

Prinzipien von Programmiersprachen  
Vorlesung  
Wintersemester 2007, 2008, 2009

Johannes Waldmann, HTWK Leipzig

4. Februar 2010

# Inhalt

(aus Sebesta: Concepts of Programming Languages)

- ▶ (3) Beschreibung von Syntax und Semantik
- ▶ (5) Namen, Bindungen, Sichtbarkeiten
- ▶ (6) Datentypen
- ▶ (7) Ausdrücke und Zuweisungen, (8) Anweisungen und Ablaufsteuerung, (9) Unterprogramme
- ▶ (11) Abstrakte Datentypen, (12) Objektorientierung
- ▶ (13) Nebenläufigkeit, (14) Ausnahmenbehandlung
- ▶ (15) Funktionale Programmierung, eingebettete domainspezifische Sprachen

# Organisation

- ▶ Vorlesung
  - ▶ mittwochs 13:45–15:15, u : Li 207, g: G 121
- ▶ Übungen (alle Z 423)
  - ▶ Do (u) 13:45 + Fr (g) 11:15
  - ▶ Fr (u) 11:15 + Fr (g) 9:30
  - ▶ Mi (u) 15:30 + Fr (g) 13:45

Übungsgruppe wählen: <https://autotool.imn.htwk-leipzig.de/cgi-bin/Super.cgi>

# Literatur

- ▶ <http://www.imn.htwk-leipzig.de/~waldmann/edu/ws09/pps/folien/pps/>
- ▶ **Robert W. Sebesta: Concepts of Programming Languages, Addison-Wesley 2004**  
siehe auch <ftp://ftp.aw.com/cseng/authors/sebesta/concepts7e>

# Übungen

1. Anwendungsgebiete von Programmiersprachen, wesentliche Vertreter

zu Skriptsprachen: finde die Anzahl der "\*.java"-Dateien unter \$HOME/workspace, die den Bezeichner String enthalten. (Benutze eine Pipe aus drei Unix-Kommandos.)

Lösungen:

```
find workspace/ -name "*.java" | xargs grep -l String
find workspace/ -name "*.java" -exec grep -l String
```

2. Maschinenmodelle (Bsp: Register, Turing, Stack, Funktion)

funktionales Programmieren in Haskell (siehe

<http://www.haskell.org/>)

```
bash
```

```
export PATH=/home/waldmann/built/bin:$PATH
```

```
ghci
```

```
:set +t
```

```
length $ takeWhile (== '0') $ reverse $ show $ prod
```

Kellermaschine in PostScript.

# Ausführungsarten

Anweisungen der Quellsprache werden in Anweisungen der Zielsprache übersetzt. (Bsp: Quelle: C, Ziel: Prozessor)

- ▶ interpretieren: jeden einzelnen Befehl: erst übersetzen, dann ausführen.  
Bsp: Skriptsprachen
- ▶ compilieren: erst gesamtes Programm übersetzen, dann Resultat ausführen  
Bsp: C, Fortran
- ▶ Mischformen: nach Zwischensprache compilieren, diese dann interpretieren.  
Bsp: Pascal (P-Code), Java (Bytecode), C# (CIL)

# Struktur eines Übersetzters

- ▶ (Quelltext, Folge von Zeichen)
- ▶ lexikalische Analyse ( $\rightarrow$  Folge von Token)
- ▶ syntaktische Analyse ( $\rightarrow$  Baum)
- ▶ semantische Analyse ( $\rightarrow$  annotierter Baum)
- ▶ Zwischencode-Erzeugung ( $\rightarrow$  Befehlsfolge)
- ▶ Code-Erzeugung ( $\rightarrow$  Befehlsfolge in Zielsprache)
- ▶ Zielmaschine (Ausführung)

# Übung: Beispiele für Übersetzer

## Java:

```
javac Foo.java # erzeugt Bytecode (Foo.class)
java Foo      # führt Bytecode aus (JVM)
```

## Einzelheiten der Übersetzung:

```
javap -c Foo # druckt Bytecode
```

## C:

```
gcc -c bar.c # erzeugt Objekt (Maschinen)code (bar.o)
gcc -o bar bar.o # linkt (lädt) Objektcode (Resultat)
./bar # führt gelinktes Programm aus
```

## Einzelheiten:

```
gcc -S bar.c # erzeugt Assemblercode (bar.s)
```

## Aufgaben:

- ▶ geschachtelte arithmetische Ausdrücke in Java und C: vergleiche Bytecode mit Assemblercode
- ▶ vergleiche Assemblercode für Intel und Sparc (einloggen auf goliath, dann gcc wie oben)

# Daten-Repräsentation im Compiler

- ▶ Jede Compiler-Phase arbeitet auf geeigneter Repräsentation ihre Eingabedaten.
- ▶ Die semantischen Operationen benötigen das Programm als Baum  
(das ist auch die Form, die der Programmierer im Kopf hat).
- ▶ In den Knoten des Baums stehen Token,
- ▶ jedes Token hat einen Typ und einen Inhalt (eine Zeichenkette).

# Token-Typen

Token-Typen sind üblicherweise

- ▶ reservierte Wörter (if, while, class, ...)
- ▶ Bezeichner (foo, bar, ...)
- ▶ Literale für ganze Zahlen, Gleitkommazahlen, Strings, Zeichen
- ▶ Trennzeichen (Komma, Semikolon)
- ▶ Klammern (runde: paren(these)s, eckige: brackets, geschweifte: braces) (jeweils auf und zu)
- ▶ Operatoren (=, +, &&, ...)

# Formale Sprachen

- ▶ ein *Alphabet* ist eine Menge von Zeichen,
- ▶ ein *Wort* ist eine Folge von Zeichen,
- ▶ eine *formale Sprache* ist eine Menge von Wörtern.

## Beispiele

- ▶ Alphabet  $\Sigma = \{a, b\}$ ,
- ▶ Wort  $w = ababaaab$ ,
- ▶ Sprache  $L =$  die Menge aller Wörter über  $\Sigma$  gerader Länge.

# Formale Sprachen: Chomsky-Hierarchie

- ▶ (Typ 0) aufzählbare Sprachen (beliebige Grammatiken, Turingmaschinen)
- ▶ (Typ 1) kontextsensitive Sprachen (monotone Grammatiken, linear beschränkte Automaten)
- ▶ (Typ 2) kontextfreie Sprachen (kontextfreie Grammatiken, Kellerautomaten)
- ▶ (Typ 3) reguläre Sprachen (rechtslineare Grammatiken, reguläre Ausdrücke, endliche Automaten)

Tokenklassen sind meist reguläre Sprachen.

Programmiersprachen werden kontextfrei beschrieben (mit Zusatzbedingungen).

# Grammatiken

*Grammatik G* besteht aus:

- ▶ Terminal-Alphabet  $\Sigma$   
(üblich: Kleibuchst., Ziffern)
- ▶ Variablen-Alphabet  $V$   
(üblich: Großbuchstaben)
- ▶ Startsymbol  $S \in V$
- ▶ Regelmenge  
(Wort-Ersetzungs-System)  
 $R \subseteq (\Sigma \cup V)^* \times (\Sigma \cup V)^*$

Grammatik

```
{ terminale
  = mkSet "abc"
, variablen
  = mkSet "SA"
, start = 'S'
, regeln = mkSet
  [ ("S", "abc")
  , ("ab", "aabbA")
  , ("Ab", "bA")
  , ("Ac", "cc")
  ]
```

von  $G$  erzeugte Sprache:  $L(G) = \{w \mid S \rightarrow^* w \wedge w \in \Sigma^*\}$ .



# Wort-Ersetzungs-Systeme

Berechnungs-Modell (Markov-Algorithmen)

- ▶ Zustand (Speicherinhalt): Zeichenfolge (Wort)
- ▶ Schritt: Ersetzung eines Teilwortes

Regelmenge  $R \subseteq \Sigma^* \times \Sigma^*$

Regel-Anwendung:

$u \rightarrow_R v \iff \exists x, z \in \Sigma^*, (l, r) \in R : u = x \cdot l \cdot z \wedge x \cdot r \cdot z = v.$

Beispiel: Bubble-Sort:  $\{ba \rightarrow ab, ca \rightarrow ac, cb \rightarrow bc\}$

Beispiel: Potenzieren:  $ab \rightarrow bba$

Aufgaben: gibt es unendlich lange Rechnungen für:

$R_1 = \{1000 \rightarrow 0001110\}, R_2 = \{aabb \rightarrow bbaaaa\}?$

# Typ-3-Grammatiken

(= rechtslineare Grammatiken)

jede Regel hat die Form

- ▶ Variable  $\rightarrow$  Terminal Variable
- ▶ Variable  $\rightarrow$  Terminal
- ▶ Variable  $\rightarrow \epsilon$

(vgl. lineares Gleichungssystem)

Beispiele

- ▶  $G_1 = (\{a, b\}, \{S, T\}, S, \{S \rightarrow \epsilon, S \rightarrow aT, T \rightarrow bS\})$
- ▶  $G_2 = (\{a, b\}, \{S, T\}, S, \{S \rightarrow \epsilon, S \rightarrow aS, S \rightarrow bT, T \rightarrow aT, T \rightarrow bS\})$

# Sätze über Reguläre Sprachen

Für jede Sprache  $L$  sind die folgenden Aussagen äquivalent:

- ▶ es gibt einen regulären Ausdruck  $X$  mit  $L = L(X)$ ,
- ▶ es gibt eine Typ-3-Grammatik  $G$  mit  $L = L(G)$ ,
- ▶ es gibt einen endlichen Automaten  $A$  mit  $L = L(A)$ .

Wenn  $L_1, L_2$  reguläre Sprachen sind, dann sind die folgenden Sprachen auch regulär:

- ▶ Mengenoperationen  $L_1 \cup L_2, L_1 \cap L_2, L_1 \setminus L_2$
- ▶ Verkettung  $L_1 \cdot L_2 = \{w_1 \cdot w_2 \mid w_1 \in L_1, w_2 \in L_2\}$
- ▶ Stern  $L_1^* = \bigcup_{k \geq 0} L_1^k$

# Reguläre Sprachen/Ausdrücke

Die Menge  $E(\Sigma)$  der *regulären Ausdrücke* über einem Alphabet (Buchstabenmenge)  $\Sigma$  ist die kleinste Menge  $E$ , für die gilt:

- ▶ für jeden Buchstaben  $x \in \Sigma : x \in E$   
(autotool: Ziffern oder Kleinbuchstaben)
- ▶ das leere Wort  $\epsilon \in E$  (autotool: Eps)
- ▶ die leere Menge  $\emptyset \in E$  (autotool: Empty)
- ▶ wenn  $A, B \in E$ , dann
  - ▶ (Verkettung)  $A \cdot B \in E$  (autotool: \* oder weglassen)
  - ▶ (Vereinigung)  $A + B \in E$  (autotool: +)
  - ▶ (Stern, Hülle)  $A^* \in E$  (autotool: ^\*)

Jeder solche Ausdruck beschreibt eine *reguläre Sprache*.

