Abstrakte Maschinen (im weiteren Sinn)

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

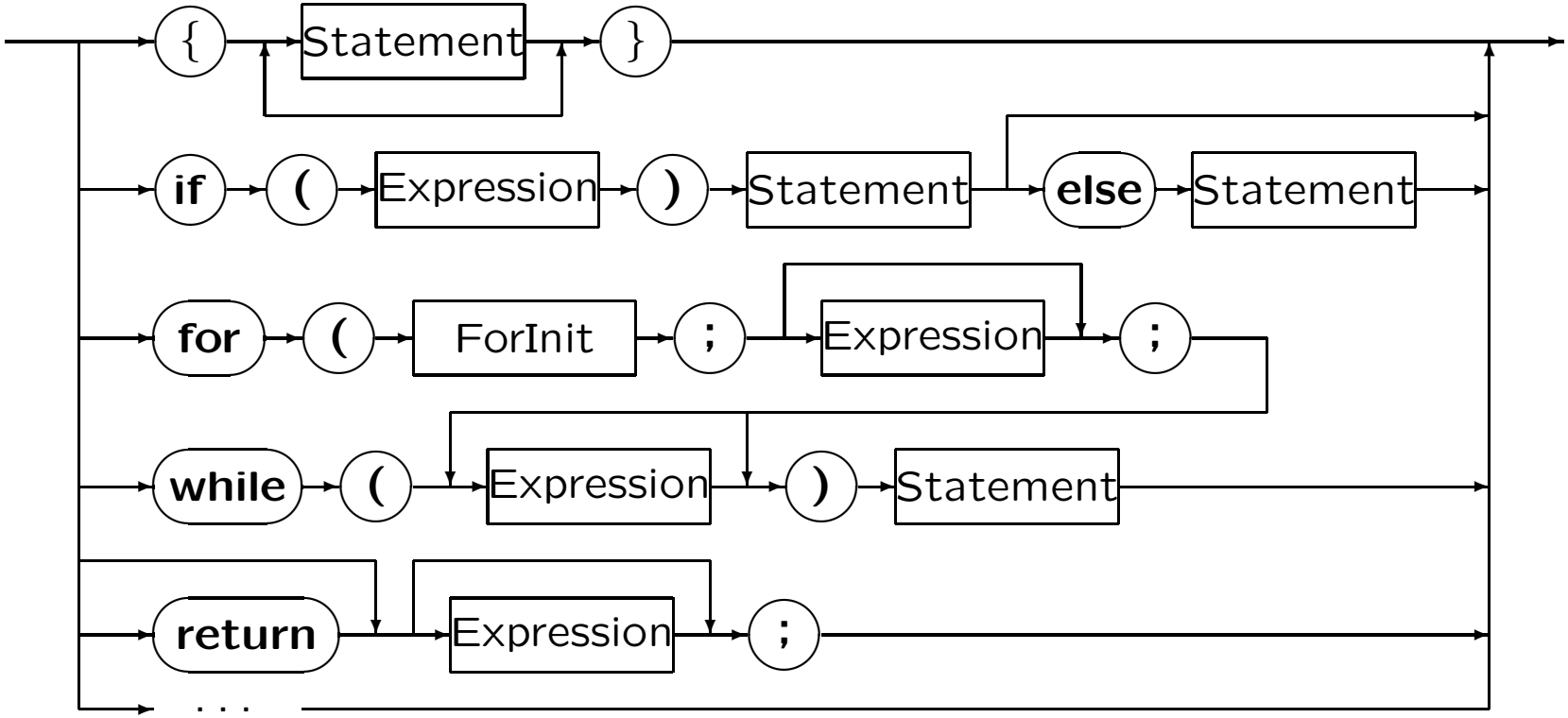
Grammatik (EBNF)

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

- Alternative $A|B$: X (**+**|**−**|*****) Y $\rightarrow$ X**+**Y, X**−**Y, X*****Y
- Wiederholung $\{A\}$: **{**$\{X\}$**}** $\rightarrow$ **{}**, **{**X**}**, **{**X X**}**, ...
- Option $[A]$: **if(**X**)** Y [**else** Z] $\rightarrow$ **if(**X**)** Y, **if(**X**)** Y **else** Z

