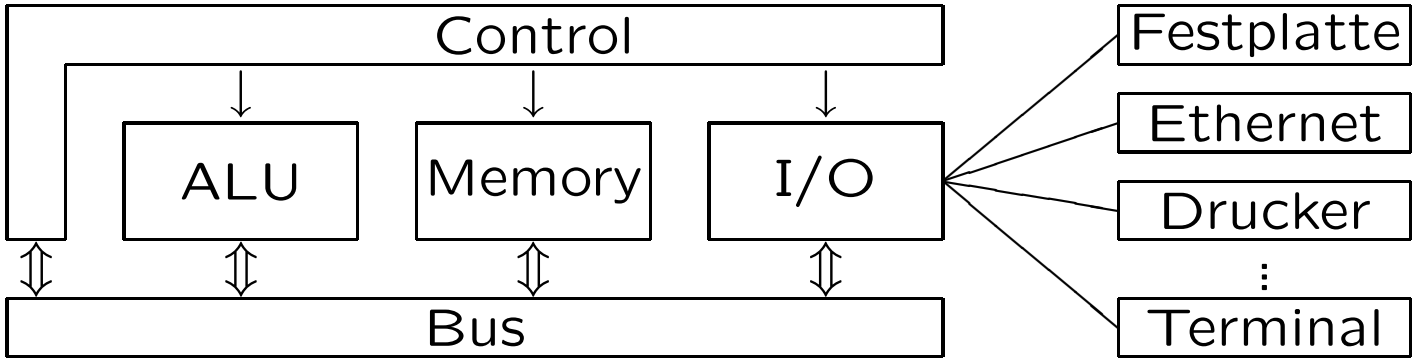
Maschinen und Architekturen

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
- Spezialregister und Spezialbefehle für Zugriffe darauf
- Prozessor fasst zentrale Teile auf einem Chip zusammen
- Virtuelle Adressierung
- 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, Übersetzer und Interpreter

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: $[A]$

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;
```

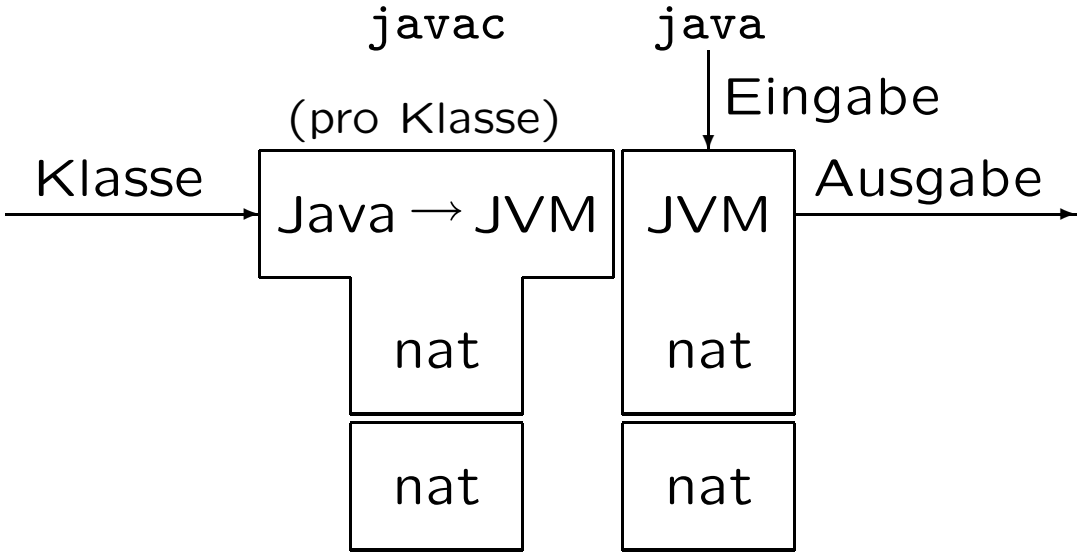
Bestandteile eines Programms

