Programmierung 1

Vorlesung 9

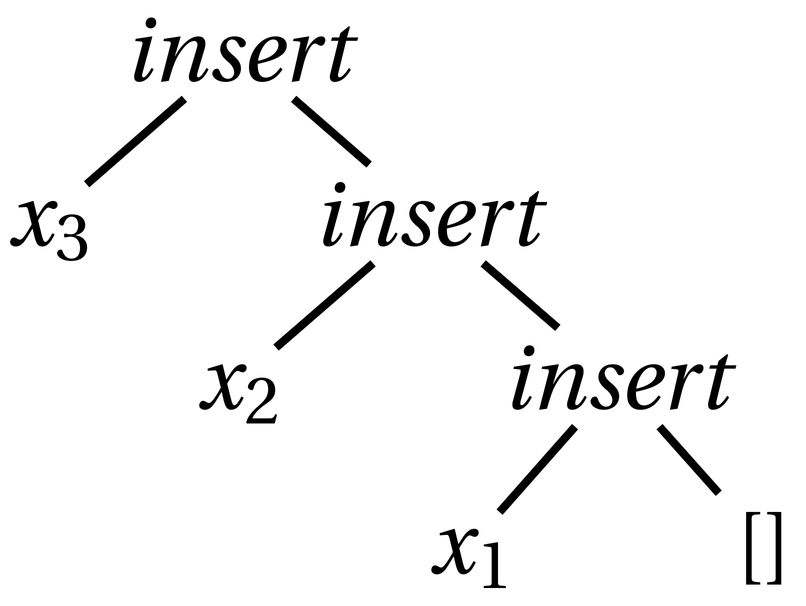
Livestream beginnt um 14:15 Uhr

Sortieren, Teil 2
Konstruktoren und Ausnahmen

Programmierung 1

Sortieren durch Einfügen

$sort\ [x_1, x_2, x_3] \ =$



$= \ foldl\ insert\ []\ [x_1, x_2, x_3]$

```
fun isort xs = foldl insert nil xs
```

val isort : int list → int list

Beispiel:

```
isort [1,3,1] = foldl insert nil [1,3,1]
              = foldl insert (insert(1,nil)) [3,1]
              = foldl insert [1] [3,1]
              = foldl insert (insert(3,[1])) [1]
              = foldl insert [1,3] [1]
              = foldl insert (insert(1,[1,3])) nil
              = foldl insert [1,1,3] nil = [1,1,3]
```

Sortieren durch Mischen (Mergesort)

- **Idee:**
  Drei Schritte:
  1. **Split:** Teile die Liste in zwei etwa gleich große Teile
  2. **Sort:** Sortiere die Teile
  3. **Merge:** Sortiere beim Zusammenfügen

- **Beispiel:** `sort [2,8,5,3]`

  - `split [2,8,5,3] = ([5,2], [3,8])`
  - `sort [5,2] = [2,5]`
  - `sort [3,8] = [3,8]`
  - `merge([2,5], [3,8]) = [2,3,5,8]`

Split: Teile in zwei etwa gleich große Listen

```
fun split xs = foldl (fn (x, (ys,zs)) => (zs, x::ys))
                     (nil, nil) xs
```

*val split : α list → α list * α list*

```
split [2,8,5,3]
      = foldl f (nil,nil) [2,8,5,3]
      = foldl f (f(2, (nil,nil))) [8,5,3]
      = foldl f (nil,[2]) [8,5,3]
      = foldl f (f(8,(nil,[2]))) [5,3]
      = foldl f ([2], [8]) [5,3]
      = foldl f (f(5, ([2], [8]))) [3]
      = foldl f ([8], [5,2]) [3]
      = foldl f (f(3, ([8], [5,2]))) nil
      = foldl f ([5,2], [3,8]))) nil
      = ([5,2],[3,8])
```

Abkürzung: `f = (fn (x,(ys,zs)) => (zs, x::ys))`

Merge: Sortiere beim Zusammenführen

```
fun merge (nil   , ys   ) = ys
  | merge (xs    , nil  ) = xs
  | merge (x::xr, y::yr) = if x<=y then x::merge(xr,y::yr)
                           else y::merge(x::xr,yr)
```

*val merge : int list * int list → int list*

```
merge ([2,5],[3,8])
      = 2::merge([5],[3,8])
      = 2::3::merge([5],[8])
      = 2::3::5::merge(nil,[8])
      = 2::3::5::[8]
      = [2,3,5,8]
```

Frage

Was ist das Ergebnis von merge([1,3], [3,2])?

