

WS 2017/18

Einführung in die Informatik 1

Felix Brandt, Harald Räcke

Fakultät für Informatik
TU München

<http://www14.in.tum.de/lehre/2017WS/info1/>

Wintersemester 2017/18

Diese Vorlesungsfolien basieren zum Grossteil auf der Einführungsvorlesung von Prof. Helmut Seidl (WS 2012/13).

WS 2017/18

Einführung in die Informatik 1

Felix Brandt, Harald Räcke

Fakultät für Informatik
TU München

<http://www14.in.tum.de/lehre/2017WS/info1/>

Wintersemester 2017/18

1 Vom Problem zum Programm

Ein **Problem** besteht darin, aus einer Menge von Informationen eine weitere (unbekannte) Information zu bestimmen.

mathematisch:

Ein Problem beschreibt eine Funktion $f : E \rightarrow A$, mit $E =$ zulässige Eingaben und $A =$ mögliche Ausgaben.

Beispiele:

Auflösen

Umkehrabb.

Suche in einem Suchraum, die Menge aller

Zustände S_0 und S_{Ziel} , die letzte Zug in Funktion

1 Vom Problem zum Programm

Ein **Problem** besteht darin, aus einer Menge von Informationen eine weitere (unbekannte) Information zu bestimmen.

mathematisch:

Ein Problem beschreibt eine Funktion $f : E \rightarrow A$, mit $E =$ zulässige Eingaben und $A =$ mögliche Ausgaben.

Beispiele:

1 Vom Problem zum Programm

Ein **Problem** besteht darin, aus einer Menge von Informationen eine weitere (unbekannte) Information zu bestimmen.

mathematisch:

Ein Problem beschreibt eine Funktion $f : E \rightarrow A$, mit $E =$ zulässige Eingaben und $A =$ mögliche Ausgaben.

Beispiele:

- ▶ Addition: $f : \mathbb{Q} \times \mathbb{Q} \rightarrow \mathbb{Q}$
- ▶ Primzahltest: $f : \mathbb{N} \rightarrow \{\text{yes, no}\}$
- ▶ Schach: $f : \mathcal{P} \rightarrow \mathcal{Z}$, wobei \mathcal{P} die Menge aller Schachpositionen ist, und $f(P)$, der beste Zug in Position P .

1 Vom Problem zum Programm

Ein **Problem** besteht darin, aus einer Menge von Informationen eine weitere (unbekannte) Information zu bestimmen.

mathematisch:

Ein Problem beschreibt eine Funktion $f : E \rightarrow A$, mit $E =$ zulässige Eingaben und $A =$ mögliche Ausgaben.

Beispiele:

- ▶ Addition: $f : \mathbb{Q} \times \mathbb{Q} \rightarrow \mathbb{Q}$
- ▶ Primzahltest: $f : \mathbb{N} \rightarrow \{\text{yes, no}\}$
- ▶ Schach: $f : \mathcal{P} \rightarrow \mathcal{Z}$, wobei \mathcal{P} die Menge aller Schachpositionen ist, und $f(P)$, der beste Zug in Position P .

1 Vom Problem zum Programm

Ein **Problem** besteht darin, aus einer Menge von Informationen eine weitere (unbekannte) Information zu bestimmen.

mathematisch:

Ein Problem beschreibt eine Funktion $f : E \rightarrow A$, mit $E =$ zulässige Eingaben und $A =$ mögliche Ausgaben.

Beispiele:

- ▶ Addition: $f : \mathbb{Q} \times \mathbb{Q} \rightarrow \mathbb{Q}$
- ▶ Primzahltest: $f : \mathbb{N} \rightarrow \{\text{yes, no}\}$
- ▶ Schach: $f : \mathcal{P} \rightarrow \mathcal{Z}$, wobei \mathcal{P} die Menge aller Schachpositionen ist, und $f(P)$, der beste Zug in Position P .

Ein **Algorithmus** ist ein **exaktes Verfahren** zur Lösung eines Problems, d.h. zur Bestimmung der gewünschten Resultate.

Man sagt auch ein Algorithmus **berechnet** eine Funktion f .



Ausschnitt aus Briefmarke, Soviet Union 1983
Public Domain [↗](#)

Abu Abdallah
Muhamed ibn Musa
al-Chwarizmi, ca.
780–835

1 Vom Problem zum Programm

Ein **Problem** besteht darin, aus einer Menge von Informationen eine weitere (unbekannte) Information zu bestimmen.

mathematisch:

Ein Problem beschreibt eine Funktion $f : E \rightarrow A$, mit $E =$ zulässige Eingaben und $A =$ mögliche Ausgaben.

Beispiele:

- ▶ Addition: $f : \mathbb{Q} \times \mathbb{Q} \rightarrow \mathbb{Q}$
- ▶ Primzahltest: $f : \mathbb{N} \rightarrow \{\text{yes, no}\}$
- ▶ Schach: $f : \mathcal{P} \rightarrow \mathbb{Z}$, wobei \mathcal{P} die Menge aller Schachpositionen ist, und $f(P)$, der beste Zug in Position P .

Algorithmus

Beobachtung:

Nicht jedes Problem lässt sich durch einen Algorithmus lösen (↑**Berechenbarkeitstheorie**).

Beweisidee: (↑Diskrete Strukturen)

- ▶ es gibt überabzählbar unendlich viele Probleme
- ▶ es gibt abzählbar unendlich viele Algorithmen

Algorithmus

Ein **Algorithmus** ist ein **exaktes Verfahren** zur Lösung eines Problems, d.h. zur Bestimmung der gewünschten Resultate.

Man sagt auch ein Algorithmus **berechnet** eine Funktion f .



Ausschnitt aus Briefmarke, Soviet Union 1983
Public Domain [↗](#)

Abu Abdallah
Muhamed ibn Musa
al-Chwarizmi, ca.
780–835

Algorithmus

Beobachtung:

Nicht jedes Problem läßt sich durch einen Algorithmus lösen (↑**Berechenbarkeitstheorie**).

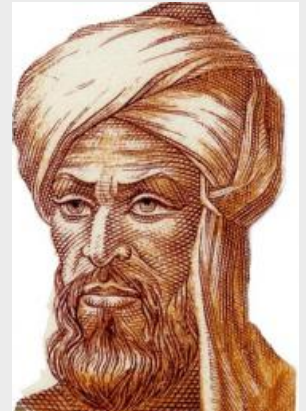
Beweisidee: (↑**Diskrete Strukturen**)

- ▶ es gibt **überabzählbar unendlich** viele Probleme
- ▶ es gibt **abzählbar unendlich** viele Algorithmen

Algorithmus

Ein **Algorithmus** ist ein **exaktes Verfahren** zur Lösung eines Problems, d.h. zur Bestimmung der gewünschten Resultate.

Man sagt auch ein Algorithmus **berechnet** eine Funktion f .



Ausschnitt aus Briefmarke, Soviet Union 1983
Public Domain [↗](#)

Abu Abdallah
Muhamed ibn Musa
al-Chwarizmi, ca.
780–835

Algorithmus

Das **exakte Verfahren** besteht i.a. darin, eine Abfolge von **elementaren Einzelschritten** der Verarbeitung festzulegen.

Beispiel: Alltagsalgorithmen

<i>Resultat</i>	<i>Algorithmus</i>	<i>Einzelschritte</i>
Pullover	Strickmuster	eine links, eine rechts, eine fallen lassen
Kuchen	Rezept	nimm 3 Eier ...
Konzert	Partitur	Noten

Algorithmus

Beobachtung:

Nicht jedes Problem läßt sich durch einen Algorithmus lösen (↑**Berechenbarkeitstheorie**).

Beweisidee: (↑**Diskrete Strukturen**)

- ▶ es gibt **überabzählbar unendlich** viele Probleme
- ▶ es gibt **abzählbar unendlich** viele Algorithmen

Beispiel: Euklidischer Algorithmus

Problem: geg. $a, b \in \mathbb{N}, a, b \neq 0$. Bestimme $\text{ggT}(a, b)$.

Algorithmus:

1. Falls $a = b$ brich Berechnung ab. Es gilt $\text{ggT}(a, b) = a$.
Ansonsten gehe zu Schritt 2.
2. Falls $a > b$, ersetze a durch $a - b$ und setze Berechnung in Schritt 1 fort. Ansonsten gehe zu Schritt 3.
3. Es gilt $a < b$. Ersetze b durch $b - a$ und setze Berechnung in Schritt 1 fort.

Algorithmus

Das **exakte Verfahren** besteht i.a. darin, eine Abfolge von **elementaren Einzelschritten** der Verarbeitung festzulegen.

Beispiel: Alltagsalgorithmen

<i>Resultat</i>	<i>Algorithmus</i>	<i>Einzelschritte</i>
Pullover	Strickmuster	eine links, eine rechts, eine fallen lassen
Kuchen	Rezept	nimm 3 Eier ...
Konzert	Partitur	Noten

Beispiel: Euklidischer Algorithmus

Warum geht das?

Wir zeigen, für $a > b$: $\text{ggT}(a, b) = \text{ggT}(a - b, b)$.

Seien $g = \text{ggT}(a, b)$, $g' = \text{ggT}(a - b, b)$.

Dann gilt:

Beispiel: Euklidischer Algorithmus

Problem: geg. $a, b \in \mathbb{N}, a, b \neq 0$. Bestimme $\text{ggT}(a, b)$.

Algorithmus:

1. Falls $a = b$ brich Berechnung ab. Es gilt $\text{ggT}(a, b) = a$.
Ansonsten gehe zu Schritt 2.
2. Falls $a > b$, ersetze a durch $a - b$ und setze Berechnung in Schritt 1 fort. Ansonsten gehe zu Schritt 3.
3. Es gilt $a < b$. Ersetze b durch $b - a$ und setze Berechnung in Schritt 1 fort.

Beispiel: Euklidischer Algorithmus

Warum geht das?

Wir zeigen, für $a > b$: $\text{ggT}(a, b) = \text{ggT}(a - b, b)$.

Seien $g = \text{ggT}(a, b)$, $g' = \text{ggT}(a - b, b)$.

Dann gilt:

$$\begin{array}{l} a = q_a \cdot g \\ b = q_b \cdot g \end{array} \quad \text{und} \quad \begin{array}{l} a - b = q'_{a-b} \cdot g' \\ b = q'_b \cdot g' \end{array}$$

Beispiel: Euklidischer Algorithmus

Problem: geg. $a, b \in \mathbb{N}, a, b \neq 0$. Bestimme $\text{ggT}(a, b)$.

Algorithmus:

1. Falls $a = b$ brich Berechnung ab. Es gilt $\text{ggT}(a, b) = a$. Ansonsten gehe zu Schritt 2.
2. Falls $a > b$, ersetze a durch $a - b$ und setze Berechnung in Schritt 1 fort. Ansonsten gehe zu Schritt 3.
3. Es gilt $a < b$. Ersetze b durch $b - a$ und setze Berechnung in Schritt 1 fort.

Beispiel: Euklidischer Algorithmus

Warum geht das?

Wir zeigen, für $a > b$: $\text{ggT}(a, b) = \text{ggT}(a - b, b)$.

Seien $g = \text{ggT}(a, b)$, $g' = \text{ggT}(a - b, b)$.