# Beispiele/Aufgaben zu regulären Ausdrücken

Wir fixieren das Alphabet  $\Sigma = \{a, b\}$ .

- ▶ alle Wörter, die mit  $a$  beginnen und mit  $b$  enden:  $a\Sigma^*b$ .
- ▶ alle Wörter, die wenigstens drei  $a$  enthalten  $\Sigma^*a\Sigma^*a\Sigma^*a\Sigma^*$
- ▶ alle Wörter mit gerade vielen  $a$  und beliebig vielen  $b$ ?
- ▶ Alle Wörter, die ein  $aa$  oder ein  $bb$  enthalten:  
 $\Sigma^*(aa \cup bb)\Sigma^*$
- ▶ (Wie lautet das Komplement dieser Sprache?)

# Übungen Reg. Ausdr.

- ▶ Tokenklassendef. für einige Programmiersprachen (welche Tokenklassen? welche reg. Ausdrücke? wie im Sprachstandard ausgedrückt?)
- ▶ String-Konstanten (Umgehen von Sonderzeichen)
- ▶ Kommentare
- ▶ Notation für reg. Ausdr. in gängigen Werkzeugen

# Kontextfreie Sprachen

Def (Wdhlg):  $G$  ist kontextfrei (Typ-2), falls

$$\forall(l, r) \in R(G) : l \in V.$$

geeignet zur Beschreibung von Sprachen mit hierarchischer Struktur.

Anweisung  $\rightarrow$  Bezeichner = Ausdruck

| if Ausdruck then Anweisung else Anweisung

Ausdruck  $\rightarrow$  Bezeichner | Literal

| Ausdruck Operator Ausdruck

Bsp: korrekt geklammerte Ausdrücke:

$$G = (\{a, b\}, \{S\}, S, \{S \rightarrow aSbS, S \rightarrow \epsilon\}).$$

Bsp: Palindrome:

$$G = (\{a, b\}, \{S\}, S, \{S \rightarrow aSa, S \rightarrow bSb, S \rightarrow \epsilon\}).$$

Bsp: alle Wörter  $w$  über  $\Sigma = \{a, b\}$  mit  $|w|_a = |w|_b$

# (erweiterte) Backus-Naur-Form

- ▶ Noam Chomsky: Struktur natürlicher Sprachen (1956)
- ▶ John Backus, Peter Naur: Definition der Syntax von Algol (1958)

Backus-Naur-Form (BNF)  $\approx$  kontextfreie Grammatik

```
<assignment> -> <variable> = <expression>  
<number> -> <digit> <number> | <digit>
```

## Erweiterte BNF

- ▶ Wiederholungen (Stern, Plus)  $\langle \text{digit} \rangle^+$
- ▶ Auslassungen

```
if <expr> then <stmt> [ else <stmt> ]
```

kann in BNF übersetzt werden

# Ableitungsbäume für CF-Sprachen

Def: ein geordneter Baum  $T$  mit Markierung  $m : T \rightarrow \Sigma \cup \{\epsilon\} \cup V$  ist Ableitungsbaum für eine CF-Grammatik  $G$ , wenn:

- ▶ für jeden inneren Knoten  $k$  von  $T$  gilt  $m(k) \in V$
- ▶ für jedes Blatt  $b$  von  $T$  gilt  $m(b) \in \Sigma \cup \{\epsilon\}$
- ▶ für die Wurzel  $w$  von  $T$  gilt  $m(w) = S(G)$  (Startsymbol)
- ▶ für jeden inneren Knoten  $k$  von  $T$  mit Kindern  $k_1, k_2, \dots, k_n$  gilt  $(m(k), m(k_1)m(k_2) \dots m(k_n)) \in R(G)$  (d. h. jedes  $m(k_i) \in V \cup \Sigma$ )
- ▶ für jeden inneren Knoten  $k$  von  $T$  mit einzigem Kind  $k_1 = \epsilon$  gilt  $(m(k), \epsilon) \in R(G)$ .

## Ableitungsbäume (II)

Def: der *Rand* eines geordneten, markierten Baumes  $(T, m)$  ist die Folge aller Blatt-Markierungen (von links nach rechts).

Beachte: die Blatt-Markierungen sind  $\in \{\epsilon\} \cup \Sigma$ , d. h.

Terminalwörter der Länge 0 oder 1.

Für Blätter:  $\text{rand}(b) = m(b)$ , für innere Knoten:

$\text{rand}(k) = \text{rand}(k_1) \text{rand}(k_2) \dots \text{rand}(k_n)$

Satz:  $w \in L(G) \iff$  existiert Ableitungsbaum  $(T, m)$  für  $G$  mit  $\text{rand}(T, m) = w$ .

# Eindeutigkeit

Def:  $G$  heißt *eindeutig*, falls  $\forall w \in L(G)$  *genau ein* Ableitungsbaum  $(T, m)$  existiert.

Bsp: ist  $\{S \rightarrow aSb \mid SS \mid \epsilon\}$  eindeutig?

(beachte: mehrere Ableitungen  $S \xrightarrow{*}_R w$  sind erlaubt, und wg. Kontextfreiheit auch gar nicht zu vermeiden.)

Die naheliegende Grammatik für arith. Ausdr.

$\text{expr} \rightarrow \text{number} \mid \text{expr} + \text{expr} \mid \text{expr} * \text{expr}$

ist mehrdeutig (aus *zwei* Gründen!)

Auswege:

- ▶ Transformation zu eindeutiger Grammatik (benutzt zusätzliche Variablen)
- ▶ Operator-Assoziativitäten und -Präzedenzen

# Assoziativität

$$(3 + 2) + 4 \stackrel{?}{=} 3 + 2 + 4 \stackrel{?}{=} 3 + (2 + 4)$$

$$(3 - 2) - 4 \stackrel{?}{=} 3 - 2 - 4 \stackrel{?}{=} 3 - (2 - 4)$$

$$(3 * 2) * 4 \stackrel{?}{=} 3 * 2 * 4 \stackrel{?}{=} 3 * (2 * 4)$$

- ▶ Grammatik-Regeln
- ▶ Plus ist nicht assoziativ (für Gleitkommazahlen)
- ▶ links oder rechts?

# Präzedenzen

$$(3 + 2) * 4 \stackrel{?}{=} 3 + 2 * 4 \stackrel{?}{=} 3 + (2 * 4)$$

- ▶ Grammatik-Regeln
- ▶ Verhältnis von plus zu minus, mal zu durch?

# Übungen

- ▶ Lexik und Syntax von Java:

[http://java.sun.com/docs/books/jls/third\\_edition/html/j3TOC.html](http://java.sun.com/docs/books/jls/third_edition/html/j3TOC.html)

- ▶ richtig oder falsch?

```
int x = /* foo /* // /** bar */ 42;
```

- ▶ “Wadler’s law of language design”
- ▶ reguläre Ausdrücke
- ▶ kontextfreie Grammatiken
- ▶ eindeutige Grammatiken für arithmetische Ausdrücke

# Statische und dynamische Semantik

Semantik = Bedeutung

- ▶ statisch (kann zur Übersetzungszeit geprüft werden)

Beispiele:

- ▶ Typ-Korrektheit von Ausdrücken,
- ▶ Bedeutung (Bindung) von Bezeichnern

Hilfsmittel: Attributgrammatiken

- ▶ dynamisch (beschreibt Ausführung des Programms)  
operational, axiomatisch, denotational

# Attributgrammatiken (I)

- ▶ Attribut: Annotation an Knoten des Syntaxbaums.  
 $A : \text{Knotenmenge} \rightarrow \text{Attributwerte}$  (Bsp:  $\mathbb{N}$ )
- ▶ Attributgrammatik besteht aus:
  - ▶ kontextfreier Grammatik  $G$  (Bsp:  $\{S \rightarrow e \mid mSS\}$ )
  - ▶ für jeden Knotentyp (Terminal + Regel)  
eine Menge von erlaubten Attribut-Tupeln  
 $(A(X_0), A(X_1), \dots, A(X_n))$   
für Knoten  $X_0$  mit Kindern  $[X_1, \dots, X_n]$

$$S \rightarrow mSS, A(X_0) + A(X_3) = A(X_2);$$

$$S \rightarrow e, A(X_0) = A(X_1);$$

$$\text{Terminale: } A(e) = 1, A(m) = 0$$

## Attributgrammatiken (II)

ein Ableitungsbaum mit Annotationen ist  
*korrekt bezüglich einer Attributgrammatik*, wenn

- ▶ zur zugrundeliegenden CF-Grammatik paßt
- ▶ in jedem Knoten das Attribut-Tupel (von Knoten und Kindern) zur erlaubten Tupelmengemenge gehört

Plan:

- ▶ Baum beschreibt Syntax, Attribute beschreiben Semantik

Ursprung: Donald Knuth: Semantics of Context-Free Languages, (Math. Systems Theory 2, 1968)

technische Schwierigkeit: Attributwerte effizient bestimmen.  
(beachte: (zirkuläre) Abhängigkeiten)

# Arten von Attributen

- ▶ synthetisiert:  
hängt nur von Attributwerten in Kindknoten ab
- ▶ ererbt (inherited)  
hängt nur von Attributwerten in Elternknoten und (linken)  
Geschwisterknoten ab

Wenn Abhängigkeiten bekannt sind, kann man Attributwerte durch Werkzeuge bestimmen lassen.

# Attributgrammatiken–Beispiele

- ▶ Auswertung arithmetischer Ausdrücke (dynamisch)
- ▶ Bestimmung des abstrakten Syntaxbaumes
- ▶ Typprüfung (statisch)
- ▶ Kompilation (für Kellermaschine) (statisch)

# Konkrete und abstrakte Syntax

- ▶ konkreter Syntaxbaum = der Ableitungsbaum
- ▶ abstrakter Syntaxbaum = wesentliche Teile des konkreten Baumes

unwesentlich sind z. B. die Knoten, die zu Hilfsvariablen der Grammatik gehören.

abstrakter Syntaxbaum kann als synthetisiertes Attribut konstruiert werden.

```
E -> E + P ; E.abs = new Plus(E.abs, P.abs)
E -> P ; E.abs = P.abs
```

# Regeln zur Typprüfung

... bei geschachtelten Funktionsaufrufen

- ▶ Funktion  $f$  hat Typ  $A \rightarrow B$
- ▶ Ausdruck  $X$  hat Typ  $A$
- ▶ dann hat Ausdruck  $f(X)$  den Typ  $B$

Beispiel

```
String x = "foo"; String y = "bar";
```

```
Boolean.toString (x.length() < y.length()));
```

(Curry-Howard-Isomorphie)

# Ausdrücke → Kellermaschine

Beispiel:

$3 * x + 1 \Rightarrow$  push 3, push x, mal, push 1, plus

- ▶ Code für Konstante/Variable  $c$ : `push c;`
- ▶ Code für Ausdruck  $x \text{ op } y$ : `code(x); code(y); op;`
- ▶ Ausführung eines Operators:  
holt beide Argumente vom Stack, schiebt Resultat auf Stack

Der erzeugte Code ist synthetisiertes Attribut!

