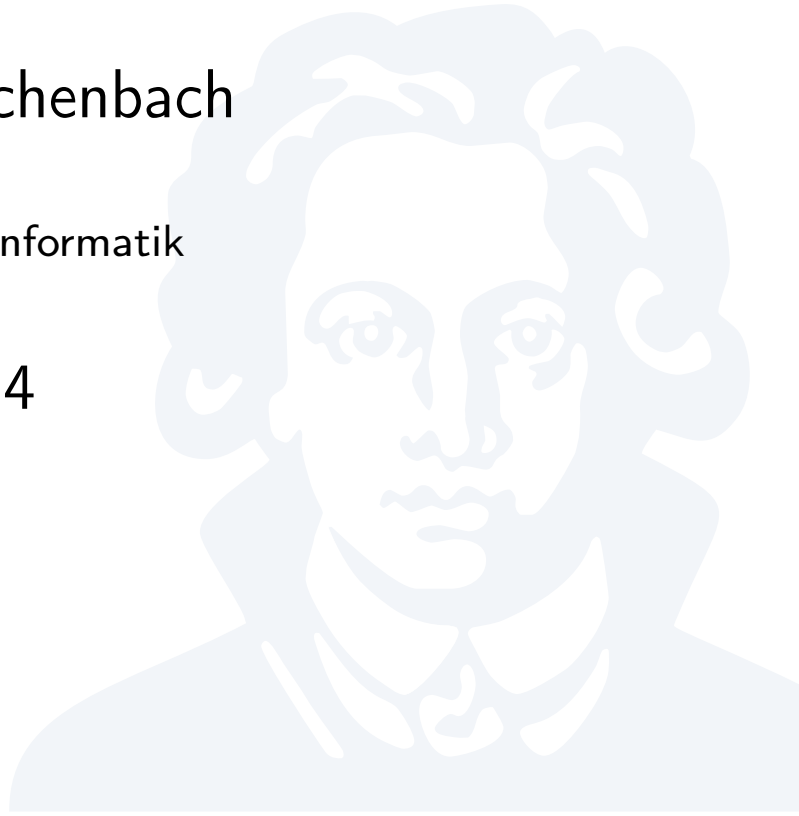
Foundations of Programming Languages

Overview: Executing High-Level Languages

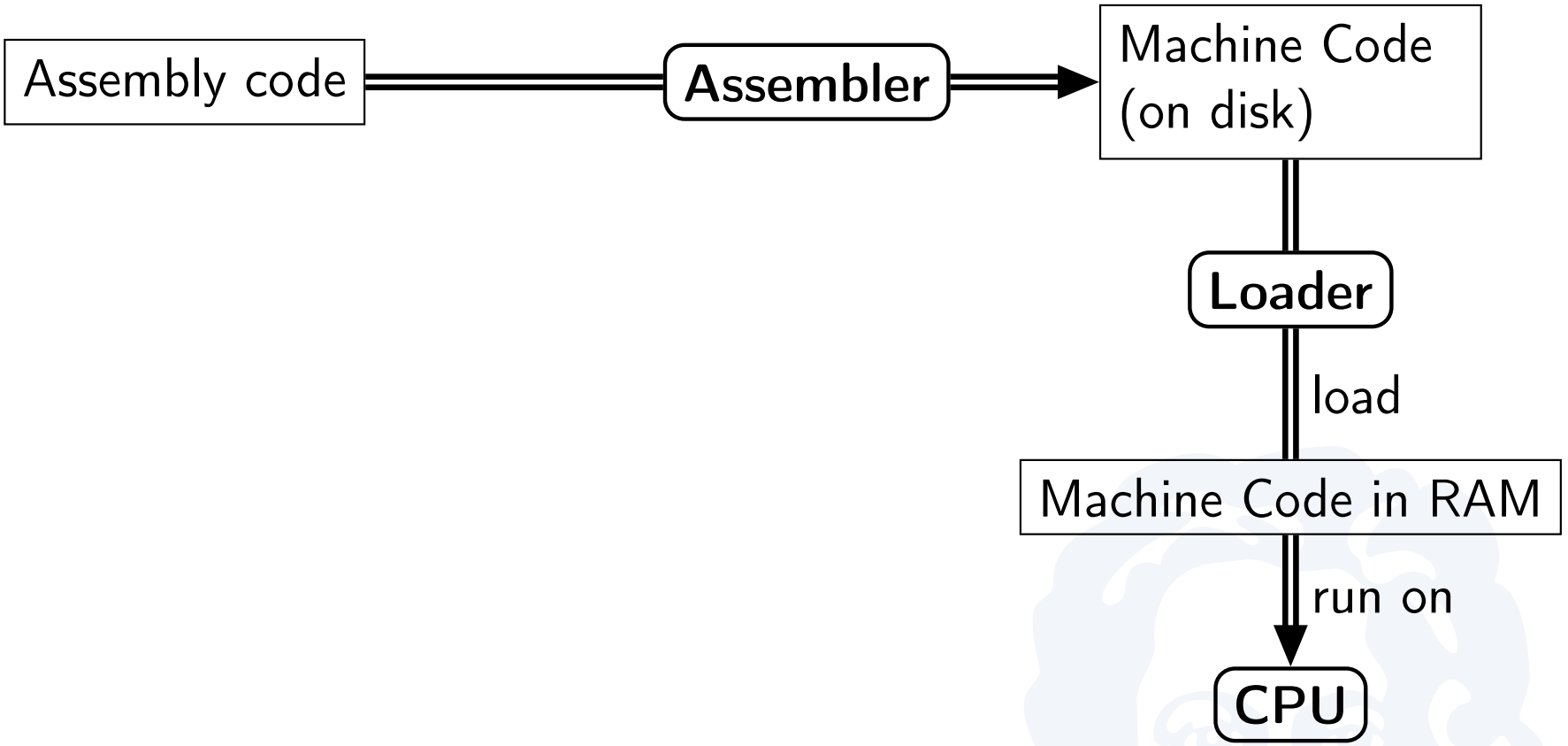
Prof. Dr. Christoph Reichenbach

Fachbereich 12 / Institut für Informatik

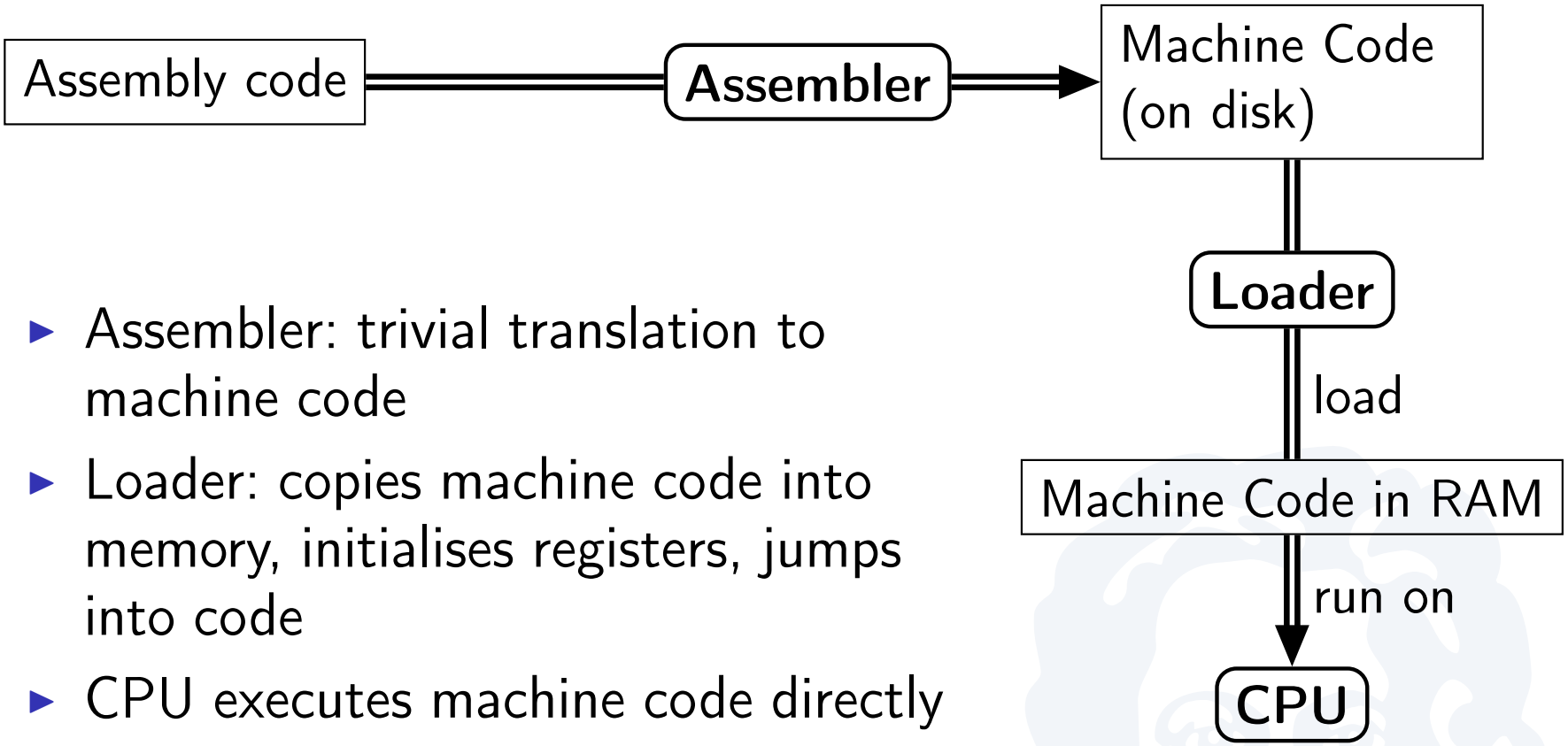
17. Oktober 2014



Program Execution

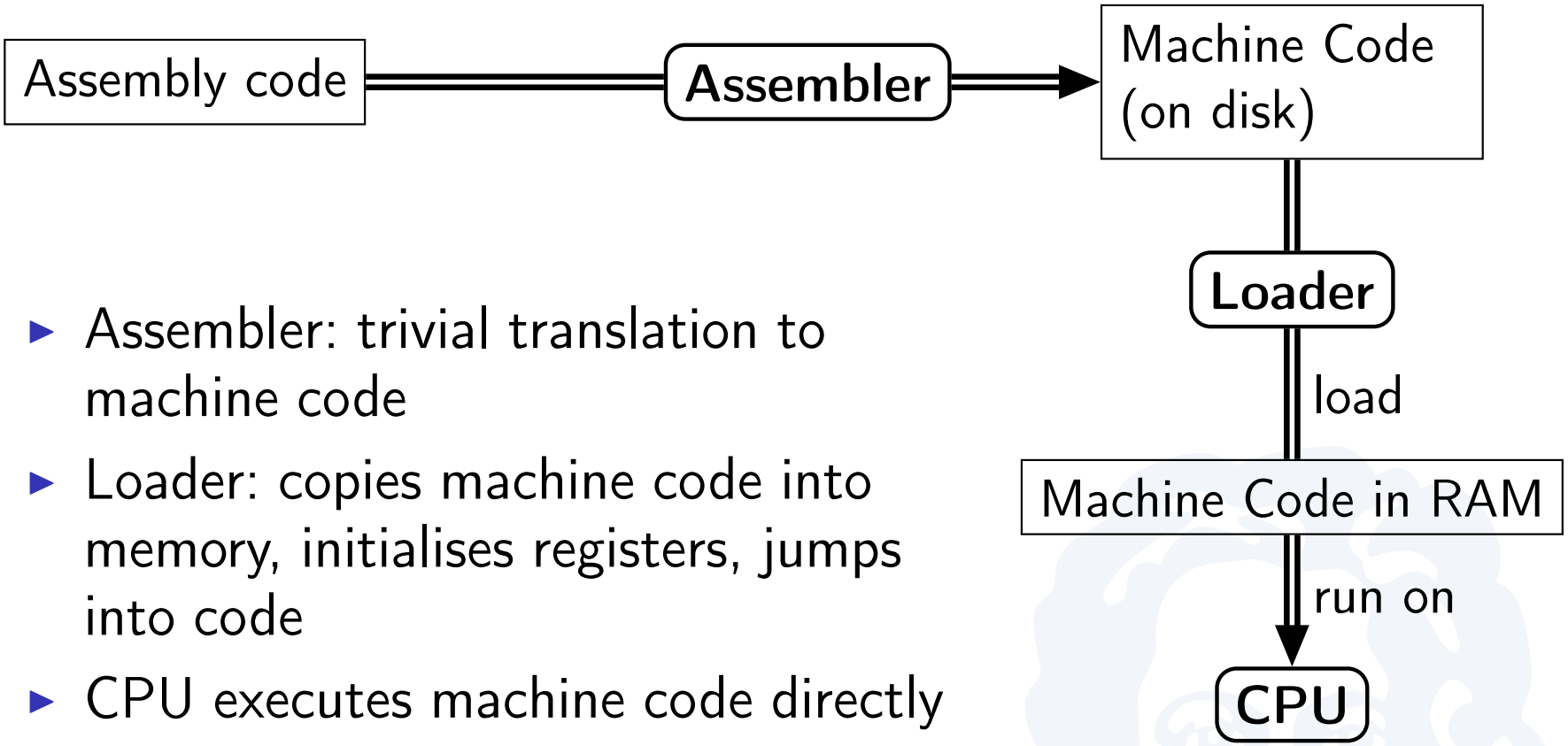


Program Execution