- `[1,2,3]` ✗
- `[1,2,3,3]` ✗
- `[1,3,3,2]` ✓
- `[2,3,3,1]` ✗

```
fun merge (nil   , ys   ) = ys
  | merge (xs    , nil  ) = xs
  | merge (x::xr, y::yr) = if x<=y then x::merge(xr,y::yr)
                           else y::merge(x::xr,yr)
```

*val merge : int list * int list → int list*

Sortieren durch Mischen (Mergesort)

```
fun msort []  = []
  | msort [x] = [x]
  | msort xs  = let val (ys,zs) = split xs
                in  merge(msort ys, msort zs) end
```

val msort : int list → int list

```
msort ([2,8,5,3])
= merge(msort [5,2], msort [3,8])
= merge(merge(msort [5], msort [2]), msort [3,8])
= merge(merge([5],[2]), msort [3,8])
= merge([2,5], merge(msort [3],msort [8]))
= merge([2,5], merge([3],[8]))
= merge([2,5],[3,8])
= [2,3,5,8]
```

Rekursionsbaum

$msort\ [5,6,6,4,3,2,1]$

$msort\ [2,4,6]$ $\quad$ $msort\ [1,3,6,5]$

$msort\ [4]$ $\quad$ $msort\ [6,2]$ $\quad$ $msort\ [6,1]$ $\quad$ $msort\ [5,3]$

$msort\ [6]$ $\quad$ $msort\ [2]$ $\quad$ $msort\ [6]$ $\quad$ $msort\ [1]$ $\quad$ $msort\ [5]$ $\quad$ $msort\ [3]$

Lineare vs. binäre Rekursion

- **Lineare Rekursion:** **ein** rekursiver Aufruf

  Rekursions**folge:**

  potenz(4,3) → *potenz*(4,2) → *potenz*(4,1) → *potenz*(4,0)

  insert(3,[1,2,4]) → *insert*(3,[2,4]) → *insert*(3,[4])

- **Binäre Rekursion:** **zwei** rekursive Aufrufe

  Rekursions**baum:**

  - $\mathit{msort}\,[5,6,6,4,3,2,1]$
    - $\mathit{msort}\,[2,4,6]$
      - $\mathit{msort}\,[4]$
      - $\mathit{msort}\,[6,2]$
        - $\mathit{msort}\,[6]$
        - $\mathit{msort}\,[2]$
    - $\mathit{msort}\,[1,3,6,5]$
      - $\mathit{msort}\,[6,1]$
        - $\mathit{msort}\,[6]$
        - $\mathit{msort}\,[1]$
      - $\mathit{msort}\,[5,3]$
        - $\mathit{msort}\,[5]$
        - $\mathit{msort}\,[3]$

- **Baumrekursion:** **nicht**lineare Rekursion

Mengen

- Eine **Menge** ist eine Zusammenfassung von mathematischen Objekten.
- Die zu einer Menge zusammengefassten Objekte werden als die **Elemente** der Menge bezeichnet.
- Zu endlich vielen Objekten $x_1,\dots,x_n$ existiert stets genau eine Menge, die genau diese Objekte als Elemente hat.
  Diese Menge wird mit $\{x_1,\dots,x_n\}$ bezeichnet.
- **Leere Menge**: $\varnothing$
- Notation $x \in X$: das Objekt x ist ein Element der Menge X.
- Man sagt, dass eine Menge ihre Elemente **enthält**
  oder auch dass eine Menge aus ihren Elementen **besteht**.

Mengen

- **Gleichheitsaxiom:**
  Zwei Mengen X und Y sind genau dann **gleich** ($X = Y$),
  wenn jedes Element **von *X*** ein Element **von *Y*** ist
  und jedes Element **von *Y*** ein Element **von *X*** ist.

- Eine Menge ist vollständig durch ihre Elemente **bestimmt**.

- **Ordnung** spielt keine Rolle: $\{1,2\} = \{2,1\}$

- Elemente können nicht **mehrfach** auftreten: $\{1,1,2\}=\{1,2\}$

- **Mengen ≠ Listen**

Darstellung von Mengen durch Listen

- Die Gleichung $\mathrm{Set}[x_1,\ldots,x_n]=\{x_1,\ldots,x_n\}$
  weist jeder Liste eine Menge zu.

- Darstellung von Mengen durch Listen ist **nicht eindeutig**:
  Set[1,2] = Set [2,1] = Set[1,1,2] = {1,2}.

- Darstellung von Mengen durch **strikt sortierte Listen**
  ist eindeutig.

