FH CAMPUS WIEN
UNIVERSITY OF APPLIED SCIENCES

DIE HOCHSCHULE, DIE MEHR KANN.

ILV Einführung in das Programmieren

Grundlagen Programmiertechniken

Mohamed Goha, BSc. | WS 2024/25

APPLIED LIFE SCIENCES | MASTERSTUDIENGANG BIOINFORMATIK

FH CAMPUS WIEN
UNIVERSITY OF APPLIED SCIENCES

Kapitelübersicht

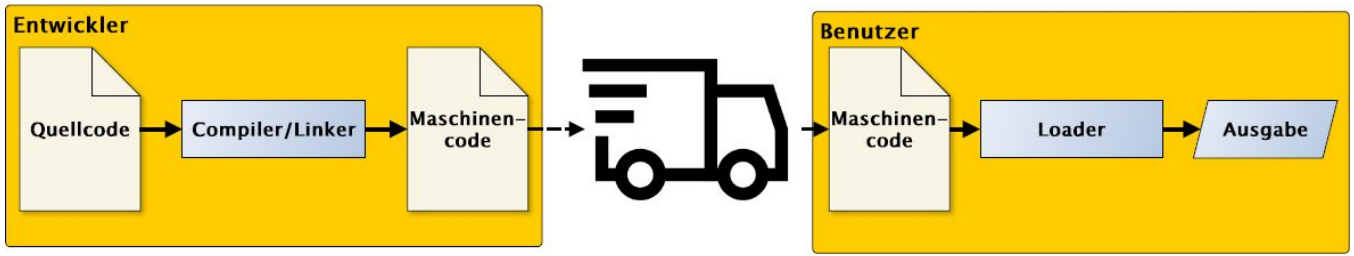
- Kompilierte und Interpretierte Ausführung
- Allgemeine Überlegungen Programmieren
- Datentypen und Variablen
- Konsole Input und Output

Compiler

- **Reminder: Ein Computer ist eine dumme Maschine**
  - Kann nur eine sehr begrenzte Anzahl von einfachen Befehlen verstehen
  - Muss ganz präzise gesagt werden, was getan werden soll.
  - Verzeiht keinen Syntaxfehler
  - Funktionalität eines Computers steckt zum allergrößten Teil in der Software
  - Um Computerprogramme niederzuschreiben, verwendet man Programmiersprachen

2 Ausführungsmodelle - Kompilierte Ausführung

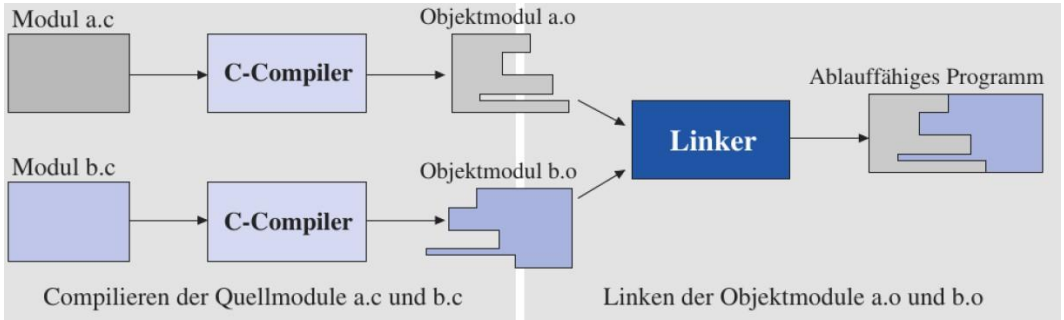
- Programm wird in Maschinencode übersetzt (kompiliert), der zu einem späteren Zeitpunkt ausgeführt wird.
- Übersetzung und Ausführung sind unabhängige Tätigkeiten.
  - **Compiler**: Übersetzt Quellcode in Maschinencode.
  - **Linker**: Baut aus mehreren Maschinencode-Modulen eine ausführbare Datei.
  - **Loader**: Ladet eine ausführbare Datei in den Hauptspeicher und startet die Ausführung.
  - Beispiel: C Sprachen, Assembler, …



Ausführungsmodelle - Kompilierte Ausführung

- **Vorteile**:
  - Sehr effizienter Code, der direkt auf der Hardware läuft
  - Kaum Overhead und geringer Ressourcenbedarf
  - Übersetzungsvorgang nur einmalig notwendig.

- **Nachteile**:
  - Code ist Plattform-spezifisch.
  - Änderungen / Neu-Kompilierung benötigen Entwicklungsumgebung.
  - Unterstützung für neue Hardware nicht ohne erneute Kompilierung möglich.



