

Programmierparadigmen

SS 2017

Prof. Dr. E. Plödereder

Lernziele:

Verständnis von Konzepten und Paradigmen, die sprachübergreifende Gültigkeit besitzen.

Beispiele kommen sowohl aus den Studierenden bekannten und als auch aus noch unbekannten Sprachen.

Fernziel: Die nächste Programmiersprache wird einfacher begreifbar und vorhersagbarer.

(0.) Administratives:

- **Termine: Aktuelles immer auf der Webseite:**
<http://www.informatik.uni-stuttgart.de/iste/ps/Lehre/>
- Vorlesung ist fast immer zweimal pro Woche; dafür hört sie im Juli früher auf; aktuelle Termin-Informationen im Web bitte beachten!
- Erste Übung ist am Do, 20.04. 14:00 Uhr, als Saalübung, danach in Gruppen (**mit Anmeldepflicht im ILIAS**); Abgabetermin für das erste Übungsblatt ist der 26.4.
- **Prüfungsmodalitäten:**
Mit 50 % der Punkte aus den abgegebenen Übungsabgaben ist Ihnen der Übungsschein sicher. Der Übungsschein ist Voraussetzung für die Klausurteilnahme. Die Note ergibt sich aus der Klausur (90 Min.; der Termin wird vom Prüfungsamt vergeben).
- **Skript:**
Erhältlich im Kopierlädle und auf der Homepage (nicht druckbar)

Inhalte:

1. Einleitendes
2. Ausführungsmodelle im Überblick
3. Semantische Beschreibungsformen
4. Speichermodelle und Unterprogrammsemantiken
5. Bindungskonzepte
6. Typsysteme
7. Herkömmliche Typen
8. Herkömmliche Kontrollstrukturen
9. Ausnahmebehandlung
10. Objekt-orientierte Programmierung
11. Funktionale Programmierung
12. Logische Programmierung
13. Programmierung von Parallelität
14. Enkapsulierungen
15. Programmierung großer Systeme

Literatur:

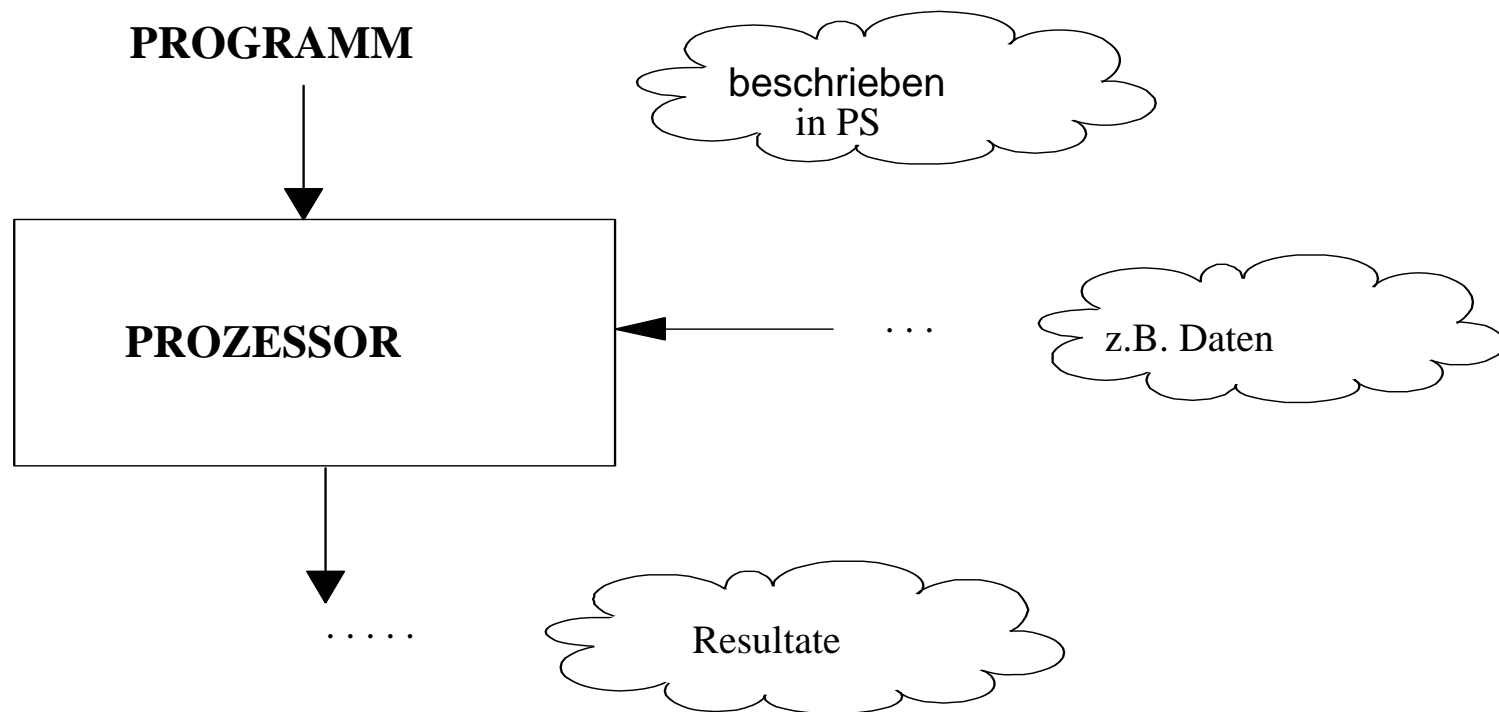
- Robert Sebesta, *Concepts of Programming Languages*, 9. Ed., Addison-Wesley, 2009
- Michael L. Scott, *Programming Language Pragmatics*, 3. Ed., Morgan Kaufmann , 2009
- David A. Watt, *Programming Language Design Concepts*, John Wiley & Sons, 2004
- Referenzmanuale und Lehrbücher zu den zitierten Programmiersprachen

Kapitel 1

Einleitendes

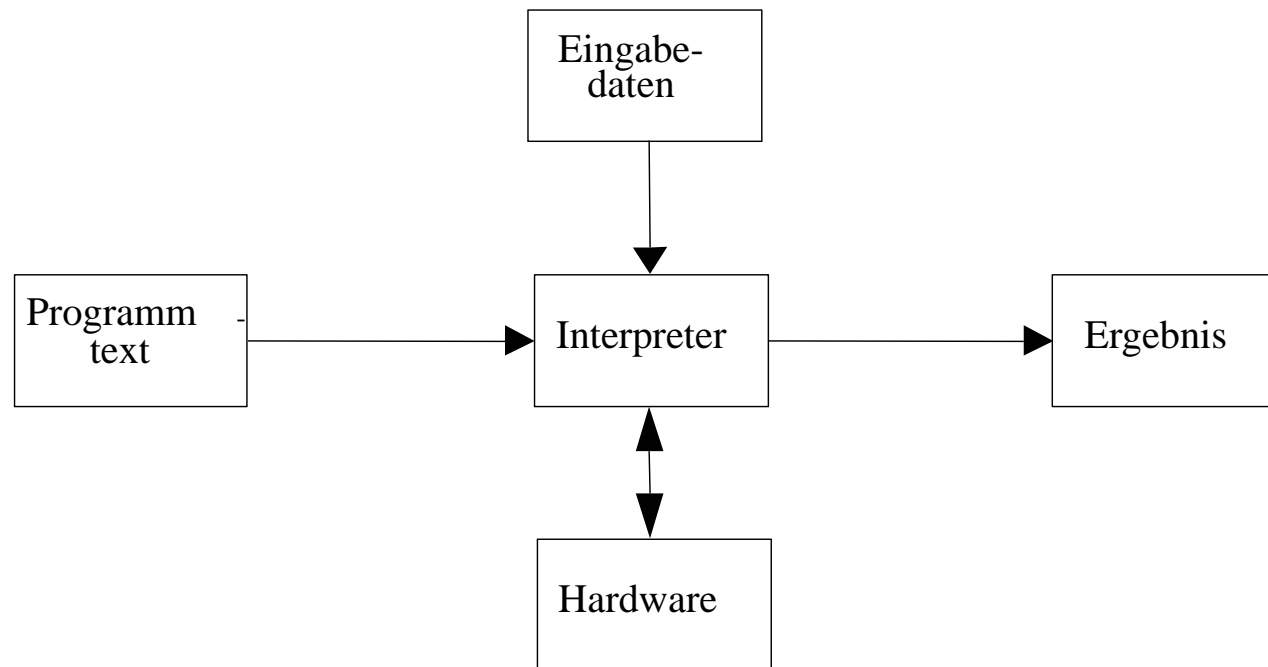
1. Einleitendes

Übliche Definition: *"Programmiersprachen sind Hilfsmittel zur Beschreibung einer Vorschrift für einen Prozessor und bilden eine Schnittstelle zwischen Mensch und Computer."*



1.1 Übersetzung oder Interpretation?

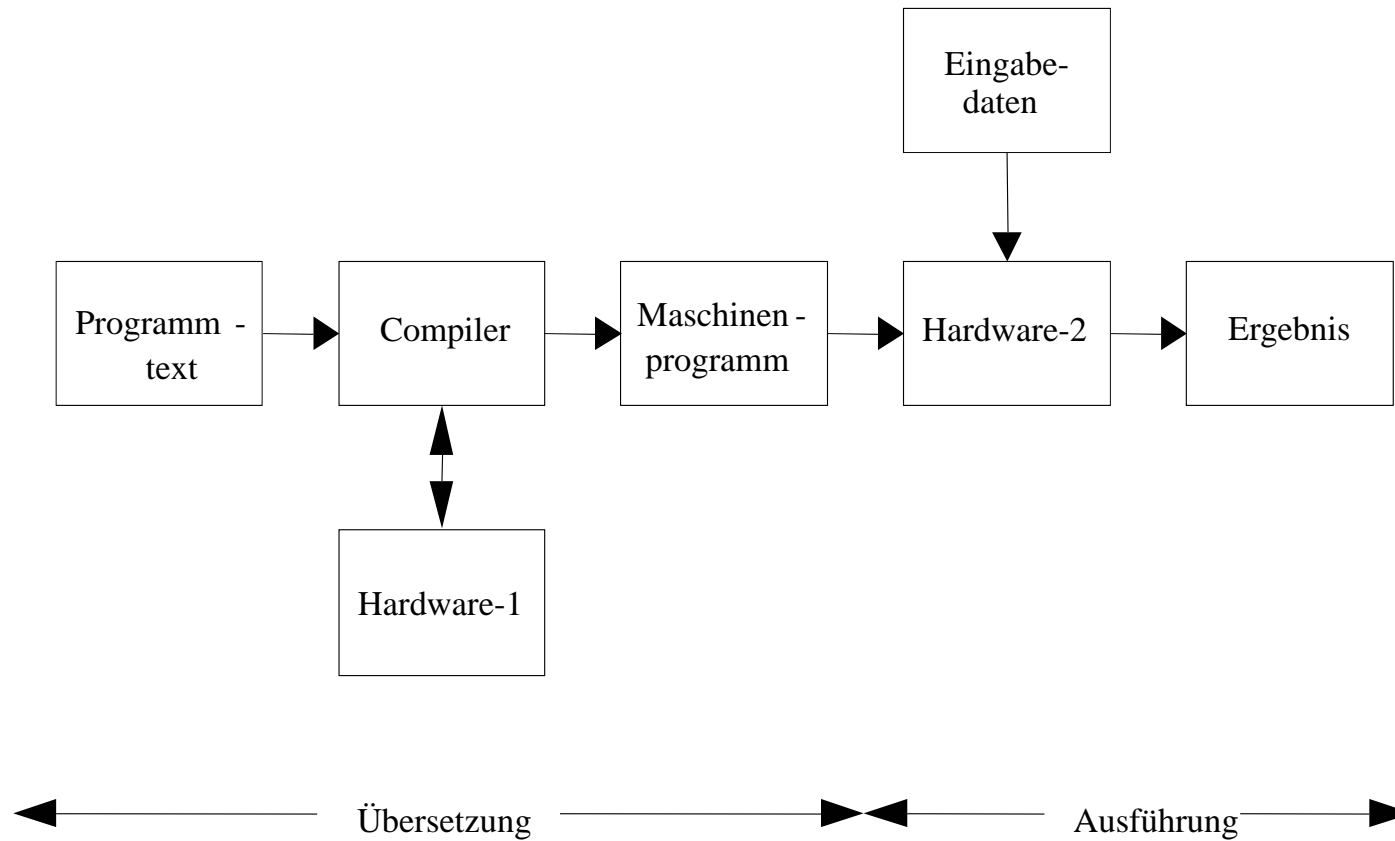
1.1.1 Interpreter



Informationsfluss bei der Ausführung eines Programms durch einen Interpreter

- Keine Erzeugung von Maschinencode
- Das Programm ist nicht eigenständig ausführbar.
- Die Analyse des Programmtextes muss bei jeder Ausführung (eines Programms bzw. eines Schleifenkörpers) wiederholt werden. Es ist keine Codeoptimierung möglich. Folge: zusätzlicher Bedarf an Rechenzeit
- Dynamische Überprüfungen: zusätzliche Flexibilität in der Sprache, aber weniger Sicherheit
- Beispiele für Sprachen, die meist durch Interpreter implementiert werden: Lisp, Prolog, Smalltalk, Scripting-Sprachen

1.1.2 Compiler



Informationsfluss bei Compilierung eines Programms und
anschließender Ausführung des Maschinencodes

- Analyse und Übersetzung des Programms finden nur einmal statt. Der Code kann "in Ruhe" optimiert werden. Das generierte Maschinenprogramm kann anschließend beliebig oft ausgeführt werden.
- Mögliche Erhöhung der Sicherheit (teilweise auf Kosten der Flexibilität): Durch statische Überprüfungen können zusätzliche Klassen von Programmierfehlern gefunden werden.
- Typische Beispiele: C, Fortran, C++, Java (!), Ada ...

1.3 Forderungen an Programmiersprachen

1.3.1 Gewünschte Eigenschaften (der Programme)

- | | |
|-------------------------------------|----------------------------|
| 1. Problem-angepasste Ausdrucksform | (<i>writability</i>) |
| 2. Gute Lesbarkeit | (<i>readability</i>) |
| 3. Leichte Änderbarkeit | (<i>modifyability</i>) |
| 4. Verlässlichkeit | (<i>reliability</i>) |
| 5. Portabilität | (<i>portability</i>) |
| 6. Effizienz | (<i>efficiency</i>) |
| 7. Wartbarkeit (bedingt 2. und 3.) | (<i>maintainability</i>) |

Pragmatische Forderungen:

- 8. (leichte) Lernbarkeit der PS
- 9. Kostengünstige Werkzeuge für PS

1.3.2 Eigenschaften von Programmiersprachen

1. Abstraktion
2. Einfachheit
3. Orthogonalität
4. Erweiterbarkeit, Flexibilität
5. Einfache Implementierung
6. Maschinenunabhängigkeit
7. Sicherheit (statische und dynamische Prüfungen)
8. Standard-/Formale Definition
9. Prinzip der geringsten Verwunderung

...

Vorsicht: Kriterien stehen z. T. im Konflikt zueinander!

Die Eigenschaften von Programmiersprachen haben sicherlich Einfluss auf das Erreichen der “...ilities” und können die Entwickler implizit in diese Richtung lenken,...

...eine Garantie für die gewünschten Eigenschaften der produzierten Software können sie aber nicht geben.

Typische SW-Kostenverteilung

	Erstentwicklung	Gesamtkosten
Entwurf	40 %	12 %
Codierung	20 %	6 %
Test	40 %	12 %
Wartung	-	70 %

⇒ nach Kosten hat Wartbarkeit höchste Priorität!

Ein Wartungsbeispiel ...

(Ein C-Programm, das von einer Firma als Weihnachtsgruß verschickt wurde....)

>original.out

Ein Wartungsbeispiel ...

>original.out

On the first day of Christmas my true love gave to me
a partridge in a per tree.

On the second day of Christmas my true love gave to me
two turtle doves
and a partridge in a per tree.

On the third day of Christmas my true love gave to me
three french hens, two turtle doves
and a partridge in a per tree.

.....

Das Programm ...

```
#include <stdio.h>
main(t,_,a)
char *a;
{
return!0<t?t<3?main(-79,-13,a+main(-87,1-_,main(-86,0,a+1)+a)):
1,t<_?main(t+1,_,a):3,main(-94,-27+t,a)&&t==2?_<13?
main(2,_+1,"%s %d %d\n"):9:16:t<0?t<-72?main(_,t,
"@n'+,#'/*{ }w+/w#cdnr/+,{ }r/*de}+,/*{ *+,/w{ %+,/w#q#n+,/#{ l+,/n{ n+,/+ #n+,/#\
;q#n+,/+k#;*+,/r : 'd* '3,} { w+K w'K: '+} e#';dq# 'l \
q#'+d'K#!/+k#;q# 'r} eKK#} w' r} eKK{ nl]'/#;#q#n') { )#} w') { ) { nl]'/+ #n';d} rw' i;# \
) { nl]!'/n { n#'; r { #w' r nc { nl]'/# { l, + 'K { rw' iK { ; [ { nl]'/w#q#n'wk nw' \
iwk { KK { nl]!'/w { % 'l##w# ' i; : { nl]'/* { q# 'ld;r' } { nlwb!/*de } 'c \
;; { nl' - { } rw]'/+, } ##' * } #nc, ', #nw]'/+kd' +e } +; # 'rdq#w! nr'/ ' ) } + } { rl# ' { n' ' ) # \
} '+ } ## (!!/'")
:t<-50?_==*a?putchar(31[a]):main(-65,_,a+1):main((*a=='/')+t,_,a+1)
:0<t?main(2,2,"%s"): *a=='/'||main(0,main(-61,*a,
"!ek;dc i@bK'(q)-[w]*%n+r3#l,{ }:\nuwloca-O;m .vpbks,fxntdCeghiry"),a+1);
}
```

Die "Korrektur" ...

> original.out | sed -e s/per/pear/

On the first day of Christmas my true love gave to me
a partridge in a pear tree.

On the second day of Christmas my true love gave to me
two turtle doves
and a partridge in a pear tree.

On the third day of Christmas my true love gave to me
three french hens, two turtle doves
and a partridge in a pear tree.

Der "Test" (diff)...

> original.out | sed -e s/per/pear/

On the first day of Christmas my true love gave to me
a partridge in a pear tree.

On the second day of Christmas my true love gave to me
two turtle doves
and a partridge in a pear tree.

On the third day of Christmas my true love gave to me
three french hens, two turtle doves
and a partridge in a pear tree.

.....

"Release" und die Folgen ...

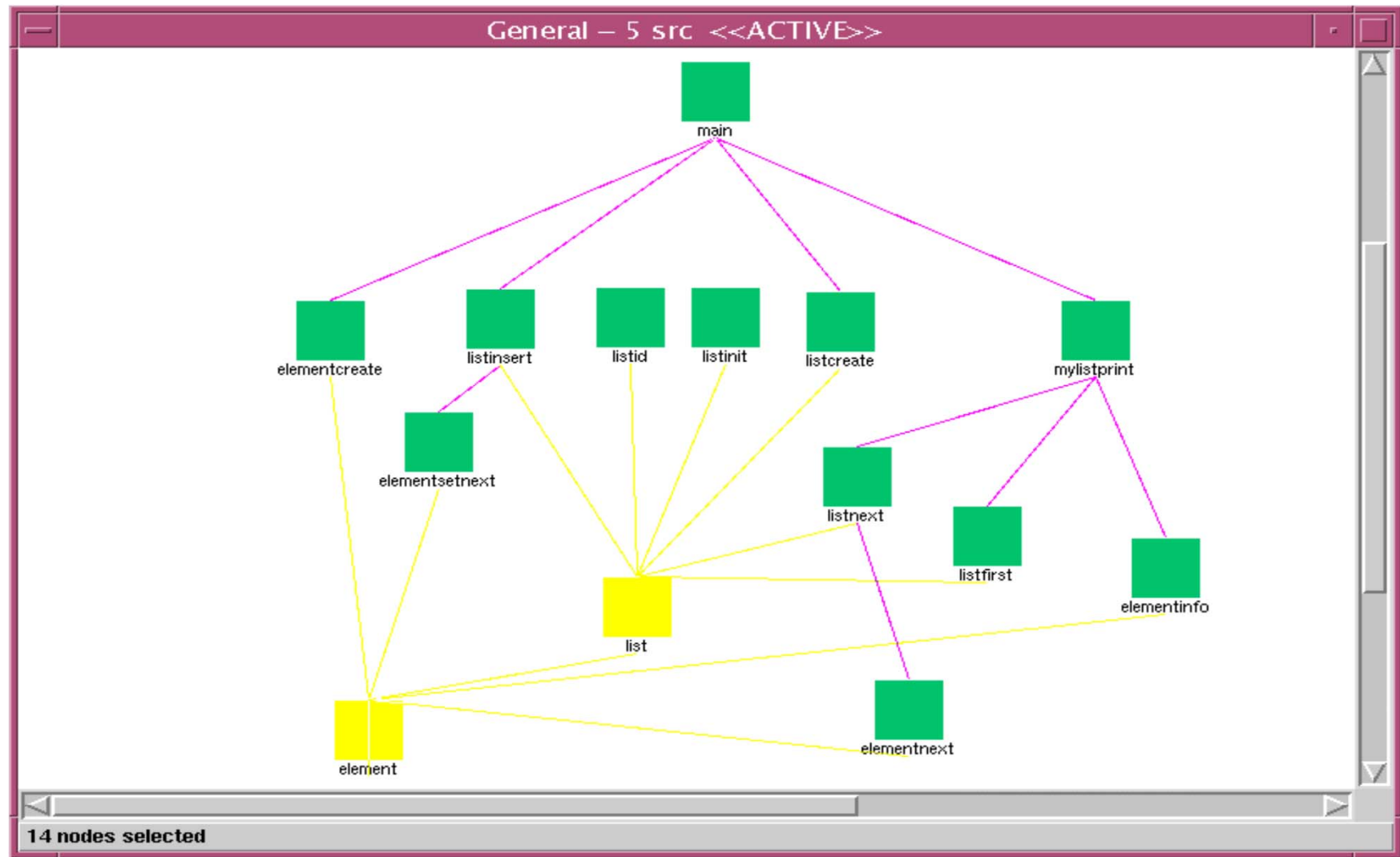
vorher:

On the eleventh day of Christmas my true love gave to me
eleven **pipers** piping, ten lords a-leaping, ...
and a partridge in a **per** tree.

nachher:

On the eleventh day of Christmas my true love gave to me
eleven **pipears** piping, ten lords a-leaping,
nine ladies dancing, eight maids a-milking, seven swans a-swimming,
six geese a-laying, five gold rings;
four calling birds, three french hens, two turtle doves
and a partridge in a **pear** tree.

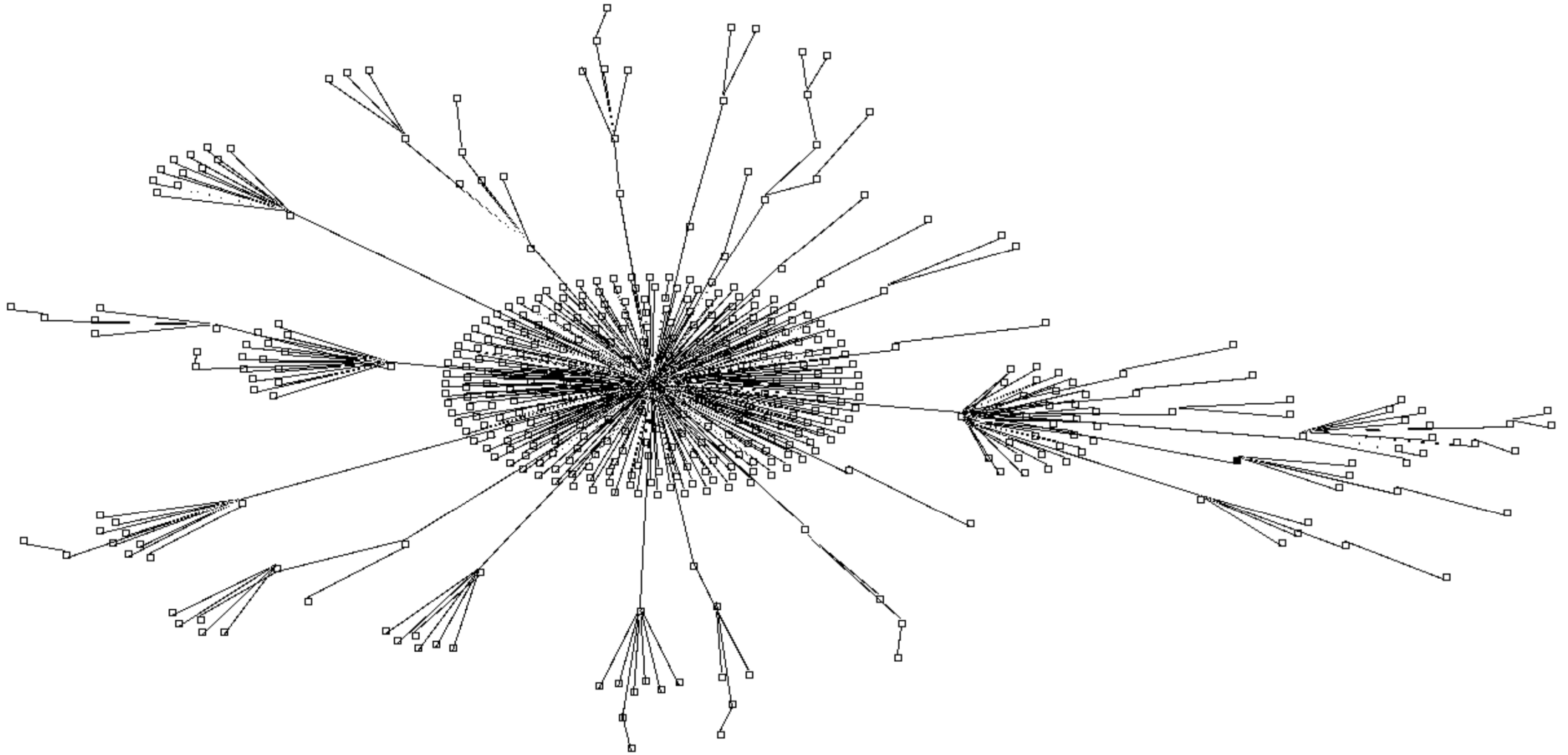
Verständliche SW-Struktur



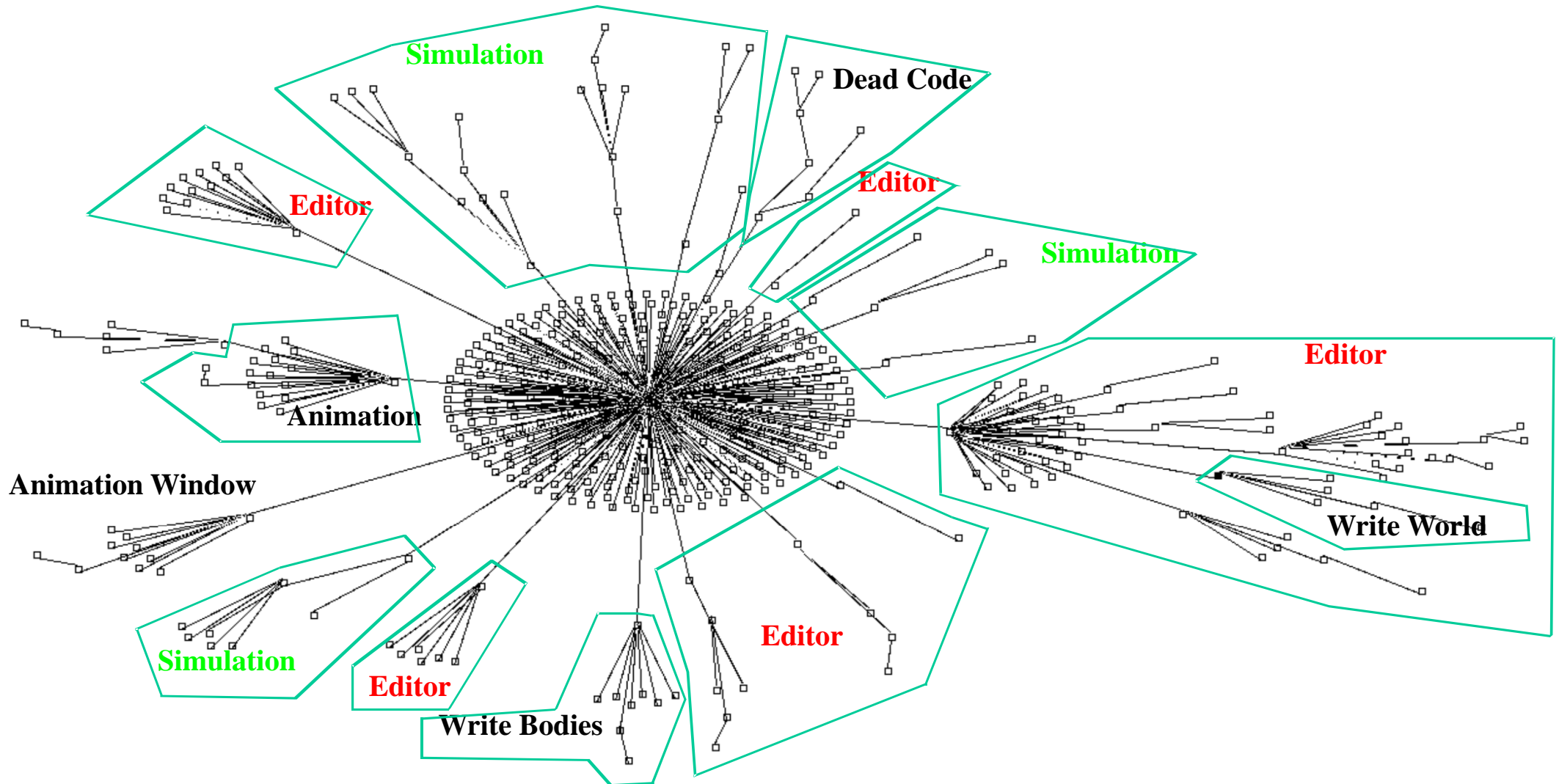
Verständliche SW-Struktur ??



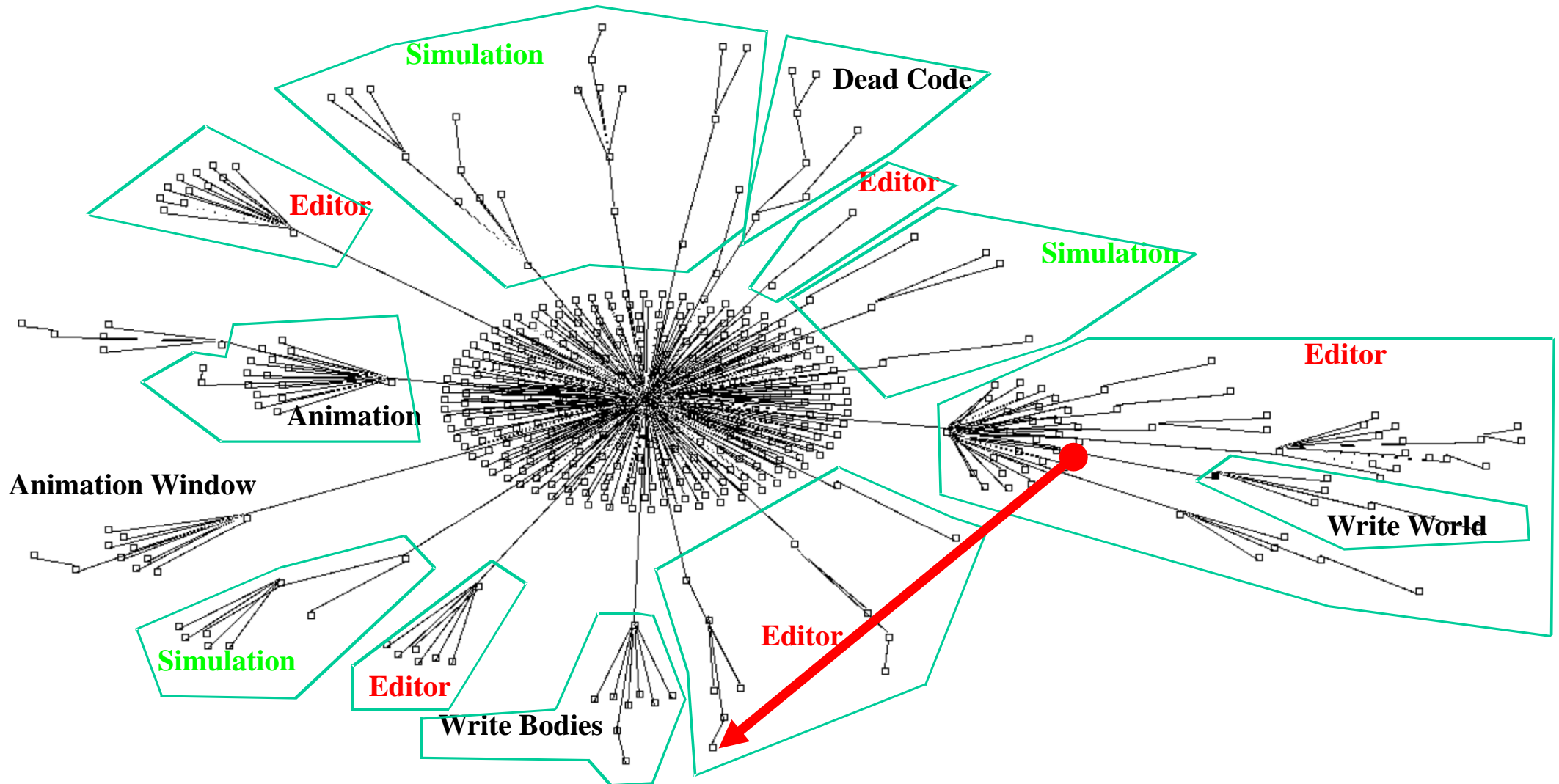
Ordnung im Chaos



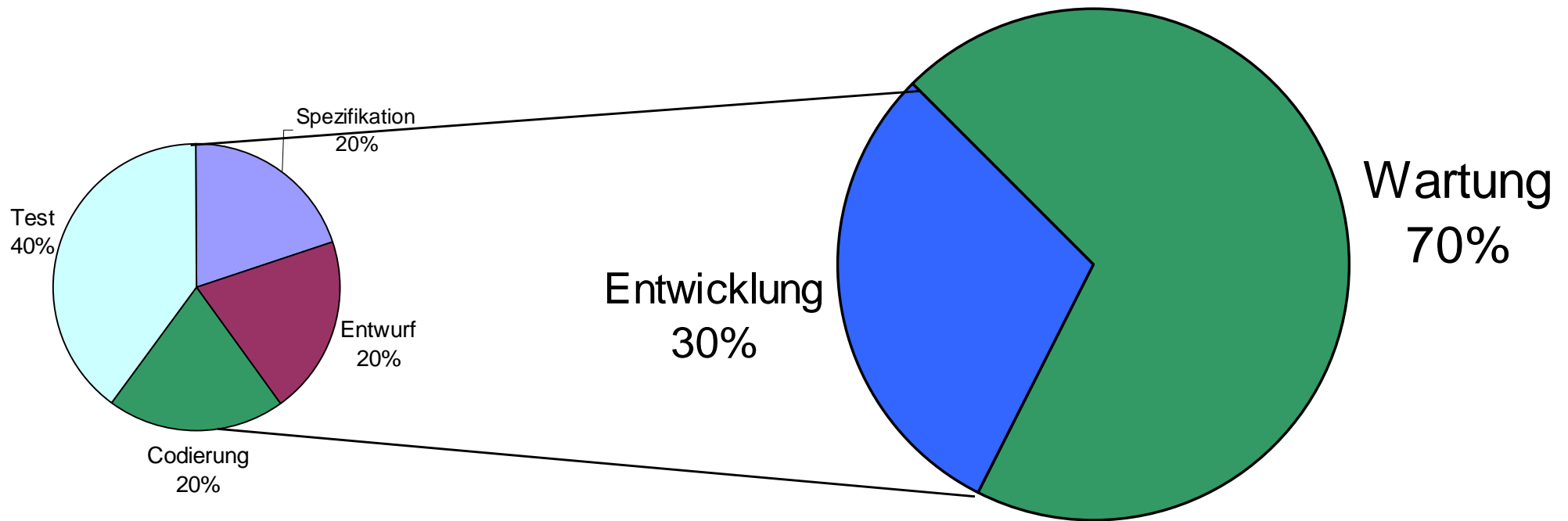
... mit Entwurfskenntnis



Architekturverletzungen

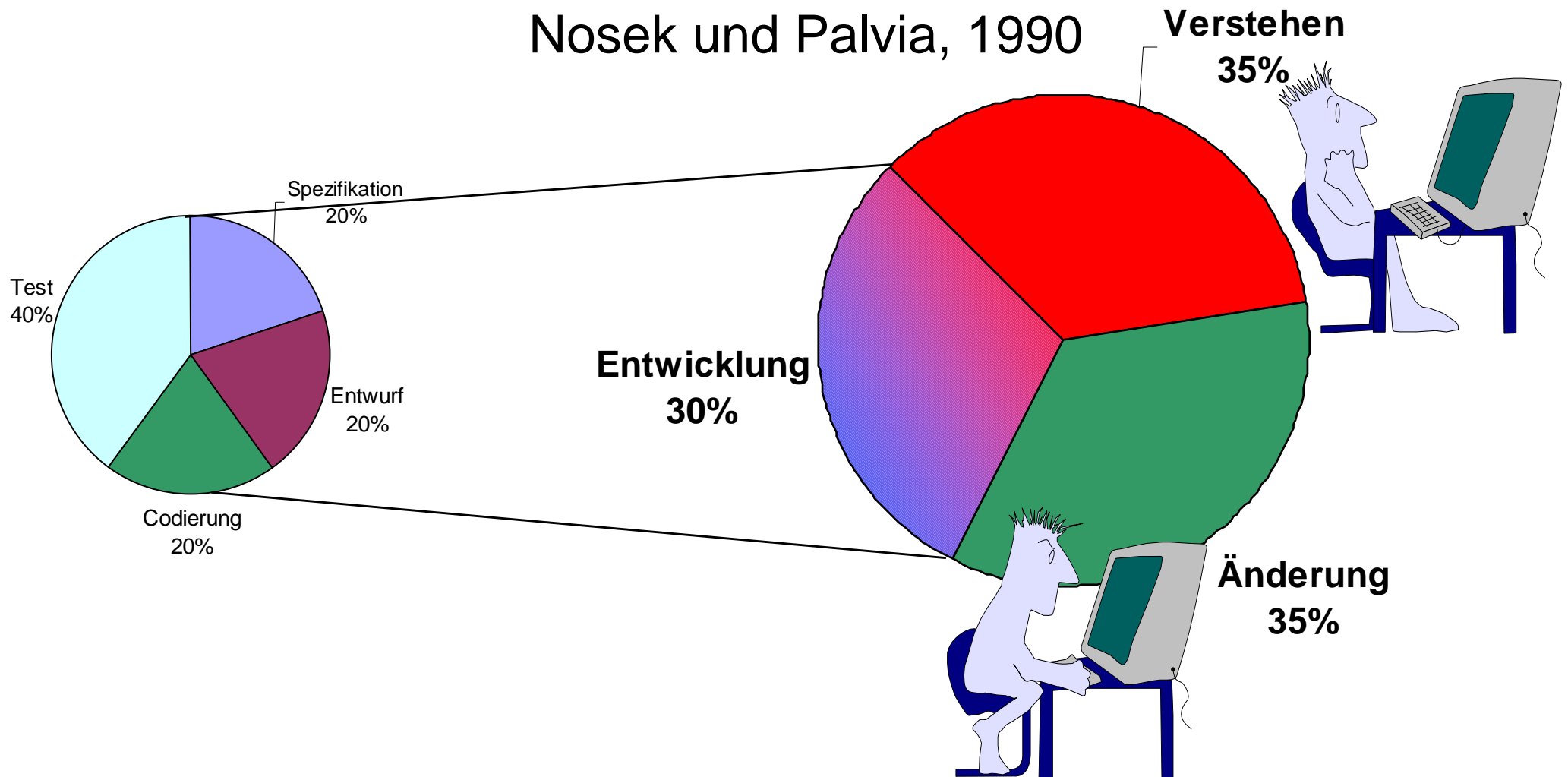


Wartungskosten



Wartungskosten

Nosek und Palvia, 1990



Typische SW-Kostenverteilung

	Erstentwicklung	Gesamtkosten
Entwurf	40%	12%
Codierung	20%	6%
Test	40%	12%
Wartung	—	70% !

=> nach Kosten hat Wartbarkeit die höchste Priorität!

... und als Korollar:

Programmiersprachen sind in erster Linie ein Hilfsmittel für die Kommunikation zwischen Projektmitgliedern und erst in zweiter Linie ein Kommunikationsmittel zwischen Mensch und Rechner.

Sicherheit

- Mariner-Sonde verloren wegen Programmierfehler (~ \$ *,000,000,000 Verlust)
- Bestrahlungsgerät „Therac 25“; Überdosis durch Programmierfehler (Verlust von Menschenleben)
- ...
- Software in der Kernreaktor-Überwachung . . . ?

⇒ **Sicherheit (Korrektheit und Fehlertoleranz) hat ggf. höchste Priorität**

1.3.3 Der Einfluss von Programmiersprachen

Es folgen einige Statistiken, die den Einfluss von Programmiersprachen auf die entwickelte Software belegen.

Nur wenige derartige Statistiken sind (frei) verfügbar, da hier Geschäftsgeheimnisse betroffen sind.

Die Darstellung ist stark auf Ada ausgerichtet, weil man sich im Rahmen der Entwicklung dieser Sprache bemüht hat, statistisch gültige Daten zu sammeln.

Error Rates (pro kloc)

	Ada	C	C++	norm
Information Systems (kom.)	4,0	7,0	5,1	7,0
Information Systems (mil.)	3,0	6,0	4,0	6,0
Commercial Products	2,8	5,0	3,0	4,0
Telecommunication (kom.)	1,6	2,0	1,7	3,1
Telecommunication (mil.)	1,0	1,5	1,2	2,5
Weapon Systems (ground)	0,5	0,8	0,7	1,0
Weapon Systems (air/space)	0,3	0,8	0,6	1,0

Source: Reifer 1996, 190 projects 1993-96, norm = avg. of 1500 projects 1989-96

SLOC per Function Point

	Min	Av.	Max
C	60	128	170
Ada83	60	71	80
C++	30	53	125
Ada95	28	49	110

(determined by SPR from a DB of about 5000 Projects; annually updated)

Source: Capers Jones, SPR, 1995 (excerpt from a list with 450 languages)

Mean Time (in weeks) to "Major Repair Incident"

	Ada	C	C++	Norm
Commercial Products	1,0	0,4	1,0	0,5
Information Systems (mil.)	0,8	0,5	k.A.	0,5
Information Systems (kom.)	1,0	0,5	0,6	0,5
Telecommunication (kom.)	3,0	1,0	2,0	1,8
Telecommunication (mil.)	4,0	2,0	3,0	2,0
Weapon Systems (ground)	6,0	3,0	k.A.	2,5
Weapon Systems (air/space)	8,0	3,0	k.A.	2,5

Source: Reifer 1996, 190 projects 1993-96, norm = avg. of 1500 projects 1989-96

Other statistics

- Reifer 87: development costs in Ada are **sublinearly (!!)** (exponent: 0,95) dependent on project size.
- ESA DB 96: Highest correlation of productivity is with the characteristic "Company" and then "Programming Language".

Ziegler's Study (C vs. Ada)

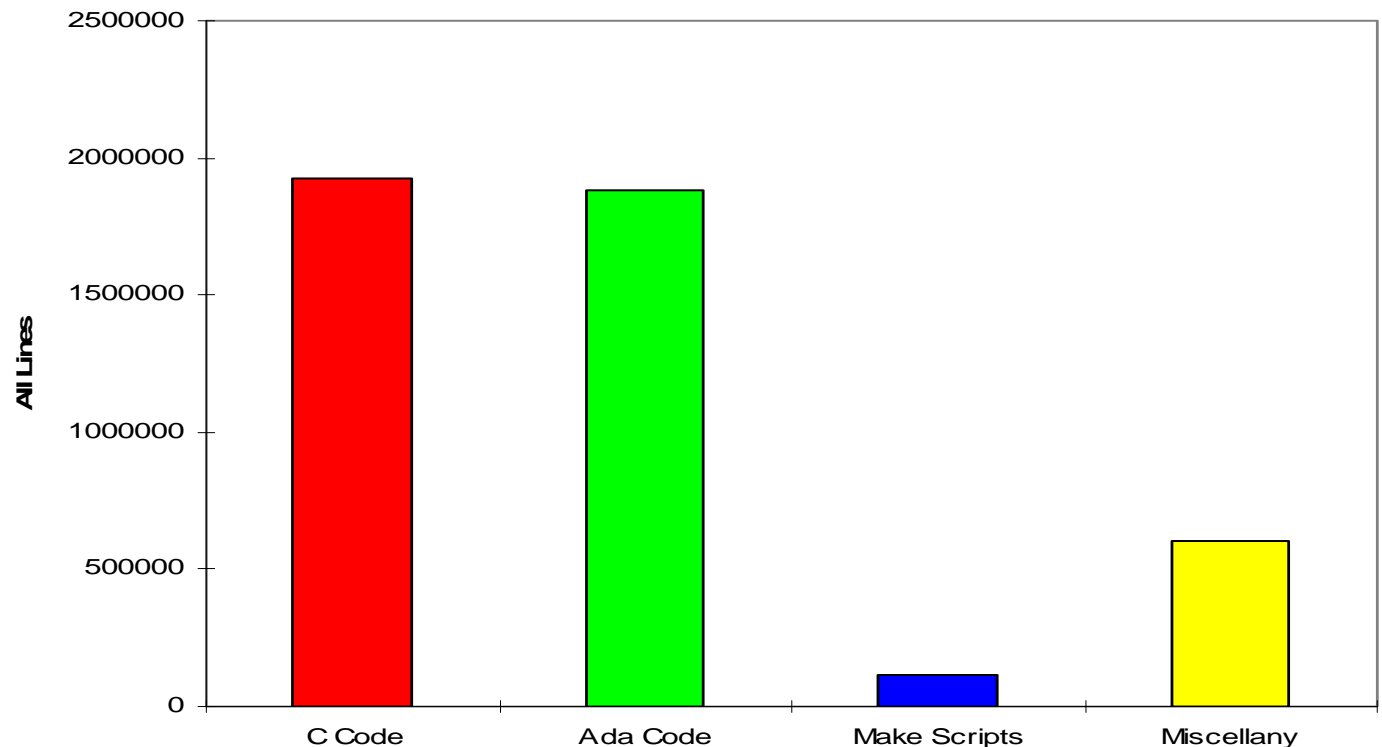
http://www.adaic.com/whyada/ada-vs-c/cada_art.html

- 1995 study on the VADS compiler

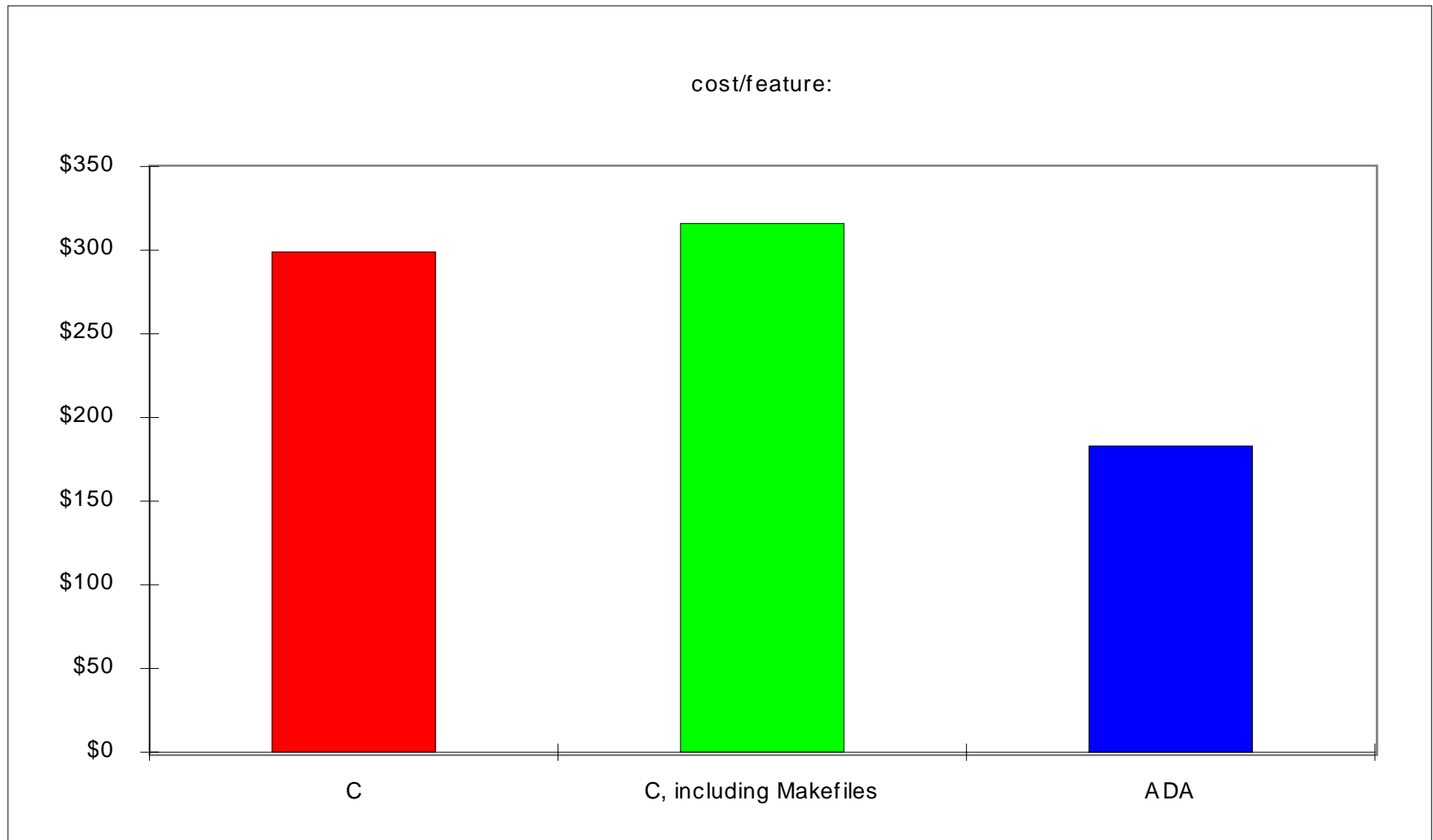
- 60 engineers, from 1984 ..1994 with MS degrees in computer science
- All knew C at hire. All programmed in both C and Ada.

