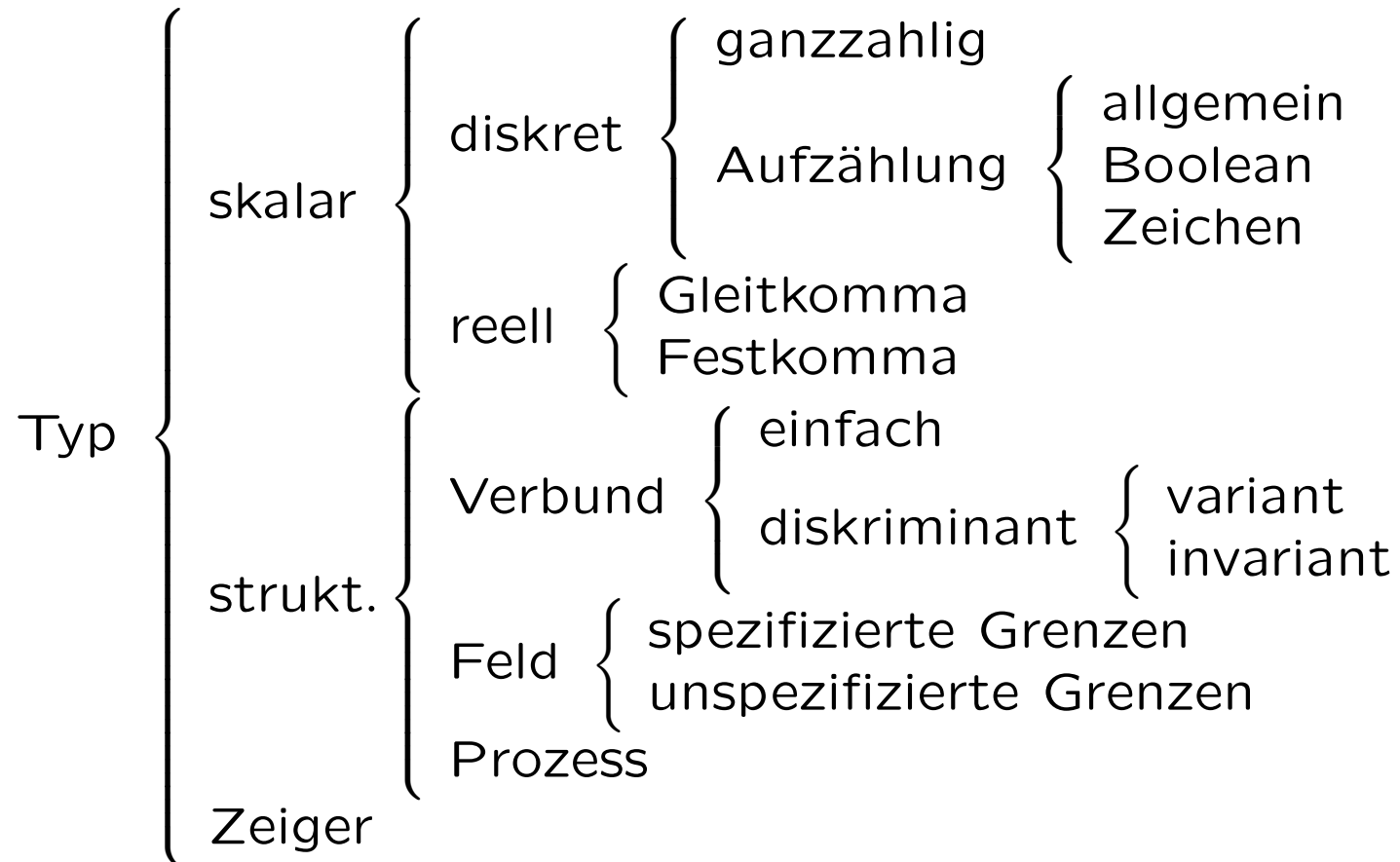


---

# Das Typsystem von Ada 83



---

# Aufzählungstypen in Ada

Der Wertebereich wird extensional festgelegt.

