



C VS. PYTHON

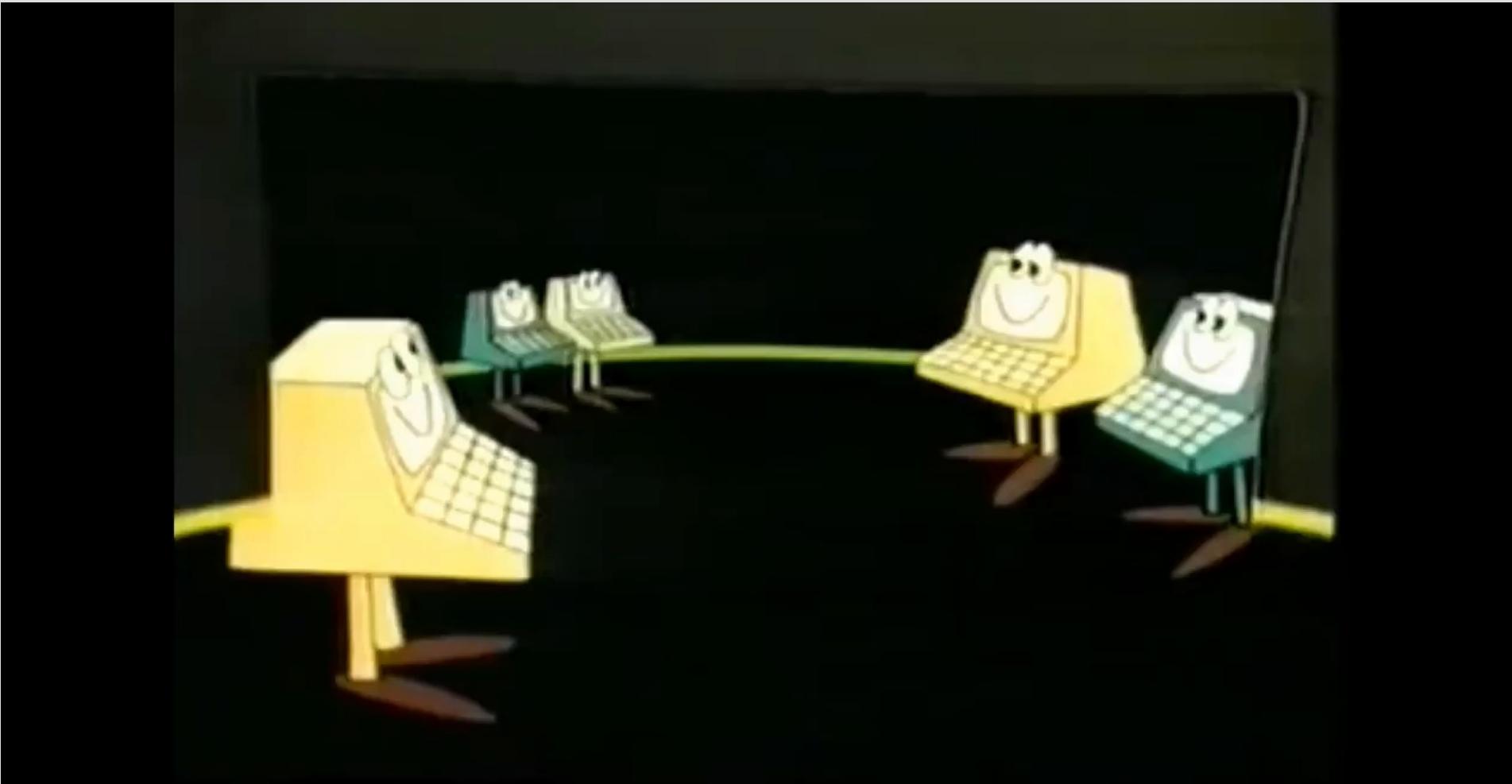
Compiler --- Interpreter --- Typisierung

