Programming Paradigms

Summer Term 2017

5th Lecture

Prof. Janis Voigtländer
University of Duisburg-Essen

Haskell evaluation semantics: on-demand (lazy evaluation)

```
fac :: Int → Int
fac n  =  if n == 0 then 1 else n * fac (n – 1)
```

```
> fac 5
120
```

```
sumsquare :: Int → Int
sumsquare i  =  if i == 0 then 0 else i * i + sumsquare (i – 1)
```

```
> sumsquare 4
30
```

Computation by step-wise evaluation:

```
> sumsquare 2
= if 2 == 0 then 0 else 2 * 2 + sumsquare (2 – 1)
= 2 * 2 + sumsquare (2 – 1)
= 4 + sumsquare (2 – 1)
= 4 + if (2 – 1) == 0 then 0 else …
= 4 + (1 * 1 + sumsquare (1 – 1))
= 4 + (1 + sumsquare (1 – 1))
= 4 + (1 + if (1 – 1) == 0 then 0 else …)
= 4 + (1 + 0)
= 5
```

Haskell evaluation semantics: on-demand (lazy evaluation)

```
a = 3
d = (a, e)
e = [fst d, f]
f = head e
```

→

```
> d
= (a, e)
```

Haskell evaluation semantics: on-demand (lazy evaluation)

```
a = 3
d = (a, e)
e = [fst d, f]
f = head e
```

→

```
> d
= (a, e)
= (3, e)
```

Haskell evaluation semantics: on-demand (lazy evaluation)

```
a = 3
d = (a, e)
e = [fst d, f]
f = head e
```

→

```
> d
= (a, e)
= (3, e)
= (3, [fst d, f])
```

Haskell evaluation semantics: on-demand (lazy evaluation)

```
a = 3
d = (a, e)
e = [fst d, f]
f = head e
```

→

```
> d
= (a, e)
= (3, e)
= (3, [fst d, f])
= (3, [fst (3, [fst d, f]), f])
```

Haskell evaluation semantics: on-demand (lazy evaluation)

```
a = 3
d = (a, e)
e = [fst d, f]
f = head e
```

→

```
> d
= (a, e)
= (3, e)
= (3, [fst d, f])
= (3, [fst (3, [fst d, f]), f])
= (3, [3, f])
```

Haskell evaluation semantics: on-demand (lazy evaluation)

```
a = 3
d = (a, e)
e = [fst d, f]
f = head e
```

→

```
> d
= (a, e)
= (3, e)
= (3, [fst d, f])
= (3, [fst (3, [fst d, f]), f])
= (3, [3, f])
= (3, [3, head e])
```

Haskell evaluation semantics: on-demand (lazy evaluation)

```
a = 3
d = (a, e)
e = [fst d, f]
f = head e
```

→

```
> d
= (a, e)
= (3, e)
= (3, [fst d, f])
= (3, [fst (3, [fst d, f]), f])
= (3, [3, f])
= (3, [3, head e])
= (3, [3, head [3, head e]])
```

Haskell evaluation semantics: on-demand (lazy evaluation)

