

Programmierung 1

Vorlesung 6

Livestream beginnt um 10:20 Uhr

Höherstufige Prozeduren, Teil 3

Listen und Strings

Programmierung 1

Polymorphe Typisierung

- **Typschema** beschreibt die **möglichen Typen**:

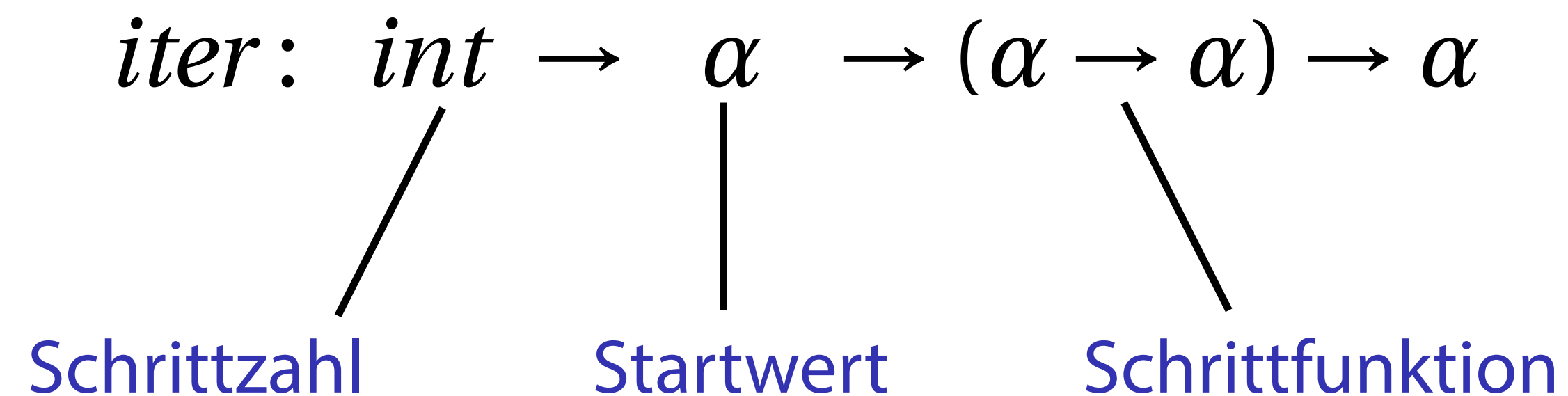
$$\forall \alpha. \text{int} \rightarrow \alpha \rightarrow (\alpha \rightarrow \alpha) \rightarrow \alpha$$

- **Typvariable** α ist durch \forall **quantifiziert**.

- Die **Instanzen des Schemas** sind die möglichen Typen:

$\text{int} \rightarrow \text{int} \rightarrow (\text{int} \rightarrow \text{int}) \rightarrow \text{int}$	$(\alpha = \text{int})$
$\text{int} \rightarrow \text{real} \rightarrow (\text{real} \rightarrow \text{real}) \rightarrow \text{real}$	$(\alpha = \text{real})$
$\text{int} \rightarrow \text{int} * \text{int} \rightarrow (\text{int} * \text{int} \rightarrow \text{int} * \text{int}) \rightarrow \text{int} * \text{int}$	$(\alpha = \text{int} * \text{int})$

Polymorphes Iter



Typvariable α steht für **beliebigen Typ**.

(**Lexikalische Syntax**: Typvariablen sind eigene Klasse von Wörtern, die mit dem Hochkomma beginnen.)

```
fun 'a iter (n:int) (s:'a) (f:'a->'a) : 'a =  
    if n<1 then s else iter (n-1) (f s) f  
val  $\alpha$  iter : int  $\rightarrow$   $\alpha \rightarrow (\alpha \rightarrow \alpha) \rightarrow \alpha$ 
```