COMPIILER UND INTERPRETER

- **Fragen zum Video:**

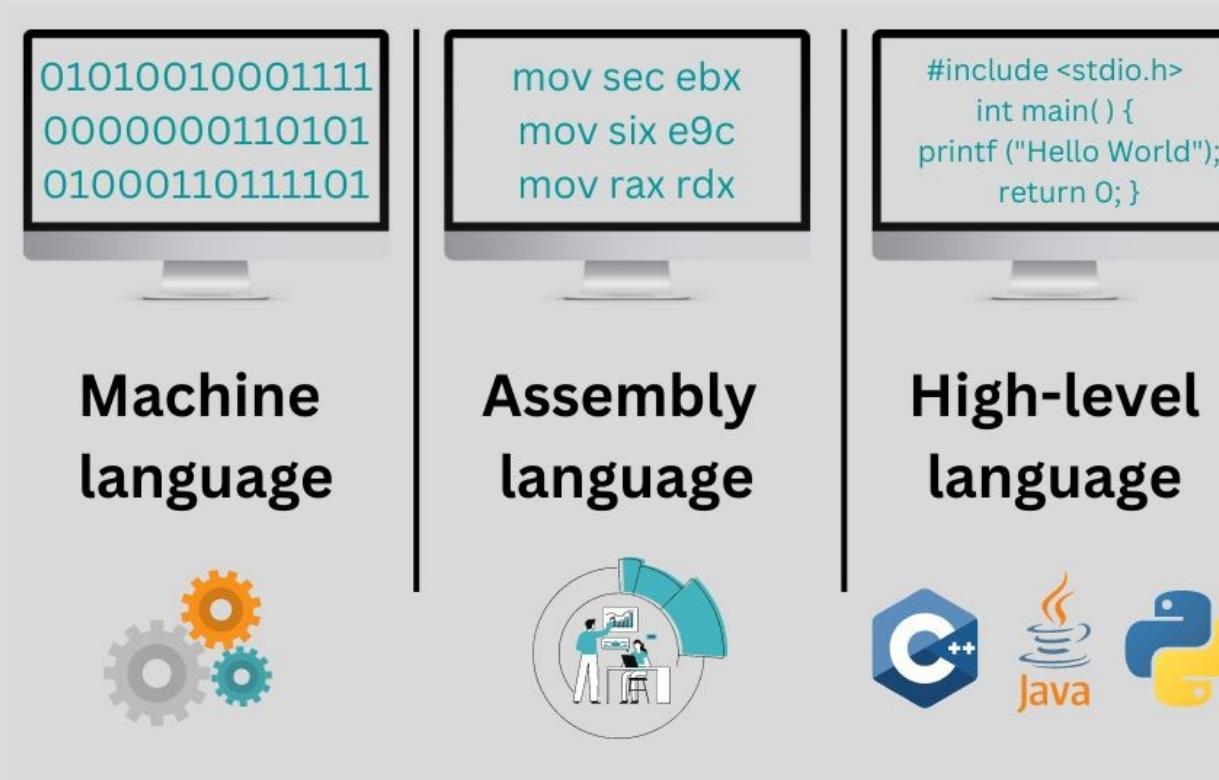
- Welche Gemeinsamkeiten haben Compiler und Interpreter?
- Welche grundlegenden Unterschiede haben Compiler und Interpreter?
- Welche Vor/Nachteile hat eine kompilierte Sprache?
- Welche Vor/Nachteile hat eine interpretierte Sprache?

COMPIILER UND INTERPRETER



COMPIILER UND INTERPRETER

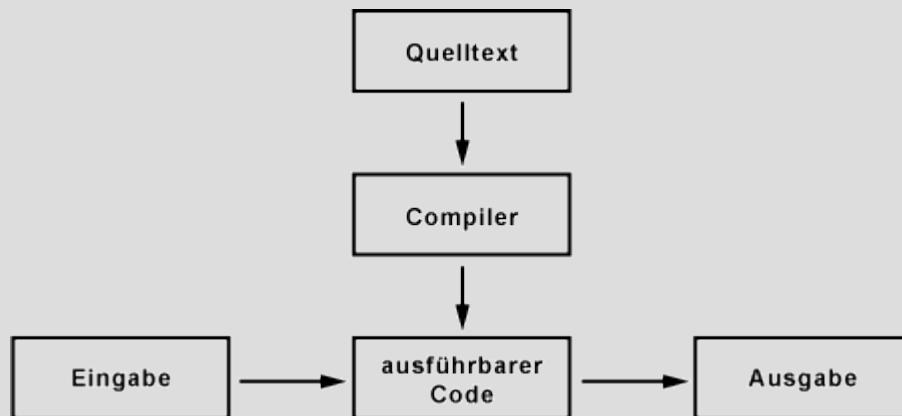
- Beide **implementieren Software auf dem Zielgerät**, indem sie den Quelltext (z.B. aus C, Java, Python, C++, ...) in **Maschinensprache** umsetzen.