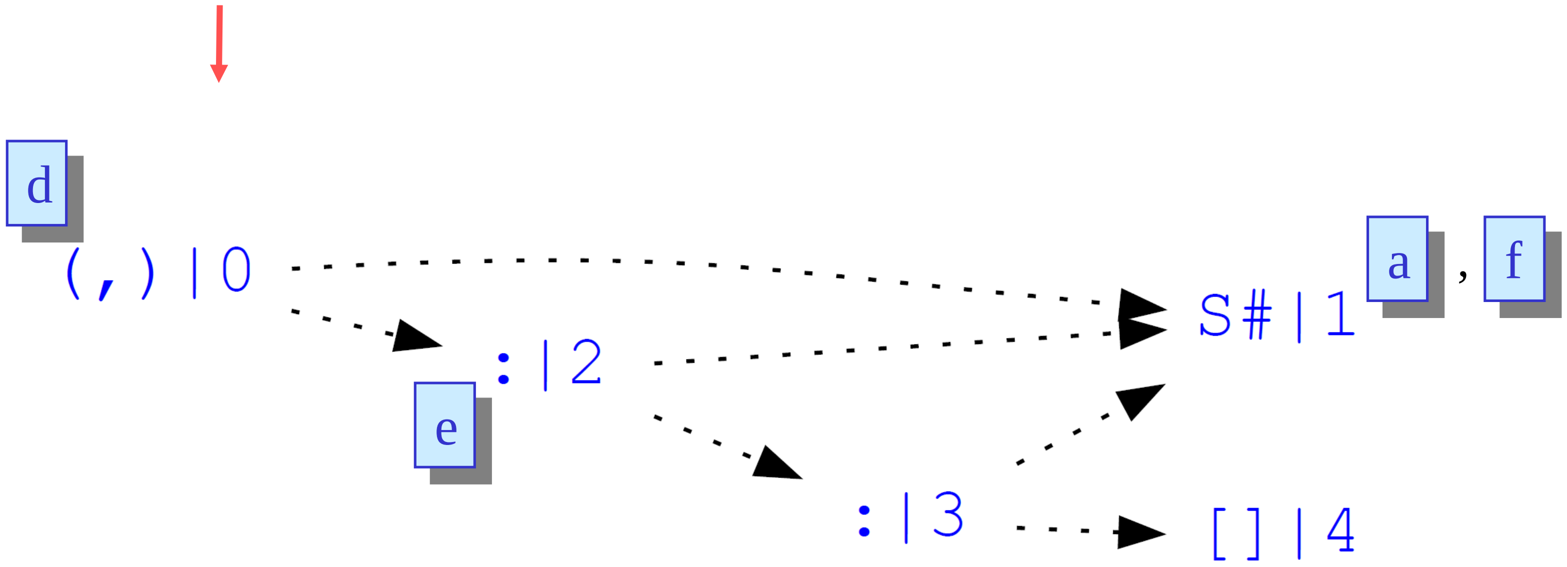
```
a = 3
d = (a, e)
e = [fst d, f]
f = head e
```

→

```
> d
= (a, e)
= (3, e)
= (3, [fst d, f])
= (3, [fst (3, [fst d, f]), f])
= (3, [3, f])
= (3, [3, head e])
= (3, [3, head [3, head e]])
= (3, [3, 3])
```

Haskell evaluation semantics: on-demand (lazy evaluation)

```
a = 3
d = (a, e)
e = [fst d, f]
f = head e
```



Pattern matching “strategies”

- Examples on Boolean values:

```
not False = True
not True  = False
```

```
True  && True  = True
True  && False = False
False && True  = False
False && False = False
```

- Somewhat more compact:

```
not False = True
not _     = False
```

```
True && True = True
_    && _    = False
```

anonymous variables

- But more efficient? Yes, for some inputs quite drastically!

```
False && (ack 4 2 > 0)
```

Pattern matching "strategies"

- Examples on Boolean values:

```
not False = True
not True  = False
```

```
True  && True  = True
True  && False = False
False && True  = False
False && False = False
```

- Somewhat more compact:

```
not False = True
not _     = False
```

```
True && True = True
_    && _    = False
```

- But more efficient? Yes, for some inputs!

another variant: →

```
b && True = b
_ && _    = False
```

Matching from left to right!

- Not possible:

```
b && b = b
_ && _ = False
```

Alternative syntax (and consideration of scoping!)

- Explicit case-expressions, for example:

```
ifThenElse i t e  =  case i of
                        True  → t
                        False → e
```

- Or, for example:

```
f x y  =  case (x + y, x – y) of
            (z, _)  | z > 0 → y
            (0, x)          → x + y
```

- What do you think is the result of the following call of this function?

```
> f  10 (–10)
```

Programming Paradigms

Elementary dealing with lists in Haskell

To make pattern matching more interesting: working on lists

- Haskell lists: sequences of elements of same type (homogeneous data structure)
- Syntax: list elements are enclosed in square brackets.

