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

Plagiatsstatistik 2016/17

Bachelor Informatik	482	17	3.5%
Bachelor Games Engineering	172	15	8.7%
Bachelor Wirtschaftsinformatik	186	23	12.4%
Master Wirtschaft mit Technologie	86	9	10.5%
Bachelor Technologie und Management	180	19	10.6%
Bachelor Maschinenwesen	2	1	50.0%
Master Maschinenwesen	1	1	100.0%
Bachelor Mathematik	88	3	3.4%
Summe	1197	88	7.4%

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

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 $f : E \to A$, mit
$E =$ zulässige Eingaben und $A =$ mögliche Ausgaben.

Beispiele:

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

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

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

Beispiel: Alltagsalgorithmen

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

Beispiel: Euklidischer Algorithmus

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

Algorithmus:

1. Falls $a = b$, brich Berechnung ab. Es gilt $\mathrm{ggT}(a, b) = a$. 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

Hier sind $q_a, q_b, q'_{a-b}, q'_b \in \mathbb{Z}$.

Warum geht das?

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

Seien $g = \mathrm{ggT}(a,b)$, $g' = \mathrm{ggT}(a-b,b)$.
Dann gilt:

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

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

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

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

Eigenschaften

Ein klassischer Algorithmus erfüllt alle Eigenschaften. Häufig spricht man aber auch von Algorithmen wenn einige dieser Eigenschaften verletzt sind.

(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 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 Programmmiersprachen: Java, C, Prolog, Fortran, TeX, PostScript, ...

Programm

Eine Programmiersprache 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.

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.

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.

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 – Input/Output

Achtung:

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

In MiniJava können auch direkt Strings ausgegeben werden:

Beispiel:

- `write("Hello World!!!");`
  Schreibt `Hello World!!!` 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.

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:

- 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:

- . . . 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:

- . . . 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 – 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);
```

Das Programm erfüllt die Spezifikation, dass es für $x, y \in \mathbb{N} \setminus \{0\}$, den GGT berechnet. Bei falscher Eingabe terminiert es eventuell nicht.
Im allgemeinen sollte man Eingaben vom Benutzer immer auf Zulässigkeit prüfen.

- 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

Eine Sprache mit dieser Eigenschaft nennt man auch turingvollständig.

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.

Für dieses Theorem ist es wichtig, dass die `int`-Variablen von MiniJava beliebige ganze Zahlen speichern können. Sobald man dies einschränkt (z.B. 32 Bit) ist die entstehende Sprache immer noch sehr mächtig aber streng formal nicht mehr turingvollständig.

2 Eine einfache Programmiersprache

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

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);
```


Ein 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;
```

Datei "GGT.java"

Ein Java-Programm

Erläuterungen:

- Jedes Programm hat einen Namen (hier GGT)
- Der Name steht hinter dem Schlüsselwort `class` (was eine Klasse ist, was `public` ist lernen wir später)
- Der Dateiname muss zum Programmnamen „passen“, d.h. in diesem Fall `GGT.java` heißen.
- Das MiniJava-Programm ist der Rumpf des Hauptprogramms, d.h. der Funktion `main()`.
- Die Programmausführung eines Java-Programms startet stets mit einem Aufruf dieser Funktion `main()`.
- Die Operationen `write()` und `read()` werden in der Klasse `MiniJava` definiert.
- Durch `GGT extends MiniJava` machen wir diese Operationen innerhalb des GGT-Programms verfügbar.

```
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",
               JOptionPane.PLAIN_MESSAGE);
16         f.dispose ();
17     }
18     public static void write (int x) { write(""+x); }
19 }
```

Datei: "MiniJava.java"

Ein Java-Programm

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
   laaaaaaaaaaaang
      */
```


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.

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


Ausblick

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!

3 Syntax von Programmiersprachen

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 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.

Beobachtung

Programmiersprachen sind

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

Syntax vs. Semantik

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...

Syntax vs. Semantik

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!!!

3.1 Reservierte Wörter

- `int`
  $\Longrightarrow$ Bezeichner für Basistypen;
- `if, else, then, while...`
  $\Longrightarrow$ Schlüsselwörter für Programmkonstrukte;
- `(,), ",', {,}, ,;`
  $\Longrightarrow$ 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?

Wir definieren hier MiniJava. Eigentliches Java erlaubt mehr Namen (z.B. sind UTF8-Symbole erlaubt).

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.

Beispiele

`_178`	`5ABC`
`Das_ist_kein_Name`	`!Hallo!`
`x`	`x'`
`_`	`a=b`
`$Password$`	`-178`
...sind legale Namen.	...sind keine legalen Namen.

Achtung
Reservierte Wörter sind als Namen verboten.

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

Reguläre Ausdrücke

Die Alternative hat eine geringere Bindungsstärke als die Konkatenation. D.h. ab|c steht für die Wörter ab oder c und nicht für ab oder ac.

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.

Gelegentlich sind auch ϵ, d.h. das „leere Wort“ sowie $\emptyset$, d.h. die leere Menge zugelassen.

Beispiele

Reguläre Ausdrücke reichen zur Beschreibung einfacher Mengen von Worten aus.

- (letter letter)*
  ⟹ alle Wörter gerader Länge (über $,_,a,\ldots,z,A,\ldots,Z$);
- 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

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

2. Phase (↑Parser)

Analyse der Struktur des Programms.

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.

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.

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.

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.

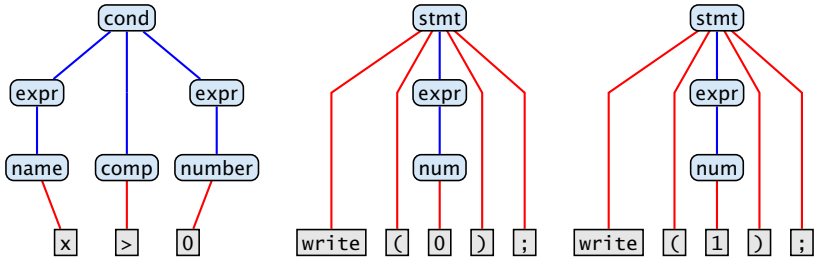
Beispiel

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

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

Syntaxbäume

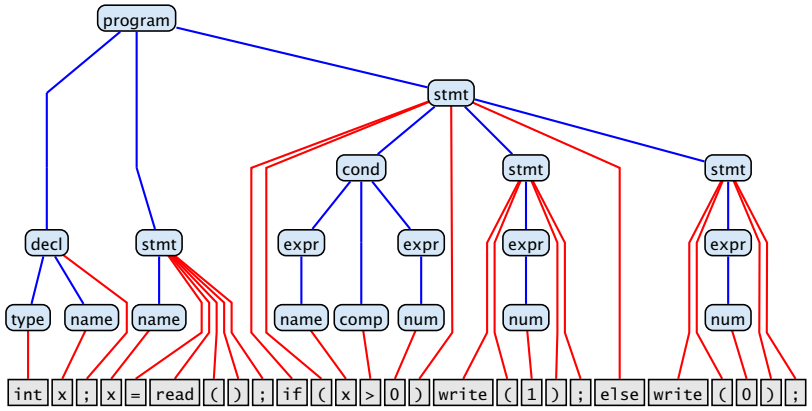
Syntaxbäume für `x > 0` sowie `write(0);` und `write(1);`



Blätter: Wörter/Tokens
innere Knoten: Namen von Programmbestandteilen

Beispiel

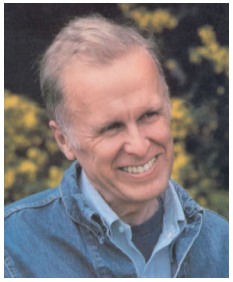
Der komplette Syntaxbaum unseres Beispiels:



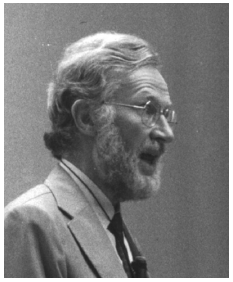
Bemerkungen

- Die vorgestellte Methode der Beschreibung von Syntax heißt EBNF-Notation (Extended Backus Naur Form Notation).
- Ein anderer Name dafür ist erweiterte kontextfreie Grammatik (↑Linguistik, ↑Automatentheorie).
- Linke Seiten von Regeln heißen auch Nichtterminale.
- Tokens heißen auch Terminale.

Noam Chomsky, MIT



John Backus, IBM
Turing Award
(Erfinder von Fortran)



Peter Naur,
Turing Award
(Erfinder von Algol60)

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:

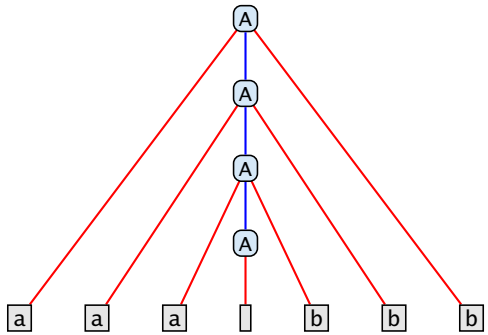
$$\mathcal{L} = \{\epsilon, ab, aabb, aaabbb, \ldots\}$$

lässt sich mithilfe einer Grammatik beschreiben:

$$A \quad ::= \quad (\; a \; A \; b \;)?$$

Kontextfreie Grammatiken

Syntaxbaum für das Wort aaabbb:

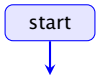


Für $\mathcal{L}$ gibt es aber keinen regulären Ausdruck (↑Automatentheorie).

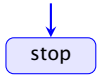
4 Kontrollflussdiagramme

In welcher Weise Programmteile nacheinander ausgeführt werden kann anschaulich durch Kontrollflussdiagramme dargestellt werden.

Zutaten:

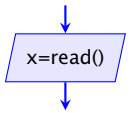


Startknoten

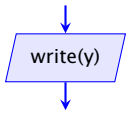


Endknoten

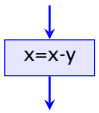
4 Kontrollflussdiagramme



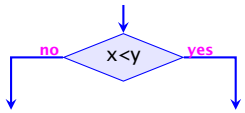
Eingabe



Ausgabe

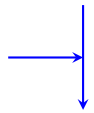


Zuweisung



bedingte Verzweigung

Kante



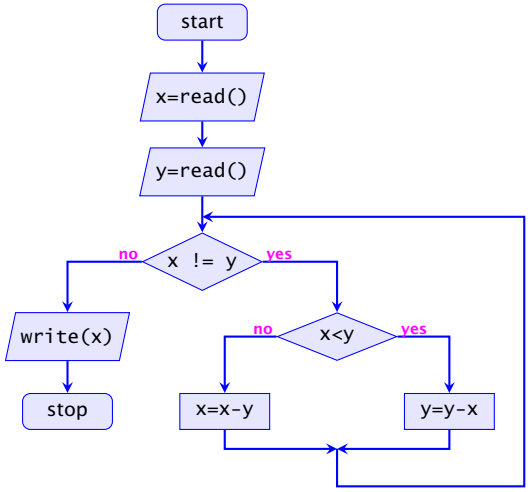
Zusammenlauf

4 Kontrollflussdiagramme

Beispiel:

```
int x, y;
x = read();
y = read();
while (x != y) {
    if (x < y)
        y = y - x;
    else
        x = x - y;
}
write(x);
```

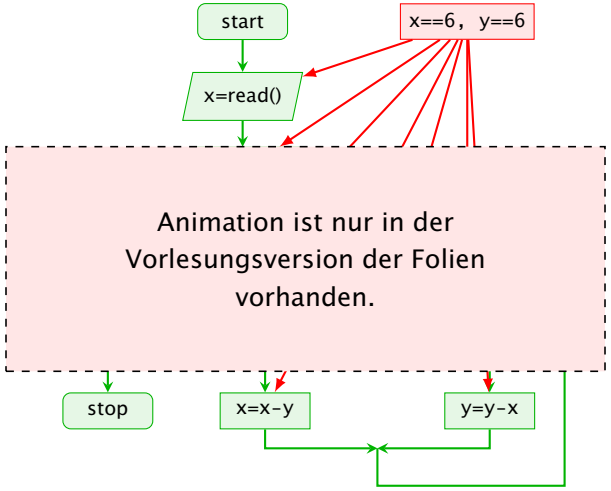
GGT



4 Kontrollflussdiagramme

- Die Ausführung des Programms entspricht einem Pfad durch das Kontrollflussdiagramm vom Startknoten zum Endknoten.
- Die Deklaration von Variablen muss man sich am Startknoten vorstellen.
- Die auf dem Pfad liegenden Knoten (außer Start- und Endknoten) sind Operationen bzw. auszuwertende Bedingungen.
- Um den Nachfolger an einem Verzweigungsknoten zu bestimmen, muss die Bedingung mit den aktuellen Werten der Variablen ausgewertet werden. (↑operationelle Semantik)

4 Kontrollflussdiagramme



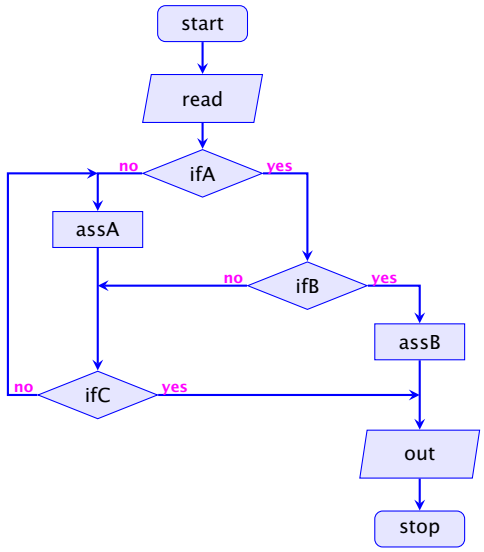
4 Kontrollflussdiagramme

- zu jedem MiniJava-Programm lässt sich ein Kontrollflussdiagramm konstruieren;
- die Umkehrung gilt auch, liegt aber nicht sofort auf der Hand

Die Umkehrung wäre sehr einfach zu bewerkstelligen, wenn wir in einem MiniJava-Programm `goto`-Befehle benutzen dürften, d.h. wenn wir von jedem Punkt zu jedem anderen innerhalb des Programms springen könnten.

