

Prinzipien von Programmiersprachen

Vorlesung

Wintersemester 2006

Johannes Waldmann, HTWK Leipzig

16. Januar 2007

Einleitung

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
dienstags, 7:30–9:00, HB207
- Übungen
 - freitags 12:00–13:30, Z423
 - freitags 17:15–18:45, Z423
 - (neu) dienstags (ungerade) 11:00–12:30, Z423 *und*
donnerstags (gerade) 19:00–20:30, Z423

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

Literatur

- `http://www.imn.htwk-leipzig.de/~waldmann/edu/ws06/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  
find workspace/ -name "*.java" -exec grep
```

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
```

Kellermaschine in PostScript.

```
42 42 scale 7 9 translate .07 setlinewidth .
setgray}def 1 0 0 42 1 0 c 0 1 1{0 3 3 90 27
arcn 270 90 c -2 2 4{-6 moveto 0 12 rlineto}
9 0 rlineto}for stroke 0 0 3 1 1 0 c 180 rot
```

Mit `gv` oder `kghostview` ansehen (Options: `watch file`).

Mit Editor Quelltext ändern. Finden Sie den Autor dieses Programms!

(Lösung: John Tromp, siehe auch

<http://www.iwriteiam.nl/SigProgPS.html>)

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
gcc -o bar bar.o # linkt (lädt) Objektcode (
./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)

gcc für Java (gcj):

```
gcj -c Foo.java # erzeugt Objektcode
```

```
gcj -o Foo Foo.o --main=Foo # linken, wie ob
```

- Assemblercode ansehen, vergleichen

```
gcj -S Foo.java # erzeugt Assemblercode (Fo
```

- Kompatibilität des Bytecodes ausprobieren zwischen
Sun-Java und GCJ (beide Richtungen)

```
gcj -C Foo.java # erzeugt Class-File (Foo.c
```

Syntax von Programmiersprachen

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

Zur Erinnerung: *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 Σ
(üblich: Kleibuchst., Ziffern)
- Variablen-Alphabet V
(üblich: Großbuchstaben)
- Startsymbol $S \in V$
- Regelmenge
(Wort-Ersetzungs-System)

Grammatik

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

$R \subseteq (\Sigma \cup V)^* \times (\Sigma \cup V)^*$
von G erzeugte Sprache: $L(G) = \{w \mid S \rightarrow^* w \wedge w \in \Sigma^*\}$. r

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:

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

Reguläre Sprachen/Ausdrücke

Die Menge $E(\Sigma)$ der *regulären Ausdrücke* über einem Alphabet (Buchstabenmenge) Σ 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 Anweis

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 $S \rightarrow_R^* w$ sind erlaubt, und wg. Kontextfreiheit auch gar nicht zu vermeiden.)

Die naheliegende Grammatik für arith. Ausdr.

`expr -> number | expr + expr | expr * 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`

- richtig oder falsch?

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

```Wadler's law of language design''`

- reguläre Ausdrücke

- kontextfreie Grammatiken

- eindeutige Grammatiken für arithmetische Ausdrücke

# Semantik von Programmiersprachen

## Statische und dynamische Semantik

Semantik = Bedeutung

- statisch (kann zur Übersetzungszeit geprüft werden)

Bsp: Typ-Korrektheit von Ausdrücken, Bedeutung (Bindung) von Bezeichnern

Hilfsmittel: Attributgrammatiken

- dynamisch (beschreibt Ausführung des Programms)

Operational, Axiomatisch, Denotational

# Attributgrammatiken

Attribut: Annotation an Knoten des Syntaxbaums.

- ererbt:

hängt nur von Attributwerten in Elternknoten ab

- synthetisiert:

hängt nur von Attributwerten in Kindknoten ab

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

# Attributgrammatiken–Beispiele

- Auswertung arithmetischer Ausdrücke (dynamisch)
- Typprüfung (statisch)
- Kompilation (für Kellermaschine) (statisch)

# 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 $\rightarrow$ Kellermaschine

Beispiel:

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

- Code für Konstante/Variable  $c$ : `push c;`
- Code für Ausdruck  $x$  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!

Beispiel: Java-Bytecode (javac, javap)



# Dynamische Semantik

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

# Bsp: Operationale Semantik

Schleife