[1, 2, 3]	list of integers (type: Int)
['a', 'b', 'c']	list of characters (type: Char)
[ ]	empty list (of any type)
[[1,2], [ ], [2]]	list of integer lists

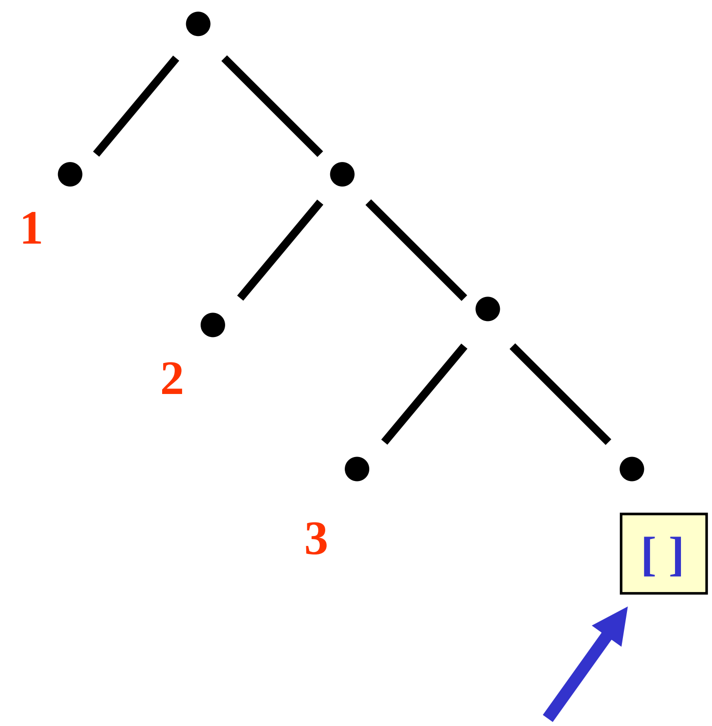
[[1,2], 'a', 3]	not a valid list (different element types)

- Contrary to what many examples in the lecture might suggest, lists are in practice often not the data structure one should use! (Instead, user defined data types, or types from libraries like Data.ByteString, Data.Array, Data.Map, …)

Tree view on lists

Internally, lists are represented as certain binary trees, whose leaves are annotated with the individual list elements:

[1, 2, 3]



(pronounced: “nil”)

Every (finite) list is terminated by an occurrence of empty list.

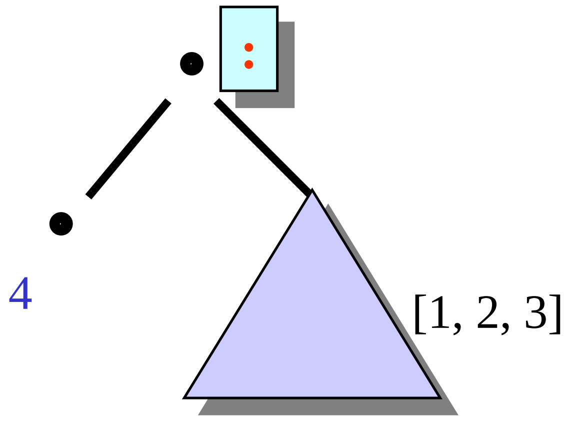
The list constructor

- Elementary constructor ("operator" for constructing) of lists:

  : (pronounced: "cons")

- The constructor ":" serves to extend a given list by an element, which is inserted at the head of the list:

```
>   4 : [1, 2, 3]
[4, 1, 2, 3]
```



- Alternative notation for lists (analogous to tree view):

```
4 : 1 : 2 : 3 : [ ]
```

