Programmierung 1

Vorlesung 7

Livestream beginnt um 14:15 Uhr

Listen, Teil 2

Programmierung 1

Listen

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

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

Beispiel:

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

[1, 2, 3, 3, 5, 6, 2, 6, 9] : int list

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 ≠ Tupel)

Null, head und tail

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

$$null : \alpha\ list \rightarrow bool$$

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

$$hd\ :\ \alpha\ list \rightarrow \alpha$$

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

$$tl\ :\ \alpha\ list \rightarrow \alpha\ list$$

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

Konkatenation

- **Konkatenation zweier Listen** mit Hilfe von **null, hd, tl**:

```
fun append (xs,ys) = if null xs then ys
                     else hd xs :: append(tl xs, ys)
```

*val append : α list * α list → α list*

- **alternative Deklaration** mit **nil**:

```
fun append (xs,ys) = if xs=nil then ys
                     else hd xs :: append(tl xs, ys)
```

*val append : ''a list * ''a list → ''a list*

Typ mit Gleichheit!

Positionen und nth

- **Nth (List.nth)** liefert das *n*-te Element einer Liste:

```
fun nth(xs,n) = if n<0 orelse null xs then raise Subscript
                else if n=0 then hd xs else nth(tl xs, n-1)
```

  *val nth : α list * int $\to$ α*

```
nth ([3,4,5], 0)
```

  3 : int

```
nth ([3,4,5], 2)
```

  5 : int

```
nth ([3,4,5], 3)
```

  ! Uncaught exception: Subscript

- **Nummerierung beginnt bei 0!**
- Eine **Grenzüberschreitung** wird mit der **Ausnahme Subscript** signalisiert.

Regelbasierte Prozeduren

- **Konkatenation zweier Listen** mit Hilfe von **null, hd, tl**:

```
fun append (xs,ys) = if null xs then ys
                     else hd xs :: append(tl xs, ys)
```

$val\ append : \alpha\ list * \alpha\ list \rightarrow \alpha\ list$

- **Regelbasierte Prozedur:**

```
fun append (nil,   ys) = ys
  | append (x::xr, ys) = x::append(xr,ys)
```

$val\ append : \alpha\ list * \alpha\ list \rightarrow \alpha\ list$

Regelbasierte Prozeduren

```
fun length nil      = 0
  | length (x::xr) = 1 + length xr
```

$length : \alpha\ list \rightarrow int$

- Dies ist eine **regelbasierte Prozedur mit zwei Regeln.**
- Normale Prozeduren kann man als regelbasierte Prozeduren mit nur einer Regel auffassen.

Prozeduren auf Listen

- **Konkatenation zweier Listen**:

```
fun append (nil,    ys) = ys
  | append (x::xr, ys) = x::append(xr,ys)
```

$val\ append : \alpha\ list * \alpha\ list \to \alpha\ list$

- **Konkatenation der Elementlisten** (List.concat):

```
fun concat nil      = nil
  | concat (x::xr) = x @ concat xr
```

$val\ concat : \alpha\ list\ list \to \alpha\ list$

Prozeduren auf Listen

- **Reversierung** (vordeklariert):

```
fun rev nil      = nil
  | rev (x::xr) = rev xr @ [x]
```

$val\ rev : \alpha\ list \to \alpha\ list$

- **Tabulierung** (List.tabulate): $tabulate(n, f) = [f(0), \ldots, f(n-1)]$

```
fun tabulate (n,f) =
       iterdn (n-1) 0 nil (fn (i,xs) => f i::xs)
```

$val\ tabulate : int * (int \to \alpha) \to \alpha\ list$

Prozeduren auf Listen

- **Map** (vordeklariert)**:** $map\ f\ [x_1, \ldots, x_n] = [f\ x_1, \ldots, f\ x_n]$

```
fun map f nil      = nil
  | map f (x::xr) = (f x) :: (map f xr)
```

$map : (\alpha \to \beta) \to \alpha\ list \to \beta\ list$

Prozeduren auf Listen

- **Filter** (List.filter)**:**

```
fun filter f nil      = nil
  | filter f (x::xr) = if f x then x :: filter f xr
                       else filter f xr
```

val filter : (α → bool) → α list → α list

```
filter (fn x => x<0) [0, ~1, 2, ~3, ~4]
```