```
while (B) A
```

wird übersetzt in Sprungbefehle

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

(vervollständige!)

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

# Axiomatische Semantik

## Hoare-Kalkül

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

(Wenn  $V$  gilt, dann  $A$  ausgeführt wird, gilt danach  $N$ .)

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

## Beispiele

- $\text{wp}(x := E, N) = N[x/E]$
- $\text{wp}(\text{if } (B) \text{ then } C \text{ else } D, N) = B \wedge \text{wp}(C, N) \vee \neg B \wedge \text{wp}(D, N)$
- Schleife ... benötigt Invariante

(siehe Vorlesung Verifikation)

# Denotationale Semantik

## Beispiele

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

# Übungen

- dangling else in Java, Ada
- Gleitkomma-Addition nicht assoziativ
- Java-Bytecode für arithmetische Ausdrücke  
(Programmrekonstruktion)

Schreiben Sie eine Java-Methode, deren Kompilation genau diesen Bytecode erzeugt. Was macht die Methode?

```
static int f(int, int);
```

Code:

```
0: iconst_0
```

```
1: istore_2
```

```
2: iload_1
```

```
3: ifle 17
6: iload_2
7: iload_0
8: iadd
9: istore_2
10: iload_1
11: iconst_1
12: isub
13: istore_1
14: goto 2
17: iload_2
18: ireturn
```

# Bezeichner, Bindungen, Bereiche

## Variablen

vereinfacht: Variable bezeichnet eine (logische)  
Speicherzelle

genauer: Variable besitzt Attribute

- Name
- Adresse
- Wert
- Typ
- Lebensdauer
- Sichtbarkeitsbereich

Bindungen dieser Attribute *statisch* oder *dynamisch*

# Namen

- welche Buchstaben/Zeichen sind erlaubt?
- reservierte Bezeichner?
- Groß/Kleinschreibung?
- Konvention: `long_name` oder `longName` (camel-case)  
(Fortran: `long name`)

im Zweifelsfall: Konvention der Umgebung einhalten

- Konvention: Typ im Namen (schlecht, weil so Implementierungsdetails verraten werden)

**schlecht:** `myStack = ...`

**besser:** `Stack<Ding> rest_of_input = ...`



# Typen für Variablen

- dynamisch
- statisch
  - deklariert (durch Programmierer)
  - inferiert (durch Übersetzer)
    - z. B. `let` in C#3

Vor/Nachteile: Lesbarkeit, Sicherheit, Kosten

bei Zuweisungen `lhs := rhs`

Wie genau muß Typ von `rhs` (Ausdruck) übereinstimmen mit deklariertem Typ von `lhs` (Variable)?

Beachte: Objekttypen, Zahltypen

# Konstanten

= Variablen, an die genau einmal zugewiesen wird

- C: `const` (ist Attribut für Typ)
- Java: `final` (ist Attribut für Variable)

Vorsicht:

```
class C { int foo; }
static void g (final C x) { x.foo ++; }
```

in funktionaler Programmierung (Haskell) sind *alle* „Variablen“ konstant und alle Objekte immutable.

das sollte man auch in imperativen Sprachen so weit wie möglich nachmachen.

Merksatz: alle Deklarationen so (lokal und so) konstant wie

möglich!

# Lebensort und -Dauer von Variablen

- statisch (global, aber auch lokal:)