Strikt sortierte Listen

- Die Prozedur *issort* sortiert eine Liste und eliminiert dabei Mehrfachauftreten von Elementen:

```
fun sinsert (x,nil)    = [x]
  | sinsert (x,y::yr) = case Int.compare(x,y) of
                          LESS  => x::y::yr
                        | EQUAL => y::yr
                        | _     => y::sinsert(x,yr)

val issort = foldl sinsert nil
```

Mengen

- X ist eine **Teilmenge** von Y, in Zeichen $X \subseteq Y$,
  wenn jedes Element von X ein Element von Y ist.
  $X = Y$ genau dann, wenn $X \subseteq Y$ und $Y \subseteq X$.

- Der **Schnitt** $X \cap Y$ ist die Menge, die genau aus den Objekten besteht, die sowohl Element von X als auch Element von Y sind.

- Die **Vereinigung** $X \cup Y$ ist die Menge, die genau aus den Objekten besteht, die Element mindestens einer der Mengen X und Y sind.

- Die **Differenz** $X - Y$ ist die Menge, die genau aus den Elementen von X besteht, die **keine** Elemente von Y sind.

Diff

```
    fun diff nil ys = nil
   | diff xs nil = xs
   | diff (x::xr) (y::yr) =
       case Int.compare(x,y) of
           LESS => x :: diff xr (y::yr)
         | GREATER => diff (x::xr) yr
         | EQUAL => diff xr yr
```

```
diff [1,2,4] [1,3,4]
= diff [2,4] [3,4]
= 2 :: diff [4] [3,4]
= 2 :: diff [4] [4]
= 2 :: diff nil nil
= 2 :: nil
= [2]
```

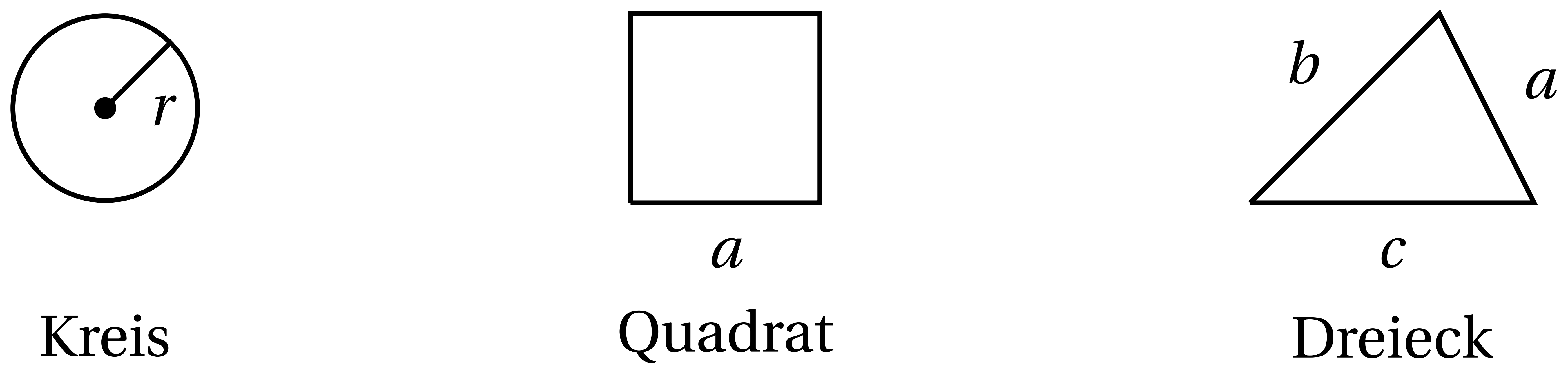
Union

```
fun union nil ys = ys
  | union xs nil = xs
  | union (x::xr) (y::yr) =
        case Int.compare(x,y) of
              LESS => x :: union xr (y::yr)
            | GREATER => y :: union (x::xr) yr
            | EQUAL => union xr (y::yr)
```

```
union [1,2,4] [1,3,4]
= union [2,4] [1,3,4]
= 1 :: union [2,4] [3,4]
= 1 :: 2 :: union [4] [3,4]
= 1 :: 2 :: 3 :: union [4] [4]
= 1 :: 2 :: 3 :: union nil [4]
= 1 :: 2 :: 3 :: [4]
= [1,2,3,4]
```