COMPIILER UND INTERPRETER

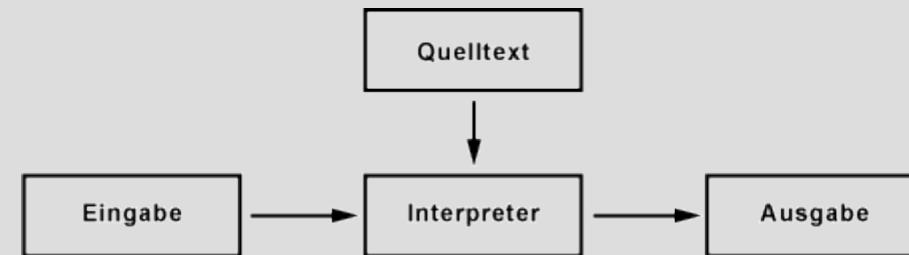
COMPILER

- Übersetzung **VOR** der Ausführung



INTERPRETER

- Übersetzung **WÄHREND** der Laufzeit



C: COMPIILER

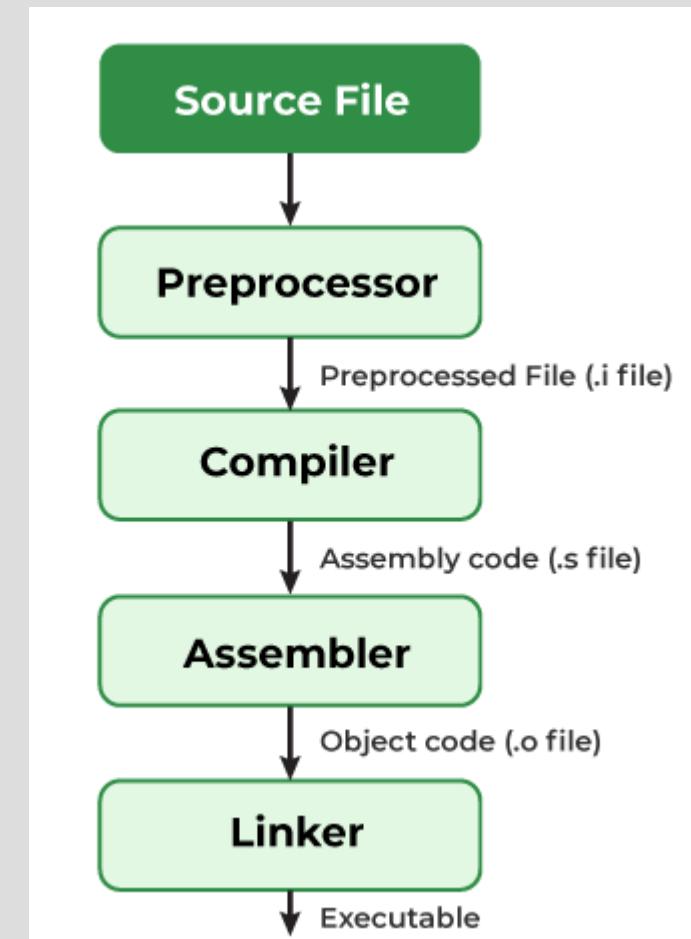
Compiler - Beispiel C („kompilierte Sprache“)

