Programmierung
Gehalten von Prof. Dr. Jürgen Giesl
Vorlesungsmitschrift

Marian Van de veire

Diese Mitschrift erhebt keinen Anspruch auf Richtigkeit oder Vollständigkeit.

10. April 2006

Es fehlen noch 3 Vorlesungen. Diese werden noch im Laufe des Semesters ergänzt

Inhaltsverzeichnis

Teil I.

Imperative und objektorientierte Programmierung

1. Grundelemente der Programmierung

1.1. Erste Schritte

Lösungsbeschreibung : *Algorithmus*
Konkrete Formulierung: *Programm*
Programm mit zugehöriger Dokumentation: *Software*
Apparatur des Rechensystems: *Hardware*

1.1.1. Algorithmus

Determinismus, Determiniertheit und Terminierung werden nicht immer verlangt.

Kennzeichen eines guten Algorithmus:

1. Allgemeinheit: Löst nicht nur ein Problem, sondern eine Klasse von Problemen;
2. Änderbarkeit: Leicht an veränderte Aufgabenstellung anpassbar;
3. Effizienz: Möglich wenige Rechenschritte;
4. Robustheit: wohldefiniertes Verhalten auf unzulässige Einheiten.

1.1.2. Programmierung Definition:

Beispiel für eine Sprache: N ohne 0

Syntax: Alle Ziffern, die nicht mit 0 beginnen;
Semantik: Wert dieser Zahl: Wert der letzten Ziffer $+10\times$ Wert dieser Zahl, ohne letzte Ziffer.

Beispiel 367: Syntaktisch Korrekt.
Semantik: $\text{Wert}(367) = 7 + 10 * \text{Wert}(36) = 7 + 10(6 + 10\ \text{Wert}(3)) = 367$

Beispiel 007: Nicht syntaktisch korrekt.

1.1.3. Alpabet f. Sprache

- Beispiele für Alphabete:
  - lateinisches Alphabet: {a,b,...z};

  - ASCII-CODE *American Standart Code for Information Interchange*: 128 Zeichen;
    A1 ={0,1};
    A2 = {(,) ,+ ,- ,* ,/ ,a}.

- Beispiele für Worte:
  - $A1^* = \{\epsilon, 0, 1, 01, 10, 00110, ...\}$;
  - $A2^* = \{\epsilon, (), (+a),) + -(, ...\}$.

- Beispiele für Sprachen:
  - L= $\{\epsilon, 1, 10, 11, 100, ...\}$, L$\subseteq A1^*$;
  - EXPR= $\{\epsilon, (a), ((a)), (a + a), ((a + a) - a), ...\}, EXPR \subseteq A2^*$.
  - Hier: Festlegung der Programmsprache durch Angabe einer geeigneten Grammatik.

1.1.4. Grammatik informell

Beispiel:
Startsymbol = Satz

Satz → Subj. Präd. Obj.
→ Art. Attr. Subst. Präd. Obj.
→ $\underbrace{der}_{terminalsymbol}$ Attr. Subst. Präd. Obj.
→ ...

1.1.5. Grammatik Definition

Grammatik (N,T,P,S)

N: Nicht-Terminalsymbol;
T: Terminalsymbol;
P: Produktionen / Regeln;
S: Startsymbol;
V: Vokabular (N ∪ T).

Regel: $x \to y \in$ P
uxv→ uyv
Falls x ein Nicht-Terminalsymbol ist, dann ist die Grammatik *KONTEXTFREI*

.

Beispiel:
L(G) $\subseteq T^* = \{a, b, c, d\}^*$

A $\rightarrow$ aBbc $\rightarrow$ dc
$\rightarrow$ aaBbbc $= a^2Bb^2c$ $\rightarrow$ adbc
$\rightarrow a^2Bb^3c$ $\rightarrow a^2db^2c$

L(G)=$\{a^ndb^n|n \geq 0\}$

Nun versuchen wir das Wort L(G) mit kontextfreier Grammatik zu erzeugen:

$A \rightarrow Bc$ B soll a^ndb^nc erzeugen
$B \rightarrow d$
$B \rightarrow aBb$

1.1.6. Beispiel in EBNF

Statz = Subj. Präd. Obj.
Sub. = Art. Attr. Sub.
Art. = [("der"|"die"|"das")]
Attr. = {Adj.}

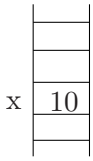
( ) = muss einmal vorkommen;
{ } = darf so oft vorkommen, wie man möchte (auch Null mal);
[ ] = kann kommen, oder auch nicht;
| = oder.

1.2. Einfache Datentypen

Jede *Variable*, die deklariert wird, besitz einen *Datentyp*.

{...
int x;

legt einen Speicherplatz x für interger Zahlen an.



1.2.1. Ganze Zahlen (byte, short, int, long)

Der Wertebereich liegt zwischen $\{-2^{8n-1}, ..., 2^{8n-1} - 1\}$

byte:	n=1
short:	n=2
int:	n=4
long:	n=8

Intern: Zahldarstellung im *Dualsystem*

Beispiel: 3 Binärstellen

3	=	011
2	=	010
1	=	001
0	=	000

Darstellung negativer Zahlen:

1. *Einerkomplement* : invertierte Binärzahlen

0	=	111
-1	=	110
-2	=	101
-3	=	100

   Nachteil des Einerkomplements: es gibt 2 veschiedene Darstellungen für die Null.

2. *Zweierkomplement*
   Bei m Binärzahlen, stelle $-z$ als $2^m - z$ dar.

Wenn m = 3:

-1: $2^3 - 1 = 7$
-2: $2^3 - 2 = 6$
-3: $2^3 - 3 = 5$

Darstellung negativer Zahlen im Zweierkomplement = Darstellung im Einerkomplement. Man kann mit m Binärstellen alle Zahlen $\{-2^{m-1}, ..., 2^{m-1} - 1\}$, im Zweierkomplement darstellen.

byte:	$\{-2^7, ..., 2^7 - 1\}$,	m	=	8,	1 byte	=	8 bit
short:	$\{-2^{15}, ..., 2^{15}\}$,	m	=	16,	2 byte	=	16bit
int:					4 byte	=	32bit
long:					8byte	=	64bit

Mit int$(-2^{31}, ..., 2^{31} - 1)$ $(2^{31} - 1 = 2.147.483.647)$
zB. int x = 2.147.483.647;
int y = 1;
System.out.print(x+y); ergibt -2.147.483.648 (kleinste int-Zahl)
Grund: *Zweierkomplement.*

Präfixoperatoren: -
Infixoperatoren: +, -, *, /(Ganzzahldivision), % (modulo)

zB. 7/2 = 3
1+2*3 = 7 (zuerst wird * gerechnet)
3 − 2 − 1 = (3 − 2) − 1 = 0 (Linksassoziativität).

1.2.2. Gleitkommazahlen (float, double)

$1,5$ = 1.5
-1.5
0.5 = .5
$51.34e12$ $(51,34 * 10^{12})$
$51.34e-3$ $(51,34 * 10^{-3})$

1.5d (Datentyp float)
1.5f = 1.5 (Datentyp double)

float: 32 bit
double: 64 bit

Präfixoperatoren: -
Infixoperatoren: +, -, *, / (division auf rationale Zahlen)

1./2. = 0.5
1/2 = 0 (Ganzzahldivision)

1.2.3. Wahrheitswerte (boolean)

Wertebereich: true, false.
Präfixoperatoren: ! (Negation)
Infixoperatoren: && ,||, >, <, >=, >=, ==, !=, ...

boolean x = true || false
System.out.println(x) ergibt true (Auswertung von links nach rechts)

2 > 3 (teste auf Gleichheit) false
x == 10 boolscher Ausdruck, ist true wenn x den Wert 10 hat.
x = 10 ist eine Zuweisung.
Prioritäten werden durch ( ) gesetzt.

1.2.4. Zeichen (char)

'a',...,'z','A',...,'Z',?,$,...(Zeichen stehen immer zwischen ")
'0','1',...,
$'\backslash n'$ (Steuerzeichen für newline)

Es existieren $2^{16} = 65\,536$ Zeichen : *UNICODE*.

Infixoperatoren: (==, ! =, >, >=, <, <= ...) (Vergleiche von Zeichen anhand ihres Codes)

'a' < 'b' (97<98) = true

1.2.5. Zeichenketten (Strings)

Vordefinierte, abkürzende Notationen:

1. Verknüpfung mit +
2. "...Schreibweise
   "hal"+"lo" ergibt "hallo"
3. Umwandlung van (beliebigen) Datentypen in Strings bei Verwendung von System.out.print/...println

   zB. System.out.println(5);
   Wird erst in String "5" umgewandelt.

   Bei selbst-definierten Datentypen geschieht die Umwandlung durch die Methode *toString* die der Benutzer selbst schreiben muss.

Infixoperatoren: ==, != (verhält sich nicht immer wie erwartet (später))

1.2.6. Typkonversion

Jedes Objekt hat einen Typ. Java ist (streng) getypt: bei jeder Operation wird angegeben, welchen Typ die Argumente und das Resultat haben. Operationen dürfen nur auf Argumente des "richtigen" Typs angewendet werden.

22	+	15	=	37
int		int		int
22	+	15.1		
int		double		bei Argumenten des falschen Types wird versucht, die Argumente in den richtigen Typ zu konvertieren.
22.0	+	15.1	=	37.1
double		double		Automatische Datentypanpassung von "eingeschränkten" zu "allgemeinen" Typen.
3	+	true		Typfehler!!!
int		bool		
'a'	+	1	=	98
char		int		int
97				

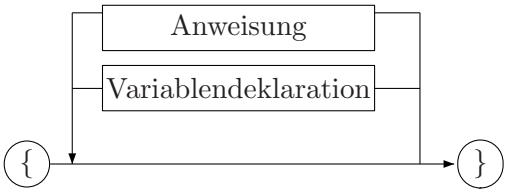
Die Datentypen können auch vom Programmierer bestimmt werden:
float x = $82.2f$ y = $8.5f$
int n = (int)(x/y) (explizite Datentypkonversion.)
n = 9

char a = (char)(int) x	a $\in R$
char b = (char)(a+1)	b ='S'

Anwendung eines Operators/Methode f, die einen Datentyp als Argument verlangt, ist möglich für alle Objekte mit Datentyp u bei denen es eine implizite Typanpassung von u nach t gibt. (Ansonsten ggf. erzwungene Typumwandlung).

1.3. Anweisungen und Kontrollstrukturen

Anweisung: Übergang von Zustand 1 zu Zustand 2;
Zustand eines Programms: Daten im Speicher (Werte der Variablen und des Programmzählers (Der Programmzähler gibt an, an welcher Stelle die Ausführung des Programms weitergehen soll.))
Datenflus: Übergabe von daten, von einer Operation an die nächste;
Kontrollfluss: Reihenfolge, in der Operationen abgearbeitet werden.



1.3.1. Methodenaufruf

1.3.2. Zuweisung

$\underbrace{Name}$ = $\underbrace{Ausdruck}$

meist Variable — Wert der in dieser Var. gespeichert werden soll

Vorausetzung: *TYPKOMPATIBILITÄT*.

Abkürzende Schreibweisen:

a =	a + 1;	stattdessen:	a +	= 1;
a =	a / 2;	stattdessen	a /	= 2;
a =	a + 1;	stattdessen	a++;	
			++a;	
a =	a - 1;	stattdessen	a–;	
			–a;	

Schlechter Programmierstil:

Verwendung von Anweisungen als Ausdrücke.

b = $\underbrace{a++}$;

Anweisung, obwohl hier ein Ausdruck stehen sollte

"a++" liefert zur gleichen Zeit den Wert von der Erhöhung zurück.

Wenn vorher a = 2 ist, dann ist hinterher b = 2, a = 3.

b = $\underbrace{++a}$;
Anweisung liefert den Wert nach der Erhöhung zurück

Wenn vorher a = 2, dann ist hinterher b = 3, a = 2.

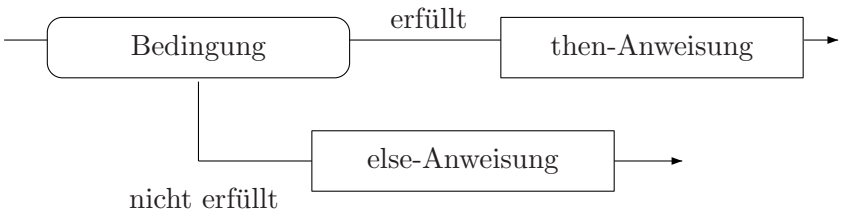
(Zur Not erlaubt) a = $\underbrace{b = 5}$; liefert 5 zurück.
Anweisung obwohl Ausdruck verlangt

Danach ist a = 5 und b = 5.
Ansonsten: Verwende keine Anweisungen als Ausdrücke!!!

1.3.3. Bedingte Anweisungen

If $\underbrace{(Bedingung)}_{boolean}$ Then-Anweisung Else-Anweisung

Semantik durch Flussdiagramm:



Switch-Anweisung: Fallunterscheidung mit mehreren Fällen.

Switch (Ausdruck){

case $Aus_1 : Anw_1$ *break*; ($Anweisung_1$ wird ausgeführt falls Ausdruck = Aus_1)

case $Aus_2 : Anw_2$ *break*;

default: Anw ⟵ Anweisung wird ausgeführt falls Ausdruck keinen der Werte $Aus_1, Aus_2, \ldots$ *hat* (default-Fall kann fehlen).

}

Wenn "break" vergessen wird, dann werden die nachfolgenden case-Ausführungen ausgeführt. $\longrightarrow$ schlechter Programmierstil.

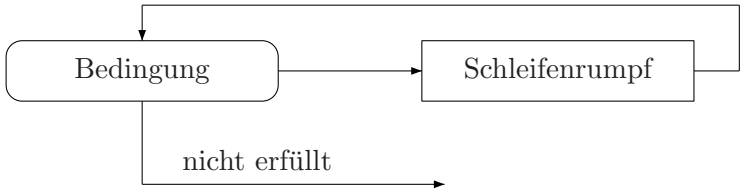
1.3.4. Schleifen

Schleifen dienen dazu, bestimmte Rechenschritte mehrmals auszuführen. (3 Schleifentypen sind "gleichmächtig").

while-Schleife:

while $\underbrace{(Bedingung)}_{boolean}$ Schleifenrumpf

Semantik:



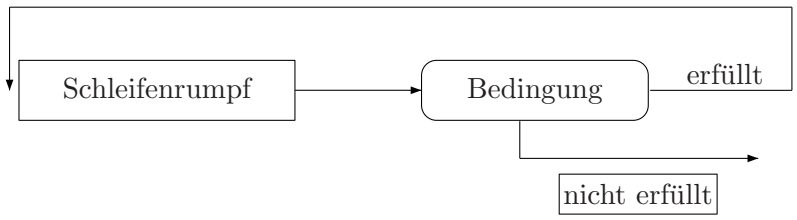
Idee von "Prim". (berechnet Primzahlen, falls n $\geq$ 2)
Falls n keine Primzahl ist, dann hat sie einen Teiler mit
$\underbrace{2}_{teiler(amAnfang)} \leq$ n $\leq \sqrt{n}$.

Gefahr bei Schleifen: *Nicht-Terinierung*

Falls man "else teiler++ ;" weglässt, dann terminitert die Schleife bei ungeraden n nicht mehr!

do-Schleife:

Hier wird der Schleifenrumpf immer mindestens 1 mal ausgeführt.
do Schleifenrumpf while (Bedingung)
Semantik:

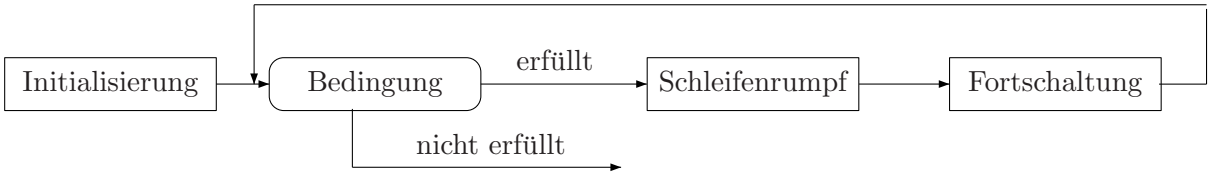


Idee von "Wurzel". Berechne $\sqrt{x}$ falls x $\geq$ 0.
Intervallschachtelung [0, x]
Stoppe wenn oG - uG $\leq \epsilon$ (10^{-3})
m ist Mittelwert des aktuellen Intervalls.
Falls m $\times$ m $>$ x, dann [uG,m]. Sonst [m, oG].

for-Schleife:

for(Initialisierung; Bedingung; Fortschaltung) Schleifenrumpf

Semantik:



Beispiel:

for(int i = 2, j = 10;i <= 5; i++, j–){. . . } i und j verwechselbar

Programmierstil: Fortschaltung sollte nur Schleifenvariable erhöhen/erniedrigen. Sonstige wiederholt ausführende Anweisungen $\longrightarrow$ Schleifenrumpf.

1.3.5. Sprunganweisung

Anweisungen können mit Namen versehen werden. (Sprungadressen)

break:

break; Springe aus momentaner Anweisung heraus (bei geschachtelten Anweisungen aus der innersten).

break Name; springe aus Anweisung "Name" heraus.

continue (nur in Schleifen): Springe zur nächsten Überprüfung der Bedingung.

!!!Sprünge nur Zurückhaltend einsetzen.!!!
(Dijkstre:"go to considered hornful").

1.4. Verifikation

Zeige, dass das Programm seine Spezifikation erfüllt.

mathematische Sprache	$\downarrow$	informell
graphisch	$\downarrow$	
logisch	$\downarrow$	formell

n = 4 i = 4
res = 1
$\downarrow$
res = 4
i = 3
$\downarrow$
res = 4 × 3
i = 2
$\downarrow$
res = 4×3×2 $\rightarrow$ =4! =24
i = 1

1.4.1. Spezifikation (zur partiellen Korrektheit)

$< \varphi > \underbrace{P}_{Programm} < \psi >$ bedeutet:

Wenn vor Ausführung von P die Vorbedingung φ gilt und wenn Ausführung P terminiert, dann gilt hinterher die Nachbedingung ψ.

Beispiel:
$< true > \underbrace{P}_{\text{Fakultät = Programm}} < res = n! >$

1.4.2. Hoare-Kalkül: (Tony Hoare)

7 Regeln zur Herleitung solcher Korrektheitsaussagen.

Vorteil:

- teilweise automatisierbar
- Verifikation überprüfbar
- Rahmen/Anleitung zur Verifikation.

1.4.3. Schreibweise der Regeln:

$\frac{Formel_1,\ldots,Formel_n}{Formel}$ bedeutet,

wenn $Formel_1, \ldots, Formel_n$ wahr ist, dann ist auch Formel wahr.
Aus $Formel_1, \ldots, Formel_n$ kann man Formeln herleiten.

$$< \underbrace{5 = 5}_{\varphi[x/t]} > x = 5; < \underbrace{x = 5}_{\varphi} >$$

$$< \underbrace{y = 5}_{\varphi[x/t]} > \underbrace{x = 5}_{t}; < \underbrace{y = x}_{\varphi} >$$

Fakultät Beispiel: $< n = n >< i = n >; < i = n >$

Schreibweise:
Füge Zusicherungen $< \cdots >$ in das Programm ein, wobei dies nur dann geschehen darf, wenn man von der obersten Zusicherung zur nächsten Zusicherung durch Anwendung einer H-Kalkül-Regel kommt.

1.4.4. Konsequenzregel 1:

Die gewünschte Vorbedingung ist oft nicht direkt die, die von der Zuweisungsregel velangt wird.

$\Longrightarrow$ Verschärfung der Vorbedingung möglich (Konzequenzregel 1)

$< true > x = 5; < x = 5 >$, denn:
$< 5 = 5 > x = 5; < x = 5 >$
true $\Longrightarrow 5 = 5$ (durch Konzequenzregel).

1.4.5. Schreibweise für Zusicherungen im Programm:

- Falls 2 Zusicherungen direkt untereinanderstehen:
  untere Zusicherung folgt aus oberer
- Falls die Anweisung zwischen 2 Zusicherungen steht:
  man kommt von der oberen Zusicherung zur unteren Zusicherung durch H-Kalkül-Regel.