Kapitel 6
Konstruktoren und Ausnahmen

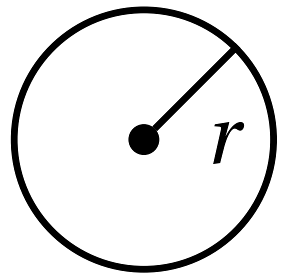
Ein neuer Typ für geometrische Objekte



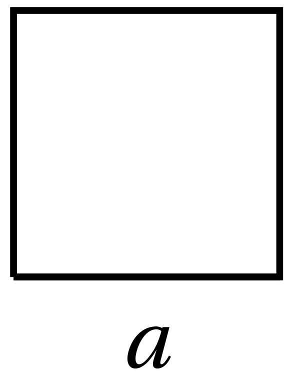

```
datatype shape =
    Circle   of real
  | Square   of real
  | Triangle of real * real * real
```

Interne Darstellung von Konstruktortypen

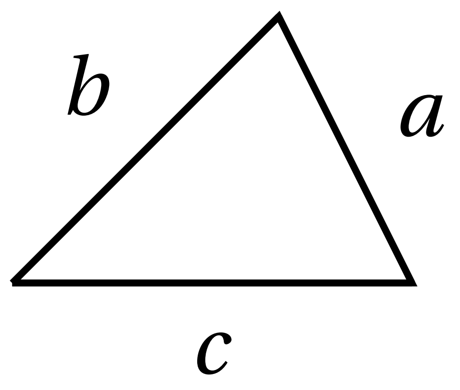
- **Interne Darstellung** als **Paare:**



`Circle r`
`(1,r)`



`Square a`
`(2,a)`



`Triangle (a,b,c)`
`(3, (a,b,c))`

- Die erste Komponente der Paare ist die **Variantennummer**, die zweite das **Datum**.

Konstruktortypen

```
datatype shape =
    Circle   of real
  | Square   of real
  | Triangle of real * real * real
```

- Typen die mit dem Schlüsselwort *datatype* deklariert sind, heißen **Konstruktortypen.**
- Die Werte eines Konstruktortyps werden mithilfe von **Konstruktoren** beschrieben.
- **Konstruktoren** können wie **Prozeduren** verwendet werden:

$$\begin{aligned} Circle &: real \to shape \\ Square &: real \to shape \\ Triangle &: real * real * real \to shape \end{aligned}$$

Konstruktortypen

- **Konvention:**
  Konstruktoren beginnen mit **Großbuchstaben**
  Typen mit **Kleinbuchstaben**

- **Baumdarstellung:**

$$\begin{tikzcd} \mathit{Circle} \arrow[d, dash] \\ 4.0 \end{tikzcd} \qquad \begin{tikzcd} \mathit{Square} \arrow[d, dash] \\ 3.0 \end{tikzcd} \qquad \begin{tikzcd} & \mathit{Triangle} \arrow[dl, dash] \arrow[d, dash] \arrow[dr, dash] & \\ 4.0 & 3.0 & 5.0 \end{tikzcd}$$

- **Gleichheit:** Ein Konstruktortyp erlaubt den **Gleichheitstest**, wenn dies die Argumenttypen **aller** seiner Konstruktoren tun.

Konstruktoren und Muster

```
fun area (Circle r)          = Math.pi*r*r
  | area (Square a)          = a*a
  | area (Triangle(a,b,c)) = let val s = (a+b+c)/2.0
                             in Math.sqrt(s*(s-a)*(s-b)*(s-c))
                             end
```

val area : shape → real

```
area (Square 3.0)
```

9.0 : real

```
area (Triangle(6.0, 6.0, Math.sqrt 72.0))
```

18.0 : real

Frage

Welchen Typ hat die Abstraktion fn (x => area (Triangle x))**?**

- real → real
- shape → real
- real * real * real → real
- real → real → real → real

