Denkweisen

Sprachen und Modelle

Programmcode dient der Kommunikation zwischen **Programmierer(inn)en**

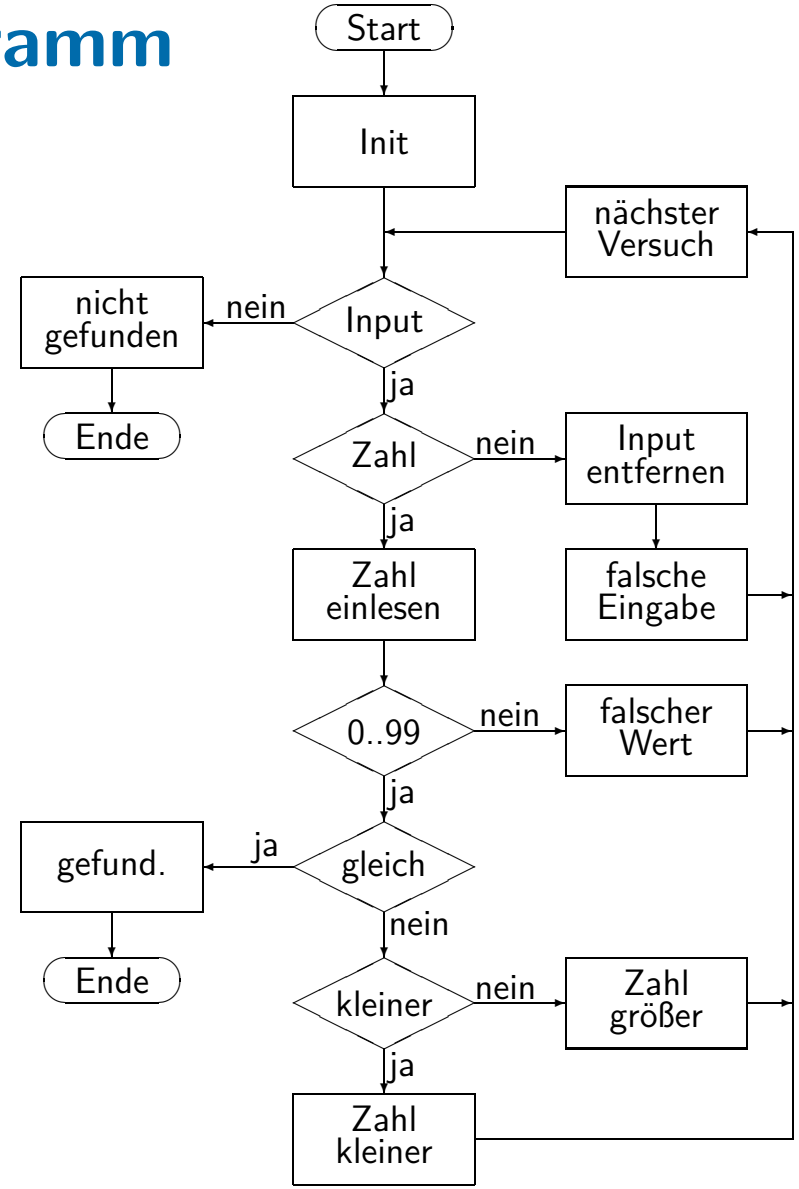
einfache und vollständige Modelle

Präzision, gleichzeitig Abstraktion über Details

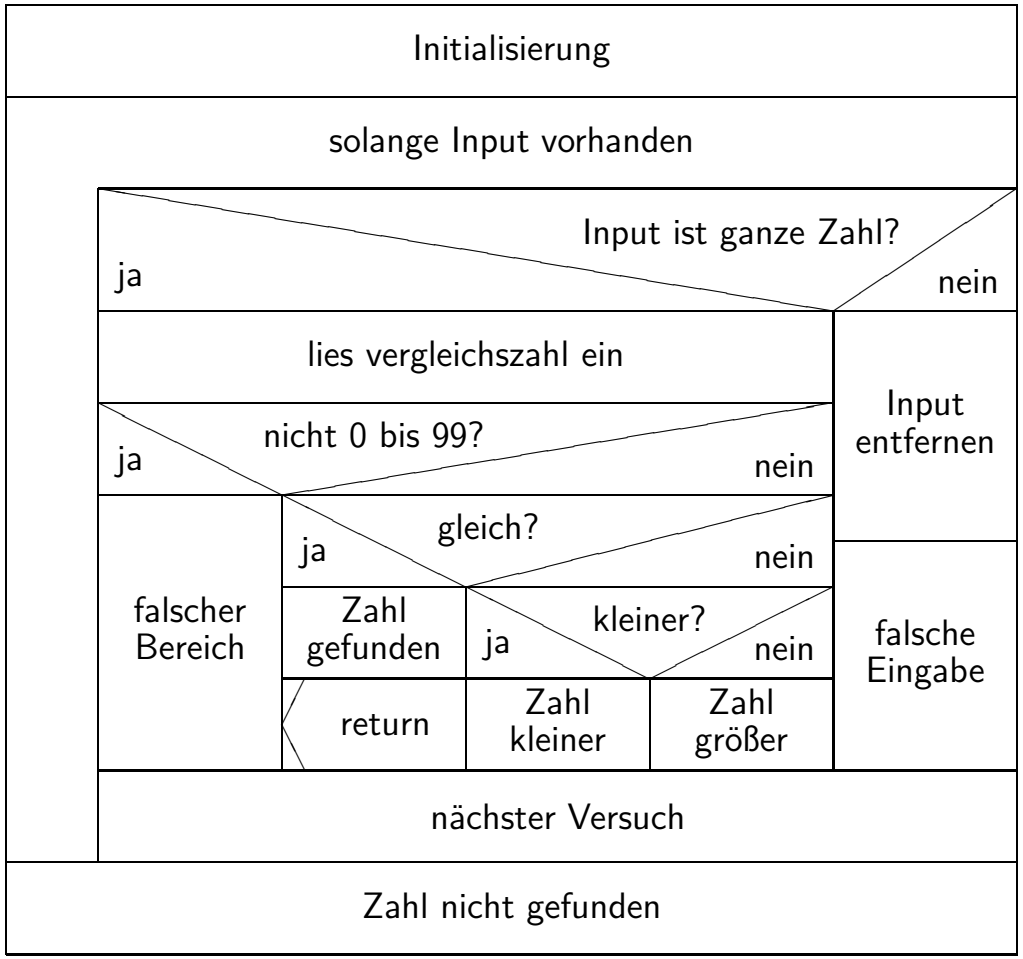
Funktionen von entscheidender Bedeutung

- Methoden, Prozeduren, Routinen, ... oft mit Seiteneffekten
- reine Funktionen ohne Seiteneffekte

Flussdiagramm



Struktogramm



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

Funktionen

Definition einfacher Abbildungen

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:
($b\,\texttt{?}\,x : y \;\equiv$ wenn b dann x sonst y)

$$\begin{array}{rcl} \texttt{true}\,\texttt{?}\,x : y & \mapsto & x \\ \texttt{false}\,\texttt{?}\,x : y & \mapsto & y \end{array}$$

aber nicht alle Funktionen so definierbar

Aufgabe: Bedingte Ausdrücke

Finden Sie in Gruppen zu zwei bis drei Personen Ausdrücke in Java, welche dieselbe Semantik wie die folgenden Ausdrücke haben, aber einfacher sind (wobei `x` und `y` Wahrheitswerte sind):

```
(x == true) ? true : false

(x == false) ? true : false

(x == y) ? (x || y) : (x && y)

(x == y) ? (x && y) : (x || y)
```

Zeit: 5 Minuten

Lambda-Kalkül

definiert mathematische (reine) Funktionen vollständig
und erlaubt „Rechnen am Papier“

Syntax: $e \;=\; v \;\mid\; e\,e' \;\mid\; \lambda v.e$ (v ist *Variable* $\equiv$ Name)

Semantik:

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

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

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