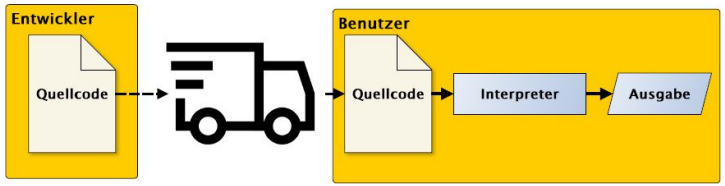
Ausführungsmodelle - Interpretierte Ausführung

- **Übersetzung und Ausführung passieren gleichzeitig:** Programm wird Befehl für Befehl in Maschinencode übersetzt und sofort ausgeführt

- **Interpreter**: Führt Quellcode Programm direkt aus ohne dass Zwischenschritt notwendig ist.
  - Liest Quellcode ein und führt Befehle direkt aus.
  - Übersetzt Quellcode in Zwischensprache und führt diesen aus.
  - Python, R, PHP

- **Vorteile**:
  - Plattform-unabhängig (für neue Plattform nur neuer Interpreter notwendig).
  - Leicht Änderbar (simpler Texteditor reicht).
  - Feedback-Schleife für den Programmierer kürzer.
  - Quellcode für Benutzer einsehbar und änderbar.

- **Nachteile**:
  - Höherer Overhead und Ressourcenbedarf.
  - Quellcode für Benutzer einsehbar und änderbar.
  - Eingeschränkter Zugriff auf Hardware.

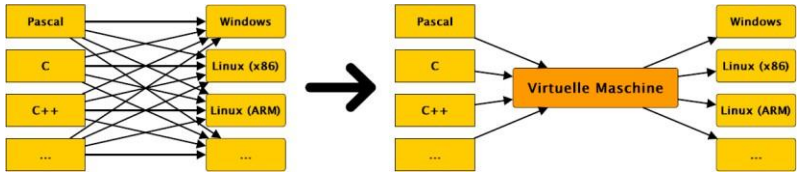


Ausführungsmodelle - Hybride Ausführung

- **Übersetzung in Zwischencode (Bytecode)**
  - Compiler übersetzt Quellcode in Zwischencode (Bytecode).
  - Virtuelle Maschine: Führt Bytecode aus (Simuliert einen virtuellen Computer, der den Bytecode direkt versteht).
  - Statt m * n Compiler (für m Programmiersprachen und n Plattformen) werden nur m Compiler und n virtuelle Maschinen benötigt. Reduktion der Komplexität von m * n auf m + n
  - Java, Scala, Groovy, .NET Sprachen

- **Vorteile**:
  - Effizienterer Code als beim interpretierten Ausführungsmodell.
  - Plattform-unabhängig.
  - Quellcode für Benutzer nicht zugänglich.

- **Nachteile**:
  - Höherer Overhead und Ressourcenbedarf als im kompilierten Ausführungsmodell.
  - Eingeschränkter Zugriff auf Hardware.



Fehlertoleranz und Programmiersprachen

- **Natürliche Sprache:** wir Menschen verstehen trotz Schreibfehlern immer noch was gemeint ist.
  - Gibst du mir deine Adrese?
  - Ich habe Hunge.
  - Ich jetzt gehen einkafen.

- **Formale Sprache:**
  - print("Hallo Welt"
  - if 10a > 49„

- **Ein Computer verzeiht keinen einzigen Schreibfehler**
  - Ein Programm ist entweder gültig oder ungültig (kein Graubereich).
  - Der kleinste Schreibfehler sorgt dafür, dass das Programm nicht funktioniert.
  - Deshalb immer darauf achten, dass die Syntax zu 100% korrekt ist.
  - In der Praxis bedeutet das regelmäßiges und vollständiges Kompilieren und Testen des Programms, ob der Computer es auch versteht.

Allgemeine Überlegungen Programmieren

> Beim Programmieren geht es darum, eine Maschine eine Aufgabe/ein Problem lösen zu lassen.

> Prozedere:
1. Problem identifizieren und **verstehen**
2. Gedanken zur Lösung dieses Problems strukturieren
   → einen **Algorithmus** entwickeln
3. Algorithmus in einer Programmiersprache formulieren
   → **Programm** schreiben
4. Programm ausführen.

Mögliche Fehler

- Syntaktische Fehler
  - Verstoß gegen Syntaktik-Regeln der Programmiersprache
  - Beispiele: fehlende Zeichen, Zeichen an der falschen Stelle, Klammer geöffnet aber nicht geschlossen, etc.

```
print("Hello World!"
```

```
def add_numbers(a:int, b:int)
    return a+b
```

- Semantische Fehler
  - Denk - oder Implementierfehler: Entweder Ihr Algorithmus ist keine (vollständige) Lösung für das Problem, oder ihr Code implementiert nicht Ihren Algorithmus, oder beides.
  - Meist viel aufwändiger zu beheben! Deshalb wichtig, Problem von Anfang an gut zu verstehen.
  - Können oft unentdeckt bleiben!