Die obige Aussage bedeutet im Prinzip, dass man `goto`-Befehle immmer durch geeignete Schleifen ersetzen kann.

4 Kontrollflussdiagramme



5 Mehr Java

Java ist statisch typisiert, d.h., Variablen, Ergebnisse von Ausdrücken, etc. haben einen Datentyp, der schon bei der Kompilierung festgelegt wird.

Java unterscheidet zwei Arten von Typen:

- Basistypen / Primitive Datentypen
  `byte`, `char`, `short`, `int`, `long`, `float`, `double`, `boolean`
- Referenzdatentypen
  kann man auch selber definieren

Beispiel – Statische Typisierung

```python
a = 5
a = a + 1
a = "Hello World." # a is now a string
a = a + 1          # runtime error
```

Python

```java
int a;
a = 5;
a = "Hello World." // will not compile
```

Java

5.1 Basistypen

Primitive Datentypen

- Zu jedem Basistypen gibt es eine Menge möglicher Werte.
- Jeder Wert eines Basistyps benötigt den gleichen Platz, um ihn im Rechner zu repräsentieren.
- Der Platz wird in Bit gemessen.

Wie viele Werte kann man mit n Bit darstellen?

Primitive Datentypen – Ganze Zahlen

Es gibt vier Sorten ganzer Zahlen:

Typ	*Platz*	*kleinster Wert*	*größter Wert*
byte	8	−128	127
short	16	−32 768	32 767
int	32	−2 147 483 648	2 147 483 647
long	64	−9 223 372 036 854 775 808	9 223 372 036 854 775 807

Die Benutzung kleinerer Typen wie byte oder short spart Platz.

Primitive Datentypen – Ganze Zahlen

Literale:

- dezimale Notation
- hexadezimale Notation (Präfix 0x oder 0X)
- oktale Notation (Präfix 0)
- binäre Notation (Präfix 0b oder 0B)
- Suffix l☹ oder L für long
- '_' um Ziffern zu gruppieren

Beispiele

- 192, 0b11000000, 0xC0, 0300 sind alle gleich
- 20_000L, 0xABFF_0078L
- 09, 0x_FF sind ungültig

Verwenden Sie niemals l als long-Suffix, da dieses leicht mit 1 verwechselt werden kann.

_ darf nur **zwischen** Ziffern stehen, d.h. weder am Anfang noch am Ende.

Übung:
Geben Sie eine reguläre Grammatik an, die diese Regeln abbildet

Primitive Datentypen – Ganze Zahlen

Achtung: Java warnt nicht vor Überlauf/Unterlauf!!!

Beispiel:

```
1 int x = 2147483647; // groesstes int
2 x = x + 1;
3 write(x);
```

liefert: -2147483648

- In realem Java kann man bei der Deklaration einer Variablen ihr direkt einen ersten Wert zuweisen (Initialisierung).
- Man kann sie sogar (statt am Anfang des Programms) erst an der Stelle deklarieren, an der man sie braucht!

Primitive Datentypen – Gleitkommazahlen

Es gibt zwei Sorten von Gleitkommazahlen:

Typ	*Platz*	*kleinster Wert*	*größter Wert*	*signifikante Stellen*
`float`	32	ca. $-3.4 \cdot 10^{38}$	ca. $3.4 \cdot 10^{38}$	ca. 7
`double`	64	ca. $-1.7 \cdot 10^{308}$	ca. $1.7 \cdot 10^{308}$	ca. 15

$$x = s \cdot m \cdot 2^e \qquad \text{mit } 1 \le m < 2$$

- Vorzeichen s: 1 bit
- reduzierte Mantisse $m - 1$: 23 bit (`float`), 52 bit (`double`)
- Exponent e: 8 bit (`float`), 11 bit (`double`)

Primitive Datentypen – Gleitkommazahlen

Literale:

- dezimale Notation.
- dezimale Exponentialschreibweise (e, E für Exponent)

Mantisse und Exponent sind dezimal; Basis für Exponent ist 10;

- hexadezimale Exponentialschreibweise. (Präfix 0x oder 0X, p oder P für Exponent)

Mantisse ist hexadezimal; Exponent ist dezimal und muß vorhanden sein; Basis für Exponent ist 2;

- Suffix f oder F für `float`, Suffix d oder D für `double` (default is `double`)

In der hexadezimalen Notation, gibt der Exponent die Anzahl der Bitpositionen an, um die das Komma verschoben wird.

Wenn der Exponent in der hexadeximalen Notation, hexadezimal wäre, wüßten wir nicht, ob ein finales 'f' zum Exponenten gehört, oder ein `float`-Suffix sein soll.

`0x1e3d` ist ein `int` und keine Gleitkommazahl

`1e3_d` ist ungültig, da '_' nicht zwischen 2 Ziffern steht (`d` ist keine Ziffer sondern das `double`-Suffix)

Beispiele

- `640.5F` == `0x50.1p3f`
- `3.1415` == `314.15E-2`
- `0x1e3_dp0`, `1e3d`
- `0x1e3d`, `1e3_d`, `0x50.1`

Primitive Datentypen – Gleitkommazahlen

- Überlauf/Unterlauf bei Berechnungen liefert `Infinity`, bzw. `-Infinity`
- Division Null durch Null, Wurzel aus einer negativen Zahl etc. liefert `NaN`

Weitere Basistypen

Typ	Platz	Werte
boolean	1	true,false
char	16	alle(?) Unicode-Zeichen

Unicode ist ein Zeichensatz, der alle irgendwo auf der Welt gängigen Alphabete umfasst, also zum Beispiel:

- die Zeichen unserer Tastatur (inklusive Umlaute);
- die chinesischen Schriftzeichen;
- die ägyptischen Hieroglyphen ...

Die ursprüngliche Idee war, dass char alle Unicodezeichen enthält. Nach der Einführung von Java, hat sich der Unicodestandard geändert. Deshalb kann ein char nur Zeichen der sogenannten Basic Multilingual Plane speichern. Andere Unicodezeichen werden über Strings codiert.

Literale:

- char-Literale schreibt man in Hochkommata: 'A', '\u00ED',';', '\n'.
- boolean-Literale sind true und false.

5.2 Strings

Der Datentyp String für Wörter ist ein Referenzdatentyp (genauer eine Klasse (dazu kommen wir später)).

Hier nur drei Eigenschaften:

- Literale vom Typ String haben die Form "Hello World!";
- Man kann Wörter in Variablen vom Typ String abspeichern;
- Man kann Wörter mithilfe des Operators '+' konkatenieren.

Beispiel

```
String s0 = "";
String s1 = "Hel";
String s2 = "lo Wo";
String s3 = "rld!";

write(s0 + s1 + s2 + s3);
```

...liefert: Hello World!

5.3 Auswertung von Ausdrücken

Funktionen in Java bekommen Parameter/Argumente als Input, und liefern als Output den Wert eines vorbestimmten Typs. Zum Beispiel könnte man eine Funktion

```
int min(int a, int b)
```

implementieren, die das Minimum ihrer Argumente zurückliefert.

Operatoren sind spezielle vordefinierte Funktionen, die in Infix-Notation geschrieben werden (wenn sie binär sind):

```
a + b = +(a,b)
```

Funktionen, werden hier nur eingeführt, weil wir sie bei der Ausdrucksauswertung benutzen möchten. Eine detaillierte Einführung erfolgt später.

5.3 Auswertung von Ausdrücken

Ein Ausdruck ist eine Kombination von Literalen, Operatoren, Funktionen, Variablen und Klammern, die verwendet wird, um einen Wert zu berechnen.

Beispiele: (x z.B. vom Typ `int`)

- `7 + 4`
- `3 / 5 + 3`
- `min(3,x) + 20`
- `x = 7`
- `x *= 2`

Operatoren

Unäre +/--Operatoren konvertieren `byte`, `short`, `char` zuerst nach `int`.

Man kann keinen legalen Ausdruck bilden, bei der die Assoziativität der Postfix-Operatoren (Gruppe Priorität 2) eine Rolle spielen würde.

Unäre Operatoren:

symbol	*name*	*types*	*L/R*	*level*
++	Post-inkrement	(var) Zahl, char	keine	2
--	Post-dekrement	(var) Zahl, char	keine	2
++	Pre-inkrement	(var) Zahl, char	rechts	3
--	Pre-dekrement	(var) Zahl, char	rechts	3
+	unäres Plus	Zahl, char	rechts	3
-	unäres Minus	Zahl, char	rechts	3
!	Negation	boolean	rechts	3

Die Spalte „L/R“ beschreibt die Assoziativität des Operators.

Die Spalte „level“ die Priorität.

Im Folgenden sind (für binäre Operatoren) beide Operanden jeweils vom gleichen Typ.

„Zahl“ steht hier für einen der Zahltypen `byte`, `short`, `int`, `long`, `float` oder `double`.

Achtung

Diese Beschreibung der Vorrangregeln in Form von Prioritäten für Operatoren findet sich nicht im Java Reference Manual. Dort wird nur die formale kontextfreie Grammatik von Java beschrieben. Die Vorrangregeln leiten sich daraus ab und erleichtern den Umgang mit Ausdrücken, da man nicht in die formale Grammatik schauen muß um einen Ausdruck zu verstehen.

Es gibt im Internet zahlreiche teils widersprüchliche Tabellen, die die Vorrangregeln von Java-Operatoren beschreiben :(. Die gesamte Komplexität der Ausdruckssprache von Java läßt sich wahrscheinlich nicht in dieses vereinfachte Schema pressen.

Prefix- und Postfixoperator

- Die Operatoranwendungen ++x und x++ inkrementieren beide den Wert der Variablen x (als Seiteneffekt).
- ++x tut das, bevor der Wert des Ausdrucks ermittelt wird (Pre-Inkrement).
- x++ tut das, nachdem der Wert ermittelt wurde (Post-Inkrement).
- `b = x++;` entspricht:

```
b = x;
x = x + 1;
```

- `b = ++x;` entspricht:

```
x = x + 1;
b = x;
```

Die Entsprechung gilt z.B. für `int`s. Für `short`s müßte es heißen:

```
b = x;
x = (short) (x + 1);
```

da `x = x + 1` nicht compiliert wenn `x` ein `short` ist.

`(short)` ist hier ein Typecast-Operator, den wir später kennenlernen.

Operatoren

Binäre arithmetische Operatoren:

`byte`, `short`, `char` werden nach `int` konvertiert

symbol	*name*	*types*	*L/R*	*level*
*	Multiplikation	Zahl, char	links	4
/	Division	Zahl, char	links	4
%	Modulo	Zahl, char	links	4
+	Addition	Zahl, char	links	5
-	Subtraktion	Zahl, char	links	5

Konkatenation

symbol	*name*	*types*	*L/R*	*level*
+	Konkatenation	String	links	5

Operatoren

Für Referenzdatentypen (kommt später) vergleichen die Operatoren == und != nur die Referenzen.