Dann gilt:

$$\begin{array}{l} a = q_a \cdot g \\ b = q_b \cdot g \end{array} \quad \text{und} \quad \begin{array}{l} a - b = q'_{a-b} \cdot g' \\ b = q'_b \cdot g' \end{array}$$

$$\begin{array}{l} a - b = (q_a - q_b) \cdot g \\ b = q_b \cdot g \end{array} \quad \text{und} \quad \begin{array}{l} a = (q'_{a-b} + q'_b) \cdot g' \\ b = q'_b \cdot g' \end{array}$$

Beispiel: Euklidischer Algorithmus

Problem: geg. $a, b \in \mathbb{N}, a, b \neq 0$. Bestimme $\text{ggT}(a, b)$.

Algorithmus:

1. Falls $a = b$ brich Berechnung ab. Es gilt $\text{ggT}(a, b) = a$. Ansonsten gehe zu Schritt 2.
2. Falls $a > b$, ersetze a durch $a - b$ und setze Berechnung in Schritt 1 fort. Ansonsten gehe zu Schritt 3.
3. Es gilt $a < b$. Ersetze b durch $b - a$ und setze Berechnung in Schritt 1 fort.

Beispiel: Euklidischer Algorithmus

Warum geht das?

Wir zeigen, für $a > b$: $\text{ggT}(a, b) = \text{ggT}(a - b, b)$.

Seien $g = \text{ggT}(a, b)$, $g' = \text{ggT}(a - b, b)$.

Dann gilt:

$$\begin{array}{l} a = q_a \cdot g \\ b = q_b \cdot g \end{array} \quad \text{und} \quad \begin{array}{l} a - b = q'_{a-b} \cdot g' \\ b = q'_b \cdot g' \end{array}$$

$$\begin{array}{l} a - b = (q_a - q_b) \cdot g \\ b = q_b \cdot g \end{array} \quad \text{und} \quad \begin{array}{l} a = (q'_{a-b} + q'_b) \cdot g' \\ b = q'_b \cdot g' \end{array}$$

Das heißt g ist Teiler von $a - b, b$ und g' ist Teiler von a, b .

Daraus folgt $g \leq g'$ und $g' \leq g$, also $g = g'$.

Beispiel: Euklidischer Algorithmus

Problem: geg. $a, b \in \mathbb{N}, a, b \neq 0$. Bestimme $\text{ggT}(a, b)$.

Algorithmus:

1. Falls $a = b$ brich Berechnung ab. Es gilt $\text{ggT}(a, b) = a$. Ansonsten gehe zu Schritt 2.
2. Falls $a > b$, ersetze a durch $a - b$ und setze Berechnung in Schritt 1 fort. Ansonsten gehe zu Schritt 3.
3. Es gilt $a < b$. Ersetze b durch $b - a$ und setze Berechnung in Schritt 1 fort.

Eigenschaften

(statische) Finitheit. Die Beschreibung des Algorithmus besitzt endliche Länge. (↑nichtuniforme Algorithmen)

(dynamische) Finitheit. Die bei Abarbeitung entstehenden Zwischenergebnisse sind endlich.

Terminiertheit. Algorithmen, die nach endlich vielen Schritten ein Resultat liefern, heißen **terminierend**. (↑Betriebssysteme, ↑reaktive Systeme)

Determiniertheit. Bei gleichen Eingabedaten gibt ein Algorithmus das gleiche Ergebnis aus. (↑randomisierte Algorithmen, ↑nicht-deterministische Algorithmen)

Determinismus. Der nächste anzuwendende Schritt im Verfahren ist stets eindeutig definiert. (↑randomisierte Algorithmen, ↑nicht-deterministische Algorithmen)

Beispiel: Euklidischer Algorithmus

Warum geht das?

Wir zeigen, für $a > b$: $\text{ggT}(a, b) = \text{ggT}(a - b, b)$.

Seien $g = \text{ggT}(a, b)$, $g' = \text{ggT}(a - b, b)$.

Dann gilt:

$$\begin{array}{l} a = q_a \cdot g \\ b = q_b \cdot g \end{array} \quad \text{und} \quad \begin{array}{l} a - b = q'_{a-b} \cdot g' \\ b = q'_b \cdot g' \end{array}$$

$$\begin{array}{l} a - b = (q_a - q_b) \cdot g \\ b = q_b \cdot g \end{array} \quad \text{und} \quad \begin{array}{l} a = (q'_{a-b} + q'_b) \cdot g' \\ b = q'_b \cdot g' \end{array}$$

Das heißt g ist Teiler von $a - b, b$ und g' ist Teiler von a, b .

Daraus folgt $g \leq g'$ und $g' \leq g$, also $g = g'$.

Eigenschaften

(statische) Finitheit. Die Beschreibung des Algorithmus besitzt endliche Länge. (↑**nichtuniforme Algorithmen**)

(dynamische) Finitheit. Die bei Abarbeitung entstehenden Zwischenergebnisse sind endlich.

Terminiertheit. Algorithmen, die nach endlich vielen Schritten ein Resultat liefern, heißen **terminierend**. (↑**Betriebssysteme, reaktive Systeme**)

Determiniertheit. Bei gleichen Eingabedaten gibt ein Algorithmus das gleiche Ergebnis aus. (↑**randomisierte Algorithmen, nicht-deterministische Algorithmen**)

Determinismus. Der nächste anzuwendende Schritt im Verfahren ist stets eindeutig definiert. (↑**randomisierte Algorithmen, nicht-deterministische Algorithmen**)

Beispiel: Euklidischer Algorithmus

Warum geht das?

Wir zeigen, für $a > b$: $\text{ggT}(a, b) = \text{ggT}(a - b, b)$.

Seien $g = \text{ggT}(a, b)$, $g' = \text{ggT}(a - b, b)$.

Dann gilt:

$$\begin{array}{l} a = q_a \cdot g \\ b = q_b \cdot g \end{array} \quad \text{und} \quad \begin{array}{l} a - b = q'_{a-b} \cdot g' \\ b = q'_b \cdot g' \end{array}$$

$$\begin{array}{l} a - b = (q_a - q_b) \cdot g \\ b = q_b \cdot g \end{array} \quad \text{und} \quad \begin{array}{l} a = (q'_{a-b} + q'_b) \cdot g' \\ b = q'_b \cdot g' \end{array}$$

Das heißt g ist Teiler von $a - b, b$ und g' ist Teiler von a, b .

Daraus folgt $g \leq g'$ und $g' \leq g$, also $g = g'$.

Eigenschaften

(statische) Finitheit. Die Beschreibung des Algorithmus besitzt endliche Länge. (↑**nichtuniforme Algorithmen**)

(dynamische) Finitheit. Die bei Abarbeitung entstehenden Zwischenergebnisse sind endlich.

Terminiertheit. Algorithmen, die nach endlich vielen Schritten ein Resultat liefern, heißen **terminierend**. (↑**Betriebssysteme**, ↑**reaktive Systeme**)

Determiniertheit. Bei gleichen Eingabedaten gibt ein Algorithmus das gleiche Ergebnis aus. (↑**randomisierte Algorithmen**, ↑**nicht-deterministische Algorithmen**)

Determinismus. Der nächste anzuwendende Schritt im Verfahren ist stets eindeutig definiert. (↑**randomisierte Algorithmen**, ↑**nicht-deterministische Algorithmen**)

Beispiel: Euklidischer Algorithmus

Warum geht das?

Wir zeigen, für $a > b$: $\text{ggT}(a, b) = \text{ggT}(a - b, b)$.

Seien $g = \text{ggT}(a, b)$, $g' = \text{ggT}(a - b, b)$.

Dann gilt:

$$\begin{array}{l} a = q_a \cdot g \\ b = q_b \cdot g \end{array} \quad \text{und} \quad \begin{array}{l} a - b = q'_{a-b} \cdot g' \\ b = q'_b \cdot g' \end{array}$$

$$\begin{array}{l} a - b = (q_a - q_b) \cdot g \\ b = q_b \cdot g \end{array} \quad \text{und} \quad \begin{array}{l} a = (q'_{a-b} + q'_b) \cdot g' \\ b = q'_b \cdot g' \end{array}$$

Das heißt g ist Teiler von $a - b, b$ und g' ist Teiler von a, b .

Daraus folgt $g \leq g'$ und $g' \leq g$, also $g = g'$.

Eigenschaften

(statische) Finitheit. Die Beschreibung des Algorithmus besitzt endliche Länge. (↑**nichtuniforme Algorithmen**)

(dynamische) Finitheit. Die bei Abarbeitung entstehenden Zwischenergebnisse sind endlich.

Terminiertheit. Algorithmen, die nach endlich vielen Schritten ein Resultat liefern, heißen **terminierend**. (↑**Betriebssysteme**, ↑**reaktive Systeme**)

Determiniertheit. Bei gleichen Eingabedaten gibt ein Algorithmus das gleiche Ergebnis aus. (↑**randomisierte Algorithmen**, ↑**nicht-deterministische Algorithmen**)

Determinismus. Der nächste anzuwendende Schritt im Verfahren ist stets eindeutig definiert. (↑**randomisierte Algorithmen**, ↑**nicht-deterministische Algorithmen**)

Beispiel: Euklidischer Algorithmus

Warum geht das?

Wir zeigen, für $a > b$: $\text{ggT}(a, b) = \text{ggT}(a - b, b)$.

Seien $g = \text{ggT}(a, b)$, $g' = \text{ggT}(a - b, b)$.

Dann gilt:

$$\begin{array}{l} a = q_a \cdot g \\ b = q_b \cdot g \end{array} \quad \text{und} \quad \begin{array}{l} a - b = q'_{a-b} \cdot g' \\ b = q'_b \cdot g' \end{array}$$

$$\begin{array}{l} a - b = (q_a - q_b) \cdot g \\ b = q_b \cdot g \end{array} \quad \text{und} \quad \begin{array}{l} a = (q'_{a-b} + q'_b) \cdot g' \\ b = q'_b \cdot g' \end{array}$$

Das heißt g ist Teiler von $a - b, b$ und g' ist Teiler von a, b .

Daraus folgt $g \leq g'$ und $g' \leq g$, also $g = g'$.

Eigenschaften

(statische) Finitheit. Die Beschreibung des Algorithmus besitzt endliche Länge. (↑**nichtuniforme Algorithmen**)

(dynamische) Finitheit. Die bei Abarbeitung entstehenden Zwischenergebnisse sind endlich.

Terminiertheit. Algorithmen, die nach endlich vielen Schritten ein Resultat liefern, heißen **terminierend**. (↑**Betriebssysteme**, ↑**reaktive Systeme**)

Determiniertheit. Bei gleichen Eingabedaten gibt ein Algorithmus das gleiche Ergebnis aus. (↑**randomisierte Algorithmen**, ↑**nicht-deterministische Algorithmen**)

Determinismus. Der nächste anzuwendende Schritt im Verfahren ist stets eindeutig definiert. (↑**randomisierte Algorithmen**, ↑**nicht-deterministische Algorithmen**)

Beispiel: Euklidischer Algorithmus

Warum geht das?

Wir zeigen, für $a > b$: $\text{ggT}(a, b) = \text{ggT}(a - b, b)$.

Seien $g = \text{ggT}(a, b)$, $g' = \text{ggT}(a - b, b)$.

Dann gilt:

$$\begin{array}{l} a = q_a \cdot g \\ b = q_b \cdot g \end{array} \quad \text{und} \quad \begin{array}{l} a - b = q'_{a-b} \cdot g' \\ b = q'_b \cdot g' \end{array}$$

$$\begin{array}{l} a - b = (q_a - q_b) \cdot g \\ b = q_b \cdot g \end{array} \quad \text{und} \quad \begin{array}{l} a = (q'_{a-b} + q'_b) \cdot g' \\ b = q'_b \cdot g' \end{array}$$

Das heißt g ist Teiler von $a - b, b$ und g' ist Teiler von a, b .