- Assembler: trivial translation to machine code
- Loader: copies machine code into memory, initialises registers, jumps into code
- CPU executes machine code directly

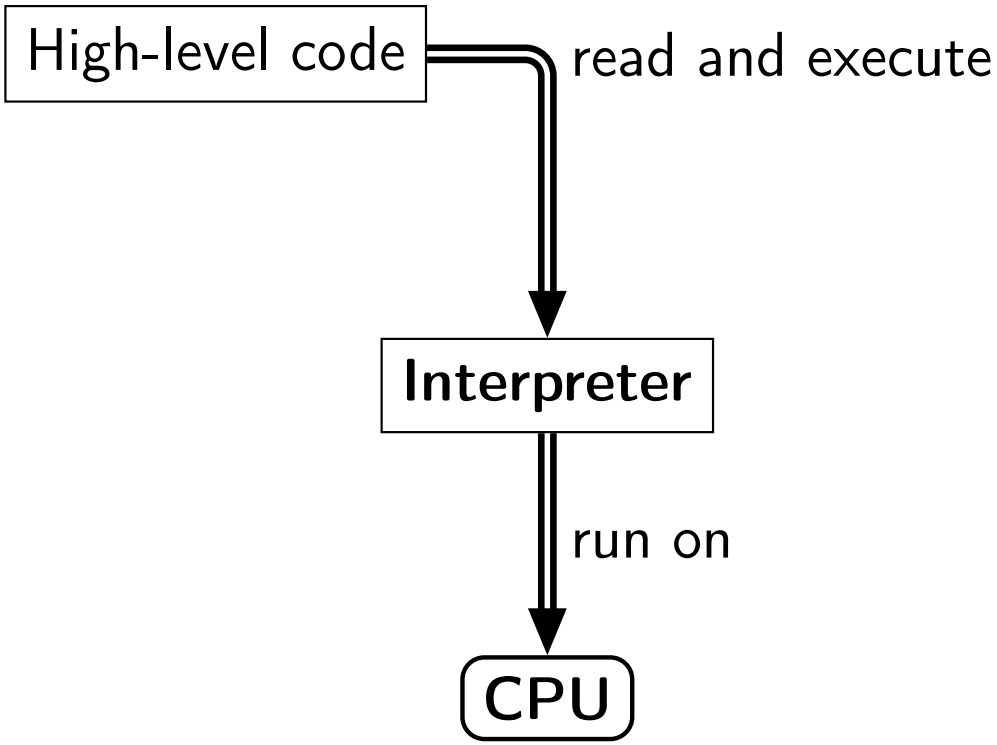
Program Execution



- Assembler: trivial translation to machine code
- Loader: copies machine code into memory, initialises registers, jumps into code
- CPU executes machine code directly

