WERTE

die <u>funktionale Facette</u> von Programmiersprachen

Wert

ein Datum, das in der Programmiersprache
berechnet werden kann (z.B. in einem Ausdruck)

Datentyp

eine Wertemenge
<u>mit</u> den passenden Operationen

Ausdrücke

verschachtelte Aufrufe von Operationen und Funktionen

Auswertung von Ausdrücken liefert Werte

Inhalt

Datenstrukturen 20
- einfache vordefinierte Datentypen 21
- Abschnitts- und Aufzählungstypen 22
- zusammengesetzte Datentypen 24
- kartesisches Produkt 25
- disjunkte Vereinigung 30
- Potenzmenge 34
- Relation 35
- Abbildung 36
- Liste (homogen) 39
- allgemeiner rekursiver Typ 42
- der Sonderfall Zeichenketten 44
- Sonstige Datentypen 45
- Datentypen abstrakt 46

Typisierung 47
- Typgleichheit 48
- Typvollständigkeit 49

Ausdrücke 50
- Funktionsaufruf, bedingter Ausdruck 51
- benannte Konstanten und Variablen 52
- Zusammenfassung: Werte, Typen und Ausdrücke 53

Praktikum
- einfache und zusammengesetzte Datentypen 54
- Typgleichheit in einigen Programmiersprachen 55
- Typvollständigkeit 56
- Übungen 58

DATENTYP

Wert

- ein Datum, das in der Programmiersprache
  - berechnet werden kann (z.B. in einem Ausdruck)

Datentyp

- eine Wertemenge
- <u>mit</u> passenden Operationen

Arten von Datentypen

- einfach (größtenteils eingebaut oder vordefiniert)
- zusammengesetzt (Datenstrukturen)
- rekursiv

zur Vereinheitlichung der Schreibweise

- mathematische Definition der Datenstrukturen

einfache vordefinierte Datentypen

sprachunabhängige Definition der einfachen Typen

Truth-Value = { true, false }
Integer = { ..., -2, -1, 0, +1, +2, ... }
Real = { ..., -1.0, ..., 0.0, ... +1.0, ... }
Character = { ..., `a´, `b´, ... `z´, ... }

Wahrheitwerte und ganze Zahlen gibt es in jeder Sprache
sie sind Voraussetzung, z.B. für bedingte Anweisungen und Felder)

viele einfache Typen sind <u>implementierungsabhängig</u>

- Character kann vom Zeichensatz abhängen
- Zahlenwerte können von der Darstellungsgenauigkeit abhängen

Gegenbeispiel: `Integer` in Miranda und Haskell

Ein Widerspruch zur Plattformunabhängiogkeit!
Wie kann man damit umgehen?

- einfach ignorieren
- transparente Implementierungsbeschränkungen angeben (z.B. in Algol-68)

`minint, maxint, maxreal, smallreal`

- Darstellungsgenauigkeit vorschreiben

Java (IEEE Standard, Unicode)

vordefinierte Datentypen sind <u>abstrakt</u>
Ihre <u>Maschinenrepräsentation</u> ist für den Programmierer irrelevant
Sie sind definiert durch Operationen
(und deren mathematische Eigenschaften)

Abschnitts- und Aufzählungstypen

Aufzählungstypen (Ada)

```
type Month is  (jan, feb, mar, apr, may, jun,
                jul, aug, sep, oct, nov, dec)
```

Bemerkung

Aufzählungen werden als Abschnitte ganzer Zahlen dargestellt

klarer wird es in Eiffel ausgedrückt:

```
jan, feb, mar, apr, may, jun,
jul, aug, sep, oct, nov, dec: Integer is unique
```

Operationen auf Aufzählungstypen

Zuweisung, Gleichheitstest, größer/kleiner, Nachfolger, Vorgänger

Abschnittstypen von einfachen Datentypen (Ada)

```
subtype   Rain  is Month range jun .. sep;
          Day   is Integer range 1..31;
          Digit is Character range ‘0’ .. ‘9’;
          Probability is Float range 0.0 .. 1.0;
```

Bemerkung

Abschnittstypen sind <u>Untertypen</u> ihres Basistyps

d.h. sie erben alle Operationen ihres Basistyps

im allgemeinen muß <u>dynamisch</u> geprüft werden,

ob ein Wert des Basistyps zum Untertyp gehört:

```
var today, yesterday: Day;
...
today := yesterday + 1;
```