Typinferenz

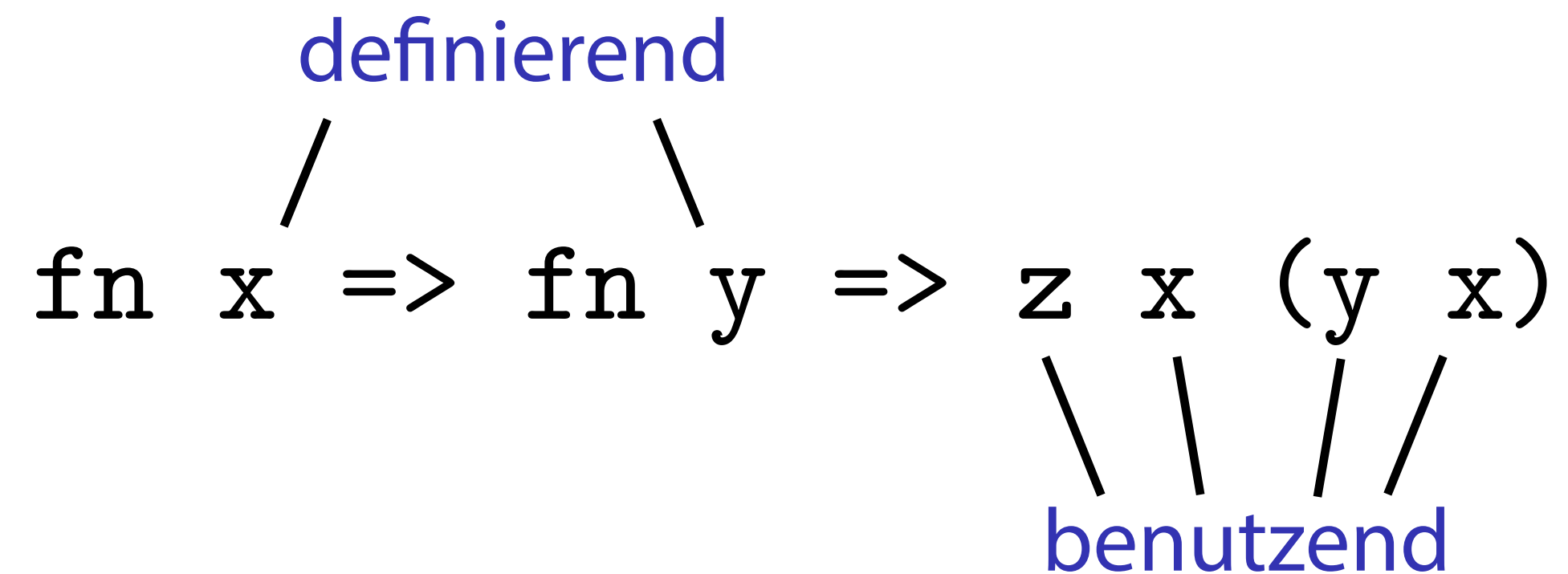
- ▶ **Typangaben für Argumentvariablen und Ergebnisse von Prozeduren können in Standard ML meist weggelassen werden.**
- ▶ **Typinferenz:** Automatisches Verfahren zur Ergänzung fehlender Typen.

```
fun 'a iter (n:int) (s:'a) (f:'a -> 'a) : 'a =  
  if n<1 then s else iter (n-1) (f s) f  
   $\forall \alpha \text{ int} \rightarrow \alpha \rightarrow (\alpha \rightarrow \alpha) \rightarrow \alpha$ 
```

```
fun iter n s f =  
  if n<1 then s else iter (n-1) (f s) f  
   $\forall \alpha \text{ int} \rightarrow \alpha \rightarrow (\alpha \rightarrow \alpha) \rightarrow \alpha$ 
```