Enumerationstypen

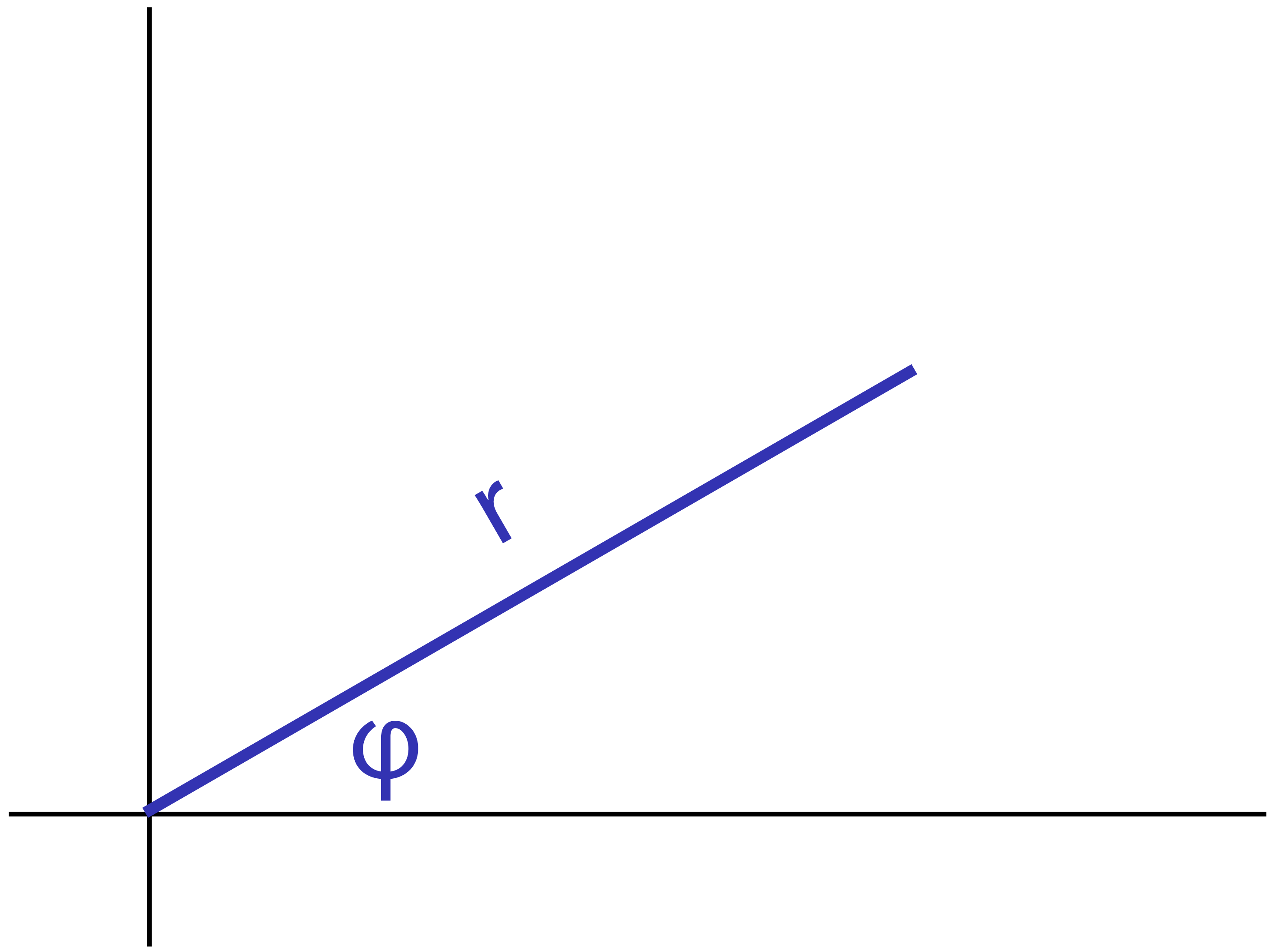
- Konstruktortypen mit **ausschließlich nullstelligen Konstruktoren** heißen **Enumerationstypen:**

```
datatype bool  = false | true

datatype order = LESS | EQUAL | GREATER

datatype day = Monday | Tuesday | Wednesday
             | Thursday | Friday | Saturday | Sunday
```

- Es gibt **nullstellige** und **einstellige** Konstruktoren, aber keine **mehrstelligen** Konstruktoren.

Typsynonyme

```
type point = real*real

type dist = real

type angle = real

type radial = dist * angle
```



