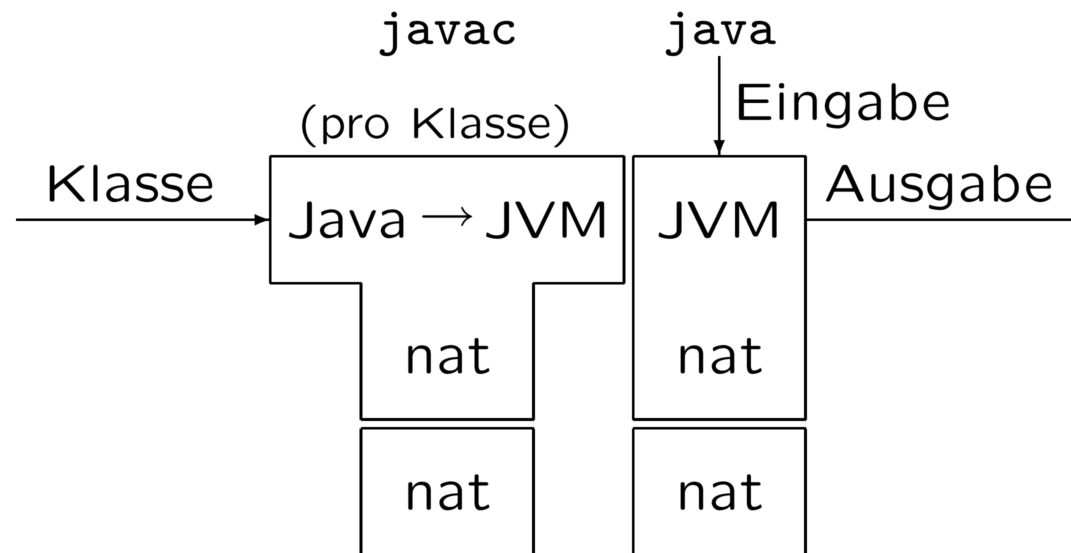
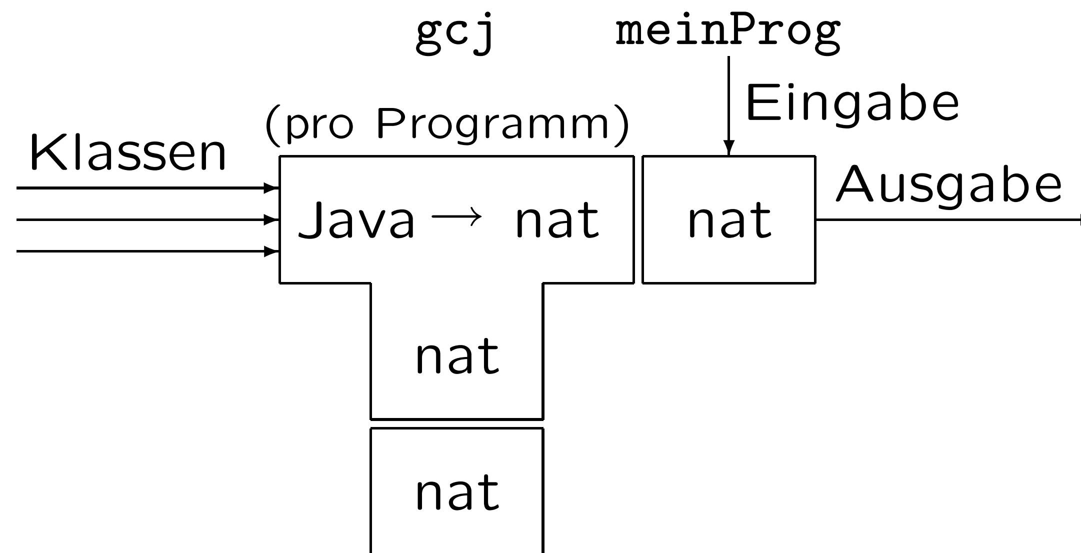
Compiler und Interpreter

