

5. Datentypen

Themen dieses Kapitels:

5.1 Allgemeine Begriffe zu Datentypen

- Typbindung, Typumwandlung
- abstrakte Definition von Typen
- parametrisierte und generische Typen

5.2 Datentypen in Programmiersprachen

- einfache Typen, Verbunde, Vereinigungstypen, Reihungen
- Funktionen, Mengen, Stellen

5.1 Allgemeine Begriffe zu Typen

Typ: Wertemenge mit darauf definierten Operationen

z. B. `int` in Java: Werte von `Integer.MIN_VALUE` bis `Integer.MAX_VALUE`
mit arithmetischen Operationen

Typ als Eigenschaft von

Literal:	Notation für einen Wert des Typs,
Variable:	speichert einen Wert des Typs,
Parameter:	übergibt einen Wert des Typs an den Aufruf,
Ausdruck:	liefert einen Wert des Typs,
Aufruf:	liefert einen Wert des Typs,
Funktion, Operator:	Signatur (Parameter- und Ergebnistypen)

Typen werden **in der Sprache definiert:**

z. B. in C: `int`, `float`, `char`, ...

Typen können **in Programmen definiert** werden:

Typdefinition bindet die Beschreibung eines Typs an einen Namen,

z. B. in Pascal:

```
type Datum = record tag, monat, jahr: integer; end;
```

Typprüfung (type checking):

stellt sicher, dass jede Operation mit Werten des dafür festgelegten Typs ausgeführt wird,

Typsicherheit

Statische oder dynamische Typbindung

Statische Typbindung:

Die **Typen** von Programmgegenständen (z.B. Variable, Funktionen) und Programmkonstrukten (z. B. Ausdrücke, Aufrufe) werden **durch den Programmtext festgelegt**.

z. B. in Java, Pascal, C, C++, Ada, Modula-2 **explizit durch Deklarationen**

z. B. in SML, Haskell **implizit durch Typinferenz** (siehe GPS-7.4 ff)

Typprüfung im Wesentlichen zur Übersetzungszeit.

Entwickler muss erkannte Typfehler beheben.

Dynamische Typbindung:

Die **Typen** der Programmgegenstände und Programmkonstrukte werden erst **bei der Ausführung bestimmt**. Sie können bei der Ausführung nacheinander Werte unterschiedlichen Typs haben.

z. B. Smalltalk, PHP, JavaScript und andere Skriptsprachen

Typprüfung erst zur Laufzeit.

Evtl. werden Typfehler erst beim Anwender erkannt.

Keine Typisierung:

In den Regeln der Sprache wird der **Typbegriff nicht verwendet**.

z. B. Prolog, Lisp

Beispiele für Typregeln

1. Eine **Variable mit Typ T** kann nur einen Wert aus der Wertemenge von T speichern.

```
float x; ... x = r * 3.14;
```

2. Der **Ausdruck einer return-Anweisung** muss einen Wert liefern, der aus der Wertemenge des **Ergebnistyps** der umgebenden Funktion ist (oder in einen solchen Wert konvertiert werden kann (siehe GPS-5.4)).

```
float sqr (int i) {return i * i;}
```

3. Im **Aufruf einer Funktion** muss die Zahl der Parameterausdrücke mit der Zahl der formalen Parameter der Funktionsdefinition übereinstimmen und jeder **Parameter-ausdruck** muss einen Wert liefern, der aus der Wertemenge des **Typs des zugehörigen formalen Parameters** ist (oder ... s.o.)).

4. Zwei Methoden, die in einer Klasse deklariert sind und **denselben Namen** haben, **überladen** einander, wenn sie in einigen **Parameterpositionen unterschiedliche Typen** haben. Z. B.

```
int add (int a, int b) { return a + b; }
```

```
Vector<Integer> add (Vector<Integer> a, Vector<Integer> b) {...}
```

In einem Aufruf einer überladenen Methode wird anhand der Typen der Parameterausdrücke entschieden, welche Methode aufgerufen wird:

```
int k; ... k = add (k, 42);
```

Streng typisiert

Streng typisierte Sprachen (strongly typed languages):

Die Einhaltung der **Typregeln** der Sprache stellt sicher, dass **jede Operation** nur mit **Werten des dafür vorgesehenen Typs** ausgeführt wird.

Jede Verletzung einer Typregel wird erkannt und als Typfehler gemeldet
- zur Übersetzungszeit oder zur Laufzeit.

FORTRAN	nicht streng typisiert	Parameter werden nicht geprüft
Pascal	nicht ganz streng typisiert	Typ-Uminterpretation in Variant-Records
C, C++	nicht ganz streng typisiert	es gibt undiscriminated Union-Types
Ada	nicht ganz streng typisiert	es gibt Direktiven, welche die Typprüfung an bestimmten Stellen ausschalten
Java	streng typisiert	alle Typfehler werden entdeckt, zum Teil erst zur Laufzeit

Typumwandlung (Konversion)

Typumwandlung, Konversion (conversion):

Der Wert eines Typs wird in einen entsprechenden Wert eines anderen Typs umgewandelt.

ausweitende Konversion:

jeder Wert ist im Zieltyp ohne Informationsverlust darstellbar, z. B.