Beispiele: Java-Bytecode (javac, javap),  
CIL (gmcs, monodis)

## Übungen (Stackmaschine)

Schreiben Sie eine Java-Methode, deren Kompilation genau diesen Bytecode erzeugt: a)

```
public static int h(int, int);
```

Code:

```
0: iconst_3
```

```
1: iload_0
```

```
2: iadd
```

```
3: iload_1
```

```
4: iconst_4
```

```
5: isub
```

```
6: imul
```

```
7: ireturn
```

b)

```
public static int g(int, int);
```

Code:

```
0: iload_0
```

```
1: istore_2
```

```
2: iload_1
```

# Dynamische Semantik

- ▶ operational:  
beschreibt Wirkung von Anweisungen durch Änderung des Programmzustandes
- ▶ denotational:  
ordnet jedem (Teil-)Programm einen Wert zu, Bsp: eine Funktion (höherer Ordnung).  
Beweis von Programmeigenschaften durch Term-Umformungen
- ▶ axiomatisch (Bsp: wp-Kalkül):  
enthält Schlußregeln, um Aussagen über Programme zu beweisen

# Bsp: Operationale Semantik

## Schleife

```
while (B) A
```

wird übersetzt in Sprungbefehle

```
if (B) ...
```

(vervollständige!)

Aufgabe: übersetze `for (A; B; C) D` in `while!`

# Denotationale Semantik

## Beispiele

- ▶ jedes (nebenwirkungsfreie) *Unterprogramm* ist eine Funktion von Argument nach Resultat
- ▶ jede *Anweisung* ist eine Funktion von Speicherzustand nach Speicherzustand

## Vorteile denotationaler Semantik:

- ▶ Bedeutung eines Programmes = mathematisches Objekt
- ▶ durch Term beschreiben, durch äquivalente Umformungen verarbeiten (equational reasoning)

## Vorteil deklarativer Programmierung:

Programmiersprache *ist* Beschreibungssprache

## Beispiel Denotationale Semantik

```
monoton :: Ord a => [a] -> Bool
monoton (x:y:zs) = x <= y && monoton (y:zs)
monoton _ = True
```

```
merge :: Ord a => [a] -> [a] -> [a]
merge [] ys = ys ; merge xs [] = xs
merge (x:xs) (y:ys) =
    if x <= y then x : merge ... ...
    else ...
```

Behauptung:  $\text{monoton } xs \ \&\& \ \text{monoton } ys$   
 $\Rightarrow \text{monoton } (\text{merge } xs \ ys)$

Beweis d. Induktion  
über  $\text{length } xs + \text{length } ys$

# Beispiel: Semantik von Unterprogr.

Welche Funktion ist das:

```
f (x) = if x > 52
        then x - 11
        else f (f (x + 12))
```

# Axiomatische Semantik

Notation für Aussagen über Programmezustände:

$$\{ V \} A \{ N \}$$

- ▶ für jeden Zustand  $s$ , in dem Vorbedingung  $V$  gilt:
- ▶ wenn  $A$  (Anweisung) ausgeführt wird,
- ▶ gilt im erreichten Zustand  $t$  die Nachbedingung  $N$

Beispiel:

$$\{ x \geq 5 \} y := x + 3 \{ y \geq 7 \}$$

Gültigkeit solcher Aussagen kann man

- ▶ beweisen (mit Hoare-Kalkül)
- ▶ prüfen (testen)

# Eiffel

**Bertrand Meyer**, <http://www.eiffel.com/>

```
class Stack [G]      feature
  count : INTEGER
  item  : G is require not empty do ... end
  empty : BOOLEAN is do .. end
  full  : BOOLEAN is do .. end
  put (x: G) is
    require not full do ...
    ensure not empty
      item = x
      count = old count + 1
```

Beispiel sinngemäß aus: B. Meyer: Object Oriented Software Construction, Prentice Hall 1997

# Hoare-Kalkül

Kalkül: für jede Anweisung ein Axiom, das die schwächste Vorbedingung (weakest precondition) beschreibt.

Beispiele

- ▶  $\{ N[x/E] \} x := E \{ N \}$
- ▶  $\{ V \text{ und } B \} C \{ N \}$   
und  $\{ V \text{ und not } B \} D \{ N \}$   
 $\Rightarrow \{ V \} \text{if } (B) \text{ then } C \text{ else } D \{ N \}$
- ▶ **Schleife ... benötigt Invariante**

# Axiom für Schleifen

wenn  $\{ I \text{ and } B \} A \{ I \}$ ,  
dann  $\{ I \} \text{ while } (B) \text{ do } A \{ I \text{ and not } B \}$

## Beispiel:

```
Eingabe int p, q;  
// p = P und q = Q  
int c = 0;  
// inv: p * q + c = P * Q  
while (q > 0) {  
    ???  
}  
// c = P * Q
```

**Moral:** erst Schleifeninvariante (Spezifikation), dann Implementierung.

# Übungen (Invarianten)

Ergänze das Programm:

```
Eingabe: natürliche Zahlen a, b;  
// a = A und b = B  
int p = 1; int c = ???;  
// Invariante:  $c^b * p = A^B$   
while (b > 0) {  
    ???  
    b = abrunden (b/2);  
}  
Ausgabe: p; //  $p = A^B$ 
```

# Warum Typen?

- ▶ Typ ist Menge von Werten mit Operationen
- ▶ für jede eigene Menge von Werten (Variablen) aus dem *Anwendungsbereich* benutze eine eigenen Typ
- ▶ halte verschiedene Typen sauber getrennt, mit Hilfe der Programmiersprache
- ▶ der Typ einer Variablen/Funktion ist ihre beste Dokumentation

# Historische Entwicklung

- ▶ keine Typen (alles ist int)
- ▶ vorgegebene Typen (Fortran: Integer, Real, Arrays)
- ▶ nutzerdefinierte Typen
- ▶ abstrakte Datentypen

# Überblick

- ▶ einfache (primitive) Typen
  - ▶ Zahlen, Wahrheitswerte, Zeichen
  - ▶ nutzerdefinierte Aufzählungstypen
  - ▶ Teilbereiche
- ▶ zusammengesetzte (strukturierte) Typen
  - ▶ Produkt (records)
  - ▶ Summe (unions)
  - ▶ Potenz (Funktionen: Arrays, (Hash-)Maps, Unterprogramme)
  - ▶ Verweistypen (Zeiger)

# Aufzählungstypen

können einer Teilmenge ganzer Zahlen zugeordnet werden

- ▶ vorgegeben: `int`, `char`, `boolean`
- ▶ nutzerdefiniert (`enum`)

```
typedef enum {  
    Mon, Tue, Wed, Thu, Fri, Sat, Sun  
} day;
```

Designfragen:

- ▶ automatisch nach `int` umgewandelt?
- ▶ automatisch von `int` umgewandelt?
- ▶ eine Konstante in mehreren Aufzählungen möglich?

# Keine Aufzählungstypen

das ist nett gemeint, aber vergeblich:

```
#define Mon 0
#define Tue 1
...
#define Sun 6
```

```
typedef int day;
```

```
int main () {
    day x = Sat;
    day y = x * x;
}
```

# Aufzählungstypen in C

im wesentlichen genauso nutzlos:

```
typedef enum {  
    Mon, Tue, Wed, Thu, Fri, Sat, Sun  
} day;
```

```
int main () {  
    day x = Sat;  
    day y = x * x;  
}
```

Übung: was ist in C++ besser?

# Aufzählungstypen in Java

```
enum Day {  
    Mon, Tue, Wed, Thu, Fri, Sat, Sun;  
  
    public static void main (String [] argv) {  
        for (Day d : Day.values ()) {  
            System.out.println (d);  
        }  
    }  
}
```

verhält sich wie Klasse

(genauer: Schnittstelle mit 7 Implementierungen)

siehe Übung (jetzt oder bei Objekten)

## Teilbereichstypen in Ada

```
with Ada.Text_IO;
procedure Day is
  type Day is ( Mon, Tue, Thu, Fri, Sat, Sun );
  subtype Weekday is Day range Mon .. Fri;
  X, Y : Day;
begin
  X := Fri;      Ada.Text_IO.Put (Day' Image(X) );
  Y := Day' Succ(X); Ada.Text_IO.Put (Day' Image(Y) );
end Day;
```

mit Bereichsprüfung bei jeder Zuweisung.  
einige Tests können aber vom Compiler statisch ausgeführt werden!

# Abgeleitete Typen in Ada

```
procedure Fruit is
  subtype Natural is
    Integer range 0 .. Integer'Last;
  type Apples is new Natural;
  type Oranges is new Natural;
  A : Apples; O : Oranges; I : Integer;
begin -- nicht alles korrekt:
  A := 4; O := A + 1; I := A * A;
end Fruit;
```

Natural, Äpfel und Orangen sind isomorph, aber nicht zuweisungskompatibel.

Sonderfall: Zahlenkonstanten gehören zu jedem abgeleiteten Typ.

# Zusammengesetzte Typen

Typ = Menge, Zusammensetzung = Mengenoperation:

- ▶ Produkt (record, struct)
- ▶ Summe (union)
- ▶ Potenz (Funktion)

# Produkttypen (Records)

$$R = A \times B \times C$$

Kreuzprodukt mit benannten Komponenten:

```
typedef struct {  
    A foo;  
    B bar;  
    C baz;  
} R;
```

```
R x; ... B x.bar; ...
```

erstmalig in COBOL ( $\leq 1960$ )

Übung: Record-Konstruktion (in C, C++)?

# Summen-Typen

$$R = A \cup B \cup C$$

disjunkte (diskriminierte) Vereinigung (Pascal)

```
type tag = ( eins, zwei, drei );
type R = record case t : tag of
    eins : ( a_value : A );
    zwei : ( b_value : B );
    drei : ( c_value : C );
end record;
```

nicht diskriminiert (C):

```
typedef union {
    A a_value; B b_value; C c_value;
}
```

# Potenz-Typen

$B^A := \{f : A \rightarrow B\}$  (Menge aller Funktionen von  $A$  nach  $B$ )  
ist sinnvolle Notation, denn  $|B|^{|A|} = |B^A|$   
spezielle Realisierungen:

- ▶ Funktionen (Unterprogramme)
- ▶ Wertetabellen (Funktion mit endlichem Definitionsbereich)  
(Assoziative Felder, Hashmaps)
- ▶ Felder (Definitionsbereich ist Aufzählungstyp) (Arrays)
- ▶ Zeichenketten (Strings)

die unterschiedliche Notation dafür (Beispiele?) ist bedauerlich.

# Felder (Arrays)

## Design-Entscheidungen:

- ▶ welche Index-Typen erlaubt? (Zahlen? Aufzählungen?)
- ▶ Bereichsprüfungen bei Indizierungen?
- ▶ Index-Bereiche statisch oder dynamisch?
- ▶ Allokation statisch oder dynamisch?
- ▶ Initialisierung?
- ▶ mehrdimensionale Felder gemischt oder rechteckig?