[~1, ~3, ~4] : int list

```
filter (fn x => x>=0) [0, ~1, 2, ~3, ~4]
```

[0, 2] : int list

Prozeduren auf Listen

- **Exists** (List.exists)**:**

```
fun exists f nil      = false
  | exists f (x::xr) = f x orelse exists f xr
```

$exists : (\alpha \to bool) \to \alpha\ list \to bool$

- **All** (List.all)**:**

```
fun all f nil      = true
  | all f (x::xr) = f x andalso all f xr
```

$all : (\alpha \to bool) \to \alpha\ list \to bool$

Regeln

- Eine **Regel** besteht aus einem **Bezeichner,** einem **Muster,** und einem **Rumpf:**

$$\langle Bezeichner \rangle \ \langle Muster \rangle \ = \ \underbrace{\langle Ausdruck \rangle}_{\text{Rumpf}}$$

- Das **Muster** entscheidet über die **Anwendbarkeit der Regel.**
- Das Muster kann **Bezeichner** einführen, die als **Variablen des Musters** bezeichnet werden.
- **Musterabgleich (pattern matching):** Prozess, der entscheidet ob ein Muster einen Wert trifft.

  Beispiel: [7,8,9] **trifft** (*x*::*xs*) und **bindet** dabei *x* an 7, *xs* an [8,9]

Frage

Entscheiden Sie für jeden der folgenden Werte, ob er das Muster

`(x, y :: _ :: z, (u,3))`

trifft.

- `(7,[1],(3,3))` ✗
- `([7],[1,2,3],[3,3])` ✗
- `([1,2],[4,5],(11,3))` ✓
- `([1,2],[4,5,6],(11,3))` ✓

Disjunkte Muster

- Eine Menge von **Mustern** heißt **disjunkt** wenn Ihre Muster jeweils verschiedene Werte treffen.

  Beispiel:

  $$[], \quad [(x, y)], \quad [(x, 5),\ (7, y)], \quad (\_ :: (x, \_) :: \_ :: ps)$$

- **Muster,** die **nicht disjunkt** sind, heißen **überlappend.**

  Beispiel:

  $$[(x, 5),\ (7, y)], \quad (\_ :: (x, \_) :: ps)$$

Disjunkte Regeln

- Eine **Regel** heißt **disjunkt/überlappend,**
  wenn ihre **Muster disjunkt/überlappend** sind.

- Bei **disjunkten Regeln** spielt die **Reihenfolge** der Muster
  keine Rolle.

```
fun length nil      = 0
  | length (_::xr) = 1 + length xr
```

```
fun length (_::xr) = 1 + length xr
  | length nil      = 0
```

- **Bei überlappenden Regeln kommt es auf die Reihenfolge der Muster an.**

```
fun or (false, false) = false
  | or      _         = true
```

Überlappende Regeln für Zahlen

```
fun power (x, 0) = 1
  | power (x, n) = x * power(x, n-1)
```

*val power : int * int → int*

- **Aber:** Rekursive Prozeduren mit **Anwendungsbedingungen** benötigen ein **Konditional.**

```
fun potenz (x,n) = if n<1 then 1 else x*potenz(x,n-1)
```

Erschöpfende Regeln

- Eine Menge von **Mustern** **erschöpft einen Typ** wenn **jeder Wert des Typs mindestens ein Muster trifft.**
- Die **Regeln** einer Prozedur heißen **erschöpfend**, wenn ihre Regeln den **Argumenttyp** erschöpfen.
- **Sie sollten nur Prozeduren mit erschöpfenden Regeln verwenden.**

```
fun test nil = 0
  | test [_] = 1
```

val test : α *list* $\to$ *int*

! Warning: pattern matching is not exhaustive

Ausnahmen

```
exception SomethingWrong

fun test nil = 0
  | test [_] = 1
  | test _   = raise SomethingWrong
```

val test : α list $\rightarrow$ int

```
test [1,2]
```

! Uncaught exception: SomethingWrong

Mehr zu Ausnahmen in **Kapitel 6.**

Frage

Betrachten Sie die Regeln der Prozedur

```
fun pairs (x::y::xs) = (x,y)::pairs xs
  | pairs nil = nil
```