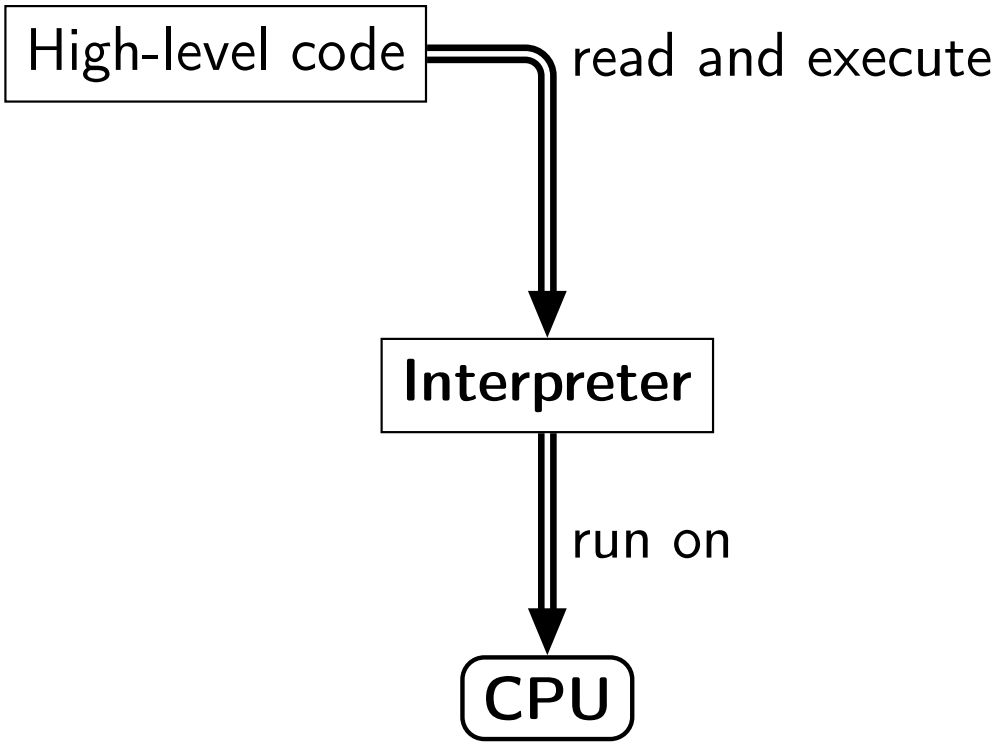
How about languages that the CPU can't execute directly?

Interpretation



- Interpreter reads high-level code, then alternates:
  - Figure out next command
  - Execute command
- May directly encode operational semantics

Interpretation



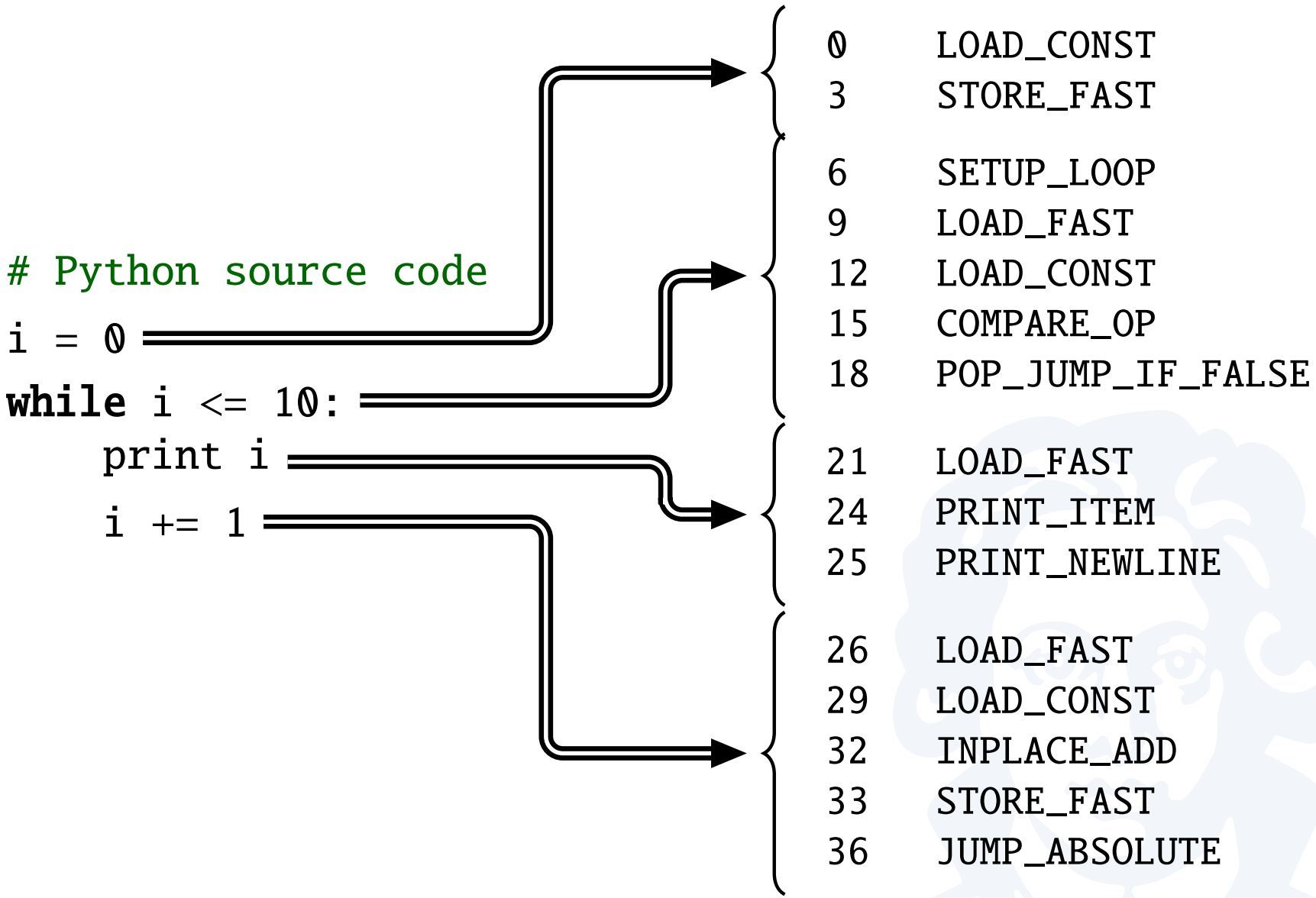
- Interpreter reads high-level code, then alternates:
  - Figure out next command
  - Execute command
- May directly encode operational semantics

Examples: Python, Perl, Ruby, Bash, AWK, ...