- VADS

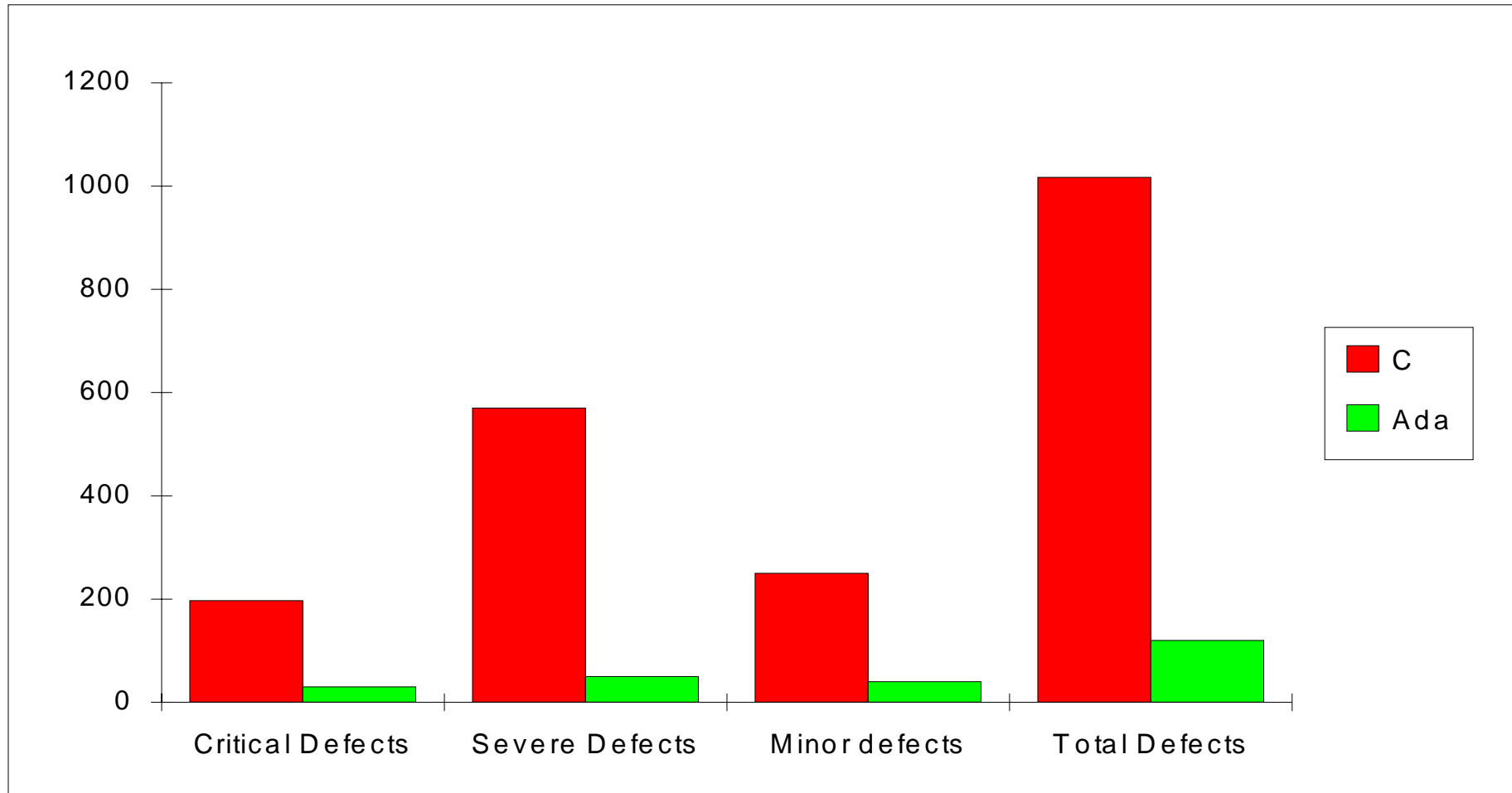
- About 4.5 million lines of code,
- 22000 files



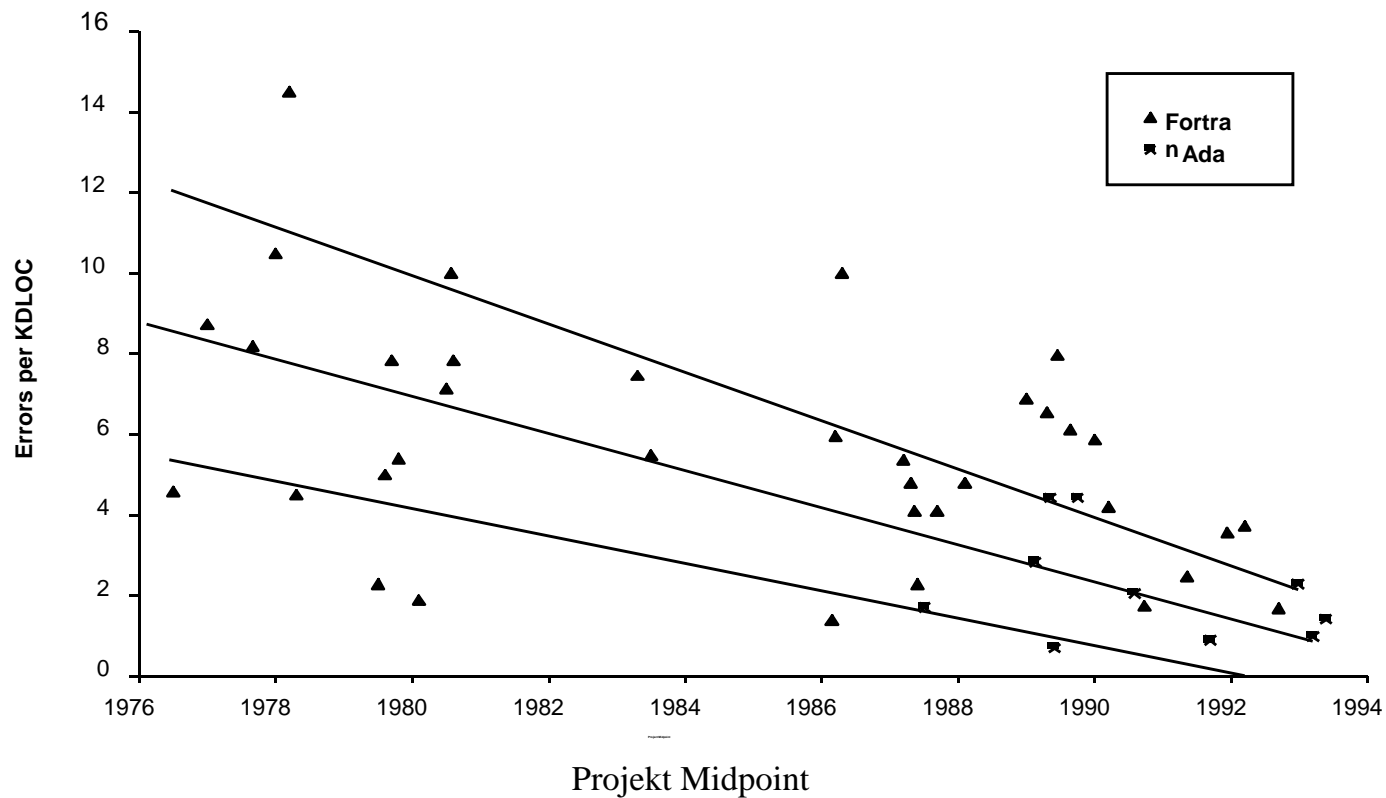
Cost per Product Feature



Post-Delivery Defects



Historische Entwicklung



Quelle: NASA-SEL Study; US National Research Council Report, Dec 1996

Kosten (\$) pro SLOC

	Ada	C	C++	Norm
Information Systems (kom.)	k.A.	25	25	30
Information Systems (mil.)	30	25	25	35
Commercial Products	35	25	30	40
Telecommunication (kom.)	55	40	45	50
Telecommunication (mil.)	60	50	50	75
Weapon Systems (ground)	80	65	50	75
Weapon Systems (air/space)	150	175	k.A.	200

Quelle: Reifer 1996, 190 Projekte 1993-96, Norm aus 1500 Projekten 1989-96

Entwicklungskosten

Kosten pro "Zeile ausgelieferter Quellcode" (SLOC): \$10 - \$200

Beeinflussende Faktoren:

- Komplexität der Anwendung und Qualitätskriterien
- Effizienz des Entwicklungsprozesses und der Ingenieure
- Verwendete Technologien

Für ein "embedded system" von 300 kSLOC: ca. 10-25 Mio. \$

Optimale Entwicklungsdauer: 2 - 3 Jahre

Wartungskosten: 30-100 Mio \$

.. und in der Wartung ...

Es ist auf absehbare Zeit jenseits des "State of the Art", fehlerfreie Software zu produzieren.

Statistiken besagen, dass die Hälfte der Fehler im ersten Einsatzjahr von den Kunden moniert wird.

Unsere 300-kSLOC-Telefonsoftware wird rund **600(!)** Fehler enthalten.

Ein kommerzielles Textverarbeitungssystem mit 5 MSLOC lässt rund **20.000 (!!)** Fehler erwarten.

Konsequenzen

Die Garantieleistungen bei Softwareprodukten werden auf Medienersatz (mit identisch fehlerhafter Software) beschränkt.

Softwareerstellung unter Vertrag mit Gewährleistungspflicht ist extrem riskant (und hat schon viele Firmen in den Konkurs getrieben).



Eine handelsübliche Garantie...

X garantiert, dass die Datenträger, auf denen dieses Produkt gespeichert ist, bei normalem Gebrauch keine Material- und Herstellungsfehler aufweisen. Falls während der Garantiezeit irgendwelche Datenträger Mängel aufweisen sollten, soll der Kunde die Datenträger an X zurückschicken, um einen Ersatz zu bekommen.

X und seine Zulieferfirmen garantieren die Leistung oder Ergebnisse, die Sie durch Verwendung der Software und der Dokumentation erzielen, nicht und kann sie nicht garantieren. Das oben Aufgeführte nennt die alleinigen und ausschließlichen Abhilfen, falls X gegen die Garantie verstößt. Außer der vorher beschriebenen beschränkten Garantie gewähren weder X noch eine seiner Zulieferfirmen irgendwelche ausdrücklichen oder stillschweigenden Garantien bezüglich handelsüblicher Qualität, Nicht-Verstoßes gegen Rechte Dritter oder Eignung für bestimmte Zwecke.

X und seine Zulieferfirmen sind auf keinen Fall Ihnen gegenüber für irgendwelche Folge- oder Zufallsschäden verantwortlich. Dazu gehören Profit- oder Einsparungsverluste oder Ansprüche durch beteiligte Personen, selbst wenn ein X-Repräsentant oder eine Zulieferfirma von X über die Möglichkeit solcher Schäden unterrichtet wurde.

Realität 2013: Beispiel Toyota (Engine Control Firmware)

- 11.000 globale Variablen zur Beschreibung des Zustands und der Kontrolle
 - Zyklomatische Komplexität von >50 in 67 Funktionen und >100 in der vermutlich die Unfälle verursachenden Funktion
 - 80.000 Verletzungen des MISRA-C:98-Standards der Automobilindustrie (nur 11 von 93 geforderten Prüfungen wurden durchgeführt)
- ⇒ Vermutlich die teuerste Software der Welt (durch die Schadenersatzforderungen)

Quelle: Expertengutachten der Barr Group vor Gericht; Michael Dunn bei EE!live

Kapitel 2

Ausführungsmodelle im Überblick

2 Ausführungsmodelle für Programmiersprachen

2.1 Das von-Neumann-Modell

Merkmale eines Computers nach dem von-Neumann-Modell:

- Wesentliche Komponenten: Speicher, Ablaufsteuerung, (arithmetisch/logische) Verarbeitungswerke
- Daten und Befehle werden in einem gemeinsamen Speicher abgelegt.
- Ein Programm ist eine Folge von Befehlen, welche durch die Ablaufsteuerung sequentiell und schrittweise ausgeführt werden. Es gibt Sprungbefehle, um den Kontrollfluß umzulenken. Ein Befehlszähler enthält die Adresse des jeweils auszuführenden Befehls.
- Die Inhalte der Speicherzellen können mit neuen Werten überschrieben werden.

Typische Beispiele:

m. E. Ihr PC (wenn man Multi-Core kurz ignoriert)

Die ***von-Neumann-Sprachen*** nehmen auf einer abstrakten Ebene auf dieses Modell Bezug:

- benannte Variablen sind symbolische Bezeichnungen für Speicherzellen
- der Kontrollfluss (bzw. der Inhalt des Befehlszählers) kann durch Kontrollstrukturen (z.B. WHILE, IF) beeinflusst werden (Prinzip der Ablaufsteuerung durch Kontrollfluss)

Programmierung bedeutet in diesem Modell die Festlegung der

- Speicherbelegung (Entwicklung von Datenstrukturen, Variablendeklarationen)
- Speichertransformationen (Zuweisungen) und deren Abfolgen (Unterprogramme, Threads, Kontrollstrukturen)

Die bekanntere Bezeichnung für dieses Modell ist „prozedurale oder imperative Programmierung“.

Beispiele für Sprachen dieser Klasse: Fortran, Pascal, C, Ada83,...

2.2 Simulationsmodell

Idee: Programmierung = Simulation von realen Systemen
(zum ersten Mal in Simula67 realisiert)

Aufgaben des Programmierers:

- Beschreibung der Bestandteile des realen Systems



- Beschreibung des Zusammenwirkens der Bestandteile



Die Ausführung besteht aus dem Austausch von Nachrichten zwischen Objekten und deren entsprechender Reaktion

Beispiele: Simula67, Smalltalk

Viele objektorientierte Programmiersprachen haben dieses Modell aufgegriffen und imperativ-prozedural ausgedrückt (kein Nachrichtenaustausch, sondern Methodenaufrufe; keine a priori unabhängig agierenden Objekte, sondern Hauptprogramme und Threads als übergreifende Kontrollstrukturen).

Das Modell ist bekannter als "objektorientiertes Modell", auch wenn seine Ursprünge in der Simulation lagen.

Beispiele: Eiffel, Java, C++, Ada95

(Weiteres im Kapitel "Objektorientierte Programmierung")

2.3 Funktionales Modell

- Ausführungsmodell: die Auswertung verschachtelter und im Allgemeinen rekursiver Funktionsaufrufe
- Alle programmiersprachlichen Konstrukte und alle vom Programmierer eingeführten Unterprogramme sind Funktionen, d.h. sie haben Parameter und liefern Werte zurück.
- Funktionen sind berechenbare Werte (-> Funktionale) und ihrerseits auswertbar. Einige Funktionale werden von der Programmiersprache vordefiniert.
- Programmierung bedeutet die Entwicklung von Funktionen und Funktionalen für die beabsichtigte Transformation von Eingabe- in Ausgabewerte.

Beispiele: LISP, ML, Haskell, F#

(Weiteres im Kapitel 7: Funktionale Programmierung)

2.4 Datenfluss-Modell

- Idee: Jede Operation kann ausgeführt werden, sobald ihre Operanden ausgewertet sind (Prinzip der Ablaufsteuerung durch Datenfluss).
- Ausführungsmodell: Darstellung der Datenabhängigkeiten durch einen gerichteten Graphen (Abhängigkeitsgraph).

- Knoten des Graphen: die Operationen des Programmes
- Kanten des Graphen: die Datenabhängigkeiten

Die Programmausführung besteht im Transport von Datenwerten entlang der Kanten. Sobald alle Operanden einer Operation vorliegen, wird diese ausgeführt und sendet ihr Ergebnis ab.

- Verzicht auf Kontrollfluss; "Single Assignment"-Forderung
- eignet sich gut zur Ausnutzung von Parallelität

Beispiele: VAL, ID, SISAL, Lustre

Das Prinzip der Auswertung gemäß Datenfluss

$h_1 = 4 * I$
 $h_2 = A[h_1]$
 $h_3 = 4 * I$
 $h_4 = B[h_3]$
 $h_5 = h_2 * h_4$
 $h_6 = \text{PROD} + h_5$
 $\text{PROD2} = h_6$
 $h_7 = I + 1$
 $I2 = h_7$

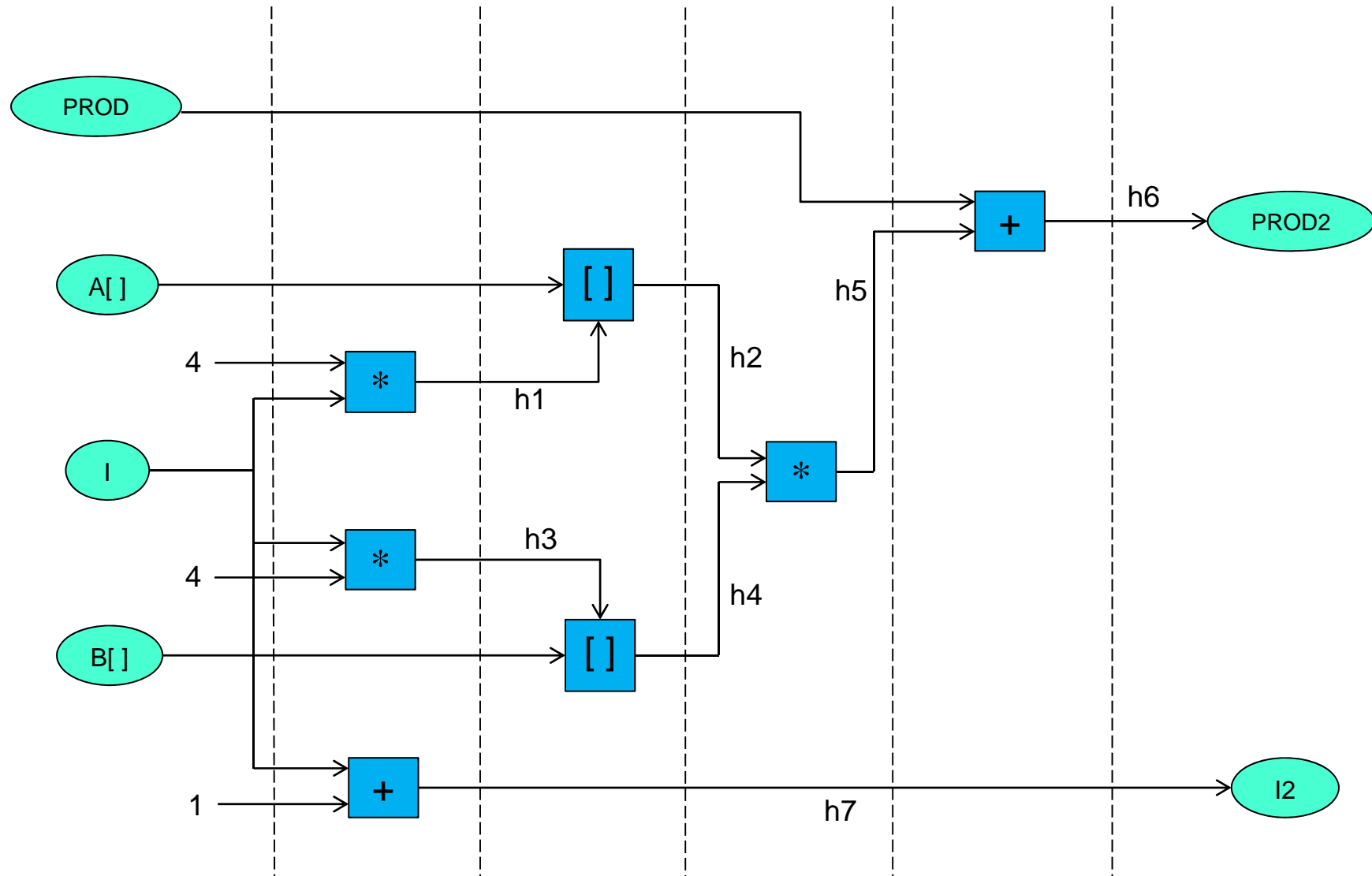
Das Beispiel ist einem Szenario aus dem "Drachenbuch" (Aho, Ullman, Sethi) über Compilerbau nachgebildet. Moderne Compiler optimieren nach Datenfluss-Regeln auch für "prozedurale Sprachen".

$h_1 = 4 * I$
 $h_2 = A[h_1]$
 $h_5 = h_2 * h_4$
 $h_6 = \text{PROD} + h_5$
 $\text{PROD2} = h_6$

$h_3 = 4 * I$
 $h_4 = B[h_3]$

$h_7 = I1 + 1$
 $I2 = h_7$

Datenflussgraph des Beispielprogramms



Ursprünglicher Quelltext des Beispiels

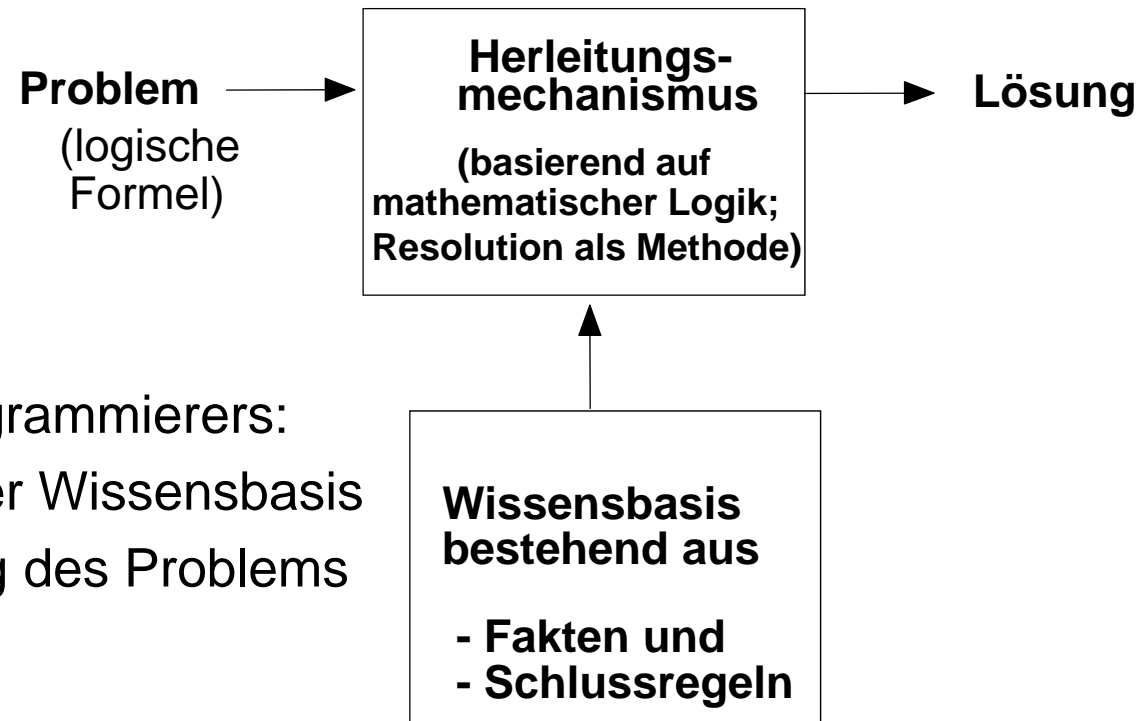
```
int I = 0;
double PROD = 0.0;
while( I < N )
{
    PROD += A[ I ] * B[ I ];
    I++;
}

long_float PROD := 0.0;
for I in 1.. N loop
    PROD := PROD +A[ I ] * B[ I ];
end loop;
```

Berechnung des Skalarprodukts zweier N-stelliger Vektoren A und B
(wobei der Compiler das Programm in eine "single-assignment-form"
gebracht hat).

2.5 Logik-Modell

Idee: Programmierung = Logische Herleitung einer Lösung aus einer Wissensbasis:



Aufgaben des Programmierers:

- Erstellung der Wissensbasis
- Formulierung des Problems

Beispiele: Prolog, Concurrent Prolog

Ein kleines Beispiel in Prolog

Wissensbasis:

```
gcd(U, 0, U) .
```

```
gcd(U, V, W) :- not zero(V), gcd(V, U mod V, W) .
```

Programm: (interaktive Anfrage)

```
?- gcd (15, 10, X)
```

Ausgabe:

```
X = 5
```

Programm:

```
?- gcd (15, X, 5)
```

Ausgabe:

```
X = 5
```

```
X = 10
```

```
X = 20
```

```
...
```

Kapitel 3

Beschreibung von Programmiersprachen

3. Beschreibung von Programmiersprachen

Bei einer Programmiersprache muss festgelegt werden,

- welche Zeichenfolgen korrekte Programme darstellen und
- welche Auswirkungen einzelne Konstrukte der Sprache bzw. ganze Programme haben sollen.

Man unterscheidet zwischen **Syntax** und **Semantik**.

Syntax:

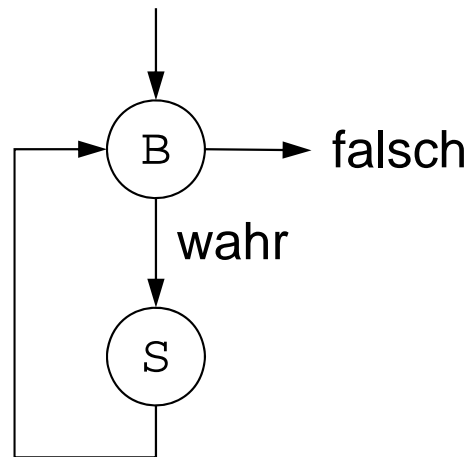
- Beschreibt alle Eigenschaften einer Sprache, die sich durch kontextfreie Grammatiken ausdrücken lassen
- Beispiele:
 - Aufbau von Bezeichnern in Modula-2
 - Struktur der WHILE-Schleife in PASCAL

`while ... do ...`
oder in Ada
`while ... loop ... end loop`

Semantik

- legt die Bedeutung einzelner Sprachbestandteile fest

Beispiel: Die Semantik der Schleife **WHILE** B **DO** S **END** soll durch folgendes Ablaufdiagramm beschrieben werden:



- Man unterscheidet
 - statische Semantik (Beispiel: die Auswirkungen von Vereinbarungen; kann (muss aber nicht) bereits zur Übersetzungszeit ausgewertet werden)
 - dynamische Semantik (beschreibt den funktionalen Zusammenhang zwischen Ein- und Ausgabewerten)

Abgrenzung Syntax - Semantik

„der Semantiker“:

„der Compilerbauer“:

kontextfreie Syntax

Syntax

durch kontextfreie Grammatik beschreibbare Eigenschaften

kontextsensitive Syntax

statische Semantik

Eigenschaften, die anhand des Programmtextes überprüft
werden können

(z. B. Typkorrektheit)

Semantik

dynamische Semantik

Eigenschaften, die die Ausführung von Programmen betreffen

(z. B. Ergebnis der Programmausführung)

3.1 Methoden für die Syntaxbeschreibung

3.1.1 Backus-Naur-Form (BNF)

Die BNF ist eine Notation für kontextfreie Grammatiken:

- Nichtterminalsymbole (NTS) werden in spitze Klammern $\langle \rangle$ eingeschlossen, um sie von Terminalsymbolen (TS) zu unterscheiden.
- Die Ableitbarkeitsrelation wird durch das Symbol $::=$ dargestellt.
- Alternative Regeln für ein NTS werden durch das Symbol $|$ getrennt.
- Die leere Zeichenkette wird durch ε dargestellt.
- Das Ende einer Produktion wird durch einen Punkt markiert,

Beispiel: (IF-Anweisung aus Modula-2)

```
<IfStatement> ::= IF <expression> THEN  
    <StatementSequence> <ElsifPart> <ElsePart>.  
<ElsifPart> ::= <ElsifPart> <ElsifPart> | e |  
    ELSIF <expression> THEN <StatementSequence>.  
<ElsePart> ::= ELSE <StatementSequence> | e.
```

Problem: die Notation ist unbequem

- Iteration muss durch Rekursion ausgedrückt werden.
- Optionale Regeln müssen durch „Alternative mit ε “ beschrieben werden.

3.1.2 Erweiterte BNF (EBNF) nach Wirth

EBNF ist eine „bequemere“ Notation als reine BNF.

Unterschiede gegenüber der BNF:

- Die Iteration wird durch geschweifte Klammern { } bezeichnet.
- Optionale Teile einer Regel werden durch eckige Klammern [] bezeichnet.
- Vorrangregeln können durch Klammerung mit () ausgedrückt werden.
- NTS werden nicht <> eingeschlossen, sondern TS werden durch Großschreibung bzw. Einschluss in „“ (bei Sonderzeichen) gekennzeichnet.

Beispiel: (IF-Anweisung aus Modula-2)

```
<IfStatement> ::= IF <expression> THEN <StatementSequence>  
    { ELSIF <expression> THEN <StatementSequence> }  
    [ ELSE <StatementSequence> ].
```

Zusätzlich kann eine Iteration durch $\{ \dots \}_i^k$ mindestens i- und höchstens k-mal eingeschränkt werden.

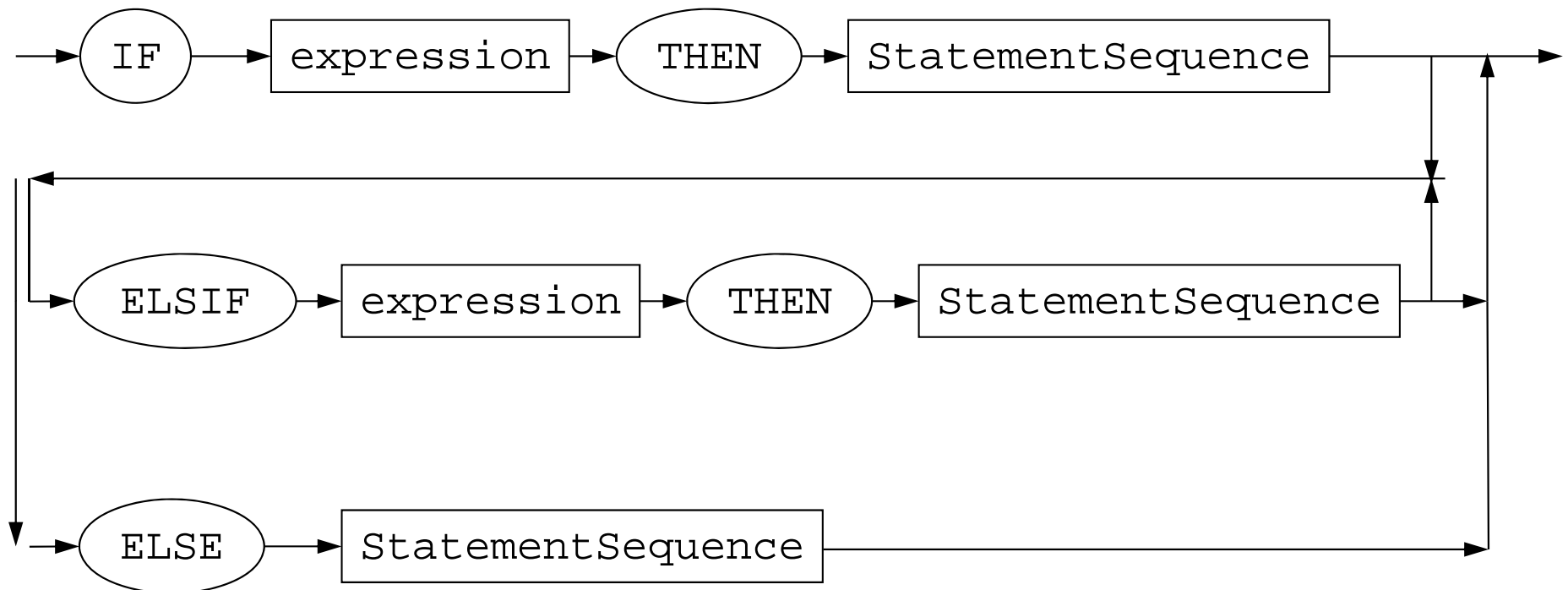
(E)BNF stellt eine Sprache zur Syntax-Beschreibung von Programmiersprachen dar und bilden die Grundlage für die Eingabe-Sprachen vieler Generatoren.

Es gibt diverse Variationen der EBNF Syntax in Literatur und Praxis.

3.1.3 Syntaxdiagramme

- Graphische Darstellung einzelner Produktionen, besonders anschaulich!
- TS werden in Ellipsen eingeschlossen.
- NTS werden in Rechtecke eingeschlossen.
- Alternativen, Konkatination, Iteration und Option werden durch gerichtete Kanten dargestellt.

Beispiel: IfStatement



3.2 Semantikbeschreibung von Programmiersprachen

Motivation:

- Programmierer muss die Wirkung von Bestandteilen eines Programms kennen.
- Implementierer muss wissen, wie die beabsichtigte Wirkung von Bestandteilen zu erreichen ist.
- Korrektheit eines Programms kann nur mittels der Kenntnis der Semantik eines Programms hergeleitet werden.
- Formale semantische Beschreibungen sind unerlässlich für die „automatische“ Implementierung von Programmiersprachen („Generatoren“)

Methoden:

- Syntax-orientierte Semantikbeschreibung durch attributierte Grammatiken
- Operationale Methoden
- Denotationale Methoden
- Axiomatische Methoden
- . . .
- (technische Prosatexte – „Referenzmanual“)
- (Referenzimplementierung)

3.2.1 Attributierte Grammatiken

Idee:

- Beschreibung der Semantik eines Programms setzt sich aus den Beschreibungen der Semantik seiner syntaktischen Bestandteile zusammen.
- Den Terminal- und Nichtterminalsymbolen einer Grammatik werden Werte („Attribute“) zugeordnet. (Tritt ein Symbol in einer Produktion mehrfach auf, so werden die verschiedenen Exemplare nummeriert.)
- Den syntaktischen Produktionsregeln werden Funktionen zur Berechnung der Attribute der in der Produktion auftretenden Symbole zugeordnet.
- Zusätzlich können Bedingungen oder Auswertungsvorschriften formuliert werden.
- Berechnung der Attribute erfolgt automatisch (während oder nach der Syntaxanalyse)
- Literatur: Waite84, ASU88, Kastens90

Eigenschaft:

- Im Wesentlichen auf statische Semantik beschränkt

Beispiel: Arithmetische Ausdrücke in einer Programmiersprache

Produktionen:

- P1: $\langle \text{expression}_1 \rangle ::= \langle \text{expression}_2 \rangle + \langle \text{term} \rangle$
- P2: $\langle \text{expression} \rangle ::= \langle \text{term} \rangle$
- P3: $\langle \text{term}_1 \rangle ::= \langle \text{term}_2 \rangle * \langle \text{factor} \rangle$
- P4: $\langle \text{term} \rangle ::= \langle \text{factor} \rangle$
- P5: $\langle \text{factor} \rangle ::= (\langle \text{expression} \rangle)$
- P6: $\langle \text{factor} \rangle ::= \langle \text{int_constant} \rangle$
- P7: $\langle \text{factor} \rangle ::= \langle \text{real_constant} \rangle$

Erforderliche Attribute:

- TYP : Typ eines Ausdrucks, Terms, Faktors, . . .
- WERT : Wert eines Ausdrucks, Terms, Faktors, . . .

Auswertungsvorschriften:

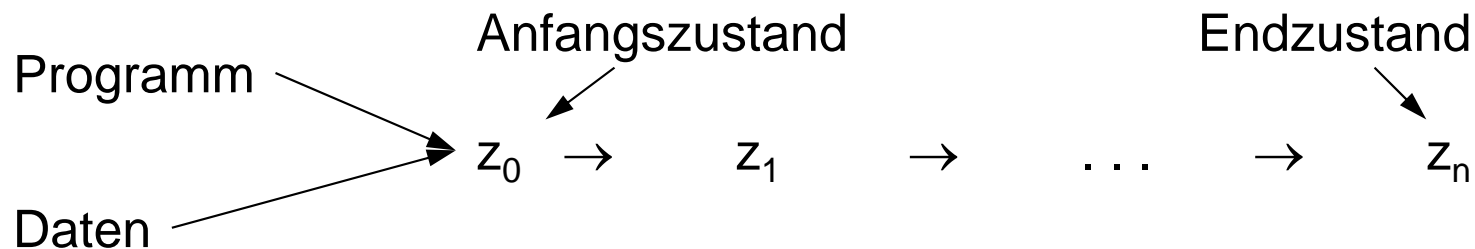
- P1: $\text{expression}_1.\text{TYP} \leftarrow \text{expression}_2.\text{TYP}$
 $\text{expression}_1.\text{WERT} \leftarrow \text{expression}_2.\text{WERT} + \text{term}.\text{WERT}$
Condition: $\text{expression}_2.\text{TYP} = \text{term}.\text{TYP}$ *Produktion P3 analog*
- P2: $\text{expression}.\text{TYP} \leftarrow \text{term}.\text{TYP}$
 $\text{expression}.\text{WERT} \leftarrow \text{term}.\text{WERT}$
Produktionen P4 und P5 analog
- P6: $\text{factor}.\text{TYP} \leftarrow \text{INT}$
 $\text{factor}.\text{WERT} \leftarrow \text{int_constant}.\text{WERT}$
Produktion P7 analog

3.2.2 Operationale Semantik

McCarthy (60-63), Landin (63)

Idee:

- Die Wirkung eines Programms ist bestimmt durch seine Ausführung auf einer abstrakten/konkreten Maschine. Zustände charakterisieren den Stand einer Berechnung



- Mathematische Beschreibung der Bedeutung von Ausdrücken und Anweisungen durch Funktionen bezüglich Zustandsvektoren:
 $\text{sem} : (\langle \text{Ausdruck} \rangle, \langle \text{Zustand} \rangle) \rightarrow \text{Wert}$
 $\text{sem} : (\langle \text{Anweisung} \rangle, \langle \text{Zustand} \rangle) \rightarrow \text{Zustand}^*$
- Folgen (rekursiver) Anwendung der mathematischen Funktionen auf Programmteile und entsprechende Zwischenzustände beschreiben Programmsemantik.
- durch Implementierung von „sem“ ausführbar (*interpretative Semantik*).

Eigenschaften der interpretativen Semantik:

- „formale“ Definition eines Interpreters für die Sprache
- **aber:** oft eine Überspezifikation der Sprache, weil durch interpretative Semantik alle in der Sprachdefinition beabsichtigten Freiheitsgrade verschwinden.
- Gefahr der rekursiven Definition („LISP durch LISP erklärt“)

Beispiel: VDL (Vienna Definition Language)

VDL-Zustände = Bäume mit Teilbäumen für

- Struktur des Programms
- Werte der Variablen
- Laufzeitdaten („Aktivierungsblöcke“)

Operationale Semantikbeschreibung erfordert:

- Festlegung von Zuständen
- Festlegung von Zustandsübergängen für jeden Bestandteil der Programmiersprachen

Beispiel: Sprache mit

```
<statement> ::= <identifier> ← <expression> |  
                while <expression> do <statement> |  
                <statement> ; <statement>
```

STATEMENT = Menge aller Anweisungsformen (Zuweisung, Schleife, Anweisungsfolge)

ZUSTAND = Menge von Paaren (Variable, Wert)

ENDE = letztes Element in einer Folge von Zuständen

EVAL = Funktion zur Ausrechnung eines Ausdrucks in einem gegebenen Zustand

SEMANTIK = STATEMENT \times ZUSTAND \rightarrow ZUSTAND*

* steht hier für eine Folge von Zuständen

Notationen:

$z[id]$ = Wert der Variablen id im Zustand z

$z[id \leftarrow w]$ = Zustand z' für den gilt:

$z'[id] = w$

$z'[x] = z[x]$ für $x \neq id$

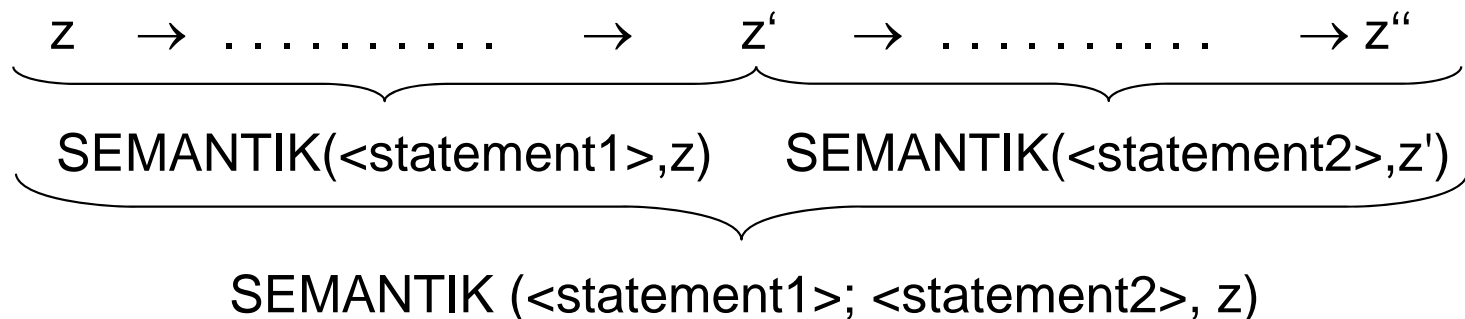
Semantik der Zuweisung:

$\text{SEMANTIK} (\langle \text{id} \rangle \leftarrow \langle \text{expression} \rangle, z) = z[\langle \text{id} \rangle \leftarrow \text{EVAL} (\langle \text{expression} \rangle, z)]$

Semantik der Anweisungsfolgen:

$\text{SEMANTIK} (\langle \text{statement1} \rangle; \langle \text{statement2} \rangle, z) =$
 $\text{SEMANTIK} (\langle \text{statement1} \rangle, z) \ \&$
 $\text{SEMANTIK} (\langle \text{statement2} \rangle, \text{ENDE} (\text{SEMANTIK} (\langle \text{statement1} \rangle, z)))$

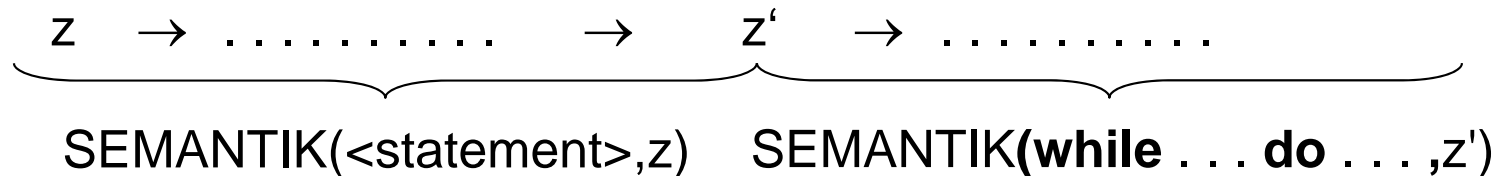
Erklärung



Semantik der Schleife:

```
SEMANTIK (while <expression> do <statement>, z) =  
  if EVAL (<expression>, z) then  
    SEMANTIK (statement,z) &  
    SEMANTIK (while <expression> do <statement>,  
              ENDE (SEMANTIK (<statement>, z)))  
  else z
```

Erklärung (falls Schleife nicht Leeraanweisung)



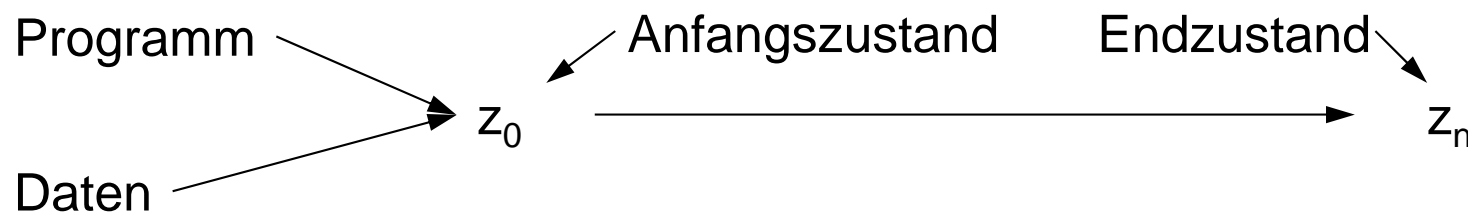
3.2.3 Denotationale Semantik

Scott, Strachey (70-71)

Idee:

- Bereits *die Regeln über die Zuordnung* zwischen den Bestandteilen einer Sprache und einer Menge von Funktionen sind ausreichend, um die Semantik eindeutig zu definieren.
- Beschreibung der Bedeutung von Anweisungen durch funktionale Eigenschaften:
 - Funktionalalküle anwendbar, speziell Fixpunktbestimmungen für Iterationen und Rekursionen
 - Möglichkeit, über Terminierungseigenschaften manchmal zu entscheiden
 - nicht ausführbar, aber zur interpretativen Semantik leicht verfeinerbar

Wirkung eines Programms entspricht einer Funktion, die einen Anfangszustand in einen Endzustand abbildet



Beachte: Im Gegensatz zur operationalen Semantik wird von Zwischenzuständen abstrahiert.

SEMANTIK = Abbildung

STATEMENT \rightarrow [ZUSTAND \rightarrow ZUSTAND]

s \rightarrow fs : ZUSTAND \rightarrow ZUSTAND

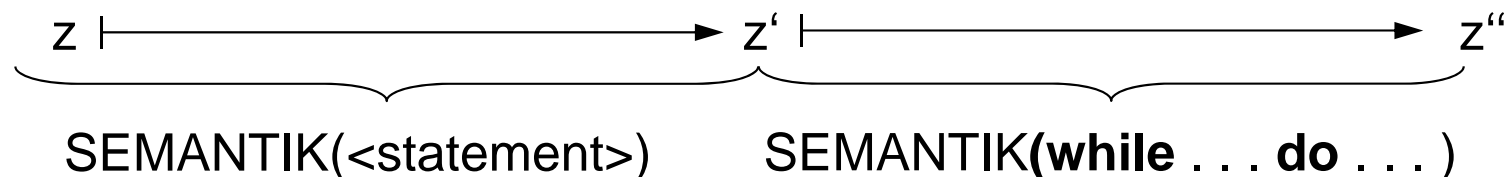
fs wird beschrieben durch fs(z) für beliebige z (, ... im Folgenden aber bereits in der Anwendung auf den jeweiligen Zustand dargestellt ...

SEMANTIK (<id> \leftarrow <expression>) (z) = z[id \leftarrow EVAL (<expression>, z)]

SEMANTIK (<statement1>; <statement2>) (z) =
SEMANTIK(<statement2>) o SEMANTIK(<statement1>) (z)

SEMANTIK (**while** <expression> **do** <statement>) (z) =
if EVAL (<expression>, z) **then**
SEMANTIK (**while** <expression> **do** <statement>)
o SEMANTIK (<statement>) (z)

else z



Hauptunterschied operationale/denotationale Semantik

- **Operationale Semantik:**

$\text{sem}(\text{while } A \text{ do } B, z) = \text{sem}(\text{if } A \text{ then } B; \text{ while } A \text{ do } B; \\ \text{else NULL; end if}, z) = \dots$

d.h. Semantik der While-Schleife wird durch rekursive Anwendungen der Semantik auf Zwischenzustände definiert.

- **Denotationale Semantik:**

$\text{sem}(\text{while } A \text{ do } B) =$
 $\text{sem}(\text{if } A \text{ then } B; \text{ while } A \text{ do } B; \text{else NULL; end if}, z) =$

$\text{sem}(\text{if } \dots \text{ then } \dots \text{ else } \dots \text{ end if}) :$

$(\text{sem}(A), \text{sem}(\text{while } A \text{ do } B) \circ \text{sem}(B), \text{sem}(\text{NULL}))$

$\equiv \text{FIX } (\lambda(f).\text{sem}(\text{if } \dots \text{ end if}) : (\text{sem}(A), f \circ \text{sem}(B), \text{sem}(\text{NULL})))$



Funktion, die den kleinsten Fixpunkt berechnet

d.h. Semantik der While-Schleife wird durch Fixpunkt des Funktionalen definiert.

3.2.4 Axiomatische Semantik

Floyd (67), Hoare (69)
Dijkstra (76)

Idee:

- Prädikatenlogik als Grundlage
- Abkehr von Zustandsorientierung (Zustände sind partiell durch Prädikate beschrieben)
- Beschreibung der Semantik durch Axiome und
 - Inferenzregeln (Floyd, Hoare)
 - „Prädikattransformatoren“ (Dijkstra)

Eigenschaften:

- benutzernah
- in Originalform sehr stark dem Programmtext verhaftet
- nicht ausführbar als semantisches Modell der Programmiersprache
- Beschreibung der „Benutzersemantik“ eines Programms und weniger auf die Beschreibung der PS-Semantik ausgerichtet.

Wirkung eines Sprachelements wird beschrieben durch eine Beziehung zwischen den Zuständen vor und nach seiner Ausführung:



Semantik elementarer Anweisungen wird beschrieben durch **Axiome** der Form

{VOR} <statement> {NACH}

Axiom der Wertzuweisung:

$\{P[\text{<id> ersetzt durch <expression>}]\} \leftarrow \text{VOR}$
 $\text{<id>} \leftarrow \text{<expression>}$
 $\{P\} \leftarrow \text{NACH}$

Die Semantik strukturierter Anweisungen wird beschrieben durch Formeln der Art:

$$\frac{\text{Voraussetzungen}}{\text{Schlussfolgerungen}}$$

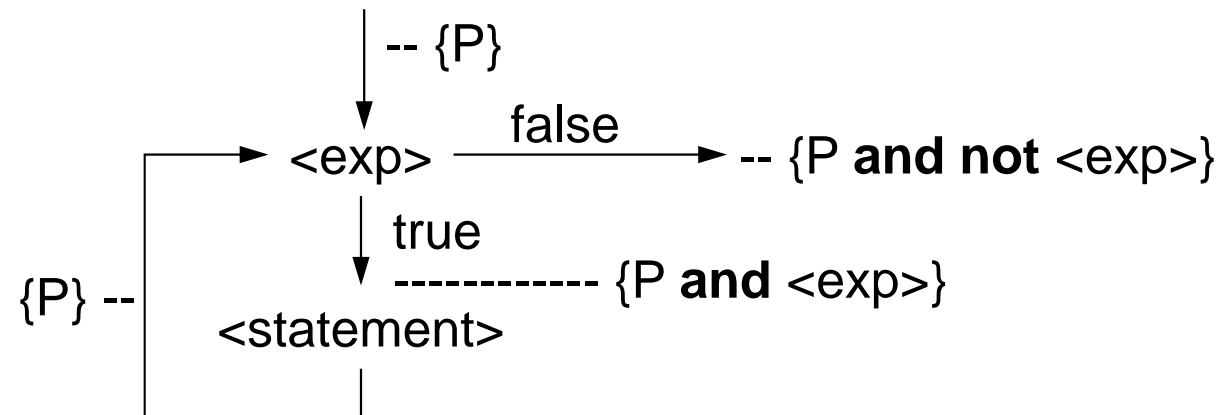
Zusammengesetzte Anweisungen:

$$\frac{\{P\} \text{ Anweisung1 } \{R\}, \{R\} \text{ Anweisung2 } \{Q\}}{\{P\} \text{ Anweisung1}; \text{ Anweisung2 } \{Q\}}$$

while-Schleife:

$$\frac{\{P \wedge Q\} \text{ Anweisung } \{P\}}{\{P\} \text{ WHILE } Q \text{ DO Anweisung } \{P \wedge \neg Q\}}$$

Erklärung:



Prädikamentransformatoren (Dijkstra):

Die Semantik der PS-Konstrukte wird durch Funktionen wiedergegeben, durch die die Prädikatsänderung von Vor- zu Nachbedingungen bzw. von Nach- zu Vorbedingungen beschrieben sind.

Informell (orientiert an „ $\{P\}$ *Konstrukt* $\{Q\}$ “ in der Hoare-Notation):

$wp(\text{Konstrukt}, Q) = P$ -- „weakest precondition“ (totale Korrektheit)

$sp(\text{Konstrukt}, P) = Q$ -- „strongest postcondition“ (totale Korrektheit)

und „wlp“, „slp“ für entsprechende Transformationen bei partieller Korrektheit
(„l“ = „liberal“)

3.3 Nicht-deterministische Semantik

```
declare
  X: Integer;

  function f return Integer is
  begin
    X := X + 1;
    return X;
  end f;

  function g return Integer is
  begin
    X := X * 2;
    return X;
  end g;

begin
  X := 1;
  PUT(f + X * g);
end;
```

Welche Ausgabe liefert dieses
Programm ?

Realisierung der Semantik in der Ausführung

Sei $S_{PS} : \text{Programm} \times \text{Eingabedaten} \rightarrow \text{Ausgabedaten}$ die Semantik der Programmiersprache PS

Sei S_C die entsprechende, durch Übersetzung und Ausführung der Programme erhaltene Abbildung („durch Compiler realisierte operationelle Semantik“)

Idealerweise:

$$\forall P, I: S_{PS}(P, I) = A \quad \text{gdw.} \quad S_C(P, I) = A$$

Realität:

$$\exists P, I: S_{PS}(P, I) = A_1 \vee A_2 \vee \dots \vee A_n$$

zulässige impl.-abhängige Alternativen

Konsequenz:

$$\begin{array}{ll} S_{C1}(P, I) = A_2 & ! \text{ dennoch beide} \\ S_{C2}(P, I) = A_5 & ! \text{ Compiler korrekt} \end{array}$$

Bedingung für korrekte Compiler:

$$\forall P, I: S_C(P, I) = A_{P, I, C} \quad \Rightarrow \quad A_{P, I, C} \in S_{PS}(P, I)$$

Kapitel 4

Speichermodelle – Stapel und Halde

4. Speichermodelle – Stapel und Halde

Speichermodelle haben einen zentralen Einfluss auf die Gestaltung von Programmiersprachen und stehen in interessanter Wechselwirkung mit den verwendeten Programmierparadigmen. Wir besprechen daher die relevanten Aspekte der Speichermodelle im Detail.