Vergleichsoperatoren:

symbol	*name*	*types*	*L/R*	*level*
>	größer	Zahl, char	keine	7
>=	größergleich	Zahl, char	keine	7
<	kleiner	Zahl, char	keine	7
<=	kleinergleich	Zahl, char	keine	7
==	gleich	alle	links	8
!=	ungleich	alle	links	8

Operatoren

Boolsche Operatoren:

symbol	*name*	*types*	*L/R*	*level*
&&	Und-Bedingung	boolean	links	12
||	Oder-Bedingung	boolean	links	13

Operatoren

Zuweisungsoperatoren:

symbol	*name*	*types*	*L/R*	*level*
=	Zuweisung	(links var) alle	rechts	15
*=,/=, %=, +=, -=	Zuweisung	(links var) alle	rechts	15

Für die letzte Form gilt:

$$v \circledcirc a \iff v = (type(v))\ (v \circ a)$$

Operatoren

Ein Seiteneffekt sind Änderungen von Zuständen/Variablen, die durch die Auswertung des Ausdrucks entstehen.

Warnung:

- Eine Zuweisung `x = y;` ist in Wahrheit ein Ausdruck.
- Der Wert ist der Wert der rechten Seite.
- Die Modifizierung der Variablen x erfolgt als Seiteneffekt.
- Das Semikolon `';'` hinter einem Ausdruck wirft nur den Wert weg.

Fatal für Fehler in Bedingungen:

In C ist diese Art des Fehlers noch wesentlich häufiger, da auch z.B. `x = 1` (für `int x`) in der Bedingung vorkommen kann. Das Ergebnis des Ausdrucks (1) wird in den boolschen Wert `true` konvertiert. Letzteres ist in Java nicht möglich.

```
boolean x = false;
if (x = true)
   write("Sorry! This must be an error ...");
```

In Java kann man durch das ';' aus den meisten Ausdrücken eine Anweisung machen, die nur den Seiteneffekt des Ausdrucks durchführt.

5.3 Auswertung von Ausdrücken

Assoziativität

- Die Assoziativität entscheidet über die Reihenfolge bei Operatoren gleicher Priorität. (links = der linkeste Operator wird zuerst ausgeführt)
- Alle Operatoren einer Prioritätsgruppe haben dieselbe Assoziativität.
- Bis auf Zuweisungsoperatoren (=, +=, etc.) sind alle binären Operatoren linksassoziativ.
- unäre Operatoren, die ihr Argument rechts erwarten sind rechtsassoziativ
- unäre Operatoren, die ihr Argument links erwarten (postfix-Operatoren ++, --) sind linksassoziativ
- Der ternäre Bedingungsoperator (später) ist rechtsassoziativ

5.3 Auswertung von Ausdrücken

Die Auswertung eines Ausdrucks liefert

- eine Variable (**var**),
- einen reinen Wert (**val**) oder
- void (**void**)

In den ersten beiden Fällen hat der Ausdruck dann einen

- Typ, z.B.: **int**, und einen
- Wert, z.B.: 42

Für z.B. Zuweisungen muss die Auswertung des Ausdrucks auf der linken Seite eine Variable ergeben!!!

5.3 Auswertung von Ausdrücken

In Java werden Unterausdrücke von links nach rechts ausgewertet. D.h. um den Wert einer Operation zu berechnen:

- werte (rekursiv) alle Operanden von links nach rechts aus
- führe die Operation auf den Resultaten aus

Ausnahmen: ||, &&, und der ternäre Bedingungsoperator :?, werten nicht alle Operanden aus (Kurzschlussauswertung).

Man sollte nie Ausdrücke formulieren, deren Ergebnis von der Auswertungsreihenfolge abhängt!!!

Eine Kurzschlussauswertung ist natürlich ok. Dafür gibt es sehr nützliche Anwendungen.

In C/C++, ist die Auswertungsreihenfolge nicht definiert, d.h., sie ist compilerabhängig.

Den Bedingungsoperator lernen wir später kennen.

5.3 Auswertung von Ausdrücken

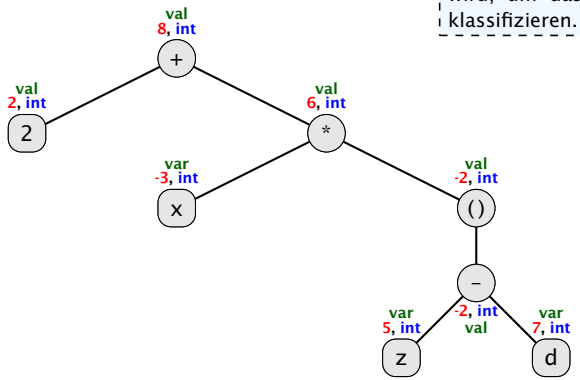
Im Folgenden betrachten wir Klammern als einen Operator der nichts tut:

symbol	*name*	*types*	*L/R*	*level*
( )	Klammerung	alle	links	0

Beispiel: 2 + x * (z - d)

Punkt geht vor Strichrechnung.

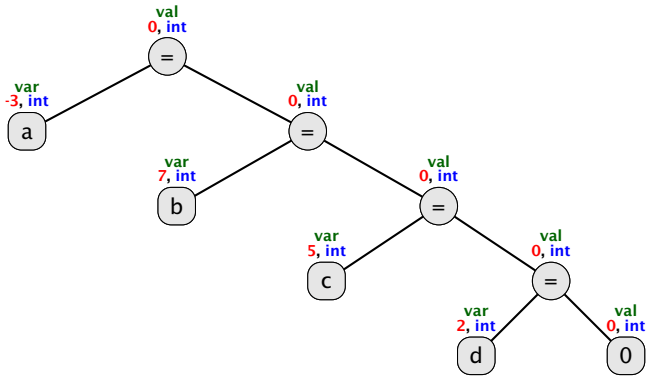
Ganzahlliterale sind vom Typ `int`, wenn nicht z.B. ein `L` angehängt wird, um das Literal als `long` zu klassifizieren.



x -3 d 7 z 5

Beispiel: a = b = c = d = 0

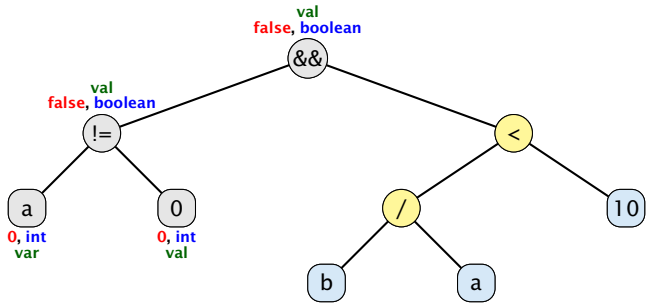
Das funktioniert nur, da der Zuweisungsoperator rechtsassoziativ ist.



a 0 b 0 c 0 d 0

Beispiel: a != 0 && b/a < 10

Die vollständige Auswertung der Operanden würde hier zu einem Laufzeitfehler führen (Division durch Null).
Mit Kurzschlussauswertung ist alles ok.

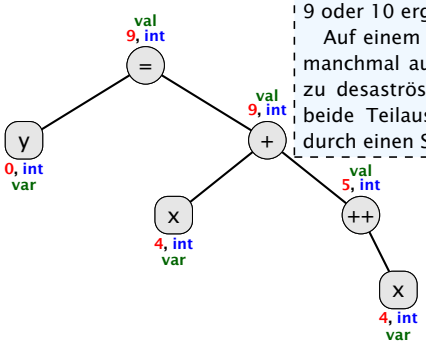


a 0 b 4

Beispiel: y = x + ++x

In C ist die Reihenfolge der Auswertung von Unterausdrücken nicht definiert. Auf einem sequentiellen Rechner hängt die Reihenfolge vom Compiler ab und in diesem Beispiel könnte dies das Resultat 9 oder 10 ergeben.

Auf einem Parallelrechner können Teilausdrücke manchmal auch parallel ausgewertet werden, was zu desaströsen Konsequenzen führen kann, falls beide Teilausdrücke eine Variable enthalten, die durch einen Seiteneffekt verändert wird.

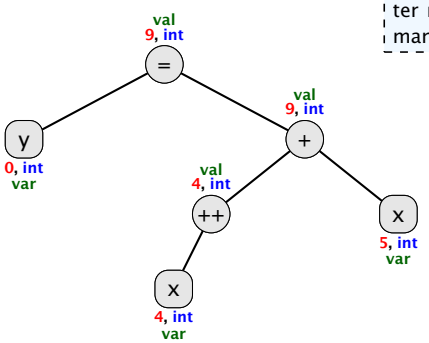


x 5 y 9

Beispiel: y = x++ + x

Der Postfix-Operator ändert die Variable nach dem der Wert des Teilausdrucks bestimmt wurde.

Wenn die Variable im Ausdruck später nochmal ausgewertet wird, bekommt man den neuen Wert.



x 5 y 9

Impliziter Typecast

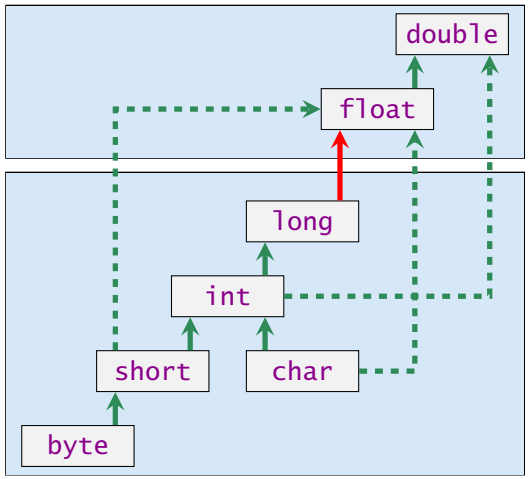
Wenn ein Ausdruck vom TypA an einer Stelle verwendet wird, wo ein Ausdruck vom TypB erforderlich ist, wird

- entweder der Ausdruck vom TypA in einen Ausdruck vom TypB gecasted (impliziter Typecast),
- oder ein Compilerfehler erzeugt, falls dieser Cast nicht (automatisch) erlaubt ist.

Beispiel: Zuweisung

```
long x = 5;
int  y = 3;
x = y; // impliziter Cast von int nach long
```


Erlaubte Implizite Typecasts – Numerische Typen



Gleitkommazahlen

Man nennt diese Art der Casts, widening conversions, da der Wertebereich im Allgemeinen erweitert wird.

ganze Zahlen, char

Keine Typumwandlung zwischen boolean und Zahltypen (weder implizit noch explizit).

Konvertierung von `long` nach `double` oder von `int` nach `float` kann Information verlieren wird aber automatisch durchgeführt.

Welcher Typ wird benötigt?

Operatoren sind üblicherweise überladen, d.h. ein Symbol (+, -, ...) steht in Abhängigkeit der Parameter (Argumente) für unterschiedliche Funktionen.

```
+ : int ⟶ int
+ : long ⟶ long
+ : float ⟶ float
+ : double ⟶ double

+ : int × int ⟶ int
+ : long × long ⟶ long
+ : float × float ⟶ float
+ : double × double ⟶ double

+ : String × String ⟶ String
```

Es gibt keinen +-Operator für `short`, `byte`, `char`.

Der +-Operator für Strings macht Konkatenation.

Der Compiler muss in der Lage sein während der Compilierung die richtige Funktion zu bestimmen.

Impliziter Typecast

Der Compiler wertet nur die Typen des Ausdrucksbaums aus.

- Für jeden inneren Knoten wählt er dann die geeignete Funktion (z.B. `+ : long × long ⟶ long` falls ein +-Knoten zwei `long`-Argumente erhält).
- Falls keine passende Funktion gefunden wird, versucht der Compiler durch implizite Typecasts die Operanden an eine Funktion anzupassen.
- Dies geschieht auch für selbstgeschriebene Funktionen (z.B. `min(int a, int b)` und `min(long a, long b)`).
- Der Compiler nimmt die Funktion mit der speziellsten Signatur.

Speziellste Signatur

1. Der Compiler bestimmt zunächst alle Funktionen, die passen könnten (d.h. die vorliegenden Typen können durch widening conversions in die Argumenttypen der Funktion umgewandelt werden).
2. Eine Funktion f_1 ist spezifischer als eine andere f_2, wenn die Argumenttypen von f_1 auch für einen Aufruf von f_2 benutzbar sind (z.B. `min(int, long)` spezifischer als `min(long, long)` aber nicht spezifischer als `min(long, int)`).
   Dieses definiert eine partielle Ordnung auf der Menge der Funktionen.
3. Unter den möglichen Funktionen (aus Schritt 1) wird ein kleinste Element bzgl. dieser partiellen Ordnung gesucht. Falls genau ein kleinstes Element existiert, ist dies die gesuchte Funtion. Andernfalls ist der Aufruf ungültig. (Beachte: Rückgabetyp spielt für Funktionsauswahl keine Rolle).