`float --> double`

einengende Konversion:

nicht jeder Wert ist im Zieltyp darstellbar, ggf. Laufzeitfehler, z. B.

`float --> int` (Runden, Abschneiden oder Überlauf)

Uminterpretation ist unsicher, ist nicht Konversion!:

Das Bitmuster eines Wertes wird als Wert eines anderen Typs interpretiert.
z. B. Varianten-Records in Pascal (GPS-5.14)

Explizite und implizite Typumwandlung

Eine Konversion kann **explizit im Programm als Operation** angegeben werden (**type cast**), z. B.

```
float x = 3.1; int i = (int) x;
```

Eine Konversion kann **implizit vom Übersetzer eingefügt** werden (**coercion**), weil der Kontext es erfordert, z. B.

```
double d = 3.1;           implizit float --> double
d = d + 1;               implizit int --> double
```

Java: ausweitende Konversionen für Grund- und Referenztypen **implizit**, **einengende** müssen **explizit** angegeben werden.

Konversion für Referenzen ändert weder die Referenz noch das Objekt:

```
Object val = new Integer (42); implizit Integer --> Object
Integer ival = (Integer) val;  explizit Object --> Integer
```

Abstrakte Definition von Typen

Datenstrukturen werden in Programmen mit Typen modelliert => Modellierungskonzepte

Abstrakte Grundkonzepte zur Bildung einfacher und zusammengesetzter Wertemengen D :
(Hier: nur Wertemengen der Typen; Operationen darauf werden davon nicht erfasst.)

1. einfache Mengen: $D = \{ e_1, e_2, \dots, e_n \}$ extensionale Aufzählung der Elemente

$D = \{ a \mid \text{Eigenschaft von } a \}$ intensionale Definition

z. B. Grundtypen, Aufzählungstypen, Ausschnittstypen

2. kartesisches Produkt: $D = D_1 \times D_2 \times \dots \times D_n$

Tupel z. B. Verbunde (records); Reihungen (arrays) (mit gleichen D_i)

3. Vereinigung: $D = D_1 \mid D_2 \mid \dots \mid D_n$

Alternativen zusammenfassen

z. B. union in C und Algol 68, Verbund-Varianten in Pascal, Ober-, Unterklassen

4. Funktion: $D = D_p \rightarrow D_e$

Funktionen als Werte des Wertebereiches D

z. B. Funktionen, Prozeduren, Methoden, Operatoren; auch Reihungen (Arrays)

5. Potenzmenge: $D = P (D_e)$

z. B. Mengentypen in Pascal

Kombination von Typen

Die Grundkonzepte zur Typkonstruktion sind prinzipiell **beliebig kombinierbar**,
z. B. Kreise oder Rechtecke zusammengefasst zu 2-dimensionalen geometrischen Figuren:

Koord2D = float × float

Form = {istKreis, istRechteck}

Figur = Koord2D × Form × (float | float × float)

Position

Kennzeichen

Radius

Kantenlängen

z. B. Signatur einer Funktion zur Berechnung von Nullstellen einer als Parameter gegebenen Funktion:

(float → float) × float × float → P (float)

Funktion

Bereich

Menge der Nullstellen

Rekursive Definition von Typen

Wertemengen können auch **rekursiv definiert** werden:

z. B. ein Typ für **lineare Listen** rekursiv definiert durch Paare:

$$\mathbf{IntList} = \mathbf{int} \times \mathbf{IntList} \mid \{\mathbf{nil}\}$$

$\{\mathbf{nil}\}$ ist eine einelementige Wertemenge. \mathbf{nil} repräsentiert hier die leere Liste.

Werte des Typs sind z. B.

$$\mathbf{nil}, (1, \mathbf{nil}), (2, \mathbf{nil}), \dots, (1, (1, \mathbf{nil})), (8, (9, (4, \mathbf{nil}))), \dots$$

Entsprechend für Bäume:

$$\mathbf{IntTree} = \mathbf{IntTree} \times \mathbf{int} \times \mathbf{IntTree} \mid \{\mathbf{TreeNil}\}$$

Eine rekursive Typdefinition ohne nicht-rekursive Alternative ist so nicht sinnvoll, da keine Werte gebildet werden können:

$$\mathbf{X} = \mathbf{int} \times \mathbf{X}$$

In funktionalen Sprachen können Typen direkt so rekursiv definiert werden, z. B. in SML:

```
datatype IntList = cons of (int × IntList) | IntNil;
```

In imperativen Sprachen werden rekursive Typen mit Verbunden (struct) implementiert, die Verbundkomponenten mit Stellen als Werte (Pointer) enthalten, z. B. in C:

```
typedef struct _IntElem  *IntList;  
typedef struct _IntElem { int head; IntList tail;} IntElem;
```

Parametrisierte Typen

Parametrisierte Typen (Polytypen):

Typangaben mit **formalen Parametern, die für Typen** stehen.