```
int f (int x) {
 static int y = 3; y++; return x+y;
}
```

- dynamisch

- Stack { int x = ... }

- Heap

- \* explizit (new/delete, malloc/free)

- \* implizit

# Sichtbarkeit von Namen

= Bereich der Anweisungen/Deklarationen, in denen ein Name benutzt werden kann.

- global
- lokal: Block (und Unterblöcke)

Üblich ist: Sichtbarkeit beginnt *nach* Deklaration und endet am Ende des umgebenden Blockes

# Überdeckungen

Namen sind auch in inneren Blöcken sichtbar:

```
int x;
while (...) {
 int y;
 ... x + y ...
}
```

innere Deklarationen verdecken äußere:

```
int x;
while (...) {
 int x;
 ... x ...
}
```

# 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)



# Sichtbarkeit und Lebensdauer

... stimmen nicht immer überein:

- static-Variablen in C-Funktionen

sichtbar: in Funktion, Leben: Programm

- lokale Variablen in Unterprogrammen

sichtbar: innere Blöcke, Leben: bis Ende Unterpr.

# Typen

## 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

```
type Day is (Mon, Tue, Thu, Fri, Sat, Sun)
subtype Weekday is Day range Mon .. Fri;
```

```
procedure Day is
 X, Y : Day;
begin
 X := Fri; Text_Io.Put (Day' Image (X));
 Y := Succ(X); 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 [][] field =
 { {1,2,3}, {3,4}, {5}, {} };
for (int [] line : field) {
 for (int item : line) {
 System.out.print (item + " ");
 }
 System.out.println ();
}
```

dynamische Dimensionierung und Allokation,  
Bereichsprüfungen. Nicht notwendig rechteckig.

# 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

# 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);
}
```

# Probleme mit Zeigern

- verschiedene Namen für gleiche Objekte (aliasing) erschweren Programmanalyse
- tote Zeiger (wg. verfrühter Freigabe)

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

- Speichermüll (wg. vergessener Freigabe)

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

# 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
- nebenläufige Ausführung, Platzbedarf, Zeitgarantien

```
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 Zähler verringern (usw.)
- Zähler brauchen Platz
- Löschen ist aufwendig
- geht 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 nur ein Bit; Zeit  $\sim$  Speicher, auch bei viel Müll.
- 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)

# 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).

# 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
- Spezialbehandlung für Verweise von alt nach neu

Siehe auch:

Hans Boehm: Allocation and GC Myths [http://www.hp1.hp.com/personal/Hans\\_Boehm/gc/myths.ps](http://www.hp1.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/>

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

# Übung GC

Richtig viel Müll erzeugen:

```
import java.util.*;
class Garbage {
 static long sum (int top) {
 List<Integer> l = new LinkedList<Int
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+=10
 System.out.println (i + " : " +
)
}
}
```

**ausführen mit:**

```
time java -Xmx400M -XX:NewSize=40M -XX:+Pri
verschiedene Werte für NewSize ausprobieren,
Messungen erklären.
```

# Ausdrücke

## Definitionen

Semantik:

- Ausdruck hat Wert (wird ausgewertet) (Wert hat Typ)
- Anweisung hat Wirkung (wird ausgeführt)

Syntax:

- einfach: Konstante, Variable
- zusammengesetzt: Operator/Funktions-Symbol mit Argumenten

wichtige Spezialfälle für Operatoren:

- arithmetische, relationale, boolesche

# 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 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");
```

# 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

lieber

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

# 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  $p$

# Ausdrücke mit Nebenwirkungen

(*side effect*; falsche Übersetzung: Seiteneffekt)

- Zuweisungen

```
int a; int b = (a = 5) + (a = 6);
System.out.println (a);
```

- verkürzte Zuweisungen

```
a += b; <==> a = a + b; (wirklich?)
```

- Inkrement/Dekrement

```
int d = 3; System.out.println ((d++) - (++d))
```

# Reihenfolge von Nebenwirkungen

```
int x = 3;
printf ("%d %d %d\n", ++x, ++x, ++x);
```

```
int y = 3;
printf ("%d\n", ++y * ++y * ++y);
```

(Übung: swap meat!)