Length of a list

- Function to determine the length of a list (actually predefined):

```
length [ ]          =   0
length (x : xs)   =   length xs  + 1
```

head element (x) — rest list (xs)

- Example for applying the length function:

```
> length [1, 2]
= length [2]  + 1
= (length [ ]  + 1)  + 1
= (      0          + 1)  + 1
= 1 + 1
= 2
```

Pattern matching with list constructors

- Pattern matching between lists and constructor expressions can only be understood by viewing <u>both</u> expressions in constructor form:

length (x : xs) = . . .

length [1, 2] → length (1 : 2 : [ ])

x ↦ 1
xs ↦ 2 : [ ]

- This perspective is also helpful when "recursively deconstructing" singleton lists, as follows:

length (x : xs) = . . .

length [2] → length (2 : [ ])

x ↦ 2
xs ↦ [ ]

All parentheses on this slide are mandatory!

Concatenation of lists

- Important operation for all list types: concatenating two lists

```
concatenation  [ ]       ys   =   ys
concatenation  (x : xs)  ys   =   x : concatenation xs ys
```

- Example application:

```
>  concatenation [1, 2]  [3, 4]
[1, 2, 3, 4]
```

- Predefined as infix operator:

```
> [1, 2] ++ [3, 4]
[1, 2, 3, 4]
```

Access to individual list elements and sublists

- Targeted access to individual elements of a list via another predefined infix operator:

  !!

- Counting of list elements starts with 0 !

```
>  [1, 2, 3] !! 1
2
```

- Access per (x : xs)-pattern of course only for non-empty lists:

```
tail (x : xs)  =  xs
```

```
>  tail [ ]
ERROR – Pattern match failure: tail []
```

```
head (x : xs)  =  x
```

```
>  head [ ]
ERROR – Pattern match failure: head []
```

(Unfortunately the source of such errors is not always so easily identified.)

More complex pattern matching (and its interaction with evaluation)

```
f :: [Int] → [[Int]]
f [ ]                 =  [ ]
f [x]                 =  [[x]]
f (x : y : zs)        =  if x <= y then (x : s) : ts else [x] : s : ts
    where s : ts = f (y : zs)
```

local definition + match (arrow pointing to the "=" in "where s : ts = f (y : zs)")

More complex pattern matching (and its interaction with evaluation)

```
f :: [Int] → [[Int]]
f [ ]                 =  [ ]
f [x]                 =  [[x]]
f (x : y : zs)        =  if x <= y then (x : s) : ts else [x] : s : ts
      where s : ts = f (y : zs)
```

Computation by step-wise evaluation:

```
> f [1, 2, 0]
= if 1 <= 2 then (1 : s) : ts else [1] : s : ts       where s : ts = f (2 : [0])
= (1 : s) : ts                                        where s : ts = f (2 : [0])
```

More complex pattern matching (and its interaction with evaluation)

```
f :: [Int] → [[Int]]
f [ ]                    =  [ ]
f [x]                    =  [[x]]
f (x : y : zs)           =  if x <= y then (x : s) : ts else [x] : s : ts
      where s : ts = f (y : zs)
```

Computation by step-wise evaluation:

```
> f [1, 2, 0]
= if 1 <= 2 then (1 : s) : ts else [1] : s : ts      where s : ts = f (2 : [0])
= (1 : s) : ts                                       where s : ts = f (2 : [0])
= (1 : s) : ts                                       where s : ts = [2] : s' : ts'
                                                           where s' : ts' = f (0 : [ ])
```

More complex pattern matching (and its interaction with evaluation)

```
f :: [Int] → [[Int]]
f [ ]                 =  [ ]
f [x]                 =  [[x]]
f (x : y : zs)        =  if x <= y then (x : s) : ts else [x] : s : ts
     where s : ts = f (y : zs)
```