Kardinalität von Datentypen

Wieviele Elemente hat eine Datentyp?

einfach, zusammengesetzt: vorhersehbar (statisch bestimmbar)

rekursiv: nicht vorhersehbar (dynamisch bestimmt, potentiell beliebig)

Kardinalität von einfachen Typen

#Truth-value	=	2	
# Month	=	12	(siehe oben)
# Integer	=	2 * maxint + 1	(Pascal)

zusammengesetzte Datentypen

auch strukturierte Datentypen, Datenstrukturen
    werden mit vordefinierten Typkonstruktoren erzeugt
    dazu gibt es Selektoren,
        mit denen die Werte wieder zerlegt werden können

Was gibt es für Beispiele
- Verbund (**record**)
- Variante (**record … case …**)
- Tupel
- Vereinigung (**union**)
- Menge
- Feld
- Liste
- Zeichenkette
- Baum

Typkonstruktoren (und Beispiele aus Programmiersprachen)

• kartesisches Produkt	(**record**, **struct**, "*")
• disjunkte Vereinigung	(**union**, "|", **record case …**)
• Menge	(**set**, **array of** Truthvalue)
• Relation	Teilmengen von Produkten
• Abbildung	(**array**, **function**)

kartesisches Produkt

Definition (als geordnetes Paar)

$$S \times T = \{ (x,y) \mid x \in S, y \in T \}$$

Operationen:

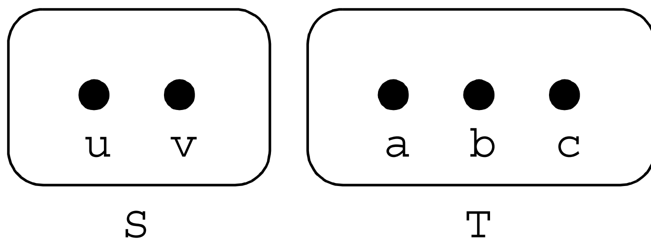
Selektion der Komponenten (Projektion)

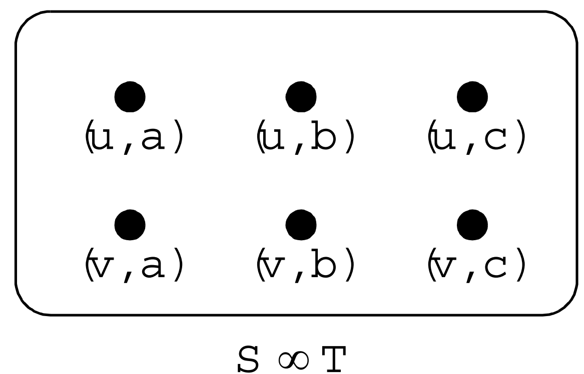
Kardinalität

$$\#(S \times T) = \#S \times \#T$$

Verallgemeinerung auf n-Tupel

$$S_1 \times S_2 \times \ldots \times S_n = \{ (x_1, x_2, \ldots, x_n) \mid x_i \in S_i ; 1 \leq i \leq n \}$$





S ∞ T

Tupel (Haskell)

Tupeltypen

realisieren kartesische Produkte direkt

Beispiel

```
type name = (string, string)
someone :: name
```

Selektion

- durch Selektorfunktion `fst` und `snd`:

```
surname = fst someone ... familyname  = snd someone
```

- durch Mustervergleich:

```
where (surname, familyname) = someone
```

Verbunde

```
data Person = Person
   with surname, familyname: string


someone :: Person
someone = Person{familyname = "Watt",
                 surname    = "David"}
```

Selektion von Verbunden

```
surname (someone) ... familyname (someone)
```

Reihenfolge in Verbunden ist unwichtig

$$\{I_1: S_1,\ I_1: S_2\} = \{I_1: S_2,\ I_1: S_1\}$$

Vergleich von Tupeln und Verbunden

benannte Selektoren sind sicherer vor Verwechslungen

Tupel in ML

Beispiel

```
type person = string * string
someone : person
```

Selektion

- durch die Selektorfunktion # :

```
surname  =  #1 someone ... familyname= #2 someone
```

- durch Mustervergleich:

```
(surname, forename, age, height) = someone
```

Verbunde

```
type person =
   {surname: string, forename: string}
someone : person
```