Beipiel:

$< x > 1 >$ $(x > 1 \Longrightarrow x - 1 > 0 Konzequenzregel)$
$< x - 1 > 0 >$
$x = x - 1$
$< x > 0 >$ (Zuweisungsregel aus $x - 1 > 0$)

1.4.6. Konsequenzregel 2:

Abschwächung der Nachbedingung

$$\frac{< true > x = 5; \qquad x = 5 \Longrightarrow x \geq 5}{< true > x = 5; < x \geq 5 >}$$

1.4.7. Sequenzregel:

$< true >$
(Konsequenregel)
x = 5;
$< x = 5 >$
(Konsequenzregel)
$< x.x + 5 = 31 >$
res = x . x + 6;
$< res = 32 >$

$$\frac{< true > x = 5; < x = 5 > \qquad < x = 5 > res = x.x + 6; < res = 31 >}{< true > x = 5; res = x.x + 6; < res = 31 >}$$

1.4.8. Bedingungsregel: (if)

Regel 1: if-Anweisungen ohne "else"
Regel 2: if-Anweisungen mit "else"

$< true >$
$< y = y >$
res = y ;
$< res = y >$
$if(x > y) res = x;$
$< res = max(x, y) >$

1)
wenn $< res = y \wedge \underbrace{x > y}_{B} >$
(Konsequenzregel $x > y \Longrightarrow x\ max(x, y)$)
$< x = max(x, y) >$
res = x;
$< res = max(x, y) >$

2)
$res = y \wedge \underbrace{\neg(x > y)}_{x \leq y} \Longrightarrow res = max(x, y)$

Beispiel:

$< true >$
if (x < 0) res = -x;
else res = x;
$< res = |x| >$ dann:

1)
$< true 1 x < 0 >$ (Konsequenzr.)
$< -x = |x| >$
$res = -x$
$< res = |x| >$

2)
$< true > \wedge \underbrace{\neg x < 0}_{x \geq 0}$
$< res = |x| >$
$res = x;$
$< res = |x| >$

1.4.9. Schleifenregel

nur für "while". (ginge auch für "do" und "for").

Suche Schleifeninvariante φ: wenn φ vor Ausführung des Schleifenrumpfes gilt, dann glit φ auch nach Ausführung des Schleifenrumpfes.

Fakultät:

res = 1; vorφ
$< i! * res = n! >$
while(i>1){
$< i! * res = n! \wedge n > 1 >$
$< ((i-1)! res * i) n! \wedge n > 1 >$
res = res * i;
$< (i-1)! res = n! \wedge i > 1 >$
i = i - 1;
$< i! * res = n! >$
}
$< res = n! >$ nach φ
}

Gesucht: Schleifenvariante φ mit:

- $< \varphi \wedge i > 1 >$
  $res = res * i;$
  $i = i - 1;$
  $< \varphi >$ φ ist Schleifeninvariante.
- $i = n \wedge re = 1 \Longrightarrow \varphi$ Schleifenvariante folgt aus Vorbedingung
- $\varphi \wedge \underbrace{\neg i > 1}_{i \leq 1} \Longrightarrow res = n!$ Aus Schleifenvariante folgt die Nachbedingung.

Das Vorgehen zum Finden von φ:
Teste die Schleife mit konkreten Werten und untersuche den Zusammenhang zwischen allen vorkommenden Variablen.

i	res	n	
4	1	4	⟵ vor Ausführen der Schleife
3	4	4	⟵ nach 1. Ausführung des Schleifenrumpfs
2	4×3	4	⟵ nach 2. Ausf.
1	$4 \times 3 \times 2$	4	⟵ nach 3.Ausf.

Wie hängen i, res, n zusammen?

$i! * \underbrace{res = n!}_{Nachbedingung}$ ist die Gesucht Schleiveninvariante φ

$< \varphi > P < \psi >$

Schleifeninvariante φ: Wenn $\varphi \wedge B$ am Anfang der Schleifenrumpfes gilt, dann gilt φ auch am Ende des Schleifenrumpfes.

Schreibweise:

- 2 Zuweisungen übereinander: aus oberer folgt unterer.
- Anweisungen zwischen 2 Zusicherungen: Hoare-Kalkül-Regel.

Überprüfe solche Zusicherungen in Java mit assert.
Compilierung: javac -source1.5. . . .java
Ausführung: java -enableassertions. . .

TOTALE KORREKTHEIT:

- partiellle Korrektheit (H-Kalkül)
- Terminierung

1.4.10. Terminierung

Gehe davon aus, dass alle (anderen) Methoden teminieren. Dann kann Nicht-Terminierung nur entstehen, wenn eine Schleife unendlich oft durchlaufen wird. $\Longrightarrow$ Schleifenvariante (ist der Ausdruck, der bei jedem Schleifenlauf kleiner wird).

Beispiel: Variante i

- $B \Longrightarrow V \geq 0$
  $i > 1 \Longrightarrow i \geq 0$

- $< i = m \wedge i > 1 >$
  $< i - 1 < m >$
  $//c\ res = res * i;$
  $< i - 1 < m >$
  $//ci = i - 1;$
  $< i < m >$

1.4.11. Verifikation der Addition

$< a \geq 0 >$
$< a = 0 \wedge b = b \wedge a \geq 0 >$
x = a;
$< x = a \wedge b = b \wedge x \geq 0 >$
res = b;
$< x = a \wedge res = b \wedge x \geq O >$
$< x + res = a + b \wedge x \geq 0 >$
while $(x > 0)\{$
$< x + res = a + b \wedge x \geq 0 \wedge x > 0 >$
$< x - 1 + res + 1 = a + b \wedge x - 1 \geq 0 >$
x = x - 1;
$< x + res + 1 = a + b \wedge x \geq 0 >$
res = res + 1;
$< x + res = a + b \wedge x \geq 0 >$
}
$< x + res = a + b \wedge x \geq 0 \wedge \neg x > 0 >$
$< res = a + b >$

Suche der Schleifeninvariante:

x	res	a	b
3	2	3	2
2	3	3	2
1	4	3	2
0	5	3	2

Schleifeninvariante: $x + res = a + b \wedge x \geq 0$

1.4.12. Verifikation der Substraktion

$< x \geq y >$
$< y = y \wedge 0 = 0 \wedge x \geq y >$
z = y;
$< z = y \wedge 0 = 0 \wedge x \geq z >$
res = 0;
$< z = y \wedge res = 0 \wedge x \geq z >$
$< res = z - y \wedge x \geq z >$
while$(x > z)\{$
$< res = z - y \wedge x \geq z \wedge x > z >$
$< res + 1 = z + 1 - y \wedge x \geq z + 1 >$
z = z + 1;
$< res + 1 = z - y \wedge x \geq z >$
res = res + 1;
$< res = z - y \wedge x \geq z >$
}
$< res = z - y \wedge x \geq z \wedge \neg x > z >$
$< res = x - y >$

Suche die Schleifeninvariante:

x	y	z	res
5	2	3	0
5	2	3	1
5	2	4	2
5	2	5	3

Schleifeninvariante ist: $res = z - y \wedge x \geq z$

Terminierung

Variante ist: x - z

- $x > z \Longrightarrow x - z \geq 0$
- $< x - z = m \wedge x > z >$
  2)
  $< x - (z + 1) < m >$
  z = z + 1;
  1)
  $< x - z < m >$

```
res = res + 1;
```

1.5. Reihungen

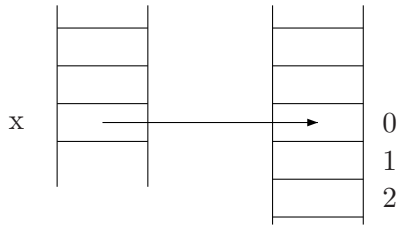
ARRAYS: Speichere in Variablen eine Sammlung von Werten.

Beispiel:
folge: Array mit 4 int-Werten (1 dimensionales Array)
zugriff: folge[0], ... folge[3]
bestand: 2 dimensionales Array.
zugriff: bestand[i][j]

Beispiel: Anzahl der Artikel 0-2 an Ort 3:(ergibt 8)
int summe = 0;
for(int i = 0, i <= 2; i ++)
summe + = Bestand[i][3]

- Array-Variable fasst eine Vielzahl von Objekten eines Datentyps (hier: int) zusammen.
- Einzelne Elemente werden durch Indizes unterschieden
- Deklaration von Array-Variablen:
  float[ ]x; deklariert eine Variable x für 1-dim Array in dem float Zahlen stehen.
  int[ ][ ]x; deklariert 2-dim int Array.
- Bei der Deklarartion von Array-Variablen wird noch kein Speicherplatz für die Elemente des Arrays zugewiesen (Inhalt der Array-Variablen ist "null" steht für "Zeiger ins Leere").
- Erzeugung des Speicherplatzes für die Array-Elemente mit "*new*".

  Bei 2-dim Arrays:
  int[ ][ ]x;
  x= newint[3][ ];



x[0] = newint[2];
Matrizen mit 3 Zeilen aber beliebig viele Spalten (1.Zeile könnte
2 Spalten haben, 2. könnte 3 Spalten haben).

auch x = newint[3][2];
$\hat{=}$ Matrix mit 3 Zeilen + 2 Spalten.

Keller (Stack): Speicherbereich der den Variablen des Programms (zB. x) entspricht.

Halde (heap): Speicherbereich, in den Verweise zeigen.

Verweise (Zeiger, Pointer) existieren in praktisch allen imperativen Programmiersprachen. Meist kann man explizit Zeigerstrukturen erzeugen und Zeigerstrukturen nachgehen (*Dereferenzierung*).
Es existieren auch Sprachen, in denen man die Speicheradressen in der Halde explizit manipulieren kann (zB. C).
⇒ in Java werden Zeigerstrukturen implizit aufgebaut (bei jedem nicht-primitiven Datentyp) und Dereferenzierung erfolgt automatisch.

1.5.1. Vorteil von Arrays

Wahlfreien Zugriff, (d.h. Lesen und Schreiben jedes Array-Elements damit gleich lang).

Grund: Adresse des Elements kann sofort aus Basisadresse des 1. Elements und aus der Grösse der Datenstruktur-Objekte berechnet werden.

Unterschiedliche Zuweisung bei $\underbrace{Wertvariablen}_{\text{Werte in Var.}}$ / $\underbrace{Referenzvariablen}_{\text{Verweise in Var.}}$

y = x: Schreibe an Stelle y den Inhalt der Stelle x.

1.5.2. Seiteneffekt

Auswirkung des Schreibzugriffs über eine Referenzvariable y auf ein Objekt, das auch über eine andere Referenzvariable x erreichbar ist.
⇒ y = x erzeugt keine Kopie des Objekts sondern nur 2 Verweise auf dasselbe Objekt.
Um wirklich Kopie zu erzeugen, muss man elementweise kopieren.

x == y: x und y zeigen auf das gleiche Objekt. Wenn x und y auf verschiedene Arrays zeigen, die dieselben Elemente enthalten, dann gilt nicht x == y. (Sonst Elementweise auf Gleichheit testen).

Array ist nicht mehr benutzbar, wenn keine Variable mehr darauf verweist. Solche Daten, in der Halde müssen wieder gelöst werden.

In Java: automatisch durch "*Garbage Collection*". Dieser löscht nicht

- nur nicht mehr benötigte Daten;

- verschiebt Daten, so dass möglich grosse zusammenhängende leere Speicherbereiche entstehen (relativ zeitaufwendig).

```
int[ ]a;          int[ ][ ]b;
a = newint[4];    b = newint[3][2];
a[0] = 8;         b[0][1] = 8
a[1] = 7;
a[2] = 6;
a[3] = 5;
```

Alternative Schreibweise zur Initialisierung von Arrays:
int[ ]a = {8,7,6,5};
int[ ][ ]b = {{9,8},{10,11},{12,13}}.
int[ ][ ]c = {{9,8},{10},{ }}.
Diese Schreibweise ist nur in der Initialisierung erlaubt.

1.5.3. Berechnung der Länge eines Arrays b

1-dim Array:

b.length = 3
a.length = 4
c.length = 3

2-dim Array:

b[0]length = 2.

1.5.4. Beispiel: Palindrom

Wort, das von vorne gelesen = von hinten gelesen ist.
Beispiel:
rentner ist ein Palindrom
rentier ist kein Palindrom.

Die main-Methode: bekommt als Eingabe einen Array von Strings.
javaPalindrom $string_0, string_1, \ldots, string_n$

⇒ arcs wird ein String-Array der Länge n + 1 zugewiesen das an
Index 0 $string_0$
Index 1 $string_1 \ldots$ enthält.

. . . . . : greife auf Eigenschaft eines Objekts zu ($\underbrace{a.length}_{Eigenschaft/Attribut}$)

$\underbrace{args[0]}_{String}$. $\underbrace{toCharArray()}_{EigenschaftMethode(wg())}$: berechne das char-Array das dem String entspricht.

Dokumentation alles Java-Bibliotheksfunktionen:
API (Application Programmer Interface)
$http://java.sun.com/j2se/1.5.0/docs/api$

Wort = $\{\underbrace{r}_{0}, \underbrace{e}_{1}, \underbrace{n}_{2}, \underbrace{t}_{3}, \underbrace{n}_{4}, \underbrace{e}_{5}, \underbrace{r}_{6}\}$

i laüft von 0 bis 3.

1.5.5. Beispiel: Sortierprogramm

Sortierprogramm mit hoher Zeit, kleiner Platzbedarf.

$\alpha\{4, 2, 5, 1\}$
{2,4,5,1}
{1,4,5,2}⟵ Array am Ende des ersten aüsseren Schleifendurchlauf.
{1,2,5,4}⟵ Array am Ende des zweiten aüsseresn Schleifendurchlauf.
{1,2,4,5}

Vertauschung von
a[i] = a[j];
a[j] = a[i];
ist nicht korrekt!!!!! Vertauscht nicht (ändert den Eintrag nicht) sondern schreibt den Wert a[j] an a[i].

2. Objekte, Klassen und Methoden

2.1. Grundzüger der Objektorienierung

ARRAYS:

- Vorteil:
  Effiziente Möglichkeit der Darstellung und Sammlungen von Elementen. Schneller Lese- und Schreibzugriff, gleichschnell für alle Array-Elemente.
- Nachteil:
  - Elementanzahl liegt ab der Erzeugung fest. (keine dynamische Datenstruktur).
  - rüken nur eine einzige Eigenschaft eines zu modellierenden Gegenstands aus.

[0]	[1]
laenge = 2,5	laenge = 3,1
breite = 1,2	breite = 1,4
Strichst. = 1	Strichst. = 3

    Daten eines Rechtecks sind auf 3 Arrays verteilt. ⇒ will man Rechteck null auf Rechteck 1 setzten so benötigt man 3 Anweisungen.
  - Arrays sind reine Datenstrukturen. Programmstücke zur Bearbeitung von Array-Elementen stehen ausserhalb des Arays.

OBJEKTORIENTIERUNG:

Gehe von den zu modellierenden Objekten aus (zB. Rechteck) und fasse alle ihre *Eigenschaften* (Methoden und Attribute) zusammen.

Klasse: beschreibt die Eigenschaft, die alle diese Objekte haben (≙Datentyp)
⇒ neuer Datentyp Rechteck;

Attribute: länge, breite, strichstärke. (Variablen die in der Klasse deklariert werden).

Methode: flaeche().

IDEE: Finde herraus welche Objekte man benötigt, arbeite Gemeinsamkeiten der Objekte heraus, definiere entsprechende Klasse (Datentyp, charakterisiert gleichartige Objekte).

2.1.1. Erzeugung und Arbeiten mit Objekten

- Variablen deklarieren. Rechteck r;
- Initialisierung mit dem leeren Objekt. r = new $\underbrace{Rechteck()}$; .
  (ruft einen Konstruktor auf, der ein leeres Rechteck erzeugt
- Zugriff auf Eigenschaften des Objekts

  r.laenge = 2.0;
  r.breite = 2.5;
  r.strich = 1;
  System.out.print(r.laenge);
  System.out.print(r.flaeche()); ergibt 5.0

Vorteil:

- Alle Eigenschaften eines Objekts sind zusammen in einer Datenstruktur gespeichert.
- Um einem Rechteck ein anderes zuzuweisen benötigt man nur eine Anweisung (r=s).
- Programmteile die Eigenschaften (Methoden) berechnen stehen in der selben Struktur wie die anderen (festgelegten) Eigenschaften (Attribute).

Bei Variablendeklaration in der Klassen sind auch Initialisierungen möglich:

public class Rechteck{

double laenge = 2.0; double breite = 2.5; int strichstaerke = 1;

⇒ bei Initialisierung r = newRechteck(); würde also r.laenge = 2.0 etc... gelten.

Bei der klassischen (objektorientierten) imperativen Programmierung:

Zusammenfassung und Eigenschaften von Objekten möglich aber nur für festgelegte Eigenschaften, nicht für zu berechnende Eigenscgaften.

$\hat{=}$ Klassen mit Attribute ober ohne Methoden.
Solche Datenstrukturen bezeichnet man als *Verbunde*(Records).

Grosser Vorteil der Objektorientierung:

Man kann Klassen zueinander in Beziehung setzen. (Ober- Unterklassen, Vererbung $\Rightarrow$ später).

2.1.2. Realisierung im Speicher

- Variablen von Klassen-Datentypen enthalten nur einen Verweis/Referenz auf das eigendliche Objekt.
- Zugriff auf Objektkomponente durch $\cdot$ ($\hat{=}$ einem Verweis folgen)
- Seiteneffekte (Zugriff über r ändert das Objekt, auf das auch s zeigt).
- Gerbage Collector muss nicht mehr benötigte Objekte aus der Halde löschen.

2.2. Methoden, Unterprogramme und Parameter

2.2.1. Generelles zum Aufruf von Methoden

Methoden: Unterprogramm (parametrisierter Anwedungsblock).

Algorithmische Abstraktion: gibt diesem Anwendungsblock einem Namen ⇒ ansprechbar an verschiedenen Stellen des Programms.

$$\underbrace{Methodenkopf : gibt \quad \underbrace{Signatur}_{\text{Typen der Ein-Ausgabe}} \quad der Methode an.}$$

Schlüsselworte Rückgabetyp Methodenname(Eingabetyp$_1$ $Argument_1$) $Block$

Schlüsselworte: public, static, ...
Rückgabetyp: (void: methode liefert keine Rückgabe)

Beim Aufruf der Methode : Methodenname$\underbrace{(Arg_1, \ldots, Arg_n)}_{\text{aktuelle Parameter}}$

⇒ die formalen Parameter werden mit dem aktuellen Parametern belegt.
→ Transport von Daten von der Aufrufstelle der Methoden in die Methode hinein.

Zuweisung von aktuellen Parametern zu den formalen Parametern: *Parameterübergabe*.

Beispiel:

Aktueller Parameter: $betrag_1 + betrag_2$.
Wert: 1570,22
Formale Parameter: kapital.

Ablauf beim Methodenaufruf (in Java):

1. Berechne den Wert des aktuellen Parameters (berechne 1000 + 570,22)
2. Parameterübergabe: Weise dem formalen Parameter den Wert des aktuellen Parameters zu (kapital = 1570,22).
3. Ausführung des Methodenrumpfs.
   (formale Parameter sind Variablen, können auch verändert werden)
4. Abbruch des Methodenrumpf, bei "*return*"
   Liefere das Ergebnis an der Aufrufstelle zurück.
   (gewinn 1570,22 × 103 = 1617,3266)

   - Methoden, die das Ergebnis zurückliefern (Rückgabe ≠ void), heissen *Funktionen*.

- Methoden, ohne Rückgabe (void), heissen *Prozeduren*.
  Sinn von Prozeduren:
  - Ein-/Ausgabeprozeduren (Bsp. Drucke Array auf Bildschirm)
  - Ausnutzung von gewünschten Seiteneffekten, um auf das Ergebnis der Methodenberechnung zuzuzgreifen.

  Prozeduraufrufe: Anweisung (drucke(x))
  Funktionsaufrufe: Ausdruck (gewinn = zins($b_1 + b_2$);)

2.2.2. Parameterübergabemechanismen

- *call-by-value*:
  1. Werte aktuellen Parameter auf
  2. Kopiere diesen Wert in den formalen Parameter der Methode.
  3. Änderungen des formalen Parameters der Methode bewirken keine Änderung des aktuellen Parameters.

  Wird in Java bei primitiven Datentypen verwendet. Bsp. Wert von s ist hinterher immer noch 2·1.

- *call-by-reference*:
  Der aktuelle Parameter muss eine Variable sein.
  1. Der formale Parameter wird nur ein Verweis auf den aktuellen Parameter (keine Kopie von Werten).
  2. Jede Änderung des formalen Parameters im Methodenrupf bewirkt auch eine Änderung des aktuellen Parameters.

  Bsp: Wert von s wäre 4.6
  ⇒ existiert in vielen Programmiersprachen, aber nicht (direkt) in Java.

- *call-by-name*:
  Wie call-by-value, aber der Wert des Parameters wird beim Aufruf der Methode nicht berechnet. (d.h. P wird nicht erweitert).
  ⇒ beeinflusst Terminierungsverhalten und Effizienz.
  ⇒ später Haskell

2.2.3. Speicherverwaltung bei Methodenaufruf

Kellerspeicher: *LIFO*-Prinzip (Last in First Out).
Jeder Block entspricht einem Speicherbereich (*frame*) auf dem Keller. Dieser enthält die Werte der Objekte, die in diesem Block deklariert wurden.
Bei Eintritt in einen neuen Block wir ein neuer Speicherbereich *oben* (in Wachstumsrichtung) auf den Keller angelegt.