Ordnungsrelationen

Relation $\preceq$: TypA $\preceq$ TypB falls TypA nach TypB (implizit) gecasted werden kann:

- **reflexiv:** $T \preceq T$
- **transitiv:** $T_1 \preceq T_2 \wedge T_2 \preceq T_3 \Rightarrow T_1 \preceq T_3$
- **antisymmetrisch:** $T_1 \preceq T_2 \wedge T_2 \preceq T_1 \Rightarrow T_1 = T_2$

d.h., $\preceq$ definiert Halbordnung auf der Menge der Typen.

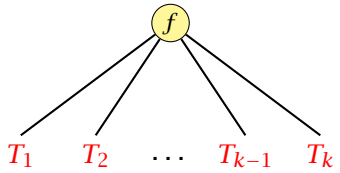
Relation $\preceq_k$: $(T_1, \ldots, T_k) \preceq_k (T'_1, \ldots, T'_k)$ falls $T_i \preceq T'_i$ für alle $i \in \{1, \ldots, k\}$:

- **reflexiv:** $\mathcal{T} \preceq_k \mathcal{T}$
- **transitiv:** $\mathcal{T}_1 \preceq_k \mathcal{T}_2 \wedge \mathcal{T}_2 \preceq_k \mathcal{T}_3 \Rightarrow \mathcal{T}_1 \preceq_k \mathcal{T}_3$
- **antisymmetrisch:** $\mathcal{T}_1 \preceq_k \mathcal{T}_2 \wedge \mathcal{T}_2 \preceq_k \mathcal{T}_1 \Rightarrow \mathcal{T}_1 = \mathcal{T}_2$

d.h., $\preceq_k$ definiert Halbordnung auf Menge der k-Tupel von Typen

Wir betrachten Relation auf der Menge von Parametertupeln für die f implementiert ist. Aus Antisymmetrie folgt, dass keine zwei Funktionen das gleiche k-Tupel an Parametern erwarten.

$R_1 \quad f(\mathcal{T}_1)$
$R_2 \quad f(\mathcal{T}_2)$
$\vdots$
$R_\ell \quad f(\mathcal{T}_\ell)$



$\mathcal{T}_1, \ldots, \mathcal{T}_\ell$ sind k-Tupel von Typen für die eine Definition von f existiert.

$\mathcal{T} = (T_1, \ldots, T_k)$ ist das k-tupel von Typen mit dem f aufgerufen wird.

Menge aller möglichen Funktionen/Tupel:

$$M := \{\mathcal{T}_i | \mathcal{T} \preceq_k \mathcal{T}_i\} \ .$$

Wähle kleinstes Element aus M falls M ein eindeutig kleinstes Element besitzt (sonst Compilerfehler).

Impliziter Typecast – Numerische Typen

Angenommen wir haben Funktionen

```
int min(int a, int b)
float min(float a, float b)
double min(double a, double b)
```

definiert.

```
1 long a = 7, b = 3;
2 double d = min(a, b);
```

würde die Funktion `float min(float a, float b)` aufrufen.

Impliziter Typecast

Bei Ausdrücken mit Seiteneffekten (Zuweisungen, ++ , --) gelten andere Regeln:

Beispiel: Zuweisungen

$= : \texttt{byte}^* \times \texttt{byte} \longrightarrow \texttt{byte}$
$= : \texttt{char}^* \times \texttt{char} \longrightarrow \texttt{char}$
$= : \texttt{short}^* \times \texttt{short} \longrightarrow \texttt{short}$
$= : \texttt{int}^* \times \texttt{int} \longrightarrow \texttt{int}$
$= : \texttt{long}^* \times \texttt{long} \longrightarrow \texttt{long}$
$= : \texttt{float}^* \times \texttt{float} \longrightarrow \texttt{float}$
$= : \texttt{double}^* \times \texttt{double} \longrightarrow \texttt{double}$

Es wird nur der Parameter konvertiert, der nicht dem Seiteneffekt unterliegt.

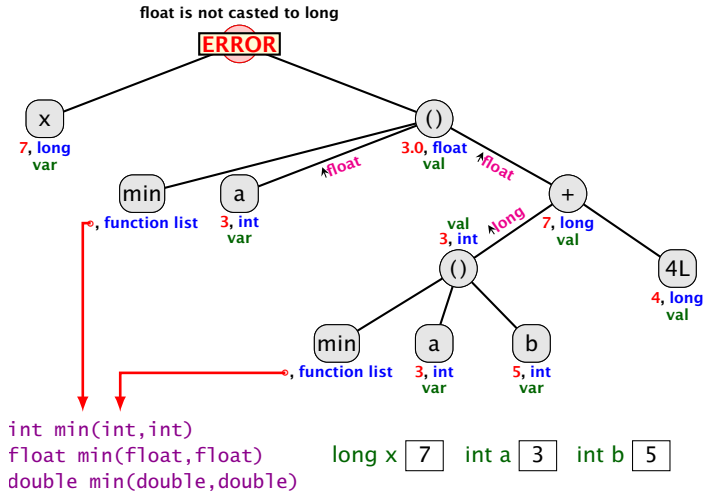
5.3 Auswertung von Ausdrücken

Der Funktionsaufrufoperator:

symbol	*name*	*types*	*L/R*	*level*
`()`	Funktionsaufruf	Funktionsname, *	links	1

Wir modellieren, den Funktionsaufrufoperator hier als einen Operator, der beliebig viele Argumente entgegennimmt. Das erste Argument ist der Funktionsname, und die folgenden Argumente sind die Parameter der Funktion. Üblicherweise hat der Funktionsaufrufoperator nur zwei Operanden: den Funktionsnamen, und eine Argumentliste.

Beispiel: x = min(a, min(a,b) + 4L)



Achtung: Dieses ist eine sehr vereinfachte und teilweise inkorrekte Darstellung. Der eigentliche Prozess, der vom Funktionsnamen zu eigentlichen Funktion führt ist sehr kompliziert. **function list** ist auch kein Typ in Java.

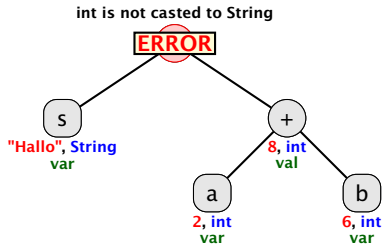
Impliziter Typecast – Strings

Spezialfall

- Falls beim Operator + ein Typ vom Typ String ist, wird der andere auch in einen String umgewandelt.
  $\Longrightarrow$ Stringkonkatenation.
- Jeder Typ in Java besitzt eine Stringrepresentation.

Funktioniert nicht bei selbstgeschriebenen Funktionen.

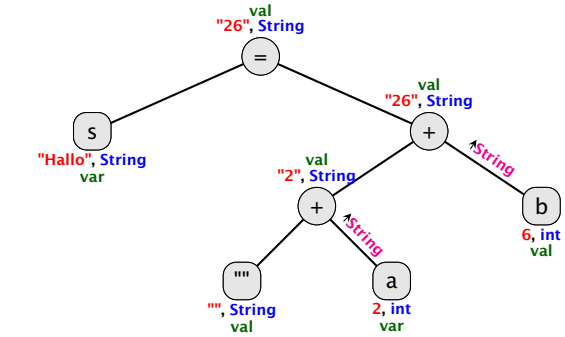
Beispiel: s = a + b



String s "Hallo" a 2 b 6

Beispiel: s = "" + a + b

Strings are immutable! Falls eine weitere Referenz auf "Hallo" verweist, hat sich für diese nichts geändert.

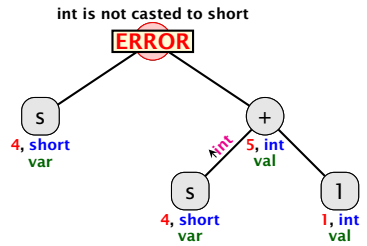


"26"

String s "Hallo" a 2 b 6

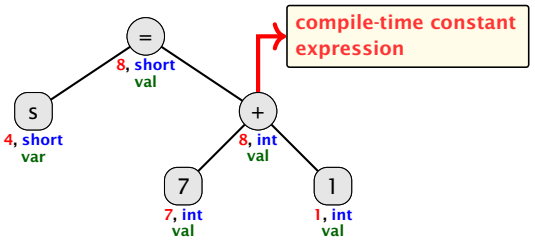
Achtung: vereinfachte Darstellung!!! Eigentlich arbeitet Java mit Ojekten vom Typ StringBuffer um den +- Operator zu realisieren...

Beispiel: s = s + 1



short s 4

Beispiel: s = 7 + 1



short s 8

Wenn der `int`-Ausdruck, der zugewiesen werden soll, zu Compilierzeit bekannt ist, und er in einen `short` „passt“, wird der Cast von `int` nach `short` durchgeführt.

Funktioniert nicht für `long`-Ausdrücke, d.h., `byte b = 4L;` erzeugt einen Compilerfehler.

Expliziter Typecast

symbol	*name*	*type*	*L/R*	*level*
(type)	typecast	Zahl, char	rechts	3

Beispiele mit Datenverlust

- `short s = (short) 23343445;`

  Die obersten bits werden einfach weggeworfen...
- ```
  double d = 1.5;
  short  s = (short) d;
  ```

  d hat danach den Wert 1.

...ohne Datenverlust:

- ```
  int   x = 5;
  short s = (short) x;
  ```

Man kann einen cast zwischen zahltypen erzwingen (evtl. mit Datenverlust). Typecasts zwischen Referenzdatentypen kommt später.

5.4 Arrays

Oft müssen viele Werte gleichen Typs gespeichert werden.

Idee:

- Lege sie konsekutiv ab!
- Greife auf einzelne Werte über ihren Index zu!

Feld:	17	3	-2	9	0	1
Index:	0	1	2	3	4	5

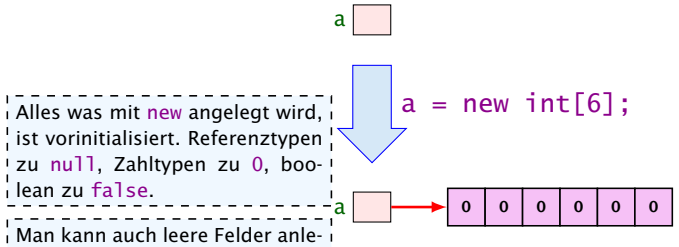
Beispiel

```
1 int[] a; // Deklaration
2 int n = read();
3
4 a = new int[n]; // Anlegen des Felds
5 int i = 0;
6 while (i < n) {
7     a[i] = read();
8     i = i + 1;
9 }
```

Einlesen eines Feldes

Beispiel

- `type[] name;` deklariert eine Variable für ein Feld (array), dessen Elemente vom Typ `type` sind.
- Alternative Schreibweise:
  `type name[];`
- Das Kommando `new` legt ein Feld einer gegebenen Größe an und liefert einen Verweis darauf zurück:

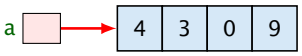


Alles was mit `new` angelegt wird, ist vorinitialisiert. Referenztypen zu `null`, Zahltypen zu `0`, boolean zu `false`.

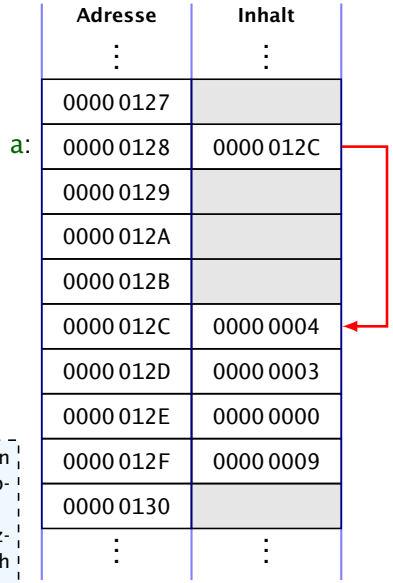
Man kann auch leere Felder anlegen: `new int[0]`.

Was ist eine Referenz?

Eine Referenzvariable speichert eine Adresse; an dieser Adresse liegt der eigentliche Inhalt der Variablen.



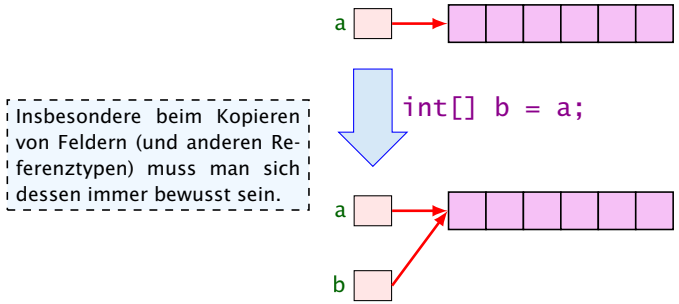
Wir können die Referenz nicht direkt manipulieren (nur über den `new`-Operator, oder indem wir eine andere Referenz zuweisen).