Die Werte eines Typs sind vollständig geordnet. Die Vergleichsoperatoren  $<$ ,  $<=$ ,  $=$ ,  $>=$ ,  $>$  und  $\neq$  sind auf jedem Aufzählungstyp  $t$  vordefiniert, alle vom Typ  $t \times t \rightarrow \text{BOOLEAN}$ .

*Allgemeine Aufzählungstypen* sind frei definierbar und haben keine weiteren Zugriffsfunktionen. Beispiele:

```
type TAG is (MO, DI, MI, DO, FR, SA, SO);  
type ARBEITSTAG is (MO, DI, MI, DO, FR);
```

Die Wertebereiche von Aufzählungstypen überschneiden sich nicht; MO in ARBEITSTAG und MO in TAG sind verschieden.

---

# Attribute von Aufzählungstypen

Attribute sind in Ada Familien von Funktionen, die einen Typ als Index haben. Für jeden Aufzählungstyp  $t$  sind diese Attribute vorgegeben:

Funktion	Typ	Beschreibung
$t'$ FIRST	$t$	kleinster Wert in $t$
$t'$ LAST	$t$	größter Wert in $t$
$t'$ POS	$t \rightarrow \text{CARDINAL}$	Stelle des Wertes in der Ordnung
$t'$ VAL	$\text{CARDINAL} \rightarrow t$	Wert an vorgegebener Stelle
$t'$ SUCC	$t \rightarrow t$	Direkt nachfolgender Wert
$t'$ PRED	$t \rightarrow t$	Direkt vorausgehender Wert

---

# Wahrheitswerte und Zeichen

Die Typen `BOOLEAN` und `CHARACTER` sind als Aufzählungstypen definiert. Wertebereiche sind `TRUE` und `FALSE` bzw. die Zeichen des ASCII-Codes.

Auf Instanzen von `BOOLEAN` sind die zusätzlichen Operationen `not`, `and`, `and then`, `or`, `or else` und `xor` (in Präfix- bzw. Infixschreibweise) vordefiniert.

Es lassen sich beliebige Aufzählungstypen definieren, deren Werte Zeichen sind. Beispiel:

```
type ZIFFER is ('0','1','2','3','4','5','6','7','8','9');
```

---

# Ganze Zahlen

Neben den Vergleichsoperatoren sind die unären Operatoren `+` und `-`, die binären Operatoren `+`, `-`, `*`, `/`, `mod`, `rem` und `**` sowie die einstellige Funktion `ABS` definiert.

Operanden und Ergebnisse sind jeweils vom selben Typ.

Es gibt mehrere ganzzahlige Typen. `SHORT_INTEGER`, `INTEGER` und `LONG_INTEGER` sind vordefiniert, aber implementationsabhängig.

Es können eigene ganzzahlige Typen definiert werden:

```
type SEITENZAHL is range 1..10000;  
type ZWEI_ZIFFERN is mod 100;
```

Statisch auswertbare Attribute für jeden ganzzahligen Typ  $t$  sind  $t$ 'FIRST und  $t$ 'LAST, jeweils vom Typ  $t$ .

---

# Gleitkommazahlen

Dieselben Operationen wie auf ganzen Zahlen (ausser `rem` und `div`). Vordefinierte Typen: `SHORT_FLOAT`, `FLOAT` und `LONG_FLOAT`.

Eigene Typen: `type MY_FLOAT is digits 8 range -1.0..1.0E30;`

Attribute:

Funktion	Typ	Beschreibung
<code>t'DIGITS</code>	<code>CARDINAL</code>	Anzahl der genauen Dezimalstellen
<code>t'MANTISSA</code>	<code>CARDINAL</code>	Länge der Binär-Mantisse
<code>t'EMAX</code>	<code>CARDINAL</code>	maximaler Exponent
<code>t'SMALL</code>	<code>t</code>	kleinste positive Zahl
<code>t'LARGE</code>	<code>t</code>	größte positive Zahl
<code>t'EPSILON</code>	<code>t</code>	maximaler Fehler pro Operation

---

# Festkommazahlen

Es gibt keine vordefinierten Festkommatypen. Typdefinitionen haben die Formen

```
type SPANNUNG is delta 0.1 range 0.0..10000.0;  
type PREIS is delta 0.01 digits 5;
```

Bei einigen Operationen sind auch ganze Zahlen als Operanden erlaubt.