⇒ sucht nach Variablen immer zuerst im innersten Block, dann im nächst aüsseren etc. . .

Bei Methoden: Speicherverwaltung analog (Methodenrumpf ist ein Block (wird als innerer Block betrachtet, der bei der Aufrufstelle der Methode besetzt wird).

Namen der Variablen in der Methoden dürfen zu den Variablennamen der aufrufenden Stelle identisch sein. Allerdings kann man aus de Methodenrumpf nicht auf die Variablen der aufrufenden Stelle zugreifen.

Parameterübergabe bei Java ist *call-by-value* bei *primitiven Datentypen*.
Bei *anderen* Datentypen findet eine "Art" *call-by-reference* statt.

Beispiel: Hinterher ist s.laenge = 4.6
Bei nicht-primitiven Datentypen wie Rechteck:

- Aktuelle Parameter enthält einen Verweis auf das eigendliche Rechteckobjekt.
- Bei Methoden wird die Adresse des Rechteckobjekts in den formalen Parameter kopiert.
  ⇒ aktueller Parameter s und formaler Parameter r zeigen auf das gleiche Objekt.
- Änderungen an diesem Objekt (über formalen Parameter r) sind auch über den aktuellen Parameter sichtbar. (*Seiteneffekt*).

In Java: Inhalt des aktuellen Parameters wird in den formalen Parameter kopiert, auch dann, wenn der aktuelle Parameter als Inhalt einen Verweis hat. Simuliert teilweise call-by-reference.

Beispiel: in dem der formale Parameter r auf ein neues Rechteckobjekt verweist:

Aktuelle Parameter s ändert sich nicht. Dies entspricht nicht dem call-by-reference. Dort würden der aktuelle Parameter und der formale Parameter miteinander identifiziert werden → geht in Java nicht.

Also: Java verwendet immer call-by-value, aber da nicht-primitive Datentypen über Referenzen realisiert sind, können Methodenrümpfe Seiteneffekte auslösen. → Sinn von eingeschränktem call-by-value.

Übergabe von Werten von einer Programmstelle an eine andere Stelle sollte kontrolliert erfolgen (über Parameter von Methoden) und nicht über globale Variablen.

Beispiel:
pubic class K{

public static int x;

```
public static int f(...){
⋮
x = x + 1;
⋮
}

public static void main(...){
⋮
... = f(x);
⋮
}
}
```

SCHLECHTER PROGRAMMIERSTIL!

Besser ist: verwende x als Parameter.
Vorteil c-b-r: Kopieren vom aktuellen Parameter in den formalen Pararmeter ist effizient.
Nachteil: Fehlerquelle!

2.2.4. statische und und nicht-statische Methoden und Attribute

HIER FEHLEN DIE ERSTEN 20 min DER VORESUNG VOM 18.11 (falls jmd die haben sollte ....:-)
Bisher:

- Klassen die nur statische Methoden und Attribute enthalten. (z.B.: Klasse Sort. Aufruf von "sortiere" ausserhalb der Klasse Sort durch:

  Sort.sortiere())

  $\hat{=}$ reine Sammlung von Unterprogrammen.

- Klasse, die nur nicht-statische Methoden + Attribute enthalten.
  (z.B.: bisherige Klasse Rechteck)
  $\hat{=}$ reine Datenstruktur.

Jetzt: sowohl statische als auch nicht-statische Methoden und Attribute.

Man kann auf statische Attribute und Methodnen auch über die Objekte der Klasse zugreifen:

Rechteck.flaechenberechnung ist äquivalent zu:

r.flaechenberechnung und
s.flaechenberechnung. Bei solch einem Aufruf wird über die Klasse des Objekts (s) auf das Attribut flaechenberechnung zugegriffen:

r.laenge
~~*Rechteck.laenge*~~ ← nicht zulässig! Jedes Rechteck-Objekt hat ja verschiedene Längen.

Generell: alle Methoden, die eine Eigenschaft des Objekts ausdrücken (z.B. Lesen/Schreiben von Objekt-Attributen) sollten nicht statisch sein.

Beispiel: für eine Methode die nicht-statisch sein sollte: *toString*

ZIEL: Konvertiere ein Datenobjekt in einen String.
Bei *System.out.print(ln)(Objekt)* wird automatisch das Objekt vorher in einen String konvertiert. → Aufruf von Objekt.toString()

System.out.println(r) =
System.out.println(r.toString()).

2.2.5. Gültigkeit von Bezeichnern

Welche Bezeichner (zB. Variablennamen) gelten wo?

Grund für merfache Verwendung gleicher Bezeichner:

- mehere Entwickler arbeiten unabhängig an verschriedenen Teilen des Programms.
- Sinnvolle Wiederverwendung von Bezeichnern erhöht die Lesbarkeit.

2.2.6. Generelle Regeln

- Jeder Bezeichner muss deklariert werden. Bezeichner gehört zum innersten Block, in dem es deklariert ist.
  Beispiel: im Rumpf der main-methode der Klasse "Gültigkeit" ist "x" der formale Parameter von main und nicht die statische Variable vom Typ Dreieck.
- Deklaration gilt bis zum Ende des Blocks.
- Bezeichner kann erst nach seiner Deklaration verwendet werden.

  Ausnahme: Deklaration von Klassen und Methoden:
  sind im ganzen Block verwendbar und nicht erst ab der Stelle wo sie deklariert wurden.
- Namensgleiche Bezeichner im inneren Block überdecken Berzeichner aus aüsseren Block.
- Bezeichner im selben Block müssen unterschiedlich sein.

$$\left.\begin{array}{l}\text{int x;}\\ \text{float x;}\end{array}\right\} Verboten!!!$$

ABER:

- Verschiedenartige *Programmelemente* (Klassen, Methoden, Variablen) dürfen gleich heissen:

  zB. int f;
  public double f(...){...}
- Methoden mit verschiedenen Parameterlisten dürfen gleich heissen (*überladene Methoden*).

  public double f(int x){...}

  public double f(int x, int y){...}

  public double f(double x){...}

  ⇒ Später

Beipiel: Gültigkeit / Dreieck

- statische Variable x gilt in der ganzen Klasse "Gültigkeit".
- Aber formale Parameter x der main-Methode überdeckt die statische Variable x.
- Objektvariablen x,y,z gelten genau in dieser Klasse.
- In der Methode "setze"
  "x": formaler Parameter der Methode setze.
  "d.x": entsprechende Objektvariable des Objekts d.
- Methode.flaeche: erste "y" Objektvariable des Dreiecks aber ab der Deklaration double y... überdeckt dieses neue y die Dreiecksvariable y.

Beispiel: java Gültigkeit test

ergibt: test

Ausgabe des Dreiecks:

d.flaeche() = 1.0...

VORSICHT bei Mehrfachverwendung von Bezeichnern!!!
⇒ kann zu unverständlichkeit der Programme führen.

2.3. Datenabstraktion

Generelles Prinzip zum Entwurf von Klassen und Methoden.

Alle Methoden, die Eigenschaften des Objekts ausdrücken, sollten nicht-statisch sein (insb. Methoden die Objektattribute lesen oder schreiben).
Insbesondere: Schreiben und Lesen von Objekt-Attributen sollte nicht durch Zuweisung ausserhalb des Objekts erfolgen, sondern durch sogennante *Selektoren*.

Zugriff auf Objekte ist dann nur durch bestimmte Methoden möglich (zB. durch Selektoren).
Ausserhalb der Klasse Rechteck sollte man nicht direkt auf Rechteck-Attribute (zB. laenge) zugreifen.

Objektvariablen (laenge) sind dann durch diese Methoden (setlaenge, getlaenge) *gekapselt*.

Vorteil:

- Durch Datenkapselung ist die innere Implementierung der Rechteck-Klasse unabhängig von den Programmteilen, die die Rechteck-Klasse benutzen. Die Rechteck-Klasse muss nur sicherstellen, dass die von aussen verwendeten Methoden (set-getlaenge) richtig arbeiten. Aber ansonsten kann man die Rechteck-Klasse beliebig implementieren.
  ⇒ Modularisierung: (Die Implementierung Klasse Rechteck kann unabhängig vom Rest erfolgen).
- Programmentwurf/Entwertung von mehreren Entwicklern.
- Änderungsfreundlichkeit (Änderungen der Klasse Rechteck betreffen den Rest des Programms nicht, solange setL, getL noch dasselbe tun).

  Beipiel: Ändere die Implementierung von Rechteck.
  Statt Attribut "laenge" nimm Attribut "flaeche".
  $\Longrightarrow$ Programmteile werden verständlich und ohne Kenntnis des Gesamtprogramms.

2.3.1. Erzwinge Datenabstraktion

Verbiete den Zugriff auf solche Teile des Objekts, die von aussen nicht sichtbar sein sollen.

⇒ gib bei jedem Attribut/Methode Zugriffsspezifikationen an.
Legt fest, von wo aus man auf diese Attribute/Methoden zugreifen kann.

privat: solche Eigenschafte, die Aussen nicht sichtbar sein sollten.

public: Schnittstelle nach aussen
⇒ Stellt sicher dass die *Datenabstraktion* eingehalten wird.

Zugriffsspezifikation auf Klassenebene (in Java)
Dadurch werden die jeweiligen Attribute aller Objekte der Klasse (privat/public). In Standarteinstellung werden dort die public-Teile aller public-Klassen beschrieben.

2.3.2. Datenabstraktion

Trenne von aussen zugängliche Schnittstelle von der Implementierung.

Geheimnisprinzip (*Information Hiding*):

- Anbieter publiziert Katalog der Dienstleistungen als öffentliche Schnittstelle:

  "*Schnittstellendokumentation*"

- Kunde interessiert nur die Schnittstelle aber nicht, wie die Leistungen erbracht werden.
  zB. API der Java-Bibliotheken.

2.3.3. Abstrakter Datentyp (ADT)

- Wird durch Schnittstellendokumentation beschrieben.
- Besteht aus Daten (Attributen) und darauf ausführbaren Operatoren (Methoden).
- Nach aussen ist nur die abstrakte Schnittstellendefinition sichtbar, konkrete Realisierung bleibt verborgen.

  Vorteil:

  - besseres Verständnis
  - leichte Änderbarkeit
  - bessere Modularisierung.

Schnittstellendokumentation kann automatisch aus dem Programm erstellt werden:

"*javadoc.Klassenname*"

erzeugt Schnittstellendokumentation in HTML-Format.

2.3.4. Datenkapselung als Entwurfsprinzip

Entwerfe erst öffentliche Schnittstelle (Signatur der Methoden/Attribute die nach aussen sichtbar sein sollen) dann Implementierung.

Beispiel: Ordner
beschrifte und lies Beschriftung ⇒ Selektoren.

Programm produziert folgende Ausgabe:

Kleine Gedichte
. . . . . .
Von aussen. . . . . .
Herr Ribeck . . . . . .

Bei Einstellung der Schnittstellen mit javadoc:

/**
⋮
*/

Kommentare vor Klassen und Methoden, die in der Schnittstellendokumentation mit ausgegeben werden sollen.

Um den Eintrag zu formatieren , unterstützt javadoc weitere Schlüsselwörter:

@author: Autor der Klasse.
Aufruf mit javadoc-author-Klasse

@return: Beschreibung des Rückgabewertes einer nicht-void Methode.

@param: Beschreibung der Eingabeparameter von Methoden.
⋮
Zusätzlich sollte in den Kommentaren die Arbeitsweise der Methode beschrieben werden, falls nicht offensichtlich.

2.4. Konstruktoren

Methoden zur Erzeugung neuer Objekte: mittels "*new*" wird ein Konstruktor ausgeführt der ein neues Objekt erzeugt.

- Konstruktoren heissen genau wie die Klasse
- Kein Rückgabetyp angegeben: Ergebnis ist immer ein Verweis auf das neue erzeugte Objekt.
- Konstruktoren können die Attribute des Objekts mit bestimmten Anfangswerten belegen.
  Bsp: r.laenge = 1.0 etc
- Konstruktoren können auch Eingabeparameter haben.
  Bsp: s.laenge = 2.1 etc
- Konstruktoren können beliebige Anweisung enthalten. (aber sie sollten nur die Anweisung enthalten, die zur Erzeugung des neuen Objekts nötig sind).
- Falls man keinen Konstruktor schreibt, dann wird automatisch ein Standart-Konstruktor erzeugt.
  public Rechteck(){...}
- Es darf mehrere Konstruktoren geben. Es muss immer eindeutig bei jedem Aufruf erkennbar sein welcher Konstruktor gemeint ist. (*Überladung* von Methoden).

Syntaxdiagramm für Konstruktordeklaration:

- wie Methodendeklaration, aber kein Rückgabetyp;
- "*Name*" muss nur der Name der Klasse sein.

Syntaxdiagramme für Ausdrücke:
hinter "*new*" darf als Methode nur ein Konstruktor aufgerufen werden.

Überladung von Methoden:

- Verschiedene Methoden können den gleichen Namen haben, falls ihre Parameterlisten "*verschieden*" sind.
- Parameterlisten sind verschieden falls:
  - es eine unterschiedliche Anzahl von Parametern gibt.
  - unterschiedliche Datentypen bei den Parametern (falls dann Indeterminismen beim Aufruf immer eindeutig aufgelöst werden können).

Beispiel:

- bei Rechteck (3.0) wird der 3.Konstruktor ausgeführt, da er der einzige 1-stellige Konstruktor ist, der den double-Wert als Eingabe zulässt.
- bei Rechteck (3) passen sowohl 3 als auch 4 Konstruktoren. Bei überladenen Methoden wird stets die Methoden mit der speziellsten Eingabesignatur (muss eindeutig sein) gewählt die passt.
  ⇒ es wird der 4. Konstruktor ausgeführt.

Beispiel:

1) Rechteck (double l, double b)
2) Rechteck (double l, int b)
sind zusammen zulässig.

Aber man darf dann nicht auch noch

3) Rechteck (int l,double b) geben.

Grund: Bei Rechteck (1,2) ist unklar welcher Konstruktor ausgeführt wird.
Konstruktor 2 ist spezieller als 1
Konstruktor 3 ist spezieller als 1
Aber Konstruktor 2 und 3 sind unvergleichbar.

Beipiel für überladene Konstruktoren:

System.out.print $\underbrace{(\dots)}_{\text{Argumente bel. Typen}}$
⇒ verschiedene Methoden dieses Namens.
Welche Ausgeführt wird, hängt vom Typ des Eingabearguments ab.

2.4.1. Schreibweise

Manchmal will man in Konstruktoren expilzit auf das Objekt zugreifen, dass gerade erzeugt wird: "*this*"

Falls Konstruktor 2 wie folgt geändert wird, dann kann man (bislang) nicht mehr auf laenge und breite des gerade erzeugten Objekts zugreifen:

public Rechteck (double laenge, double breite){

$$\underbrace{this.laenge}_{\text{laenge Attr. des gerade erzeugten Obj.}} = \underbrace{laenge}_{\text{formaler Parameter des Konstruktors}}$$

In sonstigen (nicht-statischen) Methoden:

this bezeichnet das Objekt, für das sie die Methoden aufrufen.

f(...){...this...}

r.f(...)
ist bei diesem Aufruf das Objekt r.

Weiterer typischer Konstruktor: Kopier Konstruktor.

null: zeige ind Leere.
(Initialwert bei nicht-primitiven Datentypen, wenn bei der Initialisierung nichts anderes angegeben wird).

2.5. Vordefinierte Klassen: API

2.5.1. Hier:

- Hüllklassen
- Strings

2.5.2. Hüllklasse:

- hüllen Wert eines primitiven Datentyps in ein Klassenobjekt ein
- Objekte haben im Prinzip nur ein (von außen nicht sichtbares) Attribut vom jeweiligen primitiven Datentyp
- aber: viele weitere Methoden + statische Attribute
  Integer.Min_Value = -2.147483648
  Integer x = newInteger(123);
  Integer y = newInteger("123");
  Integer.parseInt("123") = 123; : vom Typ int
  Integer.toString(123) = "123" : wird bei Ausgabe von Int-Werten benutzt
  x.toString(123) = "123" : wird bei der Ausgabe von Int-Werten benutzt
  Aus Folie(Seite???): x.equals(y) = true
  x.intValue() = 123

2.5.3. Strings: vordifinierte Klasse

- Konstruktoren : String n = new String("Wort");
- Kurzform : String s="Wort";
- Beide Formen sind nicht ganz äquivalent:
  Java verfolgt, welche Strings bereits in Kurzform erzeugt wurden und hält sie in einer Tabelle fest. Wird ein neuer String mit dem gleichen Inhalt in Kurzform erzeugt, verweist dieser auf den gleichen Speicherpaltz.

2.5.4. Methoden: Bezieht sich alles auf die Vorlesung

- equals boolean equals(String)
- char charAt(int i) Bsp: u.charAt(2)=r : char gibt Zeichen an Stelle
  int i zurück
- int length() Bsp: u.length() = 4
- char[ ] toCharArray() Bsp: u.toCharArray[2] = r

- String-Objekte kann man nicht ändern
  s = "Wort";
  s = "Worte"; : jetzt zeigt s auf ein anderes String-Objekt, aber das alte String wurde nicht verändert.

- Klasse für änderbare String: String Buffer

3. Rekursion und dynamische Datenstrukturen

3.1. Rekursive Algorithmen

- *Bisher :* *iterative* Lösung : mehrmaliges Durchlaufen bestimmter Programmabschnitte in einer Schleife
  Haüfig: Verwendung einer $Akkumulator - Variable$ (res), in der nach und nach das Ergebnis aufgesammelt wird

- Jetzt: *rekursive* Lösung
  Führe das Problem für x zurück auf das Problem bei kleinerem Wert als x:

- falls $x > 1$: berechne erst die Fakultät von $x - 1$ und multipliziere Resultat mit x

- falls $x < 1$: gib 1 zurück
  $fak(x) = \{^{x*fak(x-y), falls x>1}_{1, sonst}$

  Selbstbezügliche (rekursive) Berechnungsvorschrift: in Definition von fak tritt fak selbst wieder auf der rechten Seite auf (für kleine Argumente)
  Falls Problembeschreibung rekursiv ist, dann kann man sie oft direkt in einen rekursiven Algorithmus übersetzen (deklarative Programmierung/Programmier-Sprache)

  Algorithmus der sích selbst wieder aufruft: direkte Rekursion

  fak(4) = 4 * fak(3)
  = 4 * 3 * fak(2)
  = 4 * 3 * 2 * fak(1)
  = 4 * 3 * 2 * 1 = 24

- Bei rekursiven Algorithmen müssen die Argumente im rekursiven Aufruf "kleiner" werden, sonst: Nicht-Terminierung!!!

3.1.1. Arten von Rekursion

- direkte Rekursion / verschränkte Rekursion

- lineare Rekursion / nicht lineare Rekursion

- Endrekursion

- <u>lineare</u> Rekursion: jede Ausführung des Methodenrumpfs führt zu höchstens einem rekursiven Aufruf (fak)

- <u>$nicht - lineare$</u> Rekursion: fib (Kaninchen-Population-Vorhersage)
  $fib(x) = fib(x-1) + fib(x-2)$
  fib(x) : Anzahl Kaninchenpaare im Monat x;
  fib(x-1) : Kaninchenpaare aus Monat x;
  fib(x-2) : Je ein paar als Nachkommen, für alle Paare, die mindestens 2 Monate alt sind.

- Algorithmus fib ist ineffizient:

$$\begin{aligned} \text{fib}(20) &= \text{fib}(19) + \text{fib}(18) \\ &= \text{fib}(18) + \text{fib}(17) + \text{fib}(18) \\ &= \text{fib}(17) + \text{fib}(16) + \text{fib}(17) + \text{fib}(17) + \text{fib}(16) \end{aligned}$$

  fib(18) wird 2-mal berechnet : fib(3)
  fib(17) wird 3-mal berechnet : fib(4)
  fib(16) wird 5-mal berechnet : fib(5)
  fib(15) wird 8-mal berechnet : fib(6)
  fib(14) wird 13-mal berechnet : fib(7)

- Algorithmus hat exponentiellen Aufwand:
  Berechnung von fib(n) benötigt in etwa 2^n Rechenschrite!

- direkte Rekusion: Algorithmus ruft sich selbst wieder auf (fak,fib)

- verschränkte Rekursion: System von Funktionen, die sich gegenseitig aufrufen, r.B. f ruft g auf, g ruft f auf

  - <u>$Bsp.$:</u> even(3) = odd(2) = even(1) = odd(0) = false
    odd(-3) = even(-2) = odd(-1) = even(0) = true
    ($|x|$ wird im rekursiven Aufruf kleiner)

  Man muss Methoden benutzen können, bevor sie deklariert wurden.