# 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)
- 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);`

# Anweisungen

## Programm-Ablauf-Steuerung

durch Zusammensetzen von Anweisungen:

- Nacheinanderausführung (Block)
- Verzweigung (zweifach: if, mehrfach: switch)
- Wiederholung (Sprung, Schleife)
- Unterprogramm-Aufruf

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 immer 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...

- Bedingung

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

- Zähler

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

- Daten

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

# Zählschleifen

- Idee: vor Beginn steht Anzahl der Durchläufe fest.
- dann erhält man die Klasse der primitiv rekursiven Funktionen, die terminieren immer!

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



# Schleifen mit Bedingungen

- das ist die allgemeinste Form, ergibt (partielle) rekursive Funktionen,
- die terminieren nicht immer

Steuerung am Anfang oder Ende:

```
while Bedingung Anweisung;
```

```
do Anweisung while Bedingung;
```

# Abarbeitung von Schleifen

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; ..

⇒

```
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;  $\Rightarrow$`

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

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



# setjmp/longjmp in C

Das wird wirklich nicht zur Nachahmung empfohlen:

```
#include <setjmp.h>
```

```
static jmp_buf buf;
```

```
int dup (int x) { return succ (2*x); }
```

```
int succ (int x) { longjmp (buf, 5); }
```

```
int main () {
```

```
 int x = setjmp(buf);
```

```
 int y = x ? 3*x : dup (x);
```

```
}
```

**siehe auch**

<http://en.wikipedia.org/wiki/Longjmp>

# Was ist hier los?

```
class What {
 public static void main (String [] args)
 System.out.println ("mozilla:open");
 http://haskell.org
 System.out.println ("mozilla:close")
 }
}
```

# Unterprogramme

## 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

# Beispiele für Unterprogramme (Funktionen)

$$f(x) = \text{if } x > 100 \text{ then } x - 10 \text{ else } f(f(x + 11))$$

$$t(x, y, z) = \text{if } x \leq y \text{ then } y \\ \text{else } t(t(x - 1, y, z), t(y - 1, z, x), t(z - 1, x, y))$$

Aufgaben:  $f(7)$ ,  $t(30, 20, 10)$ , allgemein äquivalente nicht rekursive Darstellung

Beobachtung: es ist gar nicht klar,...

- (denotationale Semantik) ... ob solche Gleichungen überhaupt eine oder genau eine Funktion als Lösung haben.
- (operationale Semantik) ... ob und nach welcher Zeit

man durch mutiges Ausrechnen Funktionswerte findet.

# 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 : Flo
Paint (width => 30, height => 40);
```

nach erstem benanntem Argument keine positionellen mehr erlaubt

code smell: lange Parameterliste,

refactoring: Parameterobjekt einführen

allerdings fehlt (in Java usw.) 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.

# Semantik von Unterprogrammen

Designfragen:

- Typprüfung
- generische Typen?
- Überladung von Namen?
- Wertübergabe
- lokale Unterprogramme erlaubt?
- Bürgerrechte für Unterprogramme?

# Typprüfungen

- Typen bei Argumentübergabe und Resultatrückgabe müssen *passen*.
- vgl. Diskussion bei Variablen (Initialisierung, Zuweisung).
- – Namensgleichheit
  - Konvertierbarkeit (long  $\rightarrow$  double? usw.)
  - Instantiierung einer Typschablone

# Generische Polymorphie

generische Methode:

```
static <E> int length (Collection<E> c) {
 int n = 0; for (E x : c) {n++}; return n;
}
```

hierbei ist  $E$  eine Typvariable

Bei Benutzung

```
Collection<String> text =
 Arrays.asList (new String [] { "foo", "b
System.out.println (length (text));
```

wird Typvariable instantiiert (durch einen konkreten Typ ersetzt).

# Das Überladen von Namen

*ein* Name mit *verschiedenen* Bedeutungen:

```
void p (String x) { ... }
```

```
void p (int x) { ... }
```

```
p ("foo"); p (4);
```

Java: Überladen gestattet, Auflösung durch Liste der Argumenttypen.