Attribute:

Funktion	Typ	Beschreibung
<i>t</i> 'DELTA	CARDINAL	Spezifizierte Mindestgenauigkeit
<i>t</i> 'ACTUAL_DELTA	CARDINAL	Tatsächliche Genauigkeit
<i>t</i> 'BITS	CARDINAL	Speicherbedarf
<i>t</i> 'LARGE	<i>t</i>	größte Zahl

---

# Felder

```
type VEKTOR is array (1..5) of INTEGER;  
type MATRIX is array (1..K*J, FUNC(N)..FUNC(N*M)) of FLOAT;  
type STUDENTFL is array (TAG range MO..SA, 1..8) of TEXT;
```

Indexbereiche sind diskrete Typen oder Unterbereiche davon.

Die Grenzen brauchen in Ada nicht statisch bestimmbar sein.

Generelle Form: type  $t$  is array ( $t_1, \dots, t_n$ ) of  $t'$ ;

Attribute:

Funktion	Typ	Beschreibung
$t'$ FIRST( $i$ )	$t_i$	untere Grenze des $i$ -ten Index
$t'$ LAST( $i$ )	$t_i$	obere Grenze des $i$ -ten Index
$t'$ LENGTH( $i$ )	CARDINAL	Anzahl der Werte des $i$ -ten Index

---

## Felder mit unspezifizierten Grenzen

Eine Besonderheit von Ada sind Feldtypen mit unspezifizierten Grenzen, von denen keine Instanzen erzeugt werden können:

```
type VECTOR is array (INTEGER range <>) of FLOAT;
```

Vor Verwendungen müssen Grenzen nachgetragen werden:

```
A_VECTOR: VECTOR(1..FUNC(N));
```

STRING ist ein vordefiniertes eindimensionales Feld von Zeichen mit unspezifizierten Grenzen.

---

# Verbundtypen

```
type DATUM is record
    TAG:    INTEGER range 1..31;
    MONAT:  MONATSNAME;
    JAHR:   INTEGER range 1800..2200;
end record;
```

Verbundtypen können Diskriminanten enthalten:

```
type VECTORS (N: INTEGER) is record
    V1: VECTOR(1..2*N);
    V2: VECTOR(1..3*N);
end record;
```

Verbundtypen mit Diskriminanten sind abstrakte Typen. Diskriminanten müssen bei der Variablendeklaration angegeben werden und sind unveränderlich.

---

# Variante Verbundtypen

Varianten werden mittels case-Anweisung auf Diskriminanten festgelegt:

```
type GERAET is (DRUCKER,PLATTE,TROMMEL);
type PERIPHERIE (SORTE: GERAET) is record
  STATUS: ZUSTAND;
  case SORTE is
    when DRUCKER =>
      ZEILE:    INTEGER;
    when others =>
      ZYLINDER: INTEGER;
      SPUR:     INTEGER;
  end case;
end record;
```

---

## Speicherauslegung (einfach)

```
type Device_Register is range 0..2**8-1;
```

```
D: Device_Register;
```

```
for D'Size use 8;
```

```
for D'Address use To_Address(16#FF00#);
```

```
type Colors is (Red, Green, Blue);
```

```
for Colors use (Red=>10, Green=>20, Blue=>30);
```

---

# Speicherauslegung für Verbund

```
type Status_Word is record
    Mask      : array (0..7) of Boolean;
    Protection: Integer range 0..3;
    Something  : Address;
end record;

for Status_Word use record
    Mask      : at 0*Word range 0..7;
    Protection: at 0*Word range 10..11;
    Something  : at 1*Word range 8..31;
end record;

for Status_Word'Alignment use 8;
```

---

# Unterbereichstypen

Von Typen können Unterbereichstypen erzeugt werden. Diese enthalten zusätzliche Einschränkungen.