Man erhält aus einem Polytyp einen konkreten Typ durch **konsistentes Einsetzen eines beliebigen Typs** für jeden Typparameter.

Ein Polytyp beschreibt die **Typabstraktion**, die allen daraus erzeugbaren konkreten Typen gemeinsam ist.

Beispiele in SML-Notation mit `'a`, `'b`, ... für Typparameter:

Polytyp	gemeinsame Eigenschaften	konkrete Typen dazu
<code>'a × 'b</code>	Paar mit Komponenten beliebigen Typs	<code>int × float</code> <code>int × int</code>
<code>'a × 'a</code>	Paar mit Komponenten gleichen Typs	<code>int × int</code> <code>(int->float) × (int->float)</code>
<code>'a list = 'a × 'a list {nil}</code>	homogene, lineare Listen	<code>int list</code> <code>float list</code> <code>(int × int) list</code>

Verwendung z. B. in **Typabstraktionen** und in **polymorphen Funktionen** (GPS-5-9a)
In SML werden konkrete Typen zu parametrisierten Typen statisch bestimmt und geprüft.

Polymorphe Funktionen

(Parametrisch) **polymorphe Funktion**:

eine Funktion, deren **Signatur ein Polytyp** ist, d. h. Typparameter enthält.

Die Funktion ist auf Werte eines jeden konkreten Typs zu der Signatur anwendbar.
D. h. sie muss unabhängig von den einzusetzenden Typen sein;

Beispiele:

eine Funktion, die die Länge einer beliebigen homogenen Liste bestimmt:

```
fun length l = if null l then 0 else 1 + length (tl l);
```

polymorphe Signatur: `'a list -> int`

Aufrufe: `length ([1, 2, 3]); length ([(1, true), (2, true)]);`

eine Funktion, die aus einer Liste durch elementweise Abbildung eine neue Liste erzeugt:

```
fun map (f, l) = ...
```

polymorphe Signatur: `(('a -> 'b) × 'a list) -> 'b list`

Aufruf: `map (even, [1, 2, 3])` liefert `[false, true, false]`

```
int->bool, int list      bool list
```

Generische Definitionen

Eine **Generische Definition** hat **formale generische Parameter**. Sie ist eine **abstrakte Definition einer Klasse** oder eines Interfaces. Für jeden generischen Parameter kann ein **Typ eingesetzt** werden. (Er kann auf Untertypen eines angegebenen Typs eingeschränkt werden.)

Beispiel in Java:

Generische Definition einer Klasse `Stack` mit generischem Parameter für den **Elementtyp**

```
class Stack<Elem>
{
    private Elem [] store ;
    void push (Elem e1) {... store[top]= e1;...}
    ...
};
```

Eine **generische Definition** wird **instanciiert** durch Einsetzen von **aktuellen generischen Parametern**. Dadurch entsteht zur Übersetzungszeit eine Klassendefinition. Z. B.

```
Stack<Float> taschenRechner = new Stack<Float>();
Stack<Frame> windowMgr = new Stack<Frame>();
```

Generische Instanziierung kann im Prinzip durch **Textersetzung** erklärt werden: Kopieren der generischen Definition mit Einsetzen der generischen Parameter im Programmtext.

Der Java-Übersetzer erzeugt für jede generische Definition eine Klasse im ByteCode, in der `Object` für die generischen Typparameter verwendet wird. Er setzt Laufzeitprüfungen ein, um zu prüfen, dass die ursprünglich generischen Typen korrekt verwendet wurden.

Generische Definitionen in C++

Generische Definitionen wurden in Ada und C++ schon früher als in Java eingeführt. Außer Klassen können auch Module (Ada) und Funktionen generisch definiert werden. **Formale generische Parameter** stehen für beliebige Typen, Funktionen oder Konstante. (Einschränkungen können nicht formuliert werden.)

Beispiel in C++:

Generische Definition einer Klasse `stack` mit generischem Parameter für den **Elementtyp**

```
template <class Elem>
  class Stack
  {   private Elem store [size];
      void push (Elem el) {... store[top]=el;...}
      ...
  };
```

Eine **generische Definition** wird **instanziiert** durch Einsetzen von **aktuellen generischen Parametern**. Dadurch entsteht Übersetzungszeit eine Klassen-, Modul- oder Funktionsdefinition.

```
Stack<float>* taschenRechner = new Stack<float>();
Stack<Frame>* windowMgr = new Stack<Frame>();
```

Auch **Grundtypen** wie `int` und `float` können als aktuelle generische Parameter eingesetzt werden.

Nutzen generischer Definitionen

Typische Anwendungen:

homogene Behälter-Typen, d. h. alle Elemente haben denselben Typ:

Liste, Keller, Schlange, ...

generischer Parameter ist der Elementtyp (und ggf. die Kapazität des Behälters)

Algorithmen-Schemata: Sortieren, Suchen, etc.

generischer Parameter ist der Elementtyp mit Vergleichsfunktion

Generik sichert statische Typisierung trotz verschiedener Typen der Instanzen!