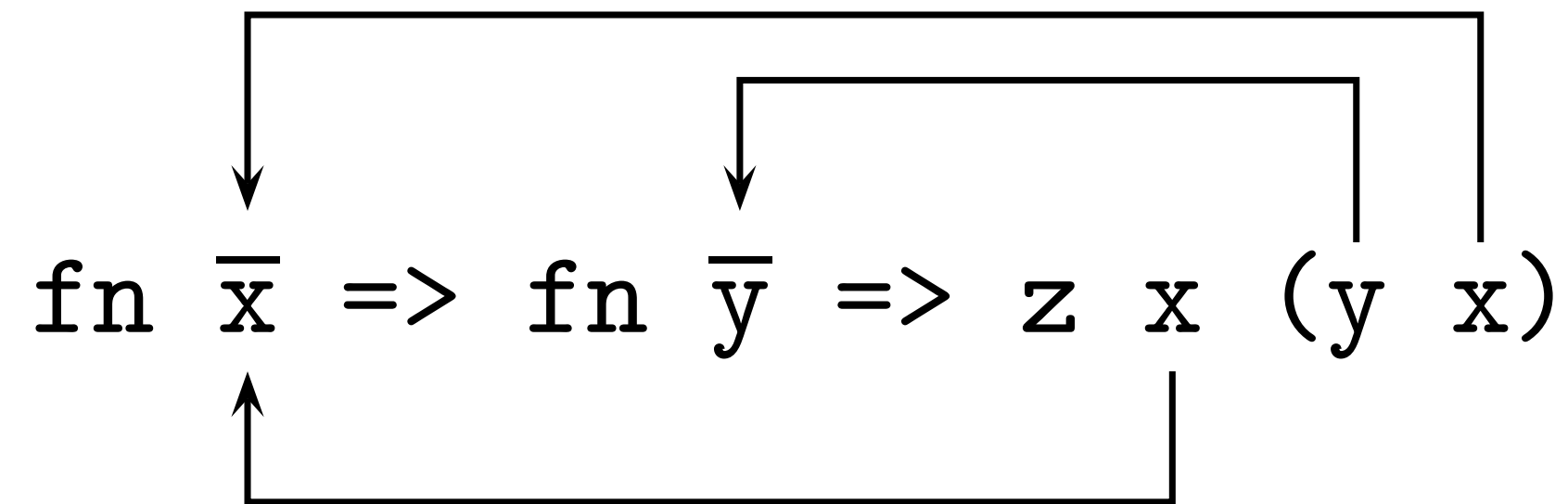
```
fun 'a iter n s f =  
  if n<1 then (s:real) else iter (n-1) (f s) f  
   $\forall \alpha \text{ int} \rightarrow \text{real} \rightarrow (\text{real} \rightarrow \text{real}) \rightarrow \text{real}$ 
```

Bezeichnerbindung: Lexikalische Bindungen



- ▶ **Definierende Bezeichnerauftreten** führen **neue Bindungen** ein
- ▶ **Benutzende Bezeichnerauftreten** liefern **Werte** gemäß bestehender Bindungen

Bezeichnerbindung: Lexikalische Bindungen



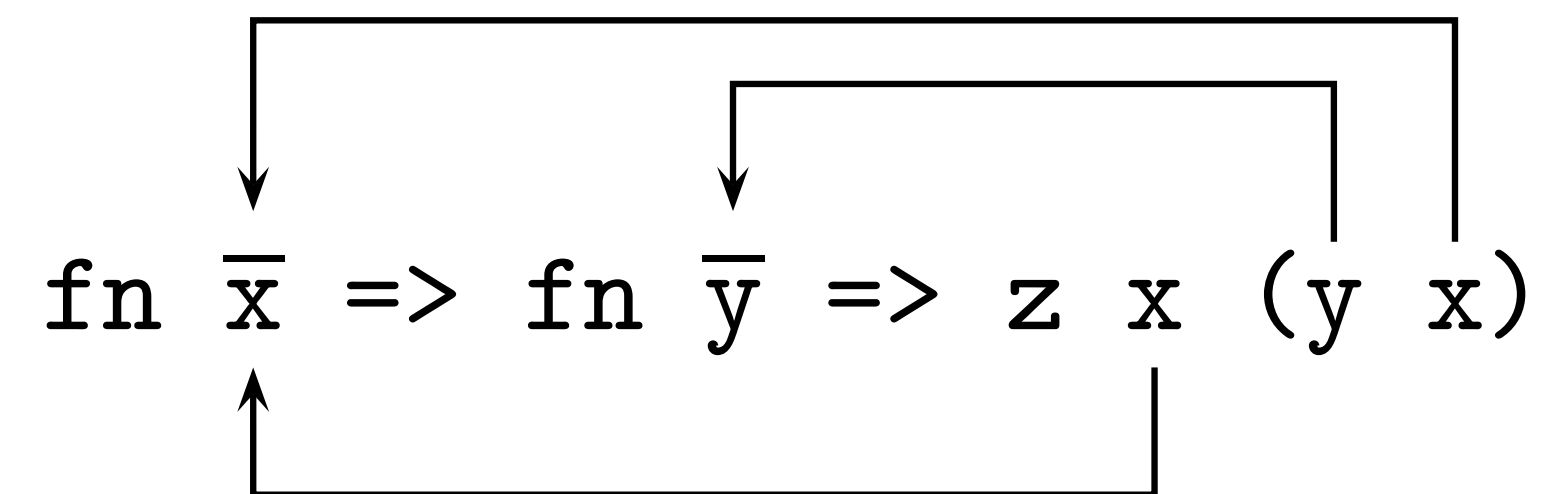
- ▶ Ein **definierendes** Bezeichnerauftreten ist **immer** auch ein **gebundenes** Bezeichnerauftreten.
- ▶ Ein **benutzendes** Bezeichnerauftreten ist **gebunden**, wenn ihm ein **definierendes** Bezeichnerauftreten **zugeordnet** werden kann und **ansonsten frei**.
Diese Zuordnung heißt **lexikalische Bindung**.
- ▶ (Darstellung: wir überstreichen definierende Benutzerauftreten und stellen die lexikalische Bindung durch Pfeile dar.)