Computation by step-wise evaluation:

```
> f [1, 2, 0]
= if 1 <= 2 then (1 : s) : ts else [1] : s : ts       where s : ts = f (2 : [0])
= (1 : s) : ts                                        where s : ts = f (2 : [0])
= (1 : s) : ts                                        where s : ts = [2] : s' : ts'
                                                           where s' : ts' = f (0 : [ ])
= (1 : [2]) : s' : ts'                                where s' : ts' = f (0 : [ ])
```

More complex pattern matching (and its interaction with evaluation)

```
f :: [Int] → [[Int]]
f [ ]               =  [ ]
f [x]               =  [[x]]
f (x : y : zs)      =  if x <= y then (x : s) : ts else [x] : s : ts
     where s : ts = f (y : zs)
```

Computation by step-wise evaluation:

```
> f [1, 2, 0]
= if 1 <= 2 then (1 : s) : ts else [1] : s : ts      where s : ts = f (2 : [0])
= (1 : s) : ts                                       where s : ts = f (2 : [0])
= (1 : s) : ts                                       where s : ts = [2] : s' : ts'
                                                          where s' : ts' = f (0 : [ ])
= (1 : [2]) : s' : ts'                               where s' : ts' = f (0 : [ ])
= (1 : [2]) : s' : ts'                               where s' : ts' = [[0]]
```

More complex pattern matching (and its interaction with evaluation)

```
f :: [Int] → [[Int]]
f [ ]              =  [ ]
f [x]              =  [[x]]
f (x : y : zs)     =  if x <= y then (x : s) : ts else [x] : s : ts
     where s : ts = f (y : zs)
```

Computation by step-wise evaluation:

```
> f [1, 2, 0]
= if 1 <= 2 then (1 : s) : ts else [1] : s : ts      where s : ts = f (2 : [0])
= (1 : s) : ts                                        where s : ts = f (2 : [0])
= (1 : s) : ts                                        where s : ts = [2] : s' : ts'
                                                           where s' : ts' = f (0 : [ ])
= (1 : [2]) : s' : ts'                                where s' : ts' = f (0 : [ ])
= (1 : [2]) : s' : ts'                                where s' : ts' = [[0]]
= (1 : [2]) : [0] : [ ] = [[1, 2], [0]]
```

More complex pattern matching (and its interaction with evaluation)

```
unzip :: [(Int, Int)] → ([Int], [Int])
unzip [ ]              =  ([ ], [ ])
unzip ((x, y) : zs) =  let (xs, ys) = unzip zs in (x : xs, y : ys)
```

variant for local definition

Computation by step-wise evaluation:

```
> unzip [(1, 2), (3, 4)]
= let (xs, ys) = unzip [(3, 4)] in (1 : xs, 2 : ys)
= let (xs, ys) = (let (xs', ys') = unzip [ ] in (3 : xs', 4 : ys')) in (1 : xs, 2 : ys)
```

More complex pattern matching (and its interaction with evaluation)

```
unzip :: [(Int, Int)] → ([Int], [Int])
unzip [ ]              =  ([ ], [ ])
unzip ((x, y) : zs) =  let (xs, ys) = unzip zs in (x : xs, y : ys)
```

↑ variant for local definition

Computation by step-wise evaluation:

```
> unzip [(1, 2), (3, 4)]
= let (xs, ys) = unzip [(3, 4)] in (1 : xs, 2 : ys)
= let (xs, ys) = (let (xs', ys') = unzip [ ] in (3 : xs', 4 : ys')) in (1 : xs, 2 : ys)
= let (xs', ys') = unzip [ ] in (1 : 3 : xs', 2 : 4 : ys')
```

More complex pattern matching (and its interaction with evaluation)

```
unzip :: [(Int, Int)] → ([Int], [Int])
unzip [ ]            =  ([ ], [ ])
unzip ((x, y) : zs) =  let (xs, ys) = unzip zs in (x : xs, y : ys)
```

variant for local definition (↑ points to `let (xs, ys) = …`)