Selektion von Verbunden

```
#surname (someone) ... #height (someone)
```

Tupel sind Abkürzungen von Verbunden

$$S_1 * S_2 * \ldots * S_n = \{1: S_1,\ 2: S_2,\ \ldots,\ n: S_n\}$$

Verbund in Ada

allgemein

die Komponenten werden mit einem <u>Selektor</u> markiert
(das ist ein Bezeichner)

```
record
    I1 : T1;
      …
    In : Tn
end
```

($I_1 : T_1; \ldots I_n : T_n$)

Beispiel (Ada)

```
type Date is record
                 m: Month;
                 d: Integer range 1..31;
             end record
```

Konstruktion

```
someday : Date:= (m => feb, d => 29);
xmas : Date:= (d => 24, m => dec);
```

Selektion

"dotnotation"

```
someday.d := 29; someday.m := feb
```

Selektionsoperator (Algol-68)

```
d of someday := 29; m of someday := feb
```

homogenes kartesisches Produkt, Unit

homogenes kartesisches Produkt

$S^n = S \times \ldots \times S$

Kardinalität

$\#(S^n) = (\#S)^n$

welchen Typ definiert der Spezialfall n=0 ?

Kardinalität

$\#(S^0) = \#S^0 = 1$

Der Typ S^0 definiert also einen Typ mit einem Element

dieses Element heißt Nulltupel und wird geschrieben als ()

Diesen Typ bezeichnen wir als Unit

Unit = { () }

Unit entspricht **void** in Algol-68 und C und `unit` in ML

dieser Typ wird dort benutzt, wo kein Wert erwartet wird

Betrachtungsweisen von Tupeln und Verbunden

- Tupel sind konstruktive Daten
  - `Person` besteht aus zwei Zeichenketten
- Verbunde können als abstrakte Daten angesehen werden
  - `Person` hat zwei Komponentne `surname` und `familyname`, die Zeichenketten sind

disjunkte Vereinigung

Definition

$S + T = \{ \text{left}\, x \mid x \in S \} \cup \{ \text{right}\, y \mid y \in T \}$

hierbei sind left und right <u>Markierungen</u> (der Herkunft)

Operationen

Abfrage der Markierung

Projektion auf Ausgangswert

Kardinalität

$\#(S + T) = \#S + \#T \}$

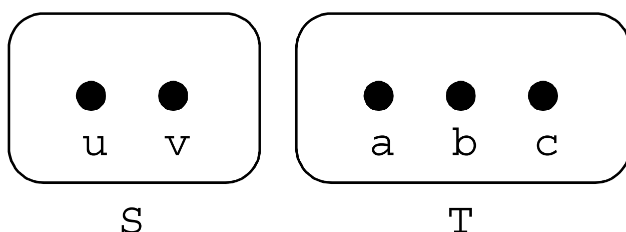
Markierung

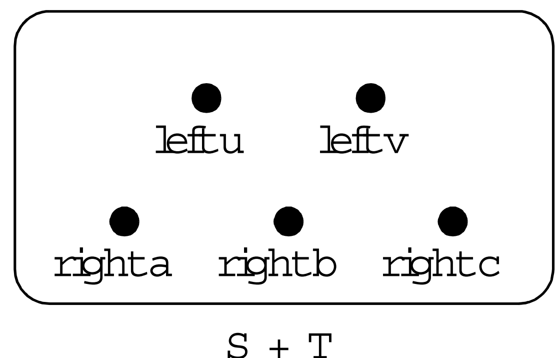
kann oft im Programm gewählt werden

Unterschied zur Mengenvereinigung

$$\begin{aligned} S \cup S &= S \\ &\neq \{ \text{left}\, x \mid x \in S \} \cup \{ \text{right}\, y \mid y \in S \} \\ &= S + S \end{aligned}$$

Weshalb gibt es keine normale Mengenvereinigung?