Einschränkungen beziehen sich auf Wertebereich, Genauigkeit, Indexbereich und Diskriminanten:

```
subtype WERKTAG is TAG range MO..SA;  
subtype SMALL is INTEGER range -1000..1000;  
subtype S_SPANNUNG is SPANNUNG delta 0.5 range 0.0..5.0;  
subtype SMALL_VECTOR is VECTOR(2..4);  
subtype PERIPHERIE_DRUCKER is PERIPHERIE(DRUCKER);
```

Unterbereichstypen sind keine eigenständigen neuen Typen. Einschränkungen werden zur Laufzeit überprüft. Bei Nichteinhaltung erfolgt eine Ausnahmebehandlung.

---

# Abgeleitete Typen

Abgeleitete Typen sind eigenständige Typen, die in Ausdrücken nicht gemischt vorkommen dürfen.

Auf einem abgeleiteten Typ sind dieselben Operationen wie auf dem Ursprungstyp anwendbar (spezielle Form von Überladen).

```
type APFEL is new INTEGER range 0..100;  
type BIRNE is new INTEGER range 0..100;
```

Abgeleitete Typen übernehmen (erben) die Typstruktur.

Explizite Typumwandlung ist erlaubt:

```
A: APFEL := 27;  
B: BIRNE := BIRNE(INTEGER(A));
```

---

# Zeigertypen

Auf jeden Datentyp kann auch ein Zeigertyp definiert werden.

```
type APFEL_P is access APFEL;  
EIN_APFEL: APFEL_P := new APFEL;
```

`new APFEL` erzeugt eine neue Instanz von `APFEL` im Heap.

Der Typ, der nach `new` steht, muss so genau spezifiziert sein, dass eine neue Instanz davon erzeugt werden kann. Die Grenzen von Feldern und Diskriminanten von Verbunden müssen festgelegt werden. Bei der Definition eines Zeigertyps brauchen diese Einschränkungen noch nicht bekannt sein.

---

# Funktionen und Prozeduren in Ada

```
procedure PUSH (EL: in ELEMENT_T; ST: in out STACK_T);  
procedure POP (EL: out ELEMENT_T; ST: in out STACK_T);  
function MAX (X,Y: in INTEGER) return INTEGER;  
function "*" (X: in MATR(A,B); Y: in MATR(B,C))  
    return MATR(A,C);
```

Es gibt eine eins-zu-eins-Beziehung zwischen Funktions- oder Prozedurnamen und -implementierungen. Funktions- und Prozedurköpfe werden nicht als Typen gesehen.

---

# Funktions- und Prozedurtypen

Sprachen, die Funktions- und Prozedurvariablen unterstützen, heben die eins-zu-eins-Beziehung zwischen Namen und Implementierungen auf.

Häufig (z.B. in Modula-2 und C) entspricht ein Funktions- oder Prozedurtyp dem -kopf; Typäquivalenz ist Strukturgleichheit und semantische Unterschiede werden nicht betrachtet.

Einige Sprachen erlauben Typäquivalenz aufgrund von Namensgleichheit und geben damit Funktions- oder Prozedurtypen semantische Bedeutung.

Es ist möglich, Typäquivalenz als Strukturgleichheit aufzufassen und dennoch die Semantik zu berücksichtigen.

---

# Typen von Prozessen (1)

Eine Besonderheit von Ada sind Prozesse (Ausführungen eines Programmstücks) als Instanzen von Typen.

```
task type DRUCKER_TREIBER is
    entry DRUCKE_ZEILE (ZL: in STRING(80));
    entry SEITENVORSCHUB;
    entry DRUCKER_STATUS (F: out STATUS);
end;

DRUCKER_POOL: array(1..MAX_DRUCKER) of DRUCKER_TREIBER;
DRUCKER_POOL(1).DRUCKE_ZEILE("Hello world!");
```

Es gibt eine eins-zu-eins-Beziehung zwischen dem Typ eines Prozesses und dessen Implementierung (Programmstück).

---

## Typen von Prozessen (2)

Der Typ eines Prozesses definiert Einstiegspunkte, die erlaubte Nachrichten festlegen. Das Senden von Nachrichten entspricht syntaktisch Prozeduraufrufen.

Gesendete Nachrichten werden durch die Ausführung von “accept statements” vom Prozess entgegengenommen und beantwortet (“Rendezvous”).