- <u>$Endrekursion$:</u> Spezielfall der direkten Rekursion. Rekursive Aufrufe dürfen nur zum "Ende" des Algorithmuses auftreten(Rekusriver Aufruf ist die letzte Anweisung, die im Methodenrumpf ausgeführt wird.)
  $\Rightarrow$ rekursiver Aufruf darf nicht in Teilausdrücken sein und nicht vor weiteren Anweisungen stehen
  fak ist nicht endrekursiv. Nach dem rekursiven Afruf muss noch die Multiplikation berechnet werden. Der "alte" Wert von x muss noch gespeichert werden, damit man ihn nachher mit dem Ergebnis des rekursiven Aufrufs multiplizieren kann

sqrt ist Endrekursiv:
Aufruf mit sqrt(0,x,x) : 0,1. x: intervall, dass $\sqrt{x}$ enthält;
2. x: es soll $\sqrt{x}$ berechnet werden
Falls Intervall $> 10^3$, dann halbiere Intervall im rekursiven Aufruf
Endrekursion lässt sich sofort in eine nicht-rekursive iterative Form überführen
Grund: lokale Variablen(uG,oG,x,m,epsilon) dürfen im rekursiven Aufruf übershrieben werden, da man ihre alten Werte nach dem rekursiven Aufruf nicht mehr braucht.
Ersetzung des rekursiven Aufrufs sqrt(uG,m,x) durch oG=m wäre nicht möglich, wenn der rekursive Aufruf z.B. im folgenden Ausdruck stände:

oG + sqrt(uG,m,x) : oG ist der alte Wert

3.1.2. Speicherorganisation bei Rekursion

Nach dem rekursiven Aufruf müssen die alten Werte der lokalen Variablen noch zr Verfügung stehen. (Ausnahme: Endrekursion)

Bei jedem Methoden-Aufruf wir ein neuer Speicherrahmen auf dem Kellerspeicher angelegt. Dieser enthält Speicherplatz für neue Ausprägungen und lokalen Variablen (+ für Rückgabewert "res")

Bei jedem Aufruf wird ein neuer Speiecherrahmen auf den Keller gelegt $\longrightarrow$ potentielle Gefahr des "Stack Overflow". Vergleich zur iterativen Version:

- Rekursion benötigt mehr Speicher und Zeit (iterative Version braucht nur 1 Speicherrahmen).
- Rekursion ist oft kürzer, übersichtlicher.
  Beispiel: Türme von Hanoi:
  (Teile und herrsche)
  - 1. Behandle einfache Fälle;
  - 2. Divide: Bei nicht-einfachen Fällen: teile Problem in 2 oder mehr Teilprobleme auf. (Aufteilung in h-1 und h, bei h>0).
  - 3. Conquer: löse Teilprobleme (typischerweise rekursiv).
  - 4. Kombiniere: setze Teillösungen zur Gesamtlösung zusammen.

  bewege Turm (3, $\underbrace{\alpha}_{von}$, $\underbrace{\delta}_{ueber}$, $\underbrace{\omega}_{nach}$))
  $\Longrightarrow bewegeTurm(2, \alpha, \omega, \delta)$ Schritt 1
  durch Züge $(3, \alpha, \omega)$ Schritt 2
  bewege Turm $(2, \delta, \alpha, \omega)$

Aufruf java Hanoi 3

3.2. Rekursive (dynamische) Datenstrukturen

Rekursive Datenstrukturen haben wir bisher schon (bei der Beschreibung der Syntax von Java) verwendet.
Beispiel: Ausdruck*

- * ist selbst-bezüglich definiert ⇒ können beliebig Gross werden (dynamische Datenstruktur).
- ** Grösse des Objekts ist nicht festgelegt, kann sich verändern.
  Tpisches Beipiel für dynamische Datenstrukturen:
  Listen, Baüme, Graphen,...
  - wie definiert man solche Datenstrukturen in Java?
  - wie definiert man typische(zB. Einfügen, Löschen, Sortieren, Ausgabe,...) Algorithmen (meisst rekursiv) auf diesen Datenstrukturen?

Liste = leer | Element Liste(Rekursion ⇒ beliebig lange Listen erzeugen.).

Hingegen in Arrays liegt die Anzahl der Elemente fest.

Idee: Verwende Datenstruktur Element mit 2 Attributen:
Wert: 1. Element in der Liste (Typ: int)
next: Verweis auf Rest der Liste. (Typ: Element (rek. Datenstruktur))

Objekte der Klasse Element haben ein Attribut der Klasse Element.

Hätten wir nur die Klasse Element, so würde die leere Liste durch null dargestellt.
Nachteil: Wenn man Methoden für Listen schreibt (zB: l.fuegeEin(...),...) dann ist solch ein Aufruf nicht möglich, wenn l = null ist. Man möchte solche Methoden aber auch für die leere Liste aufrufen können ⇒ andere Darstellung der Liste.

Klassen Listen ↷ leere Liste ist jetzt ein Objekt l der Klasse Liste mit l.Kopf = null.

Grund für "kein Schlüsselwort" bei Wert, next:
Zugriff auf die Attribute für alle Klassen des gleichen Pakets möglich (insbesondere für Klasse Liste).

3.2.1. Algorithmen auf rekursiven Datenstrukturen

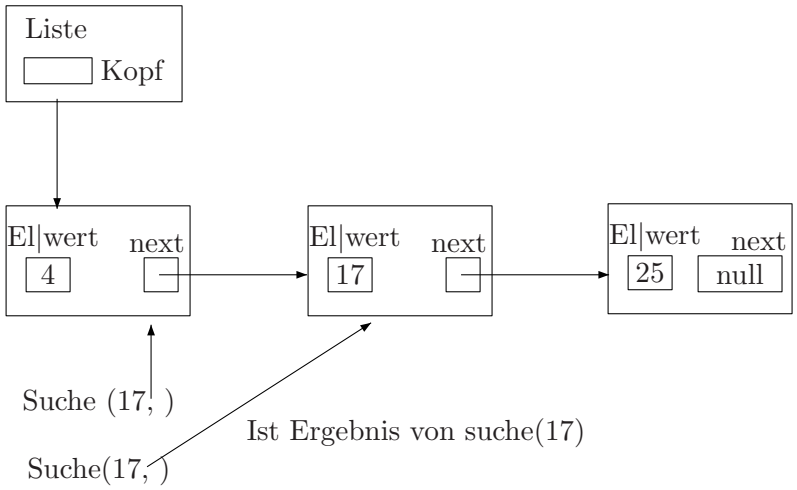
(typischerweise auch rekursiv)

Entwerfe zunächst die Schnittstellendokumentation.
Element (int wert): Erzeug ein Element ohne Nachfolger.
toString(): druckt Wert des Elements.
Liste():erzeuge leere Liste.
Suche(int wert): sucht 1. Element in der Liste mit diesem Wert.
toString: erzeugt String mit Element von vorne nach hinten.
druckeRueckw(): gibt Listenelement in umgekehrter Reihenfolge aus.
fuegeSortiertEin(int wert): füge Wert vor dem ersten grösseren Listenelement ein.

Beispiel:
(4 17 25 30)
(30 25 17 4)

(2 4 12 17 25 28 30 45)

Suche: verwende statische und rekursiv definierte Hilfsmethode "suche(int wert, Element Kopf)".
Sucht sucht nach einem Element mit diesem Wert in der Teilliste die mit Element "Kopf" beginnt.

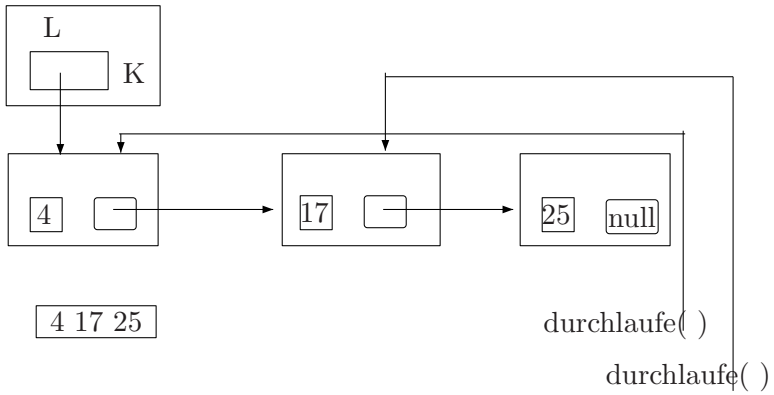


Typischer Algorithmus:

- rekursiver Hilfsalgorithmus uf der Datenstruktur;
- im rekursiven Aufruf wird das bisherige Objekt durch das Objekt (strukturelle Rekursion) im rekursiven Attribut ersetzt.

to String: verwende satische Hilfsmethode, die auf das rekursive Hilfs-Datenstruktur Element arbeitet.

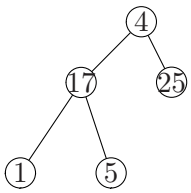
durchlaufe (Element Kopf):
liefert den String aus den Werten von Kopf und seinen Nachfolgern.



Weitere Methoden: später

Jetzt: andere rekursive Datenstruktur: *Binärbaüme*

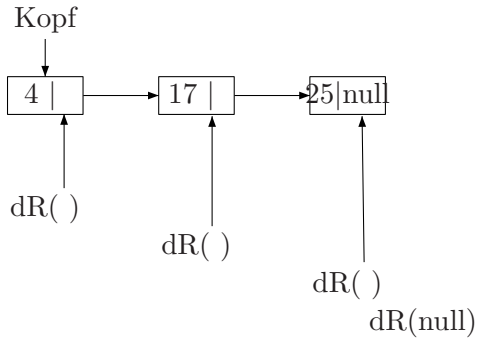
Baum = leer | Knoten Baum Baum



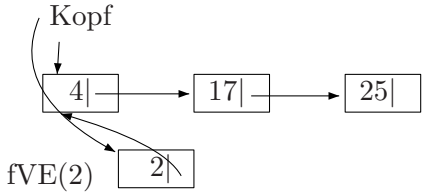
leere Baum:
Objekt b von Typ Baum mit b.wurzel = null.

Rekursiver Algorithmus auf Knoten: rekursiver Aufruf mit links- und rechts-Attributen.

drucke Rückwärts:

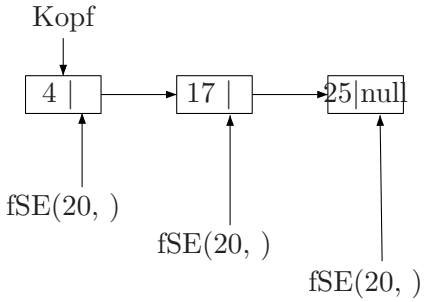


einfügen:

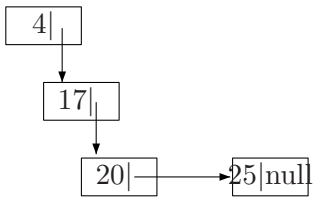


füge sortiert ein:

füge einen Wert vor dem ersten Listenelement ein, das grösser ist.
Bsp: fSE(20)

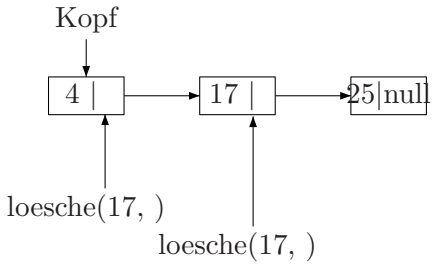


fSE(wert, element) liefert das erste Element der Liste de mit element beginnt und die Wert eingefügt wurde:

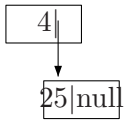


Richtige Verzeigerung, da die rekursive Hilfsmethode fSE jeweils das 1. Element der entstehenden Liste zurückliefert.

loesche:



Loesche(wert, element) gibt das erste Element der Liste zurück, die mit dem Element beginnt und aus der der Wert gelöscht wird.



4. Erweiterungen von Klassen und fortgeschrittene Konzepte

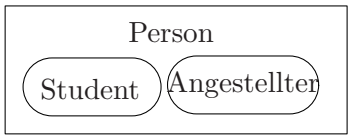
4.1. Unterklassen und Vererbung

4.1.1. Generelles Konzept der Erweiterung von Klassen

Beispiel: Student, Angestellter sind ähnlich aber jeder ein "eigenes" weiteres Attribut. Die Methode toString() ist in beiden Klassen identisch.
Besser: definiere eine neue Klasse *Person*, in der man die gemeinsamen Eigenschaften von "Student" und "Angestellter" zusammenfasst. Student und Angestellter sind Spezialfälle von Person.
⇒ muss nicht neu implementiert werden, sondern man nimmt die Klasse "Person" und *erweitert* sie um ein weiteres Attribut.
Jetzt sind die Klassen "Student", "Angestellter" *Erweiterungen* von "Person". Man sagt:

- "Student ist von Klasse Person abgeleitet"

- "Student ist *Unterklasse* von Person".



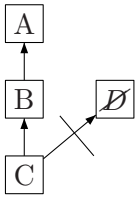
- Unterklassen erben alle Eigenschaften der Oberklasse:
  Jeder Student hat die Attribute key, name, vorname und nachname. Ausserdem hat jeder Student das Attribut matrikelnummer. ("*Vererbung*")

- In Java: nur *Einfachvererbung*, d.h. jede Klasse hat höchstens eine direkte Oberklasse.

  public A {...}

  public class B extends A{...}
  public class C extends B, ~~D~~ {...}
  ~~*publicclassD*{...}~~ verboten in Java!

Grund: Wenn B und D gleiche Attribute/ Methoden (mit gleichem Namen) haben, dann ist unklar, von wem C erbt.



- *Auch Methoden werden vererbt.*
  ⇒ implementiere Methode nur in der Oberklasse. Wenn s ein Objekt von Typ Student ist, wird bei s.toString() die Methode aus der Klasse Person ausgeführt.
- Vererbung geht auch bei statischen Attributen / Methoden. Durch Unterklasse.methode(...) kann man auf die statische Methode(...) der Oberklasse zugreifen.
- Unterklassen können Methoden besitzen, die auch auf Attribute der Oberklasse zugreifen.
  Beispiel: Klasse Srudent könnte folgende Methode enthalten:
  public void setzeKeyZurück(){
  key = 0;
  }
  Dann ist der Aufruf s.setzeKeyZurück() möglich.
- *Zugriff und Zuweisung bei Ober- und Unterklasse*:
  - *p = s erlaubt*: (jeder Student ist auch eine Person).
    Implezite Datentypanpassung vom speziellen Typ (Student) zum allgemeinen Typ (Person).
    Vergleiche implezite Datentypanpassung bei primitiven Datentypen:
    double d;
    d = 5

    Unterschied hier: die spezielle Eigenschaften ("matrikelnr") gehen bei der Datentypanpassung nicht veloren. Man kann nur nicht über p auf sie zugreifen.
    ⇒ p und s bezeichnen das gleiche Objekt, aber p sieht es nur als Person, nicht als Student.
  - *s = a verboten*: Studenten können nur Objekte der Klasse Student oder oder von Unterklassen von Student zugewiesen werden (sonst fehlt matrikelnr).
  - *Nach p = s*:
    p.key = s.key
    ~~*p.matrikelnr*~~ verboten!

    Welche Eigenschaften zugreifbar sind, hängt nicht nur vom Typ des Objekts ab (Student), sondern auch vom Typ der Variablen über die zugegriffen wird.

- *s = p verboten*: Objekte der Oberklasse können nicht automatisch in Objekte der Unterklasse konvertiert werden. (vgl. primitive Datentypen):

  int i;
  i = 5.2 verboten! Keine automatische Konversion von allgemeinen zum speziellen Typ!
  i = (int)5.2

Bei Ober- Unterklassen:

s = (Student)p. explizite Datentypanpassung vom Obertyp (Person) zum Untertyp (Student). → Die gelingt nur dann, wenn p auf ein Studentenobjekt gezeigt hat. (sonst: Exception)
⇒ spezielle Eigenschaften (matrikelnr) die bei der impiziten Datentypanpassung nicht mehr sichtbar waren sind jetzt wieder zugreifbar.
→ geht nicht bei Datentyp verloren.
Vermeidung von Exceptions bei der Datenwandlung:
instanceof (zur Überprüfung des Typs (nur für nicht -primitive Datentypen) von Objekten).
p.instanceof Student: true
p.instanceof Angestellter: false
p.instanceof Person: true.

Bei p = s zeigt p nicht auf das gesamte Studentenobjekt, sondern auf den Teil davon, der Objekt der Klasse Person ist.
Deshalb: p.matrikelnr nicht zulässig.

Objekte sind *polymorph* (vielseitig):
Objekt vom Typ Student kann sowohl als Student-Objekt als auch als Person-Objekt auftreten.

4.1.2. Erzeugung von Objekten in Klassenhierarchien

- *new* Kontruktor-Aufruf
- es existiert eine Oberklasse *Object*
  → alle Klassen sind Unterklassen von Object. Object besitzt auch Konstruktoren? Unsere bisherige Klassen befanden sich auch schon in einer Klassenhierarchie.
  - Man kann im Konstruktor der Unterklasse einen Konstruktor der direkten Oberklasse aufrufen.
    super(. . .).
    Diese Anweisung muss die erste Anweisung im Rumpf des Konstruktors sein.
    Ist bei beliebigen Konstruktoren möglich (mit beliebig vielen Parametern).

– Man kann im Konstruktor auch einen Konstruktor der gleichen Klasse aufrufen:
this...
⇒ this hat 2 verschiedene Bedeutungen:
  * this(key): Aufruf eines Konstruktors.
  * this.vorname: Attribut des Objekts, das gerade erzeugt wird.

  Beispiel: realisiere spezielle Konstruktoren durch den Aufruf von allgemeineren Konstruktoren.

  this(...) muss die 1. Anweisung im Konstruktorrumpf sein.

⇒ this(...) und super(...) können nicht im gleichen Konstruktorrumpf auftreten.

- Falls die 1. Anweisung im Konstruktorrumpf weder super(...)noch this(...) ist, dann wird automatisch die 1. Anweisung *super();* ergänzt!
Falls Person nur Konstruktor

  Person(int key){...}

  enthält, dann führt Konstruktor

  Student(){ ← super();
  matrikelnr = IO.eingabe();
  }
  zu Fehler!

  Falls man in einer Klasse gar keinen Konstruktor schreibt, wird der "leere" parameterlose Konstruktor ergänzt:

  Person () {
  super ();
  }

  Falls die Werte mancher Attribute im Konstruktor nicht gesetzt werden, so werden diese Attribute auf bestimmte Initialwerte gesetzt:
  Initialwert von int: 0
  Initialwert von nicht-primitiven Datentypen: null

4.1.3. Verdecken von Attributen / Überschreiben von Methoden

- *Bisher*:

  - Vererben von Attributen und Methoden von Ober- zu Unterklasse.
  - zusätzliche Attribute und Methoden in der Unterklasse.

- *Jetzt*: Attribute und Methoden die für verschiedene Spezialisierungen (Unterklassen) zwar gleich heissen, aber unterschiedlich arbeiten.

⇒ Regeln: Wann ist welche Definition gültig?

Erst: Namensgleichheit von Attributen ("*Verdecken*")
Dann: Namensgleichheit von Methoden ("*Überschreiben*")

Verdecken von Attributen

Attribute der Unterklasse verdecken namensgleiche Attribute der Oberklasse.
⇒ Welches Attribut gemeint ist, hängt nicht nur vom Typ des Objekts ab, sondern auch vom Typ der Variablen, über die zugegriffen wird.
(würde genauso arbeiten, wenn das Attribut Hochschule in Person und Strudent den gleichen Typ hätte).

Beispiel: Überdecken von "nachname":
Aufruf Student():

initialisiert nachname-Attribut der Klasse Person, aber nicht das nachname-Attribut der Klasse Student.

Student s = new Student();
Person p = s;
p.nachname = "Maier"
s.nachname = null

⇒ Vermeide versehentliche Namensgleichheit von Attributen in Ober- und Unterklasse, da von der Unterklasse aus die Attribute der Oberklasse nicht mehr sichtbar sind.

Überschreiben von Methoden

- Methode "mahnung" mit gleicher Signatur (Ein- Ausgabetypen, Namen der formalen Parameter sind egal) in Ober- und Unterklasse.

  Methode "mahnung" in der Klasse Student *überschreibt* die gleichnamige Methode in der Klasse Person.
  ⇒ Aufrufe der Person-Methode "mahnung" werden auf die Studenten-Methode umgeleitet.
  Bei einem Student wird immer die Methode "mahnung" aus der Klasse Student ausgeführt, auch bei Zugriff über eine Variable der Klasse Person.

  - *Überschreiben von Methoden*: welche Methode ausgeführt wird, hängt *nur* vom Typ des Objekts ab.
  - *Verdecken von Attributen*: auf welches Attribut zugegriffen wird, hängt *auch vom Typ der Variablen* ab.

  Verwendung von überschriebenen Methoden:

  - Sammlung von Objekte der Oberklasse (zB: Person), manche davon sind auch Objekte einer Unterklasse (zB: Student, Angestellter)
  - Durchlaufe die Sammlung und führe jedem Objekt die "gleiche" Methode (zB: mahnung) aus.
  - Welche Methode jeweils wirklich ausgeführt wird, hängt vom Typ des jeweiligen Objkts ab.