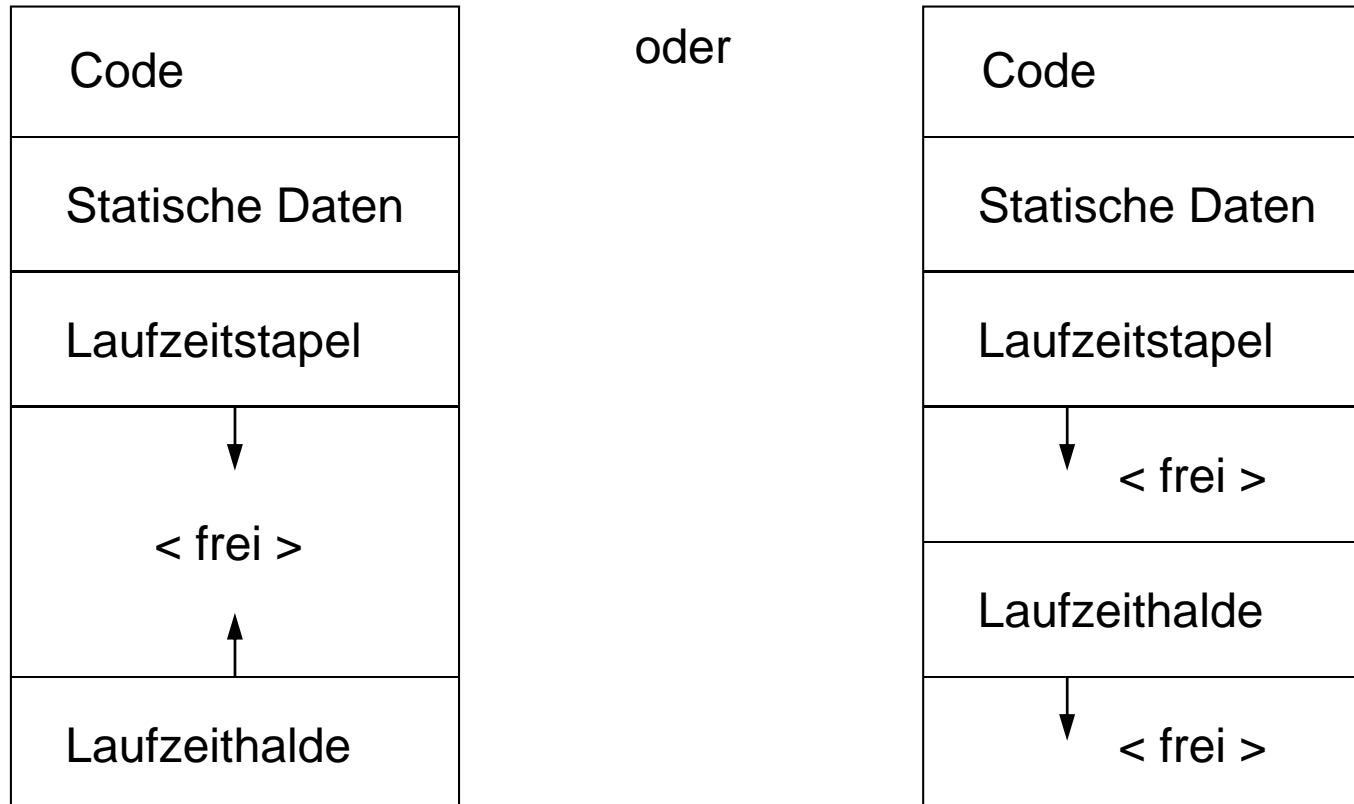
Wir benötigen Speicher für

- globale Daten (Variablen, Konstanten)
- Codesegmente für Unterprogramme und andere Kontrollstrukturen
- lokale und administrative Daten dieser Konstrukte (→ Aktivierungsblöcke)
- Daten, deren Lebensdauer nicht an Kontrollstrukturen gebunden ist (→ Halde)

Speicher wird vergeben

- vor der Programmausführung (globale Daten, Codesegmente)
- während der Programmausführung nach Stapelverfahren
- während der Programmausführung im Haldenverfahren

Prinzipielle Speicheraufteilung (ohne Threads, Coroutinen etc.)



4.1 Aktivierungsblöcke

Für die Aktivierung von Unterprogrammen (u.Ä.) wird Speicher ("Aktivierungsblock") benötigt für

- Parameter und Resultat des Unterprogramms,
- lokale Variablen,
- vom Übersetzer erzeugte Hilfsvariablen,
- administrative Daten, wie Rückkehradresse, usw.

Je nachdem, ob

- die Größe der Aktivierungsblöcke vor Ausführung bekannt ist, und
- jeweils nur ein Aufruf des selben Unterprogramms aktiv sein kann (d.h. Rekursion nicht erlaubt ist),

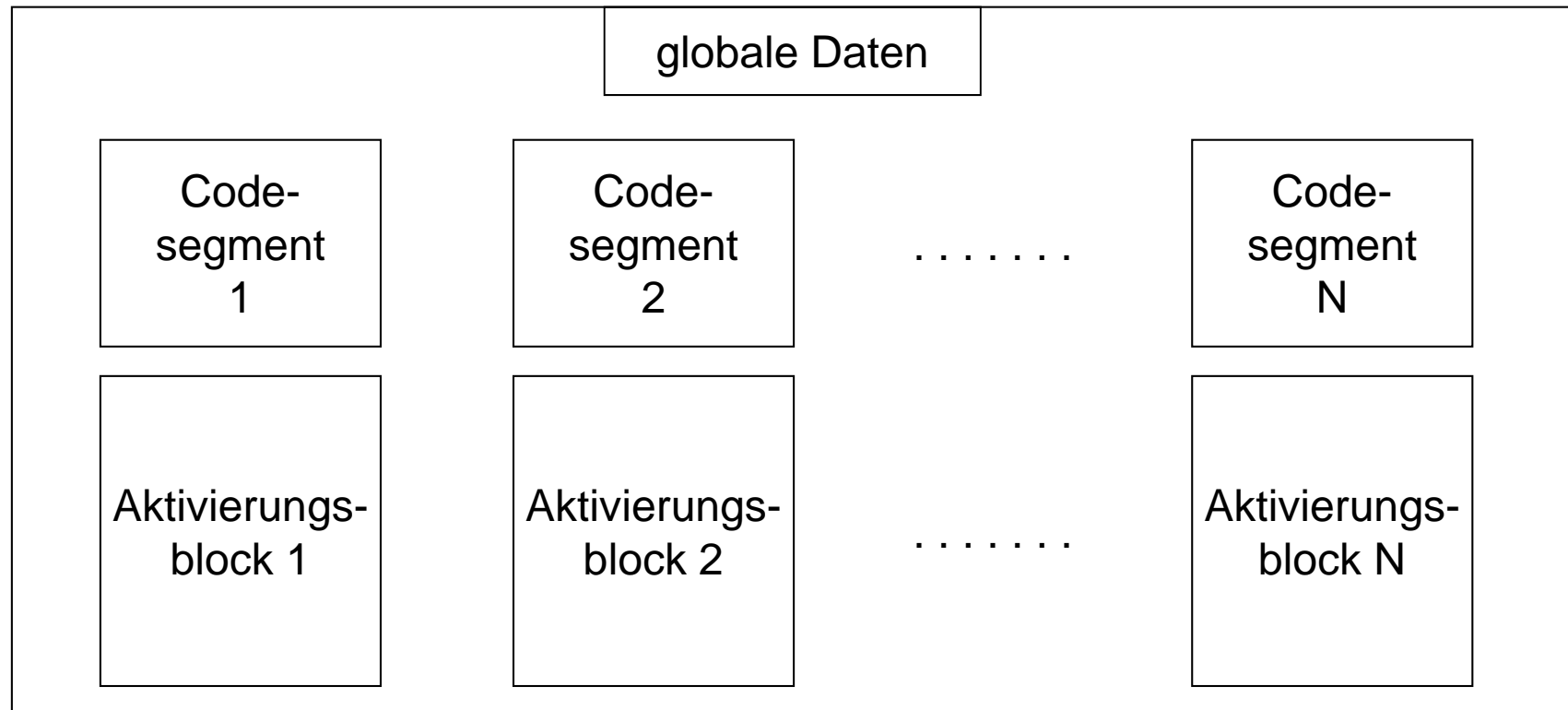
werden Aktivierungsblöcke statisch oder dynamisch angelegt.

4.2 Statische Verwaltung von Aktivierungsblöcken

Voraussetzungen: siehe oben

Beispiel: („Ur-“)FORTRAN

- Dem Hauptprogramm und allen Unterprogrammen wird je **ein** Aktivierungsblock fester Größe zugeordnet. Den Variablen werden vor der Ausführung feste Adressen zugeordnet.



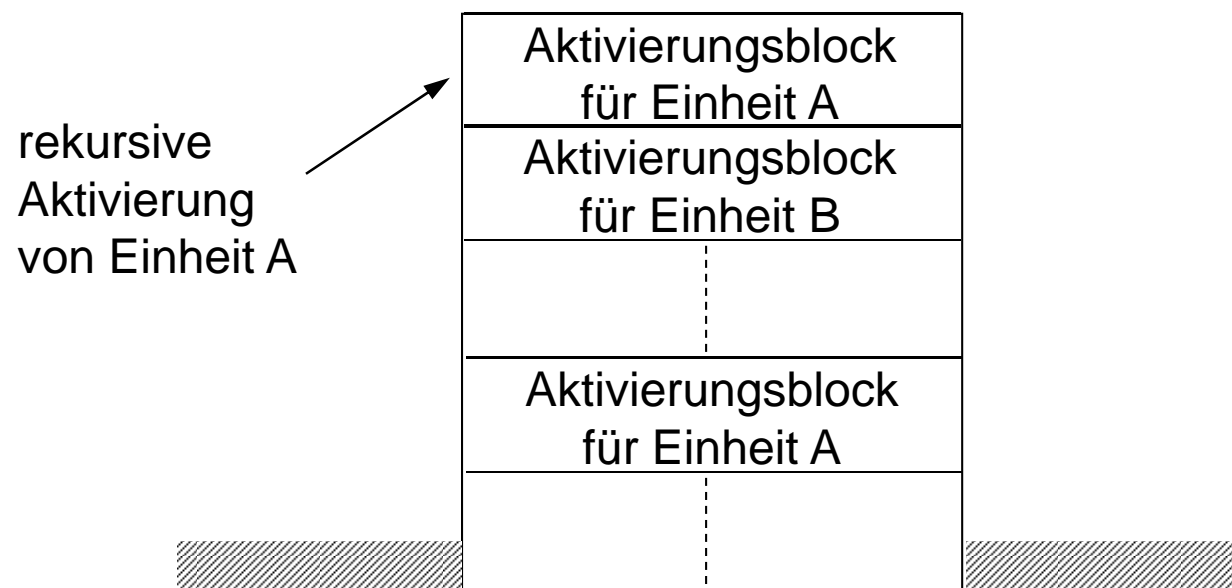
Beispiel: („Ur-“)FORTRAN Fortsetzung

- Alle Aktivierungsblöcke existieren unabhängig von Unterprogrammaufrufen während der gesamten Programmlaufzeit.
- Da bei jeder Aktivierung einer Programmeinheit derselbe Speicherbereich verwendet wird, bleiben die Werte lokaler Variablen nach Beendigung einer Aktivierung erhalten. (Einfache Implementierung der „own“-Variablen-Semantik)
- keine Rekursion möglich

Wegen des letzten Punkts findet dieses Modell nur noch auf eingeschränkten eingebetteten Rechnern Verwendung, wobei dann die Restriktion "keine Rekursion" durch Style Guide verordnet wird und der Compiler die Einhaltung der Restriktion annimmt.

4.3 Dynamische Verwaltung von Aktivierungsblöcken

- z.B. in Java, C, C++, Ada, ... fast allen weitverbreiteten Sprachen
- Aufgrund der Rekursion müssen mehrere Aktivierungsblöcke desselben Unterprogramms existieren können.
- Aufgrund der strikten Nestung von Aufrufen und deren Beendigung ist die Verwaltung der Aktivierungsblöcke in Form einer Stapelverwaltung möglich.
- Aktivierungsblöcke werden auf einem Laufzeitstapel verwaltet.



- bei Aktivierung einer Programmeinheit: Anlegen eines Aktivierungsblocks an der Spitze des Laufzeitstapels
- beim Erreichen des Endes einer Programmeinheit: Wegnahme des Aktivierungsblocks von der Spitze des Laufzeitstapels
- lokale Daten werden nach Beendigung des Aufrufs gelöscht bzw. vom nächsten angelegten Aktivierungsblock überschrieben
- Die Größe von Aktivierungsblöcken kann
 - Fall 1: bei jeder Aktivierung gleich sein,
 - Fall 2: von Aktivierung zu Aktivierung unterschiedlich sein, jedoch zum Zeitpunkt der Aktivierung bestimmt werden,
 - Fall 3: sich während der Aktivierung ändern.

Die Fallunterscheidung wird im Wesentlichen durch die Eigenschaften des Typmodells bestimmt, insbesondere die statische oder dynamische Größe der lokalen Variablen.

Zu Fall 1:

- Voraussetzung: Größe aller lokalen Daten statisch bekannt (Java, C vor C99)
- Für lokale Blöcke wird der insgesamt maximal benötigte Speicherbedarf angelegt. Die Überlagerung des Speichers für nicht verschachtelte Blöcke wird bereits zur Übersetzungszeit durchgeführt, so dass der maximale Bedarf statisch bekannt ist. (\Rightarrow Optimierungen im Compiler)
- Die Größe administrativer Daten ist statisch und daher ebenfalls zur Übersetzungszeit bekannt.
- Die Größe der Parameter ist ebenso statisch (da Daten dynamischer Größe praktisch immer per Referenz übergeben werden).
- Dementsprechend ist die Länge des notwendigen Aktivierungsblocks dem Compiler bekannt.

Zu Fall 2:

- Sprache enthält Deklarationen der Form

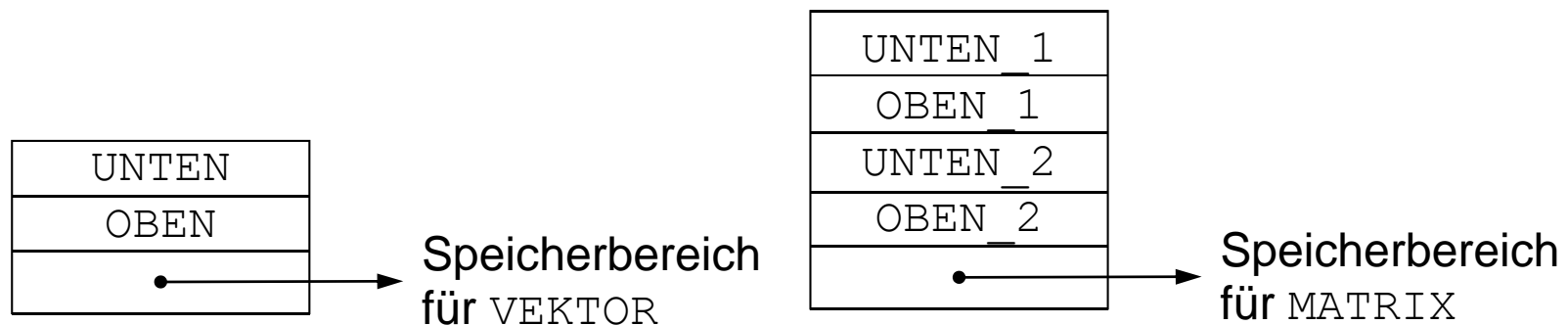
VEKTOR : **array** [UNTEN..OBEN] **of** INTEGER **oder**

MATRIX : **array** [UNTEN_1..OBEN_1, UNTEN_2..OBEN_2]
 of REAL

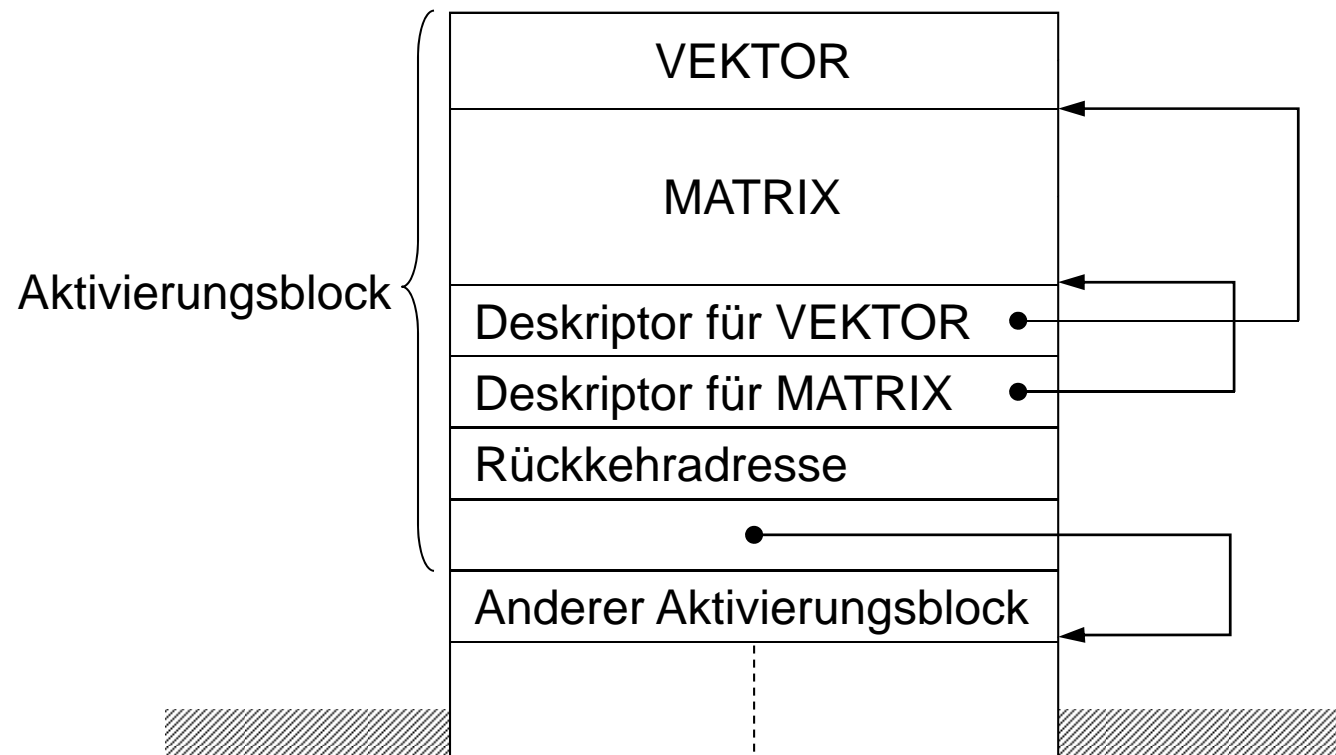
UNTEN, UNTEN_1, UNTEN_2, OBEN, OBEN_1 und OBEN_2 sind Variablen, deren Wert bei jeder Aktivierung der diese Deklaration enthaltenden Programmeinheit bekannt sein muss (z. B. als Parameter).

⇒ Der Platzbedarf für VEKTOR und MATRIX ist nicht bei der Übersetzung, aber bei jeder Aktivierung bekannt.

- für VEKTOR und MATRIX können Deskriptoren folgender Form angelegt werden:



- Bei Aktivierung der Programmeinheit von VEKTOR und MATRIX wird ein Aktivierungsblock mit den Deskriptoren und dem bei Aufruf berechenbaren Platz für VEKTOR und MATRIX an der Spitze des Laufzeitstapels angelegt.



- Zugriff auf Werte der lokalen Variablen erfolgt durch Indirektion über den Deskriptor.

Vorbemerkungen zu Fall 3:

- Viele Sprachen enthalten Ausdrucksmittel wie
$$\text{NEW } (P)$$
zur Erzeugung von Objekten während der Laufzeit.
- Freigabe des dynamisch angeforderten Speicherbereichs geschieht entweder implizit oder explizit durch $\text{DISPOSE } (P)$, $\text{FREE } (P)$, u. ä.
- Semantik von NEW impliziert meistens (aber nicht in allen Sprachen!), dass solche Objekte über die Dauer der Ausführung des Unterprogramms hinaus existieren.
 \Rightarrow Diese Objekte dürfen nicht im Aktivierungsblock angelegt werden.
- In Aktivierungsblöcken wird kein Speicherplatz für solche Objekte angelegt, lediglich ein Zeiger auf diese Speicherbereiche in der Halde



Fall 3a:

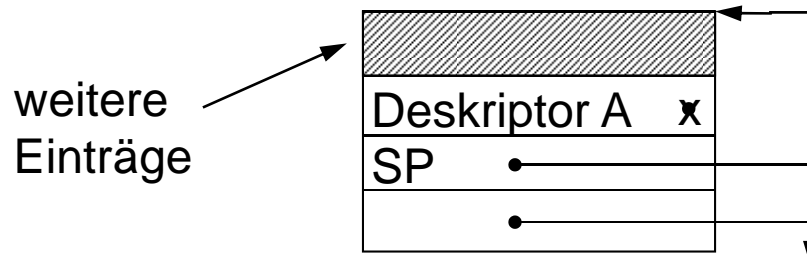
Größe von lokalen Variablen ist erst zum Zeitpunkt der Deklaration berechenbar, z. B.

```
A: Matrix(1..f(x), 1..g(y))
```

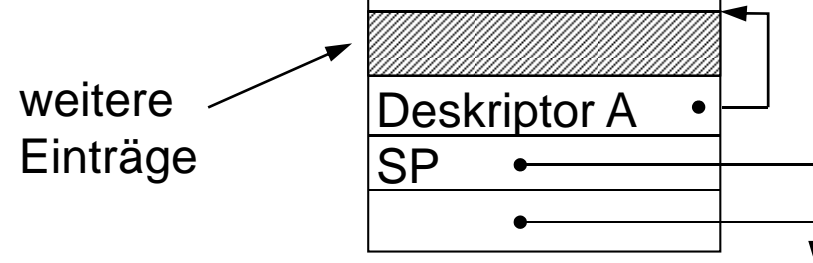
Speicherplatz wird bei Auswertung der Deklaration (durch Hochzählen eines Kellerpegels) angelegt. (Für lokale Variablen in verschachtelten Blöcken wird der Pegel nach dem Verlassen des Blocks zurückgesetzt.)

Kellerpegel wird als zusätzliches administratives Datum von ABs verwaltet.

vorher (z. B. bei Aufruf):



nachher:



Alternative: Anlage des Objekts auf der Halde und des Verweises im Stapel.

Vorteil:

- Die Größe des AB bleibt statisch

Nachteile:

- Fragmentierungsproblem der Halde (s.u.)
- erhöhter Verwaltungsaufwand für Garantie der später notwendigen impliziten Freigabe

Fall 3a in Programmiersprachen:

Ada: erlaubt und von den meisten Compilern mit Stapelallokation realisiert

Java: Alle Arrays sind Objekte, die explizit vom Benutzer (mittels `new`) auf der Halde allokiert werden müssen. Es gibt nur eindimensionale Arrays.

C/C++: Arrays mit statischen Grenzen in allen Dimensionen werden im Aktivierungsblock abgelegt. Seit C99 werden auch Arrays mit dynamischen Grenzen so behandelt. Vorher mussten sie explizit vom Benutzer (mittels `malloc`) allokiert werden. Einige (Nicht-Standard-) Bibliotheken bieten auch einen Stack-Allokator an, der wie beschrieben für den Wertespeicher sorgt.

Anmerkung: `malloc` mit Vorsicht programmieren, da eine Größenangabe erforderlich ist. `malloc(sizeof(mytype))` ist ein gutes Idiom, geht aber nicht für Arrays dynamisch fester Größe. Daher:

```
malloc(elementnumber * sizeof(elementtype))
```

Fall 3b:

Größe einer lokal deklarierten Variablen variiert dynamisch

Beispiele:

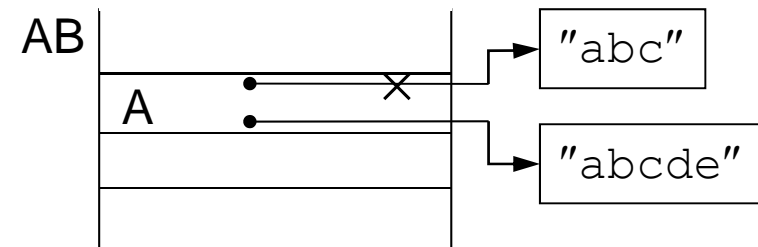
- 1) „flex array“ in ALGOL 68
- 2) Zeichenketten dynamisch veränderbarer Länge, z.B. in SNOBOL
- 3) Matrizen in APL
- 4) Klassen-Objekte in vielen OOP-Sprachen
- 5) „discriminated records“ in Ada und Pascal

Alternativen:

- Speicherzuweisung maximaler Größe
⇒ nicht möglich für 1), 2), 3) und viele Fälle von 4) und 5)
 - Dynamische Speichervergabe bei jeder Zuweisung
 - im Stapel (Fragmentierungsprobleme wie bei → Halde)
 - auf der Halde
- A ← "abc"

...

A ← "abcde"
- → Referenzsemantik der Zuweisung in OOP



Zusammenfassung Aktivierungsblöcke

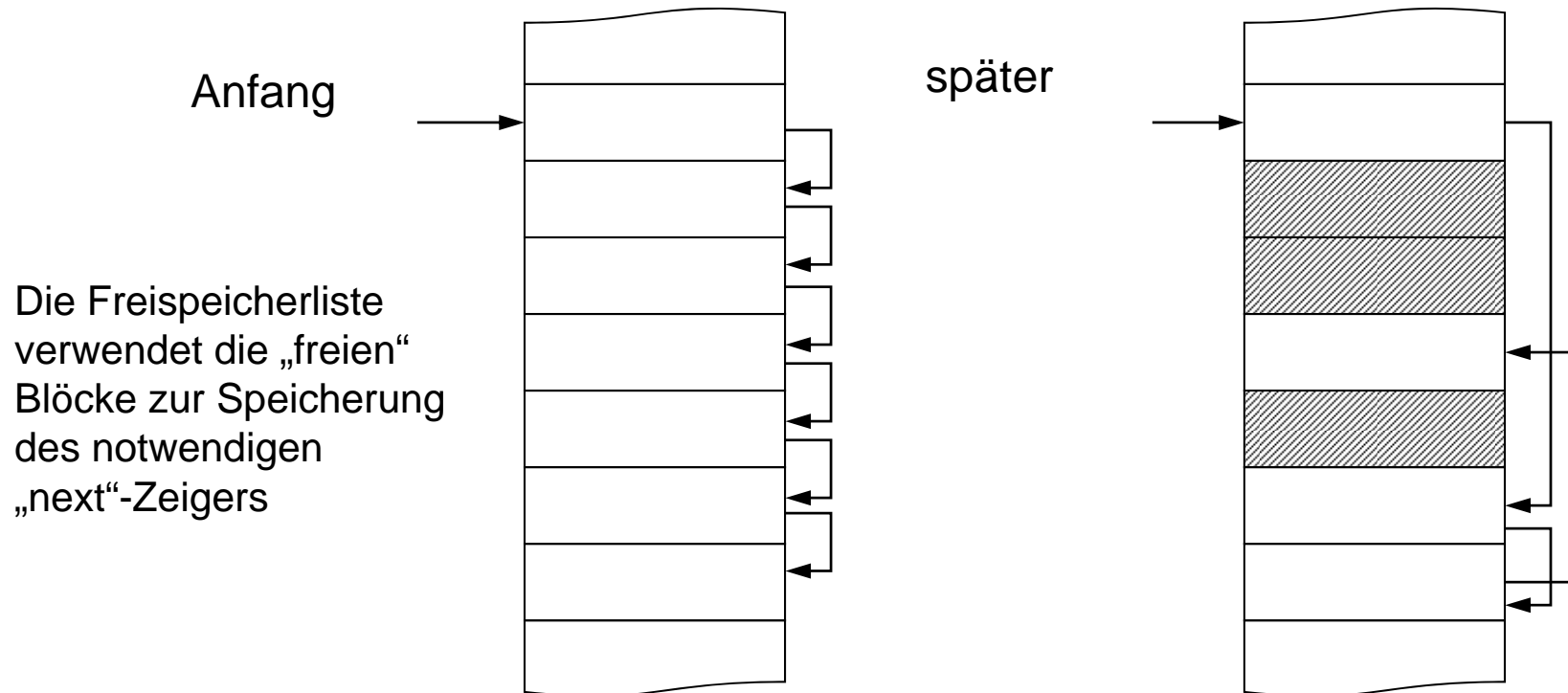
- Rekursive Aufrufe möglich
 - ⇒ Notwendigkeit je eines neuen Aktivierungsblocks pro Aufruf, i. Allg. auf dem Laufzeitstapel
- Objekte möglich, deren Größe erst zur Ausführungszeit bekannt, aber fest ist
 - ⇒ 1) Allokation der Objekte auf der Halde oder
 - 2) AB von dynamischer Größe
 - 2) ist einfacher und schneller und vermeidet Fragmentierung auf der Halde
- Objekte möglich mit dynamischer Größe
 - ⇒ typisch: Allokation dieser Objekte auf der Halde

4.4 Dynamische Speicherverwaltung nach dem Halden-Prinzip (Heap)

- Organisation der Halde mit Blöcken fester oder variabler Größe
- Grundidee:
 - freie Blöcke werden durch eine Freispeicherliste (alternativ: Bitvektor der Belegung der Speicherblöcke) verwaltet
 - bei Anforderung wird ein passender Block von der Freispeicherliste genommen und zugeteilt
 - bei Freigabe wird der Block wieder in die Freispeicherliste eingetragen
 - wenn kein Speicher mehr verfügbar ist, evtl. Versuch, vom BS einen weiteren großen Block zu erhalten. Alternativ: Stapel und Halde wachsen „gegeneinander“, bis kein freier Speicher mehr verfügbar ist.
- Typischerweise verwendet für die Realisierung der Zeigersemantik mit expliziter Allokation und Deallokation im Programm
- Gefahren bei expliziter Freigabe durch Programmierer
 - Zeiger ins Leere („zu frühe Freigabe“, dangling reference)
 - Abfall („vergessene Freigabe“, vom Programm nicht mehr erreichbarer Speicher)

4.4.1 (Teil-)Heaporganisation bei Elementen fester Größe

- Aufteilung der Halde in gleichgroße Blöcke



- Anforderung einfach („pop von Liste“)
- Freigabe einfach, falls bekannt ist, was freigegeben wird („push auf Liste“)

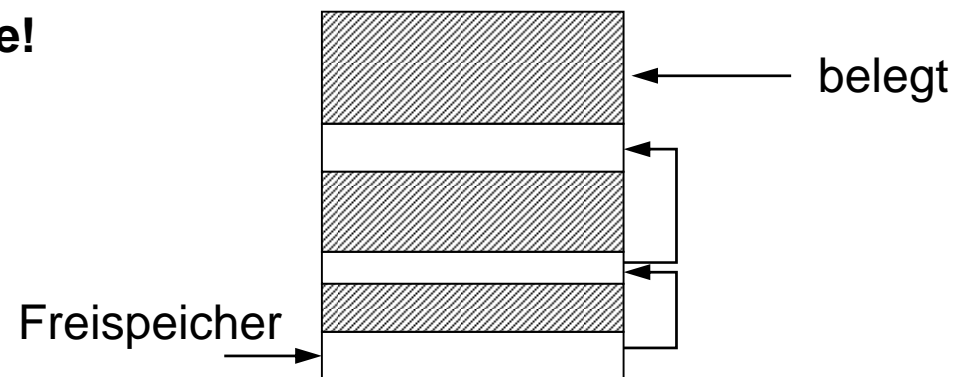
4.4.2 Heap-Organisation bei Elementen unterschiedlicher Größe

- Heap ist anfänglich ein Speicherblock



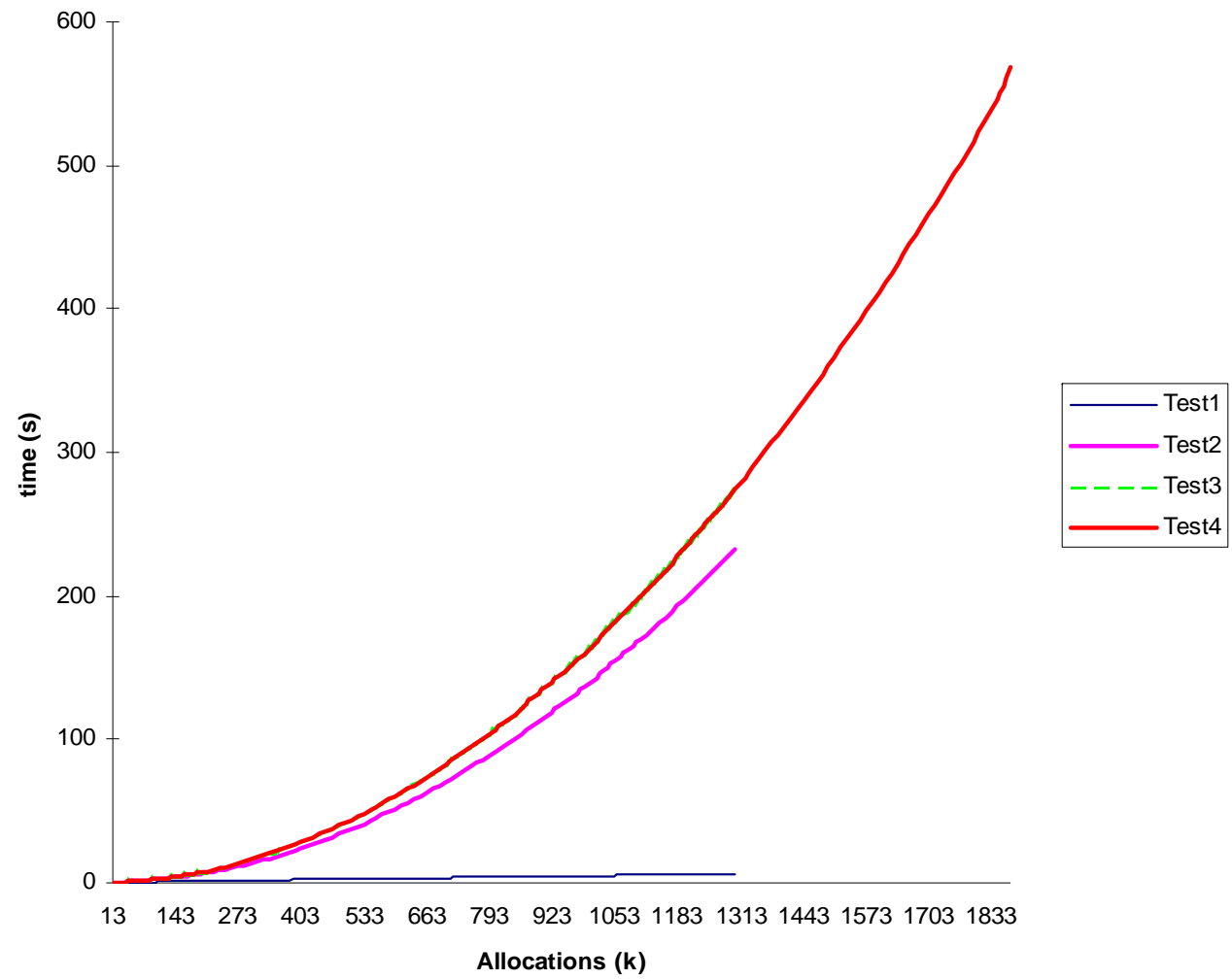
- Anforderungen werden erfüllt durch „Herausschneiden“ aus einem geeigneten (s. u.) freien Block auf der Freispeicherliste.
- Verbleibende Reste und Freigaben werden wie bisher in der Freispeicherliste mit Größenangaben festgehalten. Die Halde füllt sich mit Kleinstresten, die die Suche verlangsamen.

⇒ **Fragmentierung der Halde!**

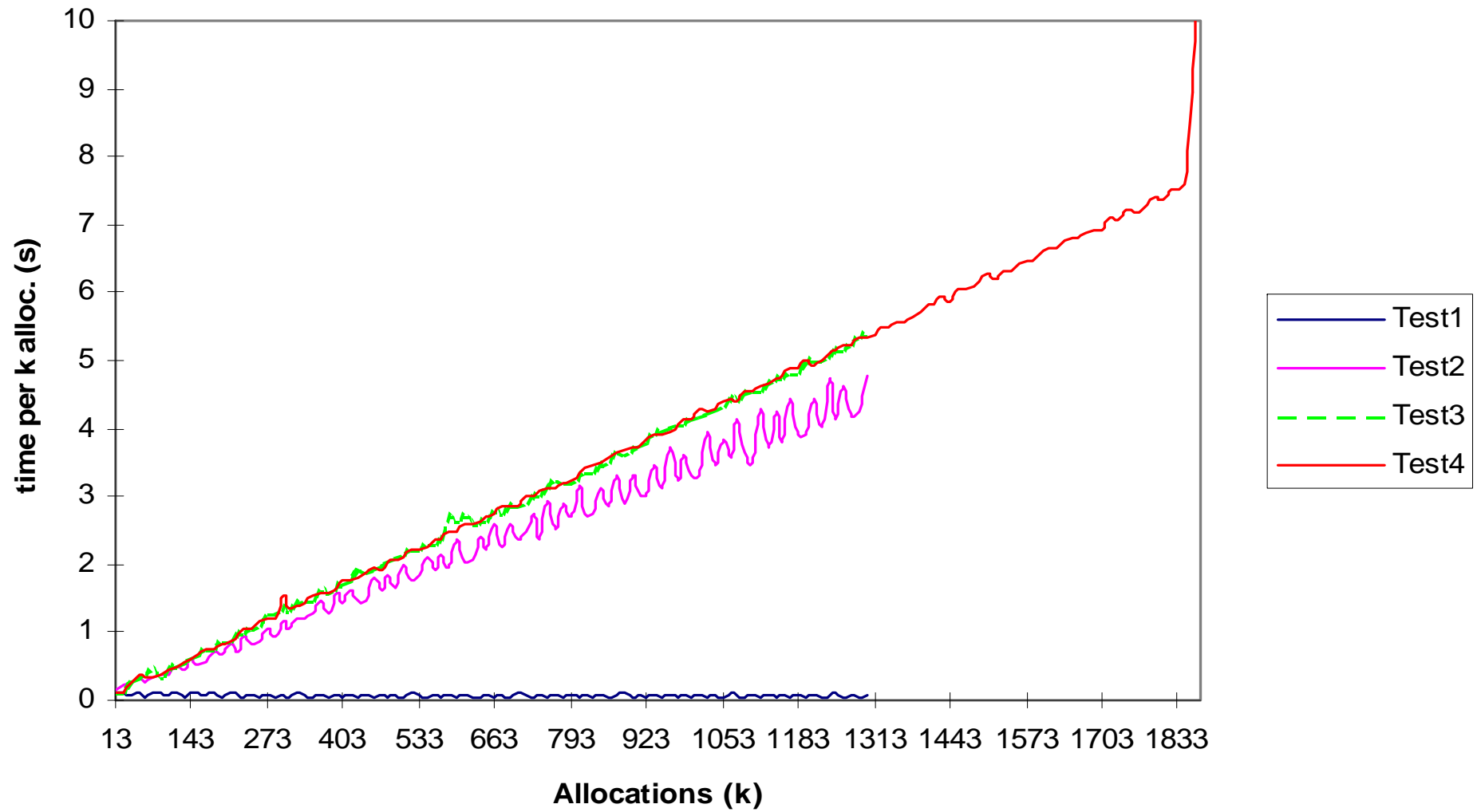


Irgendwann ist keiner der freien Bereiche ausreichend groß für die anstehende Anforderung. Wenn möglich, wird jetzt ein weiterer großer Block vom BS angefordert.

Laufzeit eines haldenintensiven Programms



1. derivative (deltas)



„Lösung“ 1: Versuche, Fragmentierung durch Vergabestrategie zu minimieren

Lösung 2: Freie Speicherblöcke zusammenschieben („Kompaktifizierung“)

- zu Lösung 1: Suche eines passenden Blocks in der Freispeicherliste nach
 - **best-fit-Methode:** Auswahl des Blocks mit dem kleinsten Überschuss
 - ⇒ kleine, unverwendbare Restblöcke bleiben übrig
 - ⇒ Suche nach best-fit ist kostspielig
 - **first-fit-Methode:** Auswahl des ersten Blocks mit ausreichender Größe
 - ⇒ kleine Teile akkumulieren am Beginn der Liste (degenerierende Suchzeiten)
 - **next-fit-Methode:** Freispeicherliste ist Ringliste; der „Anfang“ rotiert in der Liste; sonst wie first-fit
 - ⇒ verteilt die kleinen Reste über die Halde
 - **buddy-Methode:** suche nach genauer Größe; wenn nicht vorhanden, zerteile einen Block mit (z. B.) doppelter Größe
 - ⇒ größere Hoffnung auf spätere Vergabe des Rests
 - **worst-fit-Methode:** Auswahl des Blocks mit größtem Überschuss
 - usw. . . .

Bei allen diesen Varianten:

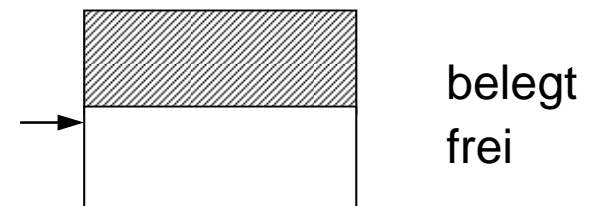
- Es gibt Beispiele für die Überlegenheit der jeweiligen Strategie gegenüber allen anderen!
- Es gilt das Knuth'sche Gesetz, dass auf Dauer diese Strategien zu einem Zustand führen, in dem belegte zu freien Blöcken im Verhältnis 2:1 stehen.

D.h. **Fragmentierung ist so nicht vermeidbar!**

empirisch: next-fit oder first-fit noch „am besten“

- zu Lösung 2:
 - teilweise Kompaktifizierung: Vereinigung aneinandergrenzender freier Speicherblöcke
⇒ teuer mit Freispeicherliste (Sortieren notwendig)
kostenlos mit Bitvektor-Implementierung

- vollständige Kompaktifizierung:
Zusammenschieben aller belegten Blöcke



Schwierigkeit: Anpassung von Zeigern auf verschobene Bereiche (siehe Garbage Collection zur Behandlung dieser Thematik)

4.4.3 Methoden zur sicheren Freigabe

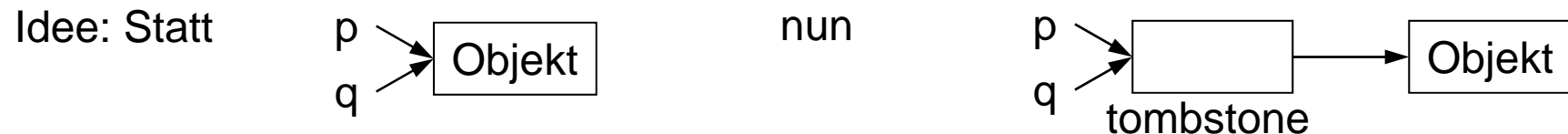
Bei expliziter Freigabe durch DISPOSE besteht die Gefahr einer zu frühen Freigabe, es entstehen „dangling references“.

Beispiel: `p := NEW Rec (...);` -- Allokation
`q := p;`
`DISPOSE(q);` -- Deallokation
`... p.comp := 10;` -- kann Freispeicherliste zerstören !!!

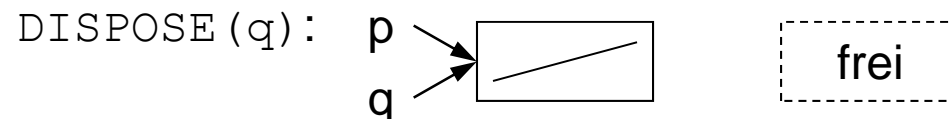
Um diese Gefahr unschädlich zu machen, gibt es die Methoden „tombstones“, „key and lock“ und „reference counting“.

Voraussetzung: keine (legale) Adressarithmetik in der PS!

„tombstone“-Methoden



„DISPOSE“ gibt das Objekt frei und setzt den „tombstone“ auf NULL.

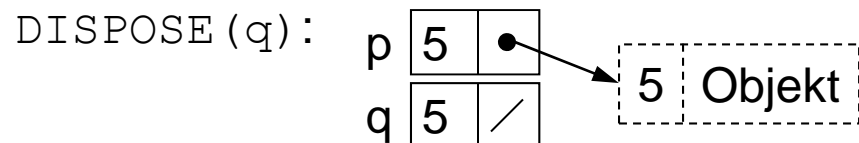
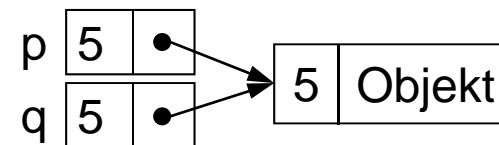


Aber die Zugriffe erfordern dann doppelte Dereferenzierung, und die Freigabe der „tombstones“ ist schwierig.

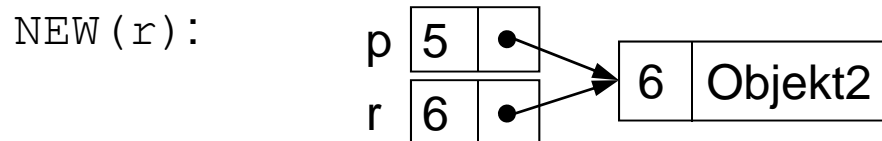
„key-and-lock“-Methoden

Idee:

- Heap-Objekte haben „locks“, die eindeutig sind und bei Allokation vergeben werden.
- Zeiger haben „keys“, die bei Allokation gleich dem Lock gesetzt werden.
- Zugriff nur legal, wenn Key = Lock



noch „vorhanden“ bis Speicherplatz wiederverwendet wird



Speicherplatz wiederverwendet

„p.all“, resp. „*p“ nun erkennbar illegal, da Key ungleich Lock ist, aber es entstehen Kosten für den Speicherplatz und die Prüfung jedes Zugriffs und Probleme bei der Kompaktifizierung.

4.4.4 Methoden zum „Abfall-Recycling“

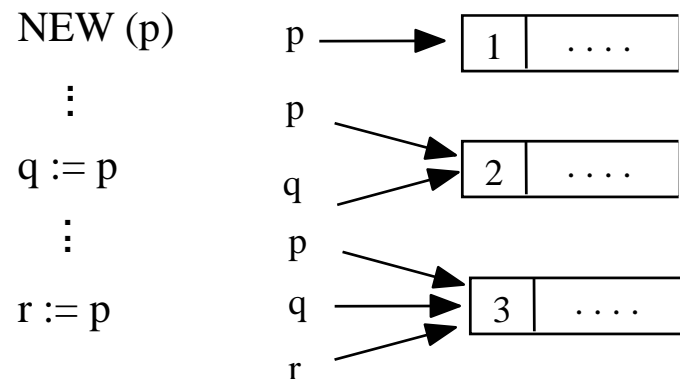
Wenn der Programmierer vergisst, Speicherplatz freizugeben bevor die Lebensdauer des letzten Zeigers auf den Speicherplatz endet, wird dieser Platz unerreichbar („Abfall“).

Methoden zur Verhinderung:

- „Reference Counting“
- „Garbage Collection“

„Reference Counting“-Methode

Jedes Heap-Objekt enthält Zähler mit Anzahl der darauf gerichteten Zeiger, z.B.



Freigabe des Speicherblocks erst und sobald der Zähler den Wert 0 erreicht.

Die Zähler müssen natürlich ihrer Semantik entsprechend verwaltet werden:

- Initialisiere bei der Allokation des Objekts
- bei Zuweisungen „ $p := q$ “ und anderen Zeiger-kopierenden Sprachkonstrukten (z. B. Parameterübergabe)
- erhöhe den Zähler von $q.all$ für ($q \neq \text{NULL}$)
- verringere den Zähler des alten $p.all$ für ($p \neq \text{NULL}$)
- erreicht der Zähler den Wert 0, gib das Objekt frei
- bei Verlassen eines Blocks oder Unterprogramms dekrementiere die Zähler von $x.all$ für alle lokal deklarierten Zeiger x .
- bei Freigabe eines Objekts dekrementiere die Zähler von $x.all$ für alle Zeigerkomponenten x des freizugebenden Objekts.

Vorteile:

- keine Zeiger ins Leere (d. h. sichere Freigabe)
- implizite Freigabe von Abfall

Nachteile:

- keine Freigabe unerreichbarer Ringlisten (oder sonstiger zyklischer Strukturen) und von dort erreichbarer Heap-Objekte!

Aufwand für Zählerverwaltung

4.4.5 Garbage-Collection-Technik

Idee: „Abfall ist weniger gefährlich als Zeiger ins Leere“

⇒ so lange Abfall produzieren, bis neue Speicheranforderungen nicht mehr erfüllbar sind, dann, oder auch parallel zur Programmausführung, Abfall einsammeln. (Explizite Freigaben werden ignoriert.)

Standardlösung:

Phase 1: Markierung aller erreichbaren Haldenobjekte

Phase 2: Freigabe der nicht markierten Haldenobjekte und evtl. Kompaktifizierung

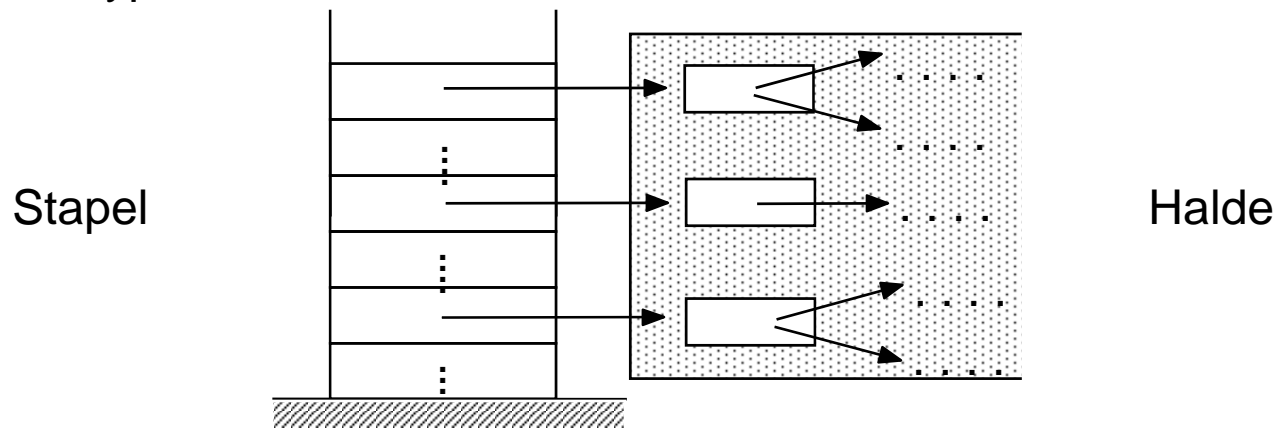
Technische Standardprobleme:

- Nicht-deterministische „Unterbrechung“ der normalen Ausführung
- Identifikation aller Zeiger auf erreichbare Elemente schwierig (s. u.)
- Umgang mit unsicherem Zeigertyp-Modell in der Programmiersprache

Markierungsphase

- Jedes Heap-Objekt enthält ein Markierungsbit.
- Anfangsbelegung sei 0. Markierung setzt das Bit der erreichbaren Objekte auf 1.
- Markierungsalgorithmen entsprechen dem Durchlaufen eines gerichteten Graphen, z. B. Tiefensuche, Breitensuche o. ä.

Ausgangspunkte sind die von außerhalb des Heaps erreichbaren Heap-Objekte. Typische Situation:

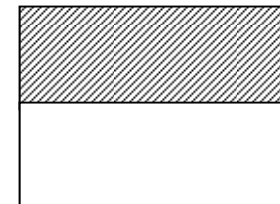


zu markieren sind (d. h. $\text{Bit} := 1$):

1. alle vom Stapel aus erreichbaren Heap-Objekte sowie
2. transitiv alle von markierten Heap-Objekten aus erreichbaren Heap-Objekte

Sammelphase

- Jedes Heap-Objekt mit `Bit=1` wird im Speicher auf einen kompaktifizierten Heap verschoben, und die es referenzierenden Zeiger werden angepasst.
- Die Markierungsbits der Heap-Objekte mit `Bit=1` werden nach der Verschiebung auf 0 zurückgesetzt.
- Der nach der Kompaktifizierung verbleibende Heap-Anteil wird als Speicherblock der Freiliste angefügt.



belegt
frei

Voraussetzung für Funktionsfähigkeit des Markierungsalgorithmus:

- 1) Zeiger von außerhalb des Heaps müssen erkennbar sein.
- 2) Zeiger aus Heap-Objekten auf andere Heap-Objekte müssen erkennbar sein.

⇒ Setzt Kenntnis der Typen gespeicherter Werte zur Laufzeit voraus !!

Lösung in LISP:

- 1) Alle Zeiger in den Heap von außerhalb beginnen bei genau definierten Startpunkten, z. B. A-List, AB-List etc.
- 2) Heap-Objekte besitzen einige wenige, vordefinierte Formate (Atome und „dotted pairs“ (im Bild)):



Allgemeine Lösung für 2) ist die Bereitstellung von Deskriptoren, die den Aufbau (insbesondere die Position von Zeigern) von Heap-Objekten beschreiben.

Bei dynamischer Typbindung besteht (ohnehin) die Notwendigkeit, Objekte/Werte mit Typdeskriptoren zu attributieren.

Bei dynamischer Namensbindung enthält die A-Liste bzw. der Stapel der Bezeichnertabelle Typdeskriptoren.

- Garbage Collection in der beschriebenen Form ist in solchen Sprachen ohne weitere Deskriptoren realisierbar.

Bei Sprachen mit statischer Namens-/Typbindung müssten speziell für die Zwecke der Garbage Collection Typdeskriptoren für die Laufzeit erzeugt werden.

- aus Effizienzgründen fordern solche Sprachen meist keine Garbage Collection
- und aus gleichem Grund wird selten eine vollständige Kompaktifizierung unterstützt

⇒ De facto besteht bei Verwendung der Halde die Gefahr des Speicherüberlaufs für langlaufende Programme.

Nutzerproblem bei der Garbage Collection

Definition von "Garbage" beim Benutzer: "Objekte, die das Programm in der Zukunft nicht mehr braucht."

Definition von "Garbage" im Garbage Collector: "Speicher, der vom Programm nicht mehr adressiert werden kann."

Differenz der Definitionen: Alle Objekte, die das Programm zwar nicht mehr braucht, die aber immer noch über Referenz(ketten) erreichbar sind. Diese werden nicht eingesammelt.

Konsequenz für die Programmierung: **Alle nicht mehr benötigten (globalen) Zeiger/Referenzen müssen auf null gesetzt werden**, denn nur dann wird das nicht mehr Benötigte auch unerreichbar und eingesammelt.

Die Realität: Häufig verhindert ein einzelner vergessener Zeiger das Einsammeln riesiger nicht mehr benötigter Graphstrukturen.