Vereinigungen in funktionalen Sprachen

Datentyp in Haskell

```
data Number = Exact Int | Approx Float
```

Selektion nur durch pattern matching

```
value :: Number -> Int
value (Exact n)    = n
value (Approx x)   = toInteger x
```

Konstruktion in ML

```
number = exact of int
       | approx of real
```

Selektion nur durch pattern matching

```
value (x) =  case x of
                exact n  = n
             |  approx x = floor (x)
```

varianter Verbund (Ada)

allgemein

Markierungen sind Elemente eines diskreten Typs M

record
   case M **is**
      when L_1 => I_1 : T_1;
      …
      when L_n => I_n : T_n;
end record;

Beispiel

```
type Accuracy is (exact, approx);
     number (acc: Accurracy) is
        record
        case acc of
          when exact  => ival: Integer;
          when approx => rval: Real
        end record;
subtype Float_Accuracy is number (approx);
```

Projektion durch Fallunterscheidung:

```
rounded : Integer;
...
case num.acc of
   when exact  => rounded := num.ival;
   when approx => rounded := round (num.rval)
end case
```

inkonsistente Benutzung ist möglich

Wenn `num.acc = exact` und `num.ival = 7`

ist `num.rval` undefiniert, kann aber trotzdem benutzt werden!

sichere Benutzung: wie oben, durch Fallunterscheidung!

Variante Verbunde versus algebraische Typen

Vergleich von Varianten und Konstruktionen

Selektion durch pattern matching verhindert inkonsistente Benutzung

ersparen Typüberprüfungen zur Laufzeit

Verbunde vermischen Produkt und Vereinigung:

```
type Shape   is (point, circle, box);
     Figure  is record (figureshape: Shape)
                x, y, Real;
                  case figureshape of
                  when point => null;
                  when circle =>radius: Real;
                  when box => height, width : Real;
                  end case
                end record;
```

$$Shape = Real \times Real \times (Unit + Real + (Real \times Real))$$

Potenzmenge

Definition

$2^S = \{ s \mid s \text{ ist Teilmenge von } S \}$

Operationen

mengentheoretisch: Enthaltensein, Vereinigung, Durchschnitt ...

Kardinalität

$\#(2^S) = 2^{\#S}$

Beispiel (Pascal, nur für einfache diskrete Elementtypen)

```
type  Basecolor  = (red, green, blue);
      Color      = set of Basecolor;
```

Benutzung

```
var c1, c2: Color;
...
c1 := c1 + c2 (* positives Mischen *)
c2 := c1 * c2 (* negatives Mischen *)
```

Mögliche Erweiterungen?

Mengen <u>beliebigen</u> Elementtyps
werden nur von einigen Sprachen unterstützt (SetL, Snobol)

Grund

- die Implementierung ist <u>aufwendig</u>
- die Implementierung ist <u>nicht kanonisch</u>
(es hängt von der Anwendung ab, welche die "beste" ist)

Ada: nicht direkt gegeben, aber generisch definierbar

Relation

Definition

jede Teilmenge von $S \times T$ ist eine <u>Relation</u> R auf $S \times T$

$$S \leftrightarrow T = 2^{S \times T}$$

Schreibweise

$$(x,y) \in R \Leftrightarrow R\ (x,y) \Leftrightarrow x\,R\,y$$

alternative Sichtweise von Relationen

das <u>charakteristische Prädikat</u> p_R einer Relation R

ist eine Funktion des Typs $S \times T \rightarrow$ Truth-value

die true liefert, genau dann wenn die Relation R gilt

$$p_R\ (x,y) = \text{true} \Leftrightarrow x\,R\,y$$

Bemerkung

Relationen und Prädikate sind in logischen Sprachen wichtig

Abbildung

Definition

Eine Relation m auf $S \times T$ ist eine <u>Abbildung</u> wenn gilt:

linkstotal: $\forall x \in S\ \exists\ (x,y) \in m$

rechtseindeutig: $\forall\ (x,y), (x',y') \in m : x=x' \Rightarrow y=y'$

Schreibweisen

$$m : S \to T \quad \Leftrightarrow \quad m \in S \to T$$
$$(x,y) \in m \quad \Leftrightarrow \quad y = m(x)$$

Die Menge aller Abbildungen

$$S \to T = \{ m \in S \leftrightarrow T \mid m \text{ ist linkstotal und rechtseindeutig} \}$$