- **Input:**

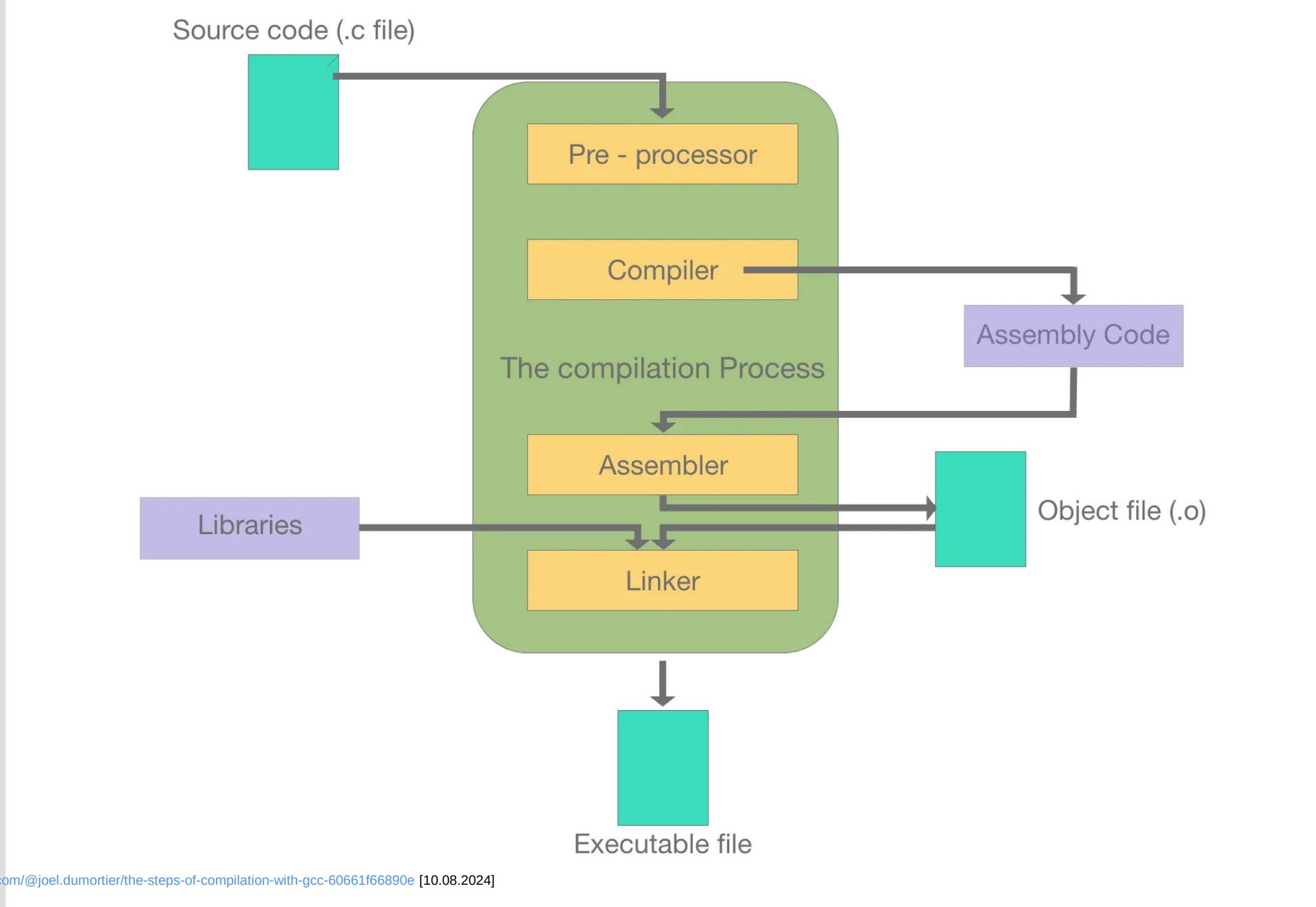
- Quellcode-Datei: *.c (obligatorisch)
- Header-Dateien: *.h
- Libraries: *.a, *.lib

- **Output:**

- Ausführbare Datei in **Maschinensprache („executable“)**: *.exe, *.out



Bildquelle:
<https://www.geeksforgeeks.org/compiling-a-c-program-behind-the-scenes/>
esL [31.08.2023]



BEISPIEL: KOMPILEIERUNG MIT GNU GCC

- **GNU Compiler Collection** (GCC) ist eine Sammlung von Compilern für C, C++, Fortran, Assembler, ...

GNU ist ein „rekursives Akronym“: GNU's not Unix

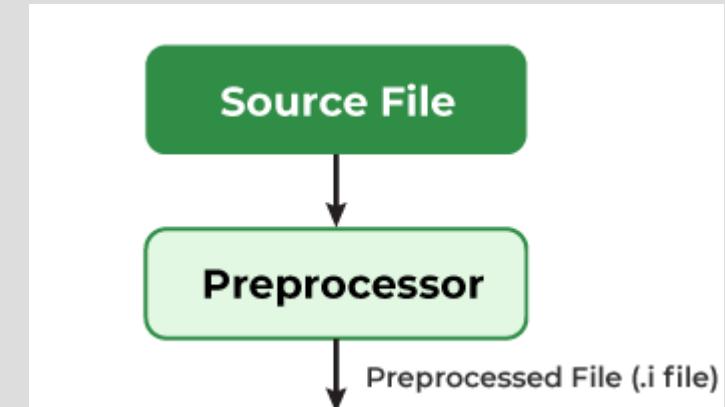
- Sammlung von vollständig freier Software: Betriebssystem-Komponenten (GNU/Linux), Software-Sammlungen, Anwendungen, Bibliotheken, ...



1: PRÄPROZESSOR

- Bereinigung des Programms (z.B. Kommentare)
- Entpacken von Makros
- Entpacken inkludierter Dateien (.h .lib)
- Bedingte Kompilierung

```
#define SQUARE(x) (x * x)
```