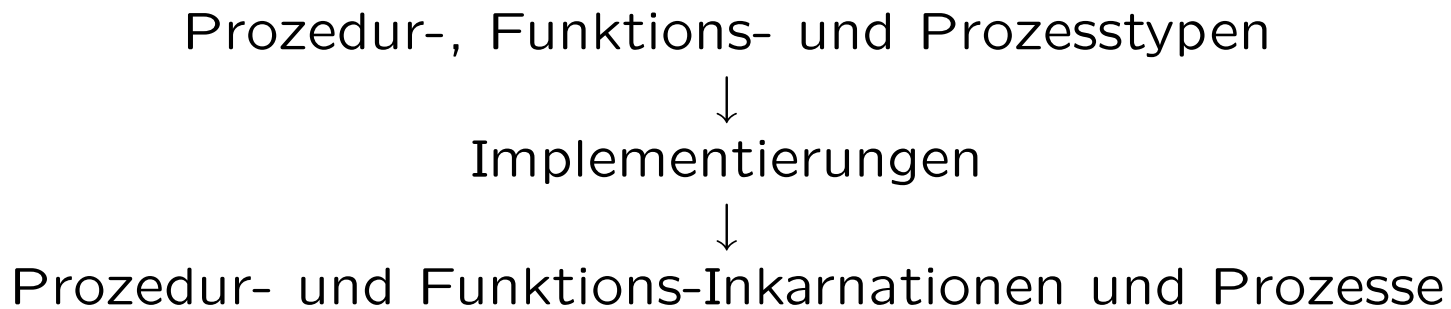
Für jeden Prozess  $p$  und seine Einstiegspunkte  $e$  gibt es diese Attribute:

Funktion	Typ	Beschreibung
$p$ 'TERMINATED	BOOLEAN	TRUE wenn $p$ beendet wurde
$p$ 'PRIORITY	CARDINAL	Priorität von $p$ , falls definiert
$p$ 'STORAGE_SIZE	CARDINAL	Platz den $p$ beansprucht
$e$ 'COUNT	CARDINAL	Anzahl wartender Aufrufe

---

# Typ-Instanz-Hierarchien

Man kann sich eine dreistufige Hierarchie vorstellen, in der jeder Typ mehrere Implementierungen und jede Implementierung mehrere Inkarnationen bzw. Prozesse haben kann:



Einige Sprachen vereinigen Prozeduren, Funktionen, Prozesse und Module zu einem einzigen Sprachkonstrukt.

---

# Generische Pakete

Ada hat Signaturen der heterogenen Algebren mit Parameter-Sorten fast unverändert zur Definition von ADT-Schnittstellen übernommen.

Terminologie: generisches Paket = parametrisches Modul.

Generische Pakete sind abstrakt. Konkrete (nichtgenerische) Pakete werden daraus durch Typableitung und Angabe der fehlenden Information erzeugt.

Pakete können globale Variablen enthalten. Abgeleitete Typen haben eigene Kopien dieser Variablen.

Globale Variablen unterscheiden Module von Algebren.

```
generic
  type ELEMENT_T is private;
  MAX_ELEM: CARDINAL;
package STACK is
  type STACK_T is limited private;
  procedure PUSH (EL: in ELEMENT_T; ST: in out STACK_T);
  procedure POP (EL: out ELEMENT_T; ST: in out STACK_T);
private
  type STACK_T is record
    EINTRAEGE: array (1..MAX_ELEM) of ELEMENT_T;
    INDEX:      INTEGER range 0..MAX_ELEM;
  end record;
end STACK;

package body STACK is ... end STACK;

package I_STACK is new STACK(INTEGER,100);
MY_STACK: I_STACK.STACK_T;
```

generic

type ELEMENT\_T is private;

MAX\_ELEM: CARDINAL;

package STACK1 is

procedure PUSH (EL: in ELEMENT\_T; ST);

procedure POP (EL: out ELEMENT\_T; ST);

end STACK1;

package body STACK1 is

EINTRAEGE: array (1..MAX\_ELEM) of ELEMENT\_T;

INDEX: INTEGER range 0..MAX\_ELEM;

...

end STACK1;

package MY\_STACK1 is new STACK1(INTEGER,100);