4.5 Speichermodelle in Sprachen

Hardware bietet für die Speicherung von Variablen mehrere Möglichkeiten an:

- Register (wenige, aber sehr schneller Zugriff); Zugriffszeit: "unmittelbar"
- Cache (auf modernen Rechnern in mehreren Stufen); Zugriffszeit: 2-10 Zyklen
- Hauptspeicher; Zugriffszeit: 20-200 Zyklen

Wo sind die Variablen X und Y gespeichert?

X := 7;

Y := true;

Warum kümmert uns das?

- (Geschwindigkeit; natürlich versucht der Compiler, diese zu maximieren)
- Beobachtung "von außen", z. B. im Debugger
- Korrektheit (und generelle Möglichkeit) paralleler Zugriffe, speziell auch auf Multi-Core-Rechnern -> "Cache-Kohärenz"

Die Konsequenzen ("Schrödingers Katze")

Fall 1:

`X := 7;`

`Y := true;`

`... if Y then Assert(X=7); -- selbstverständlich`

Fall 2:

`X := 7;`

`Y := true;`

`... >> debugger breakpoint here`

`dbx: if Y then Assert(X=7); -- nicht garantiert`

Fall 3: thread1 :

`X := 7;`

`Y := true;`

thread2:

`if Y then Assert(X=7); -- garantiert ?`

Speichermodelle ("memory models") der Sprachen
geben die Antwort (die meistens negativ ausfällt)

.. und die Antworten aus dem Memory Model

```
thread1 :                thread2:
    X := 7;                if Y then Assert(X=7); -- garantiert ?
    Y := true;
```

Eine entsprechende Garantie würde erfordern, dass jegliches Instruktions-Scheduling durch Hardware oder Compiler, das mehrere globale Variablen betrifft, unterbleibt, und dass alle Zugriffe auf globale Variablen mindestens über Caches, bei nicht-kohärenten Caches auf Multicore-Architekturen sogar nur über den Hauptspeicher abgewickelt werden. Die allgemeine Verlangsamung ist i. A. nicht akzeptabel.

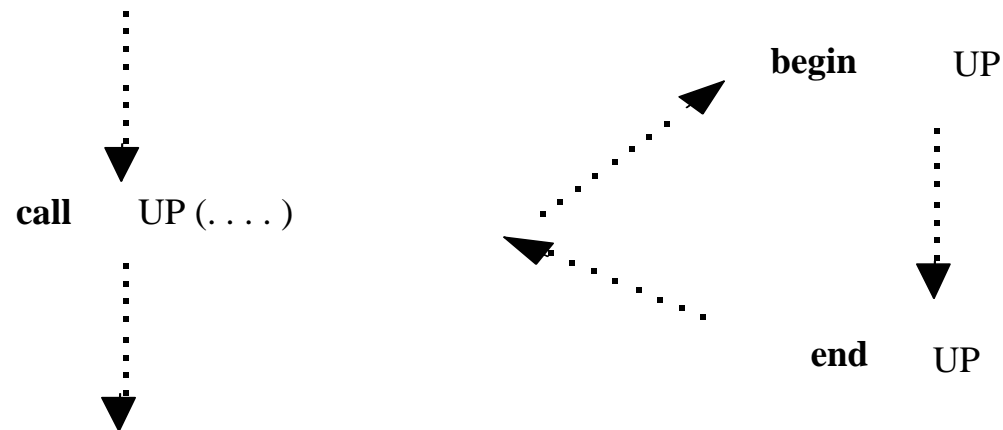
Die Antwort ist daher in fast allen Sprachen "NEIN": Die Zuweisung an X muss nicht vor der Zuweisung an Y abgeschlossen sein.

Garantien erfordern Benutzer-Spezifikation der Intention:

- "volatile" Variablen in Ada und C/C++ erzwingen den Hauptspeicherzugriff,
- eine zwischenzeitliche Synchronisation der Threads in Java und Ada erzwingt die Garantie für thread2,
- (für C++ ist gerade der neue Standard inkl. Memory Model entstanden)
- usw.

4.6 Semantik der Unterprogramme

- Speicherverwaltung bereits besprochen
- Kontrollfluss



- Als Parameter können im Allgemeinen Datenobjekte und -werte, evtl. auch Unterprogramme verwendet werden.

Formale und aktuelle Parameter

Format eines UPs:

```
subprogram    UP (F1, . . . . . Fn)
```

```
                ⋮
```



formale
Parameter

```
end UP;
```

Aufruf eines UPs:

```
call UP (A1, . . . . . , An)
```



aktuelle
Parameter

Die Zuordnung von aktuellen zu formalen Parametern erfolgt durch die Position oder durch Bindung an die Namen der formalen Parameter:

```
PROCEDURE SORT (input, output: DataType;  
                order: direction);  
  
BEGIN ... END;
```

- Beispiel für (die übliche) Zuordnung über Position:

```
SORT (Feld1, Feld2, ascending);
```

- Beispiel für Zuordnung über die Namen der formalen Parameter (Verwendung in Ada, Modula-3)

```
SORT (input => Feld1, output => Feld2,  
      order => ascending);
```

Vorteile der Assoziation über Namen:

- bessere Lesbarkeit, denn die Reihenfolge der Parameter ist oft relativ willkürlich; der Name kann dagegen aussagekräftig gewählt werden
- Voreinstellung von Parametern

Beispiel:

```
procedure SORT (input: in Datatype;  
                output: out DataType;  
                order: in Direction:= ascending) ;  
  
...  
SORT (input => Feld1, output => Feld2) ;  
-- 'order' wird implizit auf den Wert 'ascending' gesetzt
```


4.6.1 Mechanismen zur Parameterübergabe

Übergabemechanismen für konventionelle Programmiersprachen:

- call-by-copy
- call-by-reference
- call-by-name

Funktionale Programmiersprachen kennen noch call-by-need (auch "call-by-demand", "lazy evaluation" genannt)

Call-by-copy

Formale Parameter werden als lokale Variablen betrachtet.

Spezialfälle:

- **call-by-value:** Beim Eintritt wird der aktuelle Parameter ausgewertet und in den formalen Parameter kopiert
- ⇒ keine Änderung aktueller Parameter durch Zuweisung an formale Parameter

Verwendung z. B. in C, C++, Pascal, Modula-2

- **call-by-result:** Beim Austritt wird der Wert des formalen Parameters in den aktuellen Parameter kopiert.

Verwendung in Ada (OUT-Parameter)

- **call-by-value/result:** Kombination der vorangegangenen Mechanismen

Verwendung in einigen Fortran-Dialekten und in Ada (zur Implementierung der IN-OUT-Parameter)

Call-by-reference

- Adresse des aktuellen Parameters wird an formalen Parameter übergeben, dessen Verwendungen implizit dereferenzieren.
- Zugriff auf den formalen Parameter ist damit zugleich Zugriff auf den aktuellen Parameter.
- Aktuelle Parameter können im UP geändert werden.
- In vielen Sprachen sind nur Variablen als aktuelle Parameter erlaubt, z. B. in der PASCAL-Familie. Falls Ausdrücke erlaubt sind (z. B. PL/I), wird die Adresse einer Hilfsvariablen mit dem Resultat des Ausdrucks übergeben.
- Verwendung beispielsweise in Pascal, Modula-2, Ada

Beispiele für die Parametermechanismen

```
A, B: Integer := 1;  
procedure P(X: Integer) is  
begin  
  A := 2;  
  X := 5; -- illegal in einigen Sprachen für call-by-value Parameter  
  X := X * A;  
end P;
```

call-by-value:

P(B); Write(B); -- druckt 1

call-by-reference:

P(B); Write(B); -- druckt 10

P(A); Write(A); -- druckt 25 !!!! (ein sogenannter "aliasing"-Effekt)

"Aliasing" = zwei unterschiedliche Namen für die gleiche Entität

"Aliasing-Effekt" = eine Zuweisung via einen Namen beeinflusst den Wert, der unter anderem Namen gelesen wird

call-by-value/result:

P(B); Write(B); -- druckt 10

P(A); Write(A); -- druckt 10

Leider, wenn auch weniger fehlerträchtig:

procedure Q(X, Y: Integer) is

begin

 X:=7;

 Y:=5;

end Q;

Q(A,A); Write(A); -- druckt entweder 7 oder 5, abhängig von der Reihenfolge,
in der die Parameter zurückgeschrieben werden (üblicherweise unspezifiziert)

Call-by-name

Idee: Ausführung des Unterprogramm-Rumpfes als ob alle Vorkommnisse des formalen Parameters durch den aktuellen Parameter textuell ersetzt wären ..., d. h. der aktuelle Parameter(ausdruck) wird bei jedem Auftreten des formalen Parameters neu ausgewertet.

Beispiel:

```
procedure TAUSCHE (A,B : INTEGER) ;  
  var HILF : INTEGER;  
  begin  
    HILF := A;  
    A := B;  
    B := HILF;  
  end
```

Für TAUSCHE (X, Y) wird also ausgewertet

```
var HILF : INTEGER;  
HILF := X;  
X := Y;  
Y := HILF;
```

Probleme: Beim Aufruf `TAUSCHE (A[I] , HILF)` wird die Anweisungsliste

```
var HILF : INTEGER;  
HILF := A[I];  
A[I] := HILF;  
HILF := HILF; ⚡
```

ausgeführt, d. h. es findet kein Austausch statt.

Für einen Aufruf `TAUSCHE (I, A[I])` wird die Anweisungsliste

```
var HILF : INTEGER;  
HILF := I;  
I := A[I];  
A[I] := HILF;
```

ausgeführt. Für eingangs $I = 3$, $A[3] = 7$ (und $A[7] = 17$), ergibt sich als Resultat

$I = 7$, $A[3] = 7$, **$A[7] = 3$** ⚡⚡

Anm.: Es ist leicht zu zeigen, dass in einer Sprache mit dieser Parameterübergabe eine allgemeingültige 'Tausche'-Prozedur nicht geschrieben werden kann.

Modifizierte Idee (ALGOL-60-Regel): Die Bindungen der Namen im aktuellen Parameter werden an der Aufrufstelle etabliert; die Auswertung des aktuellen Parameters erfolgt für jedes Auftreten des formalen Parameters im Rumpf des Unterprogramms.

(Konsequenz: Die "Umgebung" des aktuellen Parameters muss bei Parameterübergabe vom Compiler ebenfalls übergeben werden, damit entsprechender Zugriff auf diese Variable möglich wird.)

Beispiel für die ALGOL-Regel:

```
procedure EINS;  
  var ANZAHL : INTEGER;  
  ....  
  procedure TAUSCHE (A, B : INTEGER);  
    var HILF : INTEGER;  
  begin  
    HILF := A;  
    A := B;  
    B := HILF;  
    ANZAHL := ANZAHL + 1  
  end;  
  ....
```

```
procedure ZWEI;  
  var ANZAHL, ZAHL : INTEGER;  
begin ....  
  TAUSCHE (ANZAHL, ZAHL);
```

```
end;
```

```
HILF := ANZAHL;  
ANZAHL := ZAHL;  
ZAHL := HILF;  
ANZAHL := ANZAHL + 1
```

ANZAHL, ZAHL aus ZWEI

ANZAHL aus EINS

Überraschungen bei call-by-name

```
I: Integer;
```

```
procedure F(Y: Integer) is
```

```
    I: Integer;
```

```
begin
```

```
    I := 7;
```

```
    Y := Y + 3;
```

```
end;
```

```
I := 5;
```

```
F(A[I]);
```

originale Regel

```
A[7] := A[7] + 3;
```

„ALGOL“ Regel

```
A[5] := A[5] + 3;
```

```
procedure P(Y: Integer) is
```

```
begin
```

```
    I := 7;
```

```
    Y := Y + 3;
```

```
end;
```

```
I := 5;
```

```
P(A[I]);
```

für beide Regeln

```
A[7] := A[7] + 3;
```

im Vergleich:

```
A[5] := A[5] + 3;
```

 für F und P bei call-by-reference

call-by-need (call-by-demand, lazy evaluation)

- Auswertung der Parameter wird zunächst verzögert bis zur ersten Verwendung im UP und ermöglicht so die Übergabe (potentiell) unendlicher Strukturen.
- Bei jeder weiteren Verwendung wird das Resultat dieser Auswertung benutzt.
- Verwendung in einigen funktionalen Sprachen, z. B. Miranda.

4.6.2 Parameterübergabe in einigen Programmiersprachen

PASCAL: call-by-value und call-by-reference

call-by-reference wird durch das Schlüsselwort VAR vereinbart:

```
procedure P (var Name: Typ) ;
```

wobei als aktueller Parameter nur Variablen, nicht jedoch Ausdrücke zulässig sind.

call-by-value wird deklariert durch

```
procedure P (Name: Typ) ;
```

Die Parameter verhalten sich innerhalb des Unterprogramms wie initialisierte, lokale Variablen. Der Wert des formalen Parameters kann verändert werden; diese Änderungen wirken sich aber nicht auf den Wert des aktuellen Parameters aus.

C: nur call-by-value

Allerdings ist call-by-reference explizit programmierbar:

```
void swap(x, y) {  
    int *x, *y;    .... }  
swap(&a, &b)
```

ferner:

```
int a[10];  
int sum(x)  
int x[];    // oder moderner int sum(int x[])  
{...}  
sum(a)
```

Wegen der definierten Äquivalenz "int a[] = int *a" de facto call-by-reference für Arrays (aber erstaunlicherweise nicht für struct).

Java: "nur Call-by-Value"

Allerdings sind die Variablen (von Klassen und Arrays) nicht die Instanzen selbst, sondern Referenzen auf die Instanzen, d.h., als aktuelle Parameter werden Referenzen "by-value" übergeben ...

```
int function F(X: T, Y: T)
{ ...
  X.a = 7;
  Y.a = 5;
  ...
}
```

```
A.a = 3;
```

```
F(A, Z);
```

```
Write(A.a); -- "7" ? oder vielleicht doch "5"?? (bei A==Z)
```

Ada:

Anstelle expliziter Mechanismen wird das abstrakte Verhalten der Parameter beschrieben:

- **IN-Parameter.** Der Wert des aktuellen Parameters kann innerhalb des Unterprogramms unter dem Namen des formalen Parameters angesprochen werden. Der Wert des formalen Parameters kann nicht verändert werden. Entspricht einer lokalen Konstanten. Syntax:
- **OUT-Parameter.** Das Unterprogramm gibt über eine Wertzuweisung an den formalen Parameter einen Wert an die als aktueller Parameter eingesetzte Variable zurück, kann aber nicht auf den Wert des aktuellen Parameters zugreifen (Ada 83; in Ada 95 und später ist das erlaubt). Syntax:

procedure P (Parametername: **in** Typname)

procedure P (Parametername: **out** Typname)

In vielen Präprozessor- oder Makro-basierten Sprachen:

Die Semantik der Makros wird durch textuelle Substitution erklärt und implementiert: Das Verhalten der Macro-Parameter ist daher wie bei "call-by-name".

Anmerkung: Das "Inlining", das Compiler für Unterprogrammaufrufe auf Anweisung oder auch implizit durchführen, muss die von der Sprache vorgegebenen Parameterübergabemechanismen beachten. Es ist daher im Allgemeinen NICHT eine textuelle Substitution. Ähnliches gilt für das Ausprägen generischer Templates.

- **IN OUT-Parameter.** Das Unterprogramm kann über den formalen Parameter auf den Wert des aktuellen Parameters zugreifen und diesen verändern. Syntax:

procedure P (Parametername : **in out** Typname)

Der Übergabemechanismus für zusammengesetzte Typen ist explizit nicht festgelegt und kann by-reference oder by-copy (**IN** = by-value, **OUT** = by-result, **IN OUT** = by value/result) sein. Programme, deren Verhalten durch dieses Implementierungsdetail beeinflusst werden, gelten explizit als fehlerhaft.

Für skalare und Zeiger-Typen ist call-by-copy, für tagged Typen (= OOP Klassen) und nicht kopierbare Typen ist call-by-reference vorgeschrieben.

4.6.3 Unterprogramme als Parameter

```
procedure P (...procedure FORMAL ...)  
begin ....
```

```
    call FORMAL (7)
```

```
    ....
```

```
end
```



AKTUELL wird
hier aufgerufen

Komplikation: Was passiert, wenn das
aktuelle UP ein lokales UP ist, das auf
lokale Daten eines umschließenden UP
zugreift ?

```
procedure AKTUELL(X: INTEGER) begin ... end
```

```
call P (...AKTUELL...)
```

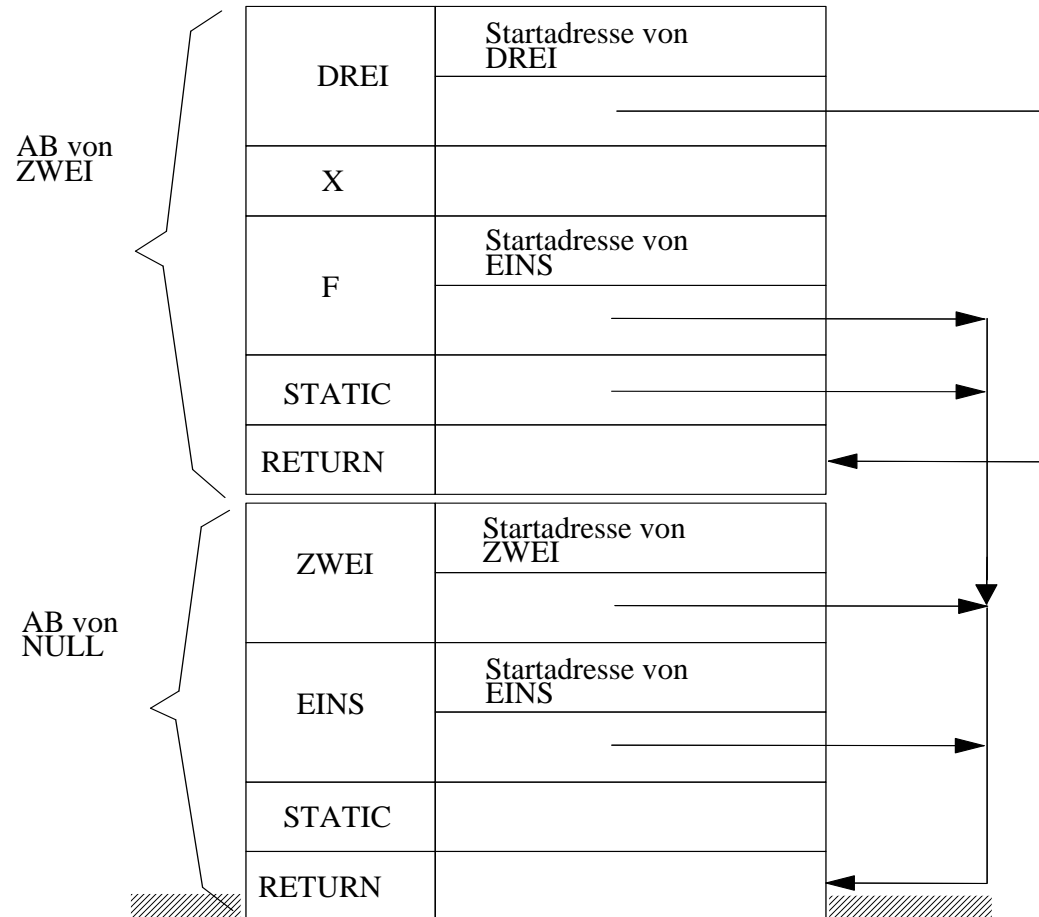


aktuelles UP

Beispiel:

```
procedure NULL;  
...procedure EINS;  
    ...  
end ; {EINS}  
  
procedure ZWEI (procedure F);  
    var X : INTEGER;  
  
    procedure DREI;  
        ... X := ...  
    end ; {DREI}  
  
begin  
    call F;  
    call ZWEI (DREI);  
end ; {ZWEI}  
begin {NULL}  
    ... call ZWEI (EINS) ...  
end {NULL}
```

Laufzeitkeller:



```

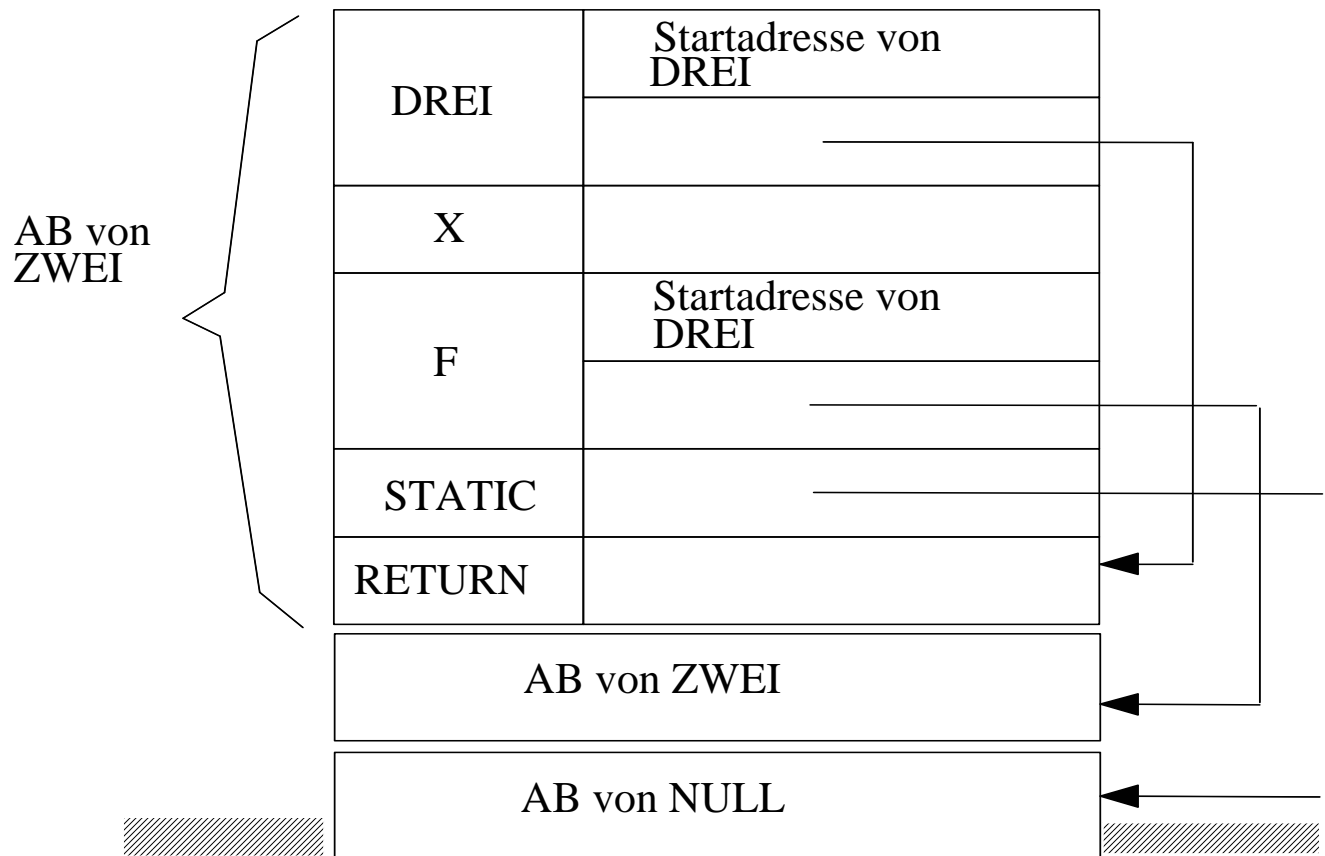
procedure NULL;
...procedure EINS;
    ....
    end ; {EINS}

procedure ZWEI (procedure F);
var X : INTEGER;

    procedure DREI;
        ... X := ...
    end ; {DREI}

begin
    call F;
    call ZWEI (DREI);
end ; {ZWEI}
begin {NULL}
... call ZWEI (EINS) ...
end {NULL}
    
```

1. Beim Aufruf **call ZWEI (EINS)**



2. Beim rekursiven Aufruf **call ZWEI (DREI)**

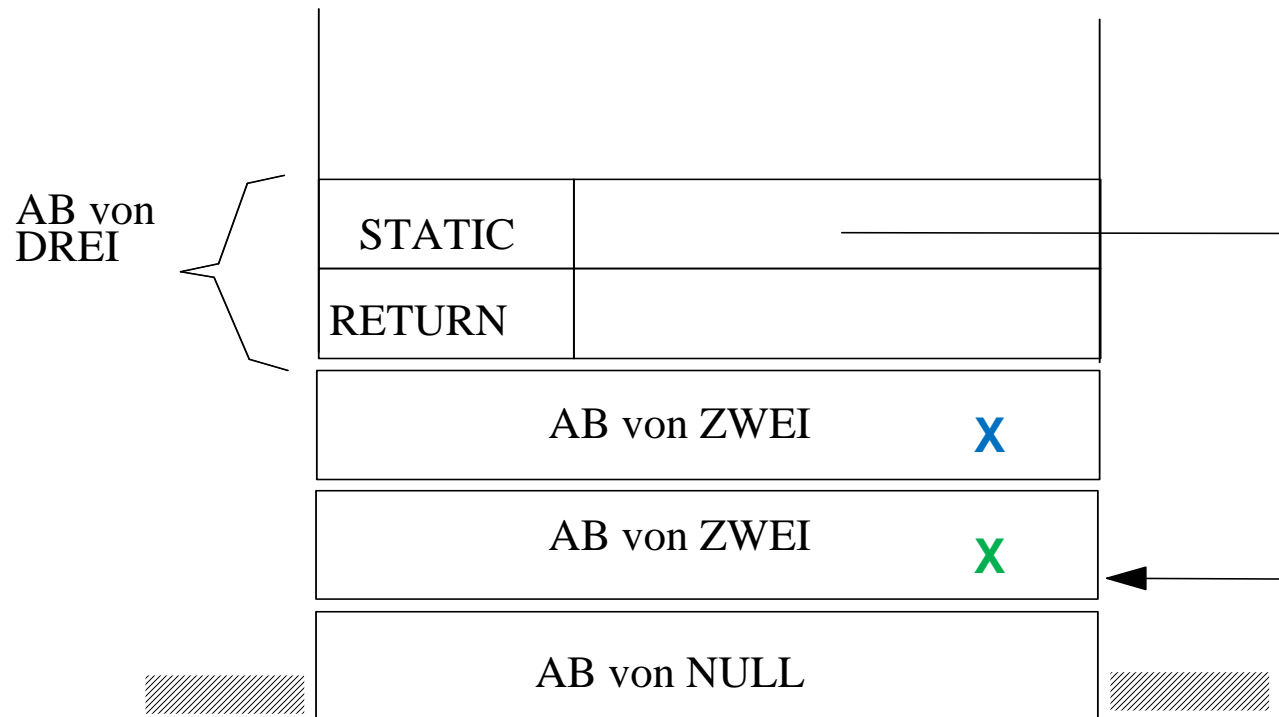
```

procedure NULL;
...procedure EINS;
    ...
    end ; {EINS}

procedure ZWEI
    (procedure F);
    var X : INTEGER;

    procedure DREI;
    ... X := ...
    end ; {DREI}

    begin
        call F;
        call ZWEI (DREI);
    end ; {ZWEI}
begin {NULL}
... call ZWEI (EINS) ...
end {NULL}
  
```



3. Beim Aufruf von **call F** für F = DREI

```

procedure NULL;
...procedure EINS;
    ...
    end ; {EINS}

```

```

procedure ZWEI
    (procedure F);
    var X : INTEGER;

    procedure DREI;
        ... X := ...
    end ; {DREI}

```

```

begin
    call F; ... X := 7;
    call ZWEI (DREI);
end ; {ZWEI}
begin {NULL}
... call ZWEI (EINS) ...
end {NULL}

```

4.6.4 Prozedurvariablen und Zeiger auf Prozeduren

Wichtige Beobachtung: Bei Prozeduren als Parametern ist immer garantiert, dass die lokale Umgebung der als Parameter übergebenen Prozedur auf dem Stapel existiert.

(Übung: Warum gilt diese Beobachtung generell?)

Bei Prozedurvariablen, die als Werte lokal deklarierte Prozeduren erhalten können, gilt diese Beobachtung nicht!

Zeiger auf lokale Prozeduren verursachen ein analoges Problem.

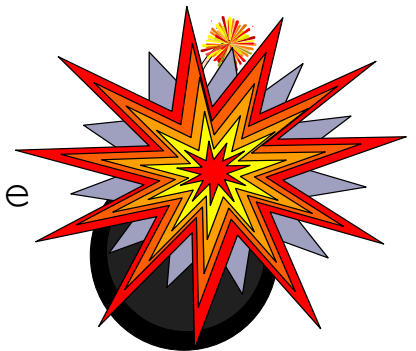
=> Gefahr des Zugriffs auf nicht mehr lebende lokale Daten !!!

Beispiel:

```
var procedure P();
```

```
procedure X() is
    Local_Var: int;
    procedure Local() is
    begin
        ... Local_Var ...
    end Local;
begin
    .. P := Local;
end X;
```

```
...
X();
Y(); -- beliebige Funktion
P(); -- ruft Local auf, das auf das nicht existente
      -- Local_Var zugreift (wo Y ein Passwd speicherte)
```



Lösungen:

- "Keine lokalen Prozeduren"
- Restriktionen bei der Prozedur- bzw. Zeigerzuweisung (Gültigkeitsbereich der Prozedurvariablen ist Teil des Gültigkeitsbereichs der zugewiesenen Prozedur).
Diese Restriktion ist nicht statisch prüfbar, wenn gleichzeitig auch Prozeduren bzw. Prozedurzeiger als Parameter zugelassen sind.
- **"Closures"** (→ Funktionale Sprachen)
Abwendung vom Stapelmodell hin zu der Idee, dass jede Prozedur neben ihren Parametern mit einer Kopie aller von ihr referenzierten freien Variablen ausgestattet wird.

Achtung: Analoge Probleme existieren, wenn designierte Routinen Zugriff auf andere Daten mit beschränkter Lebensdauer haben. Stichwort z. B.: "inner classes".

Kapitel 5

Bindungskonzepte für Programmiersprachen

5 Bindungskonzepte für PSen

- Jeder Bestandteil einer Programmiersprache,

z. B. Konstanten

 Variablen

 Typen

 Unterprogramme

besitzt eine Menge von Eigenschaften, so genannte **Attribute**.

- Die Festlegung der Attribute eines Sprachbestandteils heißt **Bindung**.
- Bindungen werden festgehalten in **Deskriptoren** („Bindungsbeschreibungen“).

- Bindungen werden festgelegt entweder
 - vor der Ausführung von Programmen (statische Bindung)
 - während der Ausführung von Programmen (dynamische Bindung)
- PSen unterscheiden sich darin,
 - welche Bindungen statisch oder dynamisch erfolgen
 - welchen semantischen Konzepten die Attribute zugeordnet werden

Einige typische Attribute (z. B. von „Variablen“):

- | | |
|-------------------------------|--------------|
| • Wert | (value) |
| • Name | (name) |
| • Gültigkeitsbereich | (scope) |
| • Adresse des Speicherplatzes | (address) |
| • Typ | (type) |
| • Lebensdauer | (lifetime) |
| • Veränderbarkeit | (mutability) |

Die folgende Diskussion von Bindungen orientiert sich an einem von vielen semantischen Modellen, dem “Objektmodell”.

Informell: “Objekt” = ein Oberbegriff für alle “Dinge”, die Werte besitzen und auf denen Operationen durchgeführt werden.

(Anm.: Die objektorientierte Programmierung prägt diesen Begriff in einer spezielleren Weise aus.)

“Passives” Objekt: ein Datenobjekt, auf dem Operationen ausgeführt werden.

“Aktives” Objekt: ein Objekt, das Operationen veranlasst.

5.1 Die Wertebindung

Objekt ↔ Wert

↔ⁿ variables Objekt (*Variable*)
→ Wert des Objekts kann zur Ausführungszeit geändert werden.

↔¹ konstantes Objekt
→ Wert des Objekts kann nicht geändert werden.
Initialwert ist erforderlich.

Initialwerte:

- durch explizite Angabe
- implizit durch Sprache vorgeschrieben
- undefinierter Wert
 - abfragbar
 - nicht abfragbar
 - "definiert undefiniert", d. h. Semantik von Operationen mit undefinierten Werten ist wohl-definiert

Änderung: durch Zuweisung

Eine Reihe von PSen (-> insb. funktionale PSen) lässt nur eine \leftrightarrow^1 -Bindung zu.

5.2 Die Speicherplatz-/Adressbindung

Objekt \leftrightarrow Speicherplatz

(~ Deskriptor für Wertebindung)

typisch: \leftrightarrow^1

- etabliert durch Speicherzuweisung (Allokation)
 - feste Zuweisung vor Programmausführung
 - implizit durch Deklaration oder Aktivierung des die Deklaration umfassenden Kontrollkonstrukts
 - explizit (\rightarrow Haldenallokation)
- aufgehoben durch Deallokation
 - implizit durch Verlassen des die Deklaration umfassenden Kontrollkonstrukts
 - explizit (\rightarrow FREE, DISPOSE)

Lebensdauer des Objekts =

Zeitraum der Ausführung zwischen Allokation und Deallokation

Anmerkungen:

Bei der Realisierung dieser Bindung kann aus diversen Optimierungsgründen auch \leftrightarrow^n entstehen (-> Haldenkompaktifizierung, -> Registervergabe), aber die logische Sicht bleibt: \leftrightarrow^1 .

Alternative (pragmatische) Sichtweise des Compilers:

Lebensdauer des Objekts = Zeitraum zwischen Allokation und letzter Verwendung des Objekts (damit danach der Speicherplatz wiederverwendet werden kann). Die beiden Versionen sind ebenfalls aus logischer Sicht nicht unterscheidbar.

5.3 Die Namensbindung

Objekt \leftrightarrow Name

typisch: \leftrightarrow^n

- expliziter Name
 - etabliert durch Deklaration, by-ref-Parameter usw.
- anonymer Name (*“Referenz”*)
 - explizite Allokation
 - Adressoperator
 - (als Wert anderer Objekte speicherbar)

Wenn $n > 1$, sprechen wir von ***Aliasing***.

Gültigkeitsbereich des expliziten Namens eines Objekts:

Durch Sprachregeln festgelegte Bereiche des Programms, in denen die Bezeichnung des Objekts durch den Namen gültig ist.

Wir unterscheiden **statische und dynamische Namensbindung**, je nachdem, ob der Gültigkeitsbereich durch die textuelle Aufschreibung oder durch die Ausführungsreihenfolge von Kontrollkonstrukten und deren Deklarationen definiert ist.

Die eigentliche Frage gilt dem durch expliziten Namen im Programmtext identifizierten Objekt. Dies ist im Allgemeinen wegen $\text{Name} \leftrightarrow^n \text{Objekt}$ nicht trivial zu beantworten.

==> Sichtbarkeitsregeln, um
 “gebundener” Name \leftrightarrow^1 Objekt
für jedes Vorkommen des Namens zu erhalten.

- **Statische Namensbindung: Gültigkeitsbereich ergibt sich aus der Programmaufschreibung. Typisch:**
 - beginnt bei Deklaration
 - endet mit umgebendem Kontrollkonstrukt
 - Die textuelle Schachtelung von Kontrollkonstrukten mit Deklarationen gleicher Namen führt zur Verdeckung von Namen ("Sichtbarkeitsregeln"), genauer ...
 - "Homographen" = zwei Deklarationen für den gleichen Bezeichner (mit gleicher Signatur -> siehe Overloading später)
 - Homographen in einem umschließenden Gültigkeitsbereich werden von der Deklaration des geschachtelten Gültigkeitsbereichs für die direkte Sichtbarkeit verdeckt.

Beispiel:

```
MODULE sichtbar;  
  CONST x = 10;  
  VAR i: INTEGER;  
  
  PROCEDURE Ebene1;  
    VAR x: REAL;  
  
    PROCEDURE Ebene2;  
      VAR x: CHAR;  
      BEGIN (* Ebene2 *)  
        x:= 'A'; (* x ist Variable vom Typ CHAR. *)  
        i:= 12;  
      END Ebene2;  
  
      BEGIN (* Ebene1 *)  
        x:= 3.141; (* x ist Variable vom Typ REAL. *)  
        i:= 999;  
      END Ebene1;  
  
      BEGIN (* sichtbar *)  
        i:= x; (* x ist Konstante vom Typ INTEGER. *)  
      END sichtbar.
```

- **Dynamische Namensbindung: Gültigkeitsbereich ergibt sich aus der Ausführungsreihenfolge**
 - beginnt bei Ausführung der Deklaration
 - endet mit der Ausführung des umgebenden Kontrollkonstrukts
 - Schachtelung der Ausführung von Kontrollkonstrukten führt zur Verdeckung von Homographen

Beispiel:

```
X: int;
procedure A is
begin
    ...X...      -- siehe unten
end A;

procedure B is
  X: real;
begin
    ...X...      -- immer: X: real
    A;           -- (1)
end B;

A;              -- (2)
B;
```

Bedeutung von X in A:

- bei statischer Bindung: immer: X: int
- bei dynamischer Bindung:
 - im Aufruf (1) von A: X: real
 - im Aufruf (2) von A: X: int

X wird als *freie Variable von A* bezeichnet.

5.3.2 Statische oder dynamische Namensbindung?

Für statische Namensbindung:

- Lesbarkeit
- Prüfbarkeit
- Schutz lokaler Variablen
- Effizienz bei übersetztem Code
- (Voraussetzung für statische Typprüfung)

Für dynamische Namensbindung:

- Flexibilität der Unterprogramm-Kommunikation (?)
- Einfachheit der Implementierung in reinen Interpretern
- triviale Erfüllung der folgenden Sicherheitsforderung ...

Sicherheitsforderung:

Die Lebensdauer eines Objekts muss die Ausführung jedes Konstrukts im Gültigkeitsbereich jedes Namens des Objekts umschließen.

Informell: *Das benannte Objekt muss auch wirklich existieren.*

Problem: “Gültigkeitsbereich” von Referenzen!

=> *dangling-references*-Syndrom

Beispiele für *dangling references*:

1. durch Adressoperator

```
int *p;  
int demo()  
{   int i;  
    p := &i;  -- & gibt die Adresse des Arguments zurück  
...}  
demo();  
*p := 10;    -- Zuweisung auf nicht mehr existentes i
```

Lösung: Referenz auf Objekt O_1 kann nur dann als Wert von Objekt O_2 gespeichert werden, wenn gilt:
Lebensdauer(O_1) umfasst Lebensdauer(O_2)

2. durch explizite Deallokation

```
p := new(...);      -- Allokation  
q := p;  
DISPOSE(p);         -- Deallokation  
q^ := 10;           -- außerhalb der Lebensdauer des Objekts
```

Lösung: ?? (siehe Haldenverwaltung)

5.3.1 „Overloading“ (Überladbarkeit)

= Sichtbarkeit mehrerer Definitionen von Unterprogrammen mit gleichem Namen.

Beobachtung: Operatoren sind in fast allen PSen überladen.

Das Überladen von benutzerdefinierten Unterprogrammen wird erreicht, indem die Verdeckungsregeln der Sprache nicht nur den Namen des Unterprogramms, sondern auch seine Signatur (*Profil*) mit einbeziehen:

Nur gleichnamige Unterprogramme mit gleicher Signatur verdecken sich.

Fragen: Worin besteht die Signatur ?
Wie wird bei Aufrufen die Bindung festgestellt ?

Signatur: typisch: Parametertypen und evtl. Resultatstyp

Bindung: a) durch die (statisch) festgestellten Typen der aktuellen
Parameter (C++, Ada, Java)
b) evtl. auch durch den im Kontext „erwarteten“ Typ des
Resultats (Ada, nicht in C++ oder Java)
wird das „richtige“ Unterprogramm ausgewählt.

Anm: Interaktion mit dem Redefinitionsbegriff , z. B. in OOP.

b) macht die Analyse kontextabhängig und ungleich schwieriger, da eine Interaktion zwischen a) und b) besteht:

```
procedure P(X: TypeOne);  
procedure P(X: Real);  
function F(X: Boolean) return TypeOne;  
function F(X: Boolean) return Real;  
X: TypeOne := F(True);  
    -- eindeutig in Ada; mehrdeutig und daher illegal in C++  
P(F(True)); -- mehrdeutig
```

5.3.2 Namensräume und Sichtbarkeit

"Namensraum" = Gruppierung von Deklarationen

"direkte Sichtbarkeit" = Benennbarkeit durch den einfachen Namen
("Bezeichner") ohne weitere Qualifikation

"qualifizierte Sichtbarkeit" = Benennbarkeit durch einen mit dem Namensraum
qualifizierten einfachen Namen

genestete (hierarchische) Namensräume

es gelten Verdeckungsregeln für die Sichtbarkeit (siehe "Namensbindung")

getypte (getrennte) Namensräume

syntaktischer Kontext entscheidet darüber, welcher Namensraum für die
Sichtbarkeitsregeln gilt

nicht-hierarchische Namensräume

- typisch für Paket-/Modulstrukturen
- Komposition durch explizite Importe
- Regeln benötigt für importierte Homographen
 - importierte Homographen sind illegal (aber wie kann das der Benutzer verhindern?)
 - Verwendung der zu Homographen gehörenden Namen ist illegal (weil mehrdeutig)
 - Homographen werden nicht importiert (d. h., löschen sich gegenseitig aus)
 - Präferenzregeln, einen der Homographen vorzuziehen (-> böartige "*Beaujolais-Effekte*", falls ein veränderbarer Namensraum präferiert wird)

Beaujolaïs-Effekt *

= Das Hinzufügen einer zusätzlichen Deklaration verändert still-schweigend die Laufzeit-Semantik der Implementierung eines anderen bislang bereits übersetzbaren Moduls.

Beispiel:

```
package Pkg1 is  
    procedure P;  
    -- wird später hinzugefügt  
end Pkg;
```

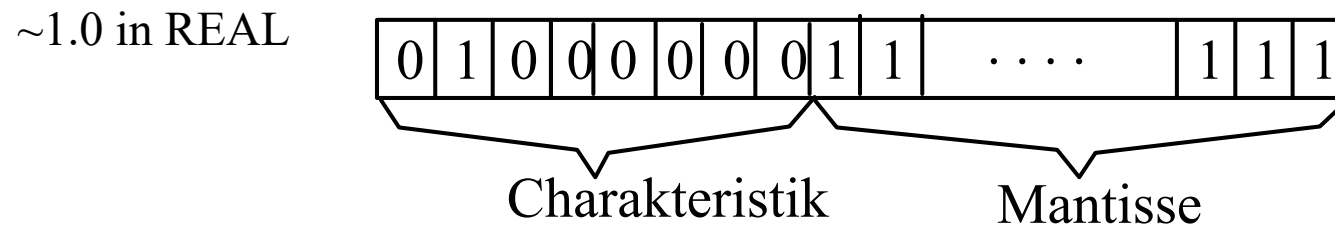
```
package Pkg2 is  
    procedure P;  
end Pkg;
```

```
with Pkg1, Pkg2;  
use Pkg1, Pkg2;  
procedure demo is  
    P; -- sollte nicht stillschweigend an Pkg1.P binden  
end Pkg;
```

** benannt nach einer Wette, dass Ada keine derartigen Effekte hat. Für Ada83 (durch ein abstruses Beispiel) verloren, für Ada95 bislang noch nicht eingefordert.*

5.4 Die Typbindung

Werte sind abgespeicherte Bitmuster, die einer Interpretationsvorschrift bedürfen. Beispiel:



aber $> 2^{23}$ in INTEGER

Festlegung der Interpretation durch Typ des Werts mittels entsprechender Bindung:

Wert \leftrightarrow Typ

Diese Bindung kann (zum Großteil) subsumiert werden durch eine Typbindung von Objekten:

Objekt \leftrightarrow ¹ Typ

\leftrightarrow ¹ Typ des Objekts wird festgelegt bei

- Deklaration oder expliziter Allokation
→ *strenge Typenbindung*
- erster Zuweisung

\leftrightarrow ⁰ “typlose” Objekte

zwangsweise dann aber: Wert \leftrightarrow ¹Typ, d. h. jeder Wert muss einen festen Typ zugeordnet bekommen.

\leftrightarrow ⁿ mit Wert \leftrightarrow ¹ Typ kombiniert.

- *union* – Typen
- *class* – Variablen

Im ersten und letzten obigen Fall muss bei der Wertänderung eine entsprechende Typprüfung erfolgen.

Eine weitere (häufige Realisierungs-)Alternative:

gebundener Name \leftrightarrow^1 Typ

d. h., Typ des Objekts aus Deklaration oder Allokation bekannt

Komplikation: wegen Aliasing ...

Objekt \leftrightarrow^n gebundener Name \leftrightarrow^1 Typ

ist es zunächst möglich, dass Objekt (und Wert) dadurch mit verschiedenen Typen assoziiert wird.

Lösungen in PSen durch Regeln:

- entweder Verbot unterschiedlicher Typen bei Aliasing
- oder Garantie, dass Typ-Eigenschaften des Objekts eine Obermenge der Eigenschaften der jeweiligen mit den Namen des Objekts assoziierten Typen sind (“views”).

Weitere Differenzierung:

Einige (statische) Typ-Eigenschaften werden wie oben behandelt. Andere Typ-Eigenschaften bleiben mit den Werten oder Objekten assoziiert (-> Typ/Subtyp-Unterscheidung).

Bindung von Variablen an einen Datentyp kann erfolgen:

- statisch vor der Laufzeit
- semi-dynamisch bei der Erzeugung
- dynamisch während der Laufzeit.

5.5 Änderbarkeit

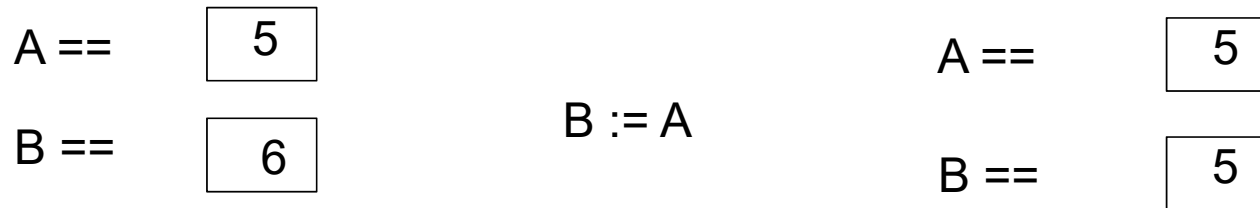
5.5.1 Der Variablenbegriff

... ist eng verknüpft mit der Zuweisungssemantik. Zwei Alternativen:

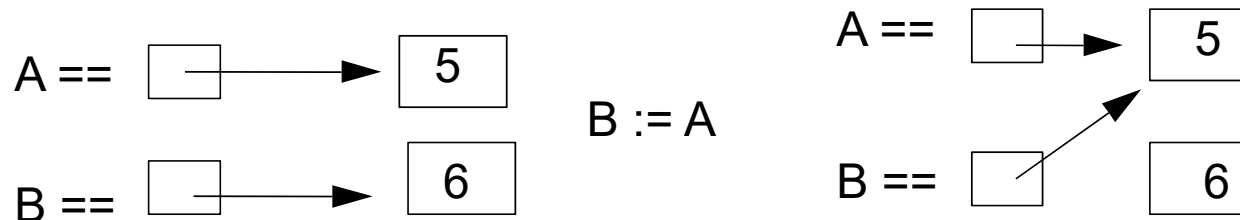
- (1) Variable = variables Objekt
Zuweisung ändert den Wert des Objekts
→ *Wertesemantik*
- (2) Variable = Referenz auf Objekt
Zuweisung an Variable ersetzt die Referenz
→ *Referenzsemantik*
→ *Halden-orientierte Sprachen*

(1) und (2) sind im (abstrakten) Programmverhalten nicht unterscheidbar, wenn (2) auf konstante Objekte beschränkt ist.

Wertesemantik:



Referenzsemantik:



aber in der Folge: assign(A, 500); assign(B, 7); eq(A, 7)? Ja!

5.5.2 Der Konstantenbegriff

Beispiele:

```
X: constant Integer := N;  
    -- Konstante mit dynamisch berechnetem Wert  
X: constant Integer := 7;  
    -- Konstante mit statisch bekanntem Wert  
Pi: constant := 3.14;  
    -- benannter Wert
```

Hier: Deklaration **konstanter Objekte**

Interaktionen:

- als call-by-ref-Parameter erlaubt?
- Adress-Operator möglich?
- welchen Typ haben benannte Werte?

Aber auch: **konstante Views**

Beispiel: (Parameter von Funktionen)

```
function f (X: in Big_Array) return XYZ;
```

X gilt innerhalb der Funktion als Konstante, kann aber Variablen als aktuelle Parameter haben, d. h. Änderbarkeit ist hier mit dem Namen, nicht aber zwangsläufig mit dem Objekt verbunden.

Beispiel: (Konstantheit im Bereich der Referenzen):

(Ada Syntax)

X: constant access T := new T;

X: access constant T := new T;

X: constant access constant T := new T;

Y: access T := X; -- legal?

Was genau ist konstant?

(C Syntax)

const T * x = malloc(..);

T * const x = malloc(..);

const T * const x = malloc(..)

T * y = x /* legal?

T obj;

x = &obj; /* legal?

Kapitel 6

Typsysteme

6. Datentypen

Motivation:

1) Auf unterster Ebene werden alle Werte durch Bitmuster dargestellt. Für einen Ausdruck „ $A+B$ “ muss der Compiler oder Interpreter entscheiden, ob z. B. eine Integer-Addition oder eine Gleitkomma-Addition durchgeführt werden soll.

⇒ **Notwendigkeit der Typisierung von Werten**

Sind die Werte von A und B so typisiert, dass eine Addition keine sinnvolle Semantik besitzt, soll ein Fehler diagnostiziert werden (insbesondere zur Übersetzungszeit).

2) Es wird in vielen PSen als wünschenswert angesehen, dass Variablen (m. E.) nur Werte eines Typs besitzen können und entsprechend fehlerhafte Zuweisungen erkannt werden.

⇒ **Wunsch nach Typisierung von Variablen u. ä.**

Eine solche Typisierung von Variablen kann ggf. die Typisierung von Werten subsumieren.

Prinzip: Datentyp legt fest

- Menge von Werten (und ihre Darstellung)
- Menge von Operationen, die auf die Werte angewendet werden können

6.1 Typbindung

Die Typbindung von Variablen kann

- statisch vor der Laufzeit (durch Deklaration, z.B. in Ada, Java, C++ ...),

```
X: Boolean;  
...   X := "abc";      -- illegal  
...   X := 7;          -- illegal
```
- semi-dynamisch bei der Erzeugung (durch 1. Zuweisung, z.B. in APL),

```
X := "abc";            -- X ist nun vom Typ String  
...   Y := Y * X;      -- i.A. zur Laufzeit illegal  
X := 7;                -- zur Laufzeit illegal
```
- dynamisch während der Laufzeit (durch jede Zuweisung, z. B. in Python, Ruby)

```
X := [1, 2, 3];        -- X ist nun eine Liste  
...   Y := X * Y;      -- i.A. zur Laufzeit illegal  
X := 7;                -- X ist nun von integer Typ  
...   Y := Y * X;      -- legal falls Y von integer (oder ...) Typ  
erfolgen. (Beachte: unterschiedliche Zuweisungen in Verzweigungen möglich!)
```

Im 2. und 3. Fall ist eine werte-bezogene Laufzeit-Darstellung der Typkennung des Werts der Variablen nötig.

Vorteile der statischen Typenbindung

- Schutz von Variablen gegen unerlaubte Operationen
- Bestimmte Klassen von Programmierfehlern können vom Compiler erkannt werden
- bessere Lesbarkeit des Programms
- Frühzeitige Bestimmung geeigneten Codes für überladene Operatoren:

A, B:	INTEGER	A+B	INTEGER Addition
C, D:	REAL	C+D	REAL Addition
E, F:	packed array[1..20] of CHAR	E+F	Konkatenation von Zeichenketten

- m. E. Vermeidung von Typdeskriptoren zur Laufzeit

Vorteile der dynamischen Typenbindung

- Flexibilität
- weniger Schreibarbeit, da Deklarationen entfallen können
- sowie . . .

Vorteile von Typdeskriptoren zur Laufzeit

(auch bei statischer Typbindung möglich)

- Garbage Collection möglich
- automatische Auswahl der „richtigen“ Operationen zur Laufzeit möglich
 - Polymorphie in der objektorientierten Programmierung
 - undurchbrechbarer Typenschutz möglich
- Laufzeitprüfungen
 - diskriminierte Vereinigungstypen, endliche Arrays

Anmerkung: Auch bei statischer Typbindung wird ggf. ein Teil der Typbeschreibung zur Laufzeit benötigt, um Laufzeitprüfungen zu ermöglichen:

⇒ z. B. „Dope“-Informationen für Array-Objekte

6.2 Standarddatentypen

- jede Programmiersprache stellt einen Satz vordefinierter Datentypen bereit
- Standarddatentypen abstrahieren von den Hardware-Eigenschaften von Computern, z. B.

INTEGER	↔	Zweierkomplement-Arithmetik
REAL	↔	Gleitkomma-Arithmetik
CHAR	↔	evtl. Bytes
BOOLEAN	↔	Bits

6.2.1 Vordefinierte Typen

Sprache stellt vordeklarierte Namen für „Standardtypen“ zur Verfügung.