1: PRÄPROZESSOR

- Bereinigung des Programms (z.B. Kommentare)
- Entpacken von Makros

```
#include <stdio.h>

#define SQUARE(x) ((x) * (x))

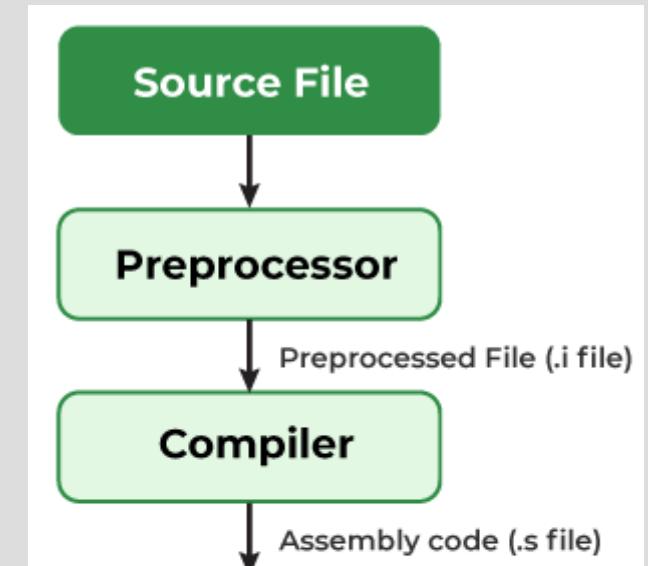
int main() {
    int num;
    scanf("%d", &num);
    int result = SQUARE(num);
    printf("Das Quadrat von %d ist %d\n", num, result);

    return 0;
}
```

2: COMPILER

- Assembler-Instruktionen (.s Datei)

```
main:  
    pushq    %rbp  
    .seh_pushreg    %rbp  
    movq    %rsp, %rbp  
    .seh_setframe    %rbp, 0  
    subq    $48, %rsp  
    .seh_stackalloc 48  
    .seh_endprologue  
    call    _main  
    movl    $3, -4(%rbp)  
    movl    -4(%rbp), %eax
```

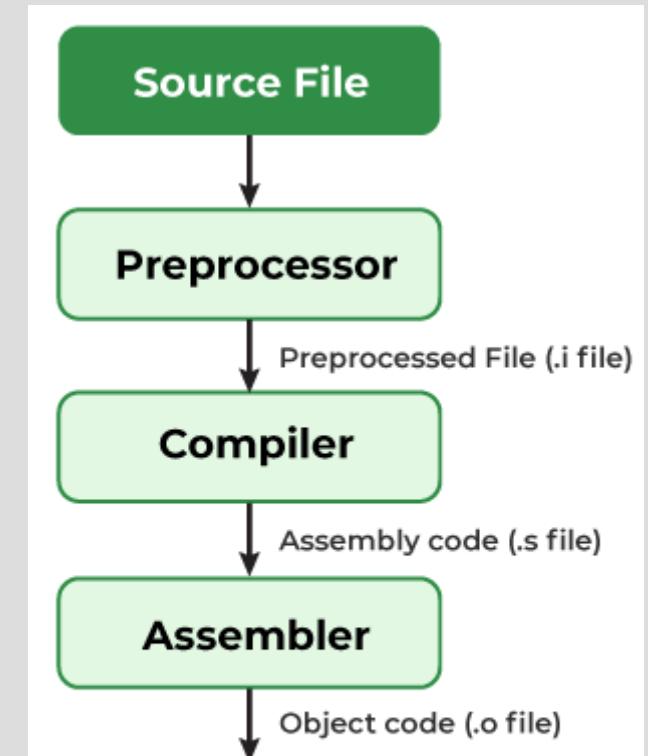


3: ASSEMBLER

- Assembler nimmt .s Datei als Input
- Übersetzung in **Maschinensprache**: .o Datei als Output
- Function calls wie *printf()* werden noch nicht aufgelöst

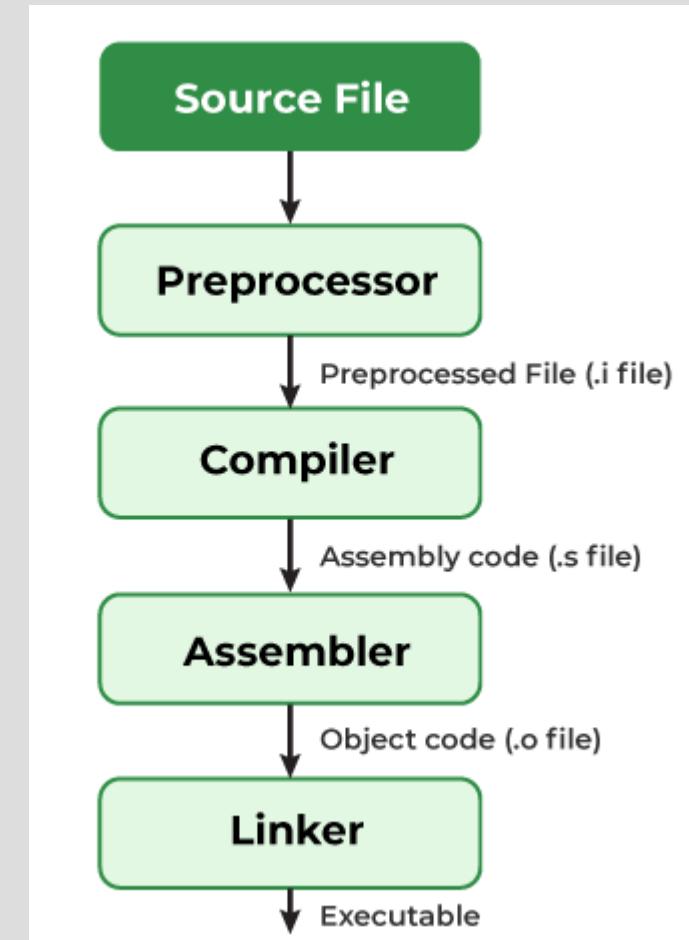
The screenshot shows a code editor window titled "main.o - Editor". The menu bar includes "Datei", "Bearbeiten", "Format", "Ansicht", and "Hilfe". The assembly code in the editor is as follows:

```
dt      .text      @ , ä      P .data
    . HfÄ0]Ä
Die Summe ist: %d      RRP :     GCC: (x86_64-
    +      .file      .file      þÿ g\main.c
```