# Felder in C

```
int main () {  
    int a [10][10];  
    a[3][2] = 8;  
    printf ("%d\n", a[2][12]);  
}
```

statische Dimensionierung, dynamische Allokation, keine Bereichsprüfungen.

Form: rechteckig, Adress-Rechnung:

```
int [M][N];  
a[x][y] ==> *(&a + (N*x + y))
```

# Felder in Java

```
int [][] feld =
    { {1,2,3}, {3,4}, {5}, {} };
for (int [] line : feld) {
    for (int item : line) {
        System.out.print (item + " ");
    }
    System.out.println ();
}
```

dynamische Dimensionierung und Allokation,  
Bereichsprüfungen. Nicht notwendig rechteckig.

# Nicht rechteckige Felder in C?

Das geht:

```
int a [] = {1,2,3};  
int b [] = {4,5};  
int c [] = {6};  
    e     = {a,b,c};  
printf ("%d\n", e[1][1]);
```

aber welches ist dann der Typ von e?  
(es ist nicht `int e [][]`.)

# Dynamische Feldgrößen

Designfrage: kann ein Feld (auch: String) seine Größe ändern?

(C: wird sowieso nicht geprüft, Java: nein, Perl: ja)

in Java: wenn man das will, dann will man statt Array eine LinkedList, statt String einen StringBuffer.

wenn man mit Strings arbeitet, dann ist es meist ein Fehler:

benutze Strings *zwischen* Programmen,  
aber niemals *innerhalb* eines Programms.

ein einem Programm: benutze immer anwendungsspezifische Datentypen.

... deren externe Syntax spiel überhaupt keine Rolle

# Speicherplatz für Objekte

- ▶ statisch (vor Programmstart bestimmt)
- ▶ dynamisch (während Programmlauf bestimmt)
  - ▶ flüchtig (Lebensdauer: Block)  
Stack, Register
  - ▶ permanent (lebt länger als Block)  
Heap

Zur Benutzung permanenter dynamische Daten benötigt man *Zeiger*.

# Zeiger- und Verweistypen

Zeiger = *Adresse* eines Wertes. — Wofür?

- ▶ dynamische Speicherzellen (im Heap)
- ▶ verkettete Strukturen (Listen, Bäume)
- ▶ gemeinsame Teilstrukturen (sharing)
- ▶ Adresse ist einfacher zu transportieren als Wert

Designfragen:

- ▶ Typsicherheit
- ▶ Zeiger oder Verweis?
- ▶ Verfolgung (De-Referenzierung) implizit oder explizit?

# Zeiger- und Verweistypen

Zeiger = *Adresse* eines Wertes. — Wofür?

- ▶ dynamische Speicherzellen (im Heap)
- ▶ verkettete Strukturen (Listen, Bäume)
- ▶ gemeinsame Teilstrukturen (sharing)
- ▶ Adresse ist einfacher zu transportieren als Wert

Designfragen:

- ▶ Typsicherheit
- ▶ Zeiger oder Verweis?
- ▶ Verfolgung (De-Referenzierung) implizit oder explizit?

# Zeiger (pointer) in C

Typ  $T$ , Zeigertyp  $T * p$

Adresse feststellen:

```
T x;
```

```
T * p = &x;
```

Zeiger (einmal) verfolgen (de-referenzieren):

```
T y = *p;
```

Vorsicht: `int* a, b;`

Zeiger-Arithmetik:

```
char * c = malloc(sizeof(int) * 20);
```

```
*((int*)c + 4) = 42;
```

```
printf ("%d\n", *((int*)(c + 4)));
```

# Verweise (references) in C++

Typ T, Verweistyp T & p

Adresse feststellen (implizit):

```
int x = 9;  
int & p = x;
```

Verweis verfolgen (implizit):

```
int y = p;
```

*keine* Änderungen von Verweisen

```
int z = 10; p = z;    p = 8; cout << z;  
    vgl.    p = &z; *p = 8;
```

# Zeiger/Verweise in Java?

- ▶ scheint es nicht zu geben...
- ▶ doch: es gibt Wert-Typen (int, boolean, ...)  
und Verweis-Typen (alle Klassen)
- ▶ (aber keine Zeiger)

```
static class T { int foo = 8; }  
public static void main (String [] args) {  
    T x = new T ();  
    System.out.println ( x.foo );  
    T y = x;  
    y.foo = 9;  
    System.out.println ( x.foo );  
}
```

# Zeiger in C#

- ▶ class: **Zeigersemantik**
- ▶ struct: **Wert-Semantik**

```
class/struct Foo { public int bar; }

public static void Main () {
    Foo x = new Foo (); x.bar = 3;
    System.Console.WriteLine (x.bar);
    Foo y = x;           y.bar = 4;
    System.Console.WriteLine (x.bar);
}
```

# Probleme mit Zeigern

- ▶ tote Zeiger (wg. verfrühter Freigabe)

```
T *p = malloc (...); T *q = p; free (p); .. *q .
```

- ▶ Speichermüll (wg. vergessener Freigabe)

```
while (.. )  
    { T *p = malloc (...); .. *p .. }
```

- ▶ verschiedene Namen für gleiche Objekte (aliasing)  
erschweren Programmanalyse

# Aliasing

... erschwert Programmanalyse (für Menschen und optimierende Compiler)

f.c:

```
void f (int * p) {  
    *p = 9;  
}
```

g.c:

```
void f (int * p);  
void g () {  
    int x = 8;  
    int *p = &x;  
    f (p);  
    printf ("%d\n", x);  
}
```

Übung: was wird besser durch `const`? Wo?

# Automatische Freigabe

## Müllsammeln (garbage collection)

- ▶ Verweiszähler
- ▶ markierende Kollektoren
- ▶ kopierende (kompaktierende) Kollektoren
- ▶ Generationen
- ▶ Platzbedarf, Zeitgarantien
- ▶ nebenläufige Ausführung (zunehmend wichtig)

Erfunden von McCarthy für LISP ( $\leq 1960$ ), seitdem in jeder F(L)P-Sprache, seit Java, C# im imperativen Mainstream.

```
java -Xloggc:Foo.gc Foo
```

# Verweiszähler

- ▶ jedes Objekt  $x$  hat Zähler für Anzahl der Verweise auf  $x$ .
- ▶ bei Anlegen eines Verweise Zähler erhöhen
- ▶ bei Löschen verringern
- ▶ bei Zähler 0 freigeben und verwiesene Objekte löschen (→ Rekursion)
- ▶ Zähler brauchen Platz
- ▶ Löschen ist aufwendig
- ▶ funktioniert nicht bei Kreisverweisen

# Markierende Kollektoren

- ▶ bei Speicheranforderung, die nicht erfüllt werden kann:
- ▶ alle *lebenden*, d. h. von Wurzeln erreichbaren Objekte werden markiert (mark)
- ▶ restliche in Freispeicherliste (sweep)
- ▶ Markierung: 1 Bit; Zeit  $\sim$  Speicher, auch bei viel Müll.
- ▶ benötigt Stack (gerade dann, wenn wir keinen Platz mehr haben)? — Nein, *pointer reversal* möglich.
- ▶ ist *konservativ*: anwendbar auch bei Sprachen, bei denen Zeiger nicht sicher erkennbar sind
- ▶ `http://www.hpl.hp.com/personal/Hans_Boehm/gc/`  
(z. B. benutzt in GNU JVM, mono)

# Kopieren/Kompaktieren

- ▶ (wenigstens) zwei Speicherbereiche: Fromspace und Tospace
- ▶ lebende Zellen im Fromspace werden nach Tospace kopiert,
- ▶ dabei Verweise passend umgebogen
- ▶ was nicht kopiert wurde (Rest des Fromspaces), ist Müll
- ▶ Zeit  $\sim$  lebender Anteil des Speichers
- ▶ muß alles kopieren (auch Nicht-Zeiger).
- ▶ muß Zeiger sicher erkennen

# Generationen usw.

- ▶ Generation  $k$ : Speicherbereich für die Objekte, die schon  $k$  garbage collections überlebt haben
- ▶ je älter, desto weniger Elemente
- ▶ Kollektor arbeitet bevorzugt auf neuester Generation

## Realisierung in JVM:

- ▶ Eden
- ▶ Survivor spaces (1, 2)
- ▶ Tenured

# Literatur GC

**Hans Boehm: *Allocation and GC Myths*** [http://www.hpl.hp.com/personal/Hans\\_Boehm/gc/myths.ps](http://www.hpl.hp.com/personal/Hans_Boehm/gc/myths.ps)

**Brian Goetz: *Garbage collection in the HotSpot JVM***  
<http://www-128.ibm.com/developerworks/java/library/j-jtp11253/>

**Sun: *Tuning Garbage Collection with the 5.0 Java[tm] Virtual Machine*** [http://java.sun.com/docs/hotspot/gc5.0/gc\\_tuning\\_5.html](http://java.sun.com/docs/hotspot/gc5.0/gc_tuning_5.html)

**Sun: *Garbage Collection FAQ*** <http://java.sun.com/docs/hotspot/gc1.4.2/faq.html>

**Bertram Felgenhauer: *Parallele GC***  
<http://141.57.11.224/~bf3/gc/>

# Übung GC (I)

Finden Sie geeignete JVM-Konfigurationen für das Programm *KnockedForLoops*:

- ▶ <http://www.imn.htwk-leipzig.de/~waldmann/edu/ws09/pps/extra/kfl/KnockedForLoops.jar>
- ▶ **Archiv mit Eingabedateien (TPDB 5.0.2):** <http://dev.aspsimon.org/projects/termcomp/downloads/>

▶ **Testfall:**

```
java -jar KnockedForLoops.jar tpdb-5.0/SRS/Gebha
```

**sollte (sehr schnell) eine Ausgabe dieser Form liefern:**

```
Enumerating derivations modulo
```

```
( RULES 0 0 0 0 -> 0 1 0 1 , 1 0 0 1 -> 0 1 0 0
```

```
NO
```

```
Loop of length 27 starting with a string of length 27  
found after enumerating 14077 derivations:
```

```
.aaaa.ababbababaa
```

```
rule aaaa -> baba at position 0
```

## Übung GC (II)

Richtig viel Müll erzeugen:

```
import java.util.*;
class Garbage {
    static long sum (int top) {
        List<Integer> l = new LinkedList<Integer>()
        for (int x = 0; x < top; x++) {
            l.add (x);
        }
        long result = 0;
        for (int x : l) {
            result += x;
        }
        return result;
    }
    public static void main (String [] args) {
        for (int i = 0; i < 1000*1000; i+=10000) {
            System.out.println ( i + " : " + sum(i)
        }
    }
}
```

# Einleitung

## Semantik:

- ▶ Ausdruck hat *Wert* (Zahl, Objekt, ...)  
(Ausdruck wird *ausgewertet*)
- ▶ Anweisung hat *Wirkung* (Änderung des Programm/Welt-Zustandes)  
(Anweisung wird *ausgeführt*)

Vgl. Trennung (in Pascal, Ada)

- ▶ Funktion (Aufruf ist Ausdruck)
- ▶ Prozedur (Aufruf ist Anweisung)

in den allen imperativen Sprachen gibt es Ausdrücke mit Nebenwirkungen

# Designfragen für Ausdrücke

- ▶ Präzedenzen (Vorrang)
- ▶ Assoziativitäten (Gruppierung)
- ▶ Ausdrücke dürfen (Neben-)Wirkungen haben?
- ▶ in welcher Reihenfolge treten die auf?
- ▶ welche impliziten Typumwandlungen?
- ▶ explizite Typumwandlungen (cast)?
- ▶ kann Programmierer Operatoren definieren? überladen?