Die Regeln sind

- überlappend ✗
- disjunkt ✓
- erschöpfend ✗
- nicht erschöpfend ✓

Regelbasierte Abstraktionen

- **Beispiel:**

```
fn true  => false
 | false => true
```

$fn : bool \rightarrow bool$

- Der **Case-Ausdruck**

$$case\;\; e\;\; of\;\; M_1 \Rightarrow e_1 \mid \ldots \mid M_n \Rightarrow e_n$$

ist eine abgeleitete Form für

$$(fn\;\; M_1 \Rightarrow e_1 \mid \ldots \mid M_n \Rightarrow e_n)\;\; e$$

Kaskadierte Prozeduren mit mehreren Regeln

Kaskadierte Prozedurdeklarationen mit mehreren Regeln werden auf **Prozedurdeklarationen mit nur einer Regel** zurückgeführt:

```
fun or false false = false
  | or   _     _    = true
```

val or : bool → bool → bool

```
fun or x y = case (x,y)
             of (false, false) => false
             |  (  _  ,   _  ) => true
```

```
fun or (x:bool) = fn (y:bool) =>
                     (fn (false , false ) => false
                      |  (a:bool, b:bool) => true ) (x,y)
```

Faltung

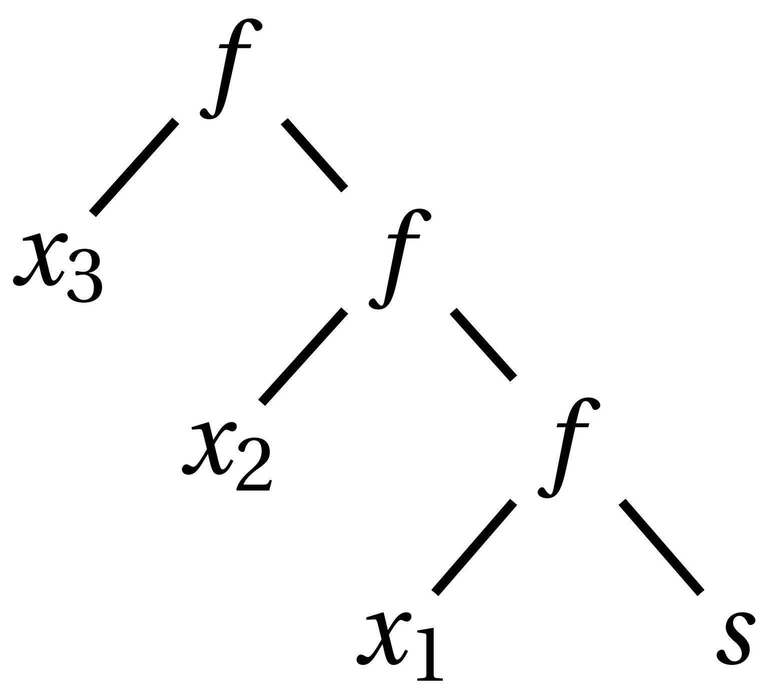
$$foldl : (\alpha * \beta \to \beta) \to \beta \to \alpha\ list \to \beta$$