Darstellung arithmetischer Ausdrücke

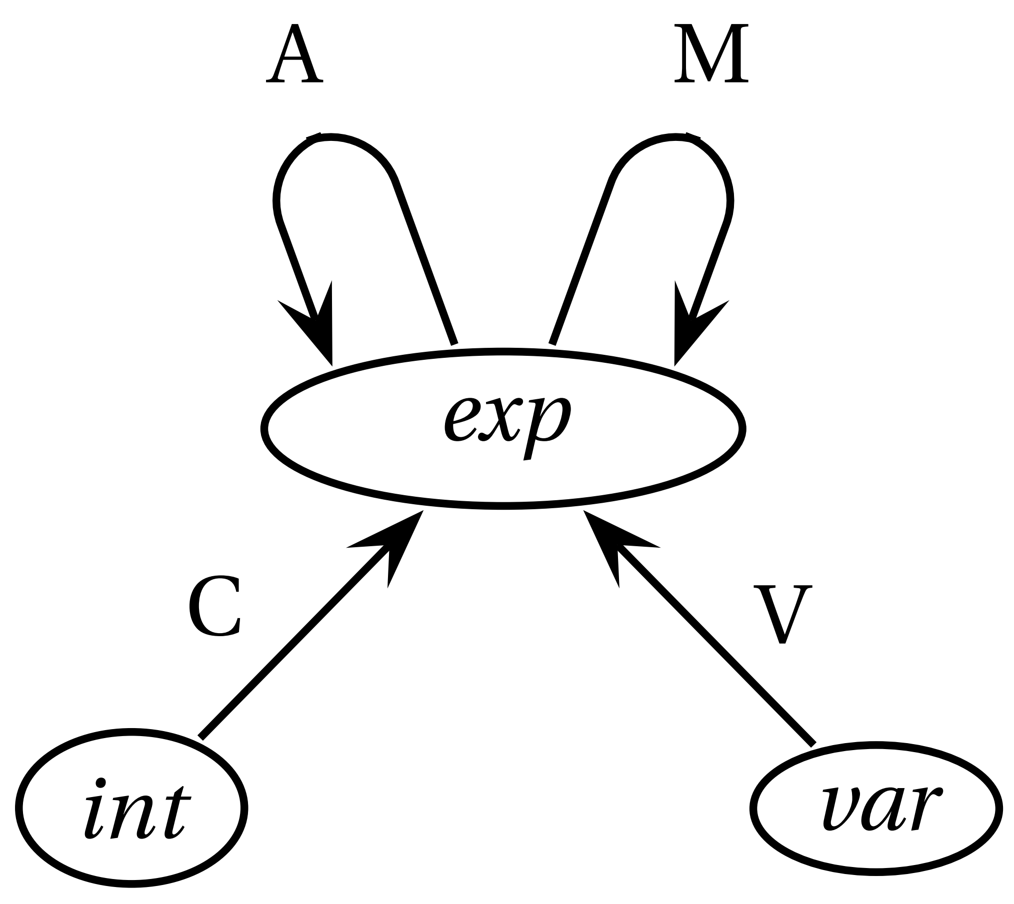
Mit Hilfe von Konstruktortypen können **syntaktische Objekte** als **Werte dargestellt werden.**

Beispiel: Arithmetische Ausdrücke (gebildet aus Konstanten, Variablen, Summe und Produkt)

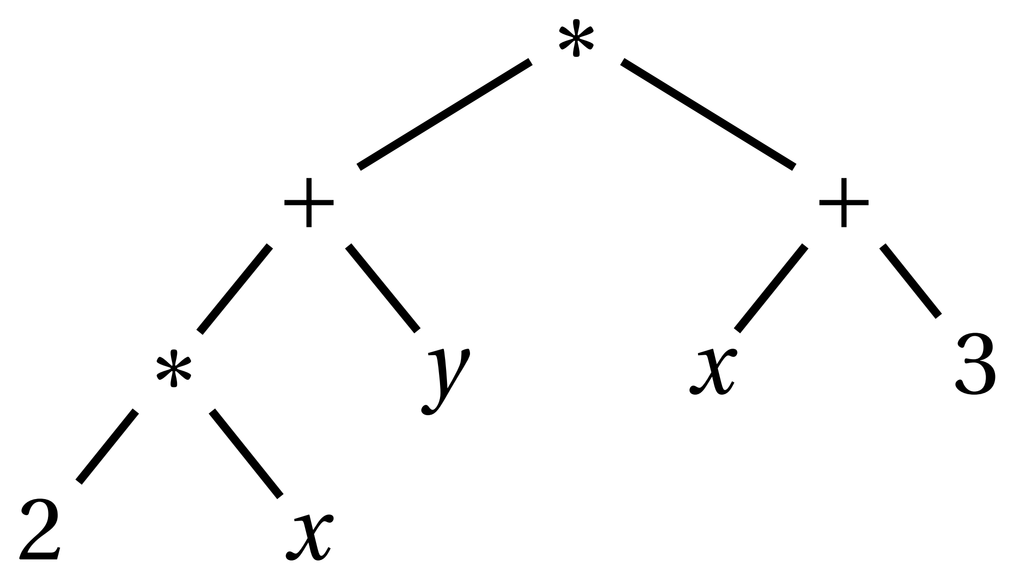
```
type var = string

datatype exp = C of int
             | V of var
             | A of exp * exp
             | M of exp * exp
```