Reminder Speicher

> Daten werden auf Computern in Form von **Bits** gespeichert
> Bits: 2 Zustände (True/False, 0/1, etc.)
> Byte = 8 Bits
> Zahlen, Text, Programme, Bilder, etc. werden zu bit patterns **codiert**
> Später **decodiert** um Bedeutung der Daten wiederherzustellen (z.B Text anzeigen)

Datentypen

- Je nach Art der Daten (Ganze Zahl, Gleitkommazahl, Zeichen, etc.) wird der Wert der Variable anders zu bits codiert.
- Diese verschiedenen Arten, Werte zu codieren nennt man **Datentypen.**
- Datentypen variieren darin, wie viele bits dafür allokiert werden, den Wert zu speichern.
- Beispiele elementare Datentypen:
  - **bool** (Boolean/Wahrheitswert) - binär: Entweder “True” oder “False”
  - **int** (integer/Ganze Zahl)
  - **float** (floating point number/Gleitkommazahl)
  - **double** (double-precision floating point number)
  - **char** (character/einzelnes Zeichen)
  - **str** (string of characters/Zeichenkette)

Häufigste Komplexe Datentypen

- array
- list
- tuple
- set
- dictionary
- struct

Auf diese werden wir später näher eingehen - wenn wir Datenstrukturen behandeln. Meist Sequenzen oder eine Mischung mehrerer elementarer Datentypen.

Elementare Datentypen: Boolean (`var = True`)

- Praktisch, um Bedingungen zu prüfen
- In Python: kann nur auf “True” oder “False” gesetzt werden.
  Intern als integer repräsentiert (0=False, 1=True)

Elementare Datentypen: Integer (`var = 5`)

- Ist “precise” - heißt, dass keine Informationen verloren gehen (siehe float)
- Oft wird zwischen verschieden großen integern unterschieden (32bits, etc.)
- In Python variable Länge - beliebig lange Zahl kann gespeichert werden

Elementare Datentypen: Float (`var = 5.6`)

- Nicht “precise” - kann nur eine bestimmte Anzahl von Ziffern ohne Verlust von Informationen speichern.
- In Python **double-precision floats** (64 bits)

Elementare Datentypen: String (`var = "hello"`)

- Ein bit pattern pro Zeichen (**character**)
- String = Konkatenation mehrerer characters
- Verschiedene mögliche Codierungen: ASCII, UTF-8, etc.
- Repräsentation variiert zwischen Programmiersprachen

Variablen

- = etwas, das einen veränderlichen Wert halten kann und einen Namen hat, um auf diesen Wert zuzugreifen.
- Wir können Informationen in einer Variablen speichern (zuweisen), darauf zugreifen und sie ändern.
- repräsentieren Speicherstellen im RAM

Variablen

- **Variablen besitzen einen Gültigkeitsbereich (Scope)**
  - z.B.: In einer Funktion definierte Variablen sind nur in dieser Funktion gültig.
  - Programmiersprachen unterscheiden sich oft signifikant in der Handhabung des Gültigkeitsbereichs

Static typing vs Dynamic typing

Static typing:

- Datentyp der Variable ist im Voraus bekannt
- Deklarierung/Initialisierung mit Datentypen: `int alter = 5`
- **Variable ist mit Datentyp asoziiert**
- z.B. C-Sprachen, Java, Go, Typescript

Dynamic typing:

- Datentyp wird bei der Ausführung bestimmt
- **Datentyp ist mit dem Wert assoziiert, nicht mit der Variable**
- Deklarierung/Initialisierung ohne explizite Nennung des Datentypen: `alter = 5`
- z.B Python, R, Javascript

Variablen in Python

> lediglich Verweise (reference) auf Objekte, die automatisch im Hintergrund erzeugt, gespeichert und verwaltet werden. Im Wesentlichen nur identifier, die an Objekte gebunden sind – sie selbst speichern keine Informationen.

> Objekte enthalten Informationen über den Datentyp und die Art der Daten, die im Speicher gespeichert sind.

> Alles in Python ist ein Objekt

Variablen in Python

- Beispiel: var=3
  - Variable/identifier var ist an 3 gebunden
  - 3 ist ein Python integer Objekt mit numerischem Wert 3

- Variablennamen:
  - müssen mit Zeichen beginnen die keine Zahlen oder Operatoren sind
  - sind case sensitive
  - konventionell: in Kleinbuchstaben geschrieben und Wörter werden mit Unterstrich getrennt
  - Bestimmte Namen (keywords) sind reserviert und können nicht verwendet werden (z.B. for, while, if, def, class)

Console Input und Output

- Die Konsole ist die Standard-Ein- und Ausgabeschnittstelle.
- Output (Etwas in der Konsole ausgeben):
  - ```
    print("Hello World!")
    ```
  - ```
    print(234)
    ```
  - ```
    print(some_variable)
    ```
- Input (Eingaben von dem User einlesen):
  - ```
    alter = input("Bitte geben Sie Ihr Alter ein: ")
    ```
  - Die eingelesene Eingabe wird ein String sein, daher muss evtl. eine Typumwandlung manuell durchgeführt werden.