4: LINKER

- **Input:** .o Dateien in Maschinensprache
- Linker weiß, welche .o Dateien aus unterschiedlichen Quellen
- Auflösung von Symbolen (Funktions- und Variablennamen)
- Einbindung dynamischer Bibliotheken (DLLs)
- Erstellung der executable .exe oder .out Datei



ANZEIGE ALLER ERSTELLTEN FILES

- Bei GCC save-temps Flag setzen:

```
korber@korberpc:~$ gcc -save-temps -o test test.c
korber@korberpc:~$ ls
'Calibre Library'  Pictures          test.c
Desktop            Public             test.i
Documents          rclone-sync.log   test.o
Downloads          Templates         test.s
Music              test               Videos
```

PRÄPROZESSOR IN DER PRAXIS

- Präprozessor Schritt: Makro aulösen

```
korber@korberpc:~$ grep "num" test.c
    int num;
    scanf("%d",&num);
    int result = SQUARE(num);
    printf("Das Quadrat von %d ist %d\n", num, result);
```

```
korber@korberpc:~$ grep num test.i
    int num;
    scanf("%d",&num);
    int result = ((num) * (num));
    printf("Das Quadrat von %d ist %d\n", num, result);
```

PYTHON: INTERPRETER

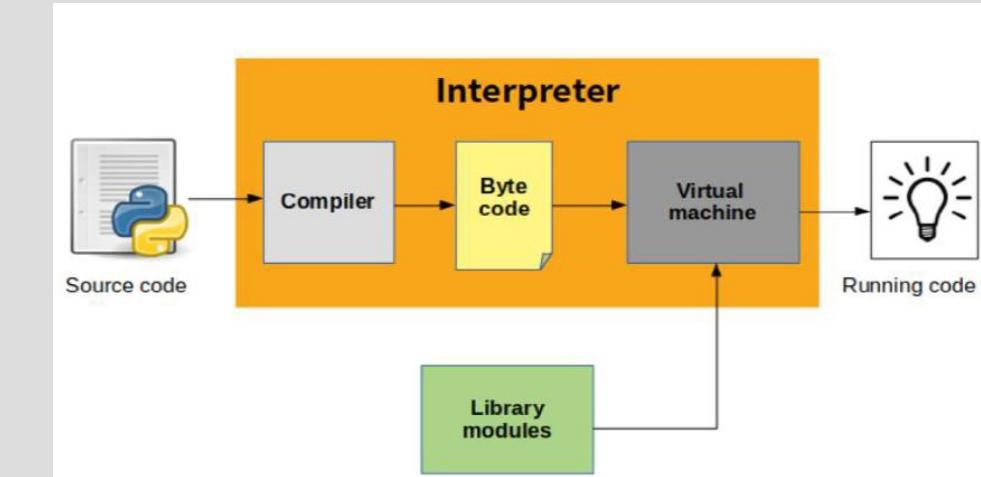
Interpreter - Beispiel Python („interpretierte Sprache“)

- **Input:**

- Quellcode-Datei: *.py (obligatorisch)
- Standardlibrary
- Module und externe Libraries (Python API)

