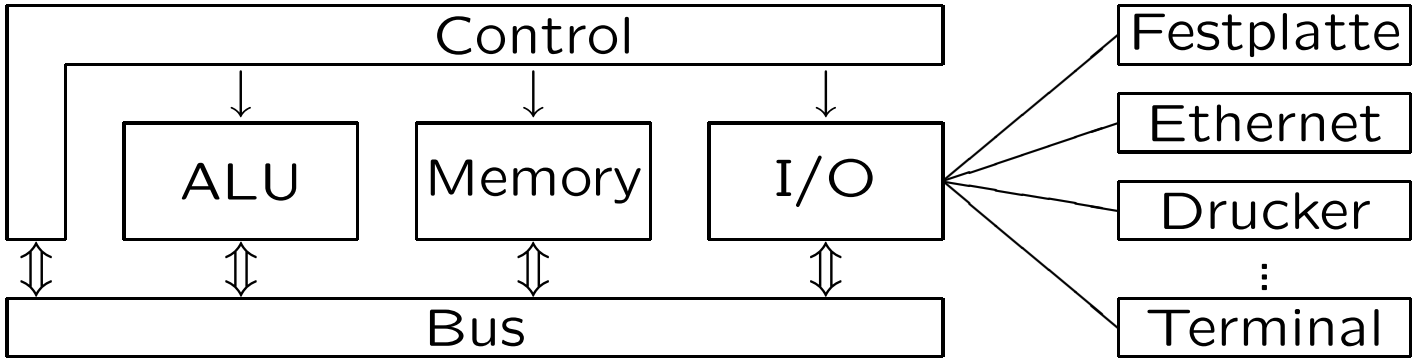
Maschinen und Architekturen

Von Neumann-Architektur



Zur **Ausführung** ständig wiederholt:

1. hole nächsten Befehl aus Speicher (durch PC adressiert)
2. erhöhe PC um 1
3. interpretiere Befehl und führe ihn aus

Aktuelle Computer-Architekturen

- *Harward-Architektur* trennt Befehls- von Datenspeicher
- Hierarchie von Speicherebenen: Register und Caches
- virtuelle Adressierung
- Spezialregister und Spezialbefehle für Zugriffe darauf
- Prozessor fasst zentrale Teile auf einem Chip zusammen
- mehrere Prozessorkerne

Assembler-Programm (i386, Linux)

```
section .text
global _start
_start:
    mov ecx, hello
    mov edx, length
    mov ebx, 1
    mov eax, 4
    int 80h
    mov ebx, 0
    mov eax, 1
    int 80h
section .data
    hello  db 'Hello World!'
    length equ $ - hello;
```

Architektur versus Implementierung

- *Architektur* beschreibt Maschine in dem Detailiertheitsgrad, den Programmierer bzw. Benutzer kennen muss
- *Implementierung* der Architektur = tatsächliche Maschine
- Alle Implementierungen derselben Architektur verstehen dieselben Programme (Portierbarkeit)

Abstrakte Maschine

- abstrakte Beschreibung aller Implementierungen
- niedriger bis hoher Abstraktionsgrad
- häufig in Software implementiert (keine Hardware)
- Beispiel: *Java Virtual Machine (JVM)* – Bytecode
- höherer Abstraktionsgrad erhöht Portabilität
- niedriger Abstraktionsgrad erhöht Effizienz

JVM-Code für Hello

```
public class Hello extends java.lang.Object{
public Hello();          //Konstruktor (nicht verwendet)
  Code:
   0: aload_0
   1: invokespecial #1; //Method java/lang/Object."<init>"
   4: return

public static void main(java.lang.String[]);
  Code:
   0: getstatic #2;     //Field java/lang/System.out
   3: ldc #3;           //String Hello World!
   5: invokevirtual #4; //java/io/PrintStream.println
   8: return
}
```

Berechnungsmodell

- abstrakte Maschine auf sehr hohem Abstraktionsgrad
- reine Gedankenmodelle, keine Implementierungen nötig
- Beispiele: Lambda-Kalkül, Turing-Maschine
- zeigen Möglichkeiten und Grenzen der Programmierung bzw. der gesamten Informatik auf
- Berechnungsmodell hinter jeder Programmiersprache
- höhere und hardware-nähere Programmiersprachen

Objekt als Maschine

- jedes Objekt entspricht einer abstrakten Maschine beschrieben in der Klasse des Objekts
- Objektzustand $\approx$ Speicherinhalt
- Objektverhalten $\approx$ Befehlsausführung
- Objektidentität $\approx$ Internetadresse
- diese Sichtweise erlaubt Konzentration auf das Wesentliche
- Unterscheidung *Schnittstelle* von *Implementierung*

Softwarearchitektur

- beschreibt wichtigste Teile der Software
- Teile heißen Module, Komponenten, Objekte
- Schnittstellen bestimmen Kombinierbarkeit
- Analogie: Auto mit Motor, Getriebe, Radaufhängung
- macht Spezialisierung auf einzelne Teile möglich
- gute Architektur erleichtert vieles