Übersetzer kann Typkonsistenz garantieren, z. B. Homogenität der Behälter

Java hat generische Definitionen erst seit Version 1.5

Behälter-Typen programmierte man vorher mit `Object` als Elementtyp,

dabei ist **Homogenität nicht garantiert**

Generische Definitionen gibt es z. B. in C++, Ada, Eiffel, Java ab 1.5

Generische Definitionen sind **überflüssig in dynamisch typisierten Sprachen** wie Smalltalk

5.2 Datentypen in Programmiersprachen

Typen mit einfachen Wertemengen (1)

- a. Ausschnitte aus den **ganzen Zahlen** mit arithmetischen Operationen unterschiedlich große Ausschnitte: Java: `byte`, `short`, `int`, `long`; C, C++: `short`, `int`, `long int`, `unsigned`; Modula-2: `INTEGER` und `CARDINAL`
- b. **Wahrheitswerte** mit logischen Operationen
Pascal, Java: `boolean = (false, true)`; in C: durch `int` repräsentiert

Kurzauswertung logischer Operatoren in C, C++, Java, Ada:

Operanden von links nach rechts auswerten bis das Ergebnis feststeht:

```
a && b || c      i >= 0 && a[i] != x
```

- c. **Zeichen eines Zeichensatzes** mit Vergleichen, z. B. `char`
- d. **Aufzählungstypen** (enumeration)
- | | |
|---------|--|
| Pascal: | <code>Farbe = (rot, blau, gelb)</code> |
| C: | <code>typedef enum {rot, blau, gelb} Farbe;</code> |
| Java: | <code>enum farbe {rot, blau, gelb}</code> |

Die Typen (a) bis (d) werden auf ganze Zahlen abgebildet (ordinal types) und können deshalb auch exakt verglichen, zur Indizierung und in Fallunterscheidungen verwendet werden.

Typen mit einfachen Wertemengen (2)

- e. Teilmenge der **rationalen Zahlen** in Gleitpunkt-Darstellung (floating point), z. B. `float`, mit arithmetischen Operationen,

Gleitpunkt-Darstellung:

Tripel (s, m, e) mit Vorzeichen s, Mantisse m, Exponent e zur Basis $b = 2$;

Wert der Gleitpunktzahl: $x = s * m * b^e$

- f. Teilmenge der **komplexen Zahlen** mit arithmetischen Operationen z. B. in FORTRAN

- g. **Ausschnittstypen** (subrange)

in Pascal aus (a) bis (d): `Range = 1..100;`

in Ada auch aus (e) mit Größen- und Genauigkeitsangaben

Zur Notation von Werten der Grundtypen sind **Literale** definiert:

z. B. `127, true, '?', 3.71E-5`

Verbunde

Kartesisches Produkt: $D = D_1 \times D_2 \times \dots \times D_n$ mit beliebigen Typen D_i ; **n-Tupel**

Verbundtypen in verschiedenen Sprachen:

SML: `type Datum = int * int * int;`

Pascal, Modula-2, Ada:

```
type Datum = record tag, monat, jahr: integer; end;
```

C, C++: `typedef struct {int tag, monat, jahr;} Datum;`

Selektoren zur Benennung von **Verbundkomponenten**:

```
Datum heute = {27, 6, 2006};
heute.monat oder monat of heute
```

Operationen:

meist nur Zuweisung; komponentenweise Vergleiche (SML) sehr aufwändig

Notation für Verbundwerte:

in **Algol-68, SML, Ada** als Tupel: `heute := (27, 6, 2006);`

in **C** nur für Initialisierungen: `Datum heute = {27, 6, 2006};`

in **Pascal, Modula-2 keine** Notation für Verbundwerte

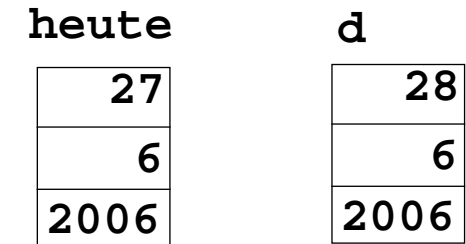
sehr lästig, da Hilfsvariable und komponentenweise Zuweisungen benötigt werden

```
Datum d; d.tag:=27; d.monat:=6; d.Jahr:=2006; pruefeDatum (d);
statt pruefeDatum ((27, 6, 2006));
```

Vergleich: Verbundwerte - Objekte

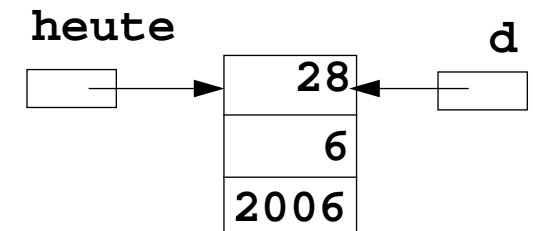
Verbundtypen in C, C++:

```
typedef struct {int tag, monat, jahr;} Datum;
Datum heute = {27, 6, 2006};
Datum d = heute; d.tag += 1;
```