Daraus folgt $g \leq g'$ und $g' \leq g$, also $g = g'$.

Programm

Ein **Programm** ist die **Formulierung** eines Algorithmus in einer **Programmiersprache**.

Die Formulierung gestattet (hoffentlich) eine maschinelle Ausführung.

- ▶ Ein **Programmsystem** berechnet i.a. nicht nur eine Funktion, sondern **immer wieder** Funktionen in Interaktion mit Benutzerinnen und/oder der Umgebung.
- ▶ Es gibt viele Programmiersprachen: **Java**, **C**, **Prolog**, **Fortran**, **TeX**, **PostScript**, ...

Eigenschaften

(statische) Finitheit. Die Beschreibung des Algorithmus besitzt endliche Länge. (↑**nichtuniforme Algorithmen**)

(dynamische) Finitheit. Die bei Abarbeitung entstehenden Zwischenergebnisse sind endlich.

Terminiertheit. Algorithmen, die nach endlich vielen Schritten ein Resultat liefern, heißen **terminierend**. (↑**Betriebssysteme**, ↑**reaktive Systeme**)

Determiniertheit. Bei gleichen Eingabedaten gibt ein Algorithmus das gleiche Ergebnis aus. (↑**randomisierte Algorithmen**, ↑**nicht-deterministische Algorithmen**)

Determinismus. Der nächste anzuwendende Schritt im Verfahren ist stets eindeutig definiert. (↑**randomisierte Algorithmen**, ↑**nicht-deterministische Algorithmen**)

Programm

Ein Programm ist **gut**, wenn

- ▶ **die Programmiererin** in ihr algorithmische Ideen **natürlich** beschreiben kann, insbesondere später noch versteht was das Programm tut (oder nicht tut);
- ▶ **ein Computer** das Programm leicht verstehen und **effizient** ausführen kann.

Programm

Ein **Programm** ist die **Formulierung** eines Algorithmus in einer **Programmiersprache**.

Die Formulierung gestattet (hoffentlich) eine maschinelle Ausführung.

- ▶ Ein **Programmsystem** berechnet i.a. nicht nur eine Funktion, sondern **immer wieder** Funktionen in Interaktion mit Benutzerinnen und/oder der Umgebung.
- ▶ Es gibt viele Programmiersprachen: **Java**, **C**, **Prolog**, **Fortran**, **TeX**, **PostScript**, ...

2 Eine einfache Programmiersprache

Eine Programmiersprache soll

- ▶ Datenstrukturen anbieten
- ▶ Operationen auf Daten erlauben
- ▶ **Kontrollstrukturen** zur Ablaufsteuerung bereitstellen

Als Beispiel betrachten wir **MiniJava**.

Variablen

Variablen dienen zur Speicherung von Daten.

Um Variablen in **MiniJava** zu nutzen müssen sie zunächst eingeführt, d.h. **deklariert** werden.

2 Eine einfache Programmiersprache

Eine Programmiersprache soll

- ▶ Datenstrukturen anbieten
- ▶ Operationen auf Daten erlauben
- ▶ **Kontrollstrukturen** zur Ablaufsteuerung bereitstellen

Als Beispiel betrachten wir **MiniJava**.

Variablen

Beispiel:

```
int x, result;
```

Diese Deklaration führt die beiden Variablen mit den Namen `x` und `result` ein.

Variablen

Variablen dienen zur Speicherung von Daten.

Um Variablen in `MiniJava` zu nutzen müssen sie zunächst eingeführt, d.h. `deklariert` werden.

Variablen

Beispiel:

```
int x, result;
```

Diese Deklaration führt die beiden Variablen mit den Namen `x` und `result` ein.

- ▶ Das Schlüsselwort `int` besagt, dass diese Variablen ganze Zahlen („Integers“) speichern sollen.

`int` heißt auch **Typ** der Variablen `x` und `result`.

- ▶ Variablen können dann benutzt werden, um anzugeben, auf welche Daten Operationen angewendet werden sollen.
- ▶ Die Variablen in der Aufzählung sind durch Kommas „`,`“ getrennt.
- ▶ Am Ende steht ein Semikolon „`;`“.

Variablen

Variablen dienen zur Speicherung von Daten.

Um Variablen in **MiniJava** zu nutzen müssen sie zunächst eingeführt, d.h. **deklariert** werden.

Variablen

Beispiel:

```
int x, result;
```

Diese Deklaration führt die beiden Variablen mit den Namen `x` und `result` ein.

- ▶ Das Schlüsselwort `int` besagt, dass diese Variablen ganze Zahlen („Integers“) speichern sollen.
`int` heißt auch **Typ** der Variablen `x` und `result`.
- ▶ Variablen können dann benutzt werden, um anzugeben, auf welche Daten Operationen angewendet werden sollen.
- ▶ Die Variablen in der Aufzählung sind durch Kommas „`,`“ getrennt.
- ▶ Am Ende steht ein Semikolon „`;`“.

Variablen

Variablen dienen zur Speicherung von Daten.

Um Variablen in **MiniJava** zu nutzen müssen sie zunächst eingeführt, d.h. **deklariert** werden.

Variablen

Beispiel:

```
int x, result;
```

Diese Deklaration führt die beiden Variablen mit den Namen `x` und `result` ein.

- ▶ Das Schlüsselwort `int` besagt, dass diese Variablen ganze Zahlen („Integers“) speichern sollen.
`int` heißt auch **Typ** der Variablen `x` und `result`.
- ▶ Variablen können dann benutzt werden, um anzugeben, auf welche Daten Operationen angewendet werden sollen.
- ▶ Die Variablen in der Aufzählung sind durch Kommas „`,`“ getrennt.
- ▶ Am Ende steht ein Semikolon „`;`“.

Variablen

Variablen dienen zur Speicherung von Daten.

Um Variablen in **MiniJava** zu nutzen müssen sie zunächst eingeführt, d.h. **deklariert** werden.

Variablen

Beispiel:

```
int x, result;
```

Diese Deklaration führt die beiden Variablen mit den Namen `x` und `result` ein.

- ▶ Das Schlüsselwort `int` besagt, dass diese Variablen ganze Zahlen („Integers“) speichern sollen.
`int` heißt auch **Typ** der Variablen `x` und `result`.
- ▶ Variablen können dann benutzt werden, um anzugeben, auf welche Daten Operationen angewendet werden sollen.
- ▶ Die Variablen in der Aufzählung sind durch Kommas „`,`“ getrennt.
- ▶ Am Ende steht ein Semikolon „`;`“.

Variablen

Variablen dienen zur Speicherung von Daten.

Um Variablen in **MiniJava** zu nutzen müssen sie zunächst eingeführt, d.h. **deklariert** werden.

Operationen – Zuweisung

Operationen gestatten es, Werte von Variablen zu ändern. Die wichtigste Operation ist die **Zuweisung**.

Beispiele:

▶ `x = 7;`

Die Variable `x` erhält den Wert `7`.

▶ `result = x;`

Der Wert der Variablen `x` wird ermittelt und der Variablen `result` zugewiesen.

▶ `result = x + 19;`

Der Wert der Variablen `x` wird ermittelt, `19` dazu gezählt und dann das Ergebnis der Variablen `result` zugewiesen.

Variablen

Beispiel:

```
int x, result;
```

Diese Deklaration führt die beiden Variablen mit den Namen `x` und `result` ein.

- ▶ Das Schlüsselwort `int` besagt, dass diese Variablen ganze Zahlen („Integers“) speichern sollen.

`int` heißt auch **Typ** der Variablen `x` und `result`.

- ▶ Variablen können dann benutzt werden, um anzugeben, auf welche Daten Operationen angewendet werden sollen.
- ▶ Die Variablen in der Aufzählung sind durch Kommas „`,`“ getrennt.
- ▶ Am Ende steht ein Semikolon „`;`“.

Operationen – Zuweisung

Operationen gestatten es, Werte von Variablen zu ändern. Die wichtigste Operation ist die **Zuweisung**.

Beispiele:

▶ `x = 7;`

Die Variable `x` erhält den Wert `7`.

▶ `result = x;`

Der Wert der Variablen `x` wird ermittelt und der Variablen `result` zugewiesen.

▶ `result = x + 19;`

Der Wert der Variablen `x` wird ermittelt, `19` dazu gezählt und dann das Ergebnis der Variablen `result` zugewiesen.

Variablen

Beispiel:

```
int x, result;
```

Diese Deklaration führt die beiden Variablen mit den Namen `x` und `result` ein.

▶ Das Schlüsselwort `int` besagt, dass diese Variablen ganze Zahlen („Integers“) speichern sollen.

`int` heißt auch **Typ** der Variablen `x` und `result`.

▶ Variablen können dann benutzt werden, um anzugeben, auf welche Daten Operationen angewendet werden sollen.

▶ Die Variablen in der Aufzählung sind durch Kommas „`,`“ getrennt.

▶ Am Ende steht ein Semikolon „`;`“.

Operationen – Zuweisung

Operationen gestatten es, Werte von Variablen zu ändern. Die wichtigste Operation ist die **Zuweisung**.

Beispiele:

- ▶ `x = 7;`
Die Variable `x` erhält den Wert `7`.
- ▶ `result = x;`
Der Wert der Variablen `x` wird ermittelt und der Variablen `result` zugewiesen.
- ▶ `result = x + 19;`
Der Wert der Variablen `x` wird ermittelt, `19` dazu gezählt und dann das Ergebnis der Variablen `result` zugewiesen.

Variablen

Beispiel:

```
int x, result;
```

Diese Deklaration führt die beiden Variablen mit den **Namen** `x` und `result` ein.

- ▶ Das Schlüsselwort `int` besagt, dass diese Variablen ganze Zahlen („Integers“) speichern sollen.
`int` heißt auch **Typ** der Variablen `x` und `result`.
- ▶ Variablen können dann benutzt werden, um anzugeben, auf welche Daten Operationen angewendet werden sollen.
- ▶ Die Variablen in der Aufzählung sind durch Kommas „`,`“ getrennt.
- ▶ Am Ende steht ein Semikolon „`;`“.

Achtung:

- ▶ **Java** bezeichnet die Zuweisung mit „`=`“ anstatt „`:=`“ (Erbschaft von **C**...)
- ▶ Eine Zuweisung wird mit „`;`“ beendet.
- ▶ In der Zuweisung `x = x + 1;` greift das `x` auf der rechten Seite auf den Wert **vor** der Zuweisung zu.

Operationen gestatten es, Werte von Variablen zu ändern. Die wichtigste Operation ist die **Zuweisung**.

Beispiele:

- ▶ `x = 7;`
Die Variable `x` erhält den Wert **7**.
- ▶ `result = x;`
Der Wert der Variablen `x` wird ermittelt und der Variablen `result` zugewiesen.
- ▶ `result = x + 19;`
Der Wert der Variablen `x` wird ermittelt, **19** dazu gezählt und dann das Ergebnis der Variablen `result` zugewiesen.

Operationen – Input/Output

Minijava enthält Operationen um Daten (Zahlen) einlesen bzw. ausgeben zu können.

Beispiele:

- ▶ `x = read();`
Liest eine Folge von Zeichen ein und interpretiert sie als ganze Zahl, deren Wert sie der Variablen `x` als Wert zuweist.
- ▶ `write(42);`
Schreibt 42 auf die Ausgabe.
- ▶ `write(result);`
Bestimmt den Wert der Variablen `result` und schreibt dann diesen auf die Ausgabe.
- ▶ `write(x-14);`
Bestimmt den Wert der Variablen `x`, subtrahiert 14 und schreibt das Ergebnis auf die Ausgabe.