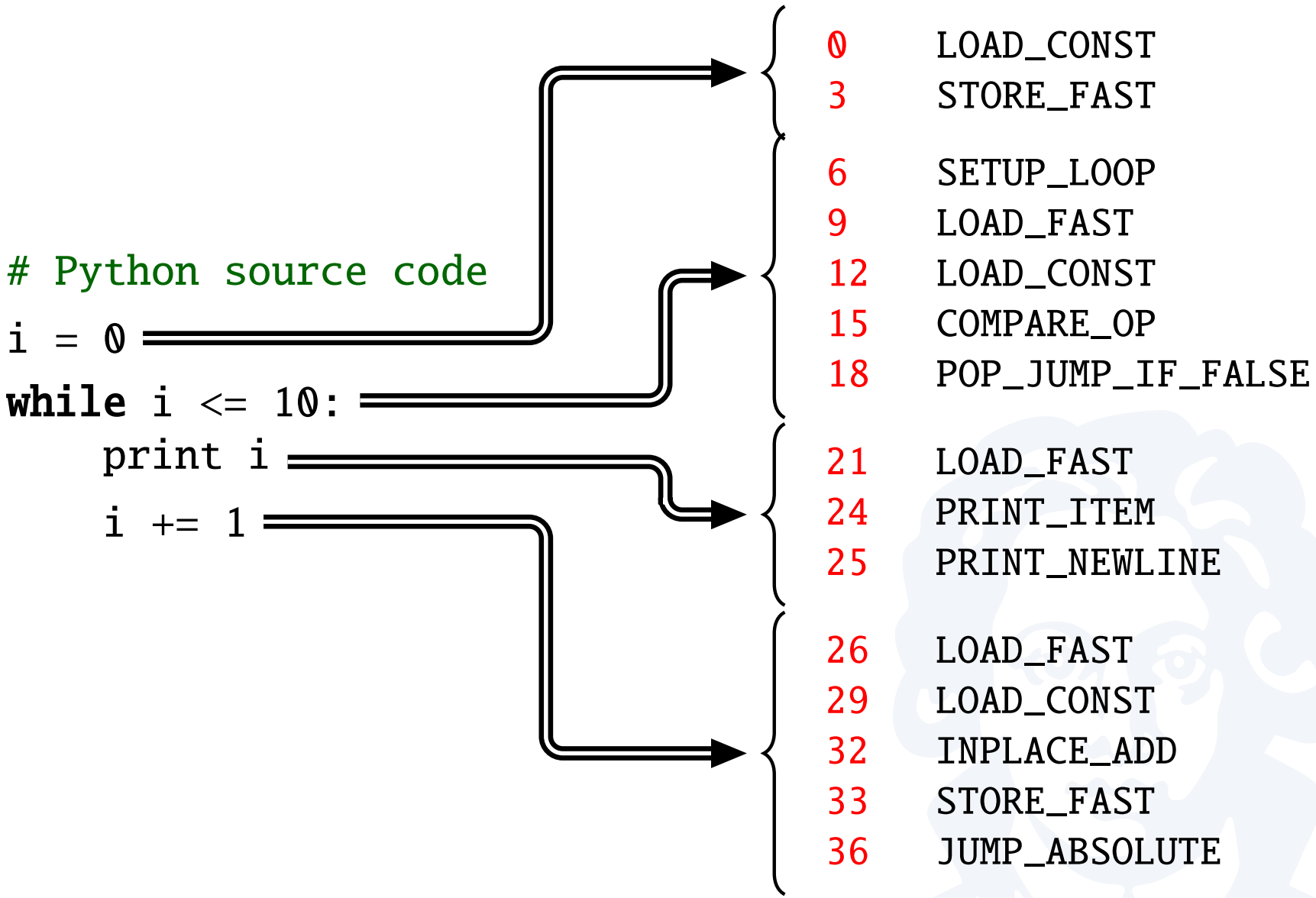
Example: CPython ('normal' Python)

```
# Python source code
i = 0
while i <= 10:
    print i
    i += 1
```

Example: CPython ('normal' Python)



Example: CPython ('normal' Python)



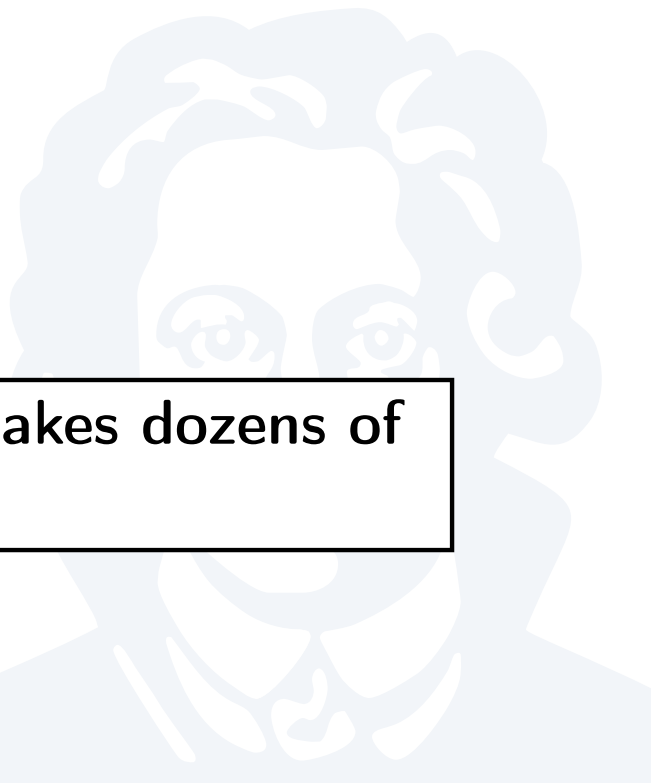
Python execution (simplified)

- Loop:
  - Load next Python operation
  - Which instruction is it? Jump to specialised code that knows how to execute the instruction:
    - Load parameters to operation
    - Perform operation
    - Continue to next operation

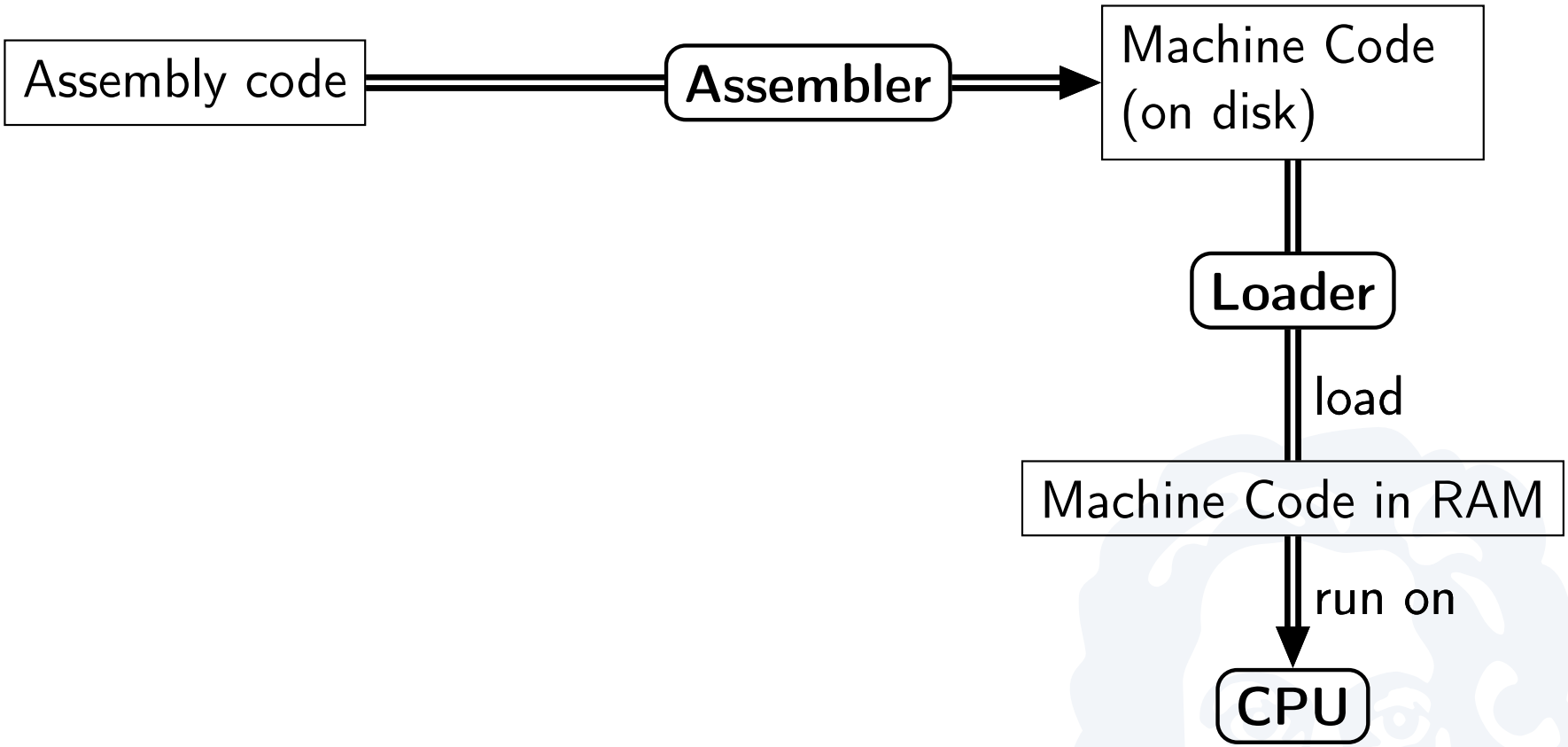
Python execution (simplified)