Reduktionen im Lambda-Kalkül

Reduktion = Regelanwendung nur von links nach rechts für β und η
bis weder β noch η anwendbar, dann **Normalform** erreicht

Beispiele: $(\lambda v.v)\,1 \leftrightarrow (\lambda x.x)\,1 \rightarrow 1$

$$(\lambda v.v * (v+1))\,3 \rightarrow 3 * (3+1) \mapsto 3 * 4 \mapsto 12$$

$$\begin{aligned}&((\lambda u.\lambda v.(u*u)+(v*v))\,2)\,3\\ &\rightarrow (\lambda v.(2*2)+(v*v))\,3\\ &\rightarrow (2*2)+(3*3) \qquad \text{(Normalform)}\\ &\mapsto 4+(3*3) \mapsto 4+9 \mapsto 13\end{aligned}$$

$\leftrightarrow$: Konversion (Äquivalenz) $\rightarrow$: Reduktion $\mapsto$: einfache Abbildung

Aufgabe: Konversion versus Reduktion

Warum sind die Definitionen von Reduktion und Normalform auf die β- und η-Regel beschränkt und beziehen nicht auch die α-Konversion mit ein?

A: Weil die Anwendungsrichtung für die α-Konversion keine Rolle spielt.

B: Weil derselbe Effekt auch durch β- und η-Reduktion erreichbar ist.

C: Weil Anwendungen der α-Regel in Reduktionen zu vermeiden sind.

D: Weil α-Konversionen in Ausdrücken der Form $\lambda x.e$ immer anwendbar sind.

Beispiel: Funktionen als Argumente

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

das ergibt:
$$F\,F = (\lambda u.\lambda v.v < 2\,?\,v : ((u\,u)\,(v-1)) + v)\,F$$
$$\rightarrow\ \lambda v.v < 2\,?\,v : ((F\,F)\,(v-1)) + v$$

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

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 ohne Ergebnisse)

nicht entscheidbar, welche Probleme entscheidbar sind

Folgerung: wenn nur entscheidbare Probleme ausdrückbar $\Rightarrow$ nicht vollständig