# Syntax von Ausdrücken

- ▶ einfache Ausdrücke : Konstante, Variable
- ▶ zusammengesetzte Ausdrücke:
  - ▶ Operator-Symbol zwischen Argumenten
  - ▶ Funktions-Symbol vor Argument-Tupel

wichtige Spezialfälle für Operatoren:

- ▶ arithmetische, relationale, boolesche

Wdhlg: Syntaxbaum, Präzedenz, Assoziativität.

# Syntax von Konstanten

Was druckt diese Anweisung?

```
System.out.println ( 12345 + 54321 );
```

dieses und einige der folgenden Beispiele aus: Joshua Bloch,  
Neil Gafter: *Java Puzzlers*, Addison-Wesley, 2005.

# Der Plus-Operator in Java

... addiert Zahlen und verkettet Strings.

```
System.out.println ("foo" + 3 + 4);  
System.out.println (3 + 4 + "bar");
```

# Überladene Operatornamen

aus praktischen Gründen sind arithmetische und relationale Operatornamen *überladen*

(d. h.: ein Name für mehrere Bedeutungen)

Überladung wird aufgelöst durch die Typen der Argumente.

```
int x = 3; int y = 4; ... x + y ...  
double a; double b; ... a + b ...  
String p; String q; ... p + q ...
```

# Automatische Typanpassungen

in vielen Sprachen postuliert man eine Hierarchie von Zahlbereichstypen:

`byte`  $\subseteq$  `int`  $\subseteq$  `float`  $\subseteq$  `double`

im allgemeinen ist das eine Halbordnung.

Operator mit Argumenten verschiedener Typen:

`(x :: int) + (y :: float)`

beide Argumente werden zu kleinstem gemeinsamen Obertyp promoviert, falls dieser eindeutig ist (sonst statischer Typfehler)  
(Halbordnung  $\rightarrow$  Halbverband)

# Implizite/Explizite Typumwandlungen

Was druckt dieses Programm?

```
long x = 1000 * 1000 * 1000 * 1000;  
long y = 1000 * 1000;  
System.out.println ( x / y );
```

Was druckt dieses Programm?

```
System.out.println ((int) (char) (byte) -1);
```

Moral: wenn man nicht auf den ersten Blick sieht, was ein Programm macht, dann macht es wahrscheinlich nicht das, was man will.

# Explizite Typumwandlungen

sieht gleich aus und heißt gleich (cast), hat aber verschiedene Bedeutungen:

- ▶ Datum soll in anderen Typ gewandelt werden, Repräsentation ändert sich:

```
double x = (double) 2 / (double) 3;
```

- ▶ Programmierer weiß es besser (als der Compiler), Repräsentation ändert sich nicht:

```
List books;  
Book b = (Book) books.get (7);
```

...kommt nur vor, wenn man die falsche Programmiersprache benutzt (nämlich Java vor 1.5)

# Der Verzweigungs-Operator

Absicht: statt

```
if ( 0 == x % 2 ) {  
    x = x / 2;  
} else {  
    x = 3 * x + 1;  
}
```

lieber

```
x = if ( 0 == x % 2 ) {  
    x / 2  
} else {  
    3 * x + 1  
} ;
```

historische Notation dafür

```
x = ( 0 == x % 2 ) ? x / 2 : 3 * x + 1;
```

?/: ist *ternärer* Operator

# Verzweigungs-Operator(II)

(... ? ... : ... ) in C, C++, Java

Anwendung im Ziel einer Zuweisung (C++):

```
int main () {  
    int a = 4; int b = 5; int c = 6;  
    ( c < 7 ? a : b ) = 8;  
}
```

# Relationale Operatoren

kleiner, größer, gleich,...

Was tut dieses Programm (C? Java?)

```
int a = -4; int b = -3; int c = -2;  
if (a < b < c) {  
    printf ("aufsteigend");  
}
```

# Logische (Boolesche) Ausdrücke

- ▶ und `&&`, `||` oder, nicht `!`, gleich, ungleich, kleiner, ...
- ▶ nicht verwechseln mit Bit-Operationen `&`, `|`  
(in C gefährlich, in Java ungefährlich—warum?)
- ▶ verkürzte Auswertung?

```
int [] a = ...; int k = ...;  
if ( k >= 0 && a[k] > 7 ) { ... }
```

(Ü: wie sieht das in Ada aus?)

## Noch mehr Quizfragen

- ▶ `System.out.println ("H" + "a");`  
`System.out.println ('H' + 'a');`
- ▶ `char x = 'X'; int i = 0;`  
`System.out.print (true ? x : 0);`  
`System.out.print (false ? i : x);`

Erklären durch Verweis auf Java Language Spec.

# Der Zuweisungs-Operator

Syntax:

- ▶ Algol, Pascal: Zuweisung  $:=$ , Vergleich  $=$
- ▶ Fortran, C, Java: Zuweisung  $=$ , Vergleich  $==$

Semantik der Zuweisung  $a = b$ :

Ausdrücke links und rechts werden verschieden behandelt:

- ▶ bestimme Adresse (lvalue)  $p$  von  $a$
- ▶ bestimme Wert (rvalue)  $v$  von  $b$
- ▶ schreibe  $v$  auf  $p$

# Weitere Formen der Zuweisung

(in C-ähnlichen Sprachen)

- ▶ verkürzte Zuweisung:  $a \ += \ b$   
entsprechend für andere binäre Operatoren
  - ▶ lvalue  $p$  von  $a$  wird bestimmt (nur einmal)
  - ▶ rvalue  $v$  von  $b$  wird bestimmt
  - ▶ Wert auf Adresse  $\&p$  wird um  $v$  erhöht
- ▶ Inkrement/Dekrement
  - ▶ Präfix-Version  $++i$ ,  $--j$ : Wert ist der geänderte
  - ▶ Suffix-Version  $i++$ ,  $j--$ : Wert ist der vorherige

# Ausdrücke mit Nebenwirkungen

(*side effect*; falsche Übersetzung: Seiteneffekt)  
in C-ähnlichen Sprachen: Zuweisungs-Operatoren bilden  
Ausdrücke, d. h. Zuweisungen sind Ausdrücke und können als  
Teile von Ausdrücken vorkommen.

Wert einer Zuweisung ist der zugewiesene Wert

```
int a; int b; a = b = 5; // wie geklammert?
```

Komma-Operator zur Verkettung von Ausdrücken (mit  
Nebenwirkungen)

```
for (... ; ... ; i++, j--) { ... }
```

# Auswertungsreihenfolgen

Kritisch: wenn Wert des Ausdrucks von Auswertungsreihenfolge abhängt:

```
int a; int b = (a = 5) + (a = 6);  
int d = 3; int e = (d++) - (++d);
```

- ▶ keine Nebenwirkungen: egal
- ▶ mit Nebenwirkungen:
  - ▶ C, C++: Reihenfolge nicht spezifiziert, wenn Wert davon abhängt, dann ist Verhalten *nicht definiert*
  - ▶ Java, C#: Reihenfolge genau spezifiziert (siehe JLS)

# Auswertungsreihenfolge in C

Sprachstandard (C99, C++) benutzt Begriff *sequence point* (Meilenstein):

bei Komma, Fragezeichen, && und ||

die Nebenwirkungen zwischen Meilensteinen müssen unabhängig sein (nicht die gleiche Speicherstelle betreffen), ansonsten ist das Verhalten undefiniert (d.h., der Compiler darf machen, was er will)

```
int x = 3; int y = ++x + ++x + ++x;
```

vgl. Aussagen zu sequence points in

<http://gcc.gnu.org/readings.html>

Gurevich, Huggins: *Semantics of C*, [http://citeseerx.](http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.29.6755)

[ist.psu.edu/viewdoc/summary?doi=10.1.1.29.6755](http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.29.6755)

# Definition

Semantik: Anweisung hat *Wirkung* (Zustandsänderung), die bei Ausführung eintritt.

abstrakte Syntax:

- ▶ einfache Anweisung:
  - ▶ Zuweisung
  - ▶ Unterprogramm-Aufruf
- ▶ zusammengesetzte Anweisung:
  - ▶ Nacheinanderausführung (Block)
  - ▶ Verzweigung (zweifach: if, mehrfach: switch)
  - ▶ Wiederholung (Sprung, Schleife)

# Programm-Ablauf-Steuerung

Ausführen eines Programms im von-Neumann-Modell:  
Was? (Operation) Womit? (Operanden) Wohin? (Resultat) Wie weiter? (nächste Anweisung)

strukturierte Programmierung:

- ▶ Nacheinander
- ▶ außer der Reihe (Sprung, Unterprogramm, Exception)
- ▶ Verzweigung
- ▶ Wiederholung

engl. *control flow*, falsche Übersetzung: Kontrollfluß;  
*to control* = steuern, *to check* = kontrollieren/prüfen

# Blöcke

Folge von (Deklarationen und) Anweisungen

Designfrage: Blöcke

- ▶ explizit (Klammern, begin/end)
- ▶ implizit (if ... then ... end if)

Designfrage: Deklarationen gestattet

- ▶ am Beginn des (Unter-)Programms (Pascal)
- ▶ am Beginn des Blocks (C)
- ▶ an jeder Stelle des Blocks (C++, Java)

# Verzweigungen (zweifach)

in den meisten Sprachen:

```
if Bedingung then Anweisung1 [ else Anweisung2 ]
```

Designfragen:

- ▶ was ist als Bedingung gestattet (gibt es einen Typ für Wahrheitswerte?)
- ▶ dangling else
  - ▶ gelöst durch Festlegung (else gehört zu letztem if)
  - ▶ vermieden durch Block-Bildung (Perl, Ada)
  - ▶ tritt nicht auf, weil man else nie weglassen darf (vgl. ?/:) (Haskell)

# Mehrfach-Verzweigung

```
switch (e) {  
    case c1 : s1 ;  
    case c2 : s2 ;  
    [ default : sn; ]  
}
```

## Designfragen:

- ▶ welche Typen für  $e$ ?
- ▶ welche Werte für  $c_i$ ?
- ▶ Wertebereiche?
- ▶ was passiert, wenn mehrere Fälle zutreffen?
- ▶ was passiert, wenn kein Fall zutrifft (default?)
- ▶ (effiziente Kompilation?)

# Switch/break

das macht eben in C, C++, Java nicht das, was man denkt:

```
switch (index) {  
    case 1 : odd ++;  
    case 2 : even ++;  
    default :  
        printf ("wrong index %d\n", index);  
}
```

C#: jeder Fall *muß* mit break (oder goto) enden.