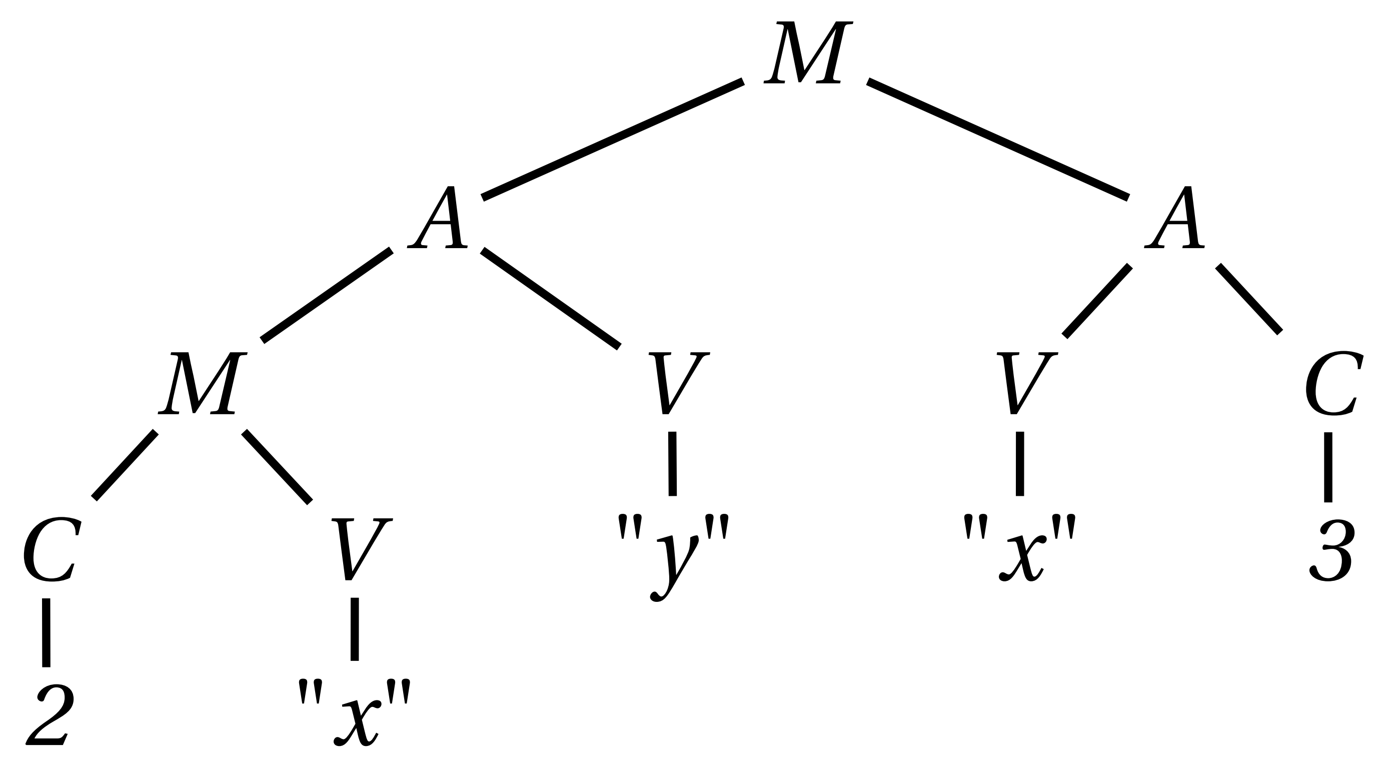
Rekursive Typdeklaration

```
type var = string

datatype exp = C of int
             | V of var
             | A of exp * exp
             | M of exp * exp
```




```
val e = M(A(M(C 2, V "x"), V "y"), A(V "x", C 3))
```





Komponenten arithmetischer Ausdrücke

```
type var = string

datatype exp = C of int
             | V of var
             | A of exp * exp
             | M of exp * exp


fun components (A(e,e')) = [e, e']
  | components (M(e,e')) = [e, e']
  | components     _     = nil
```

val components : exp → exp list

```
components (A(C 3, V "z"))
```

[C 3, V "z"] : exp list

Teilausdrücke

```
type var = string

datatype exp = C of int
             | V of var
             | A of exp * exp
             | M of exp * exp
```

```
fun subexps e = e::
    (case e of
          A(e1,e2) => subexps e1 @ subexps e2
        | M(e1,e2) => subexps e1 @ subexps e2
        |    _     => nil)
```