Klassen in objekt-orientierten Sprachen wie Java, C++:

```
class Datum {int tag, monat, jahr;}
Datum heute = new Datum (27, 6, 2006);
Datum d = heute; d.tag += 1;
```



Vergleich

Werte von Typen

habe **keine Identität**

werden z. B. **in Variablen gespeichert**

Werte haben **keinen veränderlichen Zustand**

beliebig **kopierbar**

2 Werte sind gleich,

wenn ihre Komponenten gleich sind,
auch wenn die Werte an verschiedenen
Stellen gespeichert sind

Objekte von Klassen

haben **Identität (Referenz, Speicherstelle)**

haben **eigenen Speicher**

können **veränderlichen Zustand** haben

werden **nicht kopiert, sondern geklont**

2 Objekte sind immer verschieden,

auch wenn ihre Instanzvariablen
paarweise gleiche Werte haben.

Vereinigung (undiscriminated union)

Allgemeines Konzept: **Vereinigung von Wertebereichen: $D = D_1 \mid D_2 \mid \dots \mid D_n$**

Ein Wert vom Typ D_i ist auch ein Wert vom allgemeineren Typ D .

Variable vom Typ D können einen Wert jedes der vereinigten Typen D_i aufnehmen.

Problem: Welche Operationen sind auf den Inhalt solch einer Variable sicher anwendbar?

1. undiscriminated union: $D = D_1 \mid D_2 \mid \dots \mid D_n$

z. B. zwei Varianten der Darstellung von Kalenderdaten, als Tripel vom Typ `Datum` oder als Nummer des Tages bezogen auf einen Referenztag, z. B.

union-Typ in C:

```
typedef union {Datum KalTag; int TagNr;} uDaten;
uDaten h;
```

Varianten-Record in Pascal:

```
type uDaten = record case boolean of
    true: (KalTag: Datum);
    false: (TagNr: integer);
end;
var h: uDaten;
```

Durch den **Zugriff** wird ein Wert vom Typ D als Wert vom Typ D_i interpretiert; unsicher!

z. B. `h.TagNr = 4342;` oder `t = h.KalTag.tag;`

Speicher wird für die größte Alternative angelegt und für kleinere Alternativen ggf. nicht genutzt.

Vereinigung (discriminated union)

Allgemeines Konzept: **Vereinigung von Wertebereichen: $D = D_1 \mid D_2 \mid \dots \mid D_n$** (wie auf 5.14)

Problem: Welche Operationen sind auf den Inhalt solch einer Variable sicher anwendbar?

2. discriminated union: $D = T \times (D_1 \mid D_2 \mid \dots \mid D_n)$ mit $T = \{t_1, t_2, \dots, t_n\}$

Unterscheidungskomponente vom Typ T (**tag field**) ist Teil des Wertes und kennzeichnet Zugehörigkeit zu einem D_i ; z. B.

SML (implizite Unterscheidungskomponente):

```
datatype Daten = KalTag of Datum | TagNr of int;
```

Pascal, Modula-2, Ada (explizite Unterscheidungskomponente):

```
type uDaten = record case IstKalTag: boolean of
  true: (KalTag: Datum);
  false: (TagNr: integer);
end;
```

Sichere Zugriffe durch Prüfung des Wertes der Unterscheidungskomponente oder Fallunterscheidung darüber.

Gleiches Prinzip in objekt-orientierten Sprachen (implizite Unterscheidungskomponente):
allgemeine Oberklasse mit speziellen Unterklassen

```
class Daten { ... }
class Datum extends Daten {int tag, monat, jahr;}
class TagNum extends Daten {int TagNr;}
```

Reihungen (Arrays)

Abbildung des Indextyps auf den Elementtyp:
oder kartesisches Produkt mit fester Anzahl Komponenten

$$D = I \rightarrow E$$

$$D = E \times E \times \dots \times E$$

in **Pascal**-Notation: `type D = array [I] of E`

Indexgrenzen, alternative Konzepte:

statische Eigenschaft des Typs (Pascal):	<code>array [0..9] of integer;</code>
statische Eigenschaft der Reihungsvariablen (C):	<code>int a[10];</code>
dynamische Eigenschaft des Typs (Ada):	<code>array (0..m*n) of float;</code>
dynamisch, bei Bildung von Werten, Objekten (Java):	<code>int[] a = new int[m*n];</code>

Mehrstufige Reihungen: Elementtyp ist Reihungstyp:

`array [I1] of array [I2] of E` kurz: `array [I1, I2] of E`
 zeilenweise Zusammenfassung in fast allen Sprachen; nur in FORTRAN spaltenweise

Operationen:

Zuweisung, Indizierung als Zugriffsfunktion: `x[i]` `y[i][j]` `y[i,j]`

in C, C++, FORTRAN ohne Prüfung des Index gegen die Grenzen

Notation für Reihungswerte in Ausdrücken: (fehlen in vielen Sprachen; vgl. Verbunde)