Bezeichnerbindung: Bereinigung

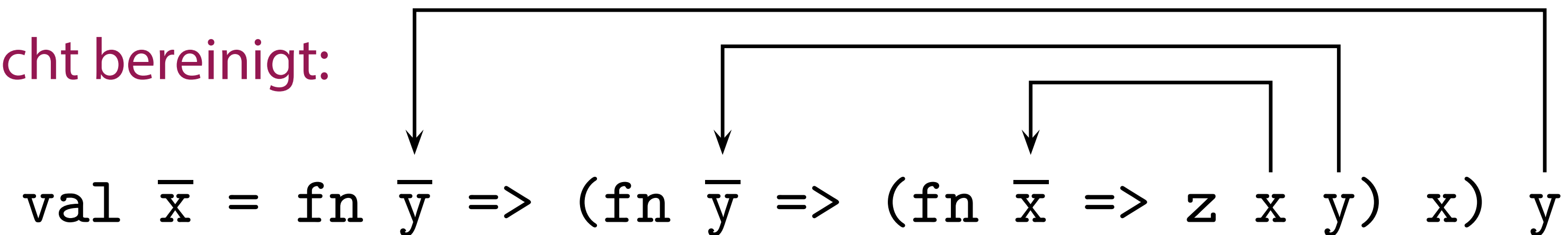
Eine Phrase heißt **bereinigt**, wenn

1. **kein** Bezeichner **mehr als ein definierendes** Auftreten hat, und
2. **kein** Bezeichner **sowohl** ein **definierendes** als auch ein **freies** Auftreten hat.

Bereinigt:



Nicht bereinigt:



Bezeichnerbindung: Bereinigung

$\text{val } \bar{x} = \text{fn } \bar{y} \Rightarrow (\text{fn } \bar{y} \Rightarrow (\text{fn } \bar{x} \Rightarrow z \ x \ y) \ x) \ y$

- ▶ Eine **konsistente Umbenennung** ist eine **Änderung** der **Bezeichner**, ohne die **Bindungen** zu ändern.
- ▶ **Durch konsistente Umbenennung läßt sich jede Phrase in eine semantisch äquivalente bereinigte Phrase überführen.**

$\text{val } \bar{x}_1 = \text{fn } \bar{y}_1 \Rightarrow (\text{fn } \bar{y}_2 \Rightarrow (\text{fn } \bar{x}_2 \Rightarrow z \ x_2 \ y_2) \ x) \ y_1$

Frage

Welche der folgenden Prozedurdeklarationen ist eine Bereinigung von

`fun f x = (fn x => (fn x => x)) ?`

- ▶ `fun f x_1 = (fn x_2 => (fn x_3 => x))`
- ▶ `fun f x_1 = (fn x_2 => (fn x_3 => x_1))`
- ▶ `fun f x_1 = (fn x_2 => (fn x_3 => x_2))`
- ▶ `fun f x_1 = (fn x_2 => (fn x_3 => x_3))`