Operationen – Zuweisung

Achtung:

- ▶ **Java** bezeichnet die Zuweisung mit „`=`“ anstatt „`:=`“ (Erbschaft von **C**...)
- ▶ Eine Zuweisung wird mit „`;`“ beendet.
- ▶ In der Zuweisung `x = x + 1;` greift das `x` auf der rechten Seite auf den Wert **vor** der Zuweisung zu.

Operationen – Input/Output

Minijava enthält Operationen um Daten (Zahlen) einlesen bzw. ausgeben zu können.

Beispiele:

- ▶ `x = read();`
Liest eine Folge von Zeichen ein und interpretiert sie als ganze Zahl, deren Wert sie der Variablen `x` als Wert zuweist.
- ▶ `write(42);`
Schreibt `42` auf die Ausgabe.
- ▶ `write(result);`
Bestimmt den Wert der Variablen `result` und schreibt dann diesen auf die Ausgabe.
- ▶ `write(x-14);`
Bestimmt den Wert der Variablen `x`, subtrahiert `14` und schreibt das Ergebnis auf die Ausgabe.

Operationen – Zuweisung

Achtung:

- ▶ **Java** bezeichnet die Zuweisung mit „`=`“ anstatt „`:=`“ (Erbschaft von **C**...)
- ▶ Eine Zuweisung wird mit „`;`“ beendet.
- ▶ In der Zuweisung `x = x + 1;` greift das `x` auf der rechten Seite auf den Wert **vor** der Zuweisung zu.

Operationen – Input/Output

Minijava enthält Operationen um Daten (Zahlen) einlesen bzw. ausgeben zu können.

Beispiele:

- ▶ `x = read();`
Liest eine Folge von Zeichen ein und interpretiert sie als ganze Zahl, deren Wert sie der Variablen `x` als Wert zuweist.
- ▶ `write(42);`
Schreibt `42` auf die Ausgabe.
- ▶ `write(result);`
Bestimmt den Wert der Variablen `result` und schreibt dann diesen auf die Ausgabe.
- ▶ `write(x-14);`
Bestimmt den Wert der Variablen `x`, subtrahiert `14` und schreibt das Ergebnis auf die Ausgabe.

Operationen – Zuweisung

Achtung:

- ▶ Java bezeichnet die Zuweisung mit „`=`“ anstatt „`:=`“ (Erbschaft von C...)
- ▶ Eine Zuweisung wird mit „`;`“ beendet.
- ▶ In der Zuweisung `x = x + 1;` greift das `x` auf der rechten Seite auf den Wert **vor** der Zuweisung zu.

Operationen – Input/Output

Minijava enthält Operationen um Daten (Zahlen) einlesen bzw. ausgeben zu können.

Beispiele:

- ▶ `x = read();`
Liest eine Folge von Zeichen ein und interpretiert sie als ganze Zahl, deren Wert sie der Variablen `x` als Wert zuweist.
- ▶ `write(42);`
Schreibt `42` auf die Ausgabe.
- ▶ `write(result);`
Bestimmt den Wert der Variablen `result` und schreibt dann diesen auf die Ausgabe.
- ▶ `write(x-14);`
Bestimmt den Wert der Variablen `x`, subtrahiert `14` und schreibt das Ergebnis auf die Ausgabe.

Operationen – Zuweisung

Achtung:

- ▶ **Java** bezeichnet die Zuweisung mit „`=`“ anstatt „`:=`“ (Erbschaft von **C**...)
- ▶ Eine Zuweisung wird mit „`;`“ beendet.
- ▶ In der Zuweisung `x = x + 1;` greift das `x` auf der rechten Seite auf den Wert **vor** der Zuweisung zu.

Operationen – Input/Output

Achtung:

- ▶ Das argument der `write`-Operation in den Beispielen ist ein `int`.
- ▶ Um es ausgeben zu können muss es erst in ein `Zeichenfolge` umgewandelt werden, d.h. einen `String`

In `Minijava` könne auch direkt Strings ausgegeben werden:

Beispiel:

- ▶ `write("Hello World!!!");`
Schreibt `Hello World!!!` auf die Ausgabe.

Operationen – Input/Output

`Minijava` enthält Operationen um Daten (Zahlen) einlesen bzw. ausgeben zu können.

Beispiele:

- ▶ `x = read();`
Liest eine Folge von Zeichen ein und interpretiert sie als ganze Zahl, deren Wert sie der Variablen `x` als Wert zuweist.
- ▶ `write(42);`
Schreibt `42` auf die Ausgabe.
- ▶ `write(result);`
Bestimmt den Wert der Variablen `result` und schreibt dann diesen auf die Ausgabe.
- ▶ `write(x-14);`
Bestimmt den Wert der Variablen `x`, subtrahiert `14` und schreibt das Ergebnis auf die Ausgabe.

Operationen – Input/Output

Achtung:

- ▶ Das argument der `write`-Operation in den Beispielen ist ein `int`.
- ▶ Um es ausgeben zu können muss es erst in ein `Zeichenfolge` umgewandelt werden, d.h. einen `String`

In `Minijava` könne auch direkt Strings ausgegeben werden:

Beispiel:

- ▶ `write("Hello World!!!");`
Schreibt `Hello World!!!` auf die Ausgabe.

Operationen – Input/Output

`Minijava` enthält Operationen um Daten (Zahlen) einlesen bzw. ausgeben zu können.

Beispiele:

- ▶ `x = read();`
Liest eine Folge von Zeichen ein und interpretiert sie als ganze Zahl, deren Wert sie der Variablen `x` als Wert zuweist.
- ▶ `write(42);`
Schreibt `42` auf die Ausgabe.
- ▶ `write(result);`
Bestimmt den Wert der Variablen `result` und schreibt dann diesen auf die Ausgabe.
- ▶ `write(x-14);`
Bestimmt den Wert der Variablen `x`, subtrahiert `14` und schreibt das Ergebnis auf die Ausgabe.

Kontrollstrukturen – Sequenz

Sequenz:

```
1 int x, y, result;  
2 x = read();  
3 y = read();  
4 result = x + y;  
5 write(result);
```

- ▶ Zu jedem Zeitpunkt wird nur eine Operation ausgeführt.
- ▶ Jede Operation wird genau einmal ausgeführt.
- ▶ Die Reihenfolge, in der die Operationen ausgeführt werden, ist die gleiche, in der sie im Programm stehen.
- ▶ Mit Beendigung der letzten Operation endet die Programm-Ausführung.

Sequenz alleine erlaubt nur sehr einfache Programme.

Operationen – Input/Output

Achtung:

- ▶ Das argument der **write**-Operation in den Beispielen ist ein **int**.
- ▶ Um es ausgeben zu können muss es erst in ein **Zeichenfolge** umgewandelt werden, d.h. einen **String**

In **Minijava** könne auch direkt Strings ausgegeben werden:

Beispiel:

- ▶ **write("Hello World!!!");**
Schreibt **Hello World!!!** auf die Ausgabe.

Kontrollstrukturen – Selektion

Selektion (bedingte Auswahl):

```
1 int x, y, result;  
2 x = read();  
3 y = read();  
4 if (x > y)  
5     result = x - y;  
6 else  
7     result = y - x;  
8 write(result);
```

- ▶ Zuerst wird die Bedingung ausgewertet
- ▶ Ist sie erfüllt, wird die nächste Operation ausgeführt.
- ▶ Ist sie nicht erfüllt, wird die nächste Operation nach dem `else`-Zweig ausgeführt.

Kontrollstrukturen – Sequenz

Sequenz:

```
1 int x, y, result;  
2 x = read();  
3 y = read();  
4 result = x + y;  
5 write(result);
```

- ▶ Zu jedem Zeitpunkt wird nur eine Operation ausgeführt.
- ▶ Jede Operation wird genau einmal ausgeführt.
- ▶ Die Reihenfolge, in der die Operationen ausgeführt werden, ist die gleiche, in der sie im Programm stehen.
- ▶ Mit Beendigung der letzten Operation endet die Programm-Ausführung.

Sequenz alleine erlaubt nur sehr einfache Programme.

Kontrollstrukturen – Selektion

Beispiel:

- ▶ Statt einer einzelnen Operation können die Alternativen auch aus **Statements** bestehen:

```
1 int x;  
2 x = read();  
3 if (x == 0)  
4     write(0);  
5 else if (x < 0)  
6     write(-1);  
7 else  
8     write(+1);
```

Kontrollstrukturen – Selektion

Selektion (bedingte Auswahl):

```
1 int x, y, result;  
2 x = read();  
3 y = read();  
4 if (x > y)  
5     result = x - y;  
6 else  
7     result = y - x;  
8 write(result);
```

- ▶ Zuerst wird die Bedingung ausgewertet
- ▶ Ist sie erfüllt, wird die nächste Operation ausgeführt.
- ▶ Ist sie nicht erfüllt, wird die nächste Operation nach dem **else**-Zweig ausgeführt.

Kontrollstrukturen – Selektion

Beispiel:

- ▶ ... oder aus (geklammerten) Folgen von Operationen und Statements:

```
1 int x, y;  
2 x = read();  
3 if (x != 0) {  
4     y = read();  
5     if (x > y)  
6         write(x);  
7     else  
8         write(y);  
9 } else  
10 write(0);
```

Kontrollstrukturen – Selektion

Beispiel:

- ▶ Statt einer einzelnen Operation können die Alternativen auch aus **Statements** bestehen:

```
1 int x;  
2 x = read();  
3 if (x == 0)  
4     write(0);  
5 else if (x < 0)  
6     write(-1);  
7 else  
8     write(+1);
```

Kontrollstrukturen – Selektion

Beispiel:

- ▶ ...eventuell fehlt auch der `else`-Teil:

```
1 int x, y;  
2 x = read();  
3 if (x != 0) {  
4     y = read();  
5     if (x > y)  
6         write(x);  
7     else  
8         write(y);  
9 }
```

Auch mit Sequenz und Selektion kann noch nicht viel berechnet werden...

Kontrollstrukturen – Selektion

Beispiel:

- ▶ ...oder aus (geklammerten) Folgen von Operationen und Statements:

```
1 int x, y;  
2 x = read();  
3 if (x != 0) {  
4     y = read();  
5     if (x > y)  
6         write(x);  
7     else  
8         write(y);  
9 } else  
10 write(0);
```

Kontrollstrukturen – Iteration

Iteration (wiederholte Ausführung)

```
1 int x, y;
2 x = read(); y = read();
3 while (x != y) {
4     if (x < y)
5         y = y - x;
6     else
7         x = x - y;
8 }
9 write(x);
```

- ▶ Zuerst wird die Bedingung ausgewertet.
- ▶ Ist sie erfüllt, wird der **Rumpf** des **while**-statements ausgeführt.
- ▶ Nach Ausführung des Rumpfs wird das gesamte **while**-statement erneut ausgeführt.
- ▶ Ist die Bedingung nicht erfüllt fährt die Programmausführung hinter dem **while**-statement fort.

Kontrollstrukturen – Selektion

Beispiel:

- ▶ ...eventuell fehlt auch der **else**-Teil:

```
1 int x, y;
2 x = read();
3 if (x != 0) {
4     y = read();
5     if (x > y)
6         write(x);
7     else
8         write(y);
9 }
```

Auch mit Sequenz und Selektion kann noch nicht viel berechnet werden...