- Loop:
  - Load next Python operation
  - Which instruction is it? Jump to specialised code that knows how to execute the instruction:
    - Load parameters to operation
    - Perform operation
    - Continue to next operation

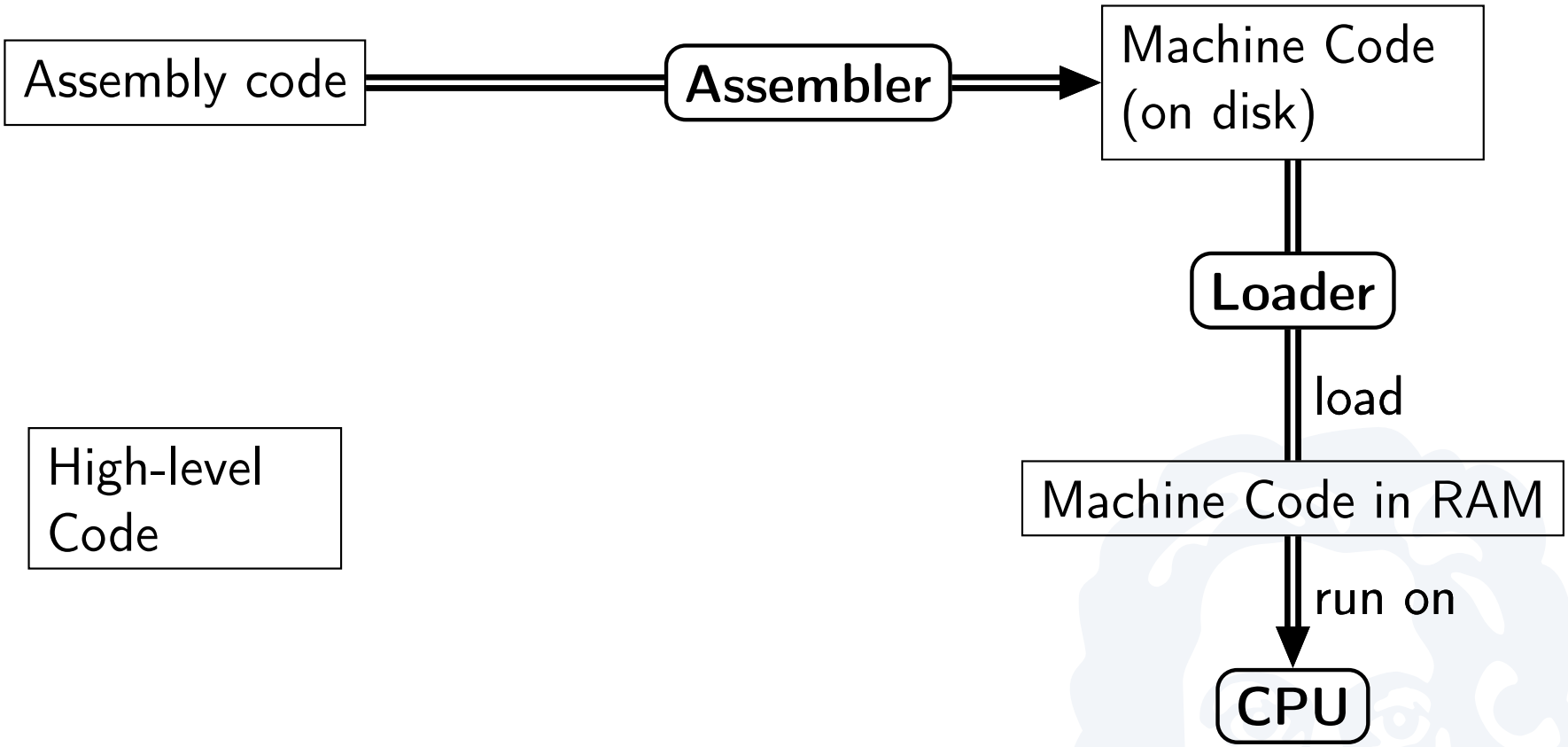
Executing e.g. an addition in CPython takes dozens of assembly instructions