Vorteile:

- Schreibbarkeit
- Lesbarkeit

Nachteile:

- sofern von der Sprache nicht festgelegt, ist der Wertebereich implementierungsabhängig
⇒ Portierungsprobleme

⇒ um das Schreiben portabler, maschinenunabhängiger Programme zu ermöglichen, sollte eine PS die Möglichkeit benutzerdefinierter (auch numerischer) Typen besitzen.

6.2.2 Benutzerdefinierte Datentypen

- Festlegung der Werte
- Festlegung der Operationen
 - implizit durch die Programmiersprache
 - explizit durch den Programmierer
- Abkapselung der Werte gegen unerwünschte Operationen
(→ Datenabstraktion, Sichtbarkeit)

Beispiele:

„(sub-)range“ Typen in Modula oder Ada

„opaque types“ in Modula, „private types“ in Ada

Viele PSen haben das explizite oder implizite Konzept von vordefinierten „Typkonstruktor-Funktionen“ (z. B. „array“, „ref“, „struct“), die als Resultat ein „Typobjekt“ mit konstantem Wert („**Typdefinition**“) liefern. Dieses Objekt kann dann nach den üblichen Sprachregeln benannt werden („**Typdeklaration**“).

Wozu benutzerdefinierte skalare Datentypen?

1. Portabilität:

```
type Feet is range -2_500 .. 100_000;  
type Meter is range -750 .. 30_000;
```

Der Ada-Compiler wählt die für den Wertebereich beste Darstellung (z.B., 16-bit Meter und 32-bit Feet, oder auch 64-bit für beide Typen, je nach Zielarchitektur)

2. Zuverlässigkeit:

```
distance: Feet;  
abstand: Meter;  
procedure calculate_Acceleration(To_Target: Meter);
```

```
distance := abstand; -- illegal, unterschiedliche Typen  
calculate_Acceleration(distance); -- genauso illegal
```

in Java: `int distance, meter;` -- keine Unterscheidung skalarer int Typen möglich
(Typtrennung allenfalls durch Wrapping in Instanzen von Feet und Meter Klassen)

in C/C++: `typedef int Feet; typedef short int Meter;`
`Feet distance; Meter: abstand;` -- gute Dokumentation
`distance = abstand;` -- leider legal, der typedef Name wird expandiert

6.3 Typkonversion

T_1, T_2 seien Datentypen, die zueinander (implizit oder explizit) konvertierbar sind.

Durch Konversion $T_1 \rightarrow T_2$ geht die Bitdarstellung der Werte aus T_1 in die Bitdarstellung der Werte aus T_2 über, z.B.

1 in INTEGER	0	0	0										0	0	1							
1.0 in REAL (nicht IEEE)	0	1	0	0	0	0	0	0	1	1										1	1	1
	Charakteristik 64											Mantisse											

Alle Programmiersprachen erlauben (sinnvolle) explizite Typkonversionen.
Manche Programmiersprachen sehen auch implizite Typkonversionen vor.

6.4 Typkompatibilität

Wir sprechen von **kompatiblen Typen**, wenn zwei Typen „äquivalent“ (s. u.) oder aber implizit konvertierbar sind.

Typische Sprachregeln:

- Zuweisung erfordert Kompatibilität
- Parameterübergabe erfordert Kompatibilität

6.5 Typäquivalenz

Wann sind zwei Datentypen T_1 und T_2 gleich (oder äquivalent)?

`T1=array[1..10] of INTEGER` `VEKTOR_1:array[1..100] of REAL`

`T2=array[0..9] of INTEGER` `VEKTOR_2:array[1..100] of REAL`

Ist Typ T1 gleich dem Typ T2?

Ist `VEKTOR_1:=VEKTOR_2` zulässig?

6.5.1 Namensäquivalenz

Die Typen zweier Variablen sind namensäquivalent, wenn

- ihre **Typnamen** die gleiche Bindung haben

`type T = ...` -- irgendein Typ

`var A : T; B : T`

oder (eventuell)

- sie in der gleichen Deklaration vereinbart werden

`var A, B : ...` -- irgendeine Typ**definition**

Der zweite Fall wird in verschiedenen PSen unterschiedlich behandelt:

- Typdefinitionen sind hier nicht erlaubt (nur Typnamen)
- die Variablen sind von gleichem, anonymem Typ
- die Variablen sind von verschiedenen anonymen Typen (z. B. in Ada)

6.5.2 Strukturäquivalenz

Die Typen zweier Variablen sind strukturäquivalent, wenn die Strukturen ihrer Typen übereinstimmen. Nachprüfung geschieht durch Ersetzung von Typnamen durch ihre Definition.

Prüfalgorithmus:

```
if s, t bezeichnen den gleichen Grundtyp then ok  
else if s = array (s1,s2) and t = array (t1,t2) then  
    teste Äquivalenz von s1, t1 und s2, t2  
else if s = s1 x s2 and t = t1 x t2 then  
    teste Äquivalenz von s1, t1 und s2, t2  
else if s = pointer (s1) and t = pointer (t1) then  
    teste Äquivalenz von s1 und t1  
else if s = s1 → s2 and t = t1 → t2 then  
    teste Äquivalenz von s1, t1 und s2, t2  
else  
    ..... weitere Konstrukturen .....
```

Probleme bei Strukturäquivalenz bereiten rekursive Typdefinitionen, z. B.

```
type ZEIGER = ↑KNOTEN;  
KNOTEN = record
```

...

K : ZEIGER

...

end

record

...

K : ↑**record**

...

K : ↑**record** -- hört nicht auf

...

Entfaltung von Knoten

Lösung dieses Problems unterschiedlich in den PSen, z. B.

- für Zeiger gilt Namensäquivalenz oder
- für Strukturen gilt Namensäquivalenz (z. B. in C, C++)

6.5.3 Vergleich von Namens- und Strukturäquivalenz

- Namensäquivalenz ist sehr strikt, z. B. ist für

```
var    R_1: RECORD a, b: REAL END;  
      R_2: RECORD c, d: REAL END;
```

die Zuweisung `R_1 := R_2;` unzulässig.
- Dies bewirkt aber eine zusätzliche Kontrolle, denn Datentypen mit gleicher Struktur können durchaus eine so verschiedene Semantik haben, dass eine Verknüpfung grundsätzlich als fehlerhaft zu betrachten ist, z. B.

```
type  Komplex      = RECORD Re, Im: REAL END;  
      Intervall    = RECORD Anfang, Ende: REAL END;  
var    k: Komplex; i : Intervall;
```

Hier wäre die Zuweisung `i:=k;` unsinnig, obwohl sie nach den Regeln der Strukturäquivalenz zulässig wäre und aus technischer Sicht (nahezu) ohne Probleme ausgeführt werden könnte.
- Vorteile der Namensäquivalenz sind die hohe Sicherheit, die erhöhte Lesbarkeit von Programmen und die einfachere Implementierung.
- Nachteile sind die geringere „Flexibilität“ (????) und evtl. Probleme mit der Spezifikation von Parametertypen (s. u.)

6.5.4 Namensäquivalenz und Parameterübergabe

```
A : array (1..10) of T;  
B : array (1..20) of T;
```

Wünschenswert wäre eine Prozedur, die die Arrayelemente sortiert (für beliebige Indexbereiche):

```
procedure sort (A: ??);
```

Probleme:

1. Form der Typangabe in „??“ (darf keinen Indexbereich festlegen)
2. falls „??“ ohne Indexangabe, wie kann die Implementierung von „sort“ die Länge der Arrays erfahren?
3. wie kann „??“ mit den Typen von A und B kompatibel sein?

Lösungen des Parameter-Typ-Problems:

- Konzept des „freien“ oder „unbeschränkten“ Arraytyps
- Attribute oder Standardfunktionen zur Abfrage der Grenzen
- spezielle Äquivalenzregeln oder Typ/Subtyp-Modell

Pascal: „conformant array“

```
procedure sort
  (var a: array [low .. high: Integer] of ...);
  for i := low to high do ...
```

Ada: „Typ/Subtyp-Modell“

```
type Vector is array (integer range <>) of ...;
subtype V_10 is Vector(1..10);
V1 : V_10; V2 : Vector(1..20);
procedure sort (A : in out Vector) is begin
  for i in A'First .. A'Last loop ...
```

Anm.: Ein sehr ähnliches Problem besteht für die Zieltypen von Zeigertypen: der Wunsch, einen Zeigertyp definieren zu können, der Array-Objekte unterschiedlicher Komponentenzahl referenzieren kann.

⇒ ähnliche Modelle von „freien“ Arraytypen oder Subtypen anwendbar

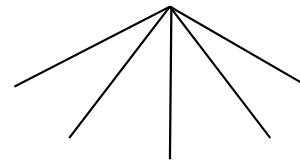
6.6 Typ/Subtyp-Modell (Ada)

Spezifikation gemeinsamer
Operationen und Komponenten

Klasse

statische (konservative)
Existenzprüfungen

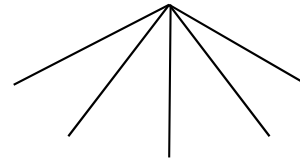
Operationen
grundsätzliche Darstellung



Typ_i

statische Prüfungen

Wertebereich
Indexbereich
Diskriminanten



Subtyp_j

dynamische Prüfungen

Objekte besitzen einen bestimmten Subtyp, der gleichzeitig auch den Typ festlegt. Subtypen sind *keine* Typen im Sinne der Typäquivalenz.

Anmerkung: `typedef` in C und C++ sind den Subtypen sehr ähnlich.

6.7 "Strong Typing", "Weak Typing", "Duck Typing"

Es gibt keine einheitliche Definition für diese Termini, sondern nur ein allgemeines Verständnis der jeweiligen Absicht.

Strong Typing: Möglichst viele Fehler sollen zur Übersetzungszeit erkannt werden. Typischerweise sind Zuweisungen zwischen unterschiedlichen Typen streng limitiert; es gibt kaum implizite Konversionen. Erlaubte Zuweisungen und Konversionen sind typsicher.

Weak Typing: Es gibt Typen. Es werden aber weniger Fehler erkannt. Der Compiler inferiert (nach bekannten Regeln), was der Benutzer wohl wollte. Implizite Konversionen werden liberal angewandt. Z. B. `5 & "abc"` ergibt `"5abc"`

Explizite Konversionen ("casts") können auch typunsicher sein.

Duck Typing: Einschränkungen orientieren sich nicht an einem Typsystem oder irgendeiner statischen Typisierung von Variablen. Stattdessen entscheidet die Existenz der gefragten Eigenschaft eines Objekts (wie auch immer erhalten) über die Zulässigkeit ihrer Verwendung.

6.6.1 "Duck Typing" (häufig in neueren Scripting-Sprachen)

Idee (für Methoden):

Die Existenz einer aufgerufenen Methode wird erst zur Laufzeit geprüft; die Herkunft der Methode ist irrelevant, d. h., ein Aufruf `obj.quack()` ist o.k., sofern die Methode `quack` für die derzeitige Klasse von `obj` definiert ist. Es ist dabei egal ob `obj` eine Ente, eine Person, ein Radio, eine Holztüre oder sonst etwas ist.

Ein Zusammenhang der jeweiligen Klassen ist nicht gefordert. Es ist der Programmierdisziplin überlassen, dass alle Methoden `quack` aller Klassen auch wirklich quackende Semantik haben.

Idee (für Komponenten):

1. Die Komponenten eines Objekts werden durch eine Zuweisung etabliert (wie alle Variablen).
2. Bei Verwendung wird die Existenz der Eigenschaft zur Laufzeit geprüft.

```
Rec.GesamtSumme = 0; -- Rec hat ab hier die Komponente GesamtSumme
for i = 1 to n
    Rec.Gesamtsumme = Rec.GesamtSumme + Rec.Inhalt(i);
    -- Rec muss bereits eine Array-Komponente Inhalt besitzen
    -- ergibt ein überraschendes Resultat (warum?)
```

6.8 Monomorphe und Polymorphe Typsysteme

Monomorphes Typsystem: Eine Variable/Parameter/Komponente/Zeigerziel kann nur Werte genau eines Typs beinhalten.

Polymorphes Typsystem: Eine Variable/Parameter/Komponente/Zeigerziel kann Werte unterschiedlicher Typen beinhalten.

Formen der Polymorphie sind z. B.

- offene Polymorphie der objektorientierten Programmierung
- statisch aufgelöste Polymorphie der generischen Programmierung
- dynamische Typbindung

Ein typisches Problem, das man mit Polymorphie lösen kann, ist die Gestaltung von Graphen mit heterogenen Knoten.

Einfache Beispiele:

Innere Knoten und Blätter eines Baums.

Eine Liste von PKW, LKW und Bussen (die offensichtlich ganz unterschiedliche Merkmale haben).

In beiden Fällen sind Referenzen nötig, für die monomorphe Zieltypen ein Problem darstellen.

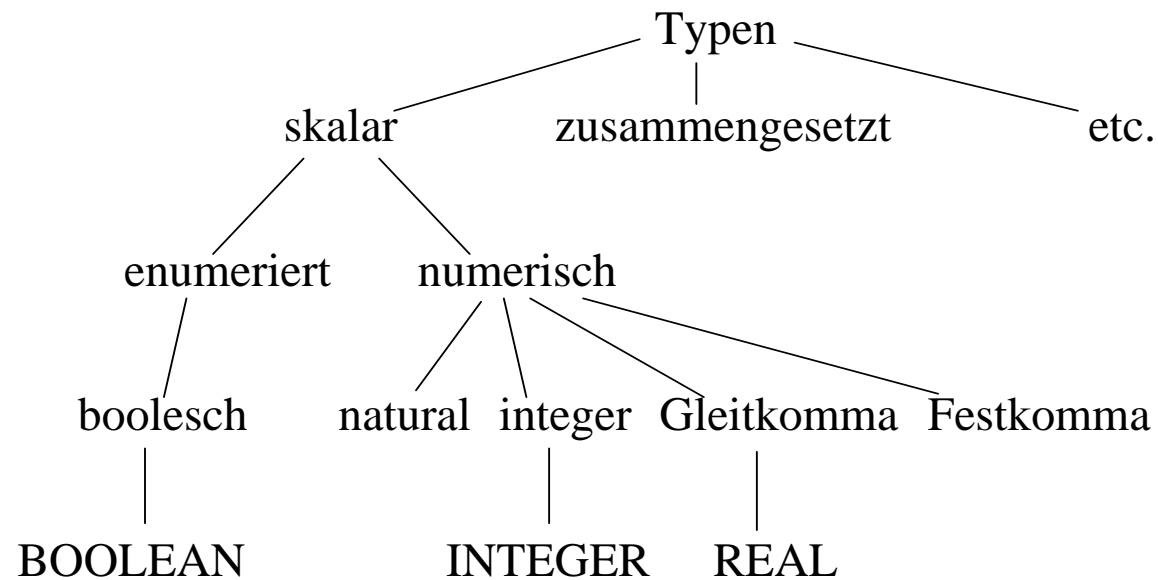
Kapitel 7

Herkömmliche Datentypen

7.1 Skalare Typen

... werden im Wesentlichen als bekannt vorausgesetzt.

Eine Klassifizierung (von vielen):



Begleitende Fragen:

- Wo werden benutzerdefinierte skalare Typen in der Hierarchie eingetragen?
- Welche Operationen sind mit welcher Klasse assoziiert?
- Wie werden Operationen mit Operanden unterschiedlichen Typs behandelt?

7.2 Strukturierte Datentypen und Typkonstruktoren

Aufbau von Datentypen mit nichtatomaren Werten, d. h. Werten, die aus mehreren Komponenten bestehen. Nachfolgend werden einige allgemein anwendbare Verfahren zum Aufbau solcher Datenstrukturen beschrieben.

7.2.1 Kartesisches Produkt (Records)

Sind T_1, T_2, \dots, T_n (nicht unbedingt identische oder paarweise verschiedene) Datentypen, dann ist auch das kartesische Produkt

$$T_1 \times T_2 \times \dots \times T_n$$

ein Datentyp. Werte haben die Form (t_1, t_2, \dots, t_n) .

Operationen enthalten u. a.

- Selektoren: Zugriffe auf einzelne Komponenten
- Konstruktoren ("Aggregate"): Zusammenbau eines Wertes aus Komponenten
- Operationen, die auf zusammengesetzte Werte anwendbar sind, z. B.
 - Test auf Gleichheit/Ungleichheit
 - Zuweisung

in PASCAL, Ada: records
PL/1, ALGOL 68: structures

Implementierung von Records

1. Sind Komponenten dynamischer Größe zugelassen?
=> (a) Speichervergabe analog zur Struktur eines Aktivierungsblocks, d. h., eventuell mit Deskriptoren und Indirektion innerhalb des vergebenen Speichers.
2. Ist die logische Sequenz der Komponenten zwingend für die physische Sequenz?
 - Wenn ja, dann ist (a) für 1. nicht zulässig. Folge: Offsets müssen dynamisch berechnet werden.
 - Wenn nein: erst die statischen Komponenten(inklusive Deskriptoren), möglicherweise nach Adressierbarkeit gepackt, dann die dynamischen Komponenten.
 - Die interne Speicherung ist für den Programmierer von Bedeutung, sobald Unterprogramme aus zwei Sprachen zusammen arbeiten sollen.

Beispiel:

record

M: vector(I..K);

init: boolean;

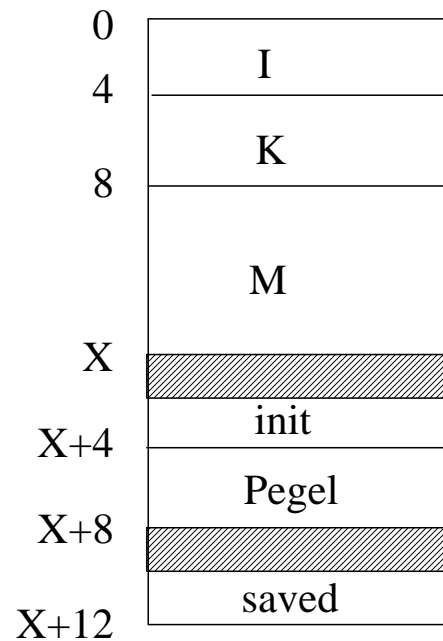
Pegel: integer;

saved: boolean;

end record;

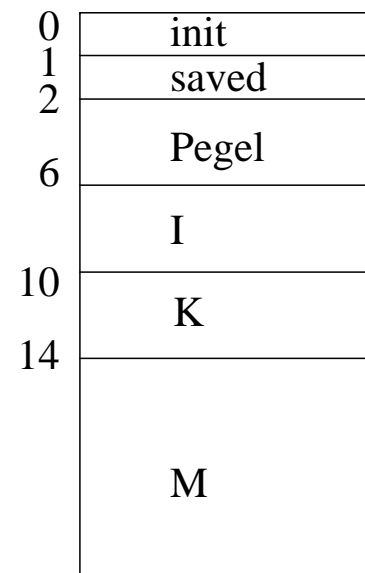
Annahme: eine Vektorkomponente belegt 4 Byte, I und K je 4 Byte.

Layout 1:



$$X = ((K - I + 1) * 4) + 8$$

Layout 2:



Alignment: mod 2

d. h., "init" und "Pegel" sind im Layout 1 nur über komplexe Adressberechnung erreichbar, während im Layout 2 alle Komponenten eine statisch bekannte Relativadresse haben. (Für eine zweite Komponente dynamisch fester Größe wird im Layout eine zweite Indirektion nötig, wie für Aktivierungsblöcke dargestellt.)

7.2.2 Endliche Abbildungen (Arrays)

- $$\begin{array}{ccc} \text{abb} : \text{DEFBEREICH} & & \rightarrow \text{BILDBEREICH} \\ & d & \rightarrow \text{abb}(d) \end{array}$$
- DEFBEREICH muss endlich sein, zum Beispiel:
 - 1..100 -- Unterbereich von Integer
 - (links, Mitte, rechts) - Aufzählungstyp
 - 1..20, 1..100 - mehrdimensional

Typisch: Einschränkung auf enumerierte und ganzzahlige Wertemengen

- $\text{abb}(d)$ heißt Indizierung

Beispiel:

VEKTOR : **array** [1..100] **of** CHAR

MATRIX : **array** [1..10, 1..20] **of** REAL

in PASCAL, Ada: Felder (arrays)

Mehrstufige contra mehrdimensionale Felder

Sind die folgenden Deklarationen semantisch äquivalent ?

(1) M: array [1..10, 1..20] of T;

(2) M: array [1..10] of array [1..20] of T;

- falls ja, dann impliziert (2), dass M[5] als Selektion möglich ist.
- falls nein, dann kann nicht a priori erwartet werden, dass für (1) M[5] (als „Slice“) zugelassen ist.

Implikation von Slices mehrdimensionaler Felder:

Frage: Was passiert bei folgender Zuweisung:

```
x: array [1..20] of T;
```

```
...
```

```
x := M[5];
```

Problem: Abspeicherung

zeilenweise

„row major ”

oder

spaltenweise

„column-major”

Speicherung:

M[1,1] M[1,2]...M[2,1]...

M[1,1] M[2,1]...M[1,2]...

"row-major": $M[5]$ bezeichnet eine zusammenhängende Sequenz, die i. A. die physische Darstellung

```
array [1..20] of T
```

hat.

"column-major": $M[5]$ bezeichnet eine physisch nicht zusammenhängende Auswahl von 20 Elementen in M .

In Modula-2 ist (1) explizit als abkürzende Schreibweise von (2) definiert, wodurch de facto die zeilenweise Darstellung festgeschrieben ist; in Ada werden (1) und (2) semantisch unterschiedlich behandelt.

mehrdimensionale Reihungen

`array (1..2, 1..3) of ...`

- „row major“ (typisch; letzter Index „läuft am schnellsten“)

A[1,1]	A[1,2]	A[1,3]	A[2,1]	A[2,2]	A[2,3]
--------	--------	--------	--------	--------	--------

- „column major“ (FORTRAN; letzter Index „läuft am langsamsten“)

A[1,1]	A[2,1]	A[1,2]	A[2,2]	A[1,3]	A[2,3]
--------	--------	--------	--------	--------	--------

Anmerkungen;

- eigentlich transparent für den Benutzer, außer bei Fragen des Interfacing mit anderen Sprachen und bei Fragen der Lokalität von Speicherzugriffssequenzen
- Berechnung der Adressen der Komponenten
⇒ später

Deskriptoren für Arrays ("Dope")

1. Bei statischen Grenzen:

- pro Typ festgelegt: kein Laufzeit-Deskriptor nötig
- pro Objekt festgelegt: Laufzeit-Deskriptor nötig für Haldenobjekte und bei Parameterübergabe

2. (a) bei dynamischen Grenzen:

- pro Typ festgelegt: ein Laufzeit-Deskriptor pro Typ
- pro Objekt festgelegt: ein Laufzeit-Deskriptor pro Objekt

typische Form:

Untergrenze
Obergrenze

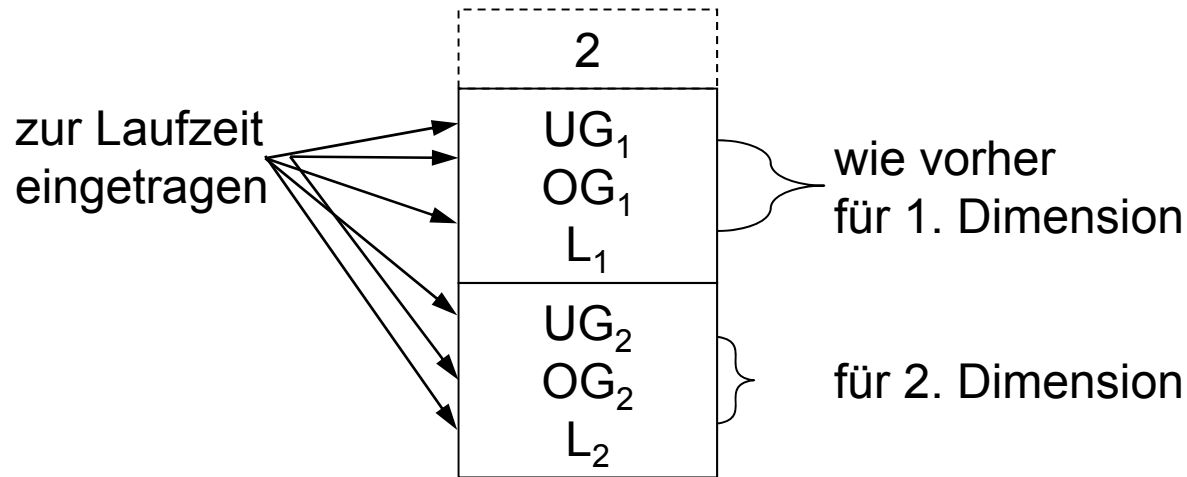
oder

U1
O1
size
U2
O2
size
:

("size" = Größe der Komponenten in dieser Dimension;
alternativ: Anzahl der Komponenten)

Dope für mehrdimensionale Arrays

type Matrix **is array** (A..B, C..D) **of** ...



für row-major:
$$L_i = \begin{cases} \text{Komponentengröße} & \text{für } i = \text{dim} \\ (OG_{i+1} - UG_{i+1} + 1) * L_{i+1} & \text{für } 0 < i < \text{dim} \end{cases}$$

für column-major:
$$L_i = \begin{cases} \text{Komponentengröße} & \text{für } i = 1 \\ (OG_{i-1} - UG_{i-1} + 1) * L_{i-1} & \text{für } 1 < i \leq \text{dim} \end{cases}$$

Zugriff auf $A[\text{index}_1, \dots, \text{index}_{\text{dim}}]$:
$$\text{adr}_A + \sum_{i=1}^{\text{dim}} (\text{index}_i - UG_i) \cdot L_i$$


(gilt für beide Ausrichtungen)

2. (b) bei fester, durch Sprache vorgegebener Untergrenze:
Vereinfachung des Deskriptors:

- auf Längenangabe:

länge	v[0]	v[1]	...	v[länge - 1]
-------	------	------	-----	--------------

- auf Endkennung:

v[0]	v[1]	...	v[n]	
------	------	-----	------	---

Die Endkennung muss ein Wert außerhalb des Wertebereichs der Komponenten sein!

7.2.3 Folgen

- Ist T ein Datentyp, dann ist auch T^* ein Datentyp.
- (Beliebig lange) Aneinanderreihungen von Elementen eines Datentyps
- Im Allgemeinen kein direkter Zugriff auf einzelne Komponenten
- In PASCAL : **file of**

7.2.4 Potenzmenge

- Ist T ein Datentyp, dann ist auch **powerset(T)** ein Datentyp.
- Der Wertebereich von **powerset(T)** ist die Menge aller Teilmengen von T
- Operationen auf **powerset(T)** sind mengentheoretische Operationen, z. B. Vereinigung, Durchschnitt, Komplement

In PASCAL: **set of** SimpleType

Anm: Üblicherweise wird die Mächtigkeit beschränkt. Oft durch Bibliothek und nicht durch ein Sprachkonstrukt unterstützt.

7.2.5 Vereinigung

- Sind T1 und T2 Datentypen mit unterschiedlichen Wertebereichen, dann ist auch

$$T = T1 \cup T2$$

ein Datentyp.

- Auf ein Element t aus T sind die Operationen aus T1 bzw. T2 anwendbar.
- Beispiele:

ALGOL68:	union-Typen
OOP-Sprachen:	classes
in PASCAL, Ada:	records mit Varianten

ALGOL68 - „Union Types“

```
union(int, real) ir1, ir2
    -- ir1, ir2 können entweder int oder real-Werte enthalten
ir1 := ir2;           -- immer möglich

int count;
ir1 := 33;
count := ir1;         -- nicht erlaubt, da nicht statisch prüfbar
```

deswegen:

```
case ir1 in
    (int iv)      : count := iv;
    (real rv)     : Print („Type Error“)
esac
```

Diese Art von "case"-Diskriminierung wird häufig in Sprachen mit Typvereinigung und strenger Typenbindung angetroffen. Sie erfordert minimale Typdeskriptoren zur Laufzeit.

Variante (diskriminierte) Records

Pascal, Modula:

```
unsafe : record
  case Schalter : BOOLEAN of
    true : (I : Integer);
    false: (PI : ↑Integer)
  end
with unsafe do
begin...
  Schalter := true;
  I := 17;
  Schalter := false; -- erlaubt eine Lüge
  PI↑ := ...         -- gleicher Speicherbereich wie I.
                      -- Interpretiert 17 als Adresse!
```

```
very_unsafe : record  
              case BOOLEAN of  
                ...    -- wie oben
```

Hier ist die „derzeitige“ Variante nicht mehr abfragbar.

=> Völliger Zusammenbruch der Typsicherheit

Ada:

```
type int_ptr is access Integer;  
type safe (SCHALTER : BOOLEAN := true) is  
  record  
    case SCHALTER of  
      when true  $\Rightarrow$  I : Integer;  
      when false  $\Rightarrow$  PI : int_ptr;  
    end case;  
  end record;
```

```
O : safe;
```

```
O.I      -- ok
```

```
O.PI     -- Laufzeitprüfung „SCHALTER = false”
```

```
          -- ergibt Ausnahme
```

```
          ( $\rightarrow$  Ausnahmebehandlung)
```

```
O.SCHALTER := false;           -- höchst illegal  
  
O := (false, new Integer'(3));  
-- ok. Diskriminantenwechsel nur durch „Gesamtzuweisung“
```

⇒ **Typsicherheit**

Die Diskriminante agiert als Laufzeitdeskriptor, der unter den "vereinigten" Typen den Typ des aktuellen Werts bestimmt.

Warum existiert das völlig unsichere Modell?

- Der Deskriptor benötigt Speicherplatz
- In importierten, durch derartige Datentypen modellierte Daten ist kein Deskriptor vorhanden
- die Variante ergibt sich als komplizierte Berechnung aus Komponenten, deren Existenz (stufenweise) garantiert werden kann.

(Identische Argumentation gilt für C unions, die in anderen Strukturen eingebettet sind.)

Streitpunkt:

Soll man die obigen Argumente zur allgemeinen Gestaltung des Typmodells heranziehen (und daher auf Sicherheit verzichten) oder das Typmodell durch Sonderfälle komplizieren, um den letzten beiden Punkten gerecht zu werden?

Vergleich "Union" vs. "Variante Records"

Union-Typen:

- beliebige Typen kombinierbar
- statische Typprüfung durch das "type-case"-Konstrukt
- Versuch der Typverletzung muss vom Programmierer im "type-case" explizit behandelt werden.
- gemeinsame Eigenschaften der vereinigten Typen können nicht ohne "type-case" benutzt werden.

Variante Records:

- "Kombination" der Typen muss vorgeplant sein.
- Typsicherheit durch Laufzeitprüfung
- Gemeinsame Komponenten der vereinigten Typen können benutzt werden; eine Laufzeitprüfung entfällt.
- (Aber: gemeinsame Komponenten von Teilmengen der vereinigten Typen sind u. U. schwierig zu modellieren.)

Wir sprechen von einer "**geschlossenen Polymorphie**" der Variablen eines derartigen Typs, weil sie Werte unterschiedlicher Typen aus einer geschlossenen Menge von Typen enthalten können. Dabei sind variante Records eine Methode, dieses konzeptuelle Modell so abzubilden, dass im tatsächlichen Typmodell die jeweilige Typenmenge in einem einzigen Typ zusammengefasst wird.

Es gibt auch eine "offene Polymorphie":

- ⇒ Klassen in OO-Sprachen
- ⇒ generische Einheiten

7.2.6 Rekursion

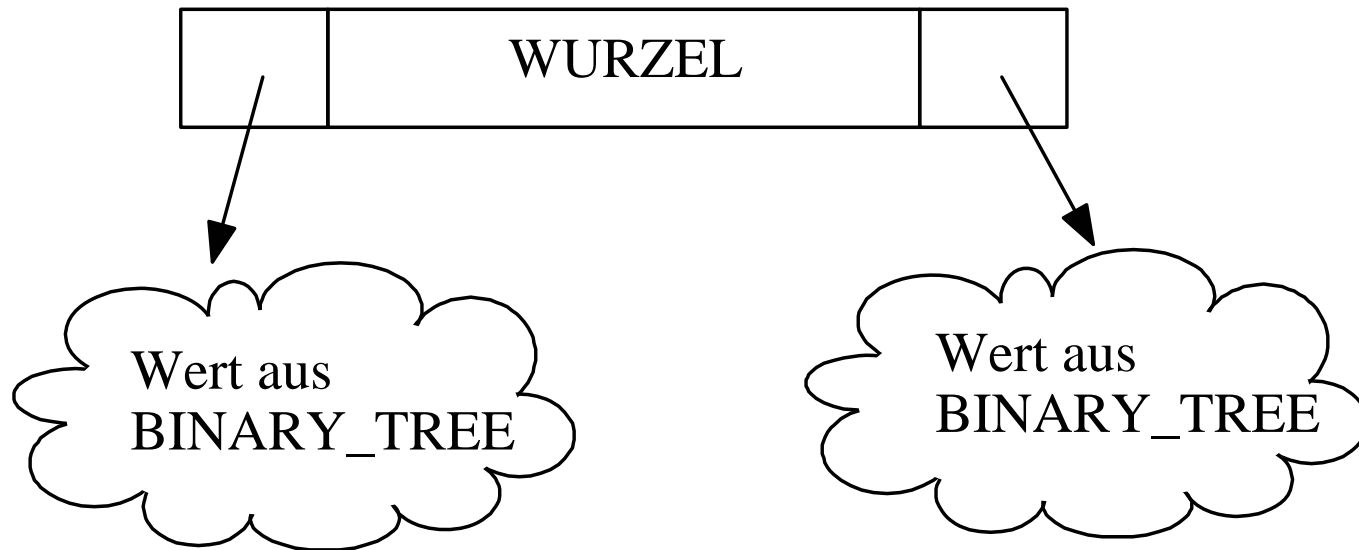
Definition eines Typs mit Komponenten dieses Typs

```
type T    = structure  
  ..  
  KOMPONENTE : T -- Rekursion  
  ..  
end
```

Beispiel:

```
type BINARY_TREE = structure  
  WURZEL: ....  
  SOHN_LINKS: BINARY_TREE  
  SOHN_RECHTS: BINARY_TREE  
end
```

Implementierung durch Zeiger, die aber dem Benutzer verborgen bleiben, z. B.



7.3 Zeiger

- Operationen: implizite und explizite Dereferenzierung, Vergleiche und Zuweisung (der Referenzen)
- Wertebereich: die durch explizite Allokation erhaltenen "anonymen Namen" (Referenzen) auf Objekte
- Typischerweise Identifikation des Typs ("Zieltyp") der von den Werten designierten Objekte (ohne Zieltyp-Angabe kann die Typsicherheit der Sprache extrem gefährdet sein)
- Gutes Werkzeug für die Beschreibung und Implementierung von rekursiven Datentypen, Graphstrukturen, veränderlichen Listen usw.
- Allerdings: eine häufige Fehlerquelle in Programmen

"Standardprobleme":

1. Typdefinitionen für verkettete Liste mit Einträgen "unterschiedlichen Typs":

- Zeiger auf Vereinigungstypen, Klassen o. ä.,
z. B. "ref union(int, real)"
- oder: Vereinigungstypen über Zeigern,
z. B. "union(ref int, ref real)"

Achtung: Semantisch ein erheblicher Unterschied; im 2. Fall kann die Zeigerdarstellung einen Zieltyp-Deskriptor erfordern.

2. Interne Darstellung von Zeigern („Adressen“) ist im Allgemeinen dieselbe wie von ganzen Zahlen. "Versuchung", Arithmetik auf Zeigern durchzuführen. Diese Darstellung ist aber im Allgemeinen nicht garantiert und kann je nach Zielarchitektur variieren (kurze und lange Zeiger, relative Adressen, "fette" Zeiger mit Zieltyp-Deskriptoren usw.)

=> Portierungsprobleme mit Adressarithmetik

3. Zeigerwerte können die explizite Freigabe des designierten Objektes überleben; ferner können Objekte existieren, deren Referenz nicht mehr in Zeigern gespeichert ist.

└─➔ Gefahr der zu frühen oder fehlenden Freigabe
(siehe Haldenverwaltung)

Beispiel für eine ungewöhnliche Zeiger-Semantik (PL/I) :

```
DECLARE P POINTER
```

```
    A FIXED BASED
```

```
    B FLOAT BASED; -- nur über Zeiger P zugänglich
```

```
ALLOCATE A SET P;      -- erzeugt A und setzt Zeiger P auf A
```

```
P > A -- Zugriff auf A;  ok
```

```
P > B -- Zugriff auf B;  Typverletzung
```

Das „Collection“-Modell

Explizites Konzept in Euklid:

```
var my_nodes : collection of nodes;  
type my_ref = my_nodes;  
var my_ref1, my_ref2 : my_ref;
```

Idee: Jeder Zeigertyp ist mit genau einer Kollektion assoziiert.

+ strenge Typprüfung

(+ kein Adressoperator)

=> Jeder Zeiger kann nur in die zugeordnete Kollektion zeigen.

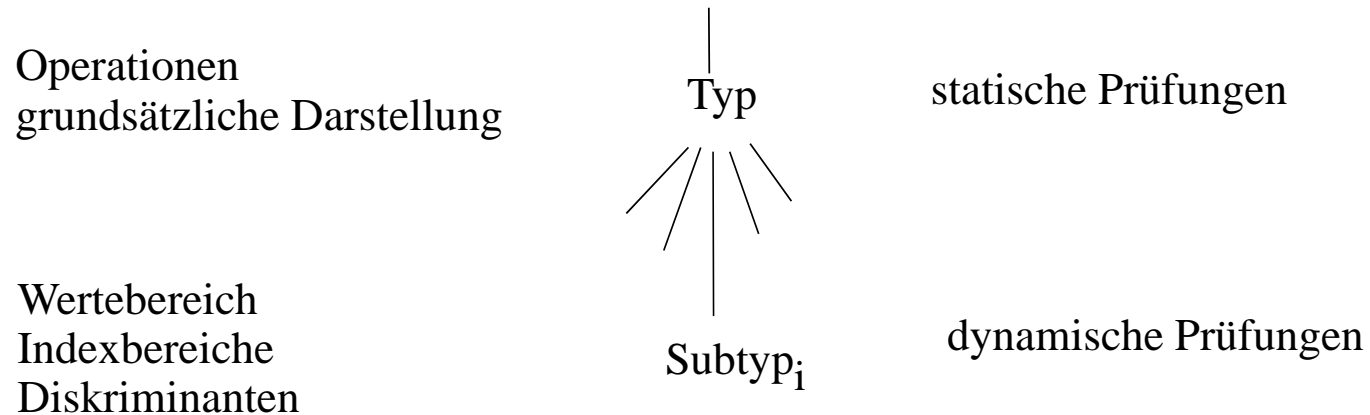
Vorteile:

- Bessere Analysemöglichkeiten (Verifikation, Compiler), mehr Sicherheit
- Eine Kollektion kann möglicherweise mit einer Haldenverwaltung von Speicherblöcken gleicher Größe realisiert sein (=> keine globalen Fragmentierungsprobleme).
- Lokale Kollektionen haben beschränkte Lebensdauer und können daher entsprechend freigegeben werden (Teillösung des Garbage-Collection-Problems).

Nachteile:

- bei Präallokation der Kollektionen eklatante Speichervergeudung
- "interne" Fragmentierung

7.4 Ein Beispiel: Typen und Subtypen in Ada



Objekte besitzen einen bestimmten Subtyp, der gleichzeitig auch den Typ festlegt. Subtypen sind *keine* neuen Typen im Sinne der Typäquivalenz.

Typ/Subtyp-Modell am Beispiel Ada:

- Typ hat
 - **statische** Attribute
 - z. B. Menge der anwendbaren Operationen,
Dimension und Basistyp von Feldern,
Struktur und Komponententyp von Records.
 - dynamische Attribute
 - z. B. Bereichsbeschränkungen von ganzen Zahlen,
Indexbeschränkungen bei Feldern
Diskriminanten bei varianten Records
- Durch Beschränkungen (constraints) können Untertypen (subtypes) von bereits existierenden Typen gebildet werden.

Beispiel :

```
type    Ski_Marke is (ATOMIC, BLIZZARD,  
                    DYNAMIC, DYNASTER, ERBACHER,  
                    FISCHER, HAGAN, OLIN,  
                    ROSSIGNOL, VOELKL) ;
```

```
subtype Ski_Marke_D_H is Ski_Marke  
    range DYNAMIC . . HAGAN;
```

```
subtype Ski_Marke_von_bis is  
    Ski_Marke range Von . . Bis;
```

```
-- Die Variablen Von und Bis müssen zum Zeitpunkt der  
-- Ausführung der Deklaration gesetzt sein
```

- Felder können dynamische Grenzen haben (Auswertung bei der Typ- oder Objektvereinbarung).

Beispiele:

```
type Ski_Angebot is
  array (Integer range < > ) of Ski_Marke;
-- das Symbol <> steht für offene Indexgrenzen
subtype Kleines_Sortiment is
    Ski_Angebot (1..5);
subtype Variables_Angebot is
    Ski_Angebot (Anfang .. Ende);
Mein_Angebot: Variables_Angebot;
Dein_Angebot: Ski_Angebot(1..5);
-- zwei Variablen von gleichem Typ, aber unterschiedlichem Subtyp
```

Typ STRING ist ein Feld mit CHARACTER-Komponenten:

```
type NATURAL is INTEGER
                        range 0..INTEGER'LAST;
type STRING is array(NATURAL range <>)
                                of
CHARACTER;
subtype NAME is STRING (1..30);
```

- Records haben die Form

```
type Name is record  
    :  
    K_i : Typ:i  
    :  
end record;
```

- Records mit Varianten haben die Form

```
type Name ( K : T ) is  
    record ....  
        case K of  
            :  
            :  
            :  
        end case;  
end record;
```

```

type FAHRZEUG is (PKW, LKW, BUS);
type VEHIKEL (ART: FAHRZEUG := PKW) is
  -- PKW ist ein Defaultwert für die Diskriminante ART

```

```

record

```



```

  ...
  case ART of
    when PKW =>
      TUEREN : natural range 1..5;
      SITZE  : natural range 2..8;
      ...
    when LKW => NUTZLAST : .....
      ...
    when BUS =>
      PLATZ : natural range 20..100;
      ...
    end case;
end record;

```

Wird bei der Vereinbarung einer Variablen HERBIE vom Typ VEHIKEL kein Anfangswert für die Komponente AR angegeben, ist HERBIE zunächst ein PKW (Defaultwert), kann aber später zum BUS werden. Dagegen ist TRUCK immer ein LKW.

```
HERBIE : VEHIKEL;  
TRUCK  : VEHIKEL (LKW) ;
```



Der Wert der Komponente

HERBIE.ART

kann nicht einzeln geändert werden, sondern nur in Verbindung mit den entsprechenden anderen Komponenten

HERBIE := (. , ART => BUS,);



Ein Zugriff auf eine in einer Variante nicht existierende Komponente löst die Ausnahme `CONSTRAINT_ERROR` aus, z. B.

HERBIE.NUTZLAST



siehe Ausnahmen

damit nun möglich: eine Liste, in der PKWs, LKWs und BUSse aufgeführt sind:

```
type Fahrzeug is access Vehikel; -- ein neuer Zeigertyp
type Eintrag;
type Fahrzeugliste is access Eintrag;
type Eintrag is record
    Next: Fahrzeugliste;
    Element: Fahrzeug;
end record;
```

```

Liste : Fahrzeugliste;
Liste := new Eintrag' ( Next => Liste;
                        Element => null);

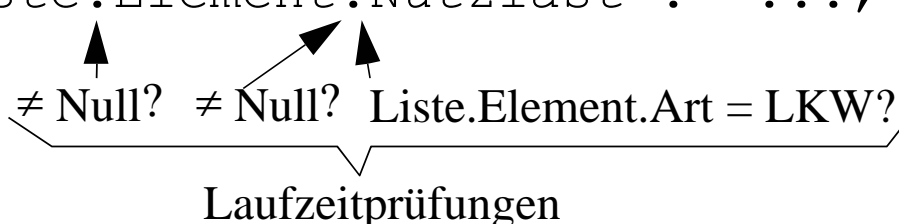
Liste.Element :=
    new Vehikel' (Art => PKW,...);
Liste := new Eintrag'
    (Next => Liste,
     Element => new Vehikel' (Art => LKW,...));
...usw.

```

```

Liste.Element.Nutzlast := ...;

```



≠ Null? ≠ Null? Liste.Element.Art = LKW?
 └──┘
 Laufzeitprüfungen

Kapitel 8

Herkömmliche Ablaufstrukturen

8 Herkömmliche Ablaufstrukturen

8.1 Semantik von Ausdrücken

Ausdrücke berechnen Werte durch Anwendung von Operatoren und Funktionen auf Literale, Namen und Unterausdrücke.

PSen unterscheiden sich deutlich im "syntaktischen Zucker" und der semantischen Mächtigkeit, die für Ausdrücke angeboten werden:

- "natürliche" Schlüsselwörter als Operatoren
Beispiel COBOL: "DIVIDE x BY 7 GIVING y"
- nur Präfix- (m. E. LISP) oder nur Postfix-Operatoren
- Infix-Operatoren mit unterschiedlichen Prioritäts- und Assoziativitätseigenschaften
- Operatoren auf zusammengesetzten Werten
- Operatoren mit Seiteneffekten (z. B. C: "c++")
- Ausdrücke mit interner Kontrollstruktur
- usw.

8.1.1 (Komplexe) Namen oder L-wertige Ausdrücke?

Frage: Was unterscheidet die beiden Indizierungen in

" $A[i] := A[j]$ " ?

Eine Alternative.....

" $A[i]$ und $A[j]$ sind *Namen*. $A[j]$ ist ein Ausdruck, der nur aus einem Namen besteht. Namen für Variablen sind auf der linken Seite von Zuweisungen zugelassen."

Eine andere Alternative....

"A[i] und A[j] sind Ausdrücke" und ferner

"Manche Ausdrücke sind als "*L-wertig*" (*L-values*) definiert. L-wertige Ausdrücke dürfen auf der linken Seite von Zuweisungen (und in anderen Kontexten mit zuweisender Semantik) stehen.

Alle (anderen) Ausdrücke sind "*R-wertig*"."

8.1.2 Präzedenz und Assoziativität von Operatoren

Wie sind Präzedenz und Assoziativität der Infix-Operatoren definiert?

$$A + B * C \equiv A + (B * C) \text{ (Priorität)}$$

$$A + B + C \equiv (A+B) + C \text{ (Assoziativität)}$$

zur Assoziativität:

$$A - B + C \equiv (A - B) + C \text{ oder } \equiv A - (B+C)?$$

$$A ** B ** C \equiv (A ** B) ** C \text{ (links-assoziativ)}$$

$$A ** B ** C \equiv A ** (B ** C) \text{ (rechts-assoziativ)}$$

Festlegung erfolgt durch:

- semantische Regeln
- implizit durch die Gestaltung der rekursiven Ebenen der Ausdrucksgrammatik
- (syntaktisch erzwungene Klammerung durch Benutzer)

Manche Sprachen legen keine unterschiedlichen Prioritäten fest

⇒ Notwendigkeit expliziter Klammerung durch den Programmierer.

Beispiel APL: Alle Operatoren haben den gleichen Vorrang, und Ausdrücke werden von rechts nach links ausgewertet.

Priorität und Assoziativität legen partielle Ausführungsreihenfolge der Operatoren fest.

Dagegen: Ausführungsordnung

$$\begin{array}{lcl} f(x) + g(x) & \equiv ?? & \begin{array}{l} t_1 := f(x); \\ t_2 := g(x); \\ t_1 + t_2 \end{array} \end{array}$$

Die meisten (neuen) Sprachen legen die Ausführungsreihenfolge der Operanden *nicht* fest.

8.1.3 Typkonversionen oder überladene Operationen?

Mit impliziter Typkonversionssemantik:

Beispiele:

$2 + 4.1$ -- zulässig? welches Resultat?

$\equiv \text{Real}(2) +_{\text{Real}} 4.1 = 6.1$

oder

$\equiv 2 +_{\text{Int}} \text{Int}(4.1) = 6$

`int i; i = 3.3 + 4.4;`

$\equiv \text{int}(3.3) +_{\text{int}} \text{int}(4.4) = 3 + 4 = 7$

oder

$\equiv \text{int}(3.3 +_{\text{real}} 4.4) = \text{int}(7.7) = 8$

Die Sprachdefinition muss festlegen, welche Alternative gelten soll und welche zulässigen impliziten Konversionen ausgeführt werden sollen.

→ Gefahr unbeabsichtigter Konversionen;
Risiko zusätzlicher Programmierfehler

Beispiele: C konvertiert sehr freizügig; Modula-2 und Ada sind sehr restriktiv.

Mit überladenen Operationen:

Sprache legt semantisch fest, welche Operatoren mit welchen Parametertypen verfügbar sind.

z. B.	$\text{int} ** \text{int} \rightarrow \text{int}$
	$\text{real} ** \text{int} \rightarrow \text{real}$
aber nicht:	$\text{int} ** \text{real} \rightarrow \text{real}$

8.1.4 „Operatoren = Funktionsaufrufe“-Modell?

$a + b \equiv "+"(a,b) ?$

- + erleichtert semantische Beschreibung
- + (nahezu) notwendig, falls Operatoren benutzerdefinierbar sind
- Probleme für Operatoren mit bedingter Auswertung

Operatoren mit bedingter Auswertung

$A \neq \text{NIL}$ and $A \uparrow .x = 7$

→ Problem, wenn $A \uparrow .x$ unbedingt ausgewertet wird.

Lösungen:

a) zusätzliche Operatoren:

„cand”, „and then”, „cor”, „or else”, "&&", "||"

→ werten zweiten Operanden nur aus, falls benötigt
("short-circuit evaluation").

b) "and" und "or" sind als short-circuit-Operationen definiert.

Aber Problem: Konflikt mit Funktionenmodell (das Argumente vor Aufruf der Funktion auswertet)

Wenn in der Sprache nicht eindeutig festgelegt ist, ob "and" und "or" short-circuit-Operationen sind:

„Caveat Emptor” -- nimm "if...then...else" her.

Beispiel: ISO Pascal: keine Festlegung
 Turbo Pascal: durch Compiler-Schalter einstellbar
 Modula-2: short circuit
 Ada, C, Java: spezielle Operatoren ("and" vs. "and then", "&" vs.
"&&")

8.1.5 Benutzerdefinierte Operatoren

Beispiel 1:

```
M,N : Matrix;
```

```
    M + N      vs.      add(M, N)
```

```
    M * N      vs.      mult(M, N)
```

Beispiel 2:

```
Gross : DM;
```

```
Klein : DPF;
```

```
Klein := Klein + Gross;
```

```
-- "+"(DPF, DM) -> DPF rechnet automatisch um vs.
```

```
Klein := Klein + Wechseln(Gross);
```


Funktionenmodell für Operatoren ist (nahezu) notwendig, um die semantische Grundlage für deren Definierbarkeit zu geben.

Frage der Bindung von benutzerdefinierten Operatoren:

nach Sichtbarkeitsregeln? (wie Funktionen)

oder

implizit immer sichtbar? (wie meistens Operatoren)

Die Antwort auf diese Frage hängt deutlich von der "Stärke der Bindung" von Operationen an Typen ab, spezifisch von der Frage, ob Operatoren ausschließlich im direkten Zusammenhang mit der Typ-Definition deklariert werden müssen.

8.1.6 Bedingte Ausdrücke

... oder, allgemeiner, Ausdrücke mit interner Kontrollstruktur

z. B. Algol68:

```
A := if B > C then B else C;
```

z. B. C und C++:

```
A = (B > C) ? B : C;
```

Beide Ausdrücke weisen A das Maximum von B und C zu.

Frage der Typbestimmung für das Resultat des Ausdrucks:

```
A : int;
```

```
B : real;
```

```
A := (if A > 7 then A else B)    -- Typ dieses Ausdrucks?
```

Bei statischer Typbindung und monomorphem Typsystem wird gefordert, dass alternative Resultate eines Ausdrucks vom gleichen Typ sein müssen.

Aber: ...

Für Sprachen mit Union-Typen oder Klassen kann diese Forderung entsprechend aufgeweicht werden. Der Resultatstyp ist die Union der Typen, bzw. die nächstliegende Oberklasse der beiden Resultatstypen.

Für Sprachen mit dynamischer Typbindung ist obiger Ausdruck sowieso unproblematisch.

8.1.7 Sprachen, in denen alle Konstrukte Ausdrücke sind

Jedes Konstrukt liefert ein Resultat, evtl. das nicht weiter verwendbare „void“. In solchen Sprachen ist eine Anweisung ein Ausdruck, dessen Resultat nicht weiter verwendet wird. Implementierung z. T. in C und Algol68. Zum Beispiel:

$$A := (B := (C := D))$$

Dabei spielt die Unterscheidung zwischen L-wertigen und R-wertigen Ausdrücken eine wesentliche Rolle.

Anmerkung: Bei Sprachen, in denen bereits die Syntax zwischen Ausdrücken und Anweisungen trennt, kann die semantische Analyse im Compiler oft vereinheitlicht/vereinfacht werden, wenn die Sprache semantisch als eine solche "Ausdruckssprache" betrachtet wird. Dasselbe gilt für ...