# Kompilation

ein switch (mit vielen cases) wird übersetzt in:

- ▶ (naiv) eine lineare Folge von binären Verzweigungen (if, elsif)
- ▶ (semi-clever) einen balancierter Baum von binären Verzweigungen
- ▶ (clever) eine Sprungtabelle

Übung:

- ▶ einen langen Switch (1000 Fälle) erzeugen (durch ein Programm!)
- ▶ Assembler/Bytecode anschauen

# Wiederholungen

- ▶ Maschine, Assembler: (un-)bedingter Sprung
- ▶ strukturiert: Schleifen

Designfragen für Schleifen:

- ▶ wie wird Schleife gesteuert? (Bedingung, Zähler, Daten)
- ▶ an welcher Stelle in der Schleife findet Steuerung statt (Anfang, Ende, dazwischen, evtl. mehreres)

# Schleifen steuern durch...

## ▶ Zähler

```
for p in 1 .. 10 loop .. end loop;
```

## ▶ Daten

```
map (\x -> x*x) [1,2,3] ==> [1,4,9]
```

```
Collection<String> c  
    = new LinkedList<String> ();  
for (String s : c) { ... }
```

## ▶ Bedingung

```
while ( x > 0 ) { if ( ... ) { x = ... } ... }
```

# Zählschleifen

Idee: vor Beginn steht Anzahl der Durchläufe fest.

richtig realisiert ist das nur in Ada:

```
for p in 1 .. 10 loop ... end loop;
```

- ▶ Zähler  $p$  wird implizit deklariert
- ▶ Zähler ist im Schleifenkörper konstant

Vergleiche (beide Punkte) mit Java, C++, C

# Termination

Satz: Jedes Programm aus

- ▶ Zuweisungen
- ▶ Verzweigungen
- ▶ Zählschleifen

terminiert (hält) für jede Eingabe.

Äquivalenter Begriff (für Bäume anstatt Zahlen): strukturelle Induktion (fold, Visitor, primitive Rekursion)

Satz: es gibt berechenbare Funktionen, die nicht primitiv rekursiv sind.

Beispiel: Interpreter für primitiv rekursive Programme.

# Schleifen mit Bedingungen

das ist die allgemeinste Form, ergibt (partielle) rekursive Funktionen, die terminieren nicht notwendig für alle Argumente.

Steuerung

- ▶ am Anfang: `while (Bedingung) Anweisung`
- ▶ am Ende: `do Anweisung while (Bedingung)`

Weitere Änderung des Ablaufes:

- ▶ vorzeitiger Abbruch (`break`)
- ▶ vorzeitige Wiederholung (`continue`)
- ▶ beides auch nicht lokal

# Abarbeitung von Schleifen

operationale Semantik durch Sprünge:

```
while (B) A;  
==>  
start : if (!B) goto end;  
        A;  
        goto start;  
end   : skip;
```

(das ist auch die Notation der autotool-Aufgabe)

Ü: do A while (B);

# vorzeitiges Verlassen

## ▶ ...der Schleife

```
while ( B1 ) {  
    A1;  
    if ( B2 ) break;  
    A2;  
}
```

## ▶ ...des Schleifenkörpers

```
while ( B1 ) {  
    A1;  
    if ( B2 ) continue;  
    A2;  
}
```

# Geschachtelte Schleifen

manche Sprachen gestatten Markierungen (Labels) an Schleifen, auf die man sich in break beziehen kann:

```
foo : for (int i = ...) {  
    bar : for (int j = ...) {  
  
        if (...) break foo;  
  
    }  
}
```

Wie könnte man das simulieren?

# Sprünge

- ▶ bedingte, unbedingte (mit bekanntem Ziel)
  - ▶ Maschinensprachen, Assembler, Java-Bytecode
  - ▶ Fortran, Basic: if Bedingung then Zeilennummer
  - ▶ Fortran: dreifach-Verzweigung (arithmetic-if)
- ▶ “computed goto” (Zeilennr. des Sprungziels ausrechnen)

# Sprünge und Schleifen

- ▶ man kann jedes while-Programm in ein goto-Programm übersetzen
- ▶ und jedes goto-Programm in ein while-Programm ...
- ▶ ... das normalerweise besser zu verstehen ist.
- ▶ strukturierte Programmierung = jeder Programmbaustein hat genau einen Eingang und genau einen Ausgang
- ▶ aber: vorzeitiges Verlassen von Schleifen
- ▶ aber: Ausnahmen (Exceptions)

# Sprünge und Schleifen (Beweis)

Satz: zu jedem goto-Programm gibt es ein äquivalentes while-Programm.

Beweis-Idee: `1 : A1, 2 : A2; .. 5: goto 7; ..`  $\Rightarrow$

```
while (true) {
    switch (pc) {
        case 1 : A1 ; pc++ ; break; ...
        case 5 : pc = 7 ; break; ...
    }
}
```

Das nützt aber softwaretechnisch wenig, das übersetzte Programm ist genauso schwer zu warten wie das Original.

# Schleifen und Unterprogramme

zu jedem while-Programm kann man ein äquivalentes angeben, das nur Verzweigungen (if) und Unterprogramme benutzt.

**Beweis-Idee:** `while (B) A; ⇒`

```
void s () {  
    if (B) { A; s (); }  
}
```

**Anwendung:** C-Programme ohne Schlüsselwörter.

# Denotationale Semantik (I)

vereinfachtes Modell, damit Eigenschaften entscheidbar werden (sind die Programme  $P_1, P_2$  äquivalent?)

Syntax: Programme

- ▶ Aktionen,
- ▶ Zustandsprädikate (in Tests)
- ▶ Sequenz/Block, if, goto/while.

Beispiel:

```
while (B && !C) { P; if (C) Q; }
```

## Denotationale Semantik (II)

Semantik des Programms  $P$  ist Menge der Spuren von  $P$ .

- ▶ *Spur* = eine Folge von Paaren von Zustand und Aktion,
- ▶ ein *Zustand* ist eine Belegung der Prädikatsymbole,
- ▶ jede Aktion zerstört alle Zustandsinformation.

Satz: Diese Spursprachen (von goto- und while-Programmen) sind *regulär*.

Beweis: Konstruktion über endlichen Automaten.

- ▶ Zustandsmenge = Prädikatbelegungen  $\times$  Anweisungs-Nummer
- ▶ Transitionen? (Beispiele)

Damit ist Spur-Äquivalenz von Programmen entscheidbar.  
Beziehung zu tatsächlicher Äquivalenz?

# Grundsätzliches

Ein Unterprogramm ist ein benannter Block mit einer Schnittstelle. Diese beschreibt den Datentransport zwischen Aufrufer und Unterprogramm.

- ▶ Funktion
  - ▶ liefert Wert
  - ▶ Aufruf ist Ausdruck
- ▶ Prozedur
  - ▶ hat Wirkung, liefert keinen Wert (void)
  - ▶ Aufruf ist Anweisung

# Argumente/Parameter

- ▶ in der Deklaration benutzte Namen heißen (formale) *Parameter*,
- ▶ bei Aufruf benutzte Ausdrücke heißen *Argumente* (... nicht: aktuelle Parameter, denn engl. *actual* = dt. tatsächlich)

Designfragen bei Parameterzuordnung:

- ▶ über Position oder Namen? gemischt?
- ▶ defaults für fehlende Argumente?
- ▶ beliebig lange Argumentlisten?

# Positionelle/benannte Argumente

Üblich ist Zuordnung über Position

```
void p (int height, String name) { ... }  
p (8, "foo");
```

in Ada: Zuordnung über Namen möglich

```
procedure Paint (height : Float; width : Float);  
Paint (width => 30, height => 40);
```

nach erstem benanntem Argument keine positionellen mehr erlaubt

code smell: lange Parameterliste,  
refactoring: Parameterobjekt einführen  
allerdings fehlt (in Java) benannte Notation für  
Record-Konstanten.

# Default-Werte

C++:

```
void p (int x, int y, int z = 8);  
p (3, 4, 5); p (3, 4);
```

Default-Parameter müssen in Deklaration am Ende der Liste stehen

Ada:

```
procedure P  
  (X : Integer; Y : Integer := 8; Z : Integer);  
P (4, Z => 7);
```

Beim Aufruf nach weggelassenem Argument nur noch benannte Notation

# Variable Argumentanzahl (C)

wieso geht das eigentlich:

```
#include <stdio.h>
char * fmt = really_complicated();
printf (fmt, x, y, z);
```

Anzahl und Typ der weiteren Argumente werden überhaupt nicht geprüft:

```
extern int printf
    (__const char *__restrict __format, ...);
```

## Variable Argumentanzahl (Java)

```
static void check (String x, int ... ys) {  
    for (int y : ys) { System.out.println (y); }  
}
```

```
check ("foo", 1, 2); check ("bar", 1, 2, 3, 4);
```

letzter formaler Parameter kann für beliebig viele des gleichen Typs stehen.

tatsächlich gilt `int [] ys`,  
das ergibt leider Probleme bei generischen Typen

# Parameter-Übergabe (Semantik)

Datenaustausch zw. Aufrufer (caller) und Aufgerufenem (callee): über globalen Speicher

```
#include <errno.h>  
extern int errno;
```

oder über Parameter.

Datentransport (entspr. Schlüsselwörtern in Ada)

- ▶ in: (Argumente) vom Aufrufer zum Aufgerufenen
- ▶ out: (Resultate) vom Aufgerufenen zum Aufrufer
- ▶ in out: in beide Richtungen

# Parameter-Übergabe (Implementierungen)

- ▶ pass-by-value (Wert)
- ▶ copy in/copy out (Wert)
- ▶ pass-by-reference (Verweis)
- ▶ pass-by-name (textuelle Substitution)  
selten ... Algol68, CPP-Macros ... Vorsicht!