Warum geht das folgende nicht?

```
int f (String x) { ... }
```

```
boolean f (String x) { ... }
```

# 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

# 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++)**

```
#include <iostream>

void sub (int & x, int & y, int & z) {
 y = 8;
 z = x;
}

int main () {
 int foo = 9;
 int bar = 7;

 sub (foo, foo, bar);
 std::cout << foo << std::endl;
 std::cout << bar << std::endl;
}
```

}

# 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?

# Resultate der Umfrage zur Vorlesung

<http://www.imn.htwk-leipzig.de/~waldmann/edu/ws06/pps/umfrage/>

# Aspekte Funktionaler Programmierung

## Warum sind Funktionen wichtig?

- eine Funktion ist ein parametrisiertes Muster.
- die Theorie dafür ist der Lambda-Kalkül (Church, 1936)
- durch (Wieder-)Verwendung von Mustern kann man sich viel Arbeit sparen und Fehler vermeiden
- das hat immense softwaretechnische Bedeutung
- vgl. code smell “duplicated code”, refactoring “introduce method”.



# Übersicht

- Funktionen als Daten
- strenges, polymorphes Typsystem
- keine Zuweisungen (nur Deklarationen mit Initialisierung)
- Bedarfs-Auswertung

# Funktionen als Daten

ein Datum (Objekt) kann man

- speichern (in Variable, in Struktur)
- übergeben (an und von Unterprogramm)

eine Funktion, deren erstes Argument eine Funktion ist:

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

```
map f xs =
 if null xs then []
 else f (head xs) : map f (tail xs)
```

**zweites Argument: Liste von Funktionen**

```
map (\ f -> f 3) [sin, cos, log]
```

# Funktionen als Daten (II)

in anderen Sprachen verfügbar:

- C#: delegates
- C# 3: anonyme Funktionen (Lambda-Ausdrücke)

oder simuliert (Java): Beispiel Strategie-Muster:  
ein Objekt, das das gewünschte Unterprogramm als  
(einzige) Methode hat,  
vgl. interface ActionListener

# Typsysteme

- statisch (Prüfung zur Übersetzungszeit, nicht zur Laufzeit)
- generisch polymorph

Bsp: allgemeinsten Typ von `map`?

vgl. generische Polymorphie in C++ (STL), Java (Collections Framework)

LISP, Prolog haben dynamisches Typsystem;  
ML, Haskell, Mercury statisches.

# Deklarative Programmierung

in streng funktionale Sprachen sind alle „Variablen“ tatsächlich Konstanten.

- nur Deklaration mit Initialisierung
- keine Zuweisung, keine Nebenwirkungen

vgl. Code-Richtlinien:

- Unveränderliche Variablen als solche deklarieren
- unveränderliche (immutable) Klassen sind einfacher zu beherrschen
- vgl. zustandsfreie Web-Service-Objekte (Beans)
- vgl. refactoring “introduce state object”

# Eifrige (eager) Auswertung

Das geht gut:

```
int fac (int x) {
 if (x>0) { return x * fac (x-1); }
 else { return 1; }
}
```

aber das nicht (in Java usw.):

```
int if_positive (int x, int j, int n) {
 if (x > 0) { return j; } else { return n
}

int fac (int x) {
 return if_positive (x, x*fac(x-1), 1);
}
```

(welche Lösung/Hack in C?)