```
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
}
```

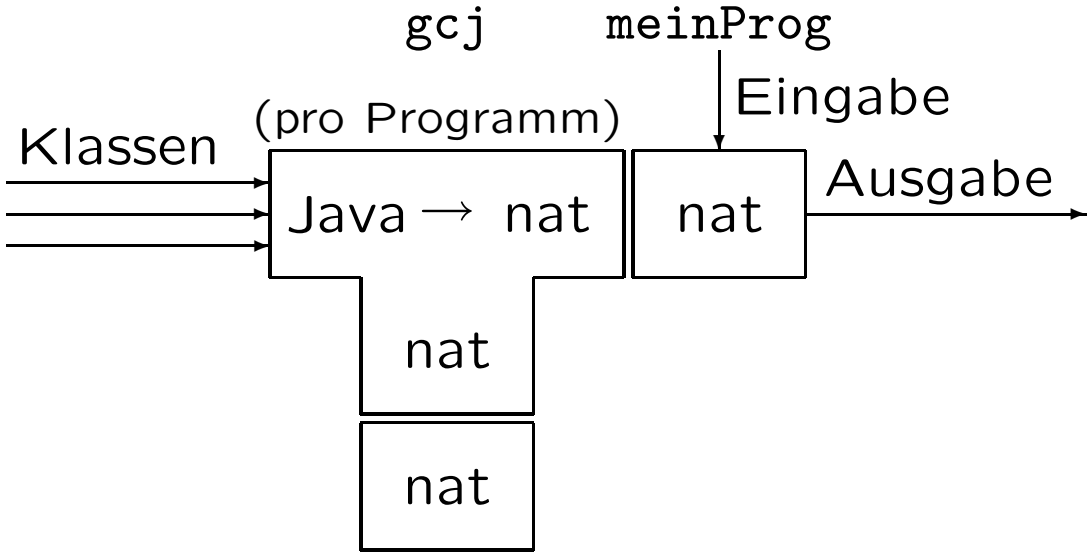
Statisch versus Dynamisch

- 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
- dynamischen: Programme leicht schreibbar, flexibler

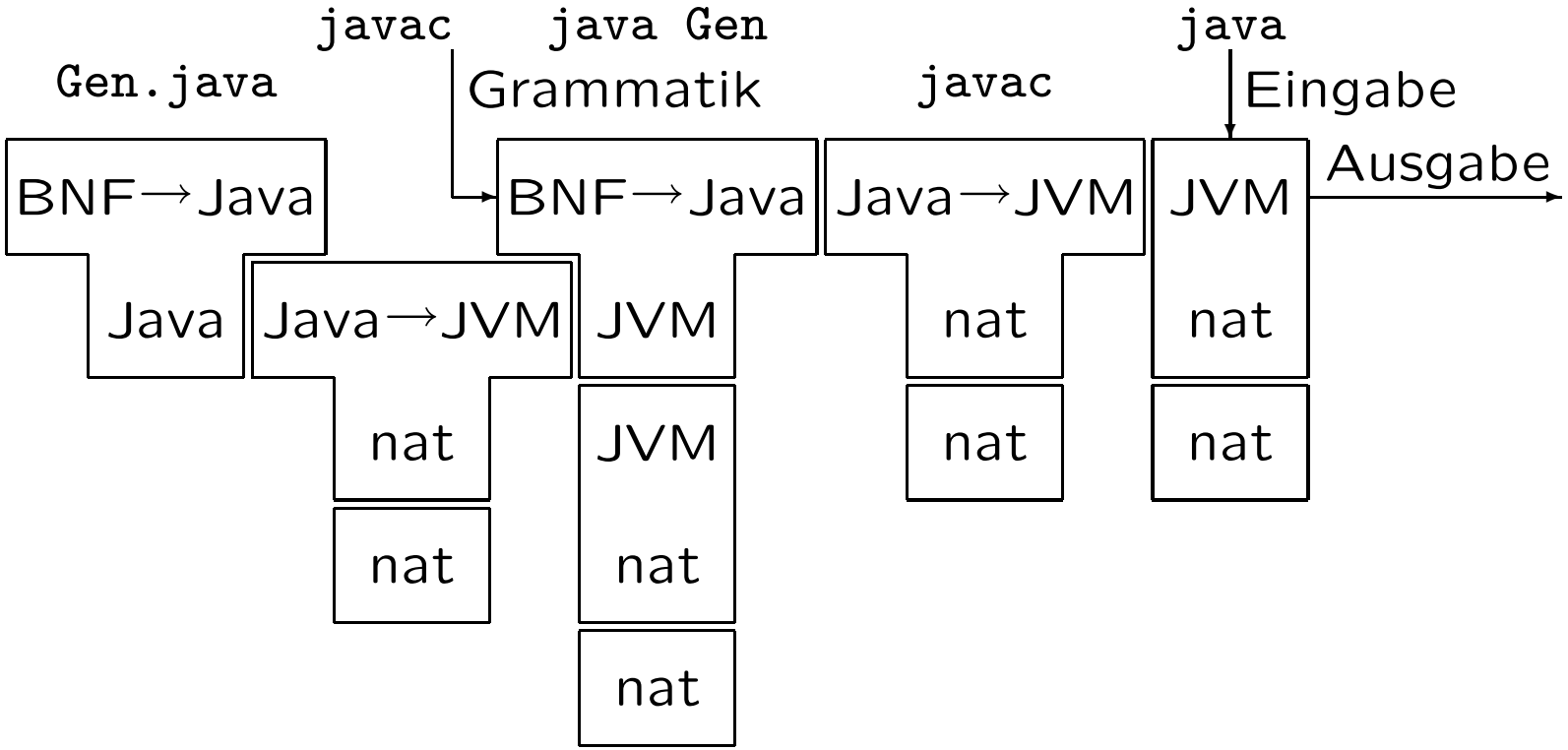
Compiler und Interpreter



Übersetzung ohne Zwischencode



Mehrere Übersetzungsschritte



Funktionsweise eines Interpreters

- Bei Ausführung ständig wiederholt (wie im Prozessor):
  1. hole nächsten Befehl aus Speicher (durch PC adressiert)
  2. erhöhe PC um 1
  3. interpretiere Befehl und führe ihn aus
- Vergleich mit Ausführung von `Zahlenraten`:
  1. hole nächste Zahl von Eingabe
  2. wobei darauffolgende Zahl zur nächsten Eingabe wird
  3. vergleiche Zahlen und gib Meldung aus

Aufgaben eines Compilers

- Zielcode erzeugen, semantisch äquivalent zu Quellcode
- Vermeiden von Laufzeitfehlern
- statische Fehlermeldungen
- Warnungen
- Optimierungen

Denkweisen

Sprachen und Modelle

- Programmierer kommunizieren über Programmcode
- einfache und vollständige Modelle
- Präzision, gleichzeitig Abstraktion über Details
- *Funktion* von entscheidender Bedeutung
  (Methode, Prozedur, Routine, . . . oft mit Seiteneffekten)
- reine Funktionen ohne Seiteneffekte

Programmiersprachen und -paradigmen

nach wichtigster Abstraktionsform unterschieden:

imperativ: Befehle, Zuweisungen, Seiteneffekte,
Modell angelehnt an Rechnerarchitektur

prozedural: Prozeduren mit Seiteneffekten

objektorientiert: Objekte wichtiger als Prozeduren

deklarativ: mathematische Modelle, ohne Seiteneffekte

funktional: reine Funktionen

logikorientiert: Beweis logischer Aussage

Definition einfacher Funktionen

- Aufzählung aller Möglichkeiten:
$$1 + 1 \mapsto 2$$
$$1 + 2 \mapsto 3$$
$$1 + 3 \mapsto 4$$
$$\ldots \mapsto \ldots$$

- mit Parametern:
$$\texttt{true}\,\texttt{\&\&}\,x \mapsto x \qquad (UND)$$
$$\texttt{false}\,\texttt{\&\&}\,x \mapsto \texttt{false}$$
$$\texttt{true}\,\texttt{||}\,x \mapsto \texttt{true} \qquad (ODER)$$
$$\texttt{false}\,\texttt{||}\,x \mapsto x$$

- auch Bedingungen so definierbar: ($b\,\texttt{?}\,x\,\texttt{:}\,y$ – wenn b dann x sonst y)
$$\texttt{true}\,\texttt{?}\,x\,\texttt{:}\,y \mapsto x$$
$$\texttt{false}\,\texttt{?}\,x\,\texttt{:}\,y \mapsto y$$

- aber nicht alle Funktionen so definierbar

Lambda-Kalkül