# Parameterübergabe

häufig benutzte Implementierungen:

- ▶ Pascal: by-value (default) oder by-reference (VAR)
- ▶ C: by-value (Verweise ggf. selbst herstellen)
- ▶ C++ unterscheidet zwischen Zeigern (\*, wie in C) und Referenzen (&, verweisen immer auf die gleiche Stelle, werden automatisch dereferenziert)
- ▶ Java: primitive Typen *und* Referenz-Typen (= Verweise auf Objekte) by-value

# Call-by-name

äquivalent zu textueller Ersetzung  
Algol(68): Jensen's device

```
sum (int i, int n; int f) {  
    int s = 0;  
    for (i=0; i<n; i++) { s += f; }  
    return s;  
}  
int [10] a; int i; sum (i, 10, a[i]);
```

# Call-by-name (Macros)

```
#define thrice(x) 3*x // gefährlich  
thrice (4+y) ==> 3*4+y
```

“the need for a preprocessor shows omissions in the language”

- ▶ fehlendes Modulsystem (Header-Includes)
- ▶ fehlende generische Polymorphie  
( $\Rightarrow$  Templates in C+)

weitere Argumente:

- ▶ mangelndes Vertrauen in optimierende Compiler (inlining)
- ▶ bedingte Übersetzung

## Aufgaben zu Parameter-Modi (I)

Erklären Sie den Unterschied zwischen (Ada)

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Check is
    procedure Sub (X: in out Integer;
                  Y: in out Integer;
                  Z: in out Integer) is
    begin
        Y := 8; Z := X;
    end;
    Foo: Integer := 9;    Bar: Integer := 7;
begin
    Sub (Foo, Foo, Bar);
    Put_Line (Integer' Image (Foo));
    Put_Line (Integer' Image (Bar));
end Check;
```

(in Datei `Check.adb` schreiben, kompilieren mit  
`gnatmake Check.adb`)

und (C++)

## Aufgaben zu Parameter-Modi (II)

Durch welchen Aufruf kann man diese beiden Unterprogramme semantisch voneinander unterscheiden:

Funktion (C++): (call by reference)

```
void swap (int & x, int & y)
    { int h = x; x = y; y = h; }
```

Makro (C): (call by name)

```
#define swap(x, y) \
    { int h = x; x = y; y = h; }
```

Kann man jedes der beiden von copy-in/copy-out unterscheiden?

# Lokale Unterprogramme

Unterprogramme sind wichtiges Mittel zur Abstraktion  
das möchte man überall einsetzen  
also sind auch lokale Unterprogramme wünschenswert  
(Konzepte *Block* und *Unterprogramm* sollen orthogonal sein)  
Dann entsteht Frage: Wie greifen lokale Unterprogramme auf  
nichtlokale Variablen zu?

# Statische und dynamische Sichtbarkeit

Was druckt dieses Programm?

```
int main () {  
    int x = 4;  
    int f(int y) { return x+y; }  
    int g(int x) { return f(3*x); }  
    printf ("%d\n", g(5) );  
}
```

- ▶ statische Sichtbarkeit: textuell umgebender Block (Pascal, Ada, Scheme-LISP, Haskell ...)
- ▶ dynamische Sichtbarkeit: Aufruf-Reihenfolge ((Common-LISP), (Perl))

Übung: Perl-Beispiel (local/my)

# Frames, Ketten

Während ein Unterprogramm rechnet, stehen seine lokalen Daten in einem Aktivationsverbund (Frame), jeder Frame hat zwei Vorgänger:

- ▶ dynamischer V. (Frame des aufrufenden UP) (benutzt zum Rückkehren)
- ▶ statischer V. (Frame des textuell umgebenden UP) (benutzt zum Zugriff auf "fremde" lokale Variablen)

Beispiel: zeichnen Frames und statische/dynamische Links für `a(3, 4)` bei

```
int a (int x, int y) {  
    int b (int z) { return z > 0 ? 1 + b (z-1) : x; }  
    return b (y);  
}
```

Übung: Assemblercode verstehen (`gcc -S`)

## Unterprogramme als Argumente

```
int d ( int g(int x) ) { return g(g(1)); }
```

```
int p (int x) {  
    int f (int y) { return x + y ; }  
    return d (f);  
}
```

Betrachte Aufruf  $p(3)$ .

Das innere Unterprogramm  $f$  muß auf den  $p$ -Frame zugreifen, um den Wert von  $x$  zu finden.

Dieser Frame lebt.

Wenn Unterprogramme nur “nach innen” als Argumente übergeben werden, können die Frames auf einem Stack stehen.

Übung: Assemblercode verstehen

# Unterprogramme als Resultate

```
int x1 = 3;
```

```
int (*s (int foo)) (int x2) {  
    int f (int y) { return x1 + y; }  
    return &f;  
}
```

```
int main (int argc, char ** argv) {  
    int (*p) (int) = s(4);  
    printf ("%d\n", (*p)(3));  
}
```

In `f` ersetze `x1` durch `x2`.

Assemblercode erklären.

# Unterprogramme/Zusammenfassung

in prozeduralen Sprachen:

- ▶ alle UP global: dynamische Kette reicht
- ▶ lokale UP: benötigt auch statische Kette
- ▶ lokale UP as Daten: benötigt Closures = (Code, statischer Link)
- ▶ UP als Argumente: Closures auf Stack
- ▶ UP als Resultate: Closures im Heap

vgl. <http://www.function-pointer.org/>

in objektorientierten Sprachen: keine lokalen UP, aber lokale (inner, nested) Klassen.

# Polymorphie

poly-morph = viel-gestaltig

ein Bezeichner (z. B. Unterprogrammname) mit mehreren Bedeutungen

Formen der Polymorphie:

- ▶ ad-hoc:  
einfaches Überladen von Bezeichnern
- ▶ parametrisch (und statisch):  
Typparameter für generische Klassen und Methoden
- ▶ dynamisch:  
Auswahl der Methoden-Implementierung durch Laufzeittyp des Objektes

# Objekte, Methoden

Motivation: Objekt = Daten + Verhalten

```
typedef struct {  
    int x; int y; // Daten  
    void (*print) (FILE *fp); // Verhalten  
} point;  
point *p; ... ; (*(p->print))(stdout);
```

# Klassen

Motivation: Klasse = gemeinsame Datenform und Verhalten von Objekten

```
typedef struct {  
    int (*method[5]) ();  
} obj;  
obj *o; ... (*(o->method) [3]) ();
```

allgemein: Klasse:

- ▶ Deklaration von Daten (Attributen)
- ▶ Deklaration und Implementierung von Methoden.

# this

Motivation: Methode soll wissen, für welches Argument sie gerufen wurde

```
typedef struct {  
    int (*data) [3];  
    int (*method[5]) ();  
} obj;  
obj *o; ... (*(o->method) [3]) (o);
```

```
int sum (obj *this) {  
    return this->data[0] + this->data[1];  
}
```

jede Methode bekommt *this* als (verstecktes) erstes Argument

# Vererbung

Klasse  $D$  abgeleitet von Klasse  $C$ :

- ▶  $D$  kann Menge der Attribute- und Methodendeklarationen von  $C$  erweitern (aber nicht verkleinern oder ändern)
- ▶  $D$  kann Implementierungen von in  $C$  deklarierten Methoden übernehmen oder eigene festlegen (überschreiben).

# Dynamische Polymorphie (OO)

```
class C {  
    int x = 2; int p () { return this.x + 3; }  
}  
C x = new C() ; int y = x.p ();
```

## Überschreiben:

```
class E extends C {  
    int p () { return this.x + 4; }  
}  
C x =                // statischer Typ: C  
    new E() ; // dynamischer Typ: E  
int y = x.p ();
```

Wo ein Objekt der Basisklasse (C) erwartet wird, kann ein Objekt einer abgeleiteten Klasse (E) benutzt werden.

# Equals richtig implementieren

```
class C {  
    final int x; final int y;  
    C (int x, int y) { this.x = x; this.y = y; }  
    int hashCode () { return this.x + 31 * this.y; }  
}
```

nicht so:

```
public boolean equals (C that) {  
    return this.x == that.x && this.y == that.y;  
}
```

## Equals richtig implementieren (II)

...sondern so:

```
public boolean equals (Object o) {  
    if (! (o instanceof C)) return false;  
    C that = (C) o;  
    return this.x == that.x && this.y == that.y;  
}
```

Die Methode `boolean equals(Object o)` wird aus `HashSet` aufgerufen.

Sie muß deswegen *überschrieben* werden.

Das `boolean equals (C that)` hat den Methodenamen nur *überladen*.

# Überladen

- ▶ wenn mehrere Methoden gleiche Namens sichtbar sind, wird die vom *passenden* Typ ausgewählt.
- ▶ wenn mehrere vom passenden Typ sichtbar sind (z. B. wegen Vererbungs-Beziehungen), dann wird die mit dem *spezifischsten* Typ ausgewählt
- ▶ ... falls das genau eine ist (ein minimales Element in der entsprechenden Halbordnung)— sonst Fehler

# Überschreiben und Überladen

- ▶ C++: Methode, die überschrieben werden darf, muß `virtual` deklariert werden
- ▶ C#: Das Überschreiben muß durch `override` angezeigt werden
- ▶ Java: alle Methoden sind `virtual`, deswegen ist Überschreiben von Überladen schlecht zu unterscheiden: Quelle von Programmierfehlern
- ▶ IDEs unterstützen Annotation `@overrides`

# Vererbung bricht Kapselung

```
class C {  
    void p () { ... q(); ... };  
    void q () { .. };  
}
```

Jetzt wird `q` überschrieben (evtl. auch unabsichtlich—in Java),  
dadurch ändert sich das Verhalten von `p`.

```
class D extends C {  
    void q () { ... }  
}
```

Man muß beim Programmieren von `D` wissen, wie `C.p()`  
implementiert ist!

# Statische Attribute und Methoden

für diese findet *kein* dynamischer Dispatch statt.

(Beispiele—Puzzle 48, 54)

Damit das klar ist, wird dieser Schreibstil empfohlen:

- ▶ dynamisch: immer mit Objektnamen qualifiziert, auch wenn dieser `this` lautet,
- ▶ statisch: immer mit Klassennamen qualifiziert (niemals mit Objektnamen)

# Lokale Klassen

- ▶ **static nested class:** dient lediglich zur Gruppierung

```
class C { static class D { .. } .. }
```

- ▶ **nested inner class:**

```
class C { class D { .. } .. }
```

jedes D-Objekt hat einen Verweis auf ein C-Objekt ( $\approx$  statische Kette) (bezeichnet durch `C.this`)

- ▶ **local inner class:** ( Zugriff auf lokale Variablen in *m* nur, wenn diese final sind. Warum? )

```
class C { void m () { class D { .. } .. } }
```

# Generische Polymorphie

parametrische Polymorphie:

- ▶ Klassen und Methoden können Typ-Parameter erhalten.
- ▶ innerhalb der Implementierung der Klasse/Methode wird der formale Typ-Parameter als (unbekannter) Typ behandelt
- ▶ bei der Benutzung der Klasse/Methode müssen alle Typ-Argumente angegeben werden (oder der Compiler inferiert diese in einigen Fällen)
- ▶ separate Kompilation (auch von generischen Klassen) mit statischer Typprüfung

## Bsp: Generische Klasse in Java

```
class Pair<A,B> {  
    final A first; final B second;  
    Pair(A a, B b)  
        { this.first = a; this.second = b; }  
}  
Pair<String,Integer> p =  
    new Pair<String,Integer>("foo", 42);  
int x = p.second + 3;
```

vor allem für Container-Typen (Liste, Menge, Keller, Schlange, Baum, ...)

## Bsp: Generische Methode in Java

```
class C {  
    static <A,B> Pair<B,A> swap (Pair<A,B> p) {  
        return new Pair<B,A>(p.second, p.first);  
    }  
}  
  
Pair<String,Integer> p =  
    new Pair<String,Integer>("foo", 42);  
Pair<Integer,String> q = C.swap(p);
```

bei Benutzung werden Typargumente inferiert  
in C#: Typargumente können auch angegeben werden.