Verknüpfung Startwert Liste

$$\alpha * \beta \to \beta$$

Argument Akku

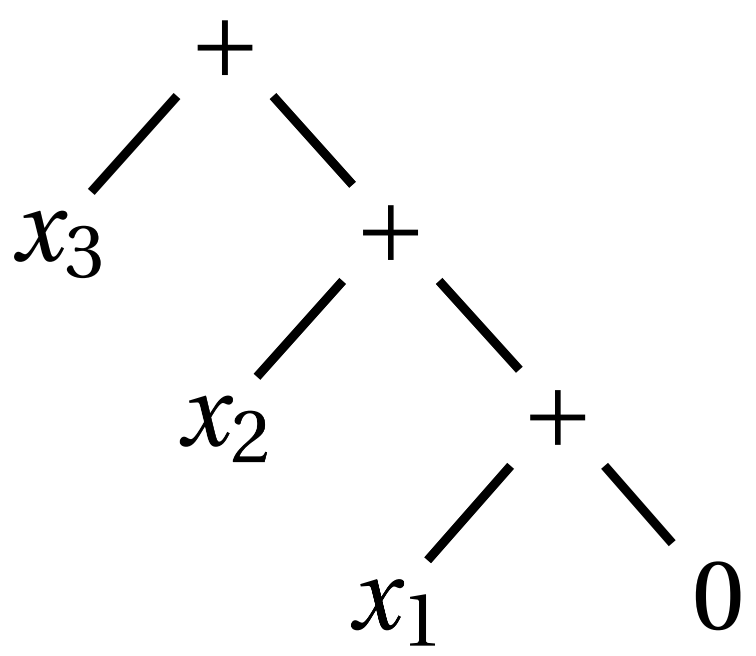
$$foldl\ f\ s\ [x_1, x_2, x_3] \;=\; f(x_3, f(x_2, f(x_1, s))) \;=$$



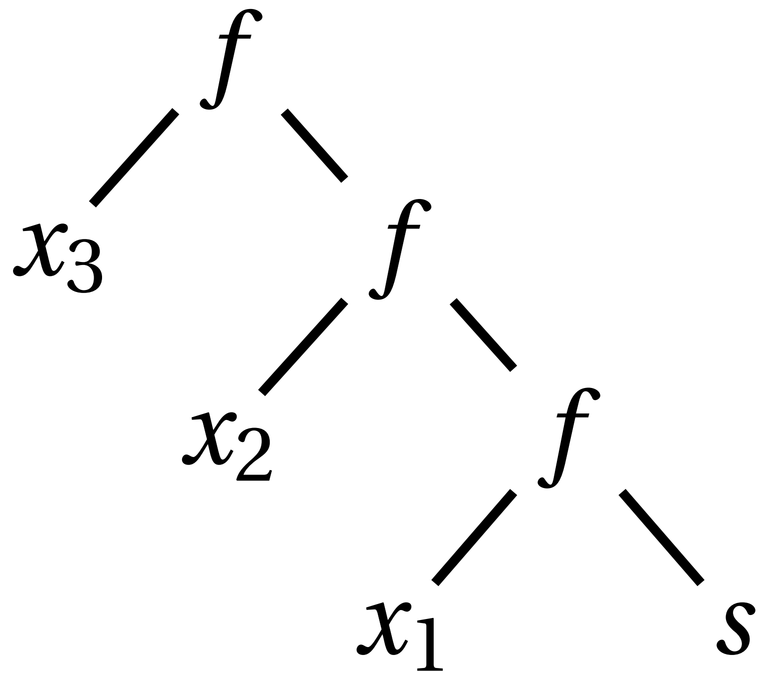
Beispiele

- **Summe der Elemente einer ganzzahligen Liste:**

$$\textit{foldl op+ 0}\ [x_1, x_2, x_3] \;=\; x_3 + (x_2 + (x_1 + 0)) \;=$$



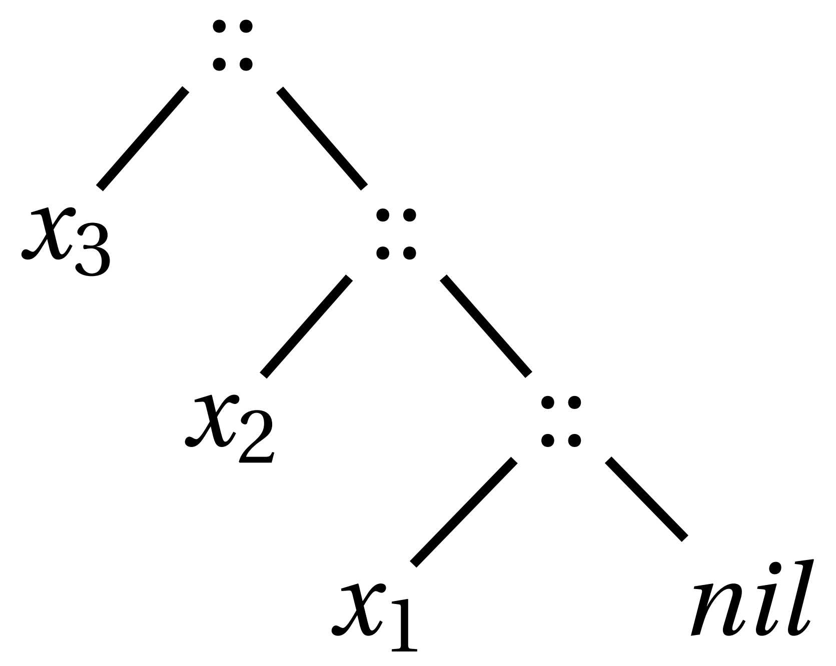
$$\textit{foldl}\ f\ s\ [x_1, x_2, x_3] \;=\; f(x_3, f(x_2, f(x_1, s))) \;=$$