# Bedarfs-Auswertung (faul, lazy)

- in den üblichen imperativen Sprachen werden alle Argumente vor Funktionsaufruf komplett ausgewertet
- das ist bei manchen Umformungen hinderlich (zu eifrige Auswertung)
- in manchen funktionalen Sprachen (LISP und ML nein, Haskell ja) werden alle Argumente erst *bei Bedarf* ausgewertet
- das ist 1. effizient und 2. unsichtbar (es gibt eine Nebenwirkungen dieser Auswertungen)

# Unendliche Datenstrukturen

Argumente für Konstruktoren werden auch verzögert ausgewertet, deswegen kann man mit unendlichen Listen, Bäumen usw. rechnen.

```
nats :: [Int]
nats = nats_from 0 where
 nats_from n = n : nats_from (n+1)
```

Das geht gut, solange man sich immer nur einen endlichen Teil ansieht

```
take 10 nats -- OK
take 10 $ map (\ x -> x * x) nats -- OK
length nats > 20 -- nicht OK
```

Operationen auf Maschinenzahlen sind strikt (erfordern ausgewertete Argumente)



# Producer/Transformer/Consumer

```
sum $ map (\ x -> x ^ 3) $ take 20 nats
```

- Datenstrukturen (hier: Listen) werden erzeugt, transformiert, verbraucht.
- die o.g. Notation ist dafür die direkteste,
- bei eifriger Auswertung verbraucht das zuviel Platz,
- man müßte das Programm umschreiben, dadurch wird es unübersichtlich,
- bei Bedarfs-Auswertung benötigt o.g. Programm genauso wenig Platz wie von Hand umgeschriebenes

# Programmtransformationen

der Compiler kann Programme umformen, damit Objekte, die gleich wieder konsumiert werden, gar nicht erst erzeugt werden, z. B.

$$\text{map } f \ . \ \text{map } g = \text{map } (f \ . \ g)$$

Umformungen (refactoring) leicht möglich, da keine Nebenwirkungen zu berücksichtigen.

Dabei ist  $(.)$  die Funktionskomposition

$$(.) \ :: \ \dots$$

$$(f \ . \ g) \ x = f \ (g \ (x))$$

(und zwar falschherum, würde jeder Algebraiker sagen)

# Fktl. Modellierg. von Zustandsänderungen

wie schreibt man in Haskell auf die Konsole? Das ist eine Zustandsänderung!

```
main = putStrLn $ show $ take 20 nats
```

es gibt eine Typschablone `IO a` für Aktionen mit Resultattyp `a`.

```
main :: IO () -- entspr. "void" (kein Result
```

```
readFile :: FilePath -> IO String
```

```
writeFile :: FilePath -> String -> IO ()
```

solche Aktionen kann man wie Daten verwalten, aber zur Ausführung muß man eine Reihenfolge angeben.

das Typsystem unterscheidet streng `IO Int` (Aktion) von `Int` (Wert), das garantiert Nebenwirkungsfreiheit und damit Sicherheit.

# Zusammenfassung

- nicht alle „funktionalen“ Sprachen haben alle genannten Eigenschaften
- auch in modernen imperativen Sprachen kann man mehr oder weniger funktional programmieren
- dieser Stil wird zunehmend durch Richtlinien, Entwurfsmuster, sprachliche Mittel unterstützt
- und wenn nicht: . . . a good programmer can write LISP in any language
- Nachteil heutiger funktionaler Programmierung: ist noch zu stark an konkreten Datentypen (z. B. Listen) orientiert,

(Haskell hat kein allgemein akzeptiertes abstraktes  
Collections-Framework)

# Nebenläufige Programme

## Einleitung

Sebesta: Conc. of Prog. Lang., Kap. 13

Nebenläufigkeit: (scheinbar) gleichzeitiges Ausführen von

- Befehlen der Maschinensprache
- Befehlen der Hochsprache
- Unterprogrammen
- Programmen

Arten der Nebenläufigkeit:

- physische NL (mehrere Prozessoren)
- logische NL (mehrere virtuelle P.)

# Nebenläufige Unterprogramme

(Task, Thread, Prozeß)

Datenaustausch über

- gemeinsame (nicht-lokale) Variable
- message passing
- Parameter

Probleme:

- ohne Synchronisation: inkonsistente Daten (siehe Beispiel)
- mit Synchronisation: . . .