- *Ad-hoc-Polymorphismus*, (dynamisches Binden):
  - erst zur Laufzeit wird aufgrund des Typs entschieden welche Methode ausgeführt wird.
  - mehrere Methoden mit verschiedener Implementierung und gleicher Signatur.
- *Parametrisierter Polymorphismus*:
  ein und dieselbe Implementierung einer Methode für Objekte verschiedenen Typs. (Kommt aus der funktionalen Programmierung, zB. bei Haskell, jetzt auch bei Java 1.5)
  "*generics*" (Nicht in dieser Vorlesung behandelt)
- *Schlüsselwort "final"*
  - vor Methoden: Methode darf in Unterklassen nicht überschrieben werden.
  - vor Klassen: die Klasse darf eine Unterklasse haben.
  - vor Attributen: das Attribut ist eine Konstante (deren Wert verändert werden darf).

Zugriffspezifikationen beim Überschreiben von Methoden:

- Überschreibende Methode muss mindestens so sichtbar sein, wie die überschriebene Methode: privat-Methode der Oberklasse darf durch public-Methoden dre Unterklasse überschreiben werden etc...
- statisch / nicht-statisch muss bei überschreibender und überschriebener Methode gleich sein.

Beim Verdecken von Attributen gibt es keine solche Einschränkungen.

Überschreiben von Methoden ist nur dann sinnvoll, wenn die Methode in der Unterklasse "in etwa" dasselbe wie die Mehtode der Oberklasse hat.

Haüfig: Methoden der Unterklasse führen alle Anwendungen der Methode der Oberklasse aus + weitere Aussagen.
Dann rufe aus der Methode der Unterklasse der Methode der Oberklasse auf.
super.mahnung(. . .)

this: aktuelle Objekt (der eigenen Klasse)
super: aktuelle Objekt, aber als Objekt der Oberklasse

Analog: Zugriff auf verdeckte Attribute der Oberklasse von der Unterklasse aus.

s.hochschule (Attribut aus der Klasse Person)

Zusammenfassung

1. Überladen von Methoden
2. Überschreiben von Methoden
3. Verdecken von Attributen.

1. *Überladen von Methoden*:
   mehrere Methoden mit gleichem Namen, aber unterschiedlicher Signatur. (auch in Klassenhierarchien möglich, auch in Kombination mit Überschreiben).
2. *Überschreiben von Methoden*:
   mehrere Methoden mit gleichem Namen und gleicher Signatur in Ober- und Unterklasse.
   Beispiel:
   p.mahnung(10,5); (2. Methode "mahnung" aus der Klasse Person
   s.mahnung(10,5); (Vererbung)

   p.mahnung(15); Methode "mahnung" aus Klasse Student
   s.mahnung(15); (Überschreiben)

   Ändere Methode mahnung in Klasse Student:
   void mahnung (double x){. . . }

   s.mahnung (15); ← Methode "mahnung" aus Klasse Person ausgeführt, dann jetzt ist mahnung nicht überschrieben, sondern überladen. Gäbe es in Klasse Person die Methode
   void mahnung (int get) {. . . } nicht,
   dann würde die mahnung Methode aus Student ausgeführt und 15 würde automatisch in 15.0 konvertiert.

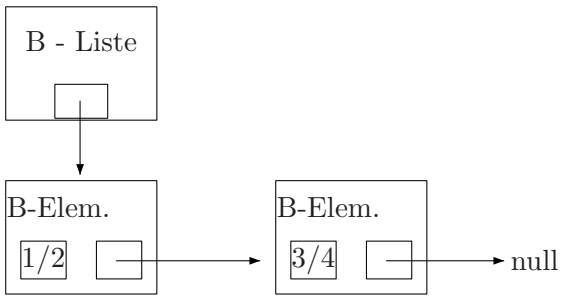
   Überschreiben von Methoden: ad-hoc Polymorphismus, dynamisches Binden. (erst

zur Laufzeit kann aufgrund des Typs des Objekts entschieden werden, welche Methode ausgeführt wird).

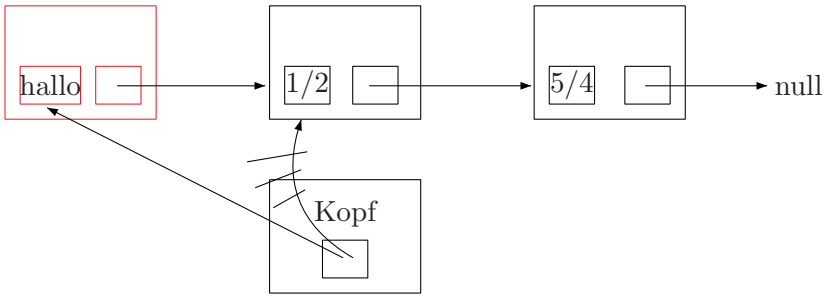
3. *Verdecken von Attributen*:
   Typ der Variable bestimmt, auf welches Attribut zugegriffen wird.
   protected: Sinnvoll für Attribute und Methoden, auf die man in Unterklassen zugreifen will, die man aber nicht überall sichtbar machen möchte.

4.2. Abstrakte Klassen und Interfaces

Listen von Büchern / Listen von Werten:
⇒ ähnliche Programmteile



Besser: Identifiziere ähnliche Aufgaben und implementiere nur einmal eine allgemeine Lösung.
Lösung: die spezielle Klassen Bruch, Wert durch allgemeine Oberklasse. *Object*.
Beispiel:



⇒ Listen mit verschiedenen Objekten durcheinander möglich. (Lässt sich zB. durch Verwendung *generischer Datentypen* (Typ Liste mit Element beliebigen aber gleichen Typs) verhindern. → sit Java 1.5

Weiterer (grössere) Nachteil

Liste kann beliebige Objekte enthalten, auch solche, die man nicht sinnvoll auf Gleichheit überprüfen kann.

Beispiel: Liste l von oben.
b = newBruch(1,2);

l.suche(b) liefert null, da das Objekt 1/2 in der Liste inhaltlich gleich ist zu b, aber es ist ein unterschiedliches Objekt.
⇒ "==" bei nicht-primitiven Datentypen vergleicht nicht inhaltlich.

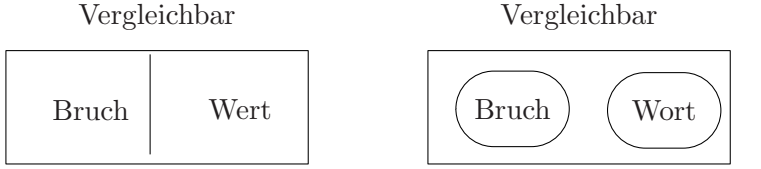
Satt "==" sollte man eine Methode "gleich" benutzen, die Objekte inhaltlich vergleicht. Man sollte nur solche Werte in die Liste einfügen dürfen, die zu Klassen gehören, in denen es die Methode "gleich" gibt. (Methode kann in jeder Klasse unterschiedlich implementiert sein)
⇒ Definiere eine neue Klasse "Vergleichbar", die Oberklasse von allen Klassen ist, in denen es die Methode "gleich" gibt.

public *abstrakt* class Vergleichbar{ (← Abstr. Klasse besitzt eine abstr. Methode)

public *abstrakt* boolean gleich (Vergleichbar zu vergleichen);
}
public class Bruch extends Vergleichbar{
public boolean gleich (Vergleichbar zuvergleichen){
...
}
}...
public class Wert extends Vergleichbar{
⋮
}

Abstrakte Klasse: legt durch abstrakte Methoden fest, dass alle Unterklassen diese Methode besitzen müssen.

Unterklasse einer abstrakten Klasse: muss alle Methoden implemtieren (dh. mit konkreten Methoden überschreiben). Sonst ist die Unterklasse auch abstrakt.



bei abstrakten Klassen bei nicht-abstrakten Klassen

⇒ es darf keine Objekte x geben, die vom Typ Vergleichbar sind und nicht auch noch vom Typ Bruch oder Wort.
Grund: x.gleich(...) ist nicht ausführbar

Daher: $Vergleichbar x = \cancel{new} Vergleichbar()$; verboten!

Konstruktoren von abstrakten Klassen darf es geben, aber man kann sie nur in Kon-

struktoren der Unterklasse aufrufen.

(super(. . .);)
Beispiel: Vergleichbar statt Object in der Klasse Liste.

Listen nur einmal realisieren.
Aber: Listen mit beliebigen Objekten haben den Nachteil, dass man nicht davon ausgehen kann, dass für diese bestimmte Methode (z.B. "gleich") vorhanden sind.

→ Listen mit Objekten, bei denen es eine Methode "gleich" gibt führen Oberklasse "Vergleichbar" ein. Dient dazu, die Unterklasse zusammenzufassen, bei denen es Methode "gleich" gibt. Es existieren aber keine Objekte der Klasse "Vergleichbar", die nicht auch noch Objekt einer Unterklasse sind.

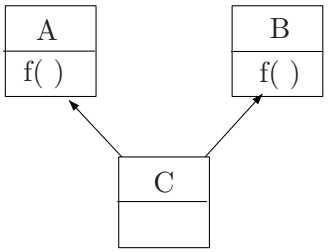
⇒ Klasse ist abstrakt, Methode "gleich" wird in dieser Klasse nicht implementiert (ist auch abstrakt).

Verwendung von abstrakten Klassen wie bei normalen Klassen, abstrakte Methoden können auch wie üblich aufgerufen werden.
Sowas nennt man auch "*generisches Programmieren*":
Arbeite Gemeinsamkeiten heraus von Datenstrukturen und Algorithmen und formuliere allgemeine Lösungen auf höherer Abstraktionsebene.

Abstrakte Klassen:

- Besitzen teilweise schon implementierte Bestandteile
- Können teilweise auch nur vorschreiben, was später in Unterklassen noch zu implementieren ist.

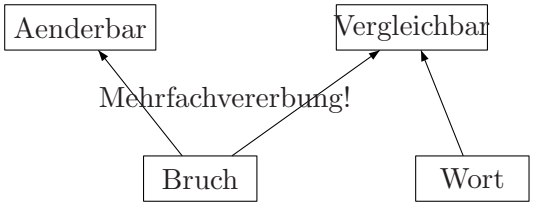
Nach wie vor: *Einfachevererbung*, d.h. jede Klasse hat höchstens eine direkte Oberklasse.
Grund: Sonst ist die Vererbung nicht eindeutig.



Bei Ausführung von Cc;
c.f( ); ist unklar, welche Methode f verwendet werden soll.

Problem tritt nur bei normalen Methoden f auf, nicht bei abstrakten Methoden? Falls A und B nur abstrakte Methoden enthalten, könnte man Mehrfachvererbung zulassen.

Beispiel: Vergleichbar $\hat{=}$ Unterklassen müssen eine Methode "aenderung" implementieren.



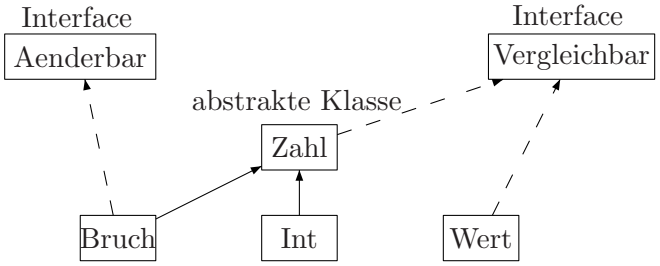
Java-Konstrukt für "vollkommen abstrakte" Klassen, in denen alse alle Methoden abstrakt sind: *interface*

Ähnlich zur Schnittstellendockumentation:
Interface beschreibt die Signatur der öffentlich sichtbaren Methoden, aber nicht ihre Implemntierung.

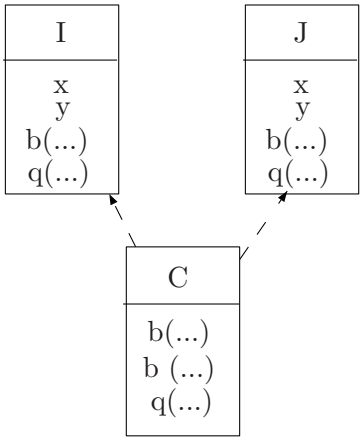
4.2.1. Deklaration von interfaces

- "interface" statt "abstract class"
- Kein "abstract" vor Methoden, denn alle Methoden eines Interfaces sind abstrakt.
- Kein "public" vor Methoden nötig, denn alle Methoden eines Interfaces sind public.
- "implements" statt "extends" für Unterklassen eines Interfaces.
- Mehrfachvererbung bei Interfaces zulässig:
  Jede Klasse kann eine direkte Oberklasse und beliebig viele direkte "Ober-Interfaces" (Interfaces, die von der Klasse implementiert werden) haben.
- Interfaces können auch Attribute / Variablen haben. (Konstruktoren des Interfaces) Diese sind automatisch public, static, final → Problem bei Mehrfachvererbung.

  Beispiel: für Interfaces und abstrakte Klassen:



Verwendung von Interfaces $\hat{=}$ Verwendung von Klassen ("Vergleichbar" wird in der Listen-Implementierung wie bisher benutzt.)



Ii = z; implizite Datentypkonversion von Unterklasse zu Oberklasse.

Ji = (C)i; explizite Datentypkonversion

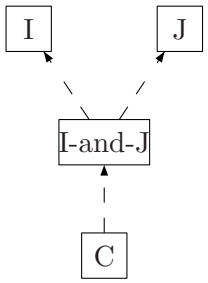
i.b(5); ← 1.Methode b in Klasse C wird ausgeführt (Überschreiben)
j.b(5); ← 2.Methode b in Klasse B wird ausgeführt :b(5.0) (Überschreiben)
z.b(5); ← 1.Methode b in Klasse C wird ausgeführt (Überladen)

i.q(5);
j.q(5);
z.q(5); in allen drei Fällen wird Methode q aus C ausgeführt.

Gäbe es in Interface I eine Methode
void p (int i);
und in Interface J Methode
int p (int i);
kann es keine Klasse geben, die sowohl I als auch J implementiert.

I.x = 4
J.x = 3
C.y = I.y = 6
C.x verboten!, da dies wegen der Mehrfachvererbung nicht eindeutig wäre.

Interfaces können andere Interfaces erweitern. Aber Interfaces können keine Klassen erweitern. (→ sonst hätte man echte Mehrfachvererbung)



4.3. Modularität und Pakete

- SW sollte modular aufgebaut sein (Module von unterschiedlichen Personen entwickelt, getestet, gewartet)
- Beispiel: Vordefinierte Komponenten in Java-Bibliotheken (JDK)
- Paket (package): Zusamenfassung von mehreren Klassen und Interfaces (über mehrere Dateien verteilt), die Inhaltlich zusammengehören.

Beispiel: Liste, Element

2 Klassen, die inhaltlich zusammengehören können in zwei verschiedene Dateien *stehen*.
(por Datei höchstens eine public-Klasse . Datei muss auch so heissen.)(Liste.java, Element.java)
⇒ sollten in das gleiche Paket "listen"
→ jede Datei sollte dann mit package listen; beginnen. Die Datei en Listen.java, Element.java sollten auch im gleichen Verzeichnis "listen"(Pfad/Name des Directories vom Verzeichnis aus, in dem Java-Programm compiliert und ausgef¨hrt wird) stehen. (Zusammenhang zwischen logischer und physischer Programmstruktur).
Zugriffspezifikationen regeln den Zugriff auf die Elemente eines Pakets.

- Kein Schlüsselwort: Zugriff erlaubt im eigenen Paket;
- public: Zugriff überall erlaubt, auch in anderen Paketen;
- protectet: Zugriff erlaubt im eigenen Paket un din Unterklassen

Bisher: keine Angabe von "package": Klassen gehören zu einem Standartpaket ohne speziellen Namen.
Beispiel: Directory Listen, mit Klasse Element, Liste Hauptdirectory enthält Test 1,2,3.
Zugriff durch Voranstellen des Paketnamens.

FEHLT LETZTE VORLESUNG !!!!

4.4. Exeptions

Teil II.

Funktionale Programmierung

5. HASKELL

5.1. Prinzipien der funktionalen Programmierung

5.1.1. Schreibweise

- **n : x** drückt die Liste aus, die aus Liste x entsteht, wenn man vorne Element n einfügt
  $15 : [70, 36] = [15, 70, 36]$

- Jede nicht-leere Liste lässt sich als
  Kopf : $\underbrace{rest}$
  Liste ohne ihr 1. Element
  darstellen.

- [ ] bezeichnet die leere Liste
  15: 70 :36 :[ ] = [15,70,36]
  15: (70: (36: [ ])) ":" assoziert nach rechts.

  Haskell-Programm: Menge von Funktionsdefinitionen (im Bsp: wird die Funktion len durch 2 Funktionsdeklarationen definiert).
  Man darf Klammern von Funktionsargumenten weglassen.
  $len([\,]) = 0$ *oder* $len[\,] = 0$

- Definition ist <u>*rekursiv*</u>:
  Len tritt selbst wieder auf der rechten Seite seiner Funktionsdeklarartion auf

  $len[\,] = 0$
  $len(kopf : rest) = 1 + len\ rest$

- Auswertung von Ausdrücke = Ausführung eines H-Programmes
  - (5+ 6)* 7 wird zu 77 ausgewertet "Taschenrechner"
  - len $\underbrace{[15, 70, 36]}_{Kopf \longrightarrow 15:[70,36] \longleftarrow rest}$ $\Longrightarrow$ 1 + len[70,36] *TERMERSETZUNG*

- *referenzielle Transparenz*: Ergebnis der Auswertung eines Ausdrucks ist immer gleich, egal wie der Speicherzustand ist.

- *Funktion Höherer Ordnung*: man kann Funktionen programmieren, die selbst wieder Funktionen als Argument oder als Ergebnis haben.

- Polymorphismus:
  - *ad-hoc Polymorphismus*: mehrere Implementierungen von "f".
    Welche davon ausgeführt wird, hängt von den Typen der Agumente ab (Überladung von Methoden).
  - *parametrischer Polymorphismus*: die selbe Implementierung von "f" kann für Argumente verschiedenen Typs ausgeführt werden.
    Beispiel: len kann für Listen von Integer, Boolean,... ausgeführt werden.

Beide Formen von Polymorphismus gibt es in Haskell und seit Java 5 auch in Java.

- Funktionale Sprachen sind oft langsamer als imperative
$\Longrightarrow$ es hängt von der Anwendung ab, welche Sprache am besten geeignet ist.

5.1.2. Verwendung von Hugs (Interpreter)

> (5 + 6)*7
77

> :$\underbrace{l}_{load}$ $\underbrace{len.hs}_{\text{File mit len-Programm}}$
> len[15,70,36]
3

5.1.3. Grundlegende Sprachkonstrukte von Haskell

Deklaration, Ausdrücke, Patterns, Typen

- H-Programm: Folge von Deklarationen, die *linksbündig* untereinander stehen.
- 2 Arten von Deklarationen:
  - *Funktionsdeklaration*: beschreibt die eigentliche Abbildungsvorschrift.
  - *Typdeklaration*: gibt Typen der Argumente des Ergebnisses an.

    Int: Datentyp der Intereger-Zahlen;
    [Int]: Datentyp der Listen von Integer;
    [Int] -> Int: Datentyp der Funktionen die eine Listen von Integer auf Integer abbildet.

- Kommentare:

  - -... bis Zeilenende
  {-...-}

- Als Funktionssymbole dienen *Variblenbezeichner* (Strings, die mit *Kleinbuchstaben* anfangen. z.B. len; square, x, kopf, rest, ...)

  $\underbrace{square}_{var} :: \underbrace{Int -> Int}_{type}$

- es existieren vordefinierte Typen:

  Int, Bool, Char,...

- wenn $[t]$ ein Typ ist (typ der Listen, deren Elemente alle den Typ t haben)

  z.B.: [Int],[Bool], $\underbrace{[[Int]]}_{[[1,2],[0],[-5,70]]}$

- wenn t_1 und t_2 Typen sind, dann ist auch t_1 -> t_2 ein Typ (Typ der Funktionen von t_1 nach t_2).

  z.B. Int -> Int,
  $[Int]$-> Int,
  $[[Int]]$-> Bool,
  $\underbrace{(Int -> Int)}_{Eingabe:Funktion}$ -> $\underbrace{(Int -> Int)}_{Ausgabe:Funktion}$ "Funktion höherer Ordnung"

- Funktionsdeklaration:

  $\underbrace{square}_{Name\ d.f.Var}\ \underbrace{x}_{Pattern} = \underbrace{x * x}_{Ausdruck(Expression)}$

  Pattern: Muster für die erwarteten Argumente

  Jeder Ausdruck hat einen Typ: die Typen müssen passen!
  (bei square :: Int -> Int darf nicht
  square x = True definieren)

- Typdeklarationen sollten zwar angegeben werden, müssen aber nicht. Dannn berechnet Haskell die Typdeklaration automatisch.