Algol-68: `a := (2, 0, 0, 3, 0, 0);`

Ada: `a := (2 | 4 => 3, others => 0);`

C: `int a[6] = {2, 0, 0, 3, 0, 0};`

nur in Initialisierungen

Java: `int[] a = {2, 0, 0, 3, 0, 0};`
`a = new int [] {2, 0, 0, 3, 0, 0};`

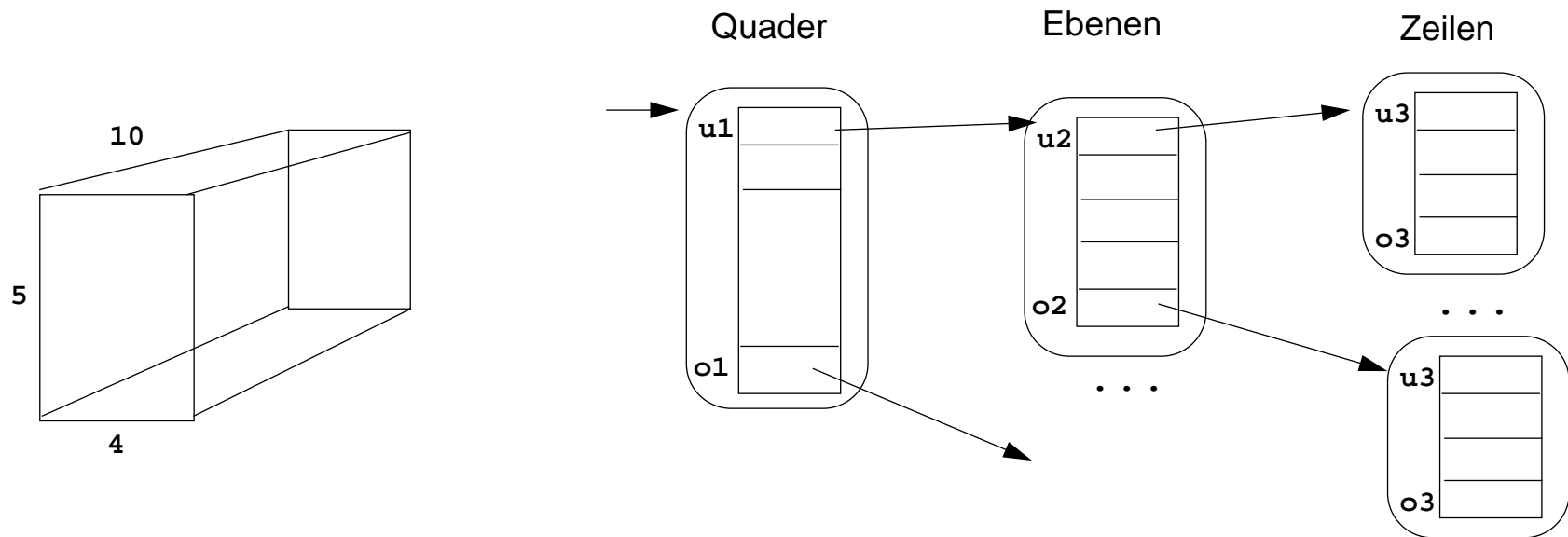
Pascal: keine

Speicherung von Arrays durch Pointer-Bäume

Ein n-dimensionales Array mit explizit gegebenen Unter- und Obergrenzen (Pascal-Notation):

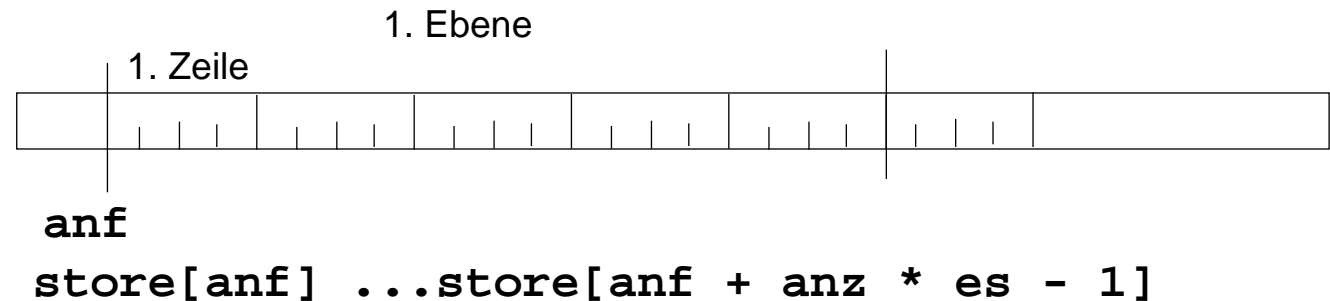
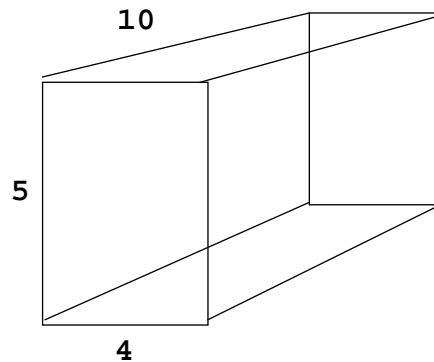
```
a: array[u1..o1, u2..o2, ..., un..on] of real;
```

wird z. B. in **Java** als **Baum von linearen Arrays** gespeichert
n-1 Ebenen von Pointer-Arrays und Daten Arrays auf der n-ten Ebene