# Shop-Simulation

Ein Angestellter (clerk) verkauft Brötchen an Kunden (customer):

```
public class Customer implements Runnable {
 public void run() {
 for (int i = 0; i < items; i++) {
 int x = clerk.get();
 } }
public class Clerk { ...
 private int current = 0;
 public int get() {
 int result = current; current++;
 return result;
 } }
```



jedes Brötchen soll genau einmal verkauft werden

# Shop-Simulation (II)

```
public static void main(String[] args) {
 Clerk bäcker = new Clerk ();
 for (int i=0; i<3; i++) {
 Customer c = new Customer
 (Integer.toString(i), bäcker, 10);
 new Thread(c).start();
 }
}

public static void abwarten() {
 int d = (int) Math.random () * 1000;
 Thread.sleep(d);
}
```

# Ziele der Synchronisation

- gegenseitiger Ausschluß (competition synch.)  
bei gleichzeitigem Versuch des Zugriffs auf Daten
- Kommunikation (cooperation synch.)  
Datentransport

# Synchronisation durch Semaphore

(Dijkstra, ca. 1960)

Semaphor enthält

- natürliche Zahl
- Warteschlange (Queue) für Tasks

Operationen

- *wait*: falls Zähler  $> 0$ , dann um eins verringern, falls Zähler  $= 0$ , dann aufrufende Task in Queue und blockieren
- *release*: falls Queue leer, dann Zähler erhöhen, falls Queue nicht leer, dann eine Task aus Queue entfernen und aktivieren

# Semaphore (II)

- Operationen müssen atomar sein, Realisierung ist Aufgabe des Betriebs/Laufzeitsystems
- S. können Aufgabe des gegenseitigen Ausschlusses (exklusive Ausführung eines Programmteils) lösen
- sind anfällig gegen Programmierfehler
- werden versteckt als Grundlage besser strukturierter Lösungen benutzt
- Bsp: synchronisierte Methoden in Java  
zu jedem Zeitpunkt darf höchstens ein Thread eine synchronized Methode eines Objektes ausführen.

# Fünf Philosophen

ein klassisches Beispiel für multi-threading:

- 5 Philosophen.  
Jeder einzelne: denkt, wird hungrig, ißt, denkt, wird hungrig, ißt, ...
- runder Tisch, in der Mitte ein Topf Spaghetti (wird nie alle)
- es gibt zwischen je zwei Tellern genau eine Gabel, zum Essen benötigt man beide.

Die Ressourcen (Gabeln) müssen den Clients (Philosophen) sinnvoll zugeteilt werden, *so daß keiner verhungert.*

# Gefahr 1: Deadlock (Verklemmen)

- wechselseitiger Ausschluß (jede Ressource von nur einem Prozeß benutzbar)
- besitzen und warten (Prozeß besitzt bereits Ressourcen und wartet auf weitere)
- kein Ressourcenentzug (kein anderer als der Besitzer-Prozeß selbst kann die Ressource freigeben)  
(beachte: bis hierher sind es sehr vernünftige Forderungen)
- Zyklisches Warten (jeder Prozeß besitzt wenigstens eine Ressource, auf die ein andere wartet)

## **Gefahr 2: Unfairness (Verhungern)**

falls es einen Ablauf der Ereignisse gibt, bei dem wenigstens ein Prozeß nie die gewünschten Ressourcen erhält.

(... weil ihm die anderen abwechselnd alles wegnehmen)



# Deadlock?

Dieser Ansatz hat Deadlock (ausprobieren):

```
class Philo {
 void run () {
 while (true) {
 System.out.println (id() + " hat Hunger");
 right.take (this);
 left.take (this);
 System.out.println (id() + " ist satt");
 right.drop (this);
 left.drop (this);
 } } }
```

Wie kann man das verhindern?