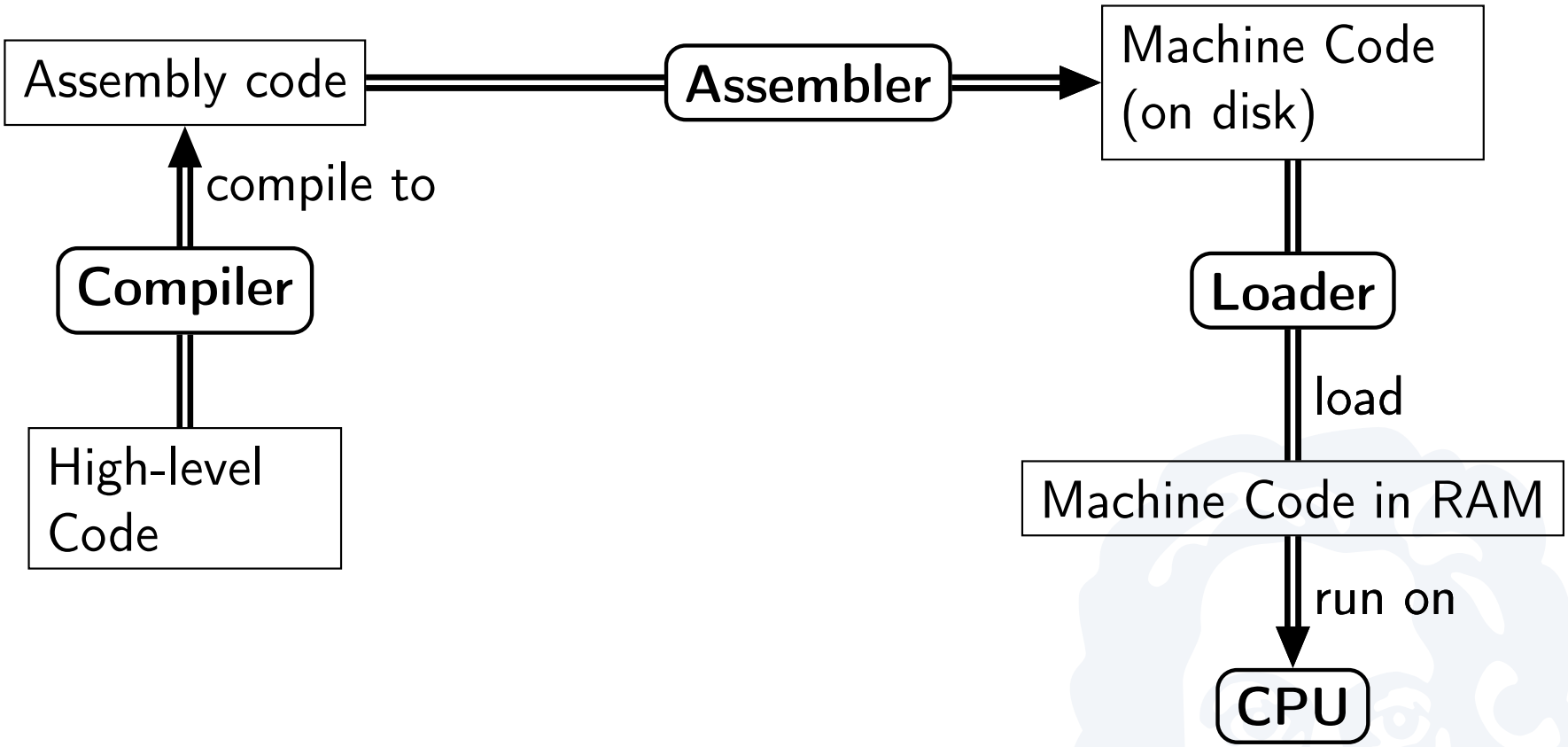
Compilation



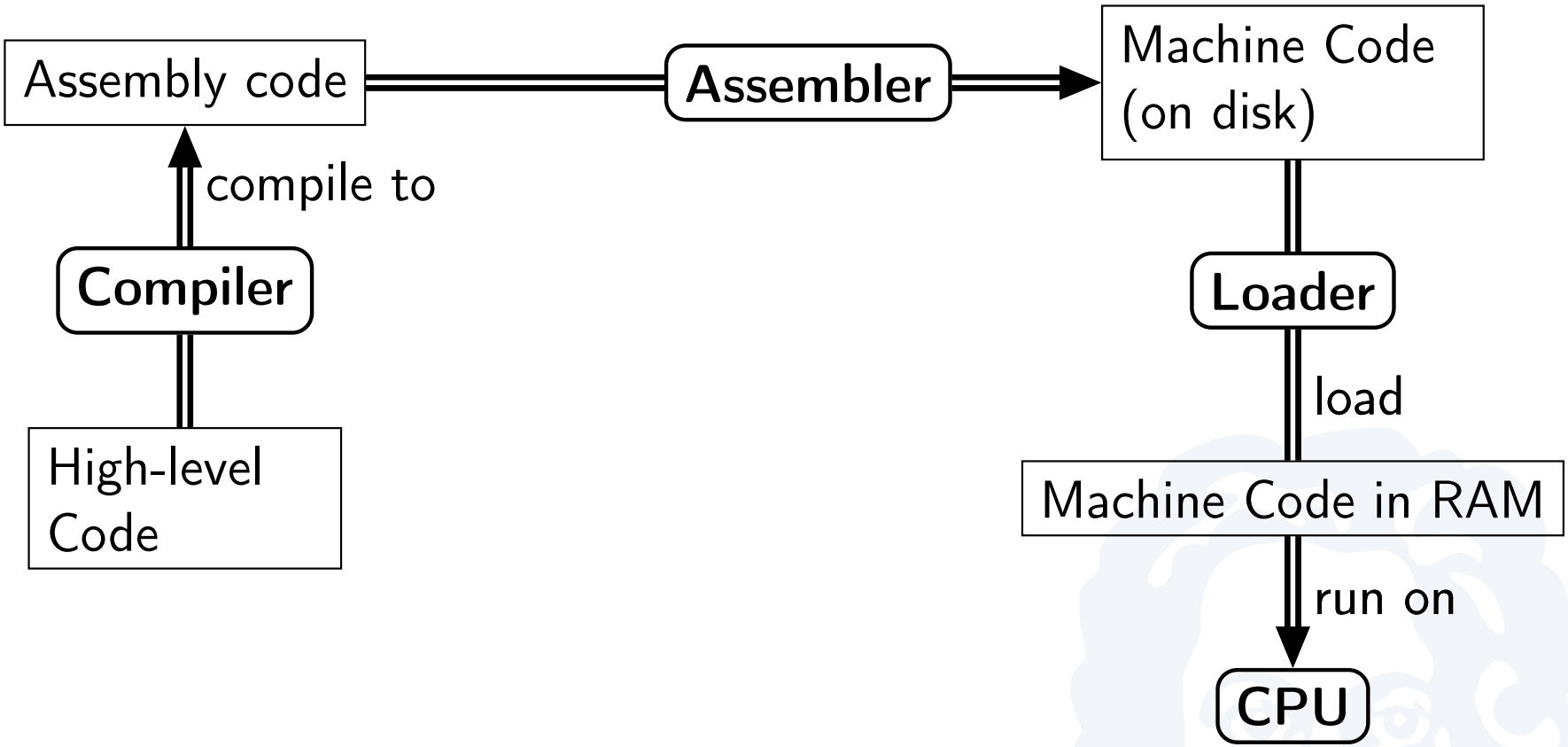
Compilation



Compilation

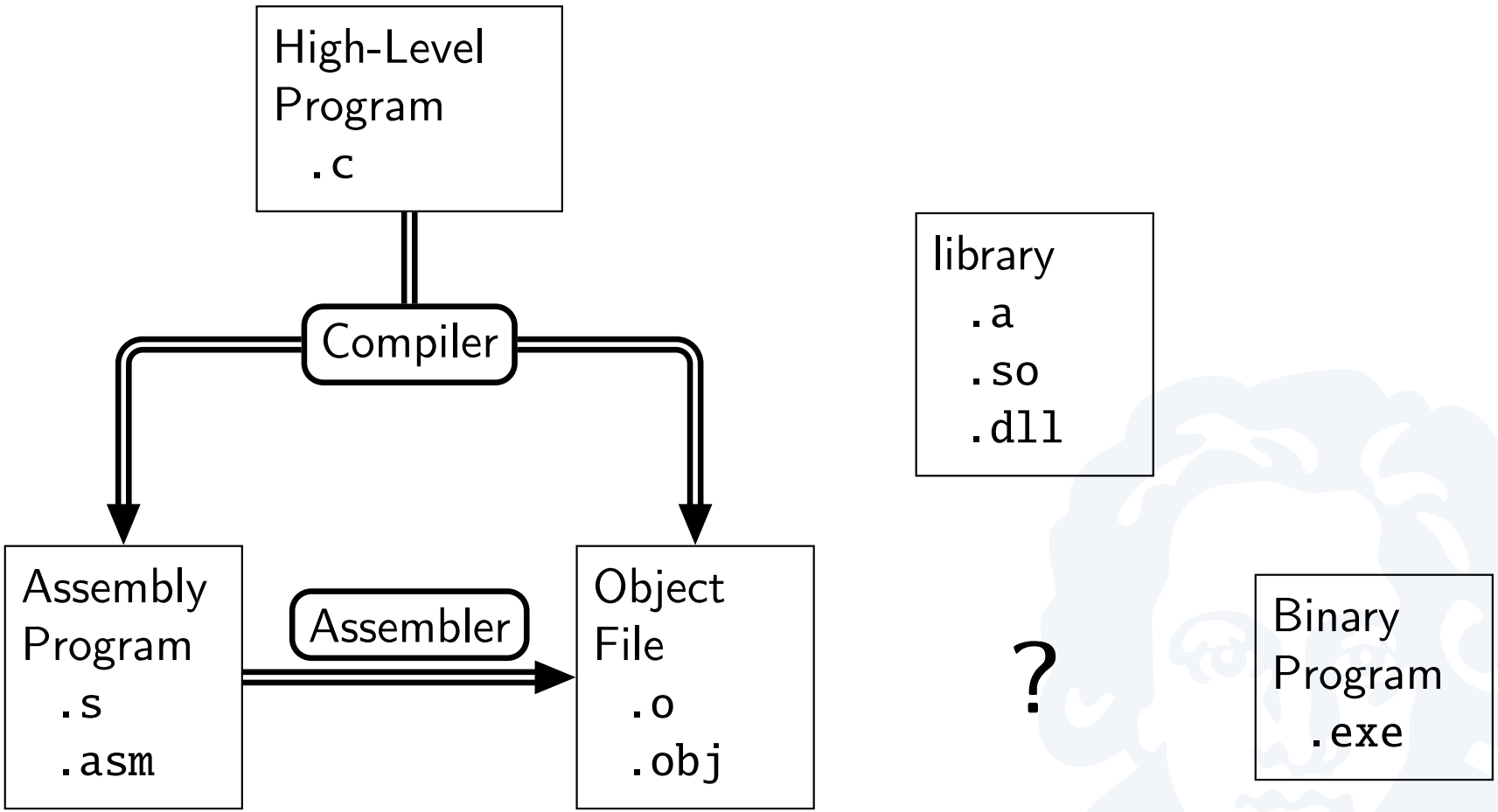


Compilation

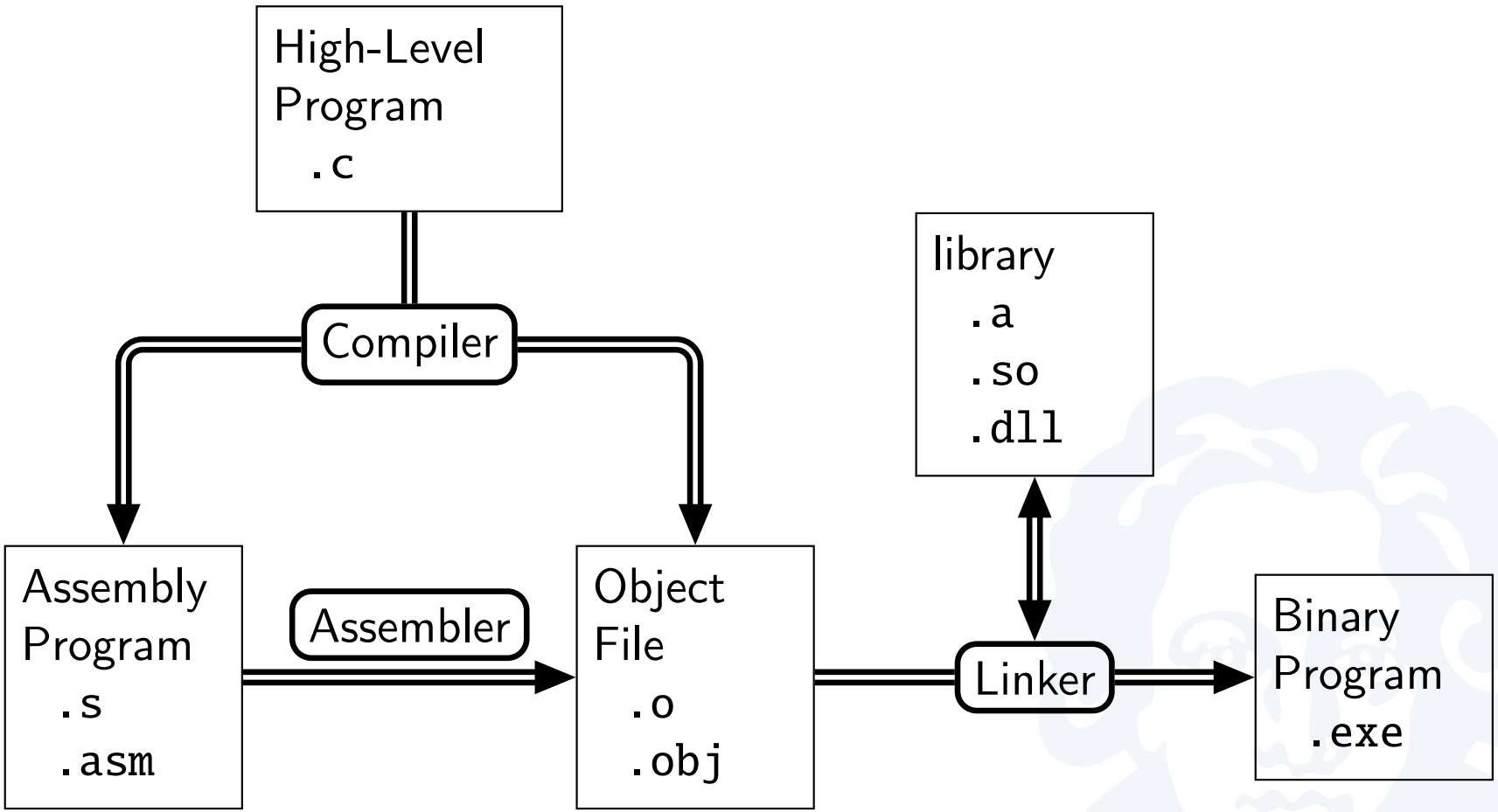


Examples: C, C++, SML, Haskell, FORTRAN, ...

Compiling and Linking in C