8.1.8 „Lokative“ Ausdrücke

Idee: Ausdrücke liefern nicht einen Wert als Resultat, sondern eine Referenz.
Der umschließende Ausdruck entscheidet über die Dereferenzierung.

Es wird nun möglich, allgemeine Ausdrücke auf der linken Seite von Zuweisungen und in anderen „L-wertigen“ Kontexten zu verwenden.

Insbesondere können nun Selektoren als Funktionen modelliert (und benutzerdefiniert) werden.

z. B. $A[7] := B[9] \equiv "["(A, 7) := "["(B, 9)$
 $\equiv " := " (" ["(A, 7), "["(B, 9))$

⇒ „L-wertige Sprachen“

„Lokative Ausdruckssprachen“

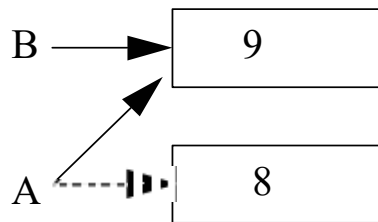
8.2 Zuweisungssemantik

Wichtige Fragen:

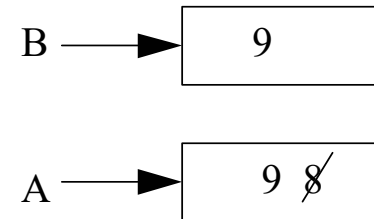
1. Referenz- oder Wertesemantik?

$A := B$

Referenzsemantik:



Wertesemantik:



Anderer Ansatz: verschiedene Zuweisungsoperatoren für Werte- und Referenzsemantik

z. B. Simula: $':='$ für Wertzuweisung

$':-'$ für Referenzzuweisung

2. Implizite Konversionen?

3. Für zusammengesetzte Objekte?

- komponentenweise oder "block-move" (semantisch meist nicht unterscheidbar); Array-"sliding" (d. h. nur Länge muss gleich sein, nicht aber Indexgrenzen);
- flache Kopie (shallow copy): Zeiger werden kopiert, nicht aber deren Zielobjekte;
- tiefe Kopie (deep copy): Zeiger und deren Zielobjekte werden (rekursiv) kopiert. Diese Semantik ist meist nicht in der Sprache verankert, sondern muss über benutzerdefinierte Zuweisung erreicht werden.
- (jedwede Zuweisungsregel sollte eine entsprechende Regel für Vergleiche ("=", "/=") bedingen)

4. Benutzerdefinierbar? Wenn ja, Zusammenhang mit Vergleichen?

5. Laufzeitprüfungen?

8.3 Bedingte Anweisungen

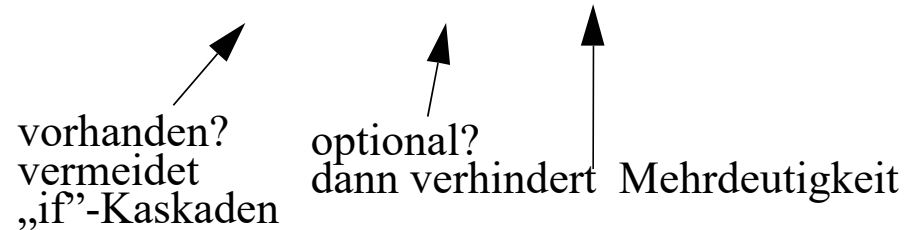
1. IF-Anweisung

if <exp> **then** ...**elsif**...**else**...**fi**

vorhanden?
vermeidet
„if“-Kaskaden

optional?
dann verhindert

Mehrdeutigkeit



Anm.: Ohne Endkennung ("fi") muss jeder Arm eine einzelne Anweisung sein.

2. CASE-Anweisung

```
case <exp> from  
    <choices> => ...;  
    <choices> => ...; } Auswertungsreihenfolge?  
    otherwise   => ...; -- vorhanden?  
end case;
```


- Welche Typen für <exp> zugelassen?
- Müssen <choices> disjunkt sein?
Können Bereiche angegeben werden?
- Wird die Abdeckung des Wertebereichs von <exp>
 - statisch
 - dynamisch
 - gar nicht (und was passiert dann, wenn keine der <choices> zutrifft?)
geprüft?

Implementierungsmodelle:

- als "syntaktischer Zucker" für IF-Anweisung
- durch <choices> indizierte Sprungtabelle

3. „Guarded Command” (Dijkstra)

```
if
    exp1    →    S1
    exp2    →    S2
    :
    expn    →    Sn
fi
```

Semantik:

1. Auswertung aller „Wächter” exp_i
2. nicht-deterministische Auswahl eines S_i , für das $\text{exp}_i = \text{true}$
3. falls $\text{exp}_i = \text{false} \ \forall i$, STOP

8.4 Sprünge

Notwendig? Nein

Verlangt? Ja

Fragen:

- beschränkte Sprungziele?
 - Sprünge von außen in Gültigkeitsbereiche deklarierter Namen müssen verhindert werden.
 - andere Restriktionen sind primär methodischen Ursprungs
- Sprungziele dynamisch berechnet?

z.B. FORTRAN

GOTO (L1, L2, ..., Ln) <exp>



Wert wählt
Sprungziel aus

Noch schlimmer:

```
ASSIGN 17 TO N
```

...-- und weitere ASSIGNments an N

```
GOTO (12, N) <exp>
```

⇒ Programmlesbarkeit geht gegen 0.

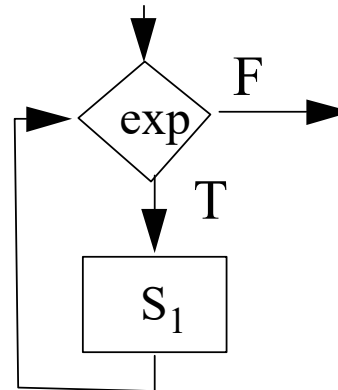
Besser: Kontrollierte Sprünge, insbesondere „exit“ aus Schleifen (und Blöcken)

Literatur: Dijkstra68

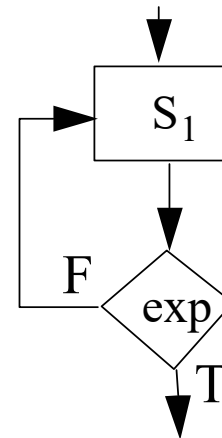
8.5 Schleifen

1. WHILE- und UNTIL-Schleifen

while <exp> **do** S_1 ; [**od**]



do S_1 ; **until** <exp>;



2. FOR-Schleifen

for <name> := <exp₁> **step** <exp₂> **to** <exp₃> **do** S₁;

Fragen:

1. exp_i dynamisch oder statisch?
2. werden exp₂, exp₃ bei jedem Schleifendurchlauf neu ausgewertet?
(z. B. in C)
3. **step** spezifizierbar?
4. Zugelassene Typen für <name>?
5. Ist <name> lokale Größe der Schleife?
6. Ist <name> in der Schleife zuweisbar?
7. Ist Schleife vorzeitig beendbar (-> 'exit')?

Insbesondere Frage 2 entscheidet über die Mächtigkeit der 'FOR'-Schleife.

3. EXIT-Ergänzung; Schleifennamen

```
B : repeat
```

```
  C : repeat
```

```
    :
```

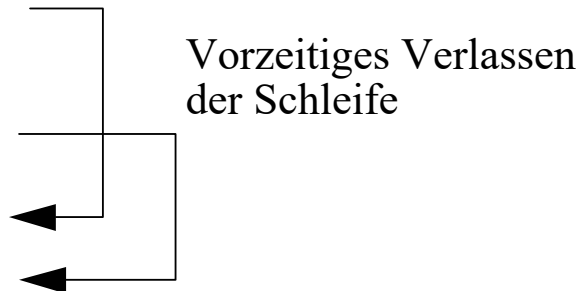
```
    exit C;
```

```
    :
```

```
    exit B;
```

```
  end C;
```

```
end B;
```



Fragen:

- Ist "exit" über mehrere Schleifen hinweg möglich?

Vorteile von "exit":

- für Suchschleifen sehr angenehm
- hat nicht die Nachteile von "goto"s

4. „Guarded Command” (Dijkstra)

do

$B_1 \rightarrow S_1$

$B_2 \rightarrow S_2 \quad \equiv$

\vdots

$B_n \rightarrow S_n$

od

1) Werte alle B_i aus

2) **while** ein $B_i = \text{true}$ **do**

 führe S_i aus

 (Auswahl nicht-deterministisch)

 Werte alle B_i aus

od

8.8.1 Benutzerdefinierte Iteratoren

Am Beispiel CLU:

```
for atom: node in list(X) do  
    action(atom)
```

Beachte: "node" ist ein benutzerdefinierter Typ. "list" ist ein so genannter *Iterator*, definiert durch

```
list = iter(z: linked_list) yields (node)  
    while z <> NIL do -- Schleife nicht CLU-Syntax  
        yield (z.entry);  
        z := z.next  
    od  
end list;
```

Semantik:

`iter` agiert wie eine Coroutine (siehe später), die implizit für jede 'for'-Iteration aufgerufen bzw. fortgesetzt wird. "`yield`" gibt eine Referenz zurück, die 'atom' zugewiesen wird. Die nächste Fortsetzung beginnt nach dem Aufruf von `yield`. Eine Beendigung des Iterators beendet das 'for'-Konstrukt.

Beachte: "`linked_list`" könnte ein abstrakter Datentyp sein, "`list`" eine Operation auf dem Typ.

Vergleiche: in C:

```
for (ptr = z; ptr != 0; ptr = ptr.next) { }
```

erreicht Ähnliches, allerdings nur für nicht-abstrakte Typen.

Kapitel 9

Ausnahmebehandlung ("exception handling")

9. Ausnahmebehandlung („exception handling“)

Idee:

Beim Eintritt einer „außerordentlichen“ Situation („exception“) erfolgt Verzweigung zu einer geeigneten Programmeinheit („exception handler“) zum Zweck der „Reparatur“ und danach Fortsetzung des Programms.

Fragen:

- Wie werden Ausnahmen vereinbart? Sind benutzerdefinierte Ausnahmen vorgesehen?
- Wie werden Ausnahmen ausgelöst?
- Wie wird die Ausnahmebehandlungs-Programmeinheit ermittelt?
- Wo fährt Kontrolle nach Abschluss der Ausnahmebehandlung fort?
- Können vordefinierte implizite Ausnahmen (z.B. Overflow) abgeschaltet werden?
- Wie wird Information zur Ausnahmebehandlungs-Programmeinheit transferiert (globale Variablen, Parameter)?

9.1 Auslösen von Ausnahmen

- Implizites Auslösen laut Sprach-Semantik, insbesondere bei Fehlschlagen einer in der Semantik verankerten Laufzeitprüfung (z. B. Indexprüfung, NULL-Check, Wertebereichsprüfungen, Speicherüberlauf usw.)
- „propagierte“ Ausnahmen (siehe später)
- explizites Auslösen im Benutzercode (Ada: „raise“, C++ „throw“)

Semantisches Modell für implizite Ausnahmen:

als ob die (fiktive) Routine, die den jeweiligen mit der Laufzeitprüfung verbundenen Operator realisiert, eine explizite Ausnahme erhoben hätte.

Implementierungsmodell:

Explizites Auslösen der Ausnahme im generierten Code aufgrund von eingefügten Prüfungen oder Umsetzungen von Traps und Condition Codes
„Auslösen“ = „Sprung“ zur Ausnahmebehandlung

9.2 Vereinbarung der Ausnahmebehandlung

. . . hängt unmittelbar mit der Sprachregel zusammen, wo nach der Behandlung der Ausnahme die Ausführung fortgesetzt wird („Continuation“-Semantik).

- PL/I: (dynamische) Assoziierung einer Routine mit der Ausnahme; diese Routine wird bei Auftreten der Ausnahme implizit aufgerufen.
- CLU, Ada, C++: statische Assoziierung von Codeteilen mit Anweisungsfolgen (z. B. Unterprogrammrümpfen, Blöcken u. ä.), zuständig für die Behandlung von Ausnahmen, die in der jeweiligen Anweisungsfolge erhoben werden.

Beachte aber: Falls eine Ausnahme über die Grenze eines Unterprogrammaufrufs propagiert wird, ist der Code zur Behandlung der Ausnahme im Allgemeinen nicht statisch bestimmbar.

C++: „catch“-Konstrukt

Ada: Exceptionhandler

9.3 „Continuation“-Semantik

„Wo soll nach Behandlung der Ausnahme die Ausführung fortgeführt werden?“

- „Resume“-Semantik:
Ausführung wird am Punkt des Auftretens der Ausnahme fortgeführt,
z. B. PL/I
- „Recovery“-Semantik:
Kontrolle wird an das Ende eines (statisch oder) dynamisch
umschließenden Konstrukts transferiert, z. B. CLU, Ada, C++

pro „Resume“-Semantik:
einige (Sonder-)Fälle elegant behandelbar

pro „Recovery“-Semantik:

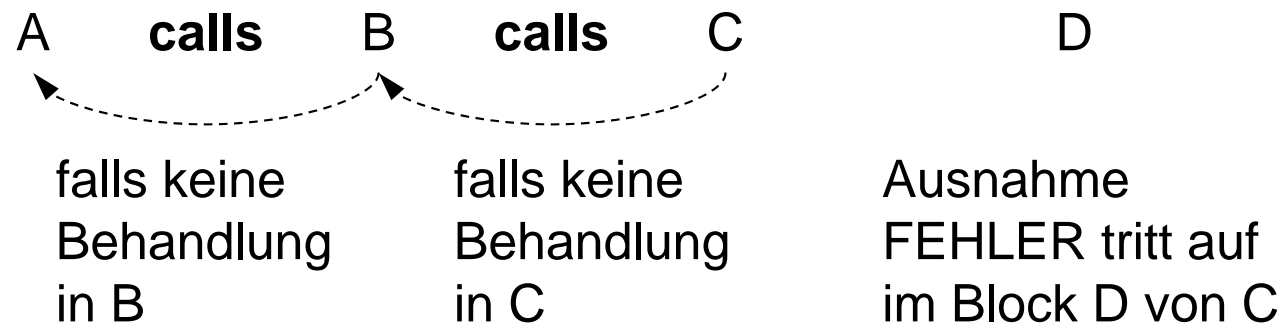
- viele Ausnahmen nicht sinnvoll lokal zu bereinigen
- schließt eine sehr lokale Recovery nicht aus, wenn sinnvoll.

9.4 Propagierung von Ausnahmen

... am Beispiel der nahezu identischen Ausnahmen-Modelle von CLU, Ada und C++:

Ada, C++: Ausnahmen propagieren zu einer passenden Ausnahmebehandlung in einem *dynamisch* umschließenden Konstrukt.

Suche einer Behandlungsroutine erfolgt (bei Blöcken (Ada), Unterprogrammen) dynamisch, d. h.



Ausnahmebehandlung beendet das Konstrukt, in dem sie enthalten ist.

Propagierung von Ausnahmen II

CLU: Ausnahmen können nur in Prozeduren auftreten. Sie propagieren zu einer passenden Ausnahmebehandlung in einem *statisch* unmittelbar umgebenden Konstrukt, oder, wenn kein solches existiert, zur aufrufenden Programmeinheit, wobei Unterprogramme spezifizieren müssen, welche Ausnahmen sie so weitersignalisieren.

„de-facto“-Unterschied:

- Die implizite Propagierung von nicht lokal behandelten Ausnahmen aus Unterprogrammen in Ada und C++ muss in CLU explizit programmiert werden.
- Die Ausnahmen, die von Unterprogrammen propagiert werden können, sind in CLU statisch bekannt (→ Übersetzungszeit-Prüfungen möglich).

9.5 Exception Handling in Ada

- 5 vordefinierte Ausnahmen: `CONSTRAINT_ERROR`
`NUMERIC_ERROR` `-- nur Ada 83`
`PROGRAM_ERROR`
`STORAGE_ERROR`
`TASKING_ERROR`

explizite Vereinbarung durch `UEBERLAUF, FEHLER : exception;`

- implizite Auslösung durch automatische Laufzeittest, z. B.
`ACCESS_CHECK, DISCRIMINANT_CHECK, INDEX_CHECK,`
`DIVISION_CHECK` usw.

explizite Auslösung durch `raise FEHLER;`

`raise;` -- nur innerhalb eines Handlers erlaubt

- Festlegung der Ausnahmebehandlung durch

`exception`

`when FEHLER => . . .`

`when UEBRLAUF => . . .`

`when others => . . .`

am Ende eines „*Rahmens*“ („*Frame*“), d. h. eines Blocks, Unterprogramms,
eines Paket- oder Task-Rumpfes

9.6 Zwei typische Verwendungen von Ausnahmebehandlung

```
for Attempt in 1 .. 10 loop
  begin
    Get(Tape, Data);
    exit;                -- falls erfolgreich
  exception
    when TAPE_INPUT_ERROR =>
      if Attempt < 10 then Back_Space(Tape);
      else raise FATAL_TAPE_ERROR;
      end if;
  end;
end loop;
```

```
loop
  begin
    Get_Sensor_Data(Data);
    Control_Airplane(Data);
  exception
    when others => null; -- lass den Zyklus aus, aber KEEP FLYING!!
  end;
end loop;
```

9.7 Implementierung von Ausnahmen

- Direkte Kosten sind niedrig
Implementierung z. B. durch handler-Eintrag im AB und Propagierung über dynamische Kette.
Alternative: über separate Tabelle von Codeadressbereichen (Vorteil: keine AB-Erweiterung notwendig; Nachteil: teurere Suche bei Auftreten von Ausnahmen)
- indirekte Kosten für implizite Ausnahmen können enorm sein, weil die entsprechenden Prüfungen
 - den Kontrollfluss-Graphen stark komplizieren und damit
 - viele Optimierungen verhindern würden.

Pragmatische Lösung: spezielle Regelung in der Sprachdefinition, diese Komplikation bei Optimierungen ignorieren zu dürfen.

⇒ wenig Garantien über genauen Zustand bei Auftreten impliziter Ausnahmen

⇒ ***implizite Ausnahmen nicht als beabsichtigten Kontrollfluss verwenden!***

Example:

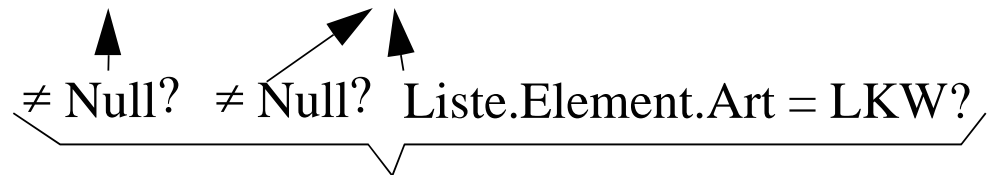
```
State := 1;  
State := 2;  
X := B/C;  
State := 3;  
Y := A/B;  
State := 4;  
Z := D/A;  
State := 4;  
exception  
when Constraint_Error => // raised for division by 0 in Ada  
    if State = 1 then Print("Division by C; C is 0");  
    elsif State = 2 then Print("Division by B; B is 0");  
    elsif State = 3 then Print("Division by A; A is 0");  
    else Print("??? utter surprise!!!");  
    end if;
```

Dieser Code könnte "utter surprise" verursachen (wegen einer Standard-optimierung in Compilern)

9.8 Implizites Auslösen von Ausnahmen

```
type Gattung is (PKW, LKW, BUS);  
type Vehikel (ART: Gattung) is  
    record  
        case ART of  
            when PKW => ...  
            when LKW => Nutzlast : Tonnen; ...  
            when BUS => ...  
        end case;  
    end record;  
type Fahrzeug is access Vehikel;  
type Eintrag;  
type Fahrzeugliste is access Eintrag;  
type Eintrag is record  
    Next: Fahrzeugliste;  
    Element: Fahrzeug;  
end record;
```

Liste.Element.Nutzlast := ...;



Liste : Fahrzeugliste; ... Laufzeitprüfungen mit ggf. Werfen von Ausnahmen

Kapitel 10

Objektorientierte Programmierung

6 Objektorientierte Programmierung

Zur objektorientierten Programmierung:

- Anderes *Denkmodell* als bei herkömmlichen (prozeduralen) Sprachen
 - Versuch einer Annäherung an die Arbeitsweise des menschlichen Gehirns (Kognitionswissenschaften)
- Umsetzung von im Entwurf etablierten, objektorientierten Konzepten in Programmiersprachen
 - Programmiersprachen bzw. Compiler überbrücken die Diskrepanz zwischen OO-Entwurf und Nicht-OO-Hardware
- Programme handeln von *Objekten* anstatt von Operationen auf Daten.

6.1 Objekte

- Datenkapseln, bestehend aus Daten ("Attributen", "Komponenten") und Operationen ("Methoden").
- Datenattribute speichern Zustand des Objekts und können nach außen verborgen werden.
- Das Objekt bestimmt, wie eine Methode realisiert ist. Ihr Effekt hängt im Allgemeinen von ihren aktuellen Parametern und dem Zustand des Objekts ab.
- Objekte werden Klassen zugeordnet, die ihre Attribute und Methoden beschreiben.

Zwei konzeptionelle Ansätze:

1. Objekte sind *aktive* Elemente, die durch *Nachrichten* kommunizieren, aufgrund derer Methoden ausgeführt werden (z. B. Smalltalk-Modell).
2. Objekte sind *passive* Elemente, die Methoden zur Anwendung bereitstellen (z. B. C++, Ada 95, Java).

6.2 Der Klassenbegriff

Existiert in der Beschreibung vieler Programmiersprachen, um Gemeinsamkeiten von Typen auszudrücken, z. B.

"die PRED- und SUCC-Funktionen der Aufzählungstypen"

"Komponentenselektion der Recordtypen" usw.

Er kann aber auch als Definitionsmechanismus für benutzerdefinierte Klassen ausgeprägt werden

- als Ersatz für das Typmodell
- als Erweiterung für das Typmodell

Zentrale Konzepte:

- Klassenhierarchie und Vererbung
- Erweiterbarkeit von Unterklassen
- Redefinierbarkeit von ererbten Methoden
- Polymorphie
- Bindung von Operationen zur Laufzeit ("dispatching")
- (abstrakte Operationen/Klassen, Interfaces)

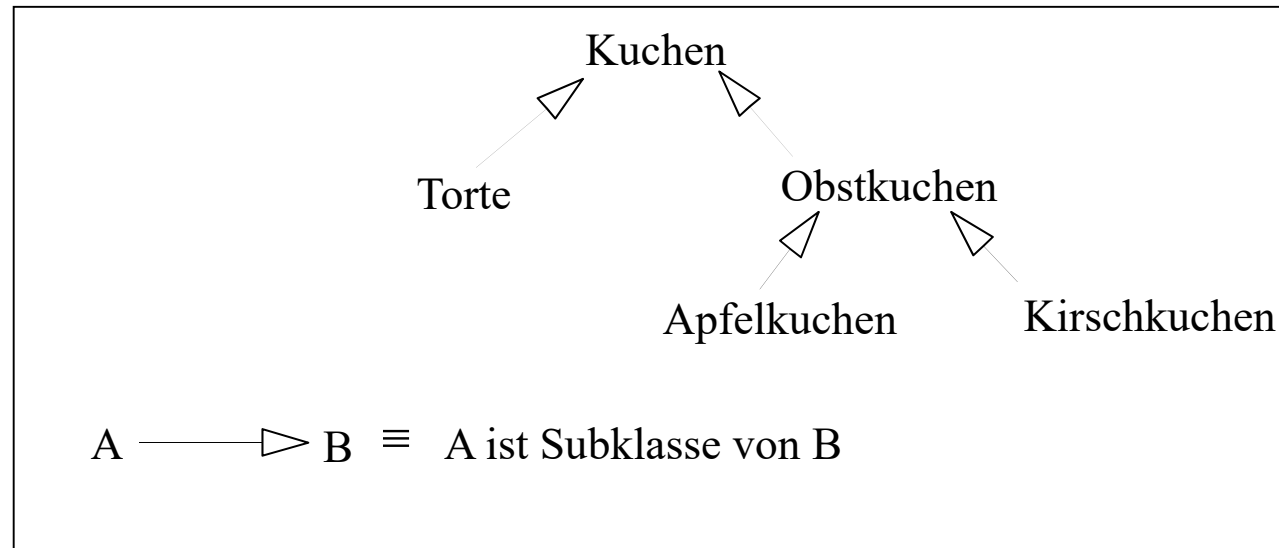
Zwei wichtige Grundsätze:

- Klassendefinitionen beschreiben die gemeinsamen Eigenschaften einer Menge von Objekten.
- Jedes Objekt ist "*Instanz*" ("*Exemplar*") genau einer (Unter-)Klasse, die alle Eigenschaften des Objekts beschreibt. Diese Klasse wird im Folgenden als der (Klassen-)Typ des Objekts bezeichnet.

6.2.1 Klassenhierarchie, Vererbung, Erweiterbarkeit

- Idee: Gemeinsame Eigenschaften von Objekten sollen nur einmal in einer Klassendefinition definiert werden.
 - Klassen werden in Beziehung gesetzt: Sub-/Superklassen- (Unter-/Oberklassen-)Beziehung ergibt Klassenhierarchie.
 - Klasse „erbt“ alle Attribute und Methoden der Superklasse, d. h., transitiv die Eigenschaften aller Vorfahren in der Klassenhierarchie.
 - Für jede Klasse in der Hierarchie können zusätzliche Datenattribute und Methoden definiert werden.
- ⇒ Unterklassen besitzen diese zusätzlichen Eigenschaften, Oberklassen besitzen sie nicht.
- Ein Objekt als Instanz einer Klasse hat genau die ererbten und zusätzlich definierten Eigenschaften dieser Klasse.

Beispiel:



6.2.2 Redefinierbarkeit

Die Realisierung ererbter Methoden kann in der Subklasse anders als in der Oberklasse definiert werden.

- Anpassung der geerbten Methoden an spezifischere Eigenschaften der Unterklasse,
 - um die Konsistenz zusätzlicher Datenattribute zu gewährleisten
 - um die Semantik der Methode generell den Eigenschaften des modellierten Objekts in der realen Welt anzupassen
 - um effizientere Implementierung zu erhalten
- Die Annahme dabei ist, dass die Redefinition nach wie vor der (allgemeiner gehaltenen) beabsichtigten Semantik der ursprünglichen Definition gerecht wird.

Gefahren (insbesondere im Zusammenhang mit Polymorphie):

- Eine als zusätzliche Methode beabsichtigte Definition ist versehentlich eine Redefinition einer ererbten Methode.
- Die obige Annahme des Erhalts der allgemeineren Semantik der sonst ererbten Operation ist verletzt.
Teillösung: Zusicherungen ("Assertions") als Semantik-Spezifikation werden mitvererbt und zur Laufzeit geprüft (Eiffel, Ada12).
- Diese Gefahren sind besonders kritisch bei nachträglicher Einführung von Redefinitionen, da selbst bei Abwesenheit von Objekten als Instanzen der geänderten Klasse ein bereits validiertes System fehlerhaft wird!

Ein Beispiel (Ada 95):

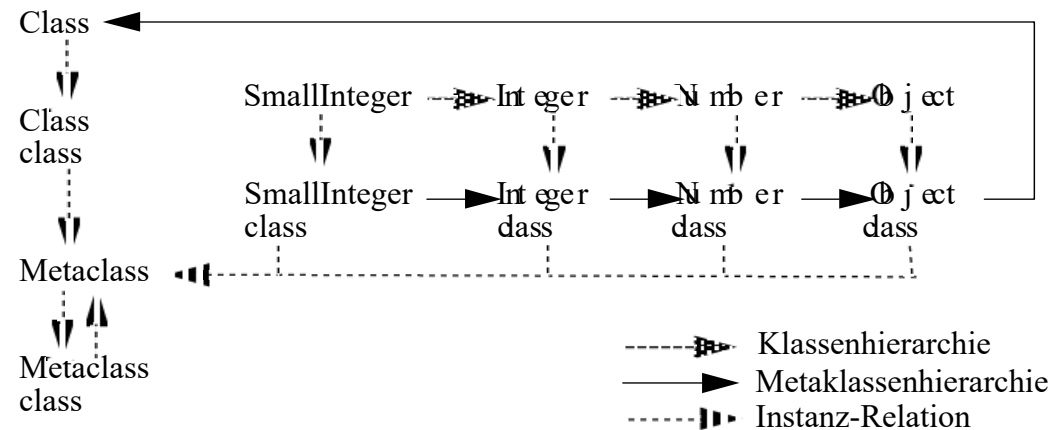
```
type Window is tagged private;
    -- Datenkomponenten bleiben verborgen
procedure display(W: Window);
procedure erase(W: Window);
....
type LabeledWindow is new Window
    with record
        Text: myString;
    end record;
-- zusätzliche Datenkomponenten hier sichtbar;
-- andere Möglichkeit: Verbergen durch "with private"
procedure display(W: LabeledWindow);
    -- 'display' wird redefiniert
-- procedure erase(W: LabeledWindow);
    -- 'erase' wird implizit ererbt
procedure highlight(W: LabeledWindow);
    -- 'highlight' ist zusätzliche Methode für LabeledWindow
```

6.2.3 Ausflug: Der Klassenbegriff in Smalltalk

... ist besonders interessant, weil ...

- Die Klasse ebenfalls Objekt ist, d. h.,
 - dynamische Manipulation von Klassen möglich
 - einheitliche Notation für Instanzen und Klassen
 - Bereitstellung von objektübergreifendem Wissen der Klasse
 - Metaklassen-Konzept: Jede Klasse ist Instanz einer *Metaklasse*. In der Metaklasse „Class“ sind Attribute und Operationen definiert, die für alle Klassen gelten (z. B. 'new').

Metaklassen-Hierarchie (Auszug):

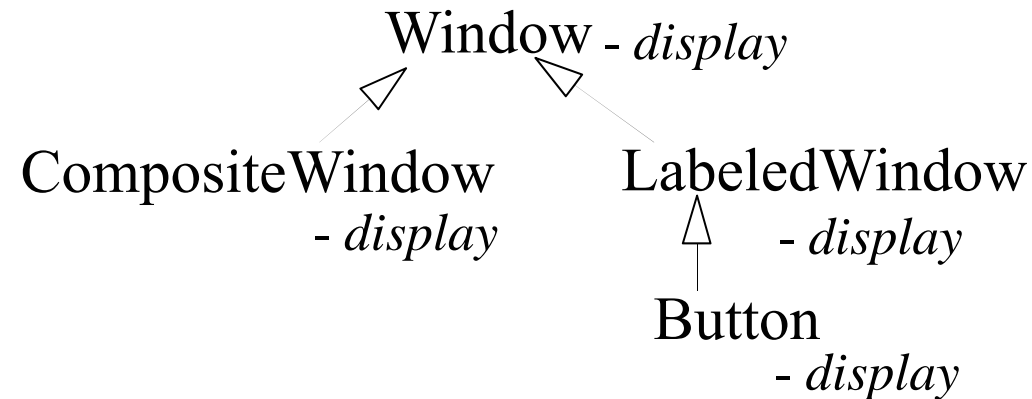


- Beschreibung einer Klasse:

Klasse als Objekt	Klassenname	
	Daten	class instance variables
	Operationen	class methods
Instanzen der Klasse	Daten	instance variables
	Operationen	instance methods

6.2.4 Polymorphismus und dynamisches Binden

Beispiel: Fenstersystem



```
w : Window;  
:  
w := windowAt(mouse_position);  
-- dieses Window kann von beliebigem Window-Typ sein  
display(w);
```

Frage 1: Sind eine solche Zuweisung und ein solcher Aufruf zulässig?

Frage 2: Wenn ja, welche display-Funktion ist anzuwenden?

Beachte: "Herkömmliches" statisches Typ- und Namensbindungsmodell (Overloading-Modell) würde hier die 'display'-Implementierung des Typs 'Window' aufrufen.

Wunsch: Der spezifische Klassen-Typ des Objekts (der erst zur Laufzeit bestimmbar ist) soll die Realisierung der anzuwendenden Operation bestimmen: dynamisches Binden (*dispatching*).

Benannte Objekte und Zuweisungssemantik in OO-Sprachen

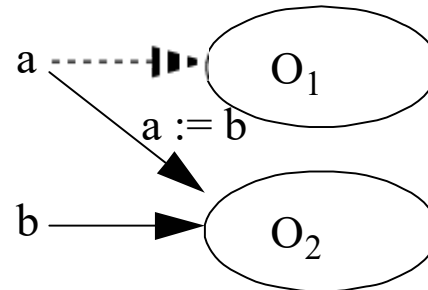
Polymorphismus = Fähigkeit einer benannten Einheit (Variable, Konstante, Parameter usw.), Objekte von verschiedenen Klassentypen zu bezeichnen.

Beachte aber: Der (Klassen-)Typ von Objekten wird bei der Erzeugung des Objekts festgelegt und ist danach unveränderbar.

Bereits bekannt: Speicherverwaltungsprobleme, wenn sich die Größe von Variablen bei Zuweisung ändern kann.

Nun zusätzlich: Frage der Bindung von Methoden je nach "Belegung" der Variablen.

Deshalb ist in OOP-Sprachen entweder die Zuweisungssemantik (zumindest bezüglich der Variablen einer Klasse) durch Referenzsemantik definiert:



Oder aber - bei Wertesemantik der Zuweisung - der Polymorphismus auf die Ziele von Zeigervariablen beschränkt.

(Kleine Ausnahme: Ada 95 lässt auch die Deklaration von initialisierten Variablen von Klassen zu, fordert dann aber, dass nur Werte des Klassen-Typs der Initialisierung zugewiesen werden können.)

Analog zur Typbindung von benannten Einheiten besteht eine **(dynamische oder statische) Klassenbindung** von benannten Einheiten.

Dynamische Klassenbindung benannter Einheiten (Smalltalk, Skriptingsprachen)

- Objekte beliebiger Klassen-Typen sind Variablen zuweisbar
⇒ uneingeschränkter Polymorphismus
- Bei Aufruf einer Methode ist nun eine Laufzeitprüfung erforderlich, ob die aufgerufene Methode für das benannte Objekt überhaupt vorhanden ist.

Statische Klassenbindung benannter Einheiten (Eiffel, C++, Java, Ada)

- Zuweisbarkeit wird durch die Klassenhierarchie eingeschränkt
⇒ eingeschränkter Polymorphismus
- Deklaration der benannten Einheit legt die Klasse (nicht den Klassentyp!) der benennbaren Objekte fest.
- Immer erlaubt: Zuweisung speziellerer Objekte, z. B.

```
myWindow : Window;  
aLabeledWindow : LabeledWindow;  
myWindow := aLabeledWindow;  -- ok!
```

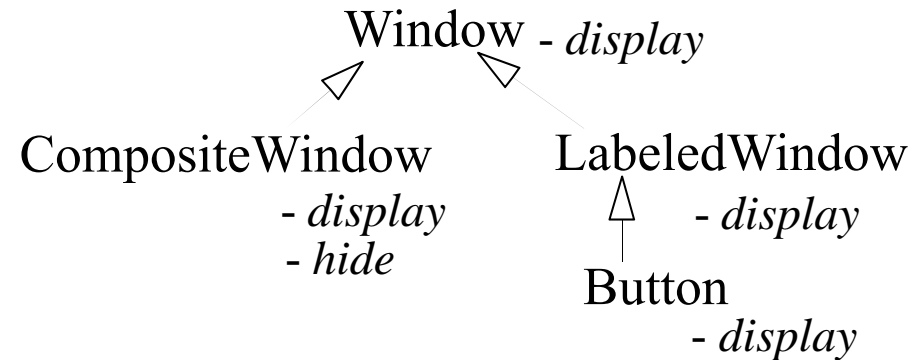
- Eine Zuweisung aus einer Oberklasse

```
aLabeledWindow := myWindow;
```

entweder nicht erlaubt (Eiffel) oder mit Laufzeitprüfung verbunden (evtl. mit expliziter Typkonversion, z.B. in Ada), ob Klassentyp des zugewiesenen Objekts Unterklasse der deklarierten Klasse der Variablen ist.

- Zulässigkeit von Methodenaufrufen wird von der deklarierten Klasse des Namens abhängig gemacht (d. h., "normale" statische Namensbindung), nicht vom Klassentyp des benannten Objekts, und wird damit statisch prüfbar (Bedingung: Operationen dürfen in Subklassen nicht verborgen werden!)
- Die Regeln, welche Aufrufe einer dynamischen Implementierungsbindung ("Dispatching") unterliegen, variieren bei den Sprachen.

Beispiel:



```
w : Window;  
cw: CompositeWindow;  
...  
w := cw;  
display(w);    -- Existenz von CompositeWindow.display garantiert  
hide(cw);      -- offensichtlich o.k.  
hide(w);       -- illegal! (obwohl 'hide' für derzeitigen Wert von w definiert wäre)
```

- Entsprechende Restriktionen gelten bei Parameterübergabe und generischen Einheiten (bei generischen Klassen muss die aktuelle Klasse gleich der formalen Klasse oder eine ihrer Unterklassen sein).

Regeln für die dynamische Bindung der Methodenimplementierung

Welche Aufrufe dispatchen zur Implementierung der Methode des Klassentyps des Empfängerobjekts?

Java: Alle (ohne weitere Qualifikationen des Aufrufs)

C++: Nur Aufrufe von Methoden, die als "virtual" deklariert wurden, und deren Redefinitionen

Ada: Nur Aufrufe, deren kontrollierendes Argument von klassenweitem Typ ("T'Class") ist, sowie Aufrufe abstrakter Methoden

Aufrufe von Implementierungen der Ober- oder Unterklassen eines Klassentyps:

C++, Ada: z.T. durch die regelgemäße statische Bindung, sonst ...

C++, Java: durch Qualifizierung der gewünschten Klasse im Aufruf

Ada: durch sogenannte View-Konversion des kontrollierenden Arguments zur gewünschten Klasse. (View-Konversionen zur Unterklasse prüfen den Klassentyp auf Inklusion.)

Realisierung der dynamischen Bindung der Methodenimplementierung

Wie wird im vorangegangenen Beispiel bestimmt, welche 'display'-Realisierung beim Aufruf 'display(w)' aufgerufen werden soll?

- Mit jedem Objekt wird ein Laufzeitdeskriptor ("tag") seines Klassentyps gespeichert.
- Aufruf einer Methode befragt den "Tag" des Objekts (bzw. des kontrollierenden Arguments - siehe später), um den Klassentyp des Objekts festzustellen.
- Implementierungstechnisch kann dieser "Tag" ein Dispatch-Vektor der Methoden sein, in dem die für den jeweiligen Klassentyp zutreffende Realisierung referenziert wird (die dann, durch Indirektion, aufgerufen wird).

6.3 Wertesemantik und Koexistenz von Typen und Klassen

In OOP-Erweiterungen herkömmlicher Sprachen muss das Typmodell mit dem Klassenmodell in Einklang gebracht werden.

C++:

```
labeledWindow lw;  
Window w;  
w = lw;    -- Zuweisung mit impliziter Wertekonversion, d. h.,  
           -- zusätzliche Komponenten gehen verloren.
```

Die Zuweisung in C++ hat Wertesemantik. Polymorphismus ist auf Zeiger beschränkt. Die implizite Konversion eines Zeigers auf eine Unterklasse zum Zeiger der Oberklasse ist zugelassen. Polymorphie wird im Wesentlichen durch implizite Zeigerkonversion gemäß dieser Regel erreicht.

Ada 95:

Die Zuweisung hat Wertesemantik. Ada unterscheidet den monomorphen Typ T von der polymorphen Klasse T'Class. Polymorphie-Semantiken (Zuweisbarkeit, Dispatching von Aufrufen) gelten nur für die als T'Class ("klassenweiter Typ") deklarierten Variablen bzw. für die Zielobjekte entsprechend deklarerter Zeigertypen.

```
lw: labeledWindow; -- spezifischer Typ
function new_one return labeledWindow;
w: Window;          -- spezifischer Typ
wc: Window'Class := lw; -- Klasse, Init. erforderlich
w := lw;             -- illegal; verschiedene Typen
w := Window(lw);     -- legal; verkürzende Wertekonversion wie C++
lw := (w with ...    -- ... = notwendige zusätzliche Attribute);
wc := new_one;       -- legal, da vom Typ des Werts von Z
```

Die Polymorphie Stack-residenter Variablen ist nur in obiger eingeschränkter Form möglich: Nach der erforderlichen Initialisierung ist der Klassentyp des Werts der Variablen unveränderlich.

Benannte Zeigertypen sind nicht polymorph, auch wenn ihr Zieltyp klassenweit ist. Anonyme Zeigertypen hingegen haben polymorphe Eigenschaften.

6.4 "Reine" und "fast reine" Klassenmodelle

Smalltalk:

Es gibt nur Klassen (und Metaklassen). Die Zuweisung hat Referenzsemantik. Alle Methodenaufrufe dispatchen, da die Namensbindung für die Methoden dynamisch ist. (Tatsächlich ist das Ausführungsmodell, dass das Objekt auf eine Nachricht reagiert, in der der Methodenaufruf verklausuliert steht.)

Java:

Einige Basistypen (z. B., boolean, int, float) haben Wertesemantik ohne Polymorphie. Arrays haben Referenzsemantik ohne Polymorphie. Alle anderen Objekte sind Instanzen von Klassen und werden mit Referenzsemantik zugewiesen. Die entsprechenden Variablen sind polymorph. Alle Methodenaufrufe außer Konstruktoren dispatchen.

6.5 Notation des Zugriffs auf Attribute und Methoden

Smalltalk: Zugriff ist "indirekt" über eine Nachricht an das Objekt:

Form einer Nachricht (Präfixnotation):

Empfänger	Operation	Argumente
-----------	-----------	-----------

z. B.:

```
myList sort  
myList merge: anotherList  
-- kein Zugriff auf Datenkomponenten möglich
```

Java, C++, Eiffel u.a.: Zugriff ist über Namen des Objekts in Präfixform:

```
myList.merge(anotherList)  
myList.DatenKomponente
```

(auch in Ada05 als Alternative zur Parameternotation)

Ada 95 u.a.: Objekte sind Parameter bei Methodenaufrufen und Präfixe bei Komponentenselektion:

Parameternotation

<code>operation (... , O₁,...)</code>
--

z. B.:

```
sort(myList)
merge(myList, anotherList)
myList.DatenKomponente
```

6.5.1 Probleme der Präfixnotation

Problem: In der Implementierung einer Methode entsteht häufig die Notwendigkeit, auf das Objekt selbst Bezug zu nehmen.

Lösung: Spezieller Name bezieht sich stets auf dieses Objekt:
self, *Current*, *this*, etc. (möglich auch Klassenname, falls Klasse nicht selbst als Objekt betrachtet wird)

(Bei der Parameternotation existiert dieses Problem nicht, da das Objekt über den formalen Parameternamen benannt ist.)

Beispiel:

```
LabeledWindow:
```

```
    display
```

```
        self displayBorder.
```

```
        self displayText.
```

```
:
```

```
myButton display
```

```
-- Button ererbt display-Methode von LabeledWindow
```

```
-- Button hat z. B. 'displayBorder' redefiniert; ererbtes 'display' soll
```

```
-- diese Redefinition aufrufen
```

Problem: Zugriff auf Methode von Vorfahren, die in darunterliegender Klasse redefiniert werden.

Lösung (in Smalltalk): *super* = "behandle das vorliegende Objekt, als ob es vom Typ der nächsten Oberklasse der Klasse wäre, in der 'super' verwendet wurde."

Beispiel: Initialisierung von Objekten

```
LabeledWindow:
```

```
  initialize
```

```
    super initialize.
```

```
    -- initialisiere entsprechend der Vorschrift der Oberklasse;
```

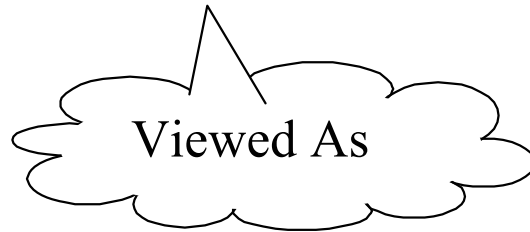
```
    -- ohne 'super' wäre dies ein (endlos) rekursiver Aufruf
```

```
    text := ''.
```

Allgemeiner: Möglichkeit, Klasse für Methodenbindung explizit zu benennen
(COOL, C++)

Beispiel: COOL

```
MyButton VAS Window display
```



-- ruft die display-Operation von Window

(Bei Parameternotation wird dieses Problem durch "Viewkonversion" des
Parameters umgangen. Z. B. in Ada

```
Window(X).display;    bzw.  display( (Window(X)) );
```

-- wobei X z. B. ein Button ist

)

6.5.2 Probleme der Parameternotation

Da mehrere Parameter möglich sind, ist nicht direkt offensichtlich, welche Methode aufgerufen werden soll.

Es muss von der Sprachsemantik festgelegt werden, welcher aktuelle Parameter für das Dispatching der Methode herangezogen werden soll ("kontrollierendes Argument"). Typische Lösung: Definitions- und Ererbungsregeln schließen aus, dass eine Methode mehrere kontrollierende Argumente besitzt.

Zulässigkeit des Aufrufs wird entsprechend den Sichtbarkeitsregeln für Unterprogramme geregelt.

(Bei der Präfixnotation existieren diese Probleme nicht, da die Methoden-selektion in Analogie zur Komponentenselektion definiert wird und Methoden damit implizit sichtbar sind.)

Beispiele:

TRELLIS/OWL: der erste Parameter ist per Definition das kontrollierende Argument

CLOS: alle (Klassen-)Parameter sind kontrollierend
 (=> Suche nach "passender" speziellster Implementierung)

Ada 95: Klassen-Typ, an den die Operation durch Paketstruktur 'gebunden' ist, bestimmt das kontrollierende Argument; mehrere solche Typen sind ausgeschlossen.

6.6 Varianz bei Vererbung/Redefinition von Methoden

Was soll die Signatur einer ererbten Methode sein, insbesondere, wenn Parameter derselben Klasse übergeben werden? Welche Signaturen ergeben Redefinition oder zusätzliche Methode?

Beispiel (hier in der Parameternotation – die gleiche Frage stellt sich auch für Präfixnotation):

```
type T is tagged .... -- ein "Klassentyp"  
function add(A,B: T) return T;  
type ST is new T with ... -- zus. Komponenten
```

Welche Parameter hat die von ST ererbte 'add'-Funktion?

```
function add(A,B: ST) return ST; -- oder  
function add(A,B: ST) return T; -- oder  
function add(A: ST; B: T) return T;
```

Die dritte Alternative ist die Variante, die typischerweise für Präfixnotation erzeugt bzw. für Redefinition gefordert wird (d.h. implizite Variation des "Empfängerobjekts", keine Variation der Parameter).

Die erste Variante wird unter der Bezeichnung "Covarianz" für Präfix- und Parameternotation vorgeschlagen (siehe auch die Frage nach dem kontrollierenden Argument).

Weiterer Vorschlag (z. B. in Eiffel): "Contravarianz" bei Redefinitionen = die neue Parameterklasse ist Oberklasse der ererbten Parameterklasse.

"Übung": Betrachte die jeweils implizierte Semantik mit den Ersetzungen:

```
T = Integer;   ST = Short_Integer
```

```
T = numerischer Typ;   ST = num. Matrix
```

Java, C++:

Die Klasse des Empfängerobjekts wird covariiert. Die Klassen der Parameter und des Resultats sind invariant.

Ada:

Alle Vorkommnisse des Vartyps in Parametern und Resultat werden covariiert.

Java, C++, Ada:

Homographie zur variierten ererbten Methode redefinieren.

In C++ zusätzlich: Resultatstyp muss invariant oder covariiert sein.

In Java zusätzlich: Resultatstyp muss invariant sein.

Nicht-Homographie überladen.

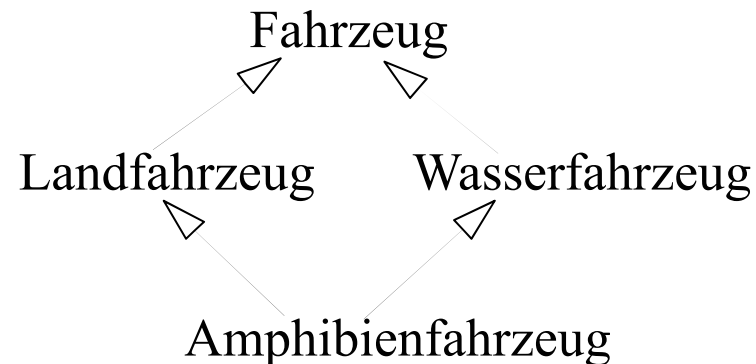
(Neuerdings kann die Intention mit spezifiziert werden, um böartige Fehler zu vermeiden. Siehe separate Präsentation.)

6.7 Einfach- und Mehrfachvererbung

Einfachvererbung: Jede Klasse hat höchstens eine direkte Oberklasse.

Mehrfachvererbung: Beliebige viele direkte Oberklassen sind zugelassen.

Beispiel für Mehrfachvererbung:



Vorteil der Mehrfachvererbung:

- Flexibilität in der Abbildung realer Gegebenheiten, z. B. um Objekte unter verschiedenen Gesichtspunkten zu betrachten.

Probleme der Mehrfachvererbung:

- Namenskonflikte ererbter Merkmale:
Was ist die Semantik, falls sowohl Land- als auch Wasserfahrzeug eine Komponente 'Motor' deklarieren?
(Mögliche Lösungen: a) illegal, b) legal, aber Aufrufe illegal, weil mehrdeutig, c) legal unter Umbenennung)
- Das "Rauten"-Problem:
Sollen Datenkomponenten, die von gemeinsamen Vorfahren der Oberklassen (im Beispiel: von Fahrzeug) ererbt wurden, doppelt oder einfach vorhanden sein?
- Implementierungsschwierigkeiten
- Unpassende Benutzersemantik der Vererbung (siehe auch das Rautenproblem)

Benutzersemantik der Vererbungsrelation:

Welche Idee soll durch die Vererbung ausgedrückt werden?

- Vererbung der Spezifikation ("*interface inheritance*", "*subtyping*")
- Vererbung der Implementierung ("*code sharing*")

Klasse als Typ(-Spezifikation) betrachtet:

Dann ist die Vererbungsrelation eine *Spezialisierungsrelation* („*is-a*“), denn die Subklasse bildet eine Spezialisierung ihrer Oberklasse.

Antwort auf Rautenproblem: einfaches Vorhandensein

Beispiel: Apfelkuchen ist spezieller Obstkuchen.

Klasse als ererbare Gruppierung von Attributen und Methoden betrachtet/
missbraucht:

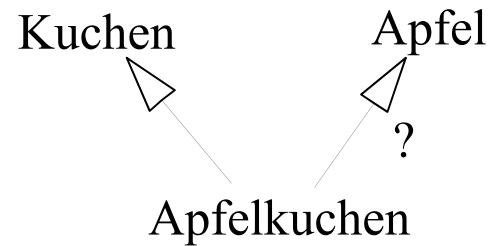
Dann wird die Vererbungsrelation zu einer *Eigenschaftsrelation* ("*has-a*"), die
zum Hinzufügen neuer Merkmale dient ("*mix-in*").

Antwort auf Rautenproblem: i. a. mehrfaches Vorkommen

Beispiel: Apfelkuchen hat Äpfel

Die "mix-in"-Semantik sollte eigentlich realisiert werden durch Daten-Attribute
vom Typ der "mix-in"-Klasse und nicht durch Mehrfachvererbung (die aber "*viel
bequemer ist*").

D. h., mit Mehrfachvererbung:



Apfelkuchen *ist* spezieller Apfel („is-a“-Beziehung)???

Eigentliche Semantik: Apfelkuchen *verwendet* Äpfel („uses“- oder "has-a"-Beziehung)

Methoden für Äpfel sind direkt auf Apfelkuchen anwendbar.

Wegen Zuweisbarkeit ist bei Referenzsemantik nun möglich:

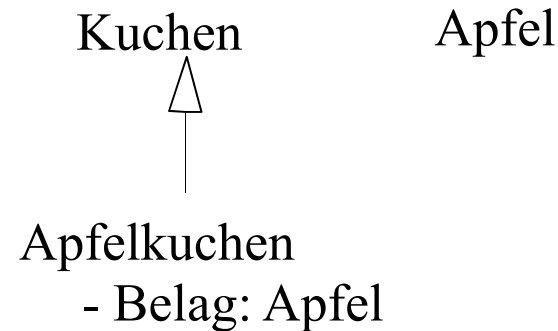
```
X: Apfelkuchen;
```

```
Y: Apfel;
```

```
Y := X; -- !!
```

d. h., hinter einem Apfel kann sich ein Apfelkuchen verbergen!

Man sollte daher besser wie folgt definieren:



Methoden für Äpfel sind nur auf die Komponente 'Belag' anwendbar, nicht auf Apfelkuchen.