Jedes einzelne Array kann separat, dynamisch, gestreut im Speicher angelegt werden;
nicht alle Teil-Arrays müssen sofort angelegt werden

Linearisierte Speicherung von Arrays



zeilenweise Linearisierung eines n-stufigen Arrays (z. B. in Pascal):

```
a: array[u1..o1, u2..o2, ..., un..on] of real;
```

abgebildet auf linearen Speicher, z. B. Byte-Array `store` ab Index `anf`:

```
store[anf] ... store[anf + anz * es - 1]
```

mit Anzahl der Elemente $anz = sp1 * sp2 * \dots * spn$

i-te Indexspanne $spi = oi - ui + 1$

Elementgröße in Bytes es

Indexabbildung: `a[i1, i2, ..., in]` entspricht `store[k]` mit

$$k = anf + (i1-u1)*sp2*sp3*\dots*spn*es + \\ (i2-u2)* sp3*\dots*spn*es + \dots + \\ (in-un)* es$$

$$= (\dots(i1*sp2 + i2)*sp3 + i3)* \dots + in)*es + \text{konstanter Term}$$

Funktionen

Typ einer Funktion ist ihre Signatur: $D = P \rightarrow R$ mit Parametertyp P , Ergebnistyp R
 mehrere Parameter entspricht Parametertupel $P = P_1 \times \dots \times P_n$,
 kein Parameter oder Ergebnis: P bzw. R ist leerer Typ (`void` in Java, C, C++; `unit` in SML)

Funktion höherer Ordnung (Higher Order Function):

Funktion mit einer Funktion als Parameter oder Ergebnis, z. B. $(\text{int} \times (\text{int} \rightarrow \text{int})) \rightarrow \text{int}$

Operationen: Aufruf

Funktionen in imperativen Sprachen: nicht als Ausdruck, nur als Deklaration

Funktionen als Parameter in den meisten Sprachen.

Geschachtelte Funktionen in Pascal, Modula-2, Ada - nicht in C.

Globale **Funktionen als Funktionsergebnis** und **als Daten** in C und Modula-2.

Diese Einschränkungen garantieren die **Laufzeitkeller-Disziplin:**

Beim Aufruf müssen alle statischen Vorgänger noch auf dem Laufzeitkeller sein.

Funktionen in funktionalen Sprachen:

uneingeschränkte Verwendung auch als Datenobjekte;

Aufrufschachteln bleiben solange erhalten, wie sie gebraucht werden

Notation für eine Funktion als Wert: Lambda-Ausdruck, meist nur in funktionalen Sprachen:

SML: `fn a => 2 * a`

Algol-68: `(int a) int: 2 * a`

Beispiel für Verletzung der Laufzeitkeller-Disziplin

In imperativen Sprachen ist die Verwendung von Funktionen so eingeschränkt, dass bei Aufruf einer Funktion die Umgebung des Aufrufes (d. h. alle statischen Vorgänger-Schachteln) noch auf dem Laufzeitkeller liegen.

Es darf z. B. nicht eine **eingeschachtelte Funktion an eine globale Variable zugewiesen** und dann aufgerufen werden (vgl. GPS-4.6):