Beispiel

```
let
  val x = 2*x
  val x = 2*x
  fun f x = if x<2 then x
             else f(x-1)
  val f = fn x => f x
in
  f x - y
end
```

```
let
  val  $\bar{x}_1$  = 2*x
  val  $\bar{x}_2$  = 2*x1
  fun  $\bar{f}_1$   $\bar{x}_3$  = if  $x_3 < 2$  then  $x_3$ 
                  else  $f_1(x_3-1)$ 
  val  $\bar{f}_2$  = fn  $\bar{x}_4$  =>  $f_1$   $x_4$ 
in
   $f_2$   $x_2$  - y
end
```

Statische und dynamische Bindungen

- ▶ **Statische Bindungen** binden Bezeichner an **Typen** (**monomorphe Bindung**) und **Typschemen** (**polymorphe Bindung**).

```
val x = 4*5
```

$x: int$

```
fun f y = if y then x else ~x
```

$f: bool \rightarrow int$

```
fun id z = z
```

$y: bool$

- ▶ **Dynamische Bindungen** binden Bezeichner an **Werte**.

$x := 20$

$f := (fun\ f\ y = if\ y\ then\ x\ else\ \sim x,\ bool \rightarrow int,\ [x := 20])$

$id := (fun\ id\ z = z,\ \forall \alpha. \alpha \rightarrow \alpha,\ [])$

Abgeleitete Formen (“syntaktischer Zucker”)

► Abkürzungen für **konditionale Ausdrücke**:

$e_1 \text{ andalso } e_2 \quad \rightsquigarrow \quad \text{if } e_1 \text{ then } e_2 \text{ else false}$

$e_1 \text{ orelse } e_2 \quad \rightsquigarrow \quad \text{if } e_1 \text{ then true else } e_2$





Beispiel: Prüfe ob eine Prozedur p für alle Zahlen zwischen m und n den Wert *true* liefert

```
fun forall m n p = m > n orelse (p m andalso forall (m+1) n p)
val forall : int → int → (int → bool) → bool
```

Frage

Für welche der folgenden Prozeduren ist

$f\ e1\ e2$ äquivalent zu $e1\ \text{andalso}\ e2$?

- ▶ `fun f x y = if x then y else false` 
- ▶ `fun f x y = if x then y else true` 
- ▶ `fun f x y = if x then true else y` 
- ▶ keine der genannten Prozeduren 

Beispiel: Primzahlberechnung

- ▶ Ist eine gegebene Zahl eine Primzahl?

```
fun prime x = x >= 2 andalso  
              forall 2 (sqrt x) (fn k => x mod k <> 0)  
val prime : int → bool
```

- ▶ Finde die erste Primzahl $> x$

```
fun nextprime x = first (x+1) prime  
val nextprime : int → int
```

- ▶ Finde die n -te Primzahl

```
fun nthprime n = iter n 1 nextprime  
val nthprime : int → int
```


Abgeleitete Formen (“syntaktischer Zucker”)

► Op-Ausdrücke:

$op + \rightsquigarrow \text{fn } (x, y) \Rightarrow x + y$

$op < \rightsquigarrow \text{fn } (x, y) \Rightarrow x < y$

$op = \rightsquigarrow \text{fn } (x, y) \Rightarrow x = y$

$op \text{ div } \rightsquigarrow \text{fn } (x, y) \Rightarrow x \text{ div } y$

$op \sim \rightsquigarrow \text{fn } x \Rightarrow \sim x$

► Wildcard-Muster:

`fun snd (_, y) = y`

`val snd : $\alpha * \beta \rightarrow \beta$`

Komposition von Prozeduren

op o

$fn : (\alpha \rightarrow \beta) * (\gamma \rightarrow \alpha) \rightarrow \gamma \rightarrow \beta$

Beispiel:

```
fun plus x y = x+y
```

```
val plus : int → int → int
```

```
fun times x y = x*y
```

```
val times : int → int → int
```

```
val foo = (plus 2) o (times 3)
```

```
val foo : int → int
```

```
foo 7
```

```
23 : int
```