- Vordefinierte Grundoperationen:

  $-\ +, *, -, /, \ldots$

  - ==, >, >=, <, <=, / =(Bilden ab in die Datenstruktur Bool mit den Werten True + False)
  - &&, //, not, ...

- Vordefinierte Funktionen stehen in Bibliotheken. Beim Start von Haskell wird standartmässig das sogennante "*Prelude*" geladen.
  Beim Start von Haslkell wird angegeben, wo die File "*Prelude.hs*" steht.

- Man kann auch Funktionen ohne Argumente deklarieren (Konstanten).
  one :: Int
  one :: 1

5.1.4. Auswertung funktionaler Programme

Auswertung eines Ausdrucks durch *Termersetzung*:

- 1. Suche Teilausdruck (*Redex* (reduzierte expression)), die der linken Seite einer Gleichung entspricht, wobei die Variable der linken Seite durch entsprechende Ausdrücke ersetzt werden müssen.

- 2. Ersetze Redex durch rechte Seite der Gleichung, wobei die Variable der rechten Seite so wie in Schritt 1 belegt werden müssen.

Auswertung wird so lange wiederholt, bis kein weiterer Ersetzungsschritt möglich ist.

Auswerungsstrategie: wählt auszuwertender Redex, falls es mehrere gibt.

- Ergebnis der Auswertung ist immer dasselbe, unabhängig von der Strategie (falls es terminiert)

- Effizienz (Auswahl der Auswertungsschritte) hängt von der Auswertungsstrategie ab.

- Terminierung hängt auch von der Auswertungsstrategie ab.

  call-by-value: werte erst Argumente aus, wende dann Funktion an. (strikte Answertung)
  call-by-name: in etwa Auswertungsstrategie von Haskell
  call-by-reference: existiert in der funktionalen Sprache nicht.

  - call-by-value:
    leftmost innermost,
    strikt,
    lager
  - call-by-name:
    leftmost outermost,
    nicht-strikt.

VORTEIL der nicht-strikten Strategie:
werte nur Teilausdrücke aus, deren Wert zum Ergebnis beiträgt:

three :: Int -> Int
three x = 3
three (12 - 1) = 3 in einem Schritt

$\Longrightarrow$ Effizienzverlust !

HASKELL: nicht-strikte Auswertung, aber doppelte Teilausdrücke, die durch die Auswertung entstehen werden nur einmal repräsentiert. (Beeinflusst die Effizienz nicht das sonstige Verhalten.) *LAZY EVALUATION*

Auswertungsstragtegie beeinflusst die Terminierung:

three :: Int -> Int
three x = 3

non_term :: Int -> Int
non_term x = non_term(x+1)

three(non_term 0)

Strikte Auswertung terminiert nicht.
Nicht-strikte Auswertung in 1 Schritt, Ergebnis ist 3.

- Wenn irgendeine Auswertungsstrategie terminiert, dann terminiert auch die nicht-strikte Auswertung.
- Wenn 2 Auswertungsstrategien terminieren, dann liefern sie das gleiche Ergebnis.

5.2. Deklaration

5.2.1. Bedingungen und Tupel

- wenn $\tau_1, \ldots, \tau_2$ Typen sind, dann ist auch $(\tau_1, \ldots, \tau_2)$ ein Typ:
  Der Typ der Tupel mit n Komponenten der Typen $\tau_1 \ldots \tau_2$

  (Int,Int): (2,3)
  (-5,7)

  ([Int],Bool,Int): ([0, 1],True,17)

  Bei Listen müssen alle Elemente der Liste den gleichen Typen haben. $[0, True]$ nicht erlaubt!

- Ausdruck auf rechter Seite einer definierten Gleichung kann durch eine Bedingung (Ausdruck vom Typ Bool) eingeschränkt werden. Es wird die erste Gleichung verwendet, deren Bedingung erfüllt ist.

  otherwise :: Bool
  otherwise = True

5.2.2. Currying (Haskell B. Curry)

plus :: Int -> (Int -> Int) (assoziert nach rechts)
plus x y = x + y

(plus 2) 3 = 5 (Funktionsanwendung assoziert nach links)
Funktion von Int -> Int die ihre Eingabe um 2 erhöht.

plus 2 3 = 5

Vorteile:

- Lesbarkeit, da weniger Klammern
- partielle Anwendungen:
  plus darf jetzt auch auf nur das 1. Argument angewendet werden.

  suc :: Int -> Int

  suc = plus 1 (Funktion die auf y wartet und dann 1+y berechnet).

5.2.3. Pattern Matching

Mehrere definierte Gleichungen (mit unterschiedlichen Patterns (beschreiben den Wert des erwarteten Arguments))für das gleiche Funktionssymbol.

Patterns:

- Variablen x, y, ...
- Terme aus Variablen und *Datenkonstruktoren*.
  Jede Datenstruktur besitzt eine Menge von Datenstrukturen (spezielle Funktionssymbole), mit denen man alle Objekte der Datenstruktur darstellen kann.

  Datenstruktur Bool: Datenstruktur True, False.

Auswertung bei Pattern Matching:

Um einen Ausdruck auszuwerten testet man die definierenden Gleichungen von oben nach unten und sucht nach der ersten Gleichung, bei der die linke Seite auf den auszuwertenden Ausdruck "$\underbrace{passt}_{\text{matcht}}$".

und True True und True y "passt"

Instantiierung: $[y/True]$

Danach wird der Ausdruck durch die entsprechende instantiierte rechte Seite ersetzt.

= y $[y/True]$
= True

5.2.4. Pattern Matching für Listen

Grammatik für $[Element]$
Datenkonstruktor für Listen: [ ], :

Patterns:
Terme aus Variaben und Datenkonstruktoren.

(z.B. $\underbrace{[\,]}_{leere\ Liste}$, x:xs, $\underbrace{x:[\,]}_{[x]}$, $\underbrace{True:x:xs}_{alle\ Listen\ mit\ min\ 2\ El.}$,...)

second :: $[Int]$ -> Int
second [ ] = 0
second (x:[ ]) = 0
second (x:y:xs) = y

len$\underbrace{[1\ 2]}$

1:2:[ ] = 1 + len(2:[ ])
1:(2:[ ])
$x/1, xs/2 : [\,] = 2$

5.2.5. Pattern Matching für Integer

Patterns

- Variablen x, y, ...
- Zahl 0, -5,...
- Variable + Zahl (x+1) fasst auf alle n $\geq$ 1, x wird mit x-1 instantiiert.

  sub7 :: Int -> Int
  sub7 (x+7) = x
  sub7 $\underbrace{9}_{7+2}$ = 2

  fac $\underbrace{2}$ = (0+1) + 1 * fac 0+1
  (0+1)+1 = ...
  x/0+1 = 2

  Pattern $\underbrace{x}_{var}$ + $\underbrace{k}_{Zahl \geq 1}$ passt auf alle Zahlen $\geq$ k

  Definierende Gleichungen müssen nicht vollständig sein.
  $\longrightarrow$ fac(-5) führt zu Fehler
  Abhilfe: füge dritte Gleichung hinzu:
  fac x = 0

5.2.6. Lokale Deklaration

Lösung grundlegender Gleichung.
Suche alle x mit $x = \frac{-b \pm \sqrt{b^2-4ac}}{2a}$ setze $d = \sqrt{b^2 - 4ac}$ *und* $e = 2a$.
Lokale Deklarationen: Deklarationen, die nur lokal (dh. nur in einer bestimmten definierten Gleichung) sichtbar sind.

"where"

Vorteil:

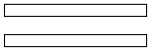
- Effizienz (im Bsp. werden d und e nur 1 man ausgewertet)

- Lesbarkeit: Hilfsfunktionen, die man nicht auf oberster Ebene braucht, sollten lokal sein.

5.2.7. Schreibweise von Deklarationen: (offside-Regel)

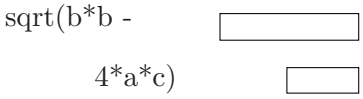
($\longrightarrow$ Lesbarkeit, Vermeidung vo,{...},;)

1. Das erste Symbol in einer Sammlung locdecls von Deklarationen bestimmt den linken Rand des Deklarationsblocks.



2. Eine neue Zeile, die an diesem linken Rand anfängt, ist eine neue Deklaration in diesem Block.

3. Eine neue Zeile, die weiter rechts anfängt, gehört zur selben Deklaration (d.h. ist die Fortsetzung der darüberliegenden Zeile).



4. Eine neue Zeile, die weiter links anfängt bedeutet, dass der locdecls-Block bendet ist und sie nicht mehr zu dieser Sammlung von Deklarationen gehört.

5.3. Ausdrücke

Ausdrücke haben eine

- Wert (zB Auswertung im Interpreter)
- Typ (zB :t Ausdruck)

Vor der Auswertung wird immer überprüft ob der Ausdruck korrekt getypt ist.

:t True ergibt Bool in Hugs
True wird in True ausgewertet.

- *Variablen*
  x, y, square, len,...
  Strings, die mit Kleinbuchstaben beginnen.

- *Datankonstruktoren*: constr;
  True (Typ:bool), False, [ ], :,...
  Strings, die mit einem Grossbuchstaben beginnen.

  [ ], : Wenn x den Typ a hat und xs [a] hat, dann hat x:xs den Typ [a].

- *Integer*
  0,1,-1,2,-2,... ⟵ Typ int ist vordefiniert

- *Float*
  -2.5, 3.4e+23,... ⟵ Typ float ist vordefiniert

- *Character*
  $'a', \ldots, 'b', 'A', \ldots, 'Z', '0', \ldots, '9'$ ⟵ Typ char
  $' ', \underbrace{'\backslash n'}_{Zeilenumbruch}$

- $[exp_1, \ldots, exp_n]$(n≥ 0)
  Liste der Ausdrücke $[exp_1, \ldots, exp_n]$.
  [0,1,2,3] ist Abkürzung für 0:1:2:3:[ ]

  Wenn $exp_1, \ldots, exp_n$ alle Typ a haben, dann hat $[exp_1, \ldots, exp_n]$ den Typ [a].

- *String*
  Liste von Zeichen (char)
  $['h', 'a', 'l', 'l', 'o']$ ⟵ Typ String = [char]
  Kurzschreibweise: "hallo"

- $(exp_1, \ldots, exp_n)$(n≥0)
  Tupel der Ausdrücke $exp_1, \ldots, exp_n$
  Wenn exp_1 den Typ a_1 hat,... exp_n den Typ a_n hat,

dann hat ($exp_1 \ldots exp_n$) den Typ ($a_1,\ldots,a_n$).

(10, False) hat den Typ (Int, Bool)
([0,1,2], False) hat den Typ ([Int], Bool)
(square, False) hat den Typ (Int -> Int, Bool)

1-elemntige Tupel (exp) wird mit exp identifiziert. (Int wird mit Int identifiziert).

0-elementige Tupel ( ) hat den Typ ( ).

- *$exp_1 \ldots exp_n$*, n≥2 *Funktionsanwendung*
  exp_1 wird auf exp_2 angewendet.
  $\underbrace{square}_{Int->Int}\ \underbrace{10}_{Int} = \underbrace{100}_{Int}$

  $$\underbrace{\underbrace{exp_1}_{Typa->b}\ \underbrace{exp_2}_{a}}_{Gesammtausdruck\ hat\ Typ\ b}$$

  $$\underbrace{plus}_{Int->Int->Int}\ \underbrace{5}_{Int}\ \underbrace{3}_{Int} = \underbrace{8}_{Int}$$

  $$\underbrace{\underbrace{exp_1}_{a->b->c}\ \underbrace{exp_2}_{a}\ \underbrace{exp_3}_{b}}_{Typ\ c}$$

  $$\underbrace{\underbrace{exp_1}_{a->(b->c)}\ \underbrace{exp_2}_{a}}_{Typ\ b->c}$$

  plus 5 hat den Typ Int->Int

- *if exp_1 then exp_2 else exp_3*
  exp_1 muss vom Typ Bool sein
  exp_2 und exp_3 müssen vom gleichen Typ sein.
  Gesammtausdruck hat auch den Typ a.

  Auswertung: Werte erst exp_1 aus.

  Falls das Ergebnis True ist, dann ist das Resultat exp_2.

  Falls das Ergebnis Fals ist, dann ist das Resulatat exp_3.

- let locdecls in exp
  Lokale Deklarationsfolge für den Ausdruck exp.
  (analog zu where...)

- $\underbrace{\backslash}_{\lambda} pat_1 \ldots pat_n \longrightarrow exp$:Lambda-Ausdruck

  Wert des Ausdrucks ist die Funktion, die $\text{pat}_1 \ldots \text{pat}_n$ auf exp abbildet.

  \ x-> 2*x : Funktion, die Zahlen verdoppelt.
  (\ x ->2*x) 5 = 2*5= 10

  \xy -> x + y : Additionsfunktion
  (\xy -> x+ y)5 3 = 5 + 3 = 8
  (\xy -> x + y)5 = $\underbrace{\backslash y- > 5 + y}_{\text{Funktion dieZahlen um 5 erhöht}}$.

  Statt plus :: Int -> Int -> Int
  plus xy = x + y kann man auch schreiben:

  plus = \xy -> x + y oder
  plus x = \y -> x + y.

  Mit Lambda-Ausdrücken kann man Funktionen mit nur einem Ausdruck darstellen.

  ("anonyme" Funktion).

  $\backslash \underbrace{pat_1}_{a_1} \ldots \underbrace{pat_n}_{Typ\ a_n} - > \underbrace{exp}_{b}$

  Gesammtausdruck hat Typ : $a_1 - > a_2 - > \cdots - > a_n - > b$

  Funktionen sind "gleichberechtigte Datenobjekte":

5.4. Muster (Patterns)

Patterns: beschreiben die Form von erwarteten Argumenten. Im wesentlichen: Terme aus Variablen und Datenkonstruktoren.

5.4.1. Ablauf des Pattern Matchings

app :: [Int] -> [Int] -> [Int]
Listenkonkaternation: app$[0,1][2,3] = [0,1,2,3]$
app [ ] ys = ys
app (x:xs)ys = x : app xs ys

Auswertungsbeispiel:

len (app $\underbrace{[1]}_{1:[\ ]}$ [2])

Man will len-Gleichung anwenden. Dazu muss man wissen mit welchem Konstruktor das Argument gebildet wird. $\Longrightarrow$ werte Argument solange aus, bis der oberste Konstruktor des Arguments fessteht. D.h. bis eine len-Gleichung anwendbar ist.

= len (1 : app [ ][2])
= 1 + len (app[ ][2])
= 1 + len [2]
= ... = 2

f :: [Int] -> [Int] -> [Int]
f [ ]ys = [ ]
f xs [ ] = [ ]

zeros :: [Int]
zeros = 0 : zeros [0,0,0,...

Auswertung von
f [ ]zeros =
f zeros [ ] = $\longleftarrow$ hier weiss man nicht, ob die 1. def. Gleichung von f anwendbar wäre. Zeros könnte ja zur leeren Liste auswerten.
f(0 : zeros)[ ] = [ ]

5.4.2. Einschränkungen

Patterns müssen linear (keine doppelten Var.) sein. Sogar die linke Seite jeder Gleichung in einer Funktionsdeklaration muss linear sein.

Grund: Sonst wäre das Ergebnis der Auswertung abhängig von der Auswertungsstra-

tegie.

zeros = 0 : zeros

equal	zeros	zeros
↓	↓	
True	equal	zeros (0 : zeros)
(1. equal-Gl.)		↓
		False (2. Gl)

Patterns:
° worauf passt (matcht) das Pattern?
° wie werden die Variablen des Patterns instantiiert?

- *Variable*

  square $\underbrace{x}_{\text{pattern}}$ = x * x

  Pattern passt auf jeden Wert, Variable wird mit dem Wert instantiiert, auf dem sie gemacht wird.

  square $\underbrace{5}_{x/5}$ = 5 * 5 = 25

- _ (*underscore*)*Joker* − *Pattern*
  Passt auf jeden Wert, es finder keine Instantiierung statt.

- *Integer*, *Float*, *Char*, *String*
  Passen nur auf sich selbst, es findet keine Instantiierung statt.
  fac $\underbrace{0}_{Pattern}$ = 1

- ($\underbrace{constr}_{n-stelliger Datenkonstr.}$ $pat_1 \ldots pat_n$)(n≥0)
  Passt auf Werte, die mit dem selben Konstruktor constr gebildet werden, falls pat_i auf das i-te Argument des Werts passt. (für alle i)

  Beispiel:
  x : xs Infix-Notation
  $\underbrace{(i)}_{Datenk.}$ $\underbrace{x}_{pat_1}$ $\underbrace{xs}_{pat_2}$ Präfix-Notation
  len(_ : xs) = 1 + len xs (auch möglich)

- *Variable* + *Integer*
  Passt auf Integer.

  − Pattern x + k (k ∈ $N_{->0}$) passt auf Zahlen n ≥k

  - x wird mit n-k instantiiert

- $[pat_1, \ldots, pat_n] (n \geq 0)$
  Passt auf Listen der Länge n, bei denen pat_1 auf das erste Listenelement passt,. . . , pat_n auf das n-te Listenelement.

- $(pat_1, \ldots, pat_n) (n \geq 0)$
  Passt auf Tupel der Länge n, bei denen pat_i auf die i-te Tupelkomponenten passt (für alle i).

5.5. Typen und datenstrukturen

Typ: Mengen von gleichartigen Werten, die durch einen entsprechenden Typausdruck bezeichnet wird.

Beispiel:
Int, Bool, Char, Float,...
(Bool, Int), [Int], Int -> Bool,...
([[Int]], Bool -> Int),...

- (typconstr$\text{type}_1 \ldots \text{type}_n$), n ≥ 0

  Erzeuge neuen Typ aus bestehenden Typen $\text{type}_1 \ldots \text{type}_n$ durch einen *Typkonstruktor* typconstr. (Strings die mit Grossbuchstaben anfangen (Syntaktisch eine Datenkonstruktion)).

  Beispiele für Typkonstruktoren : Int, Bool, ...
  Beispiele für Datenkonstruktoren: True, False, ...

- [type]

  [...] ist ein vordefinierter 1-stelliger Typkonstruktor. Er bekommt einen Typ (zB.Int) als Argument und liefert den Typ der Liste mit Elementen des Argumentstyp ([Int]).

  Beispiel: [[Int]] enthält $[[0, 1], [2], [3]]$

- typ_1 -> typ_2
  → ist ein vordefinierter 2-stelliger Typkonstruktor.
  typ_1 -> typ_2 ist der Typ der Funktionen, wobei das Argument den Typ typ_1 hat und das Resultat den Typ typ_2.

  Beispiel: Int -> Int enthält square
  Int -> (Int -> Int) enthält plus.

- ($\text{type}_1 \ldots \text{type}_n$), n ≥ 0

  (...) ist ein vordefinierter beliebig-stelliger Typkonstruktor für Tupeltypen.

  Beispiel: (Bool, Int) enthält (True, 5)

- var (*Typvariable*)

  Nötig für *parametrische Polymorphie*.

  Beispiel: Länge einer Liste von Int, Liste von Bool wird gleich berechnet.

⇒ statt 2 verschiedene Implementierungen anzugeben sollte man nur einen Algorithmus schreiben, der sowohl für [Int] als auch für [Bool] verwendet werden kann.
⇒ verwende Typvariablen!

Diese Variablen können mit beliebigen Typen instantiiert werden.

len :: [a] -> [Int]
len [True, False] a wird mit Bool instantiiert.
len [1,2,3] a = Int
len $[[0,1],[\,],[2,3,4]]$ a = [Int]
len [square, square] a = Int -> Int
*cancel*len[True, 5] nicht erlaubt!

parametrische Polymorphie: verwende gleiche Implementierung eines Algorithmus f für Argumente verschiedener Typen. (funktional)

ad-hoc Polymorphie: verwende verschiedene Implementierungen eines Algorithmus f. Welche genommen wird, hängt vom Typen des Algorithmus ab.

Sowohl Haskell als auch Java 5 haben beides.

ident :: a -> a ⇒ a muss überall gleich instantiiert werden.
ident 5 = 5 a = Int
ident True = True a = Bool

app :: [a] -> [a] -> [a] ⇒ alle 3 a's müssen gleich instantiiert werden.
⇒ ~~app[True][5]~~ nicht erlaubt!

app ist in Haskell vordefiniert und heisst dort ++
[0,1] ++ [2,3] = [0,1,2,3]
[True] ++ [False] = [True, False]

Generell: Eine Funktion von Typ type_1 -> type_2
kann auf ein Arument vom Typ type angewendet werden, falls es eine (möglichst allgemeine) Ersetzung (*Unifikation*) σ der Typvariablen in type_1 und in type_2 gibt, so dass $\sigma(\text{type}_1) = \sigma(\text{type}_2)$ ist. Das Resultat hat dann den Typ $\sigma(\text{type}_2)$.