Compiling and Linking in C

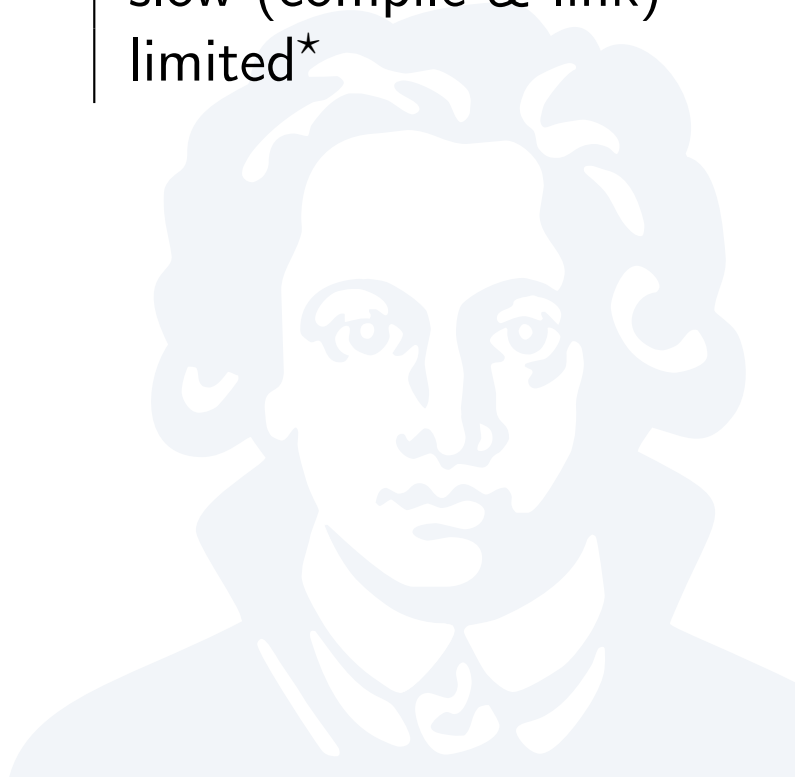


Binary program is machine code, can be passed to

Comparison: Compilation vs Interpretation

Property	Interpretation	Compilation
Execution performance	slow	**fast**
Turnaround	**fast**	slow (compile & link)
Language flexibility	**high**	limited*