Operationen

Abbildungsanwendung

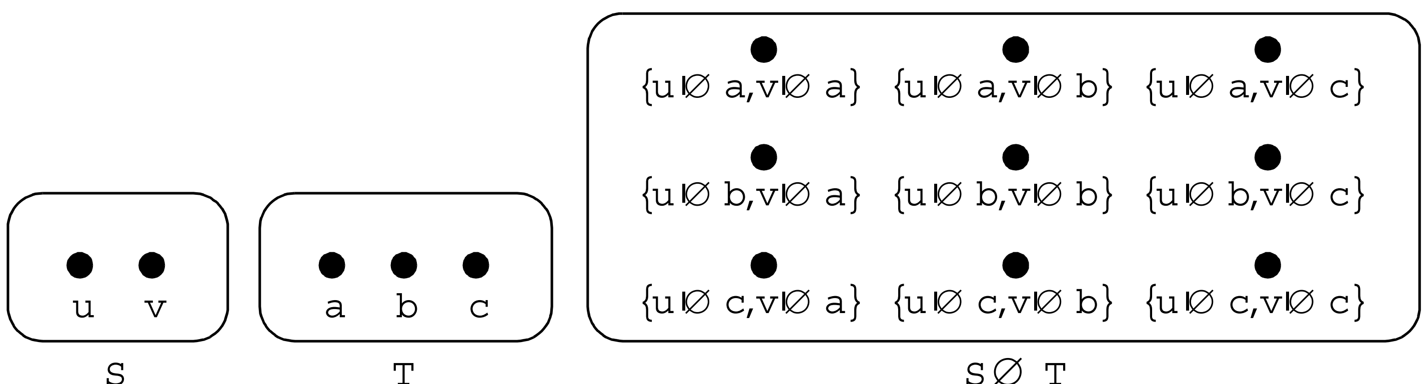
Kardinalität

$$\#(S \to T) = (\#T)^{\#S}$$

Verallgemeinerung auf n-stellige Abbildungen

Argument (bzw. Indexmenge) ist ein kartesisches Produkt

$$m : S_1 \times \ldots \times S_n \to T$$



Funktion

Funktion in Haskell

```
even :: Int -> Int
even = \n -> (n mod 2 == 0)
```

Abbildungsanwendung

```
even 3      even (2*n+1)
```

sind Funktionen Werte?

in funktionalen Sprachen: JA

Ausdrücke können Funktionen <u>berechnen</u> und in Werte <u>einbauen</u>

```
integer_functions :: [Int -> Int]
integer_functions
   = [(+ 1), (- 1), (-), \n -> (n mod 2 == 0)]
```

in imperativen Sprachen: meist nur <u>eingeschränkt</u>:

- Funktionen können als Argumente an Funktionen übergeben werden
- Funktionen sind Komponenten von Klassen (Methoden)

Funktion in Pascal

```
function even (n: Integer) : Boolean;
   begin
   even := (n mod 2 = 0)
   end;
```

Abbildungsanwendung

```
even(2*n+1)
```

Beachte

Verschiedene Funktionsdefinitionen von `even` sind möglich!

Pascal erlaubt <u>mehr</u> als nur Abbildungen (Nebeneffekte)!

Feld

Feldtyp in Pascal (S muß diskret sein)

```
type  Color = (red, green, blue);
      Pixel = array [Color] of 0..1;
```

Abbildungsanwendung

```
var point : Pixel;
...
point[red]
```

Feste, offene und flexible Felder

später bei der Behandlung von Variablen

mehrdimensionale Felder

Indexmenge ist ein kartesisches Produkt

$m : S_1 \times \ldots \times S_n \rightarrow T$

das geht in Pascal direkt nicht (weshalb eigentlich?)

Feldtyp in Haskell (S muß geordnet sein)

```
coeff :: Array Int Float
```

Abbildungsanwendung

```
coeff = array (0,4) [1.0, 3.4, 5.0, 0.0, -0.7]

... coeff!i ...
```

Liste (homogen)

Definition mit rekursiver Typgleichung

$$\text{Integer-List} = \texttt{Unit} + (\text{Integer} \times \text{Integer-List})$$

$$\text{Integer-List} = \{\ \texttt{nil}()\ \} \cup \{\ \texttt{cons}(i,x)\ |\ i \in \text{Integer}, x \in \text{Integer-List}\ \}$$

Wertemenge

alle endlichen Listen (kleinste Lösung der rekursiven Gleichung)

Operationen

empty? head, tail.