```
Pair<Integer,String> q =  
    C.swap<String,Integer>(p);
```

# Schranken für Typparameter (I)

- ▶ **kovariant:** `<T extends S>`  
als Argument ist jeder Typ *T* erlaubt, der *S* implementiert

```
interface Comparable<T>
    { int compareTo(T x); }
static <T extends Comparable<T>>
    T max (Collection<T> c) { .. }
```

## Schranken für Typparameter (II)

- ▶ **kontravariant:** `<S super T>`  
Als Argument ist jeder Typ `S` erlaubt, der Obertyp von `T` ist.

```
static <T> int binarySearch  
    (List<? extends T> list, T key,  
     Comparator<? super T> c)
```

## Anonyme Typen (Wildcards)

Wenn man einen generischen Typparameter nur einmal braucht, dann kann er `?` heißen.

```
List<?> x = Arrays.asList  
    (new String[] {"foo", "bar"});  
Collections.reverse(x);  
System.out.println(x);
```

jedes Fragezeichen bezeichnet einen anderen (neuen) Typ:

```
List<?> x = Arrays.asList  
    (new String[] {"foo", "bar"});  
List<?> y = x;  
y.add(x.get(0));
```

# Wildcards und Bounds

```
List<? extends Number> z =  
    Arrays.asList(new Double[]{1.0, 2.0});  
z.add(new Double(3.0));
```

# Typinformation zur Laufzeit

- ▶ (Java) nicht vorhanden (type erasure), die Collection weiß nicht, welches ihr Elementtyp ist
- ▶ (C#) ist bekannt

Typen haben aber verschiedene Repräsentationen (verschieden primitive, Verweistypen), also:

- ▶ Java: alle Typparameter müssen Verweistypen sein (ggf. (Auto)boxing); nur eine Instanz der Collection
- ▶ C#: beliebige Typen gestattet (auch primitive); dafür dann mehrere Instanzen.

# Generics und Subtypen

Warum geht das nicht:

```
class C { }
```

```
class E extends C { void m () { } }
```

```
List<E> x = new LinkedList<E> ();
```

```
List<C> y = x; // Typfehler
```

Antwort: wenn das erlaubt wäre, dann:

# Generics und Arrays

das gibt keinen Typfehler:

```
class C { }  
class E extends C { void m () { } }
```

```
E [] x = { new E (), new E () };  
C [] y = x;
```

```
y [0] = new C ();  
x [0].m();
```

aber ... (Übung)

## Generics und Arrays (II)

warum ist die Typprüfung für Arrays schwächer als für Collections?

Historische Gründe. Das sollte gehen:

```
void fill (Object[] a, Object x) { .. }  
String [] a = new String [3];  
fill (a, "foo");
```

Das sieht aber mit Generics besser so aus: ...

# Einleitung

Abstraktion:

- ▶ (mehrere) Einzelheiten zusammenfassen
- ▶ und gleichzeitig verstecken

typische Formen:

- ▶ Vorgangs-Abstraktion (Unterprogramme)
- ▶ Daten-Abstraktion (Typen) —uraltet Beispiel(!): `double`
- ▶ gemischt (Klassen), übergreifend (Pakete, Namensbereiche)

# Polymorphie

Abstraktionen sind nur dann sinnvoll, wenn sie flexibel benutzt werden können, `interface Map<K, V>` für beliebige Typen  $K, V$ .

D. h.: Abstraktionen benötigen Parameter. Häufig:

- ▶ Unterprogramm bekommt Argumente (Daten)
- ▶ Typschablone bekommt Argumente (Typen)

auch andere denkbar und nützlich:

- ▶ Unterprogramm bekommt Typ als Argument (statisch oder dynamisch)
- ▶ Typschablone bekommt Datum als Argument (dependent types)

# Klassen, Interfaces

## abstrakter Datentyp: Operationen mit Zusicherungen

```
interface Map<K,V> {  
    void put (K key, V value);  
    V get (K key);  
}  
class TreeMap<K,V> implements Map<K,V> { ... }  
class HashMap<K,V> implements Map<K,V> { ... }
```

**... ist aus der Mathematik längst bekannt:**

```
interface Halbgruppe<M> { M mal (M x, M y); }  
interface Monoid<M> extends Halbgruppe<M>  
    { M eins (); }  
interface Gruppe<M> extends Monoid<M>  
    { M inverse (M x); }
```

# Algebraische Spezifikation

- ▶ Interface = abstrakter Datentyp
- ▶ besitzt algebraische Spezifikation = Signatur und Axiome (Formeln)
- ▶ Implementierung = konkreter Datentyp = eine Algebra = Modell einer Spezifikation

Literatur: z. B. Ehrich, Gogolla, Lipeck: *Algebraische Spezifikation abstrakter Datentypen*. Teubner, Stuttgart, 1989.

# Signaturen, Algebren

- ▶ *Signatur*: Menge von Funktionssymbolen, jedes mit Stelligkeit (Argumentzahl).

Bsp:  $\Sigma = \{a_0, b_0, f_2, g_2\}$ .

- ▶  $\Sigma$ -*Algebra*: Trägermenge  $D$  und zu jedem  $k$ -stelligen Symbol  $f$  eine  $k$ -stellige Funktion  $[f] : D^k \rightarrow D$ .

Bsp:

$D = \mathbb{N}$ ,  $[a] = 0$ ,  $[b] = 1$ ,  $[f](x, y) = x + y$ ,  $[g](x, y) = xy$ .

Bsp:  $D = \{F, T\}$ ,  $[a] = F$ ,  $[b] = T$ ,  $[f] = \vee$ ,  $[g] = \wedge$ .

# Axiome, Modelle

*Axiom:* prädikatenlogische Formel, die (nur) Funktionssymbole der Signatur benutzt:

- $$\begin{aligned} & \forall x : F(A, x) = x \wedge F(x, A) = x \\ \wedge & \quad \forall x, y, z : F(F(x, y), z) = F(x, F(y, z)) \\ \wedge & \quad \forall x : G(B, x) = x \wedge G(x, B) = x \\ \wedge & \quad \forall x : G(A, x) = A \wedge G(x, A) = A \\ \wedge & \quad \forall x, y, z : G(G(x, y), z) = G(x, G(y, z)) \\ \wedge & \quad \forall x, y, z : G(F(x, y), z) = F(G(x, z), G(y, z)) \\ \wedge & \quad \forall x, y, z : G(x, F(y, z)) = F(G(x, y), G(x, z)) \end{aligned}$$

*Modell:* eine  $\Sigma$ -Algebra, die die Axiome erfüllt.

Ü: Gesucht: *alle* Modelle von  $A$  mit Träger  $\{F, T\}$ .

# ADT in realen Programmiersprachen

- ▶ Signatur: Interface (statische Prüfung)
- ▶ Axiome: Vor/Nachbedingungen von Methoden (Prüfung dynamisch: Test, statisch: Beweis)

Realisierung von Interfaces in OOP:

- ▶ interface ist (voll) abstrakte Klasse (alle Methoden sind abstrakt: nur deklariert, aber nicht implementiert)
- ▶ Schnittstellen-Implementierung: konkrete Klasse „erbt“ von interface
- ▶ das ist *keine* Implementierungsvererbung

# Module, Komponenten

- ▶ Softwareprojekte bestehen aus Teilen (Modulen, Komponenten), die getrennt spezifiziert, implementiert, angewendet werden.
- ▶ Programmiersprache sollte Entwicklung von Teilprogrammen unterstützen
- ▶ getrennte Typprüfung/Kompilation von Modulen: benötigt Spezifikation von anderen Modulen.

Designfrage: Spezifikation von Implementierung trennen?

- ▶ Ja: Ada (.ads/.adb), C (.h/.c)
- ▶ Nein: Java, C#

# Build-Systeme

Designfrage: Modulabhängigkeiten in der Sprache oder außerhalb der Sprache?

Wenn außerhalb, dann wird zusätzlich zu Compiler noch ein Build-System benötigt.

- ▶ C: Abhängigkeiten in Makefiles, können generiert werden (`gcc -M`)
- ▶ Java: Abhängigkeiten in Quellen (`import`) trotzdem Suchpfade für Pakete  
Build-Systeme: ant, maven

# Sichtbarkeiten

Information dadurch verstecken, daß sie keinen Namen hat:  
Java: Bezeichner sind sichtbar

- ▶ lokal: nur im Block
- ▶ private: nur in eigener Klasse
- ▶ (default): nur im eigenen Paket
- ▶ protected: in abgeleiteten Klassen
- ▶ public: in allen Paketen

# Namen (Designfragen)

- ▶ wodurch entsteht Namens-Hierarchie?  
Java: package, class; C++: namespace, class
- ▶ muß man die Benutzung eines Namens deklarieren?  
Ada: `with Text_IO; ... Text_IO.Put (..)`  
Java: `java.util.TreeMap<Foo, Bar>`
- ▶ kann der voll qualifizierte Name abgekürzt werden?  
Ada: `with Text_IO; use Text_IO; Put (..)`  
Java: `import java.util.*; TreeMap<Foo, Bar>`
- ▶ sind Abkürzungen global oder lokal?  
C++: `using namespace ...`

# Themen

- ▶ formale Methoden zur Beschreibung von
  - ▶ Syntax (lexikalisch, grammatikalisch, abstrakt)
  - ▶ Semantik (statisch: Attributgrammatiken, dynamisch: denotational, operational, axiomatisch)
- ▶ Anwendung bei:
  - ▶ Typen
  - ▶ Ausdrücke, Anweisungen
  - ▶ Unterprogramme
  - ▶ Polymorphie (OO, parametrisch)

# Hierarchien (Bäume)

- ▶ Baum: Blatt/Verzweigung (Kompositum)
- ▶ Syntax: einfache/zusammengesetzte Ausdrücke, Anweisungen
- ▶ Typen und ihre Elemente (Daten): einfach/zusammengesetzt
- ▶ abstrakte Datentypen (Signatur einer Algebra) (Bsp: reguläre Ausdrücke)

# Statik/Dynamik

## Definition:

- ▶ statisch: Analyse (+ Übersetzung) des Programmes
- ▶ dynamisch: Ausführung des Programmes

## Eigenschaften:

- ▶ (Vor-)Arbeit zur Übersetzungszeit spart (Nach-)Arbeit zur Laufzeit (Bsp: Typprüfungen, Bereichsprüfungen, Initialisierungs-Prüfungen)
- ▶ statische Prüfungen sollen *entscheidbar* sein, deswegen nicht für jede Programmeigenschaft anwendbar (Sätze von Gödel/Church/Turing/Rice)
- ▶ Auswege: Compiler approximiert, Programmierer hilft mit.

# Abstraktionen

Programmiersprache dient dazu, ...

- ▶ ausführbaren Code zu beschreiben,
- ▶ Abstraktionen auszudrücken.  
(allgemeine und anwendungsspezifische)

Abstraktion besteht aus:

- ▶ Schema (das Gemeinsame)
- ▶ Parametern (das Unterschiedliche).

Bsp: Unterprog., Klasse, abstr. Datentyp, generischer Typ.  
zugrundeliegendes Prinzip: Funktion. ( $\Rightarrow$  Lambda-Kalkül) vgl.  
Steele, Sussman: "Lambda, the ultimate ..."

<http://library.readscheme.org/page1.html>