σ: allgemeinster Unifikator von typ_1 und typ_2. (*most general unifier, (MGU)*)

app	[True]	[ ]	
	|	|	
	[Bool]	[b]	← Typen der Argumente
	[a]	[a]	← gewünschte Argumenttypen

Unifikator (σ):
a = Bool
b = Bool

Resultat hat dann den Typ: $\sigma([a])$ ([Bool])

[a] -> [a] -> [a]: Funktionen, die 2 Listen mit Argumenten des gleichen Typs erwarten.

[a] -> [b] -> [a]: Funktionen, die 2 Listen als Argumente erwarten, aber die Argumenttypen können verschieden sein.

5.5.1. Typberechnung

WIRD NOCH HINZUGREFÜGT!

5.5.2. Deklaration neuer Datentypen

topdecl: steht für Deklaration auf oberster Ebene. (Für Funktionen + Datenstrukturen)
decl: Deklaration auf oberster oder lokaler Ebene (where, let) (für Funktionen).
Das Programm ist jetzt eine Folge von topdecl.

Neue algebraische Datentypen werden durch "data" eingeführt. (EBNF-artige Grammatik).

Color : neuer 0-stelliger Typkonstruktor.
Red, Yellow, Green: Datenkonstruktor des Typs Color.

Allgemein auf selbst-definierten Datenstrukturen wie bisher Pattern ($\hat{=}$ Term aus Var. + Datenkons.)Matching genauso möglich.

Zur Angabe von Werten auf dem Bildschirm müssen die Werte in Strings umgewandelt werden. $\Rightarrow$ verwende Funktion "Show"($\hat{=}$"toString" in Java).
Existiert zunächst nicht für selbst-definierte Datentypen.

Kann man selbst schreiben. ($\rightarrow$ ad-hoc Pol. in Haskell) oder automatisch erzeugen lassen. ("deriving Show")

Neuer Datentyp Nats (0-stelliger Typkonstruktor) hat 2 Datenkonstruktoren:

Zero :: Nats
Succ :: Nats -> Nats (Nachfolgerfunktion)

Hinter den Datenonstruktoren wird jeweils angegeben welche Argumenttypen der Da-

tenkonstruktor hat.

data Typ = Constr_1 $\text{type}_1 \ldots$ $\text{type}_n | \ldots$
$\curvearrowright$ Constr_1 :: type_1 -> $\ldots$->type_n -> type.

data List a = $\ldots$
List ist ein neuer 1-steliger Typkonstruktor. $\curvearrowright$ Wenn a ein Typ ist, dann ist auch List a ein Typ.

Datenkonstruktor von Lits a :

Nil :: List a
Cons :: a -> List a -> List a

Vordefiniert in Haskell:
List a $\hat{=}$[a]
Nil $\hat{=}$[ ]
Cons $\hat{=}$:

Durch beliebig-stellige neue Typkonstruktoren kann man selbst parametrisierte Datentypen definieren.

$\underline{\text{typconstr}}$ $\text{var}_1 \ldots$ var_n = $\underline{\text{Constr}}$ $\text{typ}_1 \ldots$ $\text{typ}_n | \ldots$
$\text{var}_1 \ldots$ var_n : paarweise verschieden
$\text{typ}_1 \ldots$ typ_n : enthalten keine Typvariablen ausser $\text{var}_1 \ldots$ var_n.

5.6. Funktionale Programmiertechniken

data $\underbrace{List}_{\text{neuer 1stelliger Typkons.}}$ a = $\underbrace{Nil \mid Cons}_{\text{List a hat 2 Datenkons.}}$ $\underbrace{a\ (Lit\ a)}_{\textit{Typen der Arg. von Cons}}$

Nil :: List a
Cons :: a -> List a -> List a

Nil repräsentiert die leere Liste

Cons 1 Nil repräsentiert die Liste, die nur 1 enthält

Cons 1 (Cons 2 Nil)...

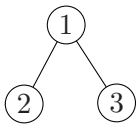
Binärbaüme

Die Knoten enthalten Werte vom Typ a

dat $\underbrace{Tree}_{\textit{Typ d. Binb. mit Knoten v. Typ a}}$ a = Leaf a | Node a (Tree a) (Tree a)

Leaf :: a -> Tree a
Node :: a -> Tree a -> Tree a -> Tree a

1 Wird repräsentiert als Leaf 1



Wird repräsentiert als Node 1 (Leaf 2) (Leaf 3)

5.6.1. Charakteristische funktionale Programmierung

- Funktionen höherer Ordnung
- Unendliche Datenstrukturen

Funktionen höherer Ordnung:
hat als Argument oder als Resultat selbst wieder Funktionen.

square :: Int -> Int	ist Funktion 1.Ordnung
plus :: Int -> (Int -> Int)	ist Funktion höherer Ordnung. Das Resultat ist eine Funktion.
plus 5	ist eine Funktion vom Typ Int -> Int

Funktionskomposition: $\underbrace{f \circ g}_{\textit{Fkt, die erst g ausf. und dann f}}$

⇒ " ∘ " ist Funktion höherer Ornung.

comp $\underbrace{f}_{b->c}\ \underbrace{g}_{a->b}$ entspricht f∘g Typ: a -> b

comp :: $\underbrace{(b->c)}_{Argument}\ \underbrace{((a->b)->(a->c))}_{Resultat}$

Funktion höherer Ordnung sowohl Argument als auch Resultat sind Funktionen.

$\underbrace{comp\ half\ square}_{Fkt.\ die\ x\ abbildet\ auf\ \frac{x^2}{2}}$ 4=8

comp (\x -> x + 1)
squarecomp ist vordefiniert in Haskell heisst dort'.'(half.square) 5 = 26

Überführung von plus :: (Int,Int) -> Int
in plus :: Int -> Int -> Int

Wenn plus die 2. Definition hat (plus :: Int -> Int -> Int) dann ergibt $\underbrace{uncurry\ plus}_{(Int->Int)->Int}$ (2,3)
= 5

Es gilt: curry(uncurry g) = g
uncurry(curry f) = f

Haupteinsatz von Funktionen höherer Ordnung:

- Implementiere bestimmtes Algorithmenschema als Funktion höherer Ordnung.
- Bei konkreten Algorithmen: verwende Funktionen höherer Ordnung, die das benötigte Schema realisieren.

⇒ Bessere Strukturierung von Programmen

2 Beispiele für typische Algorithmenschemata, die durch Funktionen höherer Ordnung (map,filter,...) dargestellt werden können?

succ :: Int -> Int
succ = plus 1 (oder: succ x = x+1)

succlist[1,2,3] = [succ1,succ2,succ3]
[2,3,4]

sqrtlist[4,9,16] = [sqrt4,sqrt9,sqrt16]
[2,3,4]

succlist und sqrtlist verwenden das gleiche Algorithmenschema:

- wenn die Liste leer ist, gib leere Liste zurück
- ansonsten wende eine Funktion auf 1. Listenelement an und starte den rekursiven Aufruf.

ZIEL: Abstahiere von den Unterschieden zwischen succlist und sqrtlist und definiere einen allgemeinen Algorithmus

1. Abstrahiere von Int bzw. Float ⇒ ersetze Typ durch Typvariable a (passende Polymorphie nötig)
2. Abstrahiere von succ bzw. sqrt ⇒ ersetze Funktion durch Variable g (Funktion höherer Ordnung nötig)

Da g eine beliebige Funktion ist, muss sie ein weiteres Eingabeargument :
(Vom Typ a ->b) der Funktion f sein.

f = map g (Funktion die eine Liste nimmt und g auf jedes Listenelement anwendet).

Neue Definition von succlist:

succlist :: [Int] -> [Int]
succlist l = map succ l (Fkt., die eine Liste nimmt und succ auf jedes Listenelement anwendet)

sqrtlist :: [Float] ->[Float]
sqrtlist l = map sqrt l

Analoge "map"-Funktion kann man natürlich auch für selbstdefinierte Datentypen schreiben.

dropEven [1,2,3,4] = [1,3]
dropUpper "GmbH" = "mb"

Abstrahiere von den Unterschieden zwischen dropEven und dropUpper:

1. Ersetze Int/Char durch Typvariable a
2. Ersetze odd/isLower durch Variable g

Die Funktion g ist eigentlich ein weiterer Eingabetyp von f.

Typ a -> Bool

⇒ Funktion filter (Funktion höherer Ordnung)

Analoge Funktionen höherer Ordnung auf eigene Datenstrukturen möglich.

5.6.2. Unendliche Datenobjekte

Verwendbar aufgrund der nicht-strikten Auswertungsstrategie von Haskell.

- generell leftmost-outermost Auswertung
- bei vordefinierten arithmetischen Operationen müssen erst alle Argumente ausgewertet werden.
  (z.B. +, -, *, \,. . . ,>, <,. . .)
- bei Pattern Matching werden Argumente nur soweit ausgerechnet, bis man entscheiden kann, welcher Pattern matcht.

infinity repräsentiert die Zahl ∞ (Anwendung von infinity terminert nicht).

mult infinity 0	um zu entscheiden , ob 1.mult-Gleichung
mult (infinity +1) 0	anwendbar ist, muss infinity solange
mult (infinity + 2) 0	ausgewertet werden, bis man entscheiden kann,
⋮	ob der Pattern 0 matcht.

Unendliche Listen

from x =[x,x+1,x+2,. . .]
Auswertung von from 0 terminiert nicht.

take 1 $[x_1, x_2, \ldots, x_n, x_{n+1}, \ldots] = [x_1, \ldots, x_n]$

Beispiel: Sieb des Eratothenes

1. Erstelle Liste aller natürlichen Zahlen ab 2 from2
2. Markiere die erste unmarkierte Zahl in der Liste
3. Streiche alle Vielfachen der letzten markierten Zahl
4. Zurück zu Schritt 2

$[\underline{2}, \underline{3}, \not{4}, 5, \not{6}, 7, \not{8}, 9, \not{10}, \ldots]$

drop_mult x xs : Streiche aus der Liste xs alle Vielfachen von x

(\y -> mod y x /=0) bildet y auf True ab, falls y kein Vielfaches von x ist.

Auswertung von primus : [2,3,5,7,11,. . .] terminiert nicht
take 5 primus = [2,3,5,7,11] terminiert.

Teil III.

Logische Programmierung

6. Prolog

6.1. Grundkonzepte der logischen Programmierung

Der Programmierer beschreibt logische Zusammenhänge eines zu lösenden Problems:
"*Wissensbasis*"
Es ist kein Wissen über maschinennahe Details nötig.

Beispiel: Wissensbasis über Verwandschafts-Beziehungen.

- : verheirartet
↓ : Kinder

Ziel: Formalisiere Wissensbasis in Prolog. Dann kann man dem Programm Fragen stellen. z.B:

- Wer ist die Mutter von susanne ?
- Welche Kinder hat renate ?

Computer $\hat{=}$ Inferenzmaschine
Rechner $\hat{=}$ Beweisen von Aussagen

($\Rightarrow$ Expertensystem)

6.1.1. Wissensbasis

- Zur Darstellung wir die Sprache der Prädikatenlogik benutzt. (Prolog $\hat{=}$ Programmierung in Logic).
- Wissensbasis betseht aus *Formeln*, genauer *Klauseln* (spezielle Formeln) der Prädikatenlogik.
- 2 Arten von Klauseln:
  - Fakten (Aussagen über Objekte)
  - Regeln (erlauben, aus bekannten Fakten neue Fakten herzuleiten)
- Fakten beschreiben Relationen zwischen Objekte.
  Relationen sind gerichtet.

- Fakt: Name der $\underbrace{\text{Eigenschaft}}_{\text{Prädikatssymbol}}$ /Relation (Objekt die diese Eigenschaft haben).

  mutterVon(monika, karin). nächster Fakt in nächster Zeile.

- $\underbrace{Eigenschaften}_{mutterVon}$ und $\underbrace{Objekte}_{monika}$ beginnen mit Kleinbuchstaben.

- Jedes Prädikatssymbol hat eine feste Stelligkeit:
  mutterVon : Stelligkeit 2
  weiblich : Stelligkeit 1

- Kommentar:
  % ... Zeilenende
  /∗ · · · ∗/

6.1.2. Ausführen des Prolog-Programms

Stelle Anfrage $\underbrace{(Goals)}_{Queries}$. Prolog versucht, die Anfrage aufgrund seines Wissens in der Wissensbasis zu beweisen.

- Starte Prolog ("pl")
- Lade Prolog datei.pl : consult(datei). oder [datei].
  (datei mit Kleinbuchstabe!)
- Stelle Anfrage: ?−maennlich(gerd)

  "Ist gert maennlich ?"

  d.h. Versuche, die Aussage "maennlich(gert)" zu beweisen.

Closed World Assumption
Prolog nimmt an, dass alle Aussagen falsch sind, die es nicht aus seiner Wissensbasis heraus beweisen kann.

Anfragen/Fakten, die Variablen enthalten

Variablen beginnen mit einem Grossbuchstaben oder mit einem Unterstrich (X,Y,. . . ,_ . . .)

Variablen in der Wissensbasis stehen für alle möglichen Belegungen.

"mensch(X)" bedeutet: alle Objekte sind menschen.

?−mensch(5)
YES

- Gleiche Variablen in gleichen Fakten bedeuten das Gleiche:

  mag(X,Y). $\hat{=}$ Jeder mag jeden
  mag(X,X). $\hat{=}$ Jeder mag nur sich selbst.

- Gleiche Variablen in verschiedenen Fakten haben nichts miteinander zu tun.

Variablen in Anfragen

?−mutterVon(X,susanne).
$\hat{=}$ existiert eine Belegung der Variablen X, so dass "mutterVon(X,susanne)" wahr ist?

Variablen in der Wissensbasis sind allquantifiziert. (bedeutet: Aussagen sind wahr für alle Instanzen der Variablen).
Variablen in der Anfrage: existenzquantifiziert (bedeutet: Gibt es eine Belegung der Variablen, so dass die Anfrage wahr ist?)

?− mutterVon(renate,Y).
$\hat{=}$ Welche Kinder hat Renate?

⇒ es gibt keine festgelegten Ein- Ausgabepositionen.
Was Ein-/Ausgabe ist, hängt von der Anfrage ab.

Beispiel:
2. Argument von mutterVon Eingabe
1. Argument von mutterVon Ausgabe. (oder Umgekehrt).

Eingaben: die Ausdrücke ohne Variablen in der Aussage
Ausgabe: die Ausdrücke mit Variablen in der Aussage.

Bei mehreren Lösungen:
Prolog arbeitet die Wissensbasis von oben nach unten ab und liefert (zunächst) die erste gefundene Lösung. Falls nach der ersten gefundenen Lösung ";" eingegeben wird ⇒ dann sucht Prolog nach der nächsten Lösung.

?−mensch(X,Y). (Hier sind beide Argumente Ausgaben).

X = monika Y = karin;
X = monika Y = klaus;
⋮ ⋮

Reihenfolge liegt an Wissensbasis:
Klauseln werden von oben nach unten abgearbeitet.

?−mensch(Y) (Wissensbasis enthält mensch(X))

Y = Z
neue Variable, kann auch anders heissen, z.B. _G192
⇒ Prolog berechnet die allgemeinste Lösungen. Variablen in der Lösung dürfen beliebig instantiiert werden.

Kombination von Fragen

Fragen können die gleichen Variablen enthalten.

[, $\hat{=}$ UND]
[; $\hat{=}$ ODER]

Gibt es ein F, so das gert mit F verheiratet ist und F die Mutter von susanne ist?
$\hat{=}$ Ist gerd der Vater von susanne?

Abarbeitung von Anfragen durch Aufbau eines *SLD-Baums*:

?−verheiratet	(gerd,F),	mutterVon(F,susanne)
	| F = renate	
?−mutterVon	(renate,susanne)	
	|	
	□	← Leere Klausel, bedeutet das die ursprüngliche Anfrage bewiesen wird.

- Klauseln der Programme werden von oben nach unten abgearbeitet.
- $\underbrace{\textit{Beweisziele}}_{\textit{Literale}}$ werden von links nach rechts abgearbeitet.
  Reihenfolge der Klauseln im Programm und der Literale in Klauseln beeinflusst das Prolog-Verhalten!

  Prolog baut einen SLD-Baum auf, bis die erste □ gefunden wird. Dann wird ausgegeben, wie die Variable der Anfrage der Variablen belegt werden musste, um zu □ zu kommen. Bei ";" wird der Baum anschliessend weiter aufgebaut.

  "Wer ist Oma von aline?"
  ?− mutterVon(Oma,Mama); mutterVon(Mama, aline)

Oma = monika	Oma = monika	Oma = renate
Mama = karin	Mama = klaus	Mama = susanne.
↯	↯	|
		□

→ Reihenfolge der Literale beeinflussen die Effizienz (und Terminierung).

Programme mit Regeln

Dienen dazu, um aus bekanntem Wissen neues herzuleiten.

Beispiel: Vater-Beziehung
Eine Person V ist Vater von Kind K falls er mit Frau F verheiratet ist und F Mutter von K ist.

":-" bedeutet "falls"

Regeln sind "*wenn-dann*" Beziehungen:

Wenn die $\underbrace{\textit{Aussagen rechts}}_{RUMPF}$ V ":-" wahr sind, dann ist auch die $\underbrace{\textit{Aussage links}}_{KOPF}$ davon wahr.

p :- q, r "p gilt, falls q und r gelten"

?− vaterVon(gerd,susanne).
  | V = gerd
  | K = susanne

Suche erste Regel/Fakt, so dass der Kopf dre momentanen Anfrage entspricht. Dann ersetze Literal aus Anfrage durch Rumpf der Regel.

?−verheiratet(gerd,F)., mutterVon(F,susanne).
  | F = renate
?−mutterVon(renate,susanne).
  |
  □

Es wird nur die Belegung der Variablen aus der Anfrage ausgegeben.

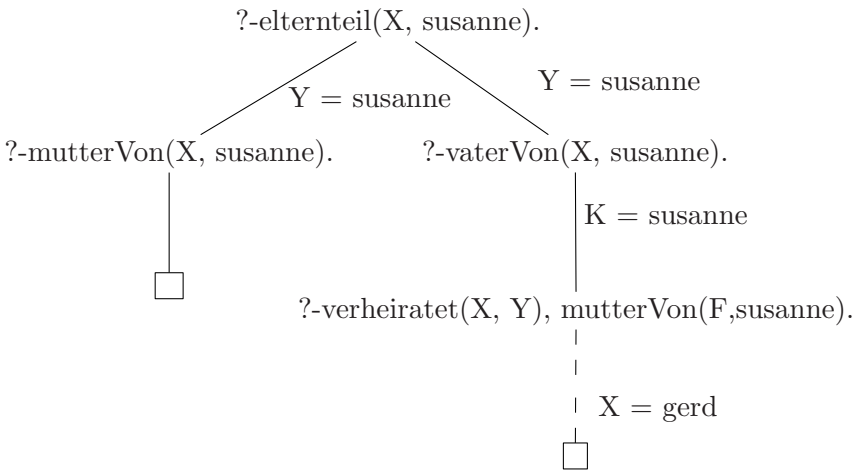
Zurücksetzten im SLD-Baum *("Backtracking")* falls man Blätter erreicht, die nicht bewiesen werden können oder wenn der Benutzter mit ";" nach weiteren Lösungen sucht.

Beispiel:

- *Konjunktion*: zur Definition eines Prädikats.("und")
- *Disjunktion*: mehrere Klauseln für dasselbe Prädikat.("oder")

SLD-Baum/Beweisbaum:

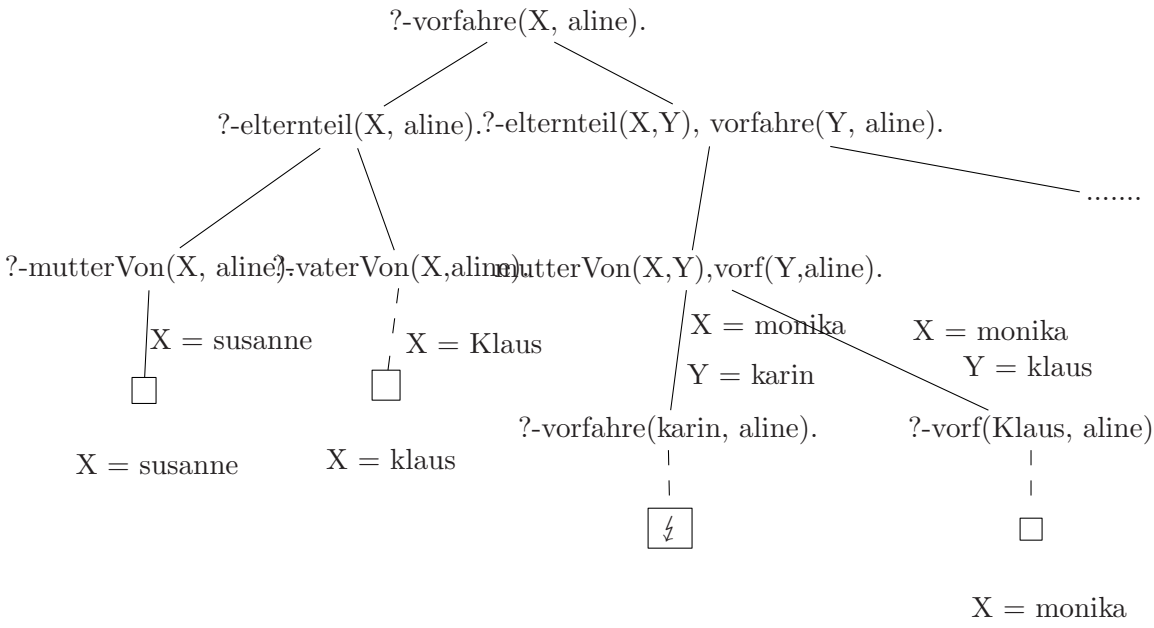
Suche erste Klauselkopf, der zum 1. Bewseisziel "passt":
Instantiiere die Variable(n) im Klauselkopf. Beweiziel gezeigt (Verifikation).