2 Eine einfache Programmiersprache

Theorem (↑Berechenbarkeitstheorie)

Jede (partielle) Funktion auf ganzen Zahlen, die überhaupt berechenbar ist, läßt sich mit Selektion, Sequenz, und Iteration, d.h., mithilfe eines Minijava-Programms berechnen.

Beweisidee

- ▶ Was heißt berechenbar?
Eine Funktion heißt berechenbar wenn man sie mithilfe einer Turingmaschine berechnen kann.
- ▶ Schreibe ein Minijava-Programm, das eine Turingmaschine simuliert.

Kontrollstrukturen – Iteration

Iteration (wiederholte Ausführung)

```
1 int x, y;  
2 x = read(); y = read();  
3 while (x != y) {  
4     if (x < y)  
5         y = y - x;  
6     else  
7         x = x - y;  
8 }  
9 write(x);
```

- ▶ Zuerst wird die Bedingung ausgewertet.
- ▶ Ist sie erfüllt, wird der Rumpf des `while`-statements ausgeführt.
- ▶ Nach Ausführung des Rumpfs wird das gesamte `while`-statement erneut ausgeführt.
- ▶ Ist die Bedingung nicht erfüllt fährt die Programmausführung hinter dem `while`-statement fort.

2 Eine einfache Programmiersprache

Theorem (↑Berechenbarkeitstheorie)

Jede (partielle) Funktion auf ganzen Zahlen, die überhaupt berechenbar ist, läßt sich mit Selektion, Sequenz, und Iteration, d.h., mithilfe eines MiniJava-Programms berechnen.

Beweisidee

- ▶ Was heißt berechenbar?
Eine Funktion heißt berechenbar wenn man sie mithilfe einer Turingmaschine berechnen kann.
- ▶ Schreibe ein **MiniJava**-Programm, das eine Turingmaschine simuliert.

Kontrollstrukturen – Iteration

Iteration (wiederholte Ausführung)

```
1 int x, y;  
2 x = read(); y = read();  
3 while (x != y) {  
4     if (x < y)  
5         y = y - x;  
6     else  
7         x = x - y;  
8 }  
9 write(x);
```

- ▶ Zuerst wird die Bedingung ausgewertet.
- ▶ Ist sie erfüllt, wird der **Rumpf** des **while**-statements ausgeführt.
- ▶ Nach Ausführung des Rumpfs wird das gesamte **while**-statement erneut ausgeführt.
- ▶ Ist die Bedingung nicht erfüllt fährt die Programmausführung hinter dem **while**-statement fort.

2 Eine einfache Programmiersprache

MiniJava-Programme sind ausführbares **Java**. Man muss sie nur geeignet **dekoriern**.

Beispiel: das GGT-Programm.

```
1 int x, y;
2 x = read();
3 y = read();
4 while (x != y) {
5     if (x < y)
6         y = y - x;
7     else
8         x = x - y;
9 }
10 write(x);
```

2 Eine einfache Programmiersprache

Theorem (\uparrow **Berechenbarkeitstheorie**)

Jede (partielle) Funktion auf ganzen Zahlen, die überhaupt berechenbar ist, lässt sich mit Selektion, Sequenz, und Iteration, d.h., mithilfe eines MiniJava-Programms berechnen.

Beweisidee

- ▶ Was heißt berechenbar?
Eine Funktion heißt berechenbar wenn man sie mithilfe einer Turingmaschine berechnen kann.
- ▶ Schreibe ein **MiniJava**-Programm, das eine Turingmaschine simuliert.

Daraus wird folgendes Java-Programm:

```
1 public class GGT extends MiniJava {
2     public static void main (String[] args) {
3         int x, y;
4         x = read();
5         y = read();
6         while (x != y) {
7             if (x < y)
8                 y = y - x;
9             else
10                x = x - y;
11        }
12        write(x);
13    } // Ende der Definition von main();
14 } // Ende der Definition der Klasse GGT;
```

"GGT.java"

Daraus wird folgendes Java-Programm:

```
1 public class GGT extends MiniJava {
2     public static void main (String[] args) {
3         int x, y;
4         x = read();
5         y = read();
6         while (x != y) {
7             if (x < y)
8                 y = y - x;
9             else
10                x = x - y;
11        }
12        write(x);
13    } // Ende der Definition von main();
14 } // Ende der Definition der Klasse GGT;
```

"GGT.java"

Daraus wird folgendes Java-Programm:

```
1 public class GGT extends MiniJava {
2     public static void main (String[] args) {
3         int x, y;
4         x = read();
5         y = read();
6         while (x != y) {
7             if (x < y)
8                 y = y - x;
9             else
10                x = x - y;
11        }
12        write(x);
13    } // Ende der Definition von main();
14 } // Ende der Definition der Klasse GGT;
```

"GGT.java"

Daraus wird folgendes Java-Programm:

```
1 public class GGT extends MiniJava {
2     public static void main (String[] args) {
3         int x, y;
4         x = read();
5         y = read();
6         while (x != y) {
7             if (x < y)
8                 y = y - x;
9             else
10                x = x - y;
11        }
12        write(x);
13    } // Ende der Definition von main();
14 } // Ende der Definition der Klasse GGT;
```

"GGT.java"

Daraus wird folgendes Java-Programm:

```
1 public class GGT extends MiniJava {  
2     public static void main (String[] args) {  
3         int x, y;  
4         x = read();  
5         y = read();  
6         while (x != y) {  
7             if (x < y)  
8                 y = y - x;  
9             else  
10                x = x - y;  
11        }  
12        write(x);  
13    } // Ende der Definition von main();  
14 } // Ende der Definition der Klasse GGT;
```

"GGT.java"

Daraus wird folgendes Java-Programm:

```
1 public class GGT extends MiniJava {
2     public static void main (String[] args) {
3         int x, y;
4         x = read();
5         y = read();
6         while (x != y) {
7             if (x < y)
8                 y = y - x;
9             else
10                x = x - y;
11        }
12        write(x);
13    } // Ende der Definition von main();
14 } // Ende der Definition der Klasse GGT;
```

"GGT.java"

Daraus wird folgendes Java-Programm:

```
1 public class GGT extends MiniJava {
2     public static void main (String[] args) {
3         int x, y;
4         x = read();
5         y = read();
6         while (x != y) {
7             if (x < y)
8                 y = y - x;
9             else
10                x = x - y;
11        }
12        write(x);
13    } // Ende der Definition von main();
14 } // Ende der Definition der Klasse GGT;
```

"GGT.java"

Daraus wird folgendes Java-Programm:

```
1 public class GGT extends MiniJava {
2     public static void main (String[] args) {
3         int x, y;
4         x = read();
5         y = read();
6         while (x != y) {
7             if (x < y)
8                 y = y - x;
9             else
10                x = x - y;
11        }
12        write(x);
13    } // Ende der Definition von main();
14 } // Ende der Definition der Klasse GGT;
```

"GGT.java"

Daraus wird folgendes **Java**-Programm:

```
1 public class GGT extends MiniJava {
2     public static void main (String[] args) {
3         int x, y;
4         x = read();
5         y = read();
6         while (x != y) {
7             if (x < y)
8                 y = y - x;
9             else
10                x = x - y;
11        }
12        write(x);
13    } // Ende der Definition von main();
14 } // Ende der Definition der Klasse GGT;
```

"GGT.java"

```
1 import javax.swing.JOptionPane;
2 import javax.swing.JFrame;
3 public class MiniJava {
4     public static int read () {
5         JFrame f = new JFrame ();
6         String s = JOptionPane.showInputDialog (f, "Eingabe:");
7         int x = 0; f.dispose ();
8         if (s == null) System.exit (0);
9         try { x = Integer.parseInt (s.trim ());
10        } catch (NumberFormatException e) { x = read (); }
11        return x;
12    }
13    public static void write (String x) {
14        JFrame f = new JFrame ();
15        JOptionPane.showMessageDialog (f, x, "Ausgabe",
16        JOptionPane.PLAIN_MESSAGE);
17        f.dispose ();
18    }
19    public static void write (int x) { write (""+x); }
20 }
```

Datei: "MiniJava.java"

Weitere Erläuterungen:

- ▶ Jedes Programm sollte Kommentare enthalten, damit man sich selbst später noch darin zurecht findet!
- ▶ Ein Kommentar in **Java** hat etwa die Form:
`// Das ist ein Kommentar!!!`
- ▶ Wenn er sich über mehrere Zeilen erstrecken soll dann
`/* Dieser Kommentar ist verdammt
laaaaaaaaaaang
*/`

```
1 import javax.swing.JOptionPane;
2 import javax.swing.JFrame;
3 public class MiniJava {
4     public static int read () {
5         JFrame f = new JFrame ();
6         String s = JOptionPane.showInputDialog (f, "Eingabe:");
7         int x = 0; f.dispose ();
8         if (s == null) System.exit (0);
9         try { x = Integer.parseInt (s.trim ());
10        } catch (NumberFormatException e) { x = read (); }
11        return x;
12    }
13    public static void write (String x) {
14        JFrame f = new JFrame ();
15        JOptionPane.showMessageDialog (f, x, "Ausgabe",
16        JOptionPane.PLAIN_MESSAGE);
17        f.dispose ();
18    }
19    public static void write (int x) { write (""+x); }
20 }
```

Datei: "MiniJava.java"

2 Eine einfache Programmiersprache

Das Programm GGT kann nun übersetzt und dann ausgeführt werden:

```
raecke> javac GGT.java
raecke> java GGT
```

- ▶ Der Compiler `javac` liest das Programm aus den Dateien `GGT.java` und `MiniJava.java` ein und erzeugt für sie JVM-Code, den er in den Dateien `GGT.class` und `MiniJava.class` ablegt.
- ▶ Das Laufzeitsystem `java` liest die Dateien `GGT.class` und `MiniJava.class` ein und führt sie aus.

Weitere Erläuterungen:

- ▶ Jedes Programm sollte Kommentare enthalten, damit man sich selbst später noch darin zurecht findet!
- ▶ Ein Kommentar in `Java` hat etwa die Form:
- ▶ Wenn er sich über mehrere Zeilen erstrecken soll dann

```
// Das ist ein Kommentar!!!
```

```
/* Dieser Kommentar ist verdammt
   laaaaaaaaaaang
   */
```

Wichtige Erweiterung – Arrays

Arrays enthalten eine Gruppe von Variablen auf die über einen **index** zugegriffen wird:

- ▶ Deklariere Variablen `a[0],...,a[99]` und `b[0],...,b[4]`:

```
int[] a = new int[100], b = new int[5];
```

- ▶ Greife auf Element `a[5]` zu:

```
int i;  
int[] a = new int[100];  
a[5] = 1; // a[5] ist jetzt 0  
i = 5;  
a[i] = 7; // a[5] ist jetzt 7  
i = 7;  
a[i-2] = 8; // a[5] ist jetzt 8
```

2 Eine einfache Programmiersprache

Das Programm GGT kann nun übersetzt und dann ausgeführt werden:

```
raecke> javac GGT.java  
raecke> java GGT
```

- ▶ Der Compiler `javac` liest das Programm aus den Dateien `GGT.java` und `MiniJava.java` ein und erzeugt für sie JVM-Code, den er in den Dateien `GGT.class` und `MiniJava.class` ablegt.
- ▶ Das Laufzeitsystem `java` liest die Dateien `GGT.class` und `MiniJava.class` ein und führt sie aus.