(Wenn Letztere z. T. erwünscht sind, dann müssen sie als zusätzliche Methoden von Apfelkuchen spezifiziert und über die 'Belag'-Komponente realisiert werden)

Ada, Java:

- nur Einfachvererbung für das Vererben von Daten und Implementierungen zwischen Klassen (zwei kleine Ausnahmen werden unter "Interfaces" identifiziert)
- Mehrfachvererbung von Spezifikationen aus "Interfaces"

C++:

Mehrfachvererbung bei Klassen

6.8 Arten von Beziehungen zwischen Objekten/Klassen

- Vererbung bzw. Spezialisierung/Generalisierung
- Realisierung/Implementierung
 - . Spezielle Vererbungsbeziehung zwischen einem Interface und einer Klasse, die dieses implementiert
- Containment-Beziehungen (Komposition)
 - . Antwort auf die Frage: Wem gehört ein Objekt? (wichtig z. B. für Lebensdauer und Freigabe des Objekts)
- Aggregation
 - . “Besteht-Aus”-Beziehung ohne “Besitzanspruch”
- Assoziationen bzw. Referenzen
- Einfache Schnittstellenabhängigkeiten, Verwendungen

6.9 Abstrakte Methoden, abstrakte Klassen und Interfaces

- Abstrakte (virtuelle) Methode: Spezifikation der Methode einer Klasse, für die aber noch keine Realisierung angegeben werden kann, andererseits Unterklassen aber in die Pflicht genommen werden sollen, solche Realisierungen bereitzustellen.
- Abstrakte Klasse: so deklariert oder, implizit, eine Klasse mit mindestens einer abstrakten Methode
- Instanzen von abstrakten Klassen sind oft nicht sinnvoll
→ die meisten Sprachen verbieten die Erzeugung von Instanzen abstrakter Klassen (und verhindern damit auch den Aufruf der fehlenden Realisierung der abstrakten Methoden durch dynamische Bindung).
- Interfaces sind spezielle abstrakte Klassen, die keine Datenkomponenten und nur abstrakte Methoden enthalten (kleine Ausnahmen: null Rümpfe in Ada12, static constants in Java, ... aber: interessante Konflikte bei Mehrfachvererbung).

Frage: Wie wird eine abstrakte Operation spezifiziert?

- unspezifiziert lassen (weglassen) → Probleme in statisch typisierten Sprachen
- Beispiel: Abstrakte Klasse 'Window' enthalte keine Operation 'display':

```
w : Window;  
lw : LabeledWindow;  
lw := LabeledWindow.Create;  
w := lw;  
w.display;      -- nicht erlaubt, da Window keine Operation  
                -- 'display' besitzt
```

- abstrakte Operation „tut nichts“
„tut nichts“ hat andere Bedeutung als gewünschte Semantik der (von Unterklassen zu redefinierenden) abstrakten Operation.
- Operation markieren: *muss* von allen (konkreten) Unterklassen redefiniert werden
→ *deferred* in Eiffel, *subclassResponsibility* in Smalltalk, *"is abstract"* in Ada
- Spezielle Gruppierung von abstrakten Methoden in einem „Interface“; implementierende Klassen sind verpflichtet, die spezifizierten Methoden zu realisieren.

Java:

abstrakte Klassen ("abstract" als Klassen-Qualifikation)

abstrakte Methoden ("abstract" als Methoden-Qualifikation)

Interfaces und deren Methoden sind implizit abstrakt

C++:

nur implizit abstrakte Klassen

abstrakte Methoden ("pure virtual", "= 0")

Ada:

abstrakte Klassentypen ("abstract" als Typ-Qualifikation)

abstrakte Methoden ("is abstract")

Interfaces und deren Methoden sind implizit abstrakt.

6.10 OOP und herkömmliche PSen

Frage: Kann OOP (Klassen, Vererbung, dynamisches Binden) auch mit herkömmlichen Sprachen simuliert werden?

Beispiel: Ada (83)

- Simulation einer Klassenhierarchie durch variante Records

```

type Window_Kind is (Window,...Button);
type WindowSystem (W: Window_Kind) is
  record
    border: ...;
    case W is
      when Window => null;
      when LabeledWindow | Button =>
        text: myString;
        case W is
          when Window |
            LabeledWindow =>
              null;
          when Button =>
            backgroundColor: ...;
        end case;
      end case; →
    end record;

```

Mehrfach geschachtelte Varianten nötig aufgrund der Einschränkung, dass kein Komponentename - auch in verschiedenen Varianten - mehrfach vorkommen darf!

- Simulation der verschiedenen (gleichnamigen) Operationen durch *eine* gemeinsame Operation.

```
procedure display
  (Window: in out WindowSystem) is
begin
  paint (Window.border);
  case Window.W is
    when Window => null;
    when LabeledWindow | Button =>
      print (Window.text);
    case ...
      ...analog zur Typdeklaration
```

Vermeiden der Schachtelung der case-Anweisungen in 'display'
führt zu Duplizierung von Code.

Vergleich mit OO-Lösung:

- Darstellung komplex und unübersichtlich
- Erweiterung um neue Klasse führt zu Änderung sowohl der Typdefinition als auch aller zur Klasse gehörigen Operationen
 - ⇒ Neuübersetzung erforderlich
 - ⇒ Wartbarkeit sinkt!
- Erweiterung der bestehenden Typstruktur äußerst schwierig und fehleranfällig
 - ⇒ aus SE-Gesichtspunkten sehr unschön !!

Kapitel 11

Funktionale Programmierung

11. Funktionale Programmierung

Grundidee:

- Beschreibung des Programms durch Definition und Anwendung von Funktionen (und Funktionalen)
- Abwendung vom zustandsorientierten, imperativen (d. h. zuweisungsorientierten) (von-Neumann-)Modell
- Ausführungsmodell ist im Prinzip der λ -Kalkül
- Funktionen können das Resultat von Funktionsanwendungen sein

Im Vergleich ...

Imperatives (von-Neumann-)Modell:

- Sequenz von Ausführungsschritten (statements)
- Variable (~ Speicherplatz) mit änderbaren Werten
- Zuweisung
- Iteration

Funktionales Modell:

- Verknüpfung von Funktionen
- Funktionsparameter und Benennung berechneter Werte als "Konstanten" (im Sinne der imperativen Programmierung)
- keine Zuweisung (in "pure functional languages")
- Rekursion (aber keine Iteration und keine anderen Kontrollstrukturen)
- Funktionen als Parameter und Resultat von Funktionen, und damit Funktionale

11.1 Grundlagen: Funktionen, λ -Kalkül

Funktion ist Abbildung von Definitionsbereich (*domain*) in Wertebereich (*range*)

z. B.

$$\textit{plus}: \quad \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z}$$

definiert durch:

$$\textit{plus}(x, y) = x + y$$

Die Definition der Funktion 'x + y' wird dabei an den Namen 'plus' gebunden.
x und y sind Funktionsparameter.

In λ -Kalkül-Schreibweise (Präfixnotation):

$$\lambda x, y. + xy$$

Die im λ -Kalkül dargestellte Funktion ist *anonym*: Ihr wird kein Name zugewiesen. Der Punkt ('.') trennt die Parameterliste von der Funktionsbeschreibung. x und y sind *gebundene Variablen*. Auswertung im λ -Kalkül:

$$(\lambda x, y. + xy)(2, 3) \equiv +23 \equiv 5$$

Bei der Auswertung werden die Werte (Argumente) *textuell* für die Parameter eingesetzt (Substitution). (Aber: bei Konflikten, siehe -> konsistente Substitution)

"currying": Rückführung von λ -Ausdrücken mit mehreren gebundenen Variablen auf λ -Ausdrücke mit genau *einer* gebundenen Variablen (nach dem Mathematiker H. B. Curry).

Aus $\lambda x, y. + xy$
wird $\lambda x. (\lambda y. + xy)$ (*unnötige Klammern nur zur Lesbarkeit*)

Funktionen als Werte, z.B. durch Currying

z. B.

x ist freie Variable für λy -Ausdruck

$plus = \lambda x. \lambda y. +xy$

└──────────┘
Ausdruck, dessen Wert eine Funktion ist

Auswertung:

$$(\lambda x. \lambda y. +xy)(2)(3) \equiv (\lambda y. +2y)(3) \equiv + 2 3$$

Achtung: vor Substitution evtl. konsistente Ersetzung der inneren gebundenen Variablen:

$$(\lambda x. \lambda y. +xy)(y)(3) \neq (\lambda y. +yy)(3)$$

11.2 Eigenschaften der funktionalen Programmierung

Vorteile:

- Funktionen sind Funktionen im mathematischen Sinne, d. h., sie besitzen keine „Seiteneffekte“. Ohne freie Variable mit dynamischer Namensbindung besitzen sie die Eigenschaft der referential transparency: Jede Funktionsanwendung bezeichnet unveränderlich einen eindeutigen Wert, unabhängig vom Kontext, in dem sie auftritt. Damit
 - ist für das Verständnis der Funktion i. Allg. keine Kenntnis des „globalen“ Programmzustands erforderlich. (Dies ist bei der imperativen Programmierung der Fall, da zu einem beliebigen Zeitpunkt der Programmausführung der Wert einer globalen Variablen geändert werden kann und das Ergebnis einer Funktion u. U. vom Wert einer solchen Variablen abhängt).
 - ergibt Anwendung auf gleiche Parameter gleiches Resultat.

- sind Funktionen beliebig komponierbar
- können (nicht verschachtelte) Funktionsanwendungen parallel ausgeführt werden
- können die Werte freier Variablen den Aufrufen in Kopie "mitgegeben" werden wie auch die Parameterwerte (in sogenannten "Closures")
- aus Letzterem folgen sehr gute Verteilungseigenschaften
- die Programmbeschreibung ist „näher“ an einer formalen Spezifikation der gewünschten Berechnung
- leichter verifizierbar
- Programmtransformationen sind relativ einfach durchführbar.

Nachteile:

- Effiziente Implementierung auf herkömmlichen Rechnerarchitekturen schwieriger als bei prozeduralen Sprachen
- (für Echtzeit- und maschinennahe Anwendungen daher bisher kaum geeignet)
- Ausführungsmodell nur für mathematische Probleme "naheliegend"
- für einige Situationen (z. B. Ein-/Ausgabe, Zufallszahlengenerator, Uhr) ist das Fehlen veränderbaren Zustands sehr schwer zu verkraften.

11.3 Funktionale Programmiersprachen

- LISP (~60)
- Scheme (~78)*
- Common LISP* (~84) (statische Namensbindung)
- APL (~60)
- FP (78) (viele Konstruktorformen)
- ML*(77) (Typprüfungen, Typinferenzsystem und überladene Funktionsdefinitionen)
- Miranda (Turner 1985)
- Hope (Burstall 1980)
- Haskell (Peyton Jones et al., 1990) (Monads)
- F#* (MicroSoft, 2002)

* sind nicht "reine" funktionale Sprachen, sondern enthalten auch imperative Elemente.

11.4 Grundelemente funktionaler Sprachen

- Strukturen für Daten-Objekte (meist sehr eingeschränkt: Basistypen, Listen, Tupel)
- Einige primitive Funktionen: arithmetische/logische Operationen + Operationen zur Manipulation der Datenstrukturen
- Funktionale (*functional forms*) zur Schaffung/Kombination neuer Funktionen aus existierenden Funktionen
→ spezielle Form höherwertiger Funktionen (*higher order functions*): nehmen Funktion(en) als Parameter und/oder liefern Funktion als Resultat
- Die „Anwendungs“-Operation

Programmierung in funktionalen Sprachen:

- Definition neuer Funktionen mittels Funktionalen
- Möglichkeit, Funktionen zu benennen, d. h., an Namen zu binden (zur Wiederverwendung bereits definierter Funktionen)

11.5 LISP

McCarthy (1960): ursprünglich rein funktionale Sprache

Grundideen:

- Datenobjekte sind entweder
 - „Atome“ (Zahlen, Strings, Namen, Funktionen ...) oder
 - „Listen“, deren Elemente entweder Listen oder Atome sind
- Für Atome und Listen stehen primitive Funktionen zur Verfügung
- Funktionen sind Datenobjekte! (als Liste definiert)
- Die primitiven Operationen auf Funktionen erlauben die Konstruktion neuer Funktionen (durch Komposition) und die Auswertung von Funktionen.

Mit sehr wenigen Konstrukten kann eine sehr ausdrucksstarke PS konstruiert werden.

LISP-Ausführungsmodell

Funktion EVAL wird auf das Programm angewandt und hat die folgende Semantik:

$$(\text{EVAL } (A \ B \ C)) \equiv$$

apply (EVAL A) auf Parameter (EVAL B) , (EVAL C)

$$(\text{EVAL } A) \equiv \text{Wert von A (falls A Name oder Atom ist)}$$

Primitive Listenoperationen in LISP

- `(QUOTE x)` ergibt als Wert 'x'

Abkürzung: 'x

- `(CAR ' (A B C))` ergibt A (1. Element der Liste)
`(CDR ' (A B C))` ergibt (B C) (Liste minus 1. Element)

wegen der Häufigkeit dieser Operationen auch

`CAAAR x = (CAR (CAR (CAR x)))`

`CAD..ADR x = (CAR (CDR .. (CAR (CDR x))))`

- `(CONS 'A ' (B C))` ergibt (A B C)
- `(CONS ' (A B) ' (C D))` ergibt ((A B) C D)
- `(NULL x)` ergibt 'T', wenn `(EVAL x) '()` ergibt
'() sonst
- '() ist benannt NIL, d. h. `(EVAL NIL)` ergibt '()
- obige Quotes sind vermeidbar: `CONS 'A ' (B C) == cons [A (B C)]`

LISP-Funktionswerte

- haben die Form
(LAMBDA param-liste definition)
 ↑
Konstruktor für Funktionen
- können (wie alle anderen Werte) benannt werden:

```
(DEFINE  
  Wert von CDAR  
  ' (CDAR (LAMBDA (list)  
          (CDR (CAR list))  
        )))
```

Anwendung:

```
(CDAR ' ((A B C) D) ≡ apply (LAMBDA ...) ' ((A B C) D)  
≡ (CDR (CAR ' ((A B C) D)))  
≡ (CDR ' (A B C))  
≡ ' (B C)
```

„COND“

COND

```
(bed1  ausdruck1)
(bed2  ausdruck2)
      :
(bedn  ausdruckn)

)
```

Semantik:

EVAL (COND...) ergibt

```
if    (EVAL bed1) = T dann (EVAL ausdruck1)
elseif      (EVAL bed2) = T dann (EVAL ausdruck2)
:
elseif      (EVAL bedn) = T dann (EVAL ausdruckn)
else      ' ()
```

Beispiel:

```
(DEFINE ' (
  fac (LAMBDA (n)
    (COND
      ((EQ n 0) 1)
      (T (TIMES n (fac (SUB1 n))))))
  )
)
```

fac[4] entspricht (FAC 4) und ergibt 24

LISP-Dialekte

- Hinzufügung weiterer Funktionale (außer Komposition), z. B. MAPCAR (entspricht α in FP)
- Hinzufügung imperativer, nicht-applikativer Elemente wie Zuweisung (SET, SETQ), Sequenz (PROG) und Manipulation von (bestehenden) Datenstrukturen (RPLACA = replace car) aus Effizienzgründen
- statische statt dynamische Namensbindung (Common-Lisp)

11.6 FP

J. Backus (Anfang 70er): rein funktionale Sprache

- Datenstrukturen bestehen aus Atomen (Zeichenfolge) bzw. Folgen von Atomen
- viele Basisfunktionen (z. B. tail, equals, reverse, length,...)
- Anwendungsoperator ':'
- fixe Menge von Funktionalen zur Erzeugung neuer Funktionen aus bestehenden basis- bzw. benutzerdefinierten Funktionen, d. h., (basis- und benutzerdefinierte) Funktionen können keine Funktionen als Parameter erhalten bzw. als Resultat liefern.

Gründe:

"Unbeschränkte Freiheit, Funktionen kombinieren zu können, führt zu unübersichtlicher Programmiersprache und unübersichtlichen Programmen."

- Sorgfältige Wahl der Funktionale ermöglicht einfache Transformationen und Korrektheitsbeweise von FP-Programmen. Transformationsregeln können ebenfalls durch FP-Syntax beschrieben werden (d. h., keine Metasprache erforderlich!).

Funktionale in FP (Auswahl):

1. Komposition:

$$(f \circ g) : x \equiv f : (g : x)$$

2. Konstruktion:

$$[f_1, f_2 \dots f_n] : x \equiv (f_1 : x \dots f_n : x)$$

3. All-Anwendung

$$\alpha f : (x_1 \dots x_n) \equiv (f : x_1 \dots f : x_n)$$

4. Bedingte Form

$$(\text{IF } p \text{ f } g): x \equiv \text{if } p:x = T \text{ then } f:x \text{ else } g:x$$

5. While-Iteration:

$$(\text{WHILE } p \text{ f}):x \equiv \text{if } p:x = T \text{ then } (\text{WHILE } p \text{ f}):(f:x) \text{ else } x$$

6. Einfügen (vereinfacht, für $n \geq 2$)

$$/f:(x_1 \dots x_n) \equiv f:(x_1, /f:(x_2, \dots, x_n))$$

11. Transposition

$$\begin{aligned} \text{trans}:((x_{11}, \dots, x_{1n}) \dots (x_{n1}, \dots, x_{nn})) \\ \equiv ((x_{11}, \dots, x_{n1}) \dots (x_{1n}, \dots, x_{nn})) \end{aligned}$$

Beispiel: Skalarprodukt

$$\text{DEF } IP \equiv (/+)^\circ (\alpha *)^\circ \text{trans}$$

Anwendung auf eine Liste ((1,2,3),(4,5,6)):

$$\begin{aligned} (/+)^\circ (\alpha *)^\circ \text{trans: } & ((1, 2, 3), (4, 5, 6)) \\ = & (/+)^\circ (\alpha *) : ((1, 4), (2, 5), (3, 6)) \\ & = (/+) : (4, 10, 18) \\ = & +: (4, +: (10, +: (18))) \\ & = +: (4, +: (10, 18)) \\ & = +: (4, 28) \\ & = 32 \end{aligned}$$

11.7 „Delayed Evaluation“, „Lazy Evaluation“

- Am Beispiel 'COND' in LISP sieht man, dass nicht zwangsläufig alle Parameter ausgewertet werden müssen.
- Beobachtung zum Prinzip erheben: Parameter müssen erst ausgewertet werden, wenn ihr Wert benötigt wird
→ „lazy evaluation“

Vorteile:

- vermeidet überflüssige Auswertung in vielen Fällen
- ermöglicht Arbeiten mit unendlichen Datenstrukturen (ML, Miranda, Haskell): Realisierung: Statt Wert-„Eintrag“ im Datenobjekt Eintrag der Funktion, die jeweils im Bedarfsfall den (nächsten) Wert berechnet.
- gewährleistet Terminierung des Programms, auch wenn nicht benötigte Parameter (-Ausdrücke) nicht terminieren würden
- in Scheme über 'Delay' und 'Force' kontrollierbar

Nachteile:

- schlecht für Argumente verteilt ausgeführter Aufrufe

11.8 Haskell 98 – eine typsichere funktionale Sprache

Haskell ist ein vergleichsweise reiches Typmodell und ist insbesondere typsicher gestaltet. Haskell hat

- ein Klassenkonzept, dessen Instanzen(!) die Typen sind; z. B. gibt es die Klasse Num ("Numbers") für alle numerischen Typen.
- vordefinierte Typen wie Integer, Char, Bool, Float, Double, etc.
- Enumerationstypen, deren Elemente null-äre Funktionen sind:
`data Color = Red | Yellow | Green`
- Benutzerdefinierte parametrisierte Typen (der jeweilige Name wird zum Typkonstruktor), deren Elemente konstruiert werden:
`data Point a = Pt a a` – Pt ist der Instanz-Konstruktor (Pt x y ergibt (x,y) als Point)
`Point Integer` -- ein konkreter Typ für Integer-Koordinaten
- Funktionstypen: `fib :: Integer -> Integer`
- gebundene "Typ variablen" für polymorphe Typen: `a, b, beliebig`
- Listen: `[Float], [a]` -- homogene Komponenten
- Tupel: `(a, b, c, b, d)` -- heterogene Komponenten (beachte b!)

11.9 Funktionsdefinition durch Mustererkennung (Haskell)

Anstatt eine Funktion in geschlossener Form zu definieren:

```
fib :: Integer -> Integer
fib x
    | x > 2 = fib(x-2) + fib(x-1)
    | otherwise = 1
```

sieht Haskell auch die Spezifikation durch ein oder mehrere Muster vor, die in der Reihenfolge der Aufschreibung geprüft werden, z. B. für atomare Argumente

```
fib 0 = 1
fib 1 = 1
fib x = fib(x-2) + fib(x-1)
```

oder für Listen, wobei [] die leere Liste ist und h:t head (1. Element) und tail (den Rest) der Liste denotiert:

```
allprimes [] = []
allprimes (h:t) = if prime h then h: allprimes t
                  else allprimes t
```

If-then-else ist Haskell-Syntax für die Anwendung der Funktion "cond".

11.10 Typklassen (Haskell)

Definition einer Typklasse (nicht zu verwechseln mit OOP-Klassen):

```
class BasicEq a where
    isEqual :: a -> a -> Bool
    -- evtl. mit Default-Implementierung hier
```

Alle Typen, die sich zur Instanz der Typklasse BasicEq erklären, haben die Funktion isEqual.

```
instance BasicEq Color where
    isEqual Red Red = True
    isEqual Green Green = True
    isEqual Blue Blue = True
    isEqual _ _ = False    -- "_" steht für "beliebig"
```

Ab sofort hat der Typ Color die Funktion isEqual; die Color-spezifische Implementierung wird hier angegeben.

11.11 Typinferenz (Haskell)

Haskell erlaubt aber fordert im Allgemeinen keine Typangaben. Ohne Angaben inferiert Haskell den Typ von Ausdrücken und Funktionen. Falls

```
prime :: Integer -> Bool
```

getypt ist, dann inferiert Haskell (das nur Typ-homogene Listen hat) aus

```
allprimes [] = []  
allprimes (h:t) = if prime h then h: allprimes t  
                  else allprimes t
```

den Typ von allprimes:

```
allprimes :: [Integer] -> [Integer]
```

Für

```
double x = x + x
```

inferiert Haskell den Typ

```
(Num a) => a -> a
```

d. h., `x` und `double` sind polymorph, aber eingeschränkt auf num(erische) Typen.

11.12 "List comprehension" in Haskell

Ein Ausdruck `"[f x | x <- xs]"` heißt "list comprehension". Der `<-` Operator ist ein sogenannter "Generator". Er beliefert `f x` mit sukzessiven Werten aus der Liste `xs`, sodass mit den Resultaten eine neue Liste entsteht. Eine list comprehension kann mehrere Generatoren beinhalten.

```
crossproduct x y = [ (a, b) | a <- x, b <- y ]
```

erzeugt das Kreuzprodukt aus den Listen `x` und `y` (dabei läuft `y` am schnellsten).

```
crossproduct [1, 2] [1, 3] == [ (1, 1), (1, 3), (2, 1), (2, 3) ]
```

Filter durch "guards" sind möglich:

```
pairs x y = [ (a, b) | a <- x, b <- y, a /= b ]
```

ist das auf unterschiedliche Tupel beschränkte Kreuzprodukt.

```
pairs [1, 2] [1, 3] == [ (1, 3), (2, 1), (2, 3) ]
```


11.13 Partielle Auswertung (Currying) in Haskell

Mit

```
mult x y = x * y
```

(Inferierter Typ: `Num(a) => a -> (a -> a)`)

kann man definieren:

```
triple = mult 3    -- es ergibt sich: \y -> 3 * y  -- (\ für lambda)
incr   = add 1      -- analog
```

```
triple incr 5 == 18
incr triple 5 == 16
```

Partielle Auswertung ist eine der Möglichkeiten, Funktionen zu konstruieren.
Es gibt viele andere, insbesondere über vordefinierte Funktionale.

11.14 Monads in Haskell

Ein Monad ist eine spezielle Kapselung für Zustandsveränderungen, sodass für den Rest der Sprache die Eigenschaft der referentiellen Transparenz und deren Vorteile erhalten bleiben. Wesentliches Element ist dabei, die Ordnung der Zustandsveränderungen durch "chaining" und "injection" festzulegen.

```
class Monad m where
```

```
  -- chaining:
```

```
    (>>=) :: m a -> (a -> m b) -> m b
```

```
    (>>) :: m a -> m b -> m b
```

```
    alternativ: sequencing syntax "do
```

```
        x
```

```
        y" statt "x >> y"
```

```
  -- injecting:
```

```
    return :: a -> m a
```

Selbst die Designer der Sprache halten Monads für schwer erklärbar.

... und daher will ich das hier auch nicht versuchen.

Kapitel 12

"Logic Programming"

12. "Logic Programming"

Idee: Es sollen die Eigenschaften des zu lösenden Problems in prädikatenlogischer Formulierung spezifiziert werden, d. h., „was soll gelöst werden“. Das „Wie“ der Lösung soll vom Ausführungsmodell bestimmt werden.

Primäres Beispiel: PROLOG (~72)

Ursprünge: Automatisierte Beweisführung (speziell Ansätze von Robinson)

12.1 (Prädikaten-)Logische Grundlagen

Prinzip der (einfachen) Resolution:

Wenn

$$P_1 \subset P_2$$

$$Q_1 \subset Q_2$$

gilt und $P_1 = Q_2$ ist, dann gilt

$$Q_1 \subset P_2$$

Beispiel:

$$\text{klüger(Anna,Maria)} \subset \text{älter(Anna,Maria)}$$

$$\text{älter(Anna,Maria)} \subset \text{Mutter(Anna,Maria)}$$

$$\text{klüger(Anna,Maria)} \subset \text{Mutter(Anna,Maria)}$$

12.2 Prinzip der Resolution mit Instantiierung und Unifizierung

Wenn $\forall x \forall y P_1(x, y) \subset P_2(x, y)$

$$Q_1(a, b) \subset P_1(a, b)$$

dann $Q_1(a, b) \subset P_2(a, b)$

Hierbei wird x mit a , y mit b instantiiert, so dass $P_1(x, y)$ mit $P_1(a, b)$ unifiziert werden kann.

Beispiel:

$$\forall x \forall y \text{ älter}(x, y) \subset \text{Mutter}(x, y)$$

$$\text{klüger}(\text{Anna}, \text{Maria}) \subset \text{älter}(\text{Anna}, \text{Maria})$$

$$\text{klüger}(\text{Anna}, \text{Maria}) \subset \text{Mutter}(\text{Anna}, \text{Maria})$$

Wenn

$$\forall x \forall y P_1(x, y) \subset P_2(x, y)$$

$$\forall a \forall b Q_1(a, b) \subset P_1(a, b)$$

dann

$$\forall a \forall b Q_1(a, b) \subset P_2(a, b)$$

d. h., auch ungebundene Variablen können unifiziert werden.

Beispiel:

$$\forall x \forall y \text{älter}(x, y) \subset \text{Mutter}(x, y)$$

$$\forall a \forall b \text{klüger}(a, b) \subset \text{älter}(a, b)$$

$$\hline \forall a \forall b \text{klüger}(a, b) \subset \text{Mutter}(a, b)$$

(+ entsprechende Regeln mit Instantiierung + Unifizierung)

12.3 „Horn-Klauseln“

Horn-Klauseln sind eine restriktive Form prädikatenlogischer Aussagen:

Conclusio \subset Prämisse

1. Als Conclusio ist nur ein Term zugelassen
2. Als Prämisse sind nur Konjunktionen von Termen zugelassen (auch die leere Konjunktion); d. h., zugelassen sind ...

$P [\subset t r u e]$ "Faktum"

$P \subset Q_1 \wedge Q_2 \wedge \dots \wedge Q_n$ Q_i atomare Terme

Resolution ist für logische Systeme in Form von Horn-Klauseln besonders einfach zu realisieren. In der Aufschreibung können Quantoren weggelassen werden mit der impliziten Annahme, dass Variablen, die in der Prämisse auftreten, universell und Variablen, die nur in der Conclusio auftreten, existentiell qualifiziert sind.

12.4 ... ein wenig Prolog-Syntax

„Regeln“: in der Form von Horn-Klauseln

z. B.

älter (anna, maria). -- ein Faktum

älter (X, Y) :- mutter (X, Y).

Vereinbarung:

Großschreibung = Variable

Kleinschreibung = Konstante, Prädikatname

Konjunktion:

durch ',' in der Prämisse

z. B.

$P(X) :- Q(X), R(X). \quad -- \forall x : P(X) \subset Q(X) \wedge R(X)$

Listen: wie in LISP

in einigen Prolog-Dialekten „gezuckert“:

$[a,b,c] \equiv \text{cons}(a, \text{cons}(b, c))$

auch:

$[a, b, L] \equiv \text{cons}(a, \text{cons}(b, L))$

„gcd” in PROLOG

durch prädikative Beschreibung:

(*) `gcd(u, 0, u) .`

(**) `gcd(u, v, w) :-
 not zero(v), gcd(v, u mod v, w) .`

„Berechnung“ durch Resolution (interaktive Fragestellung):

```

                                (*)  gcd(u, 0, u) .
                                (**) gcd(u, v, w) :-
:- gcd (15, 10, X)                not zero(v), gcd(v, u mod v, w) .
```

Resolution mit (*) schlägt fehl.

Resolution mit (**) gibt:

```
:- not zero(10), gcd(10, 15 mod 10, X)
```

...mit weiteren Regeln für zero:

```
:- gcd(10, 5, X)
```

```
:- not zero(5), gcd(5, 10 mod 5, X)
```

...

```
:- gcd(5, 0, X)
```

Resolution mit (*) erfolgreich.

Antwort: Ja, X=5

12.5 Prolog-Implementierung

Implementierungsentscheidung:

- depth-first im Alternativenbaum
- sequentiell über den Regeln

⇒ Notwendigkeit von Backtracking

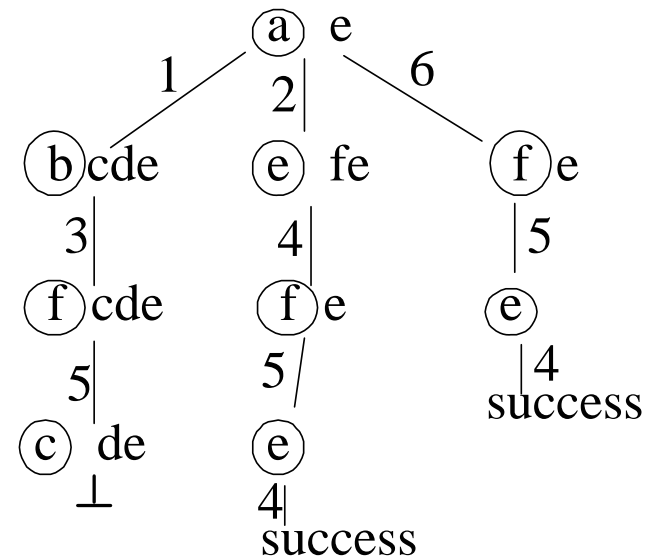
Prinzipielles Ausführungsmodell (ohne Unifizierung)

```
procedure solve(L: pList);  
begin  
  local i: integer;  
  if L ≠ nil then  
    for i over {Rules} do  
      if match(head(Rule[i]), head(L)) then  
        solve(append(tail(Rule[i]), tail(L)));  
      else  
        write("Ja"); stop  
      end;  
    end;  
  end;
```

Beispiel:

1. $a :- b, c, d.$
2. $a :- e, f.$
3. $b :- f.$
4. $e.$
5. $f.$
6. $a :- f.$

Frage: $:-a, e$



Anm.: Um die exponentielle Explosion der vollständigen Untersuchung des gesamten Entscheidungsbaums zu vermeiden, hat Prolog den 'cut' Operator, der (als Prämisse) den Entscheidungsast an dieser Stelle abschneidet (auch wenn dieser erfolgreich sein könnte).

Ausführungsmodell mit Unifikation

```
procedure solve(L, env: pList);
begin
    local    i: integer;
             newenv: pList;
    if L ≠ nil then
        for i over {Rules} do
            begin
                newenv := unify(copy(head(Rule[i])), head(L), env);
                if newenv ≠ nil then
                    solve(append(copy(tail(Rule[i])), tail(L)), newenv)
                end
            end
        else
            printenv(env); suspend
        end;
end;
```

Falls mehrere Antworten möglich sind...

Regeln:

```
dichter(goethe) .  
dichter(schiller) .
```

Frage:

```
:- dichter(X)
```

Antwort:

```
Ja, X = goethe  
      -- hier kann der Benutzer nun das Fortfahren veranlassen  
Ja, X = schiller
```


„Append” in PROLOG

Regeln:

```
append([], Y, Y) .
```

```
append([A|B], Y, [A|W]) :- append(B, Y, W) .
```

Frage:

```
:- append([a,b], [c], Z)
```

Antwort:

```
Z=[a,b,c]
```

Frage:

```
:- append(X, Y, [a,b])
```

Antwort:

```
X=[]      Y=[a,b]
```

```
X=[a]     Y=[b]
```

```
X=[a,b]   Y=[]
```

Problem: Endlose Rekursion

```
(*) vorfahr(X,Y) :- vorfahr(Z,Y), vater(X,Z).  
    vorfahr(X,Y) :- vater(X,Y).  
    vater(hans,josef).
```

Frage:

```
:- vorfahr(X,josef)
```

Problem: (*) trifft immer zu! ⇒ unendliche Rekursion

In PROLOG-Implementierungen ist die Reihenfolge der Regeln

- wichtig für Effizienz
- entscheidend für Terminierung

Anm: für Rekursionen, immer besser "tail"-Rekursion

Das 'NOT'-Problem

```
vater(hans, maria) .  
gattin(anna, hans) .
```

Frage:

```
:- not mutter(anna, maria)
```

Antwort:

Ja !!

d. h., Negierung einer nicht beweisbaren Aussage wird als wahr interpretiert.

⇒ nicht zu unterscheiden von einer Negierung einer beweisbar falschen Aussage

Problem der „Closed-World“-Annahme

Kapitel 13

Parallel-Verarbeitung

13. Parallel-Verarbeitung

13.1 Anmerkungen zum Thema "Non-Determinismus"

Non-Determinismus entsteht z. B. durch

- Auswertungsreihenfolge in Ausdrücken
- Rundung
- explizit nicht-deterministische Konstrukte, z. B. guarded commands
- Parameterübergabe-Mechanismus

Diese Fälle sind typischerweise auf Sprachebene nicht-deterministisch, aber auf Implementierungsebene trotzdem determiniert.

- Parallelität
wenn parallele Aktivitäten gegenseitige Abhängigkeiten haben,
ergeben sich u. U. (ausführungszeit-)abhängige Resultate.

In diesem Fall ist das Verhalten weder auf Sprachebene noch auf Implementierungsebene deterministisch.

13.2 Parallelität „im Kleinen“

1. Kollateral-Konstrukt ', ' (Algol68):

S_1, S_2, S_3

Die drei Anweisungen können in beliebiger Reihenfolge (evtl. auch parallel) ausgeführt werden. Durch ";"

$S_1; S_2; S_3;$

wird dagegen die sequentielle Ausführung veranlasst.

Achtung: Datenabhängigkeiten! Z. B.

$n := 7; n := n + 1, n := 17$

danach: $n = 8$ oder 18 oder 17

Die Anweisungen

$n := n + 1, m := m + 2$

sind zwar deterministisch, aber es gibt dann auch semantisch keinen beobachtbaren Unterschied zwischen ', ' und ';'.

2. "cobegin"-Konstrukt:

```
cobegin      -- Semantik wie oben  
  S1;  
  S2;  
  S3;  
coend;
```

13.2.1 Das „forall“-Konstrukt

Motivation: Parallelisierung von Schleifen

```
for i := 1 to 20 loop  
    A(i) := B(i) + C(i) ; -- kein Aliasing angenommen  
end loop;
```

In diesem Fall ist nicht beobachtbar, in welcher Reihenfolge (evtl. auch parallel) A(1)...A(20) berechnet wurden.

⇒ Grundlage für parallelisierende Optimierer (z. B. für Vektor- oder Feldrechner)

Probleme:

- Datenunabhängigkeit oft nicht vom Compiler entscheidbar
- Existierende Datenabhängigkeiten können irrelevant für den Algorithmus sein
- (Ausnahmesemantik)

⇒ „Forall“-Konstrukt zum Ausdruck erwünschter oder erzwungener logischer Parallelität.

Für Schleifen kann

- sequentielle
- asynchron-parallele
- synchron-parallele

Semantik postuliert werden ...

Beispiel:

```
A := "informatik";
```

Sequentielle Semantik:

```
for i := 1 to A'Last - 1 loop  
    A(i+1) := A(i);  
end loop;  
⇒ A = "iiii...i"
```

Asynchrone Semantik:

```
forall i in 1 to A'Last - 1 loop  
    A(i+1) := A(i);  
end loop;  
⇒ A = ?    -- viele Möglichkeiten
```

Synchrone Semantik:

```
forall i in 1 to A'Last - 1 loop  
    A(i+1) := A(i);  
end loop;  
⇒ A = "iinformati" -- garantiert!
```

13.3 Parallelität „im Großen“

Quasi-parallel:

Mehrere Programmeinheiten sind gleichzeitig aktiv, aber zu jedem Zeitpunkt ist nur jeweils eine Einheit in der Ausführung. Die Kontrollübergänge zwischen den Einheiten sind durch die Sprachsemantik festgelegt.

=> Coroutinen

Physikalisch parallel:

Mehrere Programmeinheiten sind gleichzeitig in der Ausführung begriffen (und synchronisieren und kommunizieren gelegentlich miteinander).

=> wichtig für den Sprachimplementierer

=> für Benutzer und Sprachdefinition identisch zu logisch parallel (s.u.)

Logisch parallel:

Mehrere Programmeinheiten können gleichzeitig in der Ausführung begriffen sein. Die Implementierung kann sowohl physikalisch parallele als auch quasi-parallele Ausführung realisieren.

Parallele Programmeinheiten:

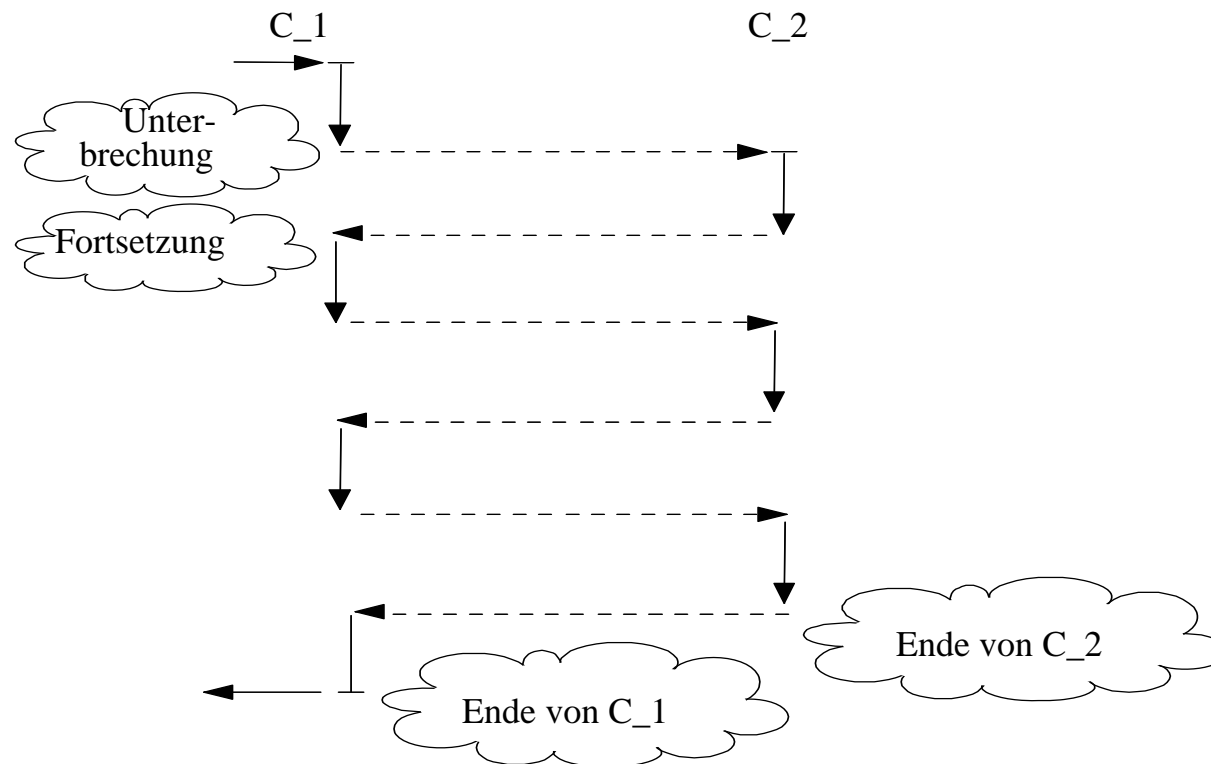
- Coroutinen - nur quasi-parallel (→ Simula)
- speziell ausgezeichnete Prozeduren (→ Modula, PL/I)
- spezielle Konstrukte: „tasks“ (→ Ada)
- (BS-Prozesse)

13.3.1 Coroutinen

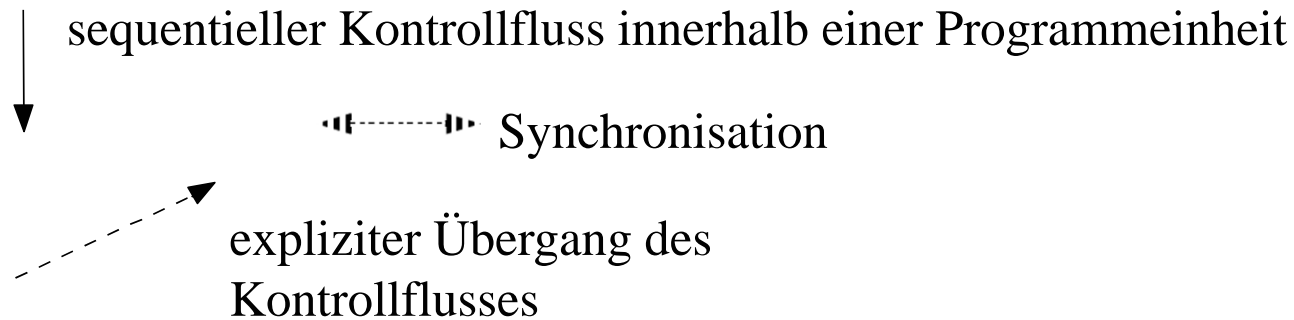
Idee:

- Symmetrische Programmeinheiten, die sich gegenseitig aufrufen.
- Fortsetzung erfolgt stets an der Stelle der letzten Unterbrechung.

Prinzip:

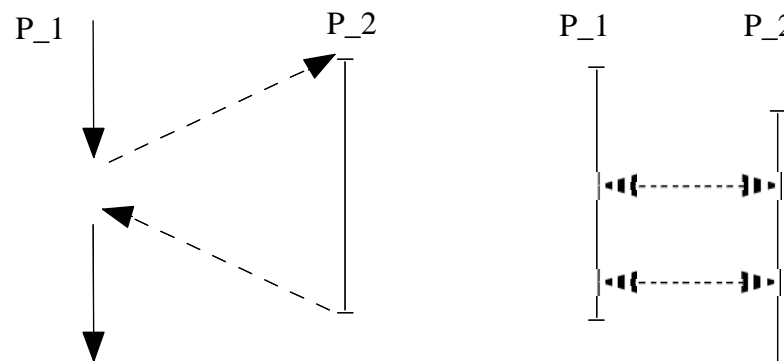


Legende:



Dieses Prinzip steht im Gegensatz zu
Unterprogramm-
Technik

parallele Prozesse



13.3.2 Coroutinen in SIMULA 67

- Operationen:
 - *detach*: suspendiert die Ausführung einer Coroutine (nach der Initialisierung)
 - *resume*(C): setzt die Ausführung der Coroutine C an der Stelle der letzten Unterbrechung fort
- Beispiel (Erzeuger-Verbraucher-Problem)

```
class producer (consumerptr);  
  ref (consumer) consumerptr;  
  begin  
    integer stuff;  
    detach;  
    while true do  
      begin  
        .  
        .    ****  erzeuge stuff  ****  
        .  
  
        buf := stuff ;  
        resume (consumerptr);  
      end  
    end;  
end;
```

```

class consumer (producerptr);
  ref (producer) producerptr;
  begin
    integer value;
    detach;
    while true do
      begin
        value := buf;
        .
        .      **** verbrauche value ****
        .
        resume (producerptr);
      end
    end;
  **** Master-Programmeinheit ****
begin
  integer buf;
  ref (producer) prod1;
  ref (consumer) cons1;
  prod1 := new producer(cons1);
  cons1 := new consumer(prod1);
  resume (prod1)
end;

```


13.3.3 Implementierung von Coroutinen

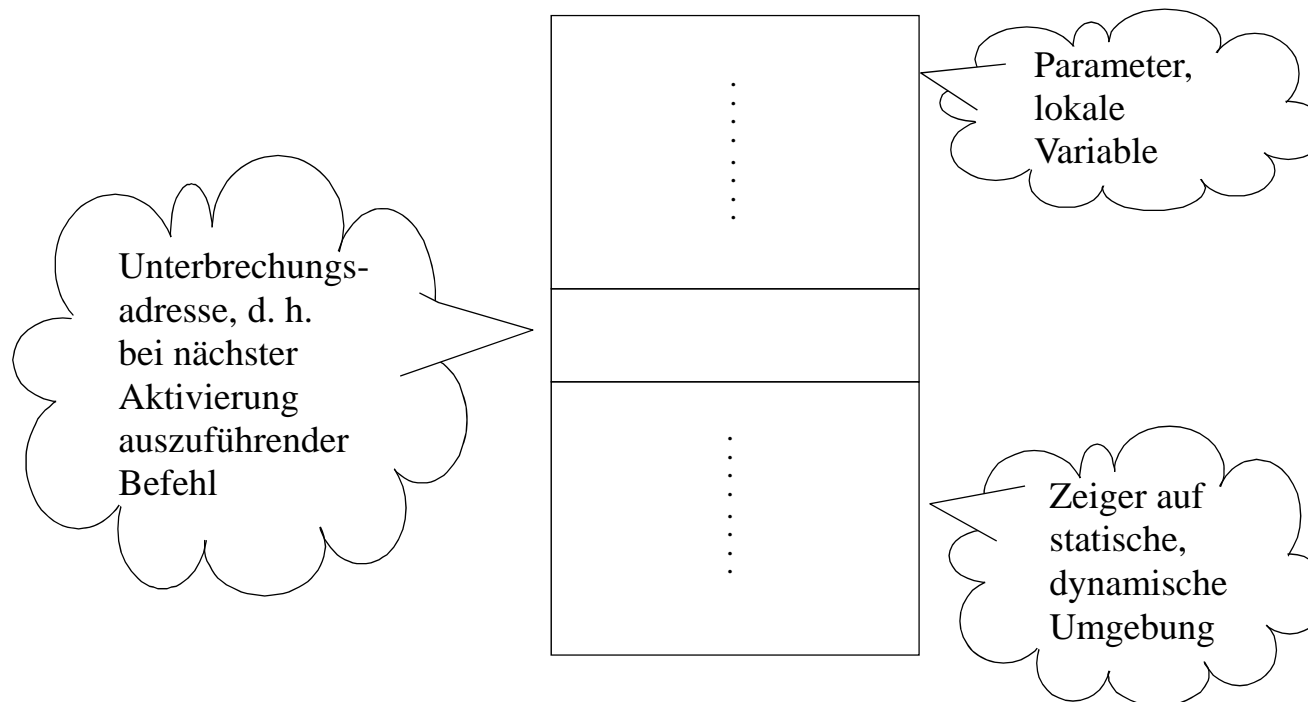
Beobachtung: *Ein* Laufzeitkeller für Aktivierungsblöcke genügt nicht.

Gründe:

- Das Verlassen einer Coroutine erlaubt es nicht, ihren AB zu zerstören.
- Die Coroutine kann ihrerseits andere Programmeinheiten (z. B. Unterprogramme) aktivieren.

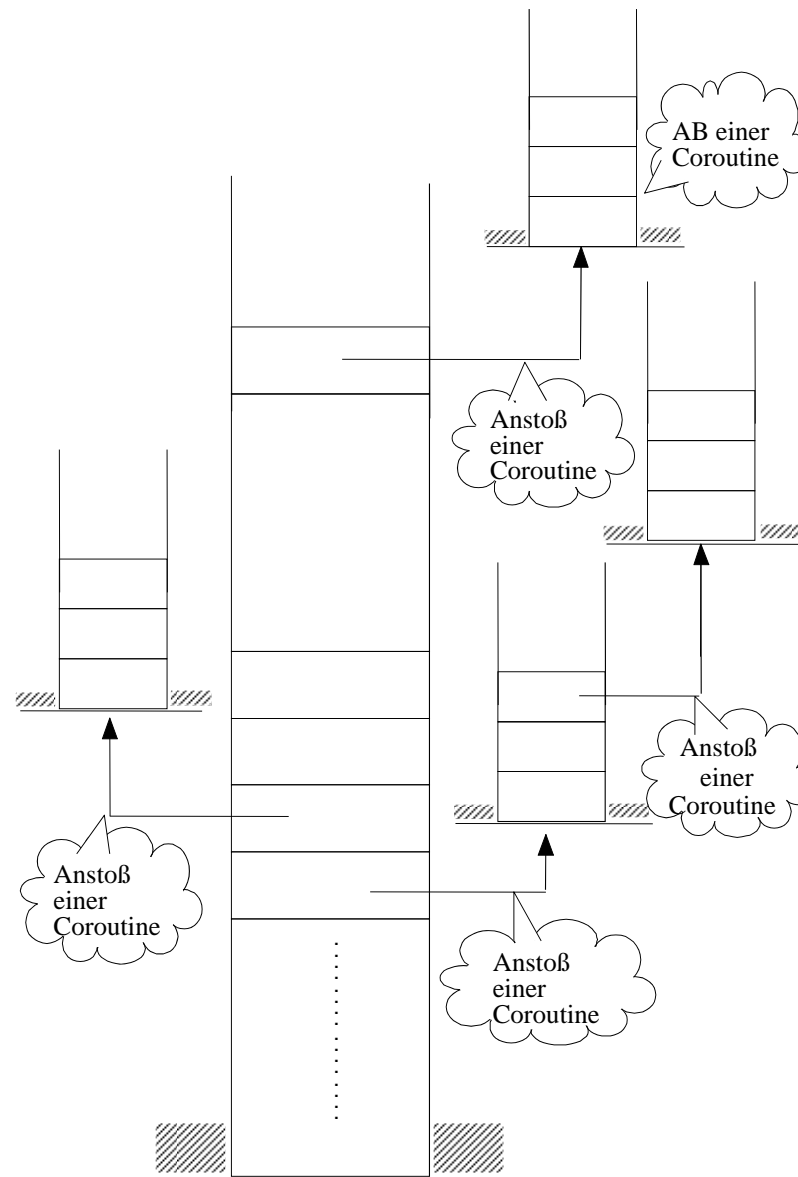
Daher ...

- bei Wechsel der Coroutine Rettung des Zustands und der Unterbrechungsadresse im Coroutinen-Aktivierungsblock:



- sowie Abkehr von der reinen Stapelverwaltung der Aktivierungsblöcke,
...

Lösung: Kaktus-Stapel:



13.3.4 Semaphor

- binärer Semaphor
- allgemeiner Semaphor

Der allgemeine Semaphor ist eine Datenstruktur der Form

record

 ZAEHLER : 0..MAX;

 SCHLANGE : Liste von Prozess-IDs;

end

Operationen für var S : SEMAPHORE

$P(S) \equiv$ **if** S.ZAEHLER > 0 **then**
 S.ZAEHLER := S.ZAEHLER-1
else reihe den P ausführenden Prozess in S.SCHLANGE **ein**

$V(S) \equiv$ **if** S.SCHLANGE nicht leer **then**
 befreie einen Prozess aus S.SCHLANGE
else S.ZAEHLER := S.ZAEHLER+1

Programmiersprachen mit Semaphoren

- ALGOL 68 ("down" and "up")
- Modula-2-Erweiterungen ("wait" and "send"/"signal")
- PL/I mittels „events“ (~ binärer Semaphor)
 P-Operation → WAIT (. . . event(s) . . .)
 V-Operation → COMPLETION (. . . event . . .)
- usw.

Beispiel:

```
semaphore access, fullspots, emptyspots;  
access.count := 1;  
fullspots.count := 0;  
emptyspots := BUFLen;  
  
process producer;  
loop                -- produce VALUE --  
    wait (emptyspots); {wait for a space}  
    wait (access);      {wait for access}  
    ENTER (VALUE);  
    signal (access);    {relinquish access}  
    signal (fullspots); {increase filled spaces}  
end loop;  
end producer;
```

```

process consumer;
loop
    wait (fullspots);    {make sure it's not empty}
    wait (access);      {wait for access}
    REMOVE (VALUE);
    signal (access);    {relinquish access}
    signal (emptyspots); {increase empty spaces}
end loop;
end consumer;

```

(Benutzer-)Probleme mit Semaphoren:

- fehlende P-Operation(en)
 - └─► gegenseitiger Ausschluss nicht gewährleistet
- fehlende V-Operation(en)
 - └─► Verklemmung(en)

13.3.5 Strukturierte Kommunikation/Synchronisation

- „conditional critical regions” (Hoare, Hansen)
- Monitore (Concurrent Pascal, Modula, Mesa) ~77
- Protected Types (Ada9X) ~92
- Rendezvous (Occam, Ada) ~79

13.3.6 Conditional Critical Regions

Idee: Eintritt in einen "kritischen Teil" der Ausführung ist nur jeweils einem Prozess möglich und kann zusätzlich von einer Bedingung abhängig gemacht werden.