Eine Referenz zeigt daduch nie auf einen beliebigen Ort im Speicher; sie zeigt immer auf ein gültiges Objekt oder auf das `null`-Objekt.
Wir geben üblicherweise nie den Wert einer Referenzvariablen an, sondern symbolisieren diesen Wert durch einen Pfeil auf die entsprechenden Daten.

5.4 Arrays

- Der Wert einer Feld-Variable ist also ein Verweis!!!
- `int[] b = a;` kopiert den Verweis der Variablen `a` in die Variable `b`:



Insbesondere beim Kopieren von Feldern (und anderen Referenztypen) muss man sich dessen immer bewusst sein.

- **Alle nichtprimitive Datentypen sind Referenzentypen, d.h., die zugehörige Variable speichert einen Verweis!!!**

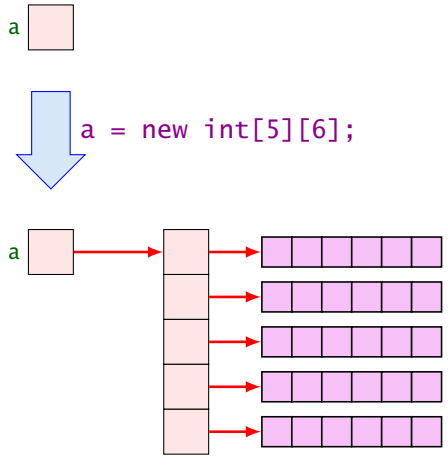
5.4 Arrays

- Die Elemente eines Feldes sind von 0 an durchnummeriert.
- Die Anzahl der Elemente des Feldes `name` ist `name.length`.
- Auf das i-te Element greift man mit `name[i]` zu.
- Bei jedem Zugriff wird überprüft, ob der Index erlaubt ist, d.h. im Intervall {0,...,`name.length-1`} liegt.
- Liegt der Index außerhalb des Intervalls, wird eine `ArrayIndexOutOfBoundException` ausgelöst (↑Exceptions).

Sie sollten beim Programmieren möglichst nie diese Exception bekommen. In anderen Sprachen (z.B. C/C++) gibt es aus Effizienzgründen keine Überprüfung. Wenn Sie in einer solchen Sprache einen solchen Fehler verursachen, ist der sehr schwierig zu finden.

Mehrdimensionale Felder

- Java unterstützt direkt nur eindimensionale Felder.
- ein zweidimensionales Feld ist ein Feld von Feldern...



Der new-Operator

So etwas wie `new int[3][][4]` macht keinen Sinn, da die Größe dieses Typs nicht vom Compiler bestimmt werden kann.

symbol	*name*	*types*	*L/R*	*level*
new	new	Typ, Konstruktor	links	1

Erzeugt ein Objekt/Array und liefert eine Referenz darauf zurück.

1. Version: Erzeugung eines Arrays (Typ ist Arraytyp)
   - `new int[3][7];` oder auch
   - `new int[3][];` (ein Array, das 3 Verweise auf `int` enthält)
   - `new String[10];`
   - `new int[]{1,2,3};` (ein Array mit den `int`s 1, 2, 3)

2. Version: Erzeugung eines Objekts durch Aufruf eines Konstruktors
   - `String s = new String("Hello World!");`

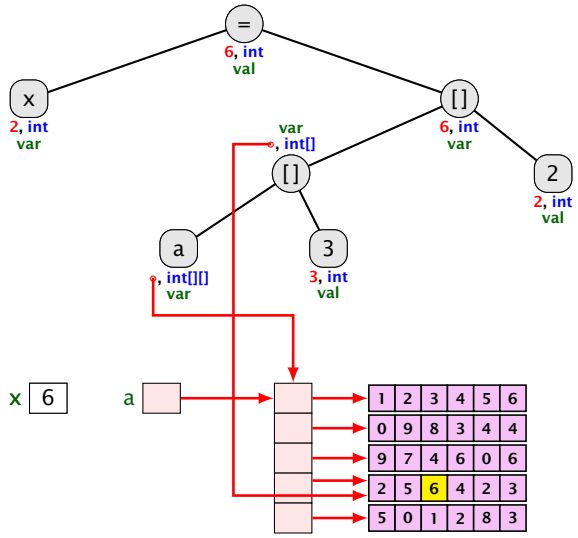
Was genau ein Konstruktor ist kommt später.

Der Index-Operator

symbol	*name*	*types*	*L/R*	*level*
[]	index	array, int	links	1

Zugriff auf ein Arrayelement.

Beispiel: x = a[3][2]



Der .-Operator

symbol	*name*	*types*	*L/R*	*level*
.	member access	Array/Objekt/Class, Member	links	1

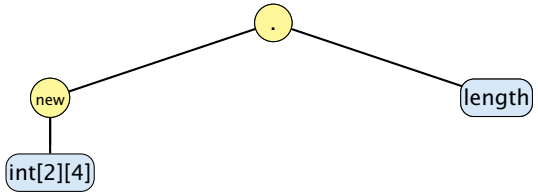
Zugriff auf Member.

Beispiel:

- `x = new int[2][4].length`
  x hat dann den Wert 2.

Beispiel: new int[2][4].length

Das Parsing für den new-Operator passt nicht in das Schema:



Beachte den Unterschied zwischen `new int[2][3]` und `(new int[2])[3]`. Bei letzterem ist das zweite Klammerpaar ein Index-Operator während es beim ersten Ausdruck zum Typ gehört.

Arrayinitialisierung

1. `int[] a = new int[3];`
   `a[0] = 1; a[1] = 2; a[2] = 3;`
2. `int[] a = new int[]{ 1, 2, 3};`
3. `int[] a = new int[3]{ 1, 2, 3};`
4. `int[] a = { 1, 2, 3};`
5. `char[][] b = { {'a','b'}, new char[3], {} };`
6. `char[][] b;`
   `b = new char[][]{ {'a','b'}, new char[3], {} };`
7. `char[][] b;`
   `b = { {'a','b'}, new char[3], {} };`

5.5 Mehr Kontrollstrukturen

Typische Form der Iteration über Felder:

- Initialisierung des Laufindex;
- `while`-Schleife mit Eintrittsbedingung für den Rumpf;
- Modifizierung des Laufindex am Ende des Rumpfs.

Beispiel

```
1 int result = a[0];
2 int i = 1;        // Initialisierung
3 while (i < a.length) {
4     if (a[i] < result)
5         result = a[i];
6     i = i + 1;    // Modifizierung
7 }
8 write(result);
```

Bestimmung des Minimums

Das For-Statement

```
1 int result = a[0];
2 for (int i = 1; i < a.length; ++i)
3     if (a[i] < result)
4         result = a[i];
5 write(result);
```

Bestimmung des Minimums

Das For-Statement

`for (init; cond; modify) stmt`

entspricht:

`{ init; while (cond) { stmt modify; } }`

Erläuterungen:

- `++i;` ist äquivalent zu `i = i + 1;`
- die `while`-Schleife steht innerhalb eines Blocks (`{...}`)

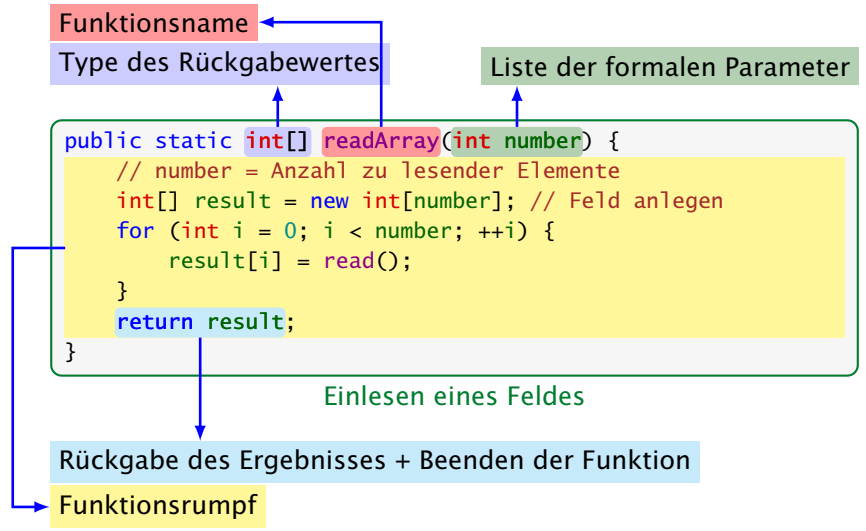
  die Variable `i` ist außerhalb dieses Blocks nicht sichtbar/zugreifbar

5.6 Funktionen und Prozeduren

Oft möchte man:

- Teilprobleme separat lösen; und dann
- die Lösung mehrfach verwenden.

Beispiel



5.6 Funktionen und Prozeduren

Erläuterungen:

- Die erste Zeile ist der Header der Funktion.
- `public` und `static` kommen später
- `int[]` gibt den Typ des Rückgabe-Werts an.
- `readArray` ist der Name, mit dem die Funktion aufgerufen wird.
- Dann folgt (in runden Klammern und komma-separiert) die Liste der formalen Parameter, hier: `(int number)`.
- Der Rumpf der Funktion steht in geschweiften Klammern.
- `return expr;` beendet die Ausführung der Funktion und liefert den Wert von `expr` zurück.

5.6 Funktionen und Prozeduren

Erläuterungen:

- Die Variablen, die innerhalb eines Blocks angelegt werden, d.h. innerhalb von `'{'` und `'}'`, sind nur innerhalb dieses Blocks sichtbar, d.h. benutzbar (lokale Variablen).
- Der Rumpf einer Funktion ist ein Block.
- Die formalen Parameter können auch als lokale Variablen aufgefasst werden.
- Bei dem Aufruf `readArray(7)` erhält der formale Parameter `number` den Wert 7 (aktueller Parameter).

Beispiel

```
public static int min(int[] b) {
    int result = b[0];
    for (int i = 1; i < b.length; ++i) {
        if (b[i] < result)
            result = b[i];
    }
    return result;
}
```

Bestimmung des Minimums

Beispiel

```
public class Min extends MiniJava {
    public static int[] readArray(int number) { ... }
    public static int min(int[] b) { ... }
    // Jetzt kommt das Hauptprogramm
    public static void main(String[] args) {
        int n = read();
        int[] a = readArray(n);
        int result = min(a);
        write(result);
    } // end of main()
} // end of class Min
```

Programm zur Minimumsberechnung

Beispiel

Erläuterungen:

- Manche Funktionen, deren Ergebnistyp `void` ist, geben gar keine Werte zurück – im Beispiel: `write()` und `main()`. Diese Funktionen heißen Prozeduren.
- Das Hauptprogramm hat immer als Parameter ein Feld `args` von `String`-Elementen.
- In diesem Argument-Feld werden dem Programm Kommandozeilen-Argumente verfügbar gemacht.

```
public class Test extends MiniJava {
    public static void main (String[] args) {
        write(args[0]+args[1]);
    }
} // end of class Test
```


Beispiel

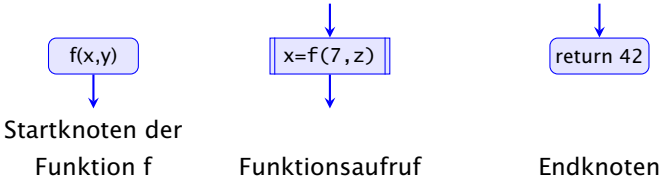
Der Aufruf

```
java Test "Hel" "lo World!"
```

liefert: `Hello World!`

5.6 Funktionen und Prozeduren

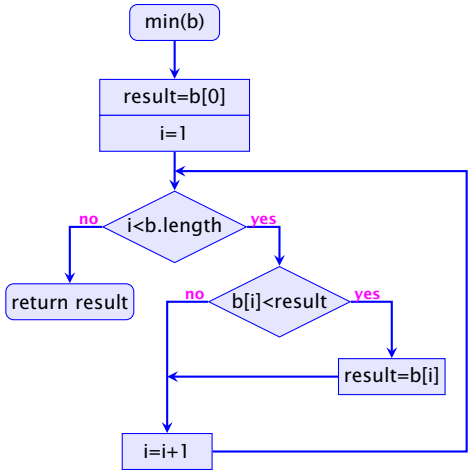
Um die Arbeitsweise von Funktionen zu veranschaulichen erweitern/modifizieren wir die Kontrollflussdiagramme



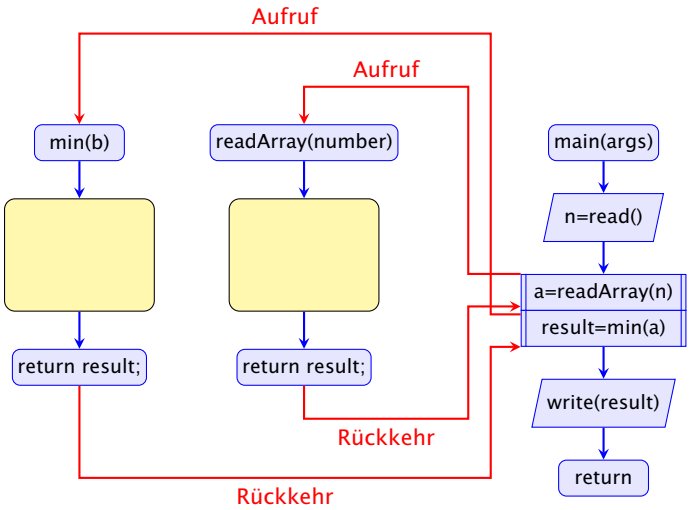
- Für jede Funktion wird ein eigenes Teildiagramm erstellt.
- Ein Aufrufknoten repäsentiert eine Teilberechnung der aufgerufenen Funktion.