Minijava ist sehr primitiv

Die Programmiersprache Java bietet noch eine Fülle von Hilfsmitteln an, die das Programmieren erleichtern sollen.

Insbesondere gibt es

- ▶ viele weitere Datentypen (nicht nur `int`) und
- ▶ viele weitere Kontrollstrukturen

... kommt später in der Vorlesung!

Wichtige Erweiterung – Arrays

Arrays enthalten eine Gruppe von Variablen auf die über einen **index** zugegriffen wird:

- ▶ Deklariere Variablen `a[0], ..., a[99]` und `b[0], ..., b[4]`:

```
int[] a = new int[100], b = new int[5];
```

- ▶ Greife auf Element `a[5]` zu:

```
int i;  
int[] a = new int[100];  
a[5] = 1; // a[5] ist jetzt 0  
i = 5;  
a[i] = 7; // a[5] ist jetzt 7  
i = 7;  
a[i-2] = 8; // a[5] ist jetzt 8
```

Syntax („Lehre vom Satzbau“)

- ▶ formale Beschreibung des Aufbaus der „Worte“ und „Sätze“, die zu einer Sprache gehören;
- ▶ im Falle einer **Programmiersprache** Festlegung, wie Programme aussehen müssen.

Hilfsmittel bei natürlicher Sprache

- ▶ Wörterbücher;
- ▶ Rechtschreibregeln, Trennungsregeln, Grammatikregeln;
- ▶ Ausnahmelisten;
- ▶ Sprachgefühl.

Syntax („Lehre vom Satzbau“)

- ▶ formale Beschreibung des Aufbaus der „Worte“ und „Sätze“, die zu einer Sprache gehören;
- ▶ im Falle einer **Programmiersprache** Festlegung, wie Programme aussehen müssen.

Hilfsmittel

Hilfsmittel bei Programmiersprachen

- ▶ Listen von **Schlüsselworten** wie `if`, `int`, `else`, `while`...
- ▶ Regeln, wie einzelne Worte (**Tokens**) z.B. **Namen** gebildet werden.
Frage: Ist `x10` ein zulässiger Name für eine Variable (oder `_ab` oder `A#B` oder `0A?B`)?...
- ▶ Grammatikregeln, die angeben, wie größere Komponenten aus kleineren aufgebaut werden.
Frage: Ist ein `while`-Statement im `else`-Teil erlaubt?
- ▶ Kontextbedingungen.
Beispiel: Eine Variable muss erst deklariert sein, bevor sie verwendet wird.

Hilfsmittel

Hilfsmittel bei natürlicher Sprache

- ▶ Wörterbücher;
- ▶ Rechtschreibregeln, Trennungsregeln, Grammatikregeln;
- ▶ Ausnahmelisten;
- ▶ Sprachgefühl.

Hilfsmittel

Hilfsmittel bei Programmiersprachen

- ▶ Listen von **Schlüsselworten** wie `if`, `int`, `else`, `while`...
- ▶ Regeln, wie einzelne Worte (**Tokens**) z.B. **Namen** gebildet werden.
Frage: Ist `x10` ein zulässiger Name für eine Variable (oder `_ab` oder `A#B` oder `0A?B`)?...
- ▶ Grammatikregeln, die angeben, wie größere Komponenten aus kleineren aufgebaut werden.
Frage: Ist ein `while`-Statement im `else`-Teil erlaubt?
- ▶ Kontextbedingungen.
Beispiel: Eine Variable muss erst deklariert sein, bevor sie verwendet wird.

Hilfsmittel

Hilfsmittel bei natürlicher Sprache

- ▶ Wörterbücher;
- ▶ Rechtschreibregeln, Trennungsregeln, Grammatikregeln;
- ▶ Ausnahmelisten;
- ▶ Sprachgefühl.

Hilfsmittel

Hilfsmittel bei Programmiersprachen

- ▶ Listen von **Schlüsselworten** wie `if`, `int`, `else`, `while`...
- ▶ Regeln, wie einzelne Worte (**Tokens**) z.B. **Namen** gebildet werden.
Frage: Ist `x10` ein zulässiger Name für eine Variable (oder `_ab` oder `A#B` oder `0A?B`)?...
- ▶ Grammatikregeln, die angeben, wie größere Komponenten aus kleineren aufgebaut werden.
Frage: Ist ein `while`-Statement im `else`-Teil erlaubt?
- ▶ Kontextbedingungen.
Beispiel: Eine Variable muss erst deklariert sein, bevor sie verwendet wird.

Hilfsmittel

Hilfsmittel bei natürlicher Sprache

- ▶ Wörterbücher;
- ▶ Rechtschreibregeln, Trennungsregeln, Grammatikregeln;
- ▶ Ausnahmelisten;
- ▶ Sprachgefühl.

Hilfsmittel

Hilfsmittel bei Programmiersprachen

- ▶ Listen von **Schlüsselworten** wie `if`, `int`, `else`, `while`...
- ▶ Regeln, wie einzelne Worte (**Tokens**) z.B. **Namen** gebildet werden.
Frage: Ist `x10` ein zulässiger Name für eine Variable (oder `_ab` oder `A#B` oder `0A?B`)?...
- ▶ Grammatikregeln, die angeben, wie größere Komponenten aus kleineren aufgebaut werden.
Frage: Ist ein `while`-Statement im `else`-Teil erlaubt?
- ▶ Kontextbedingungen.
Beispiel: Eine Variable muss erst deklariert sein, bevor sie verwendet wird.

Hilfsmittel

Hilfsmittel bei natürlicher Sprache

- ▶ Wörterbücher;
- ▶ Rechtschreibregeln, Trennungsregeln, Grammatikregeln;
- ▶ Ausnahmelisten;
- ▶ Sprachgefühl.

Programmiersprachen sind

- ▶ formalisierter als natürliche Sprache
- ▶ besser für maschinelle Verarbeitung geeignet.

Hilfsmittel bei Programmiersprachen

- ▶ Listen von **Schlüsselworten** wie `if`, `int`, `else`, `while`...
- ▶ Regeln, wie einzelne Worte (**Tokens**) z.B. **Namen** gebildet werden.
Frage: Ist `x10` ein zulässiger Name für eine Variable (oder `_ab` oder `A#B` oder `0A?B`)?...
- ▶ Grammatikregeln, die angeben, wie größere Komponenten aus kleineren aufgebaut werden.
Frage: Ist ein `while`-Statement im `else`-Teil erlaubt?
- ▶ Kontextbedingungen.
Beispiel: Eine Variable muss erst deklariert sein, bevor sie verwendet wird.

Semantik („Lehre von der Bedeutung“)

- ▶ Ein Satz einer (natürlichen) Sprache verfügt zusätzlich über eine **Bedeutung**, d.h. teilt einem Hörer/Leser einen Sachverhalt mit (↑**Information**)
- ▶ Ein Satz einer Programmiersprache, d.h. ein Programm verfügt ebenfalls über eine **Bedeutung**...

Programmiersprachen sind

- ▶ formalisierter als natürliche Sprache
- ▶ besser für maschinelle Verarbeitung geeignet.

Die Bedeutung eines Programms ist

- ▶ alle möglichen **Ausführungen** der beschriebenen Berechnung (↑**operationelle Semantik**); oder
- ▶ die definierte **Abbildung** der Eingaben auf die Ausgaben (↑**denotationelle Semantik**).

Achtung!

Ist ein Programm **syntaktisch korrekt**, heißt das noch lange nicht, dass es auch das „richtige“ tut, d.h. **semantisch korrekt** ist!!!

Semantik („Lehre von der Bedeutung“)

- ▶ Ein Satz einer (natürlichen) Sprache verfügt zusätzlich über eine **Bedeutung**, d.h. teilt einem Hörer/Leser einen Sachverhalt mit (↑**Information**)
- ▶ Ein Satz einer Programmiersprache, d.h. ein Programm verfügt ebenfalls über eine **Bedeutung**...

Die Bedeutung eines Programms ist

- ▶ alle möglichen **Ausführungen** der beschriebenen Berechnung (↑**operationelle Semantik**); oder
- ▶ die definierte **Abbildung** der Eingaben auf die Ausgaben (↑**denotationelle Semantik**).

Achtung!

Ist ein Programm **syntaktisch korrekt**, heißt das noch lange nicht, dass es auch das „richtige“ tut, d.h. **semantisch korrekt** ist!!!

Semantik („Lehre von der Bedeutung“)

- ▶ Ein Satz einer (natürlichen) Sprache verfügt zusätzlich über eine **Bedeutung**, d.h. teilt einem Hörer/Leser einen Sachverhalt mit (↑**Information**)
- ▶ Ein Satz einer Programmiersprache, d.h. ein Programm verfügt ebenfalls über eine **Bedeutung**...

Die Bedeutung eines Programms ist

- ▶ alle möglichen **Ausführungen** der beschriebenen Berechnung (↑**operationelle Semantik**); oder
- ▶ die definierte **Abbildung** der Eingaben auf die Ausgaben (↑**denotationelle Semantik**).

Achtung!

Ist ein Programm **syntaktisch korrekt**, heißt das noch lange nicht, dass es auch das „richtige“ tut, d.h. **semantisch korrekt** ist!!!

Semantik („Lehre von der Bedeutung“)

- ▶ Ein Satz einer (natürlichen) Sprache verfügt zusätzlich über eine **Bedeutung**, d.h. teilt einem Hörer/Leser einen Sachverhalt mit (↑**Information**)
- ▶ Ein Satz einer Programmiersprache, d.h. ein Programm verfügt ebenfalls über eine **Bedeutung**...

3.1 Reservierte Wörter

- ▶ `int`
⇒ Bezeichner für Basistypen;
- ▶ `if, else, then, while...`
⇒ Schlüsselwörter für Programmkonstrukte;
- ▶ `(,), ", ', {, }, ,, ;`
⇒ Sonderzeichen;

3.2 Was ist ein erlaubter Name?

Schritt 1:

Festlegung erlaubter Zeichen:

letter ::= \$ | _ | a | ... | z | A | ... | Z

digit ::= 0 | ... | 9

- ▶ letter und digit bezeichnen **Zeichenklassen**, d.h. Mengen von Zeichen, die gleich behandelt werden.
- ▶ Das Symbol „|“ trennt zulässige Alternativen.
- ▶ Das Symbol „...“ repräsentiert die Faulheit, alle Alternativen wirklich aufzuzählen.

3.2 Was ist ein erlaubter Name?

Schritt 1:

Festlegung erlaubter Zeichen:

`letter` ::= \$ | _ | a | ... | z | A | ... | Z

`digit` ::= 0 | ... | 9

- ▶ `letter` und `digit` bezeichnen **Zeichenklassen**, d.h. Mengen von Zeichen, die gleich behandelt werden.
- ▶ Das Symbol „|“ trennt zulässige Alternativen.
- ▶ Das Symbol „...“ repräsentiert die Faulheit, alle Alternativen wirklich aufzuzählen.

3.2 Was ist ein erlaubter Name?

Schritt 2:

Festlegung der Zeichenanordnung:

`name ::= letter (letter | digit)*`

- ▶ Erst kommt ein Zeichen der Klasse `letter`, dann eine (eventuell auch leere) Folge von Zeichen entweder aus `letter` oder aus `digit`.
- ▶ Der Operator „*“ bedeutet „beliebig oft wiederholen“ („weglassen“ ist 0-malige Wiederholung).
- ▶ Der Operator „*“ ist ein **Postfix**-Operator. Das heißt, er steht hinter seinem Argument.