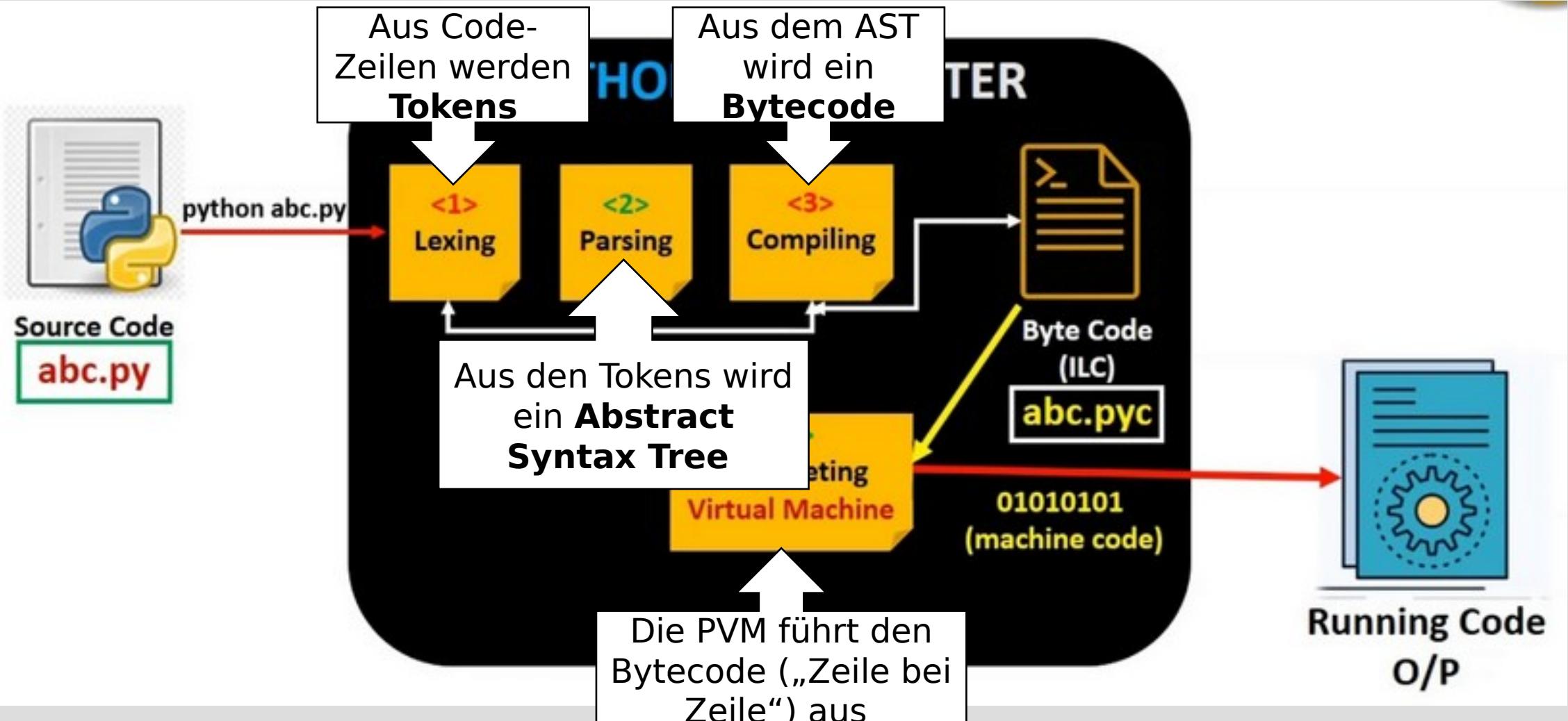
- **Output:**

- Bytecode-Datei in **maschinennaher Sprache**: *.pyc
- Explizite Ausgabe im Terminal möglich:



```
korber@korberpc:~/test$ python3 -m compileall .
Listing '...'.
korber@korberpc:~/test$ ls
__pycache__ test.py
```

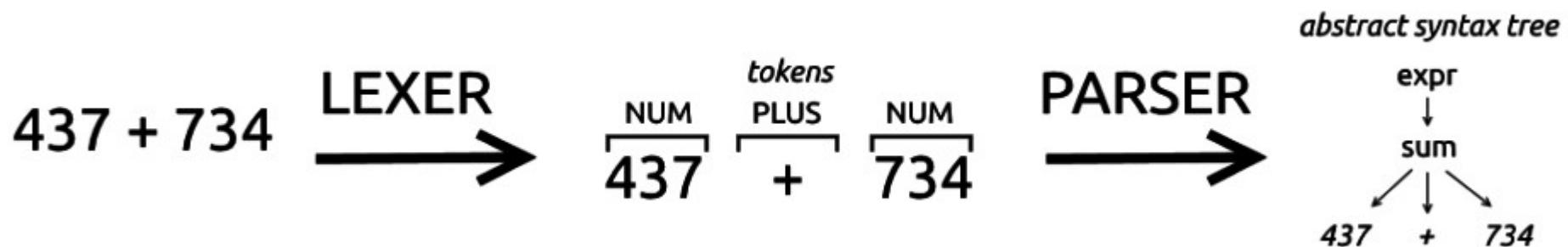
PYTHON: INTERPRETER



Bildquelle:
<https://www.tutorialspoint.com/how-does-a-python-interpreter-work>
[30.08.2023]