5.6 Funktionen und Prozeduren

Teildiagramm der Funktion `min()`:



5.6 Funktionen und Prozeduren

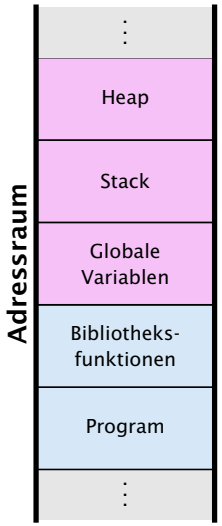


6 Speicherorganisation

Der Speicher des Programms ist in verschiedene Speicherbereiche untergliedert

- Speicherbereiche, die den eigentlichen Programmcode und den Code der Laufzeitbibliothek enthalten;
- einen Speicherbereich für globale/statische Variablen;
- einen Speicherbereich Heap, und
- einen Speicherbereich Stack.

Variablen werden üblicherweise auf dem Heap oder dem Stack gespeichert.



Heap vs. Stack vs. statisch

Heap

Auf dem Heap können zur Laufzeit zusammenhängende Speicherbereiche angefordert werden, und in beliebiger Reihenfolge wieder freigegeben werden.

Stack

Der Stack ist ein Speicherbereich, auf dem neue Elemente oben gespeichert werden, und Freigaben in umgekehrter Reihenfolge (d.h. oben zuerst) erfolgen müssen (LIFO = Last In First Out).

Statische Variablen

Statische Variablen werden zu Beginn des Programms angelegt, und zum Ende des Programms wieder gelöscht.

In Java müssen Elemente auf dem Heap nicht explizit wieder freigegeben werden. Diese Freigabe übernimmt der ↑Garbage Collector.

Statische Variablen

Statische Variablen (auch Klassenvariablen) werden im Klassenrumpf ausserhalb einer Funktion mit dem zusätzlichen Schlüsselwort `static` definiert.

Jede Funktion der Klasse kann dann diese Variablen benutzen; deshalb werden sie manchmal auch globale Variablen genannt.

Beispiel – Statische Variablen

```
1  public class GGT extends MiniJava {
2      static int x, y;
3      static void readInput() {
4          x = read();
5          y = read();
6      }
7      public static void main (String[] args) {
8          readInput();
9          while (x != y) {
10             if (x < y)
11                 y = y - x;
12             else
13                 x = x - y;
14         }
15         write(x);
16     }
17 }
```


Verwendung des Heaps

Speicherallokation mit dem Operator new:

```
int[][] arr;
arr = new int[10][]; // array mit int-Verweisen
```

Immer wenn etwas mit new angelegt wird, landet es auf dem Heap.

Wenn keine Referenz mehr auf den angeforderten Speicher existiert kann der Garbage Collector den Speicher freigeben:

```
int[][] arr;
arr = new int[10][]; // array mit int-Verweisen
arr = null; // jetzt koennte GC freigeben
```

Verwendung des Heaps

Beispiel:

```
1  public static int[] readArray(int number) {
2      // number = Anzahl zu lesender Elemente
3      int[] result = new int[number];
4      for (int i = 0; i < number; ++i) {
5          result[i] = read();
6      }
7      return result;
8  }
9  public static void main(String[] args) {
10     readArray(6);
11 }
```

Da die von readArray zurückgegebene Referenz nicht benutzt wird, kann der GC freigeben.

Verwendung des Heaps

Beispiel:

```
1  public static void main(String[] args) {
2      int[] b = readArray(6);
3      int[] c = b;
4      b = null;
5  }
```

Da c immer noch eine Referenz auf das array enthält erfolgt keine Freigabe.

Verwendung des Stacks

- Bei Aufruf einer Funktion (auch main()) werden lokale Variablen (d.h. auch Werte von aktuellen Parametern) und die Rücksprungadresse als Frames auf dem Stack gespeichert.
- Während der Programmausführung sind nur die Variablen im obersten Frame zugreifbar.
- Bei der Beendigung einer Funktion wird der zugehörige Stackframe gelöscht.

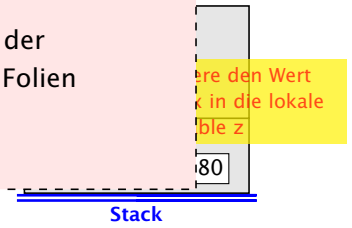
Parameterübergabe – Call-by-Value

Die Variable, die wir bei dem Aufruf übergeben, verändert ihren Wert nicht.

```
public static void setVar(int z) {
    z = 1;
}

public s
    int
    x =
    setV
    wri
}
```

Animation ist nur in der Vorlesungsversion der Folien vorhanden.

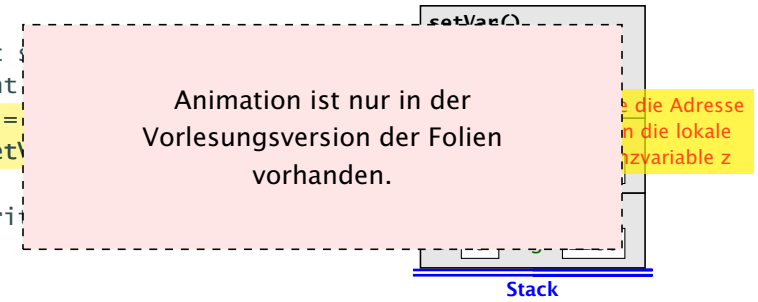


Das ist die einzige Form der Parameterübergabe, die Java unterstützt.

Parameterübergabe – Call-by-Reference

```
public static void setVar(int &z) {
    z = 1;
}

public
    int
    x =
    setV
    writ
}
```



Diese Form der Parameterübergabe ist in Java nicht möglich, aber z.B. in C++.

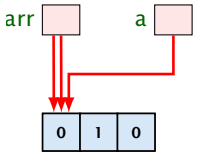
Parameterübergabe – Referenzvariablen

Auch Referenzvariablen werden per call-by-value übergeben. Man kann den Inhalt des zugehörigen Objekts/Arrays aber verändern.

```
public static void setVar(int[] a) {
    a[1] = 1;
}

public static void main(String[] args) {
    // initialize array elements to 0
    int[] arr = new int[3];

    setVar(arr);
    write("arr[1] == " + arr[1]);
}
```



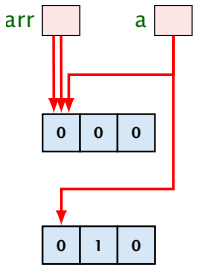
Ausgabe: arr[1] == 1

Parameterübergabe – Referenzvariablen

Wenn man das Objekt selber verändert, ist die Änderung nicht nach aussen sichtbar.

```
public static void setVar(int[] a) {
    a    = new int[3];
    a[1] = 1;
}

public static void main(String[] args) {
    // initialize array elements to 0
    int[] arr = new int[3];
    setVar(arr);
    write("arr[1] == " + arr[1]);
}
```



Ausgabe: arr[1] == 0

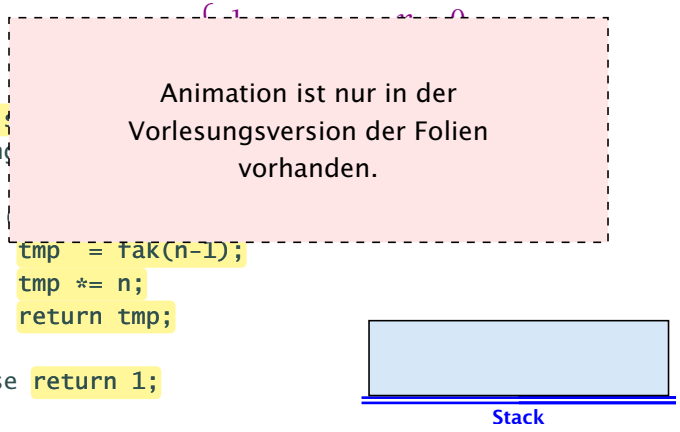
Rekursive Funktionen

Rekursive Funktionen sind Funktionen, die sich selber aufrufen (eventuell über Umwege).

Beispiel: Fakultätsberechnung

```
public
    long

    if
        tmp = fak(n-1);
        tmp *= n;
        return tmp;
    }
    else return 1;
}
```

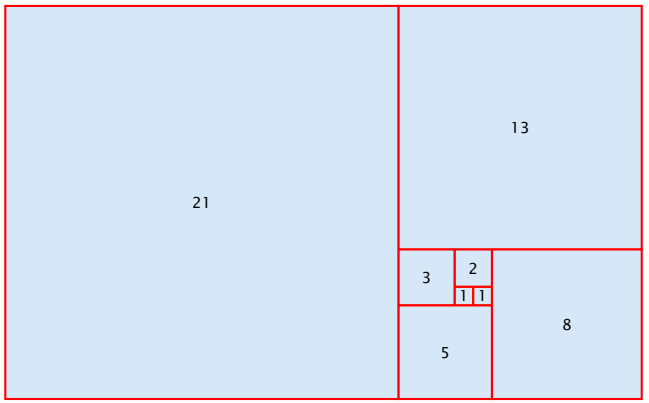


Vollständiger Code

```
1 public class Fakultaet {
2     public static long fak(int n) {
3         if (n > 0)
4             return n * fak(n-1);
5         else
6             return 1;
7     }
8     public static void main(String args[]) {
9         System.out.println(fak(20));
10    }
11 }
```


Fibonaccizahlen

$$F_n = \begin{cases} n & 0 \le n \le 1 \\ F_{n-1} + F_{n-2} & n \ge 2 \end{cases}$$



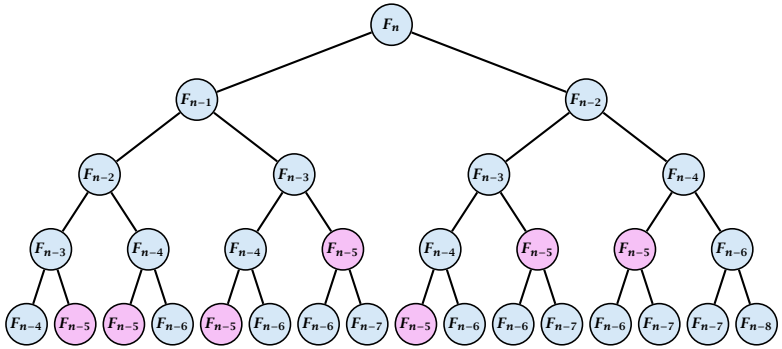
Vollständiger Code

```
1 public class Fibonacci {
2     public static long fib(int n) {
3         if (n > 1)
4             return fib(n-1)+fib(n-2);
5         else
6             return n;
7     }
8
9     public static void main(String args[]) {
10        System.out.println(fib(50));
11    }
12 }
```


Fibonaccizahlen

Programmlauf benötigt mehr als 1min.

Warum ist das so langsam?



Wir erzeugen viele rekursive Aufrufe für die gleichen Teilprobleme!

Fibonaccizahlen

Lösung

- Speichere die Lösung für ein Teilproblem in einer globalen Variable.
- Wenn das Teilproblem das nächste mal gelöst werden soll braucht man nur nachzuschauen...

Vollständiger Code

```
1  public class FibonacciImproved {
2      // F_93 does not fit into a long
3      static long[] lookup = new long[93];
4
5      public static long fib(int n) {
6          if (lookup[n] > 0) return lookup[n];
7
8          if (n > 1) {
9              lookup[n] = fib(n-1)+fib(n-2);
10             return lookup[n];
11         } else
12             return n;
13     }
14     public static void main(String args[]) {
15         System.out.println(fib(50));
16     }
17 }
```

Hier nutzen wir die Tatsache, dass der new-Operator long-Variablen mit dem Wert Null initialisert.