Programm mit geschachtelten Funktionen

```

h float a;
  fct ff;

  q int i;

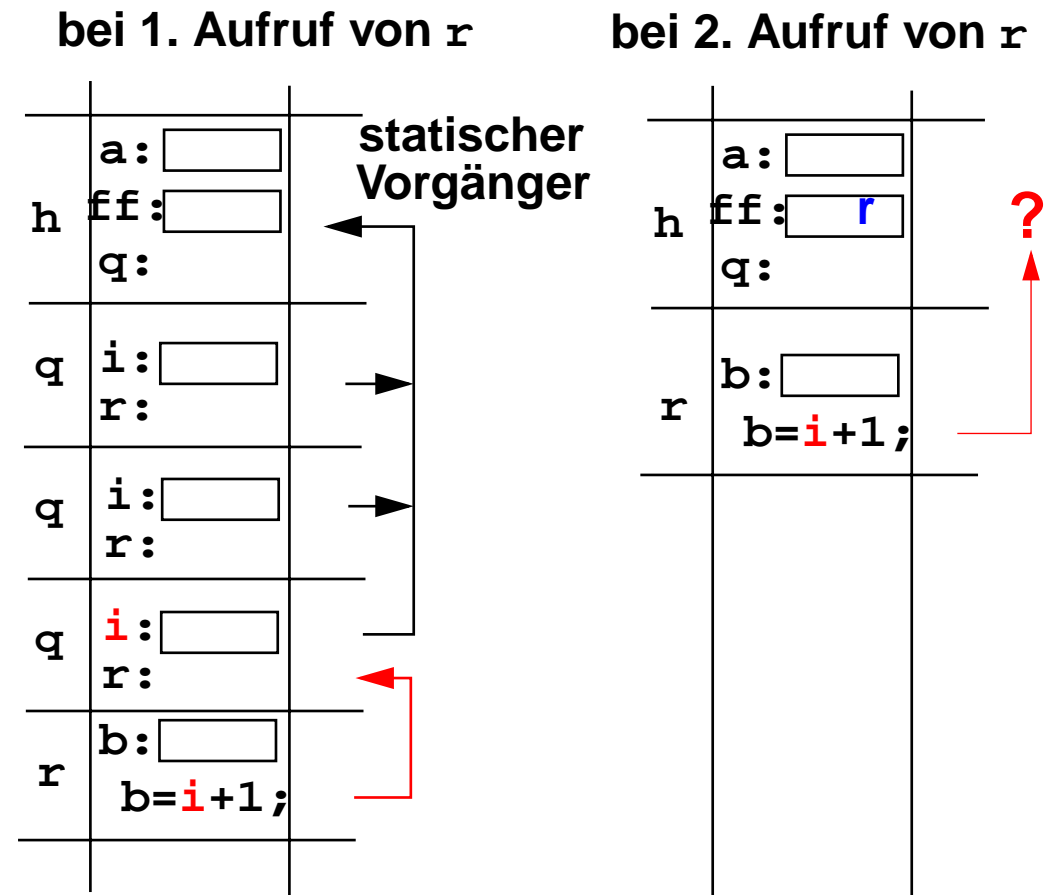
    r int b;
      b=i+1;

    if(..) q();

    r();
    ff = r;

  q();
  ff();
  
```

Laufzeitkeller



Mengen

Wertebereich ist die **Potenzmenge**: $D = P(D_e)$ oder
 Menge der charakteristischen Funktionen $D = D_e \rightarrow \text{bool}$ mit Elementtyp D_e
 D_e muss meist einfach, geordnet und von beschränkter Kardinalität sein.
 (Allgemeine Mengentypen z. B. in der Spezifikationsprache **SETL**.)

Operationen: Mengenoperationen und Vergleiche

z. B. **in Pascal:**

```
var m, m1, m2: set of 0..15;
e in m      m1 + m2      m1 * m2      m1 - m2
```

Notation für Mengenwerte: in Pascal: [1, 3, 5]

Effiziente Implementierung durch **Bit-Vektor** (charakteristische Funktion):

```
array [De] of boolean
```

mit logischen Operationen auf Speicherworten als Mengenoperationen.

in Modula-2: vordefinierter Typ

```
BITSET = SET OF [0..1-1] mit 1 Bits im Speicherwort.
```

in C:

kein Mengentyp, aber logische Operationen $|$, $\&$, \sim , \wedge
 auf Bitmustern vom Typ **unsigned**.

Stellen (Referenzen, Pointer)

Wertebereich $D = S_w \mid \{\text{nil}\}$

S_w : Speicherstellen, die Werte eines Typs w aufnehmen können.

nil eindeutige Referenz, verschieden von allen Speicherstellen

Operationen: Zuweisung, Identitätsvergleich, Inhalt

Wertnotation und Konstruktor:

a. Stelle einer deklarierten **Variable**, z. B. in C: `int i; int *p = &i;`

b. Stelle eines dynamisch generierten Objektes als Ergebnis eines **Konstruktoraufrufs**,
z. B. in Java `Circles cir = new Circles (0, 0, 1.0);`

Stellen als Datenobjekte werden nur in **imperativen Sprachen** benötigt!

Sprachen **ohne Zuweisungen** brauchen nicht zwischen einer Stelle und ihrem Inhalt zu unterscheiden ("**referentielle Transparenz**")

Objekte in objektorientierten Sprachen haben eine **Stelle**.

Sie bestimmt die Identität des Objektes.

Zusammenfassung zum Kapitel 5

Mit den Vorlesungen und Übungen zu Kapitel 5 sollen Sie nun Folgendes können:

5.1 Allgemeine Begriffe zu Datentypen

- Typeigenschaften von Programmiersprachen verstehen und mit treffenden Begriffen korrekt beschreiben
- Mit den abstrakten Konzepten beliebig strukturierte Typen entwerfen
- Parametrisierung und generische Definition von Typen unterscheiden und anwenden

5.2 Datentypen in Programmiersprachen

- Ausprägungen der abstrakten Typkonzepte in den Typen von Programmiersprachen erkennen
- Die Begriffe Klassen, Typen, Objekte, Werte sicher und korrekt verwenden
- Die Vorkommen von Typkonzepten in wichtigen Programmiersprachen kennen
- Speicherung von Reihungen verstehen