PYTHON: INTERPRETER

Lexer und Parser - Beispiel



COMPILER UND INTERPRETER

COMPILER

- + Geschwindigkeit
- + Ressourcen-Nutzung
 - („hardwarenahe“)
- - Abhängigkeit von
 - OS
 - Hardware

INTERPRETER

- + Unabhängigkeit von
 - OS
 - Hardware
- - Performance
 - Bytecode - Maschinencode
 - VM: „Zwischenschicht“ Hardware/Software

STATISCHE VS. DYNAMISCHE TYPISIERUNG

- Wie werden Daten mit Datentypen klassifiziert und verwendet?
 - Beim Erstellen der Variable? `int x = 5;`
 - Zur Laufzeit? `x = "5"`

DYNAMISCHE TYPISIERUNG

- Typprüfung erfolgt zur Laufzeit
- Variablentyp darf sich während der Lebensdauer ändern

```
# variable a is assigned to a string
a = "hello"
print(type(a))
```

```
# variable a is assigned to an integer
a = 5
print(type(a))
```

DYNAMISCHE TYPISIERUNG

- Funktion addiert zwei Variablen

```
# simple function
def add(a, b):
    return a + b
```

```
# calling the function with string
print(add('hello', 'world'))
```

```
# calling the function with integer
print(add(2, 4))
```

STATISCHE TYPISIERUNG

- Compiler überprüft die Datentypen
- Datentyp wird bei der Erstellung der Variable festgelegt
- Zuweisung eines Wertes mit „falschem“ Datentyp nicht (immer) möglich:

```
int my_var = 42; // my_var is declared as an integer  
my_var = "hello"; // Error: trying to assign a string  
to an integer
```

```
printf("%d\n", my_var);
```

STATISCHE TYPISIERUNG

- Typumwandlung (type casting) ist möglich:
 - implizit oder explizit

```
int a = 10; // Ganzzahl (Integer)
float b = 2.5; // Gleitkommazahl (Float)
float c = 9.5; // Gleitkommazahl (Float)
float d = 10.4; // Gleitkommazahl (Float)

// Implizites Typecasting: a --> float
result = a + b;
// Explizites Typecasting: c--> int 'c'
d = (int) c;
```

STATISCHE VS. DYNAMISCHE TYPISIERUNG

- Wie werden Daten mit Datentypen klassifiziert und verwendet?
- **Statische Typisierung (C, C++, C+, Java, Rust, Go ...)**
 - Bei jeder Definition/Deklaration wird ein (statischer) Typ festgelegt
 - Variablen können nie ein Objekt falschen Typs beinhalten
 - Variablen und Funktionen haben einen Typ
- **Dynamische Typisierung (Python, PHP, JavaScript, ...)**
 - Variablen haben keine fest assoziierten Typen
 - Nur die Objekte der Variablen haben einen Typ

TYPINFERENZ

- Viele Programmiersprachen (z.B. Rust, eingeschränkt C/C++, Java) bieten das „beste aus beiden Welten“ an
- Rust ist statisch typisiert, durch Typinferenz muss der Datentyp aber nicht angegeben werden
- Der Compiler analysiert hierfür den Code und ermittelt den Datentyp aus dem Kontext
- Beispiel Rust:

```
let x = 10; // x is inferred to be an i32 (integer)
```

STATISCHE VS. DYNAMISCHE TYPISIERUNG

Statische Typisierung: C

- Datentyp wird bei der **Deklaration** festgelegt

```
int x = 5;
```

- Fehler werden im Zuge des Kompilierungsprozesses erkannt

Dynamische Typisierung : Python

- Datentyp wird zur **Laufzeit** festgelegt

```
x = 5  
x = "hi"  
print(x)
```

- Fehler treten teilweise erst während der Ausführung des Programms auf

STATISCHE VS. DYNAMISCHE TYPISIERUNG

Statische Typisierung: C

- + Troubleshooting
- + Dokumentation, Klarheit
- + Performanz
- + Sicherheit
- - Aufwand, Komplexität
- - Flexibilität in der Verwendung

Dynamische Typisierung : Python

- + Flexibilität in der Verwendung
- + Lesbarkeit
- + Vielseitigkeit
- - Troubleshooting, Klarheit
- - Refactoring
- - Geschwindigkeit