3.2 Was ist ein erlaubter Name?

Schritt 1:

Festlegung erlaubter Zeichen:

`letter ::= $ | _ | a | ... | z | A | ... | Z`

`digit ::= 0 | ... | 9`

- ▶ `letter` und `digit` bezeichnen **Zeichenklassen**, d.h. Mengen von Zeichen, die gleich behandelt werden.
- ▶ Das Symbol „|“ trennt zulässige Alternativen.
- ▶ Das Symbol „...“ repräsentiert die Faulheit, alle Alternativen wirklich aufzuzählen.

Beispiele

_178
Das_ist_kein_Name
x
_
\$Password\$

...sind legale Namen.

5ABC
!Hallo!
x'
a=b
-178

...sind keine legalen Namen.

Achtung

Reservierte Wörter sind als Namen verboten.

3.2 Was ist ein erlaubter Name?

Schritt 2:

Festlegung der Zeichenanordnung:

$$\text{name} ::= \text{letter (letter | digit)}^*$$

- ▶ Erst kommt ein Zeichen der Klasse **letter**, dann eine (eventuell auch leere) Folge von Zeichen entweder aus **letter** oder aus **digit**.
- ▶ Der Operator „*“ bedeutet „beliebig oft wiederholen“ („weglassen“ ist 0-malige Wiederholung).
- ▶ Der Operator „*“ ist ein **Postfix**-Operator. Das heißt, er steht hinter seinem Argument.

Beispiele

_178
Das_ist_kein_Name
x
—
\$Password\$

...sind legale Namen.

5ABC
!Hallo!
x'
a=b
-178

...sind keine legalen Namen.

Achtung

Reservierte Wörter sind als Namen verboten.

3.2 Was ist ein erlaubter Name?

Schritt 2:

Festlegung der Zeichenanordnung:

`name ::= letter (letter | digit)*`

- ▶ Erst kommt ein Zeichen der Klasse **letter**, dann eine (eventuell auch leere) Folge von Zeichen entweder aus **letter** oder aus **digit**.
- ▶ Der Operator „*“ bedeutet „beliebig oft wiederholen“ („weglassen“ ist 0-malige Wiederholung).
- ▶ Der Operator „*“ ist ein **Postfix**-Operator. Das heißt, er steht hinter seinem Argument.

3.3 Ganze Zahlen

Werte, die direkt im Programm stehen, heißen **Konstanten**.
Ganze nichtnegative Zahlen bestehen aus einer nichtleeren Folge von Ziffern:

`number ::= digit digit*`

- ▶ Wie sähe die Regel aus, wenn wir führende Nullen verbieten wollen?

Beispiele

17

12490

42

0

00070

...sind `int`-Konstanten

"Hello World!"

0.5e+128

...sind keine `int`-Konstanten

3.3 Ganze Zahlen

Werte, die direkt im Programm stehen, heißen **Konstanten**.

Ganze nichtnegative Zahlen bestehen aus einer nichtleeren Folge von Ziffern:

`number ::= digit digit*`

- Wie sähe die Regel aus, wenn wir führende Nullen verbieten wollen?

Reguläre Ausdrücke

Ausdrücke, die aus Zeichen(-klassen) mithilfe von

- | (Alternative)
- * (Iteration)
- (Konkatenation) sowie
- ? (Option)

...aufgebaut sind, heißen **reguläre Ausdrücke** (↑**Automatentheorie**).

Der Postfix-Operator „?“ besagt, dass das Argument eventuell auch fehlen darf, d.h. einmal oder keinmal vorkommt.

Beispiele

17

12490

42

0

00070

...sind **int**-Konstanten

"Hello World!"

0.5e+128

...sind keine **int**-Konstanten

Beispiele

Reguläre Ausdrücke reichen zur Beschreibung **einfacher** Mengen von Worten aus.

▶ `(letter letter)*`

⇒ alle Wörter gerader Länge (über
`$_, a, ..., z, A, ..., Z`);

▶ `letter* test letter*`

⇒ alle Wörter, die das Teilwort `test` enthalten;

▶ `_ digit* 17`

⇒ alle Wörter, die mit `_` anfangen, dann eine beliebige Folge von Ziffern aufweisen, die mit `17` aufhört;

▶ `exp ::= (e|E)(+|-)? digit digit*`

`float ::= digit digit* exp |
digit* (digit . | . digit) digit* exp?`

⇒ alle Gleitkommazahlen...

Reguläre Ausdrücke

Ausdrücke, die aus Zeichen(-klassen) mithilfe von

| (Alternative)

* (Iteration)

(Konkatenation) sowie

? (Option)

...aufgebaut sind, heißen **reguläre Ausdrücke** (↑**Automatentheorie**).

Der Postfix-Operator „**?**“ besagt, dass das Argument eventuell auch fehlen darf, d.h. einmal oder keinmal vorkommt.

Beispiele

Reguläre Ausdrücke reichen zur Beschreibung **einfacher** Mengen von Worten aus.

- ▶ `(letter letter)*`
⇒ alle Wörter gerader Länge (über `$_, a, . . . , z, A, . . . , Z`);
- ▶ `letter* test letter*`
⇒ alle Wörter, die das Teilwort `test` enthalten;
- ▶ `_ digit* 17`
⇒ alle Wörter, die mit `_` anfangen, dann eine beliebige Folge von Ziffern aufweisen, die mit `17` aufhört;
- ▶ `exp ::= (e|E)(+|-)? digit digit*`
`float ::= digit digit* exp | digit* (digit . | . digit) digit* exp?`
⇒ alle Gleitkommazahlen...

Reguläre Ausdrücke

Ausdrücke, die aus Zeichen(-klassen) mithilfe von

- | (Alternative)
- * (Iteration)
- (Konkatenation) sowie
- ? (Option)

...aufgebaut sind, heißen **reguläre Ausdrücke** (↑**Automatentheorie**).

Der Postfix-Operator „**?**“ besagt, dass das Argument eventuell auch fehlen darf, d.h. einmal oder keinmal vorkommt.

Beispiele

Reguläre Ausdrücke reichen zur Beschreibung **einfacher** Mengen von Worten aus.

- ▶ `(letter letter)*`
⇒ alle Wörter gerader Länge (über `$_, a, ..., z, A, ..., Z`);
- ▶ `letter* test letter*`
⇒ alle Wörter, die das Teilwort `test` enthalten;
- ▶ `_ digit* 17`
⇒ alle Wörter, die mit `_` anfangen, dann eine beliebige Folge von Ziffern aufweisen, die mit `17` aufhört;
- ▶ `exp ::= (e|E)(+|-)? digit digit*`
`float ::= digit digit* exp | digit* (digit . | . digit) digit* exp?`
⇒ alle Gleitkommazahlen...

Reguläre Ausdrücke

Ausdrücke, die aus Zeichen(-klassen) mithilfe von

- | (Alternative)
- * (Iteration)
- (Konkatenation) sowie
- ? (Option)

...aufgebaut sind, heißen **reguläre Ausdrücke** (↑**Automatentheorie**).

Der Postfix-Operator „**?**“ besagt, dass das Argument eventuell auch fehlen darf, d.h. einmal oder keinmal vorkommt.

Beispiele

Reguläre Ausdrücke reichen zur Beschreibung **einfacher** Mengen von Worten aus.

- ▶ `(letter letter)*`
⇒ alle Wörter gerader Länge (über `$_, a, . . . , z, A, . . . , Z`);
- ▶ `letter* test letter*`
⇒ alle Wörter, die das Teilwort `test` enthalten;
- ▶ `_ digit* 17`
⇒ alle Wörter, die mit `_` anfangen, dann eine beliebige Folge von Ziffern aufweisen, die mit `17` aufhört;
- ▶ `exp ::= (e|E)(+|-)? digit digit*`
`float ::= digit digit* exp | digit* (digit . | . digit) digit* exp?`
⇒ alle Gleitkommazahlen...

Reguläre Ausdrücke

Ausdrücke, die aus Zeichen(-klassen) mithilfe von

- | (Alternative)
- * (Iteration)
- (Konkatenation) sowie
- ? (Option)

...aufgebaut sind, heißen **reguläre Ausdrücke** (↑**Automatentheorie**).

Der Postfix-Operator „?“ besagt, dass das Argument eventuell auch fehlen darf, d.h. einmal oder keinmal vorkommt.

Programmverarbeitung

1. Phase (↑Scanner)

Identifizierung von

- ▶ reservierten Wörtern,
- ▶ Namen,
- ▶ Konstanten

Ignorierung von

- ▶ Whitespace,
- ▶ Kommentaren

⇒ Input wird mit regulären Ausdrücken verglichen und dabei in Wörter („Tokens“) zerlegt.

2. Phase (↑Parser)

Analyse der Struktur des Programms.

Beispiele

Reguläre Ausdrücke reichen zur Beschreibung **einfacher** Mengen von Worten aus.

- ▶ `(letter letter)*`
⇒ alle Wörter gerader Länge (über `$_, a, . . . , z, A, . . . , Z`);
- ▶ `letter* test letter*`
⇒ alle Wörter, die das Teilwort `test` enthalten;
- ▶ `_ digit* 17`
⇒ alle Wörter, die mit `_` anfangen, dann eine beliebige Folge von Ziffern aufweisen, die mit `17` aufhört;
- ▶ `exp ::= (e|E)(+|-)? digit digit*`
`float ::= digit digit* exp | digit* (digit . | . digit) digit* exp?`
⇒ alle Gleitkommazahlen...

Programmverarbeitung

1. Phase (↑Scanner)

Identifizierung von

- ▶ reservierten Wörtern,
- ▶ Namen,
- ▶ Konstanten

Ignorierung von

- ▶ Whitespace,
- ▶ Kommentaren

⇒ Input wird mit regulären Ausdrücken verglichen und dabei in Wörter („Tokens“) zerlegt.

2. Phase (↑Parser)

Analyse der **Struktur** des Programms.

Beispiele

Reguläre Ausdrücke reichen zur Beschreibung **einfacher** Mengen von Worten aus.

- ▶ `(letter letter)*`
⇒ alle Wörter gerader Länge (über `$_, a, . . . , z, A, . . . , Z`);
- ▶ `letter* test letter*`
⇒ alle Wörter, die das Teilwort `test` enthalten;
- ▶ `_ digit* 17`
⇒ alle Wörter, die mit `_` anfangen, dann eine beliebige Folge von Ziffern aufweisen, die mit `17` aufhört;
- ▶ `exp ::= (e|E)(+|-)? digit digit*`
`float ::= digit digit* exp |`
`digit* (digit . | . digit) digit* exp?`
⇒ alle Gleitkommazahlen...

3.4 Struktur von Programmen

Programme sind **hierarchisch** aus Komponenten aufgebaut. Für jede Komponente geben wir Regeln an, wie sie aus anderen Komponenten zusammengesetzt sein können.

```
program ::= decl* stmt*  
decl   ::= type name ( , name )* ;  
type   ::= int
```

- ▶ Ein Programm besteht aus einer Folge von Deklarationen, gefolgt von einer Folge von Statements.
- ▶ Eine Deklaration gibt den Typ an, hier: `int`, gefolgt von einer Komma-separierten Liste von Variablennamen.