Kapitel 4

Listen und Strings

Tupel

$(7, 2, \text{true}, 2)$

$(7, 2, \text{true}, 2) : \text{int} * \text{int} * \text{bool} * \text{int}$

- ▶ **Tupel:** Folge von Werten (v_1, \dots, v_n)
- ▶ **Positionen:** Zahlen $1, \dots, n$
- ▶ **Komponenten:** Werte v_1, \dots, v_n
- ▶ **Länge:** Anzahl der Positionen
- ▶ Der **Typ** eines Tupelausdrucks ergibt sich aus den Typen der Ausdrücke, die die Komponenten beschreiben.

Tupel vs. Listen

Tupel:

- ▶ **Länge fest**
- ▶ kein Zugriff auf variable Positionen (# <Konstante>)
- ▶ Komponenten mit **unterschiedlichen Typen**

Listen:

- ▶ **Länge variabel**
- ▶ Zugriff auf variable Positionen
- ▶ **Typen** der Elemente **identisch**

Listen

- ▶ **Schreibweise:** $[x_1, \dots, x_n]$
- ▶ Gegeben eine Liste $[x_1, \dots, x_n]$, bezeichnet man die Werte x_1, \dots, x_n als die **Elemente der Liste**.
- ▶ Eine Liste heißt **nichtleer**, wenn sie wenigstens ein Element hat.
- ▶ Die Liste $[]$ wird als **leere Liste** bezeichnet.
- ▶ Bei einer nichtleeren Liste bezeichnet man das erste Element als den **Kopf** und die restliche Liste als den **Rumpf** der Liste.

Beispiel: $[1, 2, 3]$ hat **Kopf** 1 und **Rumpf** $[2, 3]$.

Listen

- ▶ Eine Liste $[x_1, \dots, x_n]$ hat die **Länge** n .
- ▶ Die leere Liste $[]$ hat die Länge 0.
- ▶ Die **Konkatenation zweier Listen**:

$$[x_1, \dots, x_m] @ [y_1, \dots, y_n] := [x_1, \dots, x_m, y_1, \dots, y_n]$$

Beispiel:

$$[1, 2, 3] @ [3, 5, 6] @ [2, 6, 9]$$

$$[1, 2, 3, 3, 5, 6, 2, 6, 9] : \text{int list}$$

Konstruktion von Listen

$$nil : \forall \alpha. \alpha \text{ list}$$
$$:: : \forall \alpha. \alpha * \alpha \text{ list} \rightarrow \alpha \text{ list}$$

- ▶ **Nil** (*nil*) ist eine **Konstante** für die leere Liste.
- ▶ **Cons** (*::*) ist der **Operator** für die Konstruktion nichtleerer Listen.

Beispiele:

$$1 :: 2 :: 3 :: nil = [1, 2, 3]$$
$$true : bool$$
$$1 :: 2 :: 3 :: nil = 1 :: [2, 3]$$
$$true : bool$$