Achtung: lokale Variablen werden nicht initialisiert.

7 Anwendung: Sortieren

Gegeben: eine Folge von ganzen Zahlen.

Gesucht: die zugehörige aufsteigend sortierte Folge.

Idee:

- speichere die Folge in einem Feld ab;
- lege ein weiteres Feld an;
- füge der Reihe nach jedes Element des ersten Felds an der richtigen Stelle in das zweite Feld ein!

$\Longrightarrow$ Sortieren durch Einfügen (↑InsertionSort)

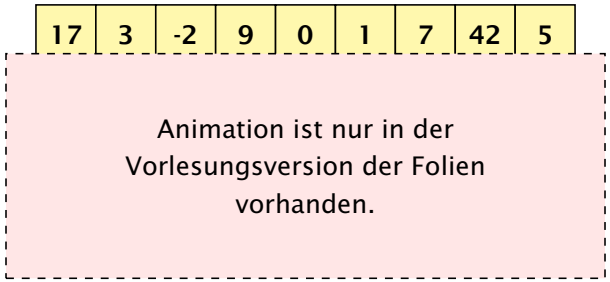
7 Anwendung: Sortieren

```
1  public static int[] sort(int[] a) {
2      int n = a.length;
3      int[] b = new int[n];
4      for (int i = 0; i < n; ++i)
5          insert(b, a[i], i);
6              // b    = Feld, in das eingefuegt wird
7              // a[i] = einzufuegendes Element
8              // i    = Anzahl von Elementen in b
9      return b;
10 } // end of sort ()
```

Sortieren durch Einfügen

Teilproblem: wie fügt man ein?

Beispiel



7 Anwendung: Sortieren

```
1 public static void insert(int[] b, int x, int i) {
2     // finde Einfuegestelle j fuer x in b
3     int j = locate(b,x,i);
4     // verschiebe in b Elemente b[j],...,b[i-1]
5     // nach rechts
6     shift(b,j,i);
7     b[j] = x;
8 }
```

Einfügen

- Wie findet man Einfügestelle?
- Wie verschiebt man nach rechts?

7 Anwendung: Sortieren

Das Programm ist immer noch korrekt, da locate mit Werten $i = 0, \ldots, n-1$ aufgerufen wird, d.h., die Arraygrenzen werden nicht überschritten.

```
public static int locate(int[] b, int x, int i) {
    int j = 0;
    while (j < i && x > b[j]) ++j;
    return j;
}
public static void shift(int[] b, int j, int i) {
    for (int k = i-1; k >= j; --k)
        b[k+1] = b[k];
}
```

- Warum läuft Iteration in `shift()` von `i-1` abwärts nach `j`?

7 Anwendung: Sortieren

Erläuterungen

- Das Feld b ist (ursprünglich) lokale Variable von `sort()`.
- Lokale Variablen sind nur im eigenen Funktionsrumpf sichtbar, nicht in den aufgerufenen Funktionen.
- Damit die aufgerufenen Hilfsfunktionen auf b zugreifen können, muss b explizit als Parameter übergeben werden!

  Achtung:
  Das Feld wird nicht kopiert. Das Argument ist der Wert der Variablen b, also nur eine Referenz!
- Deshalb benötigen weder `insert()`, noch `shift()` einen separaten Rückgabewert...
- Weil das Problem so klein ist, würde eine erfahrene Programmiererin hier keine Unterprogramme benutzen...

7 Anwendung: Sortieren

```
1  public static int[] sort(int[] a) {
2      int[] b = new int[a.length];
3      for (int i = 0; i < a.length; ++i) {
4          // begin of insert
5          int j = 0;
6          while (j < i && a[i] > b[j]) ++j;
7          // end of locate
8          for (int k = i-1; k >= j; --k)
9              b[k+1] = b[k];
10         // end of shift
11         b[j] = a[i];
12         // end of insert
13     }
14     return b;
15 } // end of sort
```


7 Anwendung: Sortieren

Diskussion

- Die Anzahl der ausgeführten Operationen wächst quadratisch in der Größe des Felds a.
- Glücklicherweise gibt es Sortierverfahren, die eine bessere Laufzeit haben (↑Algorithmen und Datenstrukturen).

8 Anwendung: Suchen

Gegeben: Folge a ganzer Zahlen; Element x

Gesucht: Wo kommt x in a vor?

Naives Vorgehen:

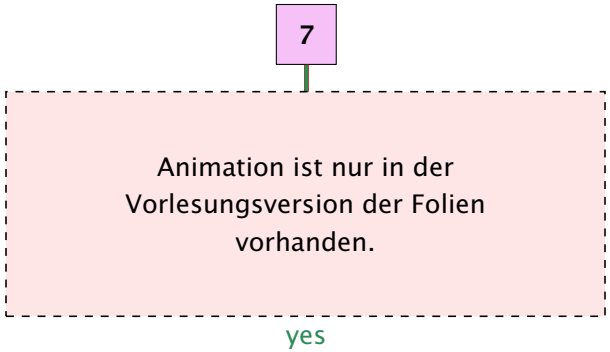
- Vergleiche x der Reihe nach mit a[0], a[1], usw.
- Finden wir i mit a[i] == x, geben wir i aus.
- Andernfalls geben wir -1 aus: „Element nicht gefunden“!

Naives Suchen

```
1 public static int find(int[] a, int x) {
2     int i = 0;
3     while (i < a.length && a[i] != x)
4         ++i;
5     if (i == a.length)
6         return -1;
7     else
8         return i;
9 }
```

Naives Suchen

Beispiel



Naives Suchen

- Im Beispiel benötigen wir 7 Vergleiche
- Im schlimmsten Fall (worst case) benötigen wir bei einem Feld der Länge n sogar n Vergleiche.
- Kommt x tatsächlich im Feld vor, benötigen wir selbst im Durchschnitt $(n+1)/2$ Vergleiche.

...geht das nicht besser?

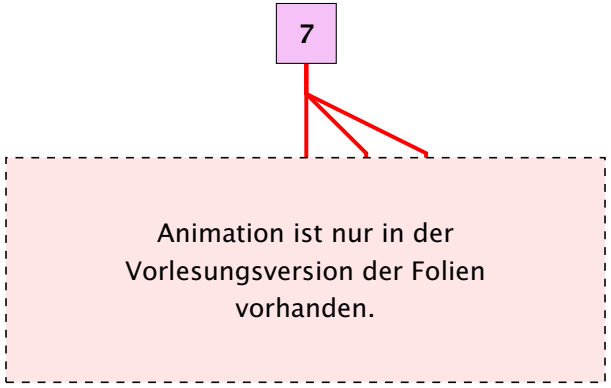
Binäre Suche

Idee:

- Sortiere das Feld.
- Vergleiche x mit dem Wert, der in der Mitte steht.
- Liegt Gleichheit vor, sind wir fertig.
- Ist x kleiner, brauchen wir nur noch links weitersuchen.
- Ist x größer, brauchen wir nur noch rechts weiter suchen.

$\Longrightarrow$ binäre Suche

Beispiel



- wir benötigen nur drei Vergleiche
- hat das Feld $2^n - 1$ Elemente, benötigen wir maximal n Vergleiche

Implementierung

Idee:

Führe Hilfsfunktion

```
public static int find0(int[] a,int x,int n1,int n2)
```

ein, die im Interval [n1,n2] sucht.

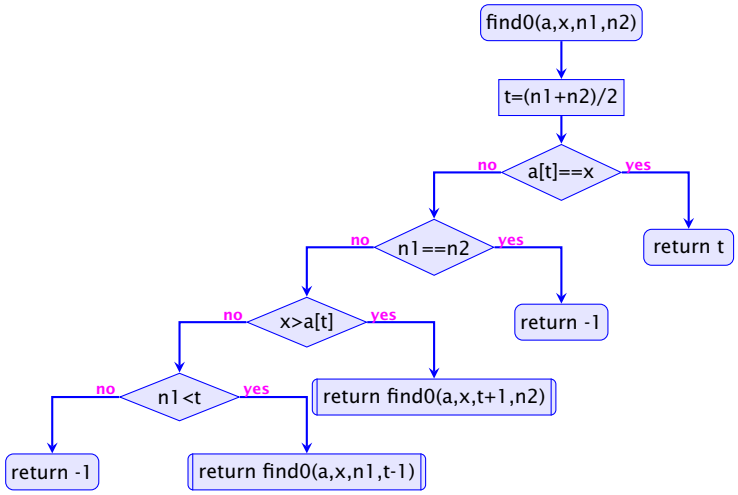
Damit:

```
public static int find(int[] a, int x) {
    return find0(a,x,0,a.length-1);
}
```


Implementierung

```
1  public static int find0(int[] a, int x, int n1, int n2) {
2      int t = (n1 + n2) / 2;
3      if (a[t] == x)
4          return t;
5      else if (n1 == n2)
6          return -1;
7      else if (x > a[t])
8          return find0(a, x, t+1, n2);
9      else if (n1 < t)
10         return find0(a, x, n1, t-1);
11     else return -1;
12 }
```


Kontrollflussdiagramm für `find0`

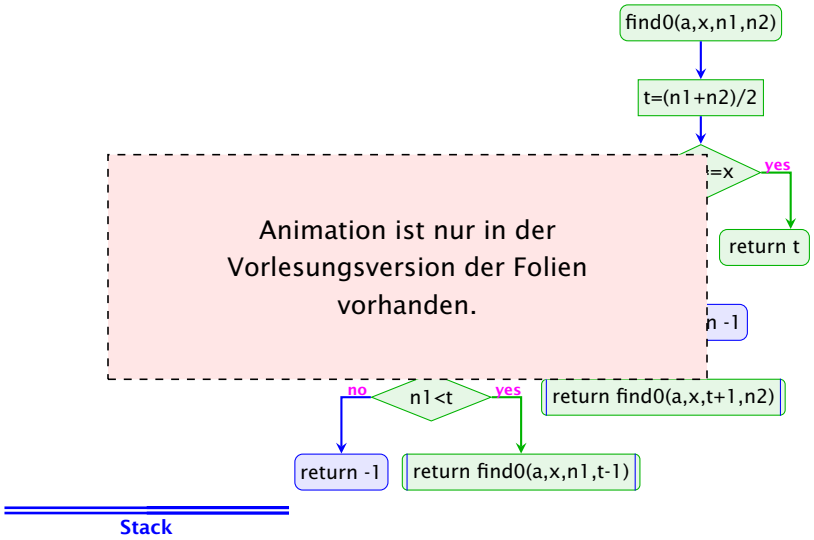


Implementierung

Erläuterungen:

- zwei der `return`-Statements enthalten einen Funktionsaufruf – deshalb die Markierungen an den entsprechenden Knoten.
- (Wir hätten stattdessen auch zwei Knoten und eine Hilfsvariable `result` einführen können)
- `find0()` ruft sich selbst auf.
- Funktionen, die sich selbst (evt. mittelbar) aufrufen, heißen rekursiv.

Ausführung



Terminierung

Um zu beweisen, dass `find0()` terminiert, beobachten wir:

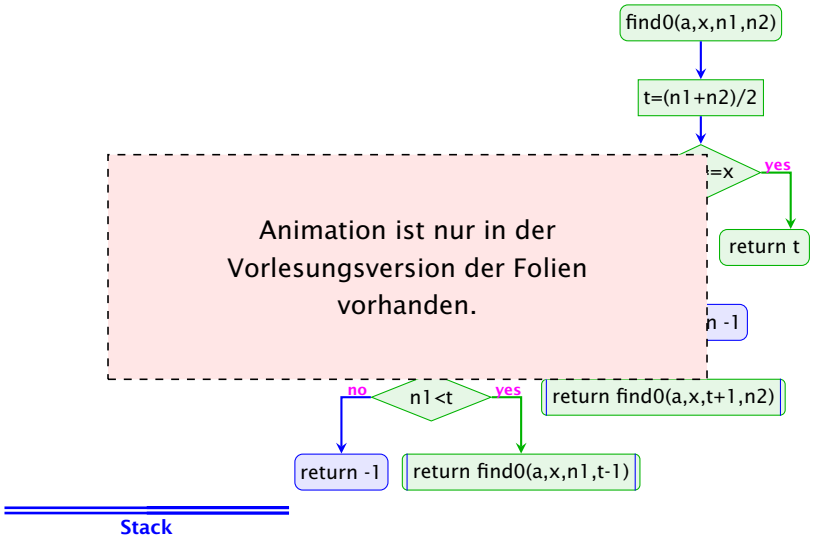
1. Wird `find0()` für ein einelementiges Intervall `[n,n]` aufgerufen, dann terminiert der Funktionsaufruf direkt.
2. wird `find0()` für ein Intervall `[n1,n2]` aufgerufen mit mehr als einem Element, dann terminiert der Aufruf entweder direkt (weil `x` gefunden wurde), oder `find0()` wird mit einem Intervall aufgerufen, das echt in `[n1,n2]` enthalten ist, genauer: sogar maximal die Hälfte der Elemente von `[n1,n2]` enthält.