Das Programm findet erst die Mütter dann die Väter. ⇒ Reihenfolge der Programmklauseln beeinflusst die Reihenfolge der Lösungen.

6.1.3. Rekursive Regeln

Beispiel:
vorfahre(V, X). falls elternteil(V, x) oder falls V elternteil von Y ist und Y ist vorfahre von X.



- Prolog arbeitet diesen Baum mit *Tiefensuche* ab.
- Bei jedem erfolgreichen Pfed gibt Prolog die *Antwortsubstitutionen* aus: benötigte Belegung der Variablen aus der Anfrage. (gleiche Lösung kann mehrmals gefunden werden).

6.1.4. Vergleich mit Haskell

Haskell	*Prolog*
- Werte Grundterme aus (Terme oder Variablen)	- Anfragen (bzw. Beweisziele) können Variablen enthalten
- nur die Variablen aus den Gleichungen (d.h. aus dem Programm) werden instantiiert.	- Variablen aus Beweisziel und aus Prolog-Klauseln werden beide instantiiert.

Matching	*Unifikation*
s = $\sigma(t)$	$\sigma(s) = \sigma(t)$
Existiert eine Substitution σ, so dass damit t den Term s *matcht*.	Existiert eine Substitution σ, so dass damit s und t *unifizieren*

X matcht f(X), aber sie unifizieren nicht.
f(X, b) matcht f(a, Y) nicht, aber sie unifizieren: X = a, Y = b.

Horristik:

- erst die nicht rekursiven Klauseln, dann die rekursive Klauseln

- im Klauselrumpf sollte der rekursive Aufruf möglich spät kommen.

6.2. Syntax von Prolog

- *Programm*: Folge von Klauseln (hier sind es spezielle Klauseln, sagenannte "Hornklauseln".) Nach jeder Klausel kommt ein Zeilenumbruch / Leerzeichen.
- *Klausel*: Faktum / Regel

  Faktum: Literal.
  Regel: $\underbrace{\text{Literal}}_{\text{Kopf}}$:- $\underbrace{Lit, Lit, \ldots Lit.}_{\text{Rumpf}}$
- *Literal*: weibleich(monika), vaterVon(V,K),...
  Prädikat($Term_1, \ldots, Term_n$*): bei n-stelliges Prädikatssymbol*
  Prädikat: bei 0-stelliges Pradikatssymbol
- *Prädikatssymbol*
  - beschreibt Relationen / Funktionen von Objekte, die True / False liefert
  - für n-stellige Prädikatssymbole p schreibt man auch oft p/n (weiblich/1, vaterVon/2,...)
  - Pädikate mit gleichem Namen aber verschiedener Stelligkeit sind verschieden. p/0, p/2, p/5 haben nichts miteinander zu tun.
  - Prädikatssymbole beginnen mit Kleinbuchstaben.
- *Terme*
  - *Variablen*: beginnen mit Grossbuchstaben oder mit _.
    '_' ist die anonyme Variable:
    → ihre Belegung interessiert nicht
    → falls sie mehrmals vorkommt, darf sie jedes mal unterschiedlich instantiiert werden.

    istEhemann(Person) :- verheiratet(Person,_).
    Falls die Variable nur einmal in der Klausel (im Rumpf) auftritt, dann nimmt man üblicherweise _.

    verheiratet(_, _). bedeutet, dass jeder mit jedem verheiratet ist.
  - *Funktionen(*$Term_1, \ldots, Term_n$*)* für n-stellige Funktionssymbole, n $\geq$ 1.

    Funktion für 0-stelliges Funktionssymbol $\Rightarrow$ *Konstante*

    Trifft f mit mehreren Stelligkeiten auf, dann sind diese f's verschieden. (f/0, f/1, f/2,...)

    Mehrstellige Funktionssymbole:

geboren(monika, $\underbrace{25, 7, 1972}$)
Objekte die eigentl. zusammengehören

geboren sollte eigentlich keine 4-stellige Relation sein, sondern eine 2-stellige (zwischen Person und Geburtsdatum).
⇒ Verwende 3-stelliges Funktionssymbol "datum"
datum(25, 7, 1972) ist ein Term.

$$\underbrace{\underbrace{geboren}_{Pradsymb}(\underbrace{monika}_{Term}, \underbrace{\underbrace{datum}_{Fktsymb}(25, 7, 1972)}_{Term})}_{Literal}.$$

?-geboren(X, datum(_,7,). bedeutet:
Wer wurde im Juli geboren?

Funktionssymbole beginnen mit Kleinbuchstaben. (aber es gibt noch mehr: 0,1,2,..., (sind Konstanten)$\cdots$ ⇒ betrifft vorallem voreingebauten Datenstruktur).

6.2.1. Definition neuer Datenstrukturen

(analog zu Haskell)

Datenobjekte müssen durch Terme repräsentiert werden. ⇒ verwende dafür geeignete Funktionssymbole ("Datenkonstruktor" in Haskell)

z.B. repräsentiere N durch
zero/0 (zero $\hat{=}$ 0)
succ/1 (succ(zero) $\hat{=}$ 1)

Haskell: Datenkonstruktoren werden nicht ausgewertet,
sonstige Funktionssymbole werden ausgewertet.
Prolog: Funktionssymble werden nicht ausgewertet.
Prädikatsymbole werden "ausgewertet".

Additionsbeispiel

Addition in Prolog: add
Keine 2-stellige Fkt add, sondern 3-selliges Pradikatsymbol add.
add(X,Y,Z) bedeutet: Die Addition X und Y ergibt Z.
add(zero,succ(zero),succ(zero)) TRUE

add(zero,succ(zero),zero) FALSE

A: X + 0 = X
B: X + succ(Y) = succ(Z), falls X + Y = Z

Berechne 1 + 1

?-add(s(zero),s(zero),U)
| X = succ(zero)
| Y = zero
| U = succ(Z)
?-add(s(zero),zero, Z)
| X' = succ(zero)
| Z = succ(zero)
| □
Antwortsubstitution U = succ(succ(zero))

Das Programm addiert falls das erste und zweite Argument von add in der Anfrage Grundterme sind (ohne Variablen).

Berechne 2 - 1:

?-add(s(zero),V,s(s(zero)))
| X = s(zero)
| V = s(Y)
| Z = s(zero)
?-add(s(zero),Y,s(zero))
| X' = s(zero)
| Y = zero
| □
Antwortsubstitution: V = s(zero).

Gibt man erstes und drittes Argument vor, subtrahiert das Programm.

Gibt X + Y = 0 für ein X > 0 ?
?-add(X,Y,s(s(zero))).
Bedingung: für welche Zahlen X,Y gilt X + Y = 2? (3 Lösungen).

Multiplikationsbeispiel

mult(t_1,t_2,t_3) wahr falls t_1 * t_2 = t_3.

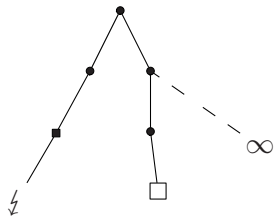
A: X * 0 = 0
B: X * s(Y) = Z, falls X * Y = U und X + U = Z

*Berechne 1 * 1*:

?-mult(s(zero),s(zero),U)
| X = s(zero)
| Y = zero
| Z = U
?-mult(s(zero),zero,U'), add(s(zero),U',U)
| U' = zero
?-add(s(zero),zero,U)
| U = s(zero)
| □
Antwortsubstitution: U = s(zero)

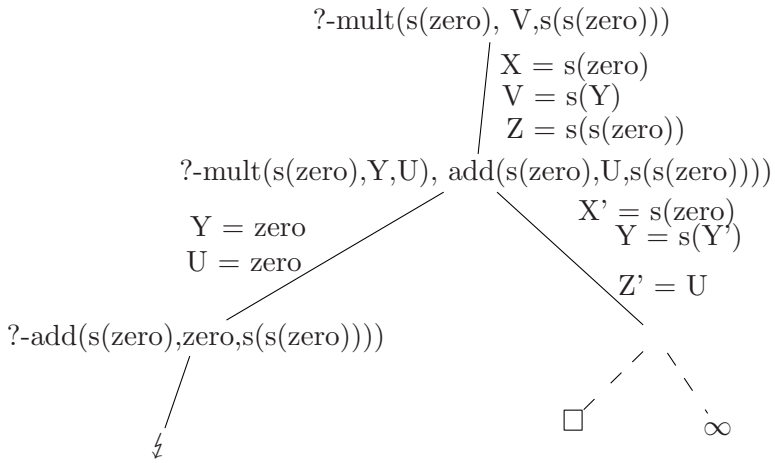
Generell:

- Fakten von rekursiven Regeln (so besseres Termverhalten)
- rekursive Aufrtufe in Regeln:
  manchmal soweit nach hinten wie möglich, aber manchmal auch nicht.
  → würde man die Literale in der rekursiven mult-Regel vertauschen, wäre der Beweisbaum für mult(s(zero),s(zero),U) unendlich.



⇒ Lösung wird gefunden, aber danach (bei ";") terminiert Prolog nicht mehr.
Will man das Programm zum dividieren benutzen, sollte man die Literale lieber vertauschen.

Berechne 2 / 1:



Um die geeignete Reihenfolge der Literale zu bekommen, überlege was beabsichtigter Input + Output ist.

$$\text{mult}(\underbrace{X}_{I}, \underbrace{s(Y)}_{I}, \underbrace{Z}_{O}) :- \; mult(\underbrace{X}_{I}, \underbrace{Y}_{I}, \underbrace{U}_{O}), add(\underbrace{X}_{I}, \underbrace{U}_{I}, \underbrace{Z}_{O})$$

Prolog ist eine untypisierte Sprache (anders als Java und Haskell)

↷ jedes Prädikatsymbol/Funktionssymbol ist auf jeden Term anwendbar, nicht nur auf Terme von geeigneten Typen!

?-mult(monika, zero, zero).
YES.

Verantwortung des Programmierers und Benutzer: nur "sinnvolle" Terme bilden.

6.2.2. Weitere Datenstruktur: Listen

2 Funktionssymbole als Konstruktoren:
nil/0 (leere Liste)
const/2 (Einfügen vorne in die Liste)

cons(zero, cons(succ(zero), nil)) repräsentiert [0,1]

Statt 1-stellige Funktion len jetzt jetzt 2-stelliges Prädikat:
len(t_1,t_2) wahr, falls die Liste t_1 die Länge t_2 hat.

Antwort auf

?-len(L, s(s(zero)))enthält Variablen A und B.
Prolog liefert die allgemeinste Lösung.↷ bei jeder Belegung von A und B ist dies eine korrekte Lösung. (A,bzw B heissen meisst:_G192,...
Datenstruktur für Zahlen (später) und Listen ist in Prolog vordefiniert.

Listen:

Satt nil [ ] (wie in Haskell)
Satt cons . (wie : in Haskell)

.(1,.(2,.3,([ ])))
Hierfür existiert eine Kurzschreibweise [1,2,3]
Weitere Kutzschreibweise: [$t_1 \dots t_n$|l] steht für die Liste, die aus der Liste l entsteht, indem man vorne die Elemente $t_1 \dots t_n$ einfügt.

[1,2,3] = [1,|[2,3]]
= [1,2|[3]
= [1,2,3|[ ]]

Prolog		*Haskell*
[Kopf|Rest]	$\hat{=}$	Kopf : Rest

6.3. Rechnen in Prolog

6.3.1. Unifikation und Resolution

Konzepte für das Abarbeiten in Prolog. Bei Abarbeitung eine Anfrage: entscheide ob das 1. Literal der Anfrage einem Klauselkopf (bzw Faktum) entspricht (*unifiziert*(belege die Variable in der Anfrage und Klausel so, dass das Literal de Anfrage = dem Literal der Klausel)).

σ(add(s(zero),s(zero),U)) = σ(add(X,s(Y),s(Z)))?
X = s(zero)
Y = zero
U = s(zero)

σ = (X = s(zero),Y = zero, U = s(Z))

Substitution: σ: Variable $\rightarrow$ Terme
wobei σ nur endlich viele Variablen belegt.
(σ(X) = X für unendlich viele Variablen
σ(X) $\neq$ X für endlich viele Variablen)

Üblicherweise gibt es mehrere Unifikatoren:
$\Rightarrow$ oberste Unifikator ist der allgemeinste Unifikator (most general unifier *MGU*) denn alle anderen Unifikatoren sind spezialfälle davon.

Bei unifizierbare Literale / Terme verwenden wir den MGU.

Beispiel: μ\{X = s(zero), Y = zero, U = s(Z)\} ist MGU.
ALLE weiteren Unifikatoren enstehen durch weitere Instantiierungen von Z (oder von anderen Variablen).

$\tau_1 = \mu = X = s(zero), Y = zero, U = s(zero), Z = zero$ $\quad$ $\tau_1 = \{z = zero\}$
$\tau_2 = \mu$ $\quad$ $\tau_2 = \{Zs(W)\}$

Mit einem MGU kann man die (meist unendliche) Menge aller Unifikatoren repräsentieren.

Für MGU μ muss also gelten:
Für alle Unifikatoren σ existiert eine Susbtitution τ, so dass $\sigma(X) = \tau(\mu(X))$ für alle Variablen X in den zu unifizierenden Literalen / Termen.

Sätze:

- Falls 2 Terme / Literale unifizierbar sind, dann haben sie auch einen MGU.
- MGU ist eindeutig bis auf Variablenumbenennung (Subst, die paarweise verschie-

dene Variablen (die zu unif. Terme / Literale) durch paarweise verschiedenen Variablen ersetzt.)

Weiterer MGU im Beispiel:

{X = s(zero), Y = zero, U = s(V), Z = V}
(ersetze Z durch neue Variable V)
Dieser MGU würde von Prolog berechnet um Konflikte mit gleichen Variablennamen zu vermeiden.

6.3.2. Algorithmus zur Berechnung des MGU's

Robinson 1965 (leicht aber unheffizient). Es existieren bessere (oder komplexere) Unifikationsalgorithmen.

ZIEL: Unifiziere 2 Ausdrücke s und t.

1. Fall:
   s = t, beides Variablen ⇒ MGU ist leere Susbstitution.
   Beispiel: X und X

2. Fall:
   s ist Variable, t enthählt Variable s nicht ⇒ MGU:{s = t}
   Beispiel: X und succ(Y) MGU:{X = succ(Y)}
   X und succ(X) nicht unifizierbar.

   OCCURE FAILURE
   (Unifikation schlägt fehl, falls man eine Variable mit einem Ausdruck unifizieren will, der diese Variable enthählt und nicht identisch zu dieser Variable ist.)

   OCCUR CHECK überprüft, ob die Variable s im Ausdruck t auftritt. (Occur Check fehlt in der Prolog-Implementierung aus Effizienzgründen.

3. Fall:
   Wie 2. Fall, nur mit s und t vertauscht.
   Beispiel: succ(Y) und X ⇒ MGU = {X = succ(Y)}

4. Fall:
   s uns t beginnen mit einem Prädikatsymbol / Funktionssymbol.

   Falls s = f(...), t = g(...)
   ⇒ s und t sind nicht unifizierbar.
   CLASH FAILURE
   Falls s = f(s_1,...,s_n), t = f(t_1,...,t_n), dann untersuche die Argumente auf Unifizierbarkeit.
   Beispiel:

s = f(,Z,s(s(X))) t = f(s(Y),X,Z)
σ_1 = MGU(X, s(Y)) = {X = s(Y)}
σ_2 = MGU(σ_1(Z),σ1(X)) = MGU(Z, s(Y)) = {Z = succ(Y)}
die bereits gefundene Teil-Instantiierung aus Arg 1 muss beim Unifizieren von Arg 2 benutzt werden.

σ_3 = MGU(σ_2(σ_1(succ(succ(W))),σ2(σ_1(Z))))
die bereits gefundenen Teil-Instantiierungen aus Arg 1 und 2 muss beim Unifizieren von Arg 3 benutzt werden.
= MGU(s(s(W)),s(Y))
= {Y = s(W)}

Lösung : $\sigma = \sigma_3 \circ \sigma_2 \circ \sigma_1$ = {X = s(s(W)), Z = s(s(W)), Y = s(W)}

Beweisverfahren von Prolog: *SLOVE* (benutzt MGU)

In Schritt 2: Löse das erste Literal G_1 aus der Anfragemit der ersten noch nicht probierten Programmklausel (Die Variablen darin so umbenennen, dass sie von den Variablen in Anfrage G_1 verschieden sind).

In Schritt 3: $\mu(G_1) = \mu(H)$

- Wende μ auf Anfrage an:
  ?-$\underbrace{\mu(G_1)}_{\mu(H)}$,...,$\mu(G_m)$
- Ersetze Klauselkopf μ(H) durch Klauselrumpf $\mu(B_1), \ldots, \mu(B_n)$
  ?-$\mu(B_1), \ldots, \mu(B_n)$,$\mu(G_2), \ldots, \mu(B_m)$
  Fakten = Regeln mit leerem Rumpf (H :- .)

RESOLUTION:
Wenn aus $B_1, \ldots, B_n$, die Aussage H folgt und wenn man H,G_1,...,G_n beweisen will, dann reicht es dafür B_1,...,B_n,G_2,...,B_m zu beweisen. (entsprechende Erweiterung für die Behandlung von Variablen durch Unifikation)
Antwortsubstitution: Komposition der allgemeinsten Unifikatoren, die bei der Abarbeitung von Slove benutzt wurden.

Angegeben wird nur der Teil der Antwortsubstitution, die der Variable aus der Anfragen betrifft.

Beweisbaum

1. Antwortsubstitution Z = susanne
2. Antwortsubstitution Z = aline

Falls man nochmal ";" eingibt, dann ergibt sich "NO".
⇒ Beweisbaum ist endlich in diesem Beispiel. Prolog durchsucht den Beweisbaum mit Tiefensuche.
Falls man Klauseln / Literale vertauscht, ändert sich eventuell das Terminierungsverhalten.

Beispiel: Vertauschen von Literale: Prolog terminiert schon bei der Suche nach der 1. Lösung nicht.

Generelles Problem: "Unentscheidbarkeit der Prädikatenlogik". D.h. es existiert kein automatisches Verfahren, das bei jeder Anfrage terminiert und feststellt, ob die Anfrage aus der Programmklausel folgt oder nicht

6.3.3. Unifikation

Kann man in prolog sehr leicht implementieren.

gleich(X,X).
gleich(t_1,t_2) ist wahr genau dann wenn es einen MGU μ gibt mit $\mu(t_1) = \mu(X) = \mu(t_2)$, d.h. genau dann wenn t_1 und t_2 unifizierbar sind.

Ist in Prolog vordefiniert und heisst "="
$\underbrace{(a, L)}_{[a|L]}$ = [X,b|K].
X = a, $\underbrace{L = [b|K]}_{L=[b|S],K=S}$
?-X = succ(X).

X = s(s(...)) Term aus ∞ vielen succs.
⇒ Prolog verzichtet auf Occur Check
⇒ ∞ Terme können entstehen.

6.3.4. Eingebaute Datenstruktur für Zahlen

Eigenes Prädikat "*is*" für Gleichheit.

+, -, *,... sind zunächst nur syntaktische Funktionssymbole.
2 + 5 und 7 sind verschiedene Terme! (unifiziert nicht)
X is 2 + 5 (wird zunächst ausgewertet: 7)

len-Algorithmus:
len([1, 2], 2).

Falls man "is" durch "=" ersetzt:
$\text{len}([1,2], 0+1+1)$.

Weitere vordefinierte Prädikate $<, >, >=, =<$
Hier müssen zum Zeitpunkt der Auswertung beide Argumente vollständig instantiiert sein. Dann werden beide ausgewertet und danach verglichen.

$2 + 5 < 1 + 7$ ist wahr.