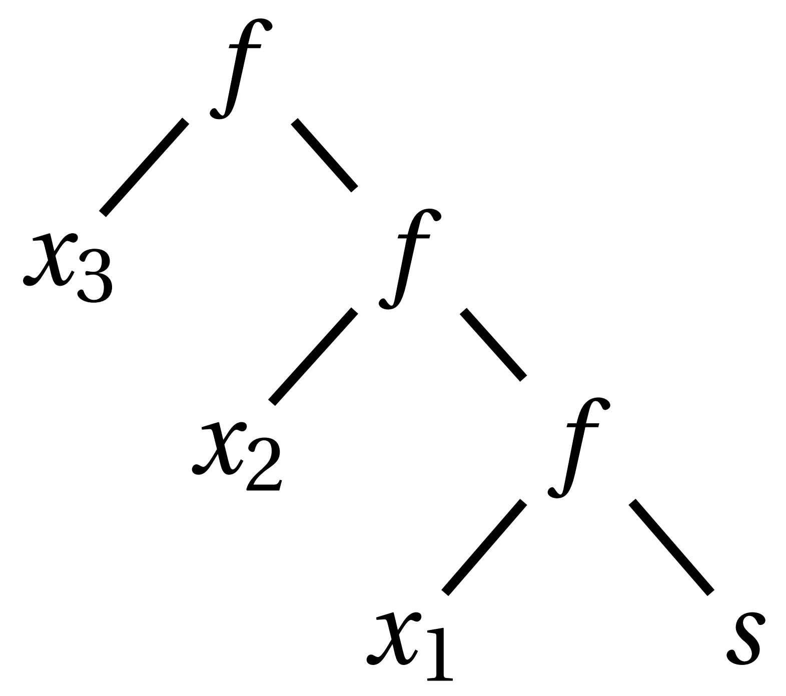
Beispiele

- **Reversierung einer Liste:**

$$\mathit{foldl}\ \mathit{op}{::}\ \ \mathit{nil}\ [x_1, x_2, x_3]\ =\ x_3 :: (x_2 :: (x_1 :: \mathit{nil}))\ =$$