Kardinalität

unendlich

Abkürzung

$$s^* = \text{Unit} + (s \times s^*)$$

Listen in ML und Haskell

Definition als algebraischer Datentyp

```
intlist =  nil   of ()
        |  cons  of int * intlist
```

Werte von `intlist`

```
nil
cons(1, nil)
cons(1, cons(7, cons(2 , cons(1, nil))))
...
```

tatsächlich (in ML)

`list` ist ein parameterisierter Typkonstruktor

```
intlist ≡ int list
```

mehr syntaktischer Zucker in Haskell

```
intlist             ⇔  [Int]
nil                 ⇔  []
cons                ⇔  :
1: 7: 2: 1: []      ⇔  [1, 7, 2, 1]
```

Listendarstellung mit Zeigern

Ada

```
type intlist is access cell;
type cell is
   record
      head: Integer;
      tail: intlist;
   end record;
procedure cons (in x: Integer; in l: intlist)
         return intlist is
   begin
   return new cell(head => x; tail => l);
   end cons;
```

Darstellung von Listenwerten

```
null
cons(1, null)
cons(1, cons(7, cons(2 , cons(1, null))))
...
```

Unterschied zum funktionalen Typ

nicht parameterisiert

Zugriff auf Zeigerkomponenten ohne Abfrage auf **null** nicht verboten.

es können zyklische Datenstrukturen aufgebaut werden:

```
list :  intlist
     := cons(1, cons(7, cons(2 , cons(1, null))));

list^.tail^.tail^.tail^.tail := list;
```

allgemeiner rekursiver Typ

Definition durch rekursive Typgleichungen (mit + und ×)

$$
\begin{aligned}
T = \ & (S_{11} \times \ldots \times S_{k\,k1}) \\
+ \ & (S_{21} \times \ldots \times S_{2\,k2}) \\
+ \ & \ldots \\
+ \ & (S_{n1} \times \ldots \times S_{n\,kn})
\end{aligned}
$$

Operationen

Variantenabfrage, Komponentenzugriff.

Kardinalität

unendlich

rekursive Potenzmengen führen zu Widersprüchen

rekursive Abbildungstypen sind möglich

$$M = M \rightarrow M$$

Dann braucht man einen spezielleren Typbegriff (partiell geordnete Mengen)

wichtig bei der denotationellen Semantik

zum Glück aber nicht im Programmieralltag!

algebraische Typen

Definition (Haskell)

```
data Inttree ::= Leaf int
               |  Branch inttree inttree
```

Werte von **inttree**

```
Leaf 1
Branch (Leaf 1) (Leaf 1)
Branch (Branch (Leaf 1) (Leaf 2)) (Leaf 3)
...
```

in ML heißen diese Typen Konstruktionen

sind aber sonst ganz ähnlich

```
inttree  = leaf of int
         | branch of inttree * inttree
```

der Sonderfall Zeichenketten

Möglichkeit 1 (ML)

ein vordefinierter einfacher Typ

Möglichkeit 2 (Pascal, Modula, Ada)

ein Feld von Zeichen: **packed array** [1..max] **of** char

Nachteil: feste Längen

keine Konkatenation

Möglichkeit 3 (Algol-68)

ein <u>flexibles</u> Feld von Zeichen: **flex** [1..max] **of** char

Möglichkeit 4 (Haskell)

eine Liste von Zeichen: **type** string = [char]

Sonstige Datentypen

Sequentielle Dateien?

- **file of** $T \equiv T^*$ oder $(T^* \times T^*)$
- siehe auch Speicher

Zeiger

- sind keine Werte, sondern Adressen
- ein maschinennahes imperatives Konstrukt
- siehe Speicher

Datentypen abstrakt

Datentyp	Konstruktoren	Destruktoren	Operationen
alle		—	$=$ $\neq$
geordnete		—	pred succ < <= >= >
Boolean	true false	—	$\neg$ $\wedge$ $\vee$ $\Rightarrow$ $\Leftrightarrow$
Integer	..., -2, -1, 0, 1, 2, ...	—	+ - * div mod **
Character	'a', ... 'Z', '.' ...	ord	
Real	..., -1.0, ..., 0.0, 1.0, ...	round trunc	+ - * / **
$S \times T$	(x, y)	first second	
$S + T$	left(x) right(y)	asleft asright	isleft isright
$S \rightarrow T$	$\{x \mapsto a, y \mapsto b, z \mapsto a\}$	—	—
$\wp S$	$\{a, b, \ldots\}$		$\in$ $\subset$ $\cap$ $\cup$ $\setminus$