3.4 Struktur von Programmen

Programme sind **hierarchisch** aus Komponenten aufgebaut. Für jede Komponente geben wir Regeln an, wie sie aus anderen Komponenten zusammengesetzt sein können.

```
program ::= decl* stmt*  
decl    ::= type name ( , name )* ;  
type    ::= int
```

- ▶ Ein Programm besteht aus einer Folge von Deklarationen, gefolgt von einer Folge von Statements.
- ▶ Eine Deklaration gibt den Typ an, hier: **int**, gefolgt von einer Komma-separierten Liste von Variablennamen.

```
stmt ::= ; | { stmt* } |  
      name = expr; | name = read(); |  
      write( expr ); |  
      if ( cond ) stmt |  
      if ( cond ) stmt else stmt |  
      while ( cond ) stmt
```

- ▶ Ein Statement ist entweder „leer“ (d.h. gleich ;) oder eine geklammerte Folge von Statements;
- ▶ oder eine Zuweisung, eine Lese- oder Schreiboperation;
- ▶ eine (einseitige oder zweiseitige) bedingte Verzweigung;
- ▶ oder eine Schleife.

Programme sind **hierarchisch** aus Komponenten aufgebaut. Für jede Komponente geben wir Regeln an, wie sie aus anderen Komponenten zusammengesetzt sein können.

```
program ::= decl* stmt*  
decl    ::= type name ( , name )* ;  
type    ::= int
```

- ▶ Ein Programm besteht aus einer Folge von Deklarationen, gefolgt von einer Folge von Statements.
- ▶ Eine Deklaration gibt den Typ an, hier: **int**, gefolgt von einer Komma-separierten Liste von Variablennamen.

Ausdrücke

```
expr ::= number | name | ( expr ) |  
      unop expr | expr binop expr  
unop ::= -  
binop ::= - | + | * | / | %
```

- ▶ Ein Ausdruck ist eine Konstante, eine Variable oder ein geklammerter Ausdruck
- ▶ oder ein unärer Operator, angewandt auf einen Ausdruck,
- ▶ oder ein binärer Operator, angewandt auf zwei Argumentausdrücke.
- ▶ Einziger unärer Operator ist (bisher) die Negation.
- ▶ Mögliche binäre Operatoren sind Addition, Subtraktion, Multiplikation, (ganzzahlige) Division und Modulo.

Anweisungen

```
stmt ::= ; | { stmt* } |  
      name = expr; | name = read(); |  
      write( expr ); |  
      if ( cond ) stmt |  
      if ( cond ) stmt else stmt |  
      while ( cond ) stmt
```

- ▶ Ein Statement ist entweder „leer“ (d.h. gleich ;) oder eine geklammerte Folge von Statements;
- ▶ oder eine Zuweisung, eine Lese- oder Schreiboperation;
- ▶ eine (einseitige oder zweiseitige) bedingte Verzweigung;
- ▶ oder eine Schleife.

Bedingungen

```
cond ::= true | false | ( cond ) |  
      expr comp expr |  
      bunop cond | cond bbinop cond  
comp  ::= == | != | <= | < | >= | >  
bunop  ::= !  
bbinop ::= && | ||
```

- ▶ Bedingungen unterscheiden sich dadurch von Ausdrücken, dass ihr Wert nicht vom Typ `int` ist sondern `true` oder `false` (ein **Wahrheitswert** – vom Typ `boolean`).
- ▶ Bedingungen sind darum Konstanten, Vergleiche
- ▶ oder logische Verknüpfungen anderer Bedingungen.

Ausdrücke

```
expr ::= number | name | ( expr ) |  
      unop expr | expr binop expr  
unop  ::= -  
binop ::= - | + | * | / | %
```

- ▶ Ein Ausdruck ist eine Konstante, eine Variable oder ein geklammerter Ausdruck
- ▶ oder ein unärer Operator, angewandt auf einen Ausdruck,
- ▶ oder ein binärer Operator, angewand auf zwei Argumentausdrücke.
- ▶ Einziger unärer Operator ist (bisher) die Negation.
- ▶ Mögliche binäre Operatoren sind Addition, Subtraktion, Multiplikation, (ganzzahlige) Division und Modulo.

Beispiel

```
int x;  
x = read();  
if (x > 0)  
    write(1);  
else  
    write(0);
```

Die hierarchische Untergliederung von Programm-Bestandteilen veranschaulichen wir durch **Syntax-Bäume**.

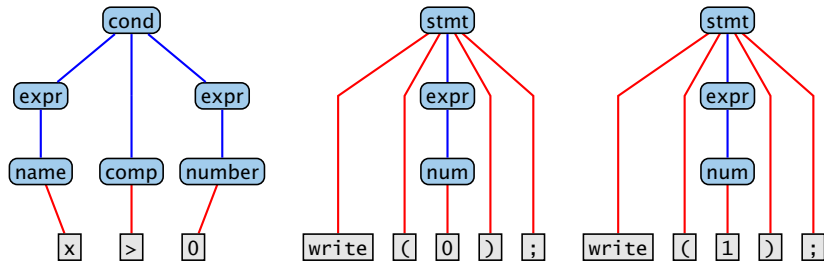
Bedingungen

```
cond ::= true | false | ( cond ) |  
      expr comp expr |  
      bunop cond | cond bbinop cond  
comp ::= == | != | <= | < | >= | >  
bunop ::= !  
bbinop ::= && | ||
```

- ▶ Bedingungen unterscheiden sich dadurch von Ausdrücken, dass ihr Wert nicht vom Typ `int` ist sondern `true` oder `false` (ein **Wahrheitswert** – vom Typ `boolean`).
- ▶ Bedingungen sind darum Konstanten, Vergleiche
- ▶ oder logische Verknüpfungen anderer Bedingungen.

Syntaxbäume

Syntaxbäume für $x > 0$ sowie `write(0);` und `write(1);`



Blätter: Wörter/Tokens
innere Knoten: Namen von Programmbestandteilen

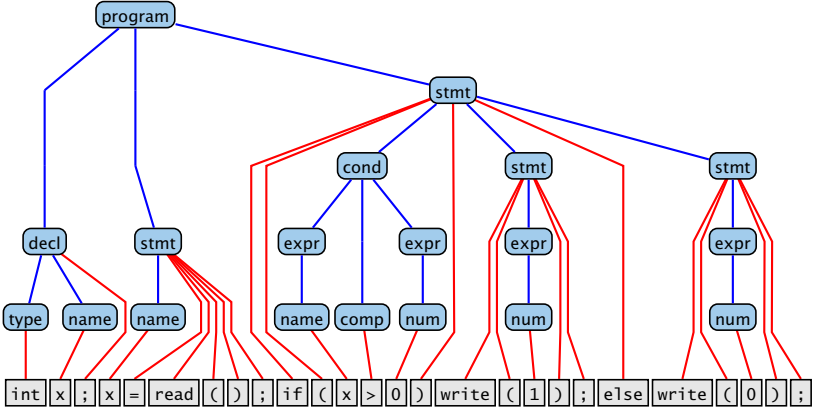
Beispiel

```
int x;  
x = read();  
if (x > 0)  
    write(1);  
else  
    write(0);
```

Die hierarchische Untergliederung von Programm-Bestandteilen veranschaulichen wir durch **Syntax-Bäume**.

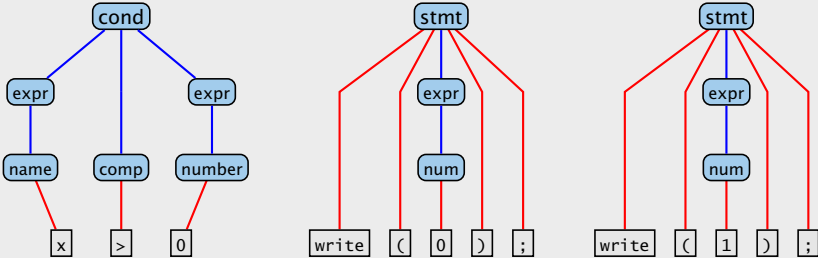
Beispiel

Der komplette Syntaxbaum unseres Beispiels:



Syntaxbäume

Syntaxbäume für $x > 0$ sowie `write(0);` und `write(1);`



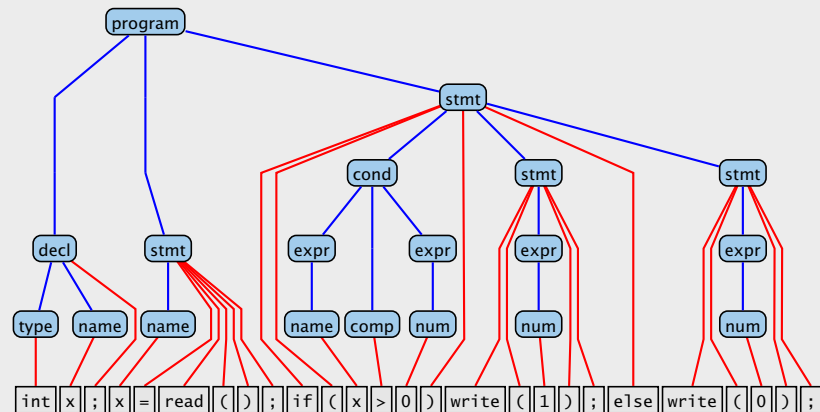
Blätter: Wörter/Tokens
innere Knoten: Namen von Programmbestandteilen

Bemerkungen

- ▶ Die vorgestellte Methode der Beschreibung von Syntax heißt **EBNF-Notation** (**E**xtended **B**ackus **N**aur **F**orm **N**otation).
- ▶ Ein anderer Name dafür ist **erweiterte kontextfreie Grammatik** (↑**Linguistik**, ↑**Automatentheorie**).
- ▶ Linke Seiten von Regeln heißen auch **Nichtterminale**.
- ▶ Tokens heißen auch **Terminale**.

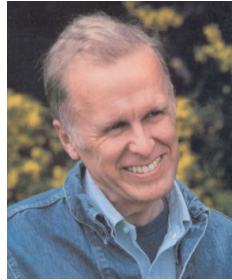
Beispiel

Der komplette Syntaxbaum unseres Beispiels:

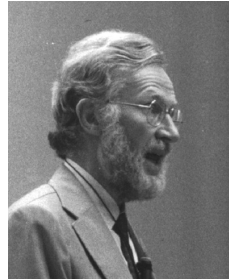




Noam Chomsky,
MIT



John Backus, IBM
Turing Award
(Erfinder von Fortran)



Peter Naur,
Turing Award
(Erfinder von Algol60)

Bemerkungen

- ▶ Die vorgestellte Methode der Beschreibung von Syntax heißt **EBNF**-Notation (**E**xtended **B**ackus **N**aur **F**orm Notation).
- ▶ Ein anderer Name dafür ist **erweiterte kontextfreie Grammatik** (↑Linguistik, ↑Automatentheorie).
- ▶ Linke Seiten von Regeln heißen auch **Nichtterminale**.
- ▶ Tokens heißen auch **Terminale**.

Kontextfreie Grammatiken

Achtung:

- ▶ Die regulären Ausdrücke auf den rechten Regelseiten können sowohl Terminale wie Nichtterminale enthalten.
- ▶ Deshalb sind kontextfreie Grammatiken **mächtiger** als reguläre Ausdrücke.

Beispiel:

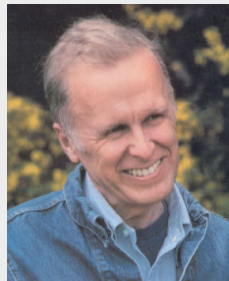
$$\mathcal{L} = \{\epsilon, ab, aabb, aaabbb, \dots\}$$

lässt sich mithilfe einer Grammatik beschreiben:

$$A ::= (aAb)?$$



Noam Chomsky,
MIT



John Backus, IBM
Turing Award
(Erfinder von Fortran)



Peter Naur,
Turing Award
(Erfinder von Algol60)