Formale Sprachen

Beschreibung von Sprachen

im Prinzip ähnlich: formale und natürliche Sprachen

Syntax: Aufbau der Sätze bzw. Programme

Grammatik = Regelsystem zur Beschreibung der Syntax

Semantik: Bedeutung von Begriffen, Sätzen, Programmen

in formalen Sprachen: Semantik = komplexe Regeln

Pragmatik: praktische Aspekte der Sprachverwendung

Verhältnis der Sprache zu Sprecher und Angesprochenem

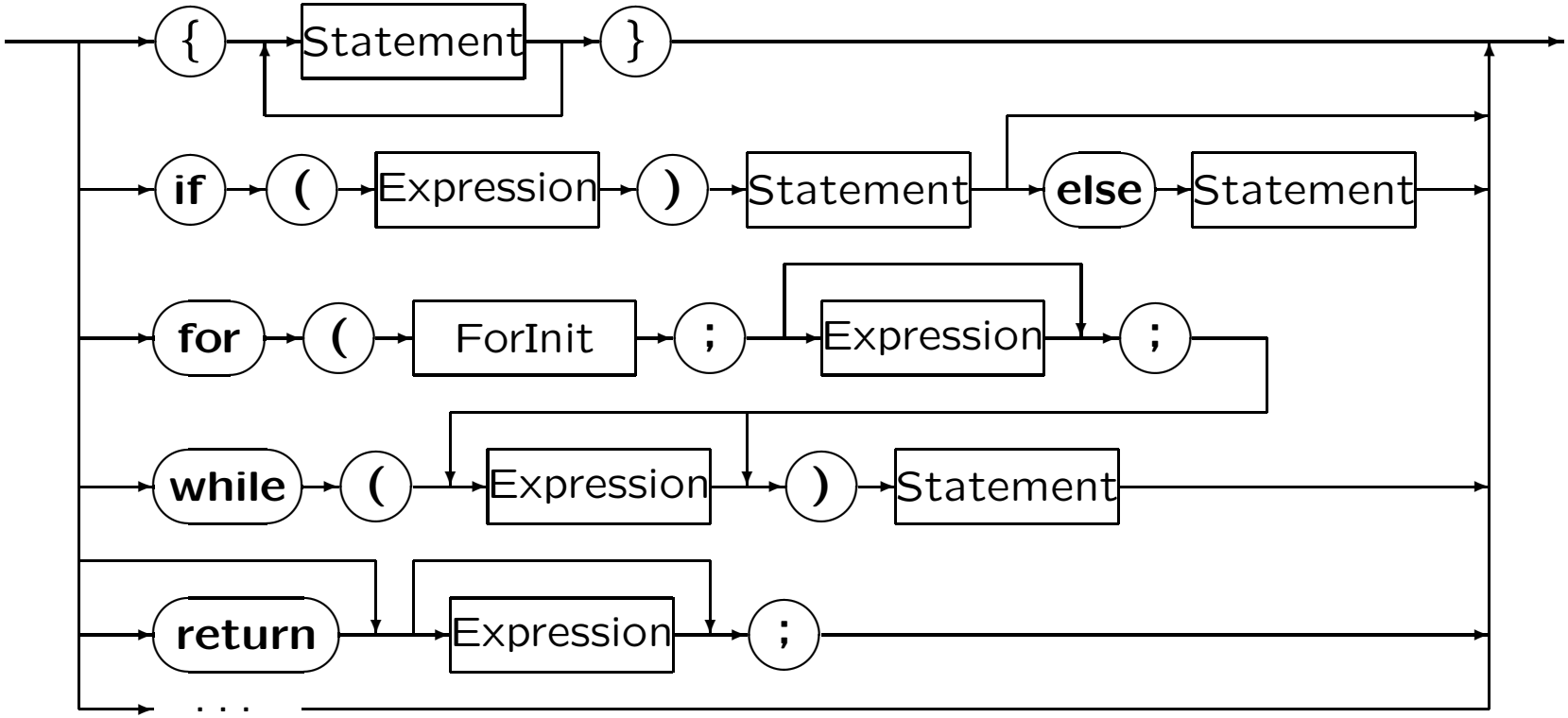
Grammatik (EBNF)

Statement = **{** {Statement} **}**
| **if (** Expression **)** Statement [**else** Statement]
| **while (** Expression **)** Statement
| **return** [Expression] **;**
| [Expression] **;**
| . . .

- Alternative: $A \mid B$
- Wiederholung: **{**$\{A\}$**}** (**{}**, **{**A**}**, **{**$A\ A$**}**, . . .)
- Option: $[A]$

Grammatik (grafisch)

Statement:



Beispiel: Syntax, Semantik, Pragmatik

- Ausdruck entsprechend der Grammatik:

```
if(zahl<0){zahl=zahl+grenze;}
```

- gleiche Syntax und Semantik, andere Pragmatik:

```
if (zahl < 0) {
    zahl = zahl + grenze;
}
```

- gleiche Semantik, andere Syntax und Pragmatik:

```
if(zahl<0)zahl=zahl+grenze;
```

Bestandteile eines Programms

```
import java.util.Random;                    // Programmorganisation
public class UnbekannteZahl {
    private int zahl;                          // Datenstrukturen
    public UnbekannteZahl (int grenze) {
        zahl = (new Random()).nextInt() % grenze;
        if (zahl < 0) {
            zahl = zahl + grenze;
        }                                          // Algorithmen
    }
    public boolean gleich (int vergleichszahl) {
        return (zahl == vergleichszahl);
    }
    ...                                           // Umgebung
}
```

Statisch versus Dynamisch

- Algorithmen, Datenstrukturen, Programmorganisation und Umgebung sind *statisch* – ändern sich zur Laufzeit nicht
- Daten in Datenstrukturen sind *dynamisch*
- statische versus dynamische Sprachen: Grenzen fließend
- Unterscheidungsmerkmale:
  deklarierte Typen, statische Typüberprüfungen
- statisch: Programme leicht lesbar, besser überprüft
- dynamisch: Programme leicht schreibbar, flexibler