$$\mathit{foldl}\ f\ s\ [x_1, x_2, x_3]\ =\ f(x_3, f(x_2, f(x_1, s)))\ =$$

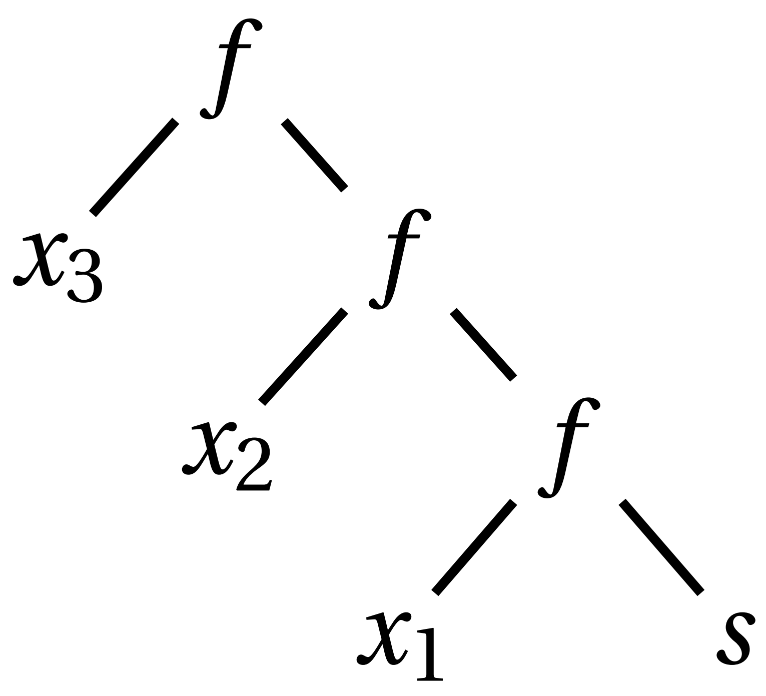


Foldl

```
fun foldl f s nil      = s
  | foldl f s (x::xr) = foldl f (f(x,s)) xr
```

*val foldl : (α * β → β) → β → α list → β*

$$foldl\ f\ s\ [x_1, x_2, x_3] \ = \ f(x_3, f(x_2, f(x_1, s))) \ =$$



Summe der Elemente einer ganzzahligen Liste

```
fun foldl f s nil      = s
  | foldl f s (x::xr) = foldl f (f(x,s)) xr
```

val foldl : $(\alpha * \beta \rightarrow \beta) \rightarrow \beta \rightarrow \alpha\ list \rightarrow \beta$

```
foldl op+ 0 [1,4,6]
          = foldl op+ 0 (1::[4,6])
          = foldl op+ (op+(1,0)) [4,6]
          = foldl op+ 1 (4::[6])
          = foldl op+ (op+(4,1)) [6]
          = foldl op+ 5 (6::nil)
          = foldl op+ (op+(6,5)) nil
          = foldl op+ 11 nil
          = 11
```

Reversierung einer Liste

```
fun foldl f s nil      = s
  | foldl f s (x::xr) = foldl f (f(x,s)) xr
```

val foldl : $(\alpha * \beta \rightarrow \beta) \rightarrow \beta \rightarrow \alpha\ list \rightarrow \beta$

```
foldl op:: nil [1,4,6]
         = foldl op:: nil (1::[4,6])
         = foldl op:: (op::(1,nil)) [4,6]
         = foldl op:: [1] (4::[6])
         = foldl op:: (op::(4,[1])) [6]
         = foldl op:: [4,1] (6::nil)
         = foldl op:: (op::(6,[4,1])) nil
         = foldl op:: [6,4,1] nil
         = [6,4,1]
```

Beispiele

- **Reversierung:**

```
fun rev nil      = nil
  | rev (x::xr) = rev xr @ [x]
```

$val\ rev : \alpha\ list \rightarrow \alpha\ list$

```
fun rev xs = foldl op:: nil xs
```

- **Länge:**

```
fun length nil      = 0
  | length (x::xr) = 1 + length xr
```

$length : \alpha\ list \rightarrow int$

```
fun length xs = foldl (fn (x,n) => n+1) 0 xs
```

Beispiele

- **Exists:**

```
fun exists f nil      = false
  | exists f (x::xr) = f x orelse exists f xr
```

exists : ($\alpha \to$ *bool*) $\to \alpha$ *list* $\to$ *bool*

```
fun exists p = foldl (fn (x,b) => b orelse p x) false
```

- **All:**

```
fun all f nil      = true
  | all f (x::xr) = f x andalso all f xr
```

all : ($\alpha \to$ *bool*) $\to \alpha$ *list* $\to$ *bool*

```
fun all p = foldl (fn (x,b) => b andalso p x) true
```

Konkatenation?

- **Append:**

```
fun append (xs,ys) = if null xs then ys
                     else hd xs :: append(tl xs, ys)
```

val append : α *list ** α *list* $\rightarrow$ α *list*

```
foldl op:: [3,4] [1,2]
        = foldl op:: [3,4] (1::[2])
        = foldl op:: (op::(1,[3,4])) [2]
        = foldl op:: [1,3,4] (2::nil)
        = foldl op:: [2,1,3,4] nil
        = [2,1,3,4]
```

www.prog1.saarland