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

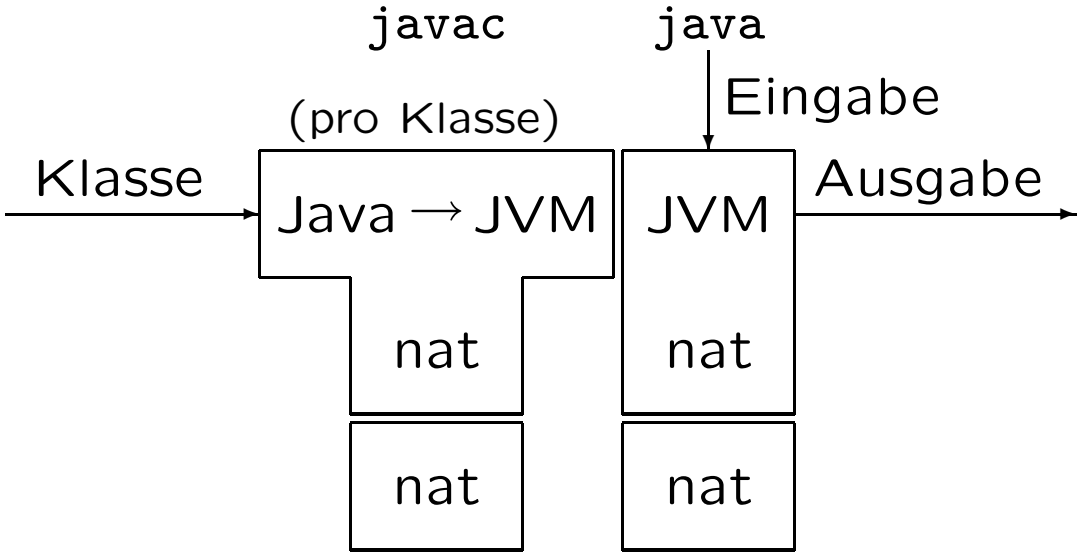
```
import java.util.Random;                    // Programmorganisation
public class UnbekannteZahl {
    private int zahl;                          // Datenstrukturen
    public UnbekannteZahl (int grenze) {
        zahl = (new Random()).nextInt() % grenze;
        if (zahl < 0) {
            zahl = 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
- Programmablauf ist *dynamisch*
- Daten in Datenstrukturen sind *dynamisch*
- statische versus dynamische Sprachen:
  - worauf mehr Wert gelegt wird
  - Grenzen fließend
- statisch: leichter lesbar, besser überprüft
- dynamisch: leichter schreibbar, flexibler

Compiler und Interpreter

Java-Programme: übliche Ausführung



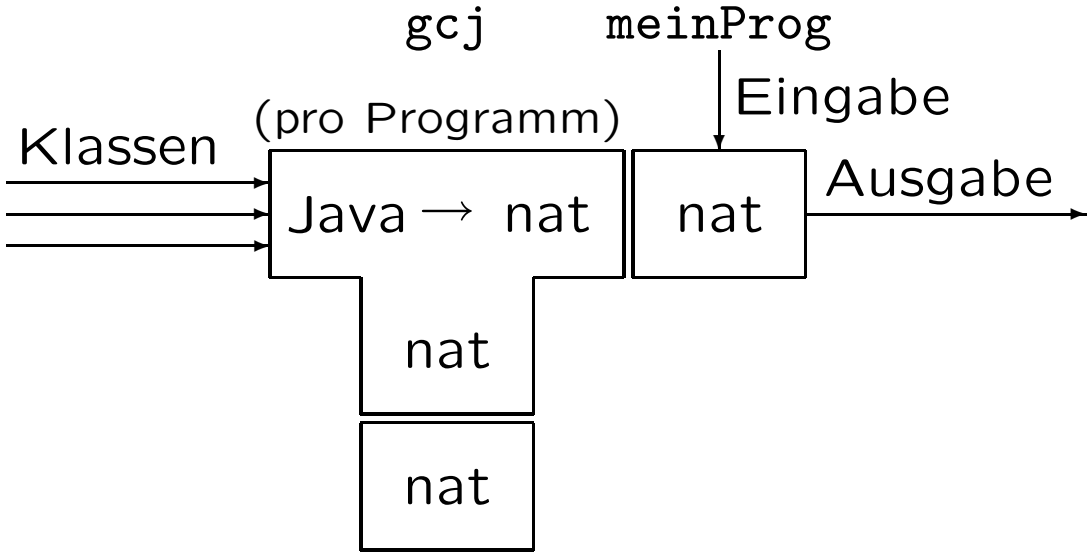
Funktionsweise eines Interpreters

- 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

Übersetzung ohne Zwischencode



Mehrere Übersetzungsschritte