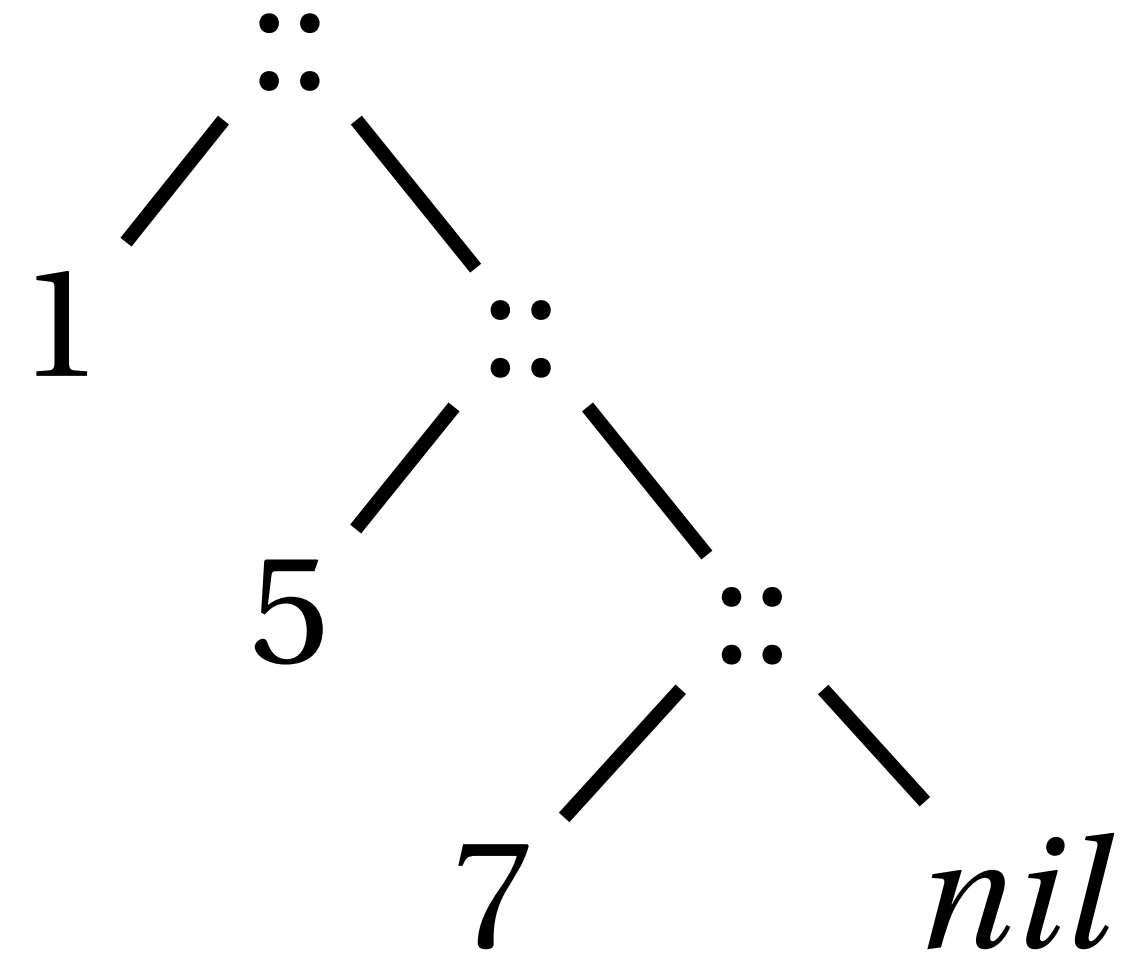
Rekursive Konstruktionsvorschrift

- ▶ Sei ein **Typ** t gegeben.
- ▶ Die **Listen über t** werden gemäß der folgenden **Konstruktionsvorschrift gebildet**:
 - ▶ Die **leere Liste** nil ist eine **Liste über t** .
 - ▶ Wenn x ein **Wert** des Typs t ist und xs eine **Liste über t** , dann ist $x::xs$ eine **Liste über t** .

(Vorsicht: Das Textbuch spricht hier von Tupeln.
Das ist als Gedankenexperiment nützlich, aber:
Listen \neq Tupel)

Baumdarstellung

[1, 5, 7]



Klammerregeln:

- ▶ :: und @ klammern rechts.
- ▶ :: und @ klammern gleichberechtigt.

Null, head und tail

- ▶ **Null** testet ob eine Liste leer ist.

$$null : \alpha \text{ list} \rightarrow bool$$

- ▶ **Hd** liefert den **Kopf** (engl. **head**) einer nichtleeren Liste:

$$hd : \alpha \text{ list} \rightarrow \alpha$$

- ▶ **Tl** liefert den **Rumpf** (engl. **tail**) einer nichtleeren Liste:

$$tl : \alpha \text{ list} \rightarrow \alpha \text{ list}$$

Wenn *hd* oder *tl* auf die **leere Liste** angewendet werden, werfen sie die **Ausnahme Empty**.

Frage

Welche der folgenden Ausdrücke sind äquivalent?

- ▶ $(e1 :: e2) @ e3$ und $e1 :: (e2 @ e3)$
- ▶ $(e1 @ e2) @ e3$ und $e1 @ (e2 @ e3)$
- ▶ $(e1 :: e2) :: e3$ und $e1 :: (e2 :: e3)$



www.prog1.saarland