val subexps : exp → exp list

toString

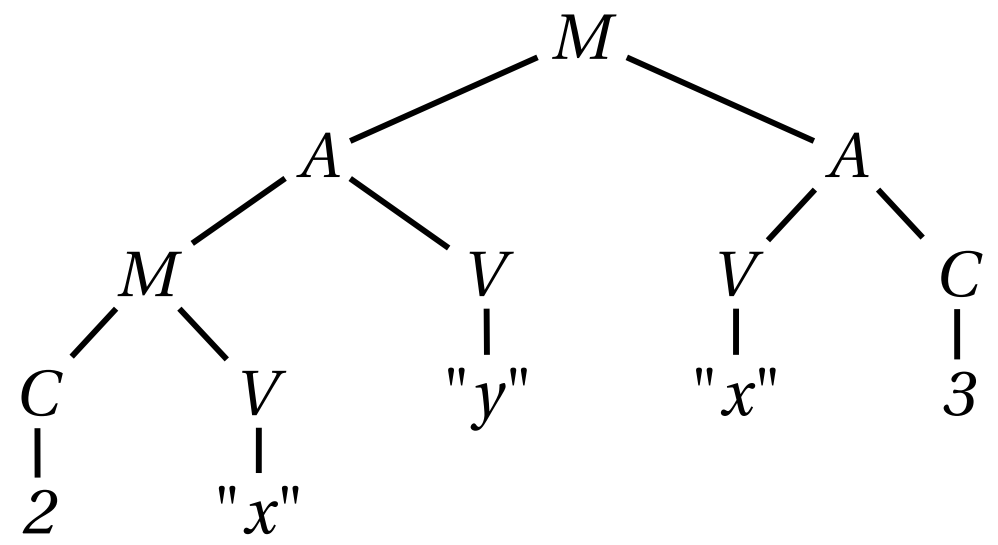
```
type var = string

datatype exp = C of int
             | V of var
             | A of exp * exp
             | M of exp * exp
```

```
fun toString (C r)     = Int.toString r
  | toString (V x)     = x
  | toString (A (x,y)) = "("^toString x ^ "+" ^ toString y ^")"
  | toString (M (x,y)) = "("^toString x ^ "*" ^ toString y ^")"
```

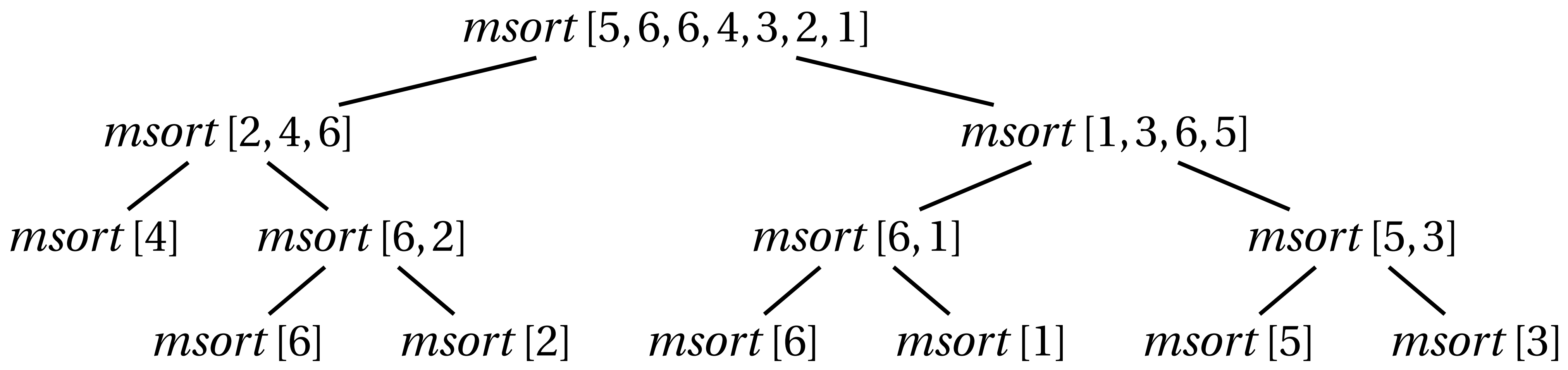
Strukturelle Rekursion

- **Strukturelle Rekursion:** Rekursion über die **Struktur** des **Datentyps**
- Rekusion in *foldl* ist **strukturell:**

  foldl op+ 0 [1,2,3] → *foldl* op+ 1 [2,3] → *foldl* op+ 3 [3] → *foldl* op+ 6 nil
- Rekursion in *subexp* und *toString* ist **strukturell:**



  Rekursion in *msort* ist **nicht** strukturell:



www.prog1.saarland