TYPISIERUNG

Typsystem

Regeln für den Umgang von Operationen mit Werten

<u>der</u> Unterschied zu maschinennahen Sprachen

statische Typisierung

Jeder Name für einen Wert (Variable, Parameter, ...)

hat im Programm einen Typ

- Typüberprüfungen bei der Übersetzung ("statisch")

```
function even (n: Integer) return Boolean is
   begin
   return (n mod 2 = 0)
   end even;

... even(i+1) ...
```

dynamische Typisierung

Jeder Wert hat bei der Ausführung einen Typ

- Typüberprüfungen bei der Ausführung ("dynamisch")

```
function even (n) is
   begin
   return (n mod 2 = 0)
   end even;

even(i+1) ...
```

statische Typisierung ist sicherer

und wird bei fast allen Sprachen benutzt (Ausnahmen: Lisp, Smalltalk)

Typgleichheit

Wann sind zwei Typen T und T′ gleich?

in statisch getypten Sprachen

strukturelle Gleichheit (Algol-68)

T ≡ T′ genau dann wenn T und T′ die gleiche Wertemenge haben

namentliche Gleichheit (Ada)

T ≡ T′ genau dann wenn T und T′

an der gleichen Stelle vereinbart wurden

BEISPIEL: A und C strukturell gleich, namens-ungleich

```
type A = array [1..10] of B;
     B = array [0..99] of Integer;
     C = array [1..10] of [0..99] of Integer;
```

Vor- und Nachteile

strukturelle Gleichheit

- hart zu überprüfen (z.B. bei rekursiven Typen)
- unverträglich mit dem Geheimnisprinzip (information hiding)

Namengleichheit

- manchmal zu streng (Zeichenketten)

Typvollständigkeit

Klassengesellschaft der Werte (Pascal)

1. Prozeduren und Funktionen können Argumente sein,
   aber <u>nicht</u> zugewiesen werden, und <u>nicht</u> in Werte eingefügt werden

2. zusammengesetzte Werte könnenn nicht Funktionsergebnisse sein

Typvollständigkeitsprinzip

keine Sprachkonstruktion für Werte
sollte <u>willkürlich</u> eingeschränkt sein

AUSDRÜCKE

Literal

Schreibweise von Werten einfachen Typs

`365`   `3.1415`   `'%'`   `'Lovelace'`

Aggregat

Konstruktion von Werten zusammengesetzten Typs

ein Wert vom Tupeltyp `(real, real)` (in Haskell)

```
(a*2.0, b/2.0)
```

ein Wert vom Feldtyp **array** Month **of** Natural (in Ada)

```
(31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31)
```

ein Wert vom Verbundtyp `Date` (in Ada)

```
(y =>1993, m =>'Oct', d => 24)
```

Ein Wert vom Typ `Inttree` (in Haskell)

```
Branch (Branch (Leaf 1) (Leaf 2)) (Leaf 3)
```

Funktionsaufruf, bedingter Ausdruck

Anwendung einer <u>Funktion</u> auf einen <u>aktuellen Parameter</u>

beide können von Ausdrücken geliefert werden (in ML, Algol-68)

```
(if a < 0 then sin else cos) (14.0*x)
```

Operationen sind infix geschriebene Funktionsaufrufe

spart Klammern, sonst nichts

in manchen Sprachen trotzdem <u>leicht</u> verschieden (overloading)

$$\oplus E \equiv \oplus(E)$$
$$E \otimes E' \equiv \otimes(E, E')$$

Typselektoren für zusammengesetzte Werte sind Operationen

- Projektion (kartesisches Produkt, disjunkte Vereinigung)
- Markierungsabfrage (disjunkte Vereinigung)
- Abbildungsanwendung (Feldindizierung, ...)
- ...

bedingter Ausdruck (Haskell)

Auswahl zwischen Werten (von Ausdrücken) gleichen Typs

```
if a < 0 then sin else cos

case thismonth of {
    Feb -> if leap(thisyear) then 29 else 28;
  | Apr -> 30;
  | Jun -> 30;
  | Sep -> 30;
  | Nov -> 30;
  |     -> 31;
}
```

auch **if** und **case** sind Operationen

weshalb können sie nicht immer in der Sprache ausgerückt werden?

benannte Konstanten und Variablen

Zugriff auf Werte benannter Konstanten

```
const pi = 3.1416;
...
... (2*pi*r) ...
```