$\star$) Compiler Optimisation $\lightning$ Flexibility



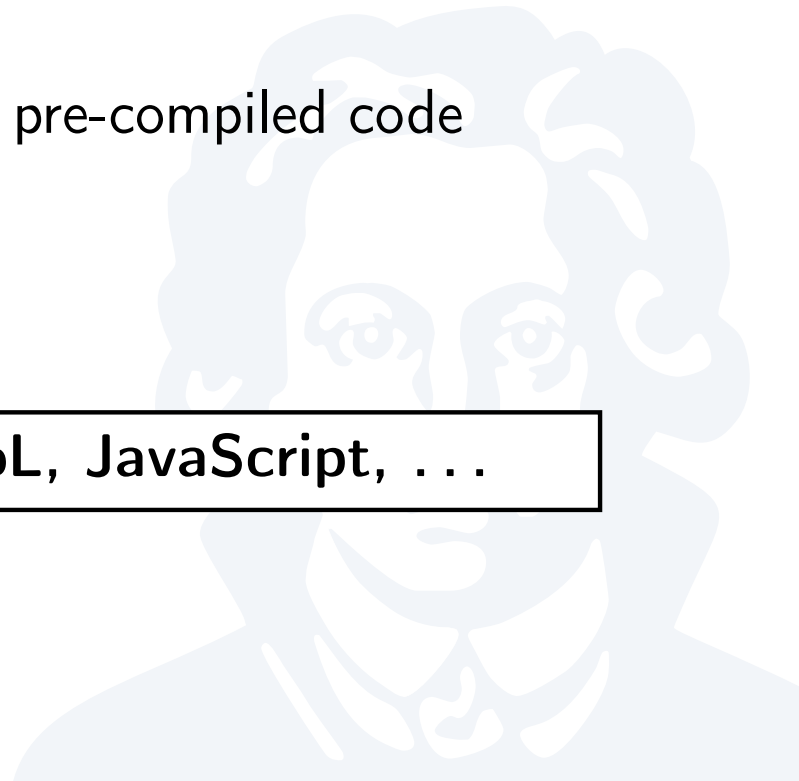
Dynamic Compilation

- Idea: compile code *while executing*
- Theory: best of both worlds

Dynamic Compilation

- Idea: compile code *while executing*
- Theory: best of both worlds
- Practice:
  - Difficult to build
  - Memory usage can increase
  - Performance can be higher than pre-compiled code

Examples: Java, Scala, C#, AttoL, JavaScript, ...



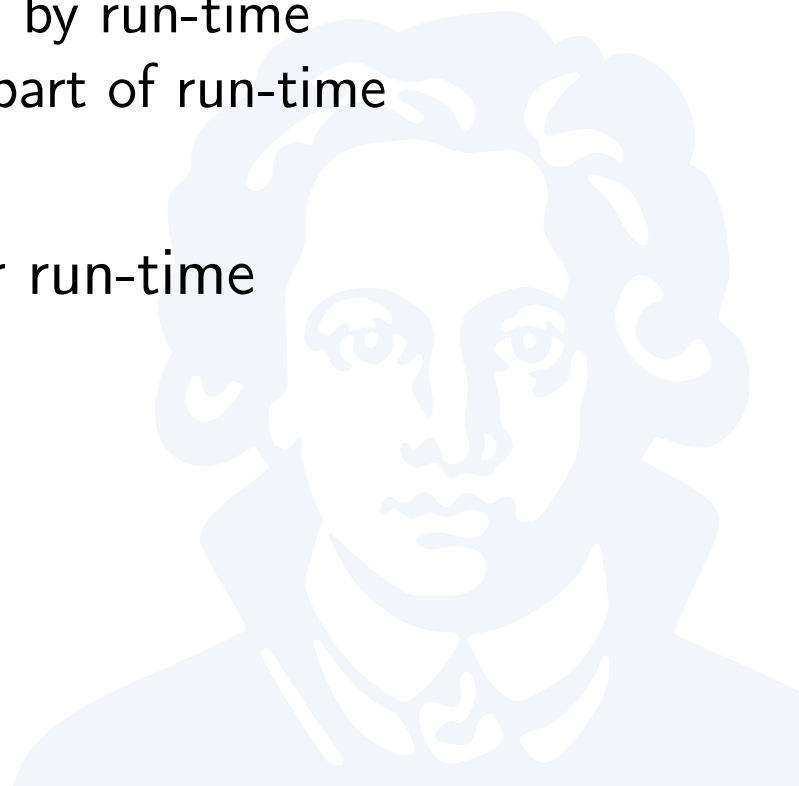
Run-Time Systems

Language features are provided by one of:

- **Compiler**
- **Run-time system**
  - Interpreter: All features provided by run-time
  - Dynamic Compiler: Compiler is part of run-time

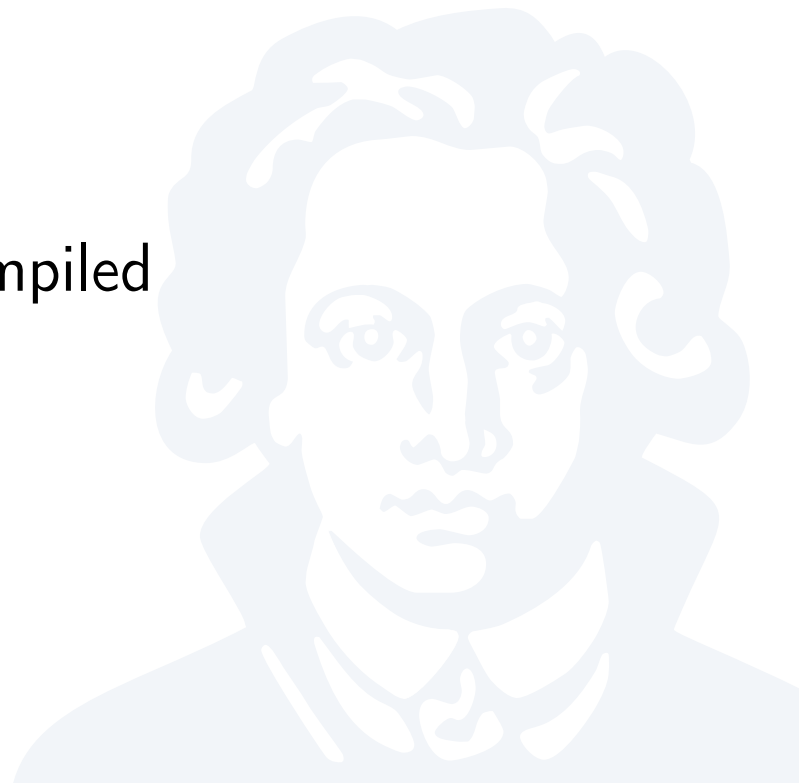
Rules of thumb:

- More complex semantics $\rightarrow$ bigger run-time
- More features $\rightarrow$ bigger run-time



Example: Java Run-Time System

- **Classloader**
  Load `.class` files
  - **Bytecode Verifier**
- **Security Manager**
- **Memory Manager**
  Allocate, deallocate heap memory
- **Just-In-Time Compiler** (JIT)
- **Interpreter**
  Execute code that hasn't been compiled
- **Native Bridge**
  Connect to C/C++ code
- **Reflection Manager**
- Misc. services (standard library)



Summary

- Languages implemented via:
  - stand-alone **Compiler**
  - **Interpreter**
  - **Dynamic Compiler**
    - May include interpreter
- Trade-off between:
  - Language flexibility
  - CPU time / RAM usage
- Languages may have multiple implementations
  - Example: CPython vs. Jython
  - gcc vs. `llvm/clang` vs. MSVC
- Any features not fully handled by stand-alone compiler are part of language run-time