Semantik für „conditional critical regions“:

"region X do" assoziiere X mit binärem Semaphor X_s
 bei Eintritt: $P(X_s)$
 bei Austritt: $V(X_s)$

„await C“ suspendiere Prozess und $V(X_s)$, falls $C \neq \text{true}$, sonst fahre fort.

für alle suspendierten Prozesse: wenn $C = \text{true}$ wird*, $P(X_s)$ und erneutes „await C“.

Probleme:

1. Wie und wann erfolgt die erneute Prüfung von C?
2. „Race condition“ am Punkt (*) beachten!
d. h., zwischen $C = \text{true}$ und $P(X_s)$ kann ein anderer Prozess erfolgreich
„ $P(X_s); C := \text{false}; V(X_s)$ “ ausgeführt haben
⇒ erneutes „await C“ erforderlich!

zu 1. "Spin-Lock" als Implementierung - ist aber nur sinnvoll bei physikalischer Parallelität.

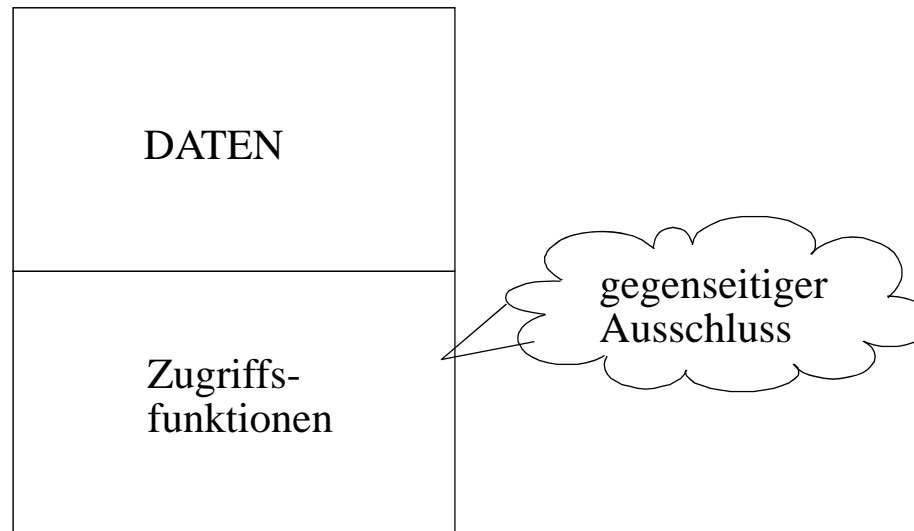
Beispiel:

```
type MessageBuffer =  
  shared record  
    size, front, rear : 0..capacity;  
    items : array[1..capacity] of Message  
  end;  
  
procedure sendmessage(newitem: Message;  
  var buffer: MessageBuffer);  
  
  begin  
    region buffer do  
      begin  
        await buffer.size < capacity;  
        buffer.size := buffer.size + 1;  
        buffer.rear := buffer.rear mod capacity + 1;  
        buffer.items[buffer.rear] := newitem  
      end  
    end;  
  end;
```

```
procedure receivemessage(var olditem: Message;  
                        var buffer: MessageBuffer);  
  
  begin  
    region buffer do  
      begin  
        await buffer.size > 0;  
        buffer.size := buffer.size - 1;  
        olditem := buffer.items[buffer.front];  
        buffer.front := buffer.front mod capacity + 1  
      end  
    end;
```

13.3.7 Monitore

Idee: Abstrakter Datentyp mit Zugriffsfunktionen, deren Ausführung nicht gleichzeitig erfolgen kann



Monitore in Concurrent Pascal:

```
type MONI = monitor  
    ...  
    Monitor-Daten  
    ...  
    procedure entry EINS ....  
    procedure entry ZWEI ....  
    ...  
begin ... Initialisierung ... End;  
  
var PUFFER : MONI
```

- Durch Anweisung
 init PUFFER
wird Puffer initialisiert.

- Aufruf einer Monitoroperation, z. B.
PUFFER.EINS
- Abstimmung der Zugriffsprozeduren EINS, ZWEI, . . . geschieht mittels
 - Variablen von Typ „queue“ (als Daten des Monitors)
 - Operationen **delay(...)**
 continue(...)

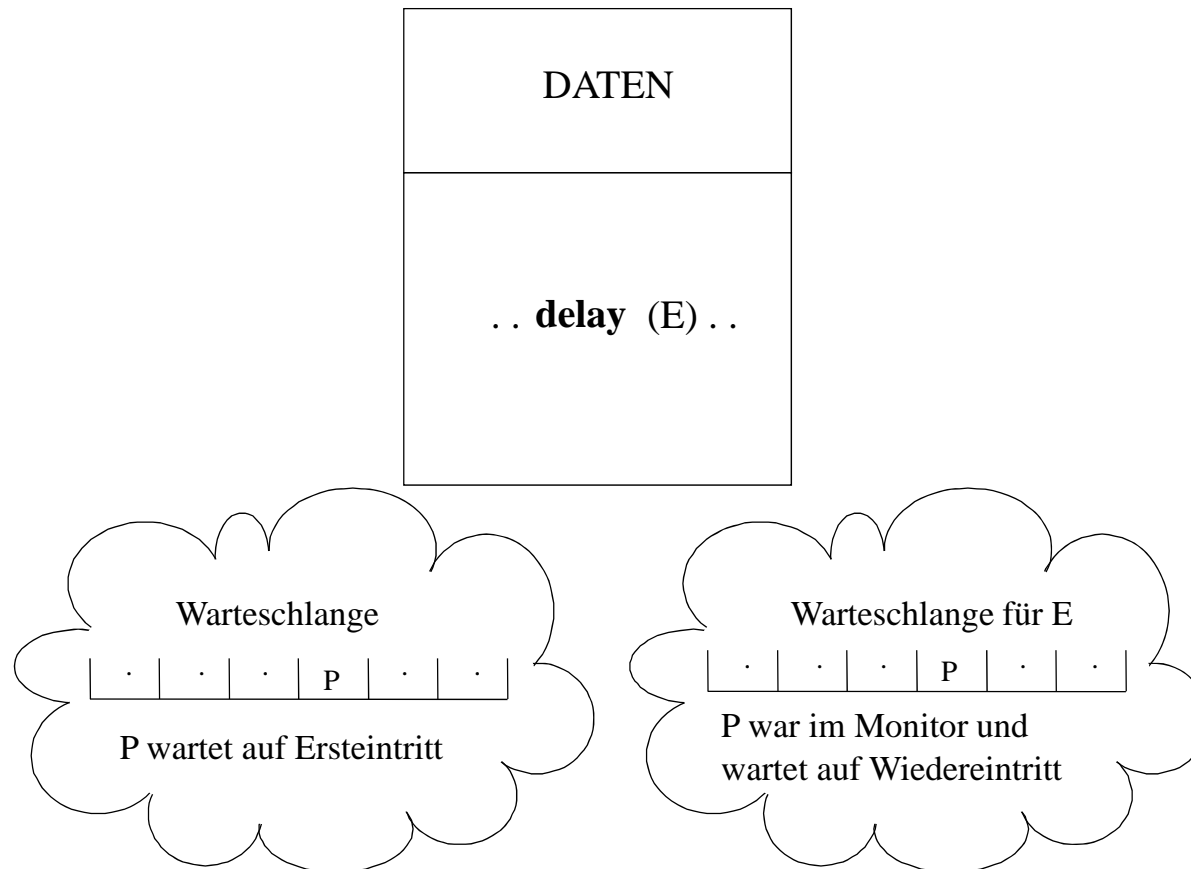
z. B. SENDER, EMPFÄNGER : **queue**

delay (SENDER) reiht den aktuellen Prozess in die Schlange SENDER ein,
 Prozess verlässt den Monitor

continue (SENDER) übergibt die Monitorkontrolle an einen Prozess
 aus der Schlange SENDER

(analog für EMPFÄNGER)

Zwei Arten von Warteschlangen bei Monitoren:



Beispiel:

```
type databuf =  
monitor  
    const    bufsize = 100;  
    var buf: array[1..bufsize] of integer;  
        next_in, next_out: 1..bufsize;  
        filled: 0..bufsize;  
        sender_q, receiver_q: queue;  
  
    procedure entry insert(item: integer); ... see next page  
    procedure entry remove(var item: integer); ... see next  
page  
  
begin  
    filled := 0;  
    next_in := 1;  
    next_out := 1  
  
end;
```

```

procedure entry insert(item: integer);
    begin
        if filled = bufsize then delay(sender_q);
        buf[next_in] := item;
        next_in := (next_in mod bufsize) + 1;
        filled := filled + 1;
        continue(receiver_q)
    end;

procedure entry remove(var item: integer);
    begin
        if filled = 0 then delay(receiver_q);
        item := buf[next_out];
        next_out := (next_out mod bufsize) + 1;
        filled := filled - 1;
        continue(sender_q)
    end;

```

```
type producer = process(buffer: databuf);  
    var stuff : integer;  
begin  
    cycle  
        -- produce stuff --  
        buffer.insert(stuff)  
    end  
end;
```

```
type consumer = process(buffer: databuf);  
    var stored_value : integer;  
begin  
    cycle  
        buffer.remove(stored_value);  
        -- consume stored_value --  
    end  
end;
```

```
var new_producer : producer;  
    new_consumer : consumer;  
    new_buffer : databuf;  
  
begin  
init new_buffer, new_producer(new_buffer),  
      new_consumer(new_buffer)  
end;
```

13.3.8 Ada 9X Protected Types

- Datenkapselung wie Monitore
- Kontroll-Abstraktion wie bei Conditional Critical Regions

Möglichkeit, die bei Conditional Critical Regions aufgetretenen Probleme elegant zu lösen:

- Vor Austritt aus dem geschützten Bereich wird im Falle von Operationen, die die geschützten Daten ändern können (procedure, entry), geprüft, ob sich die Werte von Eintrittsbedingungen geändert haben.
- Da dies noch unter dem Schutz des impliziten Semaphors erfolgt, keine Race Condition.

Beispiel 1: Semaphor-Implementierung

```
protected type Counting_Semaphore (Initial: Integer := 0) is  
  entry Acquire;           -- „P“-Operation  
  procedure Release;       -- „V“-Operation  
  function Current_Count return Integer;  
private  
  Count: Integer := Initial;  
end Counting_Semaphore;
```

```

protected body Counting_Semaphore is

    entry Acquire when Count > 0 is
        -- Suspend until Count > 0, then decrement it
    begin
        Count := Count - 1;
    end Acquire;

    procedure Release is
    begin
        Count := Count + 1;
    end Release;

    function Current_Count return Integer is
    begin
        return Count;
    end Current_Count;

end Counting_Semaphore;

```

Beispiel 2: Mailbox

```
protected type Mailbox(Box_Size: Positive) is

    entry Put(Item: in Item_Type);  -- Add item to mailbox
    entry Get(Item: out Item_Type); -- Remove item from mailbox
private
    Data : Item_Array(1..Box_Size);
    Count: Natural := 0;
    In_Index, Out_Index : Positive := 1;
end Mailbox;
```



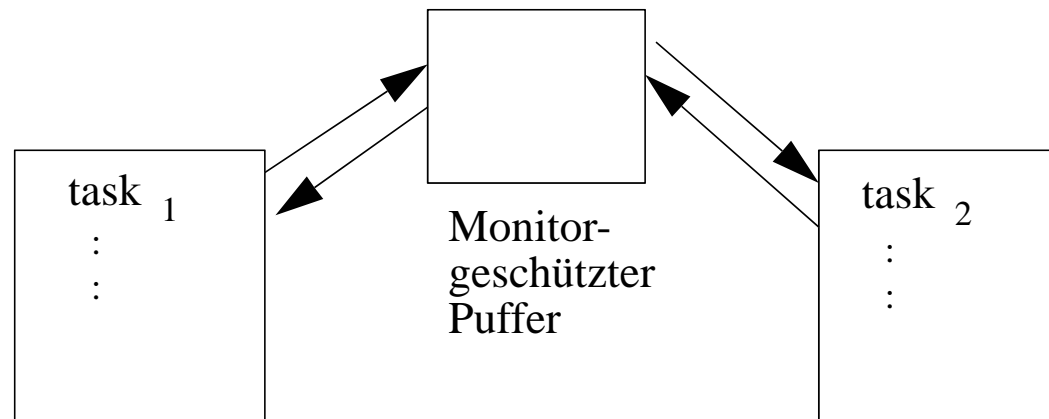
```

protected body Mailbox is
    entry Put(Item: in Item_Type)
        when Count < Box_Size is
    begin
        Date(In_Index) := Item;
        In_Index := In_Index mod Box_Size + 1;
        Count := Count + 1;
    end Put;

    entry Get(Item: out Item_Type)
        when Count > 0 is
    begin
        Item := Data(Out_Index);
        Out_Index := Out_Index mod Box_Size + 1;
        Count := Count - 1;
    end Get;
end Mailbox;

```

13.3.9 Mailbox-Prinzip



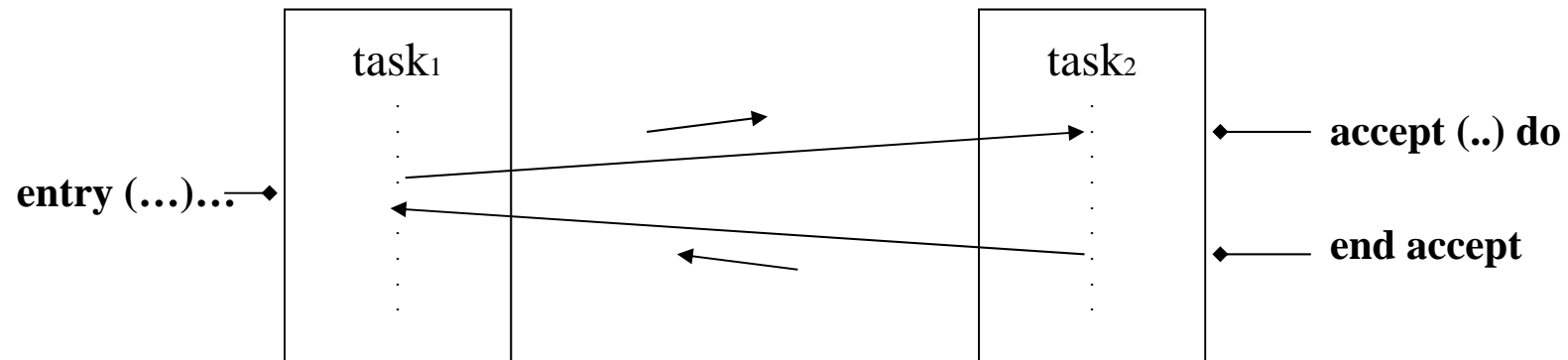
Vorteile:

- Tasks bleiben anonym
→ Hinzufügen weiterer Tasks (Producer bzw. Consumer) unproblematisch.
- Variationen in der Laufgeschwindigkeit können ausgeglichen werden.

Nachteile:

- Tasks bleiben anonym
→ Kommunikation zwischen zwei bestimmten Tasks schwierig zu verifizieren (Gesamtsystem muss betrachtet werden)
- Informationsaustausch ist bei jeder Übermittlung nur in einer Richtung möglich.
- Eine zentrale Mailbox stellt bei massiv-parallelen Systemen einen Engpass dar.

13.3.10 Rendezvous-Prinzip



Kommunikation/Synchronisation wird von `task1` „angefordert“ (entry call) und irgendwann von `task2` „akzeptiert“ (accept).

Wenn beide Tasks diese Punkte erreicht haben, werden Parameter übergeben, `task2` führt den Rendezvous-Code aus und gibt evtl. Parameter zurück. Danach fahren beide Tasks fort.

Vorteile:

- Kommunikation in beiden Richtungen möglich
- gleichzeitige Synchronisation
- einfachere Verifizierbarkeit der Kommunikation zwischen bestimmten Tasks
- Realisierbarkeit auf verteilten Systemen

Nachteile:

- gleichzeitige Synchronisation
- anfordernde Task muss akzeptierende Task kennen (aber nicht umgekehrt)

Rendezvous-Technik in Ada

- Prozess entspricht einer "Task"

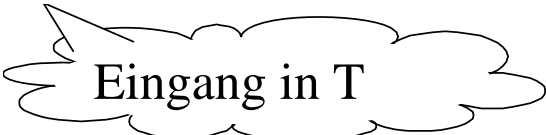
- Vereinbarung in 2 Teilen:

```
task ... is ... end;           -- Spezifikation
```

```
task body ... is ... end;     -- Rumpf
```

- Spezifikation eines Prozesses:

```
task T is
  :
  :
  entry E (...Parameter...);
  :
  :
end T;
```



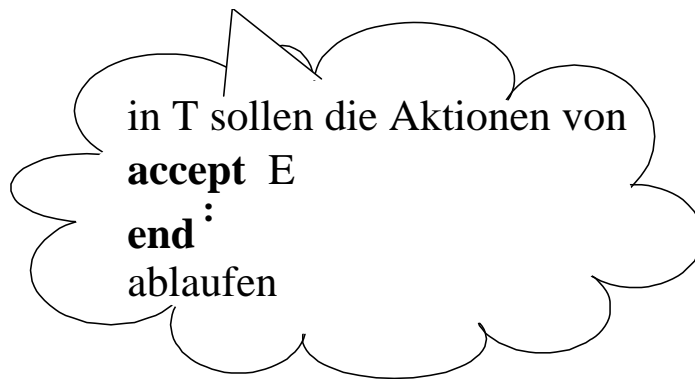
Eingang in T

- Rumpf eines Prozesses:

```
task body T is
  :
  accept E (...Parameter...) do
    :
  end;
  :
end T;
```

- Ansprechen eines Prozesses (entry call)

```
T.E (...Argumente...);
```



Behandlung des "entry calls" T.E:

Fall 1:

T hat die **accept**-Anweisung noch nicht erreicht

└─> rufender Prozess wartet in Schlange bei Eingang T.E

Fall 2:

T hatte bereits die **accept**-Anweisung erreicht

└─> T wartet bei dieser Anweisung

Fall 3:

T und der rufende Prozess treffen bei der **accept**-Anweisung zusammen



└─> T und rufender Prozess werden gekoppelt und führen die Anweisungen zwischen **do** und **end** durch. Danach trennen sich die beiden Prozesse wieder.

select - Anweisung zur Auswahl von **accept**-Anweisungen:

Form:

```
select
    when ... => accept ... do ... end; ...
or
    when ... => accept ... do ... end; ...
or
    :
    : accept ... do ... end; ...
else
    ...-- wenn kein Rendezvous ansteht, tue etwas anderes
end select;
```

Wirkung: Auswahl einer **accept**-Anweisung, deren Bedingung in **when** ... wahr ist oder die keinen Zusatz **when** ... besitzt.

Beispiel:

```
task BUF_TASK is
  entry INSERT (ITEM : in Integer);
  entry REMOVE (ITEM : out Integer);
end BUF_TASK;
task body BUF_TASK is
  SIZE          : constant Integer := 100;
  BUF           : array(1..SIZE) of Integer;
  FILLED        : Integer range 0..SIZE := 0;
  NEXT_IN,
  NEXT_OUT      : Integer range 1..SIZE := 1;
begin
  loop
    select
      when FILLED < BUFSIZE =>
        accept REMOVE(ITEM : out Integer) do
          ITEM := BUF(NEXT_OUT);
        end REMOVE;
        NEXT_OUT := (NEXT_OUT mod SIZE) + 1;
        FILLED := FILLED - 1;
      end select;
    end loop;
end BUF_TASK;
```

Der Puffer kann nun wie folgt von anderen Tasks angesprochen werden:

```

task PRODUCER;
task CONSUMER;
task body PRODUCER is
    STUFF : Integer;
    begin
        loop
            -- produce STUFF --
            BUF_TASK.INSERT(STUFF);
        end loop;
    end PRODUCER;

task body CONSUMER is
    STORED_VALUE : Integer;
    begin
        loop
            BUF_TASK.REMOVE(STORED_VALUE);
            -- consume STORED_VALUE --
        end loop
    end CONSUMER;

```

Anmerkung: Auch möglich sind (andere Formen von) Entry-Aufrufe(n), in denen nur unmittelbar oder innerhalb einer spezifizierten Zeit durchführbare Rendez-vous veranlasst werden.

13.3.11 Task/Prozess „Start“ und „Ende“

Schaffung (Anlegen von Speicherbereich, Deskriptoren etc.):

- statisch
- dynamisch

Aktivierung:

- implizit (bei Schaffung oder sonstigem Zeitpunkt in der Ausführung)
- explizit (z. B. „activate“- oder „init“-Aufruf)

Beendigung:

- Ende des Task-Codes
problematisch (für Benutzer) bei kommunizierenden Tasks
- „abort“ von außen
problematisch wegen der willkürlichen Unterbrechung
→ Dateninkonsistenz
- kooperative Terminierung
Tasks „einigen“ sich, gemeinsam zu terminieren
→ Implementierungskosten

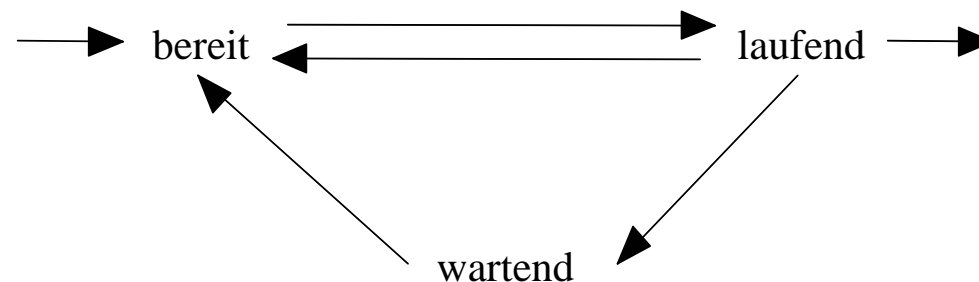
Ada:

```
select
    ...
or
    terminate;    -- signalisiert Bereitschaft zu terminieren
end select;
```

13.4 Implementierung paralleler Konstrukte

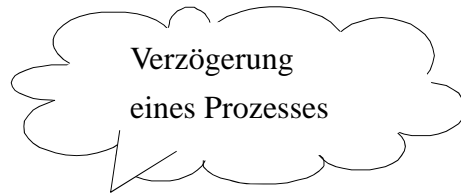
Durch ein Laufzeitsystem (ggf. das Betriebssystem), das für die entsprechende Prozess/Task-Verwaltung sorgt.

- Prozesszustände:
 - bereit
 - laufend
 - wartend
 - mit Zustandsübergängen



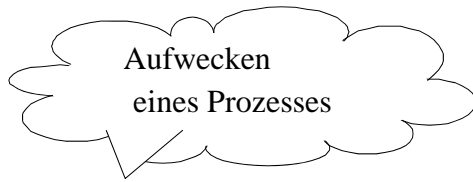
- Information über Prozesse sind enthalten in Prozessdeskriptoren mit
 - Angabe über Priorität
 - Zustand
 - Inhalt von Maschinenregistern bei der letzten Unterbrechung
 - usw.
- Operationen und Datenstrukturen für Prozessverwaltung
 - READY_QUEUE: Warteschlange für Prozessdeskriptoren bereiter Prozesse.
 - CONDITION_QUEUEs: Warteschlangen für Prozesse, die auf Eintreten einer Bedingung warten (Semaphoren, queue-Variablen, Entries u. ä.).
 - RUNNING: Variable, die den gerade laufenden Prozess beschreibt.

13.4.1 Semaphore-Implementierung



Suspendieren bei P(S):

- sichere Status des laufenden Prozesses in RUNNING
- hänge RUNNING in S.CONDITION_QUEUE ein
- speichere einen Deskriptor aus READY_QUEUE in RUNNING
- restauriere den Maschinenstatus aus RUNNING und aktiviere den Prozess.



Aufwecken bei V(S):

- sichere Status in RUNNING
- hänge RUNNING in READY_QUEUE ein
- speichere einen Deskriptor aus S.CONDITION_QUEUE in READY_QUEUE
- speichere einen Deskriptor aus READY_QUEUE in RUNNING
- restauriere den Maschinenstatus aus RUNNING und aktiviere den Prozess.

Wichtig! Unteilbarkeit und gegenseitiger Ausschluss der obigen Aktivitäten (Unterstützung durch die Hardware, z. B. Test-and-Set-Instruktion)

13.4.2 Monitor-Implementierung

- Gegenseitiger Ausschluss von Zugriffsfunktionen durch Hardwareunterstützung (Unterbrechungssperren)
- **delay** (C) analog zu P(S) für Semaphore . . .
- **continue**(C)
 - sichere Prozessstatus in RUNNING
 - hänge RUNNING in READY_QUEUE ein
 - **if** C nicht leer then
 - lege einen Deskriptor aus dieser Schlange in RUNNING
 - else**
 - lege einen Deskriptor aus READY_QUEUE in RUNNING
 - restauriere den Maschinenstatus aus RUNNING und aktiviere den Prozess

13.4.3 Scheduling

Festlegung, welche der zur Ausführung bereiten Tasks (d. h. in `READY_QUEUE`) auch wirklich zur Ausführung kommt. Notwendig, da im Allgemeinen

Anzahl der bereiten Tasks > Anzahl der verfügbaren CPUs

Fragen an die Semantik der PS:

- vorgegebenes Scheduling-Regime (z. B. FIFO-Queues, Prioritäten)
- Einflussnahme durch PS-Konstrukte
(z. B. Prioritäten-Spezifikation
 - statisch
 - dynamisch)
- an welchen Ausführungspunkten findet Scheduling statt?

13.5 Parallele Ausführungseinheiten in PS oder BS?

Parallelität durch BS-Prozesse:

- + weniger Implementierungsaufwand für PS
- + BS-Prozesse können in mehreren PSen geschrieben sein

- Prozesswechsel im Allgemeinen wesentlich zeitaufwendiger
- im Allgemeinen keine Möglichkeit für gemeinsamen Datenraum
 - Kommunikation wesentlich teurer (Ausnahme: neuere UNIX-Versionen)
- kein Typenschutz bei Kommunikation
- BS-Scheduler möglicherweise ungeeignet für Anwendung
- nicht immer ist ein geeignetes BS vorhanden

Parallelität durch „Tasks“ in PS:

- + hat die obigen Nachteile nicht
 - + bessere Integration mit anderen Sprachelementen
 - + Teilprüfbarkeit der Einhaltung von Restriktionen
-
- hat die obigen Vorteile nicht
 - bei Implentierung auf BS manchmal Probleme, weil BS von sequentiellm Verhalten der Prozesse ausgeht (z. B. UNIX „Errno-Problem“)
 - Laufzeitsystem nicht einfach ersetzbar

Kapitel 14

Enkapsulierung

14 Abstrakte Datentypen

Prinzipien:

- Darstellung der Werte eines Typs bleiben dem Benutzer des Typs verborgen
- Implementierung von darauf definierten Operationen bleiben verborgen.
- Nur bewusst sichtbar gemachte Typeigenschaften und Spezifikationen der Operationen sind dem Benutzer des Datentyps zugänglich.
- "Feste" Bindung der sichtbaren Operationen an den Typ.
- Kontrolle durch entsprechende semantische Sichtbarkeitsregeln, ggf. auch durch (textuelle) Trennung von *Spezifikation* und *Implementierung*.

Gegenbeispiel Pascal:

Pascal erlaubt zwar

- Definition neuer Wertemengen: **type** T =
- Definition von Operationen: **procedure** P
function F

erlaubt aber ***nicht***

- Abkapselung von Werten und Operationen

Beispiel:

```
type EHESTAND_EINTRAG = (LEDIG, VERHEIRATET,  
    GESCHIEDEN, VERWITWET);  
BEZUGS_PERSON = ↑PERSON;  
PERSON = record
```

```
    case EHESTAND: EHESTAND_EINTRAG of  
        VERHEIRATET:  
            (PARTNER: BEZUGS_PERSON;  
             :      );
```

```
end;
```

```
var JANE, TARZAN : PERSON;
```

Wenn JANE und TARZAN heiraten, müssen beide durch einen Record gleichen Aufbaus, d. h., mit Komponente

EHESTAND = VERHEIRATET

sowie konsistenten Einträgen in der Komponente PARTNER beschrieben werden.

Gefahr in PASCAL: Inkonsistente Einträge, da jede Komponente von JANE und TARZAN direkt zugänglich ist.

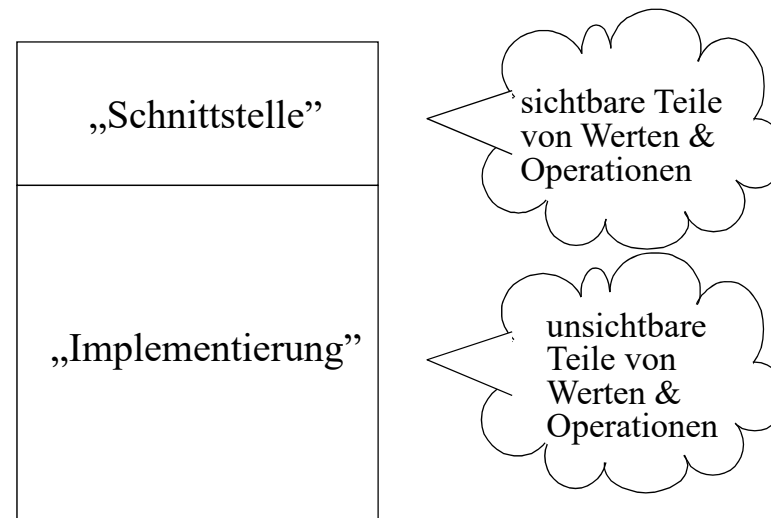
Ferner ein Wartungsproblem, wenn der Typ PERSON geändert werden soll. ("Wo sind alle Bezüge auf Komponenten im Programmtext?")

Beispiele für Abstrakte Datentypen:

- Frühe Ansätze für Datenabstraktion sind enthalten in
 - SIMULA 67: **class**-Konzept
 - Concurrent Pascal: **class**-Konzept
 - CLU: **cluster**-Konzept

Bemerkung: SIMULA 67 bindet erstmalig die Operationen stärker an die Typen, alias Klassen, hat aber noch die Schwäche mangelnden Schutzes von Objekten gegen unerwünschte Operationen.

- Weiterführung der obigen Idee von Datenabstraktion
 - genaue Kontrolle über sichtbare und unsichtbare Bestandteile von Werten und Operationen



Obiges Bild zu abstrakten Datentypen ist dem üblichen Bild für Module und ihren Implementierungen sehr ähnlich,
denn

14.1 Zusammenwirkende Gesichtspunkte

- Gruppierung von Deklarationen, Modularisierung
 - Typ-Prinzip, Operationen und Werte als zusammengehörig zu betrachten
 - „Information hiding”
 - der Typdarstellung (Typ ist allein durch die Operationen „definiert”)
 - der Implementierung der Operationen
 - zur Implementierung benötigter Hilfsdaten
 - Klassenbildung und „Ererbbarkeit” von Typ-Eigenschaften
 - Effizienzgesichtspunkte (Übersetzung, Laufzeit)
 - getrennte Übersetzbarkeit
- => Modulkonzepte und Datenabstraktion sind einander oft ergänzende Konzepte.**

14.2 Beispiele für Abstrakte Typen und Module

14.2.1 Simula-Klassen

```
class stack;
begin
  -- Daten-Deklarationen
  integer array list(1..100);
  integer top;
  :
  -- Deklaration der Operationen
  boolean procedure empty;

  procedure push (element);
  :
  integer procedure pop();
  :
  top := 0;  -- Initialisierung der Daten
  :
end stack;
```

```
ref (stack) Stack1;  
stack1 :- new stack  
    -- Anlegen einer Instanz der Klasse (nur via Pointer)  
stack1.push(7);  
    -- Aufruf der Operationen via Objekt-Name  
stack1.top := 7; -- ! leider auch möglich !
```

14.2.2 CLU-Cluster

Ähnlich zu Simula-Klassen, aber Daten bleiben verborgen

```
stack = cluster is empty, push, pop,  
        create, ...      -- Nennung der Operationen  
    rep = record [list: ...; top: ...; ]  
        -- Repräsentation der Daten  
  
    empty = proc...      -- Deklaration der Operationen  
    push = proc...  
    pop = proc...  
    create = proc...  
  
end stack;
```



```
stack1 : stack;  
stack1 := stack$create(...);  
    -- explizite Schaffung und Initialisierung des Objekts  
    -- Zugriff auf Operationen via Cluster-Name  
stack$push(..., stack1);  
  
stack1.list -- (oder Ähnliches) nicht möglich!
```

14.2.3 Parametrisierte Typen alias Generische Typen

Typen als Parameter anderer Typen, z. B. in CLU:

```
stack = cluster [t : type] is ...  
    where t has  
        <Operationsspezifikationen für t>  
        :  
        :  
    push = proc (E : t, ...) end stack;  
S1 : stack[integer];  
S2 : stack[real];  
S1 := stack[integer]$create (...);  
S2 := stack[real]$create (...);  
  
stack[integer]$push (8, S1);  
stack[real]$push (14.0, S2);
```

14.2.4 C++-Klassen

```
class stack{  
    private: // eigentlich unnötig, da Voreinstellung  
        -- nach außen nicht sichtbare Deklarationen  
    int top;  
    int list[100];  
    protected:  
        -- nur für Objekte aus Unterklassen sichtbar  
    int special_op() {..};
```

public: -- ab hier: sichtbare Deklarationen

```
    int empty() {...};  
    void push (int number) {...};  
    int pop() {...};  
    stack() {...};    -- constructor  
    ~stack() {...}; -- destructor  
}  
  
{  
    stack stk;    -- impliziter Aufruf von stack ( )  
    stk.push(7); -- Zugriff auf Push via Objekt-Name  
    stk.top      -- illegal  
}                -- impliziter Aufruf des Destruktors
```

14.2.5 Modula-2-Module

```
DEFINITION MODULE stack;
    TYPE stacktype;      --"opaque types"
                        -- mehrere solche Typen möglich im Modul
    :
    PROCEDURE empty(s : stacktype) : BOOLEAN;
    :                    -- Spezifikation der Operationen
END stack;
```

```

IMPLEMENTATION MODULE stack;

    TYPE stacktype = POINTER TO ...;
                                -- muss Zeigertyp sein

    :

    PROCEDURE empty(s: stacktype) : BOOLEAN;
    BEGIN ... END empty;
        -- Implementierung der Operationen

    :

END stack;

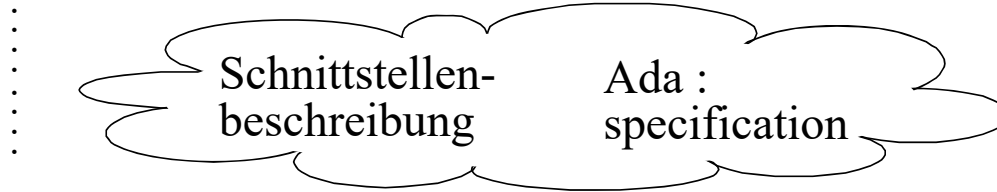
S : stacktype;
initialize(S) ; -- explizite Initialisierungsprozedur nötig

```

14.2.6 Ada-Pakete

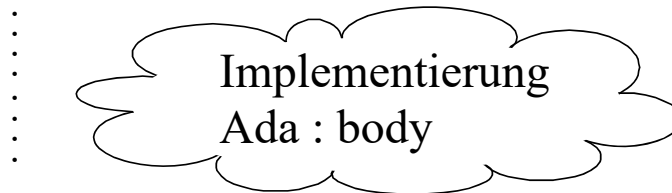
- Syntax

package Name **is**



end Name;

package body Name **is**



end Name;

Beispiel :

```
package TEILNEHMER_LISTE is
    type PERSON is record ..... end;
    procedure SORT;
    procedure EINFUEGEN (P: in PERSON);
    function IST_TEILNEHMER (P: in PERSON)
        return BOOLEAN;
    ...
end TEILNEHMER_LISTE;

package body TEILNEHMER_LISTE is
    LISTE : array (1..N) of PERSON;
    procedure SORT is ..;
    procedure EINFUEGEN(P: in PERSON) is
        ... begin ... end;
    function IST_TEILNEHMER(P: in PERSON)
        return BOOLEAN is ...;
end TEILNEHMER_LISTE;
```

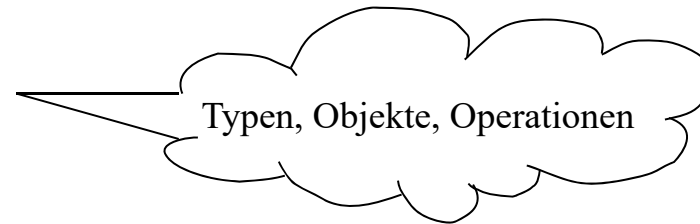

Rumpf von `TEILNEHMER_LISTE` enthält

- die Implementierung der im Spezifikationsteil aufgelisteten Operationen
 - die Implementierung der Teilnehmerliste
 - evtl. Hilfsoperationen, Hilfsdaten, Hilfstypen
- In diesem Beispiel bleibt die Struktur des Typs `PERSON` sichtbar, da durch `TEILNEHMER_LISTE.PERSON` ein Zugriff auf Komponenten des Typs möglich ist.

Die Struktur von PERSON kann verborgen werden durch

```
package TEILNEHMER_LISTE is  
    type PERSON is private;
```

```
    ...  
private  
    type PERSON is ....  
end TEILNEHMER_LISTE;
```



Auf **private**-Typen sind durch den Benutzer des Pakets nur die im sichtbaren Teil der Paketschnittstelle enthaltenen Operationen anwendbar. Darüber hinaus noch

- Zuweisungen an Variablen solchen Typs
- Tests auf Gleichheit/Ungleichheit

Im privaten Teil stehende Deklarationen bleiben dem Benutzer des Pakets verborgen. (Ausnahme in Ada 9X: "Child units" haben Sichtbarkeit auf privaten Teil der "Vorfahren").

Allgemeines Muster:

```
package P is
```

sichtbarer Teil
der Schnittstelle

```
private
```

für Benutzer verborgener
Teil der Schnittstelle; aber
für "Child Units" sichtbar

```
end P;
```

Im Folgenden wird demonstriert, wie das Paketkonzept durch verschiedene Ausgestaltungen

- zur modularen Gruppierung
- zur Schablone einer Gruppierung
- zum abstrakten Objekt
- zum abstrakten Typ wird.

Paket als Gruppierung

```
package Stack_Mgr is  
  type Stack is private;  
  procedure push (S : Stack; E : Integer);  
  function pop (S : Stack) return Integer;  
  function empty (S : Stack) return Boolean;  
  
private  
  type Stack is ...;  
end Stack_Mgr;
```

```
Stack1 : Stack_Mgr.Stack;  
Stack_Mgr.push(Stack1, 7);  
    -- Zugriffe auf Typen und Operationen über Paketnamen
```

```
Stack2 : Stack_Mgr.Stack;  
Stack_Mgr.push(Stack2, 8);  
-- oder .....  
use Stack_Mgr;          -- macht Namen aus Stack_Mgr direkt sichtbar
```

```
Stack3 : Stack;  
push(Stack3, 10);
```

Generisches Paket

Ada-Pakete sind mit

- Objekten
 - Typen
 - Unterprogrammen (Prozeduren, Funktionen)
- parametrisierbar.

```
generic
```

```
    type T is private;
```

```
    with function xyz (A : T) return T;
```

```
    -- Operationen von T, die im Paket benötigt werden
```

```
package Stack_Mgr is
```

```
    type Stack is private;
```

```
    procedure push (S : Stack; E : T);
```

```
    :
```

```
end Stack_Mgr;
```

```

package Real_Stack_Mgr is
    new Stack_Mgr(Real, "-");
package Int_Stack_Mgr is
    new Stack_Mgr(Integer, "+");
    -- „Instantiierungen“ ergeben geeignet „ausgeprägte“ Pakete

S1 : Real_Stack_Mgr.Stack;
Real_Stack_Mgr.push(S1, 7.0);

S2 : Int_Stack_Mgr.Stack;
Int_Stack_Mgr.push(S2, 8);

Stack_Mgr.push(..)      -- illegal

```

Paket als abstraktes Objekt

```
package Stack is
  procedure push (E : Integer);
  function pop return Integer;
  function empty return Boolean;
end Stack;

package body Stack is
  top : Integer := 0;
  data : array (1..100) of Integer;
  : -- Implementierung der Operationen
end Stack;

Stack.push(7); -- es gibt genau ein Stackobjekt
               -- Paket = Objekt
:
etc.
```


(Generisches) Paket als abstrakter Typ

```
generic
package Stack is
    -- wie gehabt
end Stack;
package body Stack is
    -- wie gehabt
end Stack;
package Stack1 is new Stack;
    -- Instantiierung, kreiert ein neues Paket/"Objekt"

Stack1.push(7);    -- Zugriff über "Objekt"namen
package Stack2 is new Stack;
    -- neues Paket/"Objekt"

Stack2.push(8);
usw.
```

"Don't do this at home"! (denn der Compiler wird das wohl ineffizient abbilden)

14.3 Abstrakte Datentypen (Zusammenfassung)

... werden realisiert durch "Typ-" oder "Modulkapselung".

Argumente für Typkapselung:

- deutlichere Bindung von Typ und seinen Operationen
- konzeptionell einfacher

Für beide Kapselungen gilt aber:

Unterteilung der Eigenschaften in

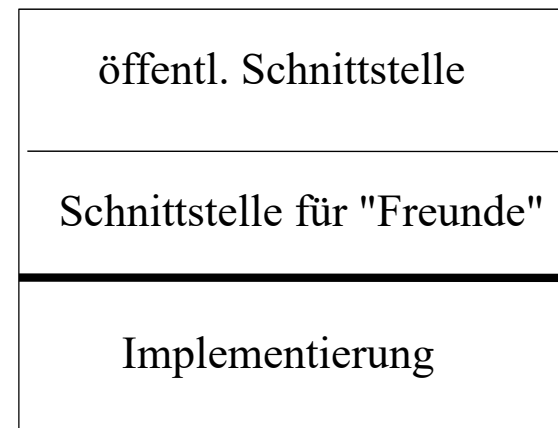
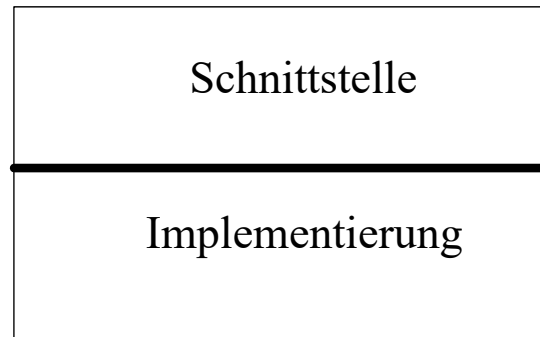
- "sichtbar für jedermann" und
- "sichtbar nur für die Implementierung der Operationen" ist bisweilen zu restriktiv.

=> "Freund"-Konzepte

14.4 "Freund"-Konzepte

Eine weitere Differenzierung der Unterscheidung von

in



Beispiele:

"protected" in C++-Klassen:

Unterklassen sehen "protected", aber nicht "private" Eigenschaften.

"friend"-Funktionen in C++- Klassen:

haben vollen Zugriff auf alle Eigenschaften der Klasse.

"Child units" in Ada 95:

Child Units sehen "private" Deklarationen der Vorfahren, aber nicht Deklarationen in deren Rümpfen.

Explizite Identifikation der Freunde in Exportangaben
(Nachteil: Code-Modifikation bei Nachtrag von Freunden)

(Klassisches Beispiel für befreundete Klassen: Matrix und Vektor, z. B. für die Effizienz der Multiplikation $M \times V \rightarrow V$ oder der Gleichungslösung $Mv = b$)

Child Units in Ada 95

```
package A is
  type T is private;
private
  type T is record
    Val: Integer;
  end record;
end A;
```

```
package body A is
  Y: Integer;
end A;
```

-- Child Unit wird durch Namenskomposition identifiziert

```
package A.Friend is
  type X is private;
private
  type X is array(1..10) of T;
end A.Friend;
```

```
package body A.Friend is  
    O: X;  
begin  
    O(7).Val := 7; -- legal ! Struktur von T hier sichtbar  
    Y := 9; -- illegal. Keine Sichtbarkeit in den Rumpf  
end A.Friend;
```

Pakete 'Hans_Mayer' oder 'Uni.Student' können dagegen den Typ T nur als privaten Typ verwenden.

Kapitel 15

Strukturierungsmechanismen für größere Programmsysteme

15 Strukturierungsmechanismen für größere Programmsysteme

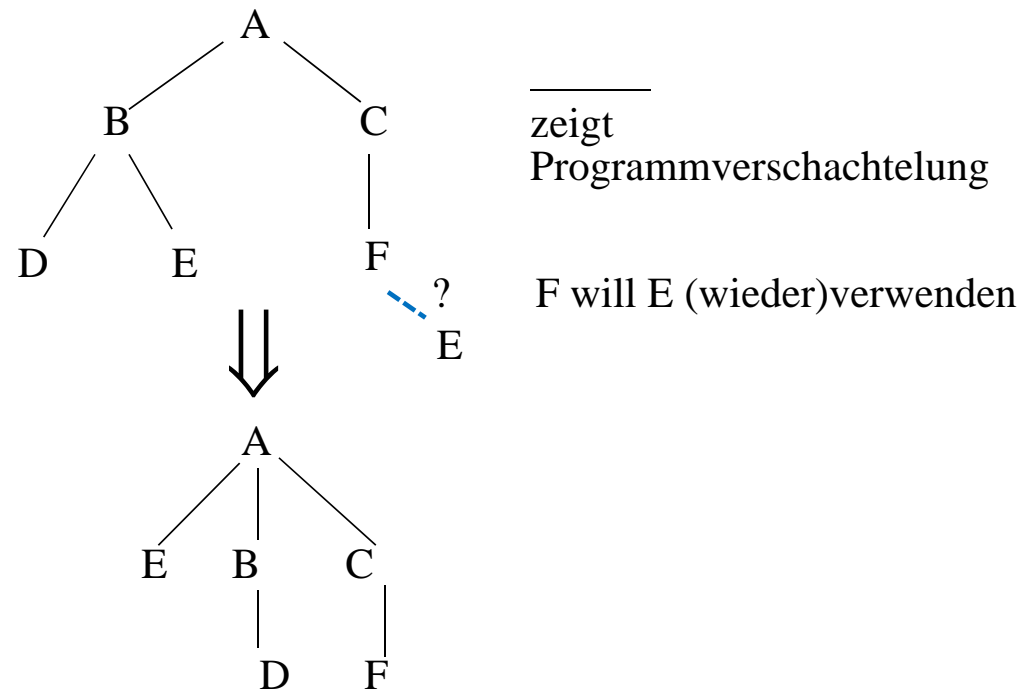
Anforderungen:

- Aufteilbarkeit in möglichst separat entwickelbare und testbare Teile
- Beschreibbarkeit der Abhängigkeit zwischen den Teilen
- Kontrollierbarkeit gegenüber „ungeplanten“ Abhängigkeiten
- Ausnutzung der Mehrfachverwendung (und Wiederverwendung) von Teilen
- Reflexion der Architektur des Systementwurfs

Welche Sprachmittel unterstützen diese Anforderungen?

Top-down-Entwurf (stepwise refinement)

Sprachmittel: z. B. hierarchische Schachtelung von Unterprogrammen (Pascal) oder Modulen

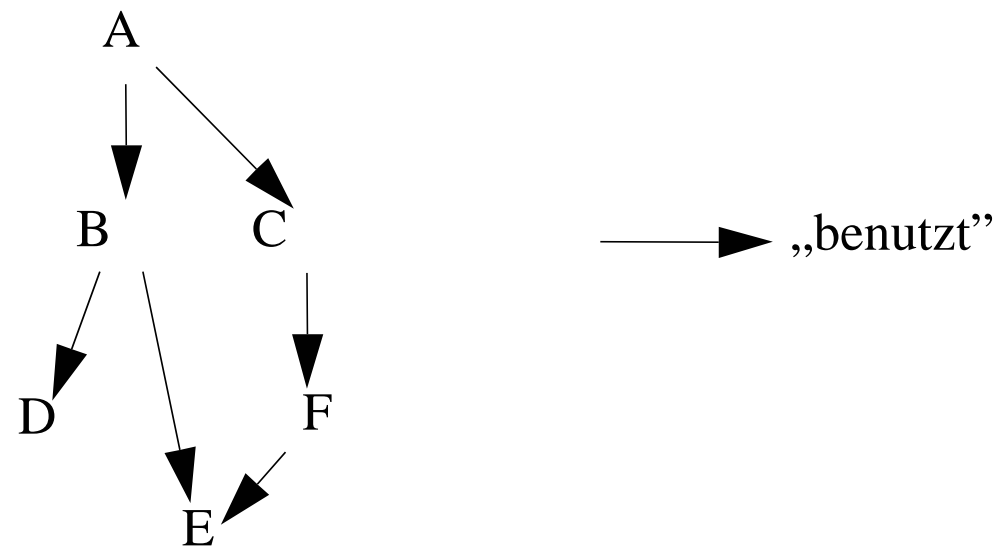


Probleme:

- Gemeinsame Dienste sind von "verfeinernden" Diensten nicht unterschieden.
- Gemeinsame Dienste müssen auf einer Hierarchie-Ebene angesiedelt sein, die den Verwendern gemeinsam sichtbar ist
 - ⇒ unnötig breite Sichtbarkeit und damit Möglichkeit ungeplanter Abhängigkeit
 - ⇒ Systemarchitektur nicht mehr offensichtlich

Bottom-Up-Entwurf u. ä.

Sprachmittel: Sammlung separater Bausteine



mit entsprechenden Abhängigkeitsbeziehungen

Baustein: Modul, Klasse, abstrakter Typ etc.

- ⇒ Möglichkeit für wiederverwendbare Bausteine
- ⇒ aber: globale Sichtbarkeit der Bausteine
- ⇒ jedoch: Abhängigkeiten evtl. differenziert spezifizierbar

Beschreibung der Abhängigkeiten:

1. Was wird durch einen Baustein angeboten ("Exporte")
2. Was wird von einem Baustein benötigt ("Importe")
 - ⇒ Prüfbarkeit der Existenz und Abdeckung der Abhängigkeiten

Exporte:

- Implizit aus der Gruppierung (Trennung von Schnittstelle und Implementierung)
- Explizite Export-Angaben

eventuell: nach „Importeuren“ differenzierte Exporte

Importe:

- Durch Einzelangabe pro importierter Deklaration
- Durch Angabe einer Gruppierung für alle jeweiligen Exporte
- Durch Objekttypen/Klassen impliziert

Unabhängige Übersetzung

Bausteine werden in beliebiger Reihenfolge übersetzt. Linker fügt Bausteine zusammen (Abgleich von Exporten und Importen).

- ⇒ Verlust von Konsistenzprüfungen (z. B. Typ-Prüfungen, Signatur-Prüfungen) an den Grenzen der Übersetzung
- ⇒ "Notwendigkeit" separater Werkzeuge zum Nachholen dieser Prüfungen (z. B. "lint" für C)

Abhängige (separate) Übersetzung

Ein Modul kann erst übersetzt werden, wenn alle Import-liefernden Module übersetzt sind.

Vorteil: Volle Konsistenzprüfung bei Übersetzung möglich.

Probleme:

1. Schafft Projektabhängigkeiten
2. Gegenseitig abhängige Module
⇒ Teillösung: Trennung von Spezifikation und Implementierung der Module
3. Konsequenzen bei Veränderungen von Modulen

Konsequenzen bei Veränderung exportierender Module

... existieren natürlich bei beiden Übersetzungsmodellen

⇒ "Make"-Funktion der Sprachumgebung erforderlich

oder

⇒ Konzept der Programmbibliothek in der Sprache verankert (LIS, Ada)

oder

⇒ Konzept der Programmbibliothek in der Entwicklungsumgebung verankert (Studio ...)

"Make":

Aufgrund einer (separaten) Beschreibung der Modulabhängigkeiten wird von einem Werkzeug

- die Übersetzungsreihenfolge
- die Notwendigkeit von Neuübersetzungen

erkannt und veranlasst.

- ⇒ Konsistenzprüfung über PS hinaus möglich
- ⇒ Konsistenz nur bei Anwendung des Werkzeugs garantiert
- ⇒ Gefahr unvollständiger/falscher Beschreibung der Modulabhängigkeiten
- ⇒ Programmgeneratoren können eingebunden werden

Programmbibliothek/Entwicklungsumgebung:

Sprachregeln oder Regeln der Entwicklungsumgebung schreiben vor, dass nach Übersetzung eines Moduls alle nicht neu übersetzten, abhängigen Module nicht importiert (oder gebunden) werden können.

- ⇒ Erzwingt Anwendung der „Make“-Funktionalität
- ⇒ Abhängigkeitsbeziehungen direkt in Sprachkonstrukten verankert, daher immer korrekt
- ⇒ Konsistenz immer garantiert
- ⇒ Automatische "Closure" für Binder

ENDE