# Eine Gabel (im Prinzip)

```
class Fork {
 private int me;
 String id () { return "Fork-" + me; }
 private boolean taken;
 Fork (int m) { me = m; taken = false; }

 synchronized void drop (Philo p) { .. }
 synchronized void take (Philo p) { .. }
}
```

# Ein Philosoph (im Prinzip)

```
class Philo implements Runnable {
 private int me;
 String id () { return "Philo-" + me; }
 private Fork left; private Fork right;
 Philo (int m, Fork l, Fork r) {
 me = m; left = l; right = r;
 }
 public void run () {
 while (true) {
 right.take (this); left.take (this);
 right.drop (this); left.drop (this);
 }
 }
}
```

}

# Das Hauptprogramm

```
class Philo { ...
 public static void main (String [] argv) {
 int size = 5;
 Fork fo [] = new Fork [size];
 for (int i=0 ; i<size; i++) {
 fo[i] = new Fork (i);
 }
 Philo ph [] = new Philo [size];
 for (int i=0 ; i<size; i++) {
 ph[i] = new Philo
 (i, fo[i], fo[(i+1) % size]);
 new Thread (ph[i]).start ();
 } } }
```

# Eine Gabel (genauer)

```
class Fork {
 boolean taken;
 synchronized void drop (Philo p) {
 taken = false;
 notify ();
 }
 synchronized void take (Philo p) {
 while (taken) {
 wait ();
 }
 taken = true;
 }
}
```

# Eine Gabel (`wait/notify`)

Durch `wait()` wird der aufrufende Thread angehalten (er wartet auf das Objekt (die Gabel)).

Jedes Objekt hat eine Warteschlange von Threads.

Durch `notify` wird *irgendein* Thread aus der Warteschlange ausgewählt und fortgesetzt.

# Synchrone Kommunikation (Rendezvous)

```
task body Server is
```

```
 Sum : Integer := 0;
```

```
begin loop
```

```
 accept Foo (Item : in Integer)
```

```
 do Sum := Sum + Item; end Foo;
```

```
 accept Bar (Item : out Integer)
```

```
 do Item := Sum; end Bar;
```

```
end loop;
```

```
end Server;
```

```
A : Server; B : Integer;
```

```
begin
```

```
 A.Foo (4); A.Bar (B); A.Foo (5); A.Bar (B
```

```
end B;
```



# Rendezvous (II)

- beide Partner müssen aufeinander warten
- implementiert in Ada
- allgemeinere Form

```
select
 when X < Y =>
 accept Foo (Item : in Integer) do .. e
 or when Y < Z =>
 accept Bar (...)
end select;
```

- viele Fehlermöglichkeiten, Korrektheit nicht offensichtlich
- Server sollte zustandslos sein

# Datentransport durch Kanäle

- häufig gibt es eine Trennung:
  - Prozesse erzeugen Daten (producer)
  - Prozesse verbrauchen Daten (consumer)
- Verbindung durch einen Kanal (vgl. Unix: Pipe)
- beschränkte Kapazität des Kanals führt auch zu Synchronisation (producer blockiert, wenn Kanal voll ist)
- mehrere producer können Kanal nach Bedarf nutzen

# Kanäle (Beispiel)

```
collector <- newChan
let m = 10
sequence $ do
 d <- [0 .. m - 1]
 return $ forkIO $ do
 let p = sum [d , d + m .. n]
 writeChan collector p
xs <- sequence $ replicate (fromIntegral m
 $ readChan collector
print $ sum xs

ghc -smp --make Foo -o Foo; ./Foo +RTS -N2
http://hackage.haskell.org/trac/ghc/wiki/GhcConcurren
http://pugs.blogs.com/pugs/2006/10/smp_paralleliza.ht
```



# Abstraktion und Kapselung

eingebaute (Bsp: double) und selbst definierte  
private Attribute

Namensbereiche, qualifizierte Namen, import/use

generische Polymorphie (Container weiß nichts über  
Inhaltstyp)

wenn er doch etwas wissen soll, braucht man Schnittstellen