- definiert mathematische (reine) Funktionen vollständig
- „Rechnen am Papier“
- Syntax: $e = v \mid e_1\, e_2 \mid \lambda v.e_3$ (v ist Variable bzw. Name)
- Semantik:

$$\lambda v.f \leftrightarrow \lambda u.[u/v]f \quad \text{wobei } u \notin \mathit{FV}(\lambda v.f) \qquad (\alpha)$$
$$(\lambda v.f)\, e \leftrightarrow [e/v]f \qquad (\beta)$$
$$\lambda v.(f\, v) \leftrightarrow f \quad \text{wobei } v \notin \mathit{FV}(f) \qquad (\eta)$$

$\mathit{FV}(e)$ = Menge aller freien Variablen in e

$[e/v]f$ erzeugt durch Ersetzung alle freien v in f durch e

Reduktionen im Lambda-Kalkül

- Regeln nur von links nach rechts (β und η)
- Ausdruck in Normalform wenn weder β noch η anwendbar
- $(\lambda v.v)\,1 \leftrightarrow (\lambda x.x)\,1 \leftrightarrow 1$
- $(\lambda v.v * (v + 1))\,3 \leftrightarrow 3 * (3 + 1) \mapsto 3 * 4 \mapsto 12$
- $((\lambda u.\lambda v.(u * u) + (v * v))\,2)\,3$
  $\leftrightarrow (\lambda v.(2 * 2) + (v * v))\,3$
  $\leftrightarrow (2 * 2) + (3 * 3)$ (Normalform)
  $\mapsto 4 + (3 * 3) \mapsto 4 + 9 \mapsto 13$

Beispiel: Funktionen als Argumente

Zur Vereinfachung: $F = (\lambda u.\lambda v.v < 2\,?\,v : ((u\,u)\,(v-1)) + v)$

Reduktion: $(F\,F)\,2$

$$\begin{aligned}
&\leftrightarrow (\lambda v.v < 2\,?\,v : ((F\,F)\,(v-1)) + v)\,2\\
&\leftrightarrow 2 < 2\,?\,2 : ((F\,F)\,(2-1)) + 2\\
&\mapsto ((F\,F)\,(2-1)) + 2\\
&\mapsto ((F\,F)\,1) + 2\\
&\leftrightarrow ((\lambda v.v < 2\,?\,v : (((F\,F)\,(v-1)) + v))\,1) + 2\\
&\leftrightarrow (1 < 2\,?\,1 : (((F\,F)\,(1-1)) + 1)) + 2\\
&\mapsto 1 + 2\\
&\mapsto 3
\end{aligned}$$

Eigenschaften des Lambda-Kalküls

- Turing-vollständig (kann alles Berechenbare berechnen)
- Endlos-Reduktionen möglich: $(\lambda v.v\,v)\,(\lambda v.v\,v)$
- Reihenfolge der Reduktionen bestimmt Berechnungsdauer
- Funktionen erster Ordnung
- keine Kontrollstrukturen nötig
- kann mit mehreren Argumenten umgehen (Currying)

Allgemeine Erkenntnisse

- nicht alle Probleme entscheidbar (= lösbar)
- viele unterschiedliche Turing-vollständige Systeme, die alle die gleichen entscheidbaren Probleme lösen können
- jedes Turing-vollständige System kann auch unentscheidbare Probleme ausdrücken (endlose Berechnungen)
- nicht entscheidbar, welche Probleme entscheidbar sind
- Folgerung: Systeme, die nur entscheidbare Probleme ausdrücken können, sind nicht Turing-vollständig

Zusicherungen als Kommentare