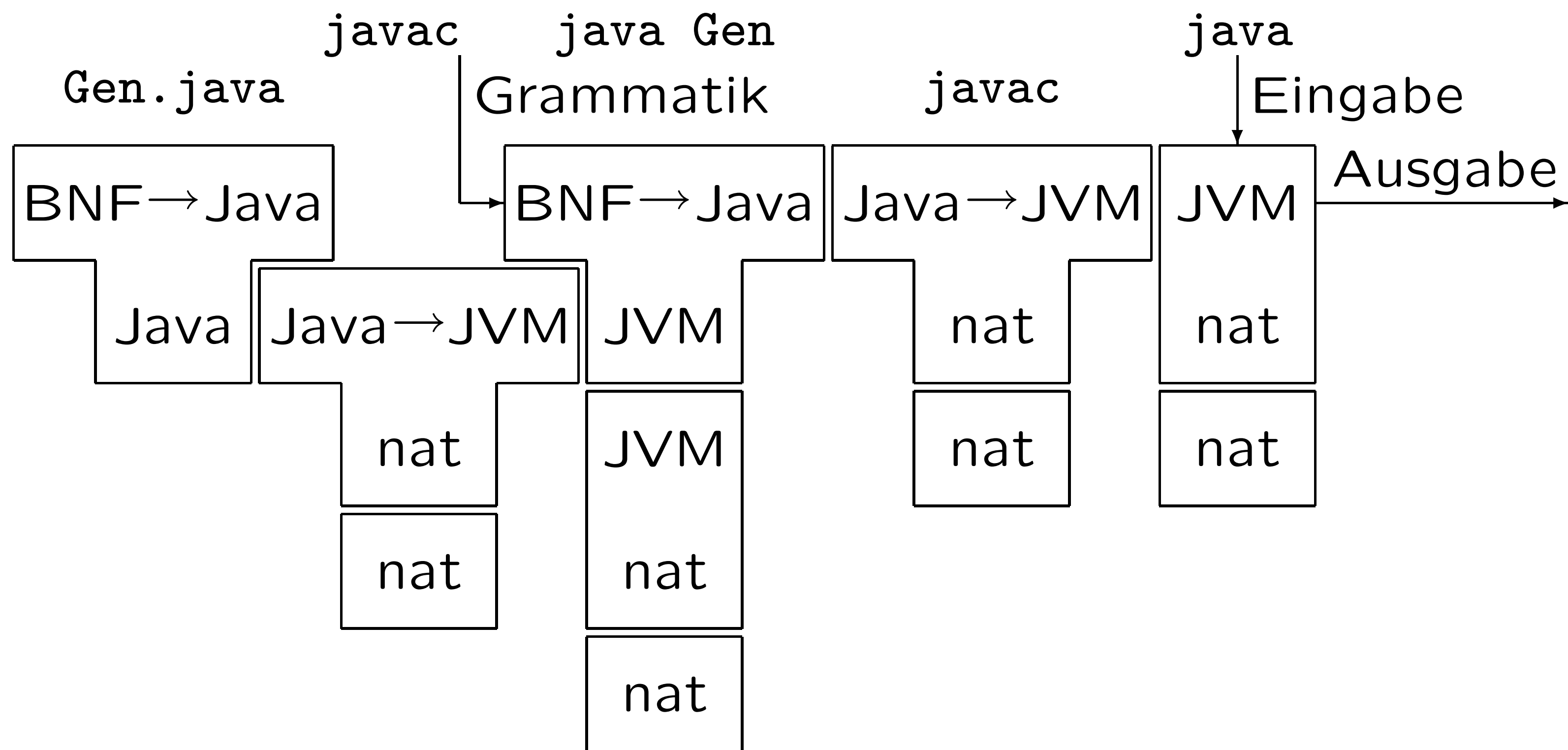
Java-Programme: übliche Ausführung



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 Aussagen

Definition einfacher Funktionen

- Aufzählung aller Möglichkeiten:
$$\begin{array}{rcl} 1+1 & \mapsto & 2 \\ 1+2 & \mapsto & 3 \\ 1+3 & \mapsto & 4 \\ \ldots & \mapsto & \ldots \end{array}$$

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

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

- 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:

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

$$(\lambda v.f)\, e \leftrightarrow [e/v]f \quad (\beta)$$

$$\lambda v.(f\, v) \leftrightarrow f \quad \text{wobei } v \notin \mathit{FV}(f) \quad (\eta)$$

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

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

Reductionen 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