Zugriff auf (momentane) Werte von Variablen

```
var r : Real;
...
... (2*pi*r) ...
```

Zusammenfassung: Werte, Typen und Ausdrücke

Typen sind Wertemengen

$P ::= \text{Truth-value} \mid \text{Integer} \mid \text{Real} \mid \text{Char} \ldots$

$$
\begin{aligned}
T ::= & \; P P \\
\mid & \; S \times T \\
\mid & \; S + T \\
\mid & \; S \rightarrow T \\
\mid & \; 2^{T}
\end{aligned}
$$

Typdisziplin erhöht die Sicherheit

statische Typisierung ist sicherer und effizienter als dynamische

strukturelle Typäquivalenz ist schwierig zu entscheiden (Rekursion)

namentliche Typäquivalenz verträgt sich besser mit Modularität

Prinzip der Typvollständigkeit

keine willkürlichen Einschränkungen von Operationen

Ausdrücke berechnen Werte (aus Werten)

Literale: Schreibweise für primitive Werte

Aggregate: Komposition von zusammengesetzten Werten

Zugriff auf benannte Konstanten und Variable

Funktionaufrufe:

- mehrstellige Funktionen sind einstellige Funktionen auf Tupeln
- Operatoren sind infix geschriebene Funktionsaufrufe
- Typselektoren sind vordefinierte Funktionen / Operatoren

bedingte Ausdrücke

Praktikum: einfache und zusammengesetzte Datentypen

Datentyp	C	Algol-68	Pascal	Ada	Haskell	ML	Eiffel
Unit	**void**	**void**		**null**	()		–
Truthvalue	–	bool	boolean	Boolean	Bool		
Integer	int	*L* int	integer	Integer	Integer Int		
Character	char	char	character	Character	Char		
Float	real	*L* real	real	Float	Float Double		
(a, …b)	–	–	ja	ja	ja	ja	ja
u ..o	–	–	ja	ja	–	–	–
sonstige					–		
$S \times T$	**struct**	**struct**	* **record**	**record**	(, ,) **…with …**	* **record**	
$S + T$	**union**	**union**	|	**record**	|	|	|
$S \rightarrow T$	[*n*]	[*n*]	**array**	**array**	array	**array**	
2^T	–	–	**set**	–	–		
sonstige			**file**		request	list	

Ausdrücke, Typgleichheit in einigen Programmiersprachen

Konstrukt	C	Algol-68	Pascal	Ada	Haskell	ML	Eiffel
Literal	ja	ja	ja	ja	ja	ja	ja
Aggregate	–	ja	`string`	ja	ja	ja	ARRAY
Funktionsanwendung							
Oparatoranwendung							
bedingter Ausdruck	`? :`	**if** **case**	–	–	**if** **case**	**if** **case**	–
Konstantenzugriff							
Variablenzugriff							
Typgleichheit							
strukturell gleich	alle	alle	`string`	**subtype**	**data**	**datatype** **abstype**	–
namentlich gleich		–	sonst	**type**	**type**	**type**	alle

Typvollständigkeit in Pascal und Ada

Pascal	Konstante Aggregate	Operand	Operator-Ergebnis	Argument	Funktions-Ergebnis	Komponente
einfach	•	•	•	•	•	•
Verbund / Feld				•		•
Zeichenkette	•	•		•		•
Menge		•	•	•		•
Datei						•
Zeiger		•		•	•	•
Referenz				•		
Abstraktion				•		

Ada	Konstante Aggregate	Operand	Operator-Ergebnis	Argument	Funktions-Ergebnis	Kompo-nente
einfach	•	•	•	•	•	•
zusammengesetzt	•	•	•	•	•	•
Zeiger	•	•	•	•	•	•
Referenz				•		
Abstraktion						

Typvollständigkeit in ML

ML	Konstante Aggregate	Operand	Operator-Ergebnis	Argument	Funktions-Ergebnis	Komponente
einfach	●	●	●	●	●	●
zusammengesetzt	●	●	●	●	●	●
Referenz	●	●	●	●	●	●
Funktion	●	●	●	●	●	●

Übungen

Realisiere folgende Abbildung als Feld und Funktion

`not : bool → bool`

In welchen Eigenschaften unterscheiden sich diese Definitionen, und allgemein Felder und Funktionsdefinitionen?