Ähnliche Beweistechnik wird auch für andere rekursive Funktionen verwendet.

Beobachtung

- Das Ergebnis eines Aufrufs von `find0()` liefert direkt das Ergebnis auch für die aufrufende Funktion!
- Solche Rekursion heißt End- oder Tail-Rekursion.
- End-Rekursion kann auch ohne Aufrufkeller implementiert werden...
- **Idee:** lege den neuen Aufruf von `find0()` nicht oben auf den Stapel drauf, sondern ersetze den bereits dort liegenden Aufruf!

Verbesserte Ausführung

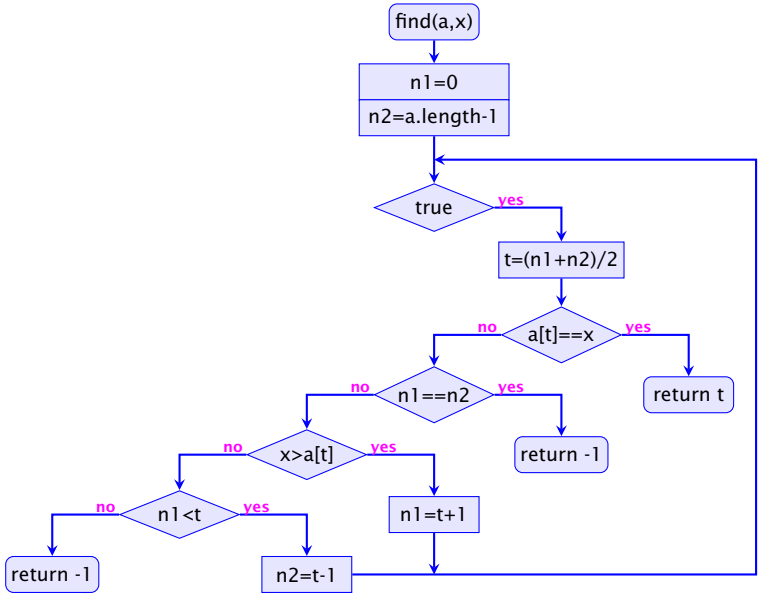


Endrekursion

Endrekursion kann durch Iteration ersetzt werden...

```
1  public static int find(int[] a, int x) {
2      int n1 = 0;
3      int n2 = a.length-1;
4      while (true) {
5          int t = (n2 + n1) / 2;
6          if (x == a[t]) return t;
7          else if (n1 == n2) return -1;
8          else if (x > a[t]) n1 = t+1;
9          else if (n1 < t) n2 = t-1;
10         else return -1;
11     } // end of while
12 } // end of find
```

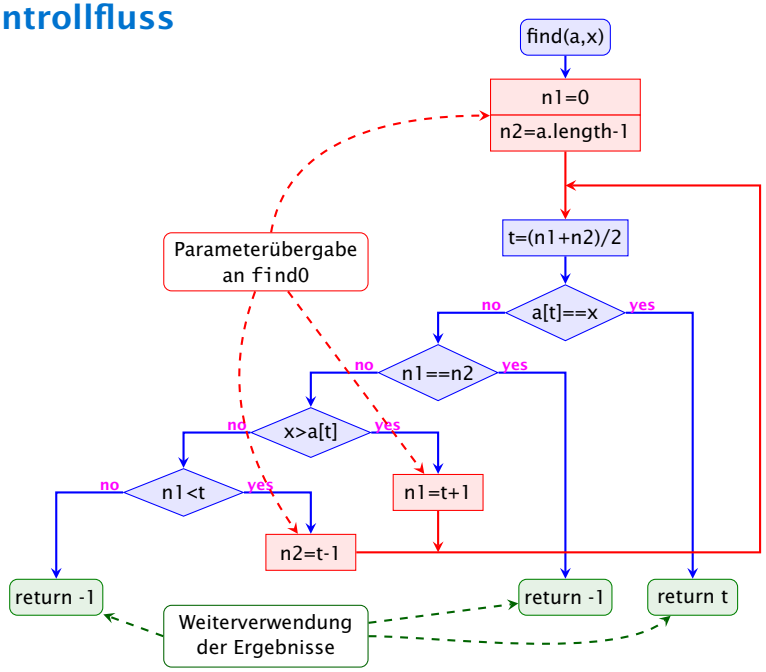

Kontrollfluss



Verlassen von Schleifen

- Die Schleife wird hier alleine durch die `return`-Anweisungen verlassen.
- Offenbar machen Schleifen mit mehreren Ausgängen Sinn.
- Um eine Schleife zu verlassen, ohne gleich ans Ende der Funktion zu springen, kann man das `break`-Statement benutzen.
- Der Aufruf der endrekursiven Funktion wird ersetzt durch:
  1. Code zur Parameter-Übergabe;
  2. einen Sprung an den Anfang des Rumpfs.

Kontrollfluss

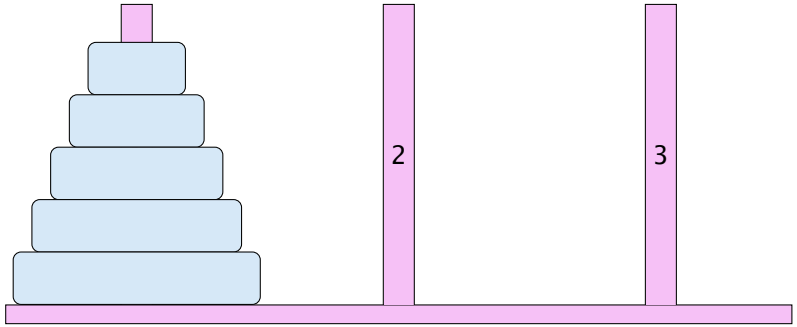


Rekursion

Bemerkung

- Jede Rekursion läßt sich beseitigen, indem man den Aufruf-Keller **explizit** verwaltet.
- Nur im Falle von Endrekursion kann man auf den Keller verzichten.
- Rekursion ist trotzdem nützlich, weil rekursive Programme oft leichter zu verstehen sind als äquivalente Programme ohne Rekursion...

9 Türme von Hanoi



- Bewege Stapel von links nach rechts.
- In jedem Zug darf genau ein Ring bewegt werden.
- Es darf nie ein größerer auf einen kleineren Ring gelegt werden.

9 Türme von Hanoi

Idee

- Für Turm der Höhe $h = 0$ ist das Problem trivial.
- Falls $h > 0$ zerlegen wir das Problem in drei Teilprobleme:
  1. Versetze oberen $h - 1$ Ringe auf freien Platz
  2. Bewege die unterste Scheibe zum Ziel
  3. Versetze die zwischengelagerten Ringe zum Ziel
- Versetzen eines Turms der Höhe $h > 0$ erfordert also zweimaliges Versetzen eines Turms der Höhe $h - 1$.

Es gibt keine andere Möglichkeit!!!

Implementierung

```
1 public static void move(int h, byte a, byte b) {
2     if (h > 0) {
3         byte c = free(a,b);
4         move(h-1,a,c);
5         write("move "+a+" to "+b+"\n");
6         move(h-1,c,b);
7     }
8 }
```

...bleibt die Ermittlung des freien Rings

Beobachtung

Offenbar hängt das Ergebnis nur von der Summe der beiden Argumente ab...

	0	1	2
0		2	1
1	2		0
2	1	0	

free(x,y)

	0	1	2
0		1	2
1	1		3
2	2	3	

sum(x,y)

Implementierung

Um solche Tabellen leicht implementieren zu können stellt Java das switch-statement zur Verfügung:

```
1 public static byte free(byte a, byte b) {
2     switch (a + b) {
3         case 1:   return  2;
4         case 2:   return  1;
5         case 3:   return  0;
6         default:  return -1;
7     }
8 }
```


Allgemeines Switch-Statement

```
switch (expr) {
    case const_0: (ss_0)? (break;)?
    case const_1: (ss_1)? (break;)?
                  ...
    case const_{k-1}: (ss_{k-1})? (break;)?
   (default: ss_k)?
}
```

- expr sollte eine ganze Zahl/char, oder ein String sein.
- Die $const_i$ sind Konstanten des gleichen Typs.
- Die ss_i sind alternative Statement-Folgen.
- default ist für den Fall, dass keine der Konstanten zutrifft
- Fehlt ein break-Statement, wird mit den Statement-Folgen der nächsten Alternative fortgefahren!

Beispiel

Dies dient nur als Beispiel für die switch-Anweisung. Im vorliegenden Fall wäres es übersichtlicher für jeden Monat einen eigenen case einzuführen (d.h., kein default), und den „fall-through“ in den jeweils nächsten case zu vermeiden.

```
1  int numOfDays;
2  boolean schaltjahr = true;
3  switch (monat) {
4      case "April":
5      case "Juni":
6      case "September":
7      case "November":
8          numOfDays = 30;
9          break;
10     case "Februar":
11         if (schaltjahr)
12             numOfDays = 29;
13         else
14             numOfDays = 28;
15         break;
16     default:
17         numOfDays = 31;
18 }
```

monat darf nicht null sein; man kann nicht mithilfe eines switch-statetments gegen null testen.

Der Bedingungsoperator

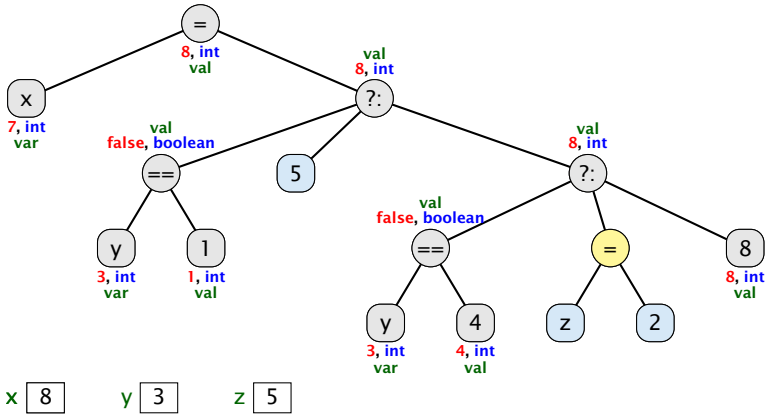
Eine Alternative zu einem switch ist der Bedingungsoperator:

condition ? expr1 : expr2

Der Bedingungsoperator

symbol	*name*	*types*	*L/R*	*level*
? :	Bedingungsoperator	boolean, 2*typ	rechts	14

Beispiel: x = y == 1 ? 5 : y == 4 ? z = 2 : 8



Beispiel

String ist ein Referenzdatentyp. Ein Vergleich monat == "Januar" vergleicht nur die Referenzen, der Strings monat und "Januar". Die sind im allgemeinen unterschiedlich, auch wenn monat und "Januar" den gleichen Inhalt haben.

```
numOfDays =
        "Januar".equals(monat) ? 31 :
       "Februar".equals(monat) ? (schaltjahr ? 29 : 28) :
         "Maerz".equals(monat) ? 31 :
         "April".equals(monat) ? 30 :
           "Mai".equals(monat) ? 31 :
          "Juni".equals(monat) ? 30 :
          "Juli".equals(monat) ? 31 :
        "August".equals(monat) ? 31 :
     "September".equals(monat) ? 30 :
       "Oktober".equals(monat) ? 31 :
      "November".equals(monat) ? 30 :
      "Dezember".equals(monat) ? 31 :
                                 -1 ;
```


Implementierung

Für unseren Fall geht das viel einfacher:

```
1 public static byte free(byte a, byte b) {
2     return (byte) (3-(a+b));
3 }
```

9 Türme von Hanoi

Bemerkungen:

- `move()` ist rekursiv, aber nicht end-rekursiv.
- Sei $N(h)$ die Anzahl der ausgegebenen Moves für einen Turm der Höhe $h \geq 0$. Dann ist

$$N(h) = \begin{cases} 0 & \text{für } h = 0 \\ 1 + 2N(h-1) & \text{andernfalls} \end{cases}$$

- Folglich ist $N(h) = 2^h - 1$.
- Bei genauerer Analyse des Problems lässt sich auch ein nicht ganz so einfacher nicht-rekursiver Algorithmus finden.

  Hinweis: Offenbar rückt die kleinste Scheibe in jedem zweiten Schritt eine Position weiter...