```
zahl = (new Random()).nextInt() % grenze;
// -grenze < zahl < grenze     (wegen ... % grenze)
if (zahl < 0) {
    // -grenze < zahl < 0      (wegen Bedingung zahl < 0)
    zahl = zahl + grenze;
    // 0 < zahl < grenze       (wegen Addition von grenze)
}
// 0 < zahl < grenze           (wenn Bedingung wahr war)
// oder 0 <= zahl < grenze     (wenn Bedingung falsch war)
// ergibt: 0 <= zahl < grenze
```

assert-Anweisung in Java

```
zahl = (new Random()).nextInt() % grenze;
assert((-grenze < zahl) && (zahl < grenze));
if (zahl < 0) {
    assert((-grenze < zahl) && (zahl < 0));
    zahl = zahl + grenze;
    assert((0 < zahl) && (zahl < grenze));
}
assert((0 <= zahl) && (zahl < grenze));
```

Zusicherungen auf Schnittstellen

```
// Initialisierung mit Zufallszahl x; 0 <= x < grenze
// Voraussetzung: grenze > 0
public UnbekannteZahl (int grenze) { ... }
```

Vorbedingung: vor Methodenausführung erfüllt (Parameter)

Nachbedingung: nach Methodenausführung erfüllt

Softwareentwicklung

Softwarelebenszyklus

- Idee
- Entwicklung (development)
- Anwendung (use) und Wartung (maintenance)
- letzte Anwendung

Softwareentwicklungsschritte

zyklisch wiederholt (und oft überlappend):

- Analyse (analysis)
- Entwurf (design) !
- Implementiertung (implementation) !!
- Verifikation (verification) !
- Testen (testing) !
- Validierung (validation)

Schritte beim Programmieren

zyklisch wiederholt:

- Planen
- Editieren
- Übersetzen
- Testen
- Debuggen

Programmierwerkzeuge

- Editor
- Compiler und Interpreter
- Debugger
- Integrierte Entwicklungsumgebung

Softwarequalität (1)

Brauchbarkeit entsprechend Bedürfnissen der Anwender

- Zweckerfüllung
- Bedienbarkeit
- Effizienz

Zuverlässigkeit je nach Anwendungsbereich, Kostenfaktor

- Bewährtheit
- formale Korrektheit
- Fehlerresistenz

Softwarequalität (2)

Wartbarkeit als wichtiger Kostenfaktor

- Einfachheit
- Lesbarkeit
- Lokalität
- Faktorisierung

Festlegung von Softwareeigenschaften

- informelle Beschreibung
- Anwendungsfälle (use cases)
- Testfälle
- formale Spezifikation

Arten von Softwareeigenschaften

- funktionale Eigenschaften
- nichtfunktionale Eigenschaften