Computation by step-wise evaluation:

```
> unzip [(1, 2), (3, 4)]
= let (xs, ys) = unzip [(3, 4)] in (1 : xs, 2 : ys)
= let (xs, ys) = (let (xs', ys') = unzip [ ] in (3 : xs', 4 : ys')) in (1 : xs, 2 : ys)
= let (xs', ys') = unzip [ ] in (1 : 3 : xs', 2 : 4 : ys')
= let (xs', ys') = ([ ], [ ]) in (1 : 3 : xs', 2 : 4 : ys')
```

More complex pattern matching (and its interaction with evaluation)

```
unzip :: [(Int, Int)] → ([Int], [Int])
unzip [ ]              =  ([ ], [ ])
unzip ((x, y) : zs) =  let (xs, ys) = unzip zs in (x : xs, y : ys)
```

↑ variant for local definition

Computation by step-wise evaluation:

```
> unzip [(1, 2), (3, 4)]
= let (xs, ys) = unzip [(3, 4)] in (1 : xs, 2 : ys)
= let (xs, ys) = (let (xs', ys') = unzip [ ] in (3 : xs', 4 : ys')) in (1 : xs, 2 : ys)
= let (xs', ys') = unzip [ ] in (1 : 3 : xs', 2 : 4 : ys')
= let (xs', ys') = ([ ], [ ]) in (1 : 3 : xs', 2 : 4 : ys')
= ([1, 3], [2, 4])
```

Excurse: layout in Haskell

```
let  y = a * b
     f x = (x + y) / y
in  f c + f d
```

implicit layout
("offside rule")

```
let { y = a * b; f x = (x + y) / y }
in  f c + f d
```

equivalently, explicit layout

```
let y = a * b
            f x = (x + y) / y
in  f c + f d
```

not equivalent, incorrect

```
let  y = a * b
  f x = (x + y) / y
in  f c + f d
```

(analogously for other language constructs, e.g., where, case)

Pattern matching on several arguments (and "outdated" (n + k)-patterns)

```
drop :: Int → [Int] → [Int]
drop 0       xs        =  xs
drop n       [ ]       =  [ ]
drop (n + 1) (x : xs)  =  drop n xs
```

in Haskell 98 allowed, in Haskell 2010 not anymore!

```
> drop 0 [1, 2, 3]
[1, 2, 3]
```

```
> drop 5 [1, 2, 3]
[ ]
```

```
> drop 3 [1, 2, 3, 4, 5]
[4, 5]
```

Order in pattern matching

- Again as a warning, this:

```
zip :: [Int] → [Int] → [(Int, Int)]
zip (x : xs) (y : ys)   =  (x, y) : zip xs ys
zip xs        ys         =  [ ]
```

is okay:

```
> zip [1 .. 3] [10 .. 15]
[(1, 10), (2, 11), (3, 12)]
```

- But this:

```
zip :: [Int] → [Int] → [(Int, Int)]
zip xs        ys         =  [ ]
zip (x : xs) (y : ys)   =  (x, y) : zip xs ys
```

is problematic:

```
> zip [1 .. 3] [10 .. 15]
[ ]
```

Programming Paradigms

List Comprehensions

Arithmetic sequences

- A useful notation for lists of numbers:

arithmetic sequences

- Abbreviation for lists of numbers with identical step size:

```
>   [ 1 .. 10 ]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

- Other step size than 1 achieved by denoting a second element:

```
>  [ 1, 3 .. 10 ]
[1, 3, 5, 7, 9]
```

- Alternative definition of the factorial function (without explicit recursion):

```
fac n  =  prod  [ 1 .. n ]
```

List comprehensions (1)

- Powerful and elegant language construct in Haskell:

list comprehension

from "comprehensive"

- Modelled after implicit set notation in mathematics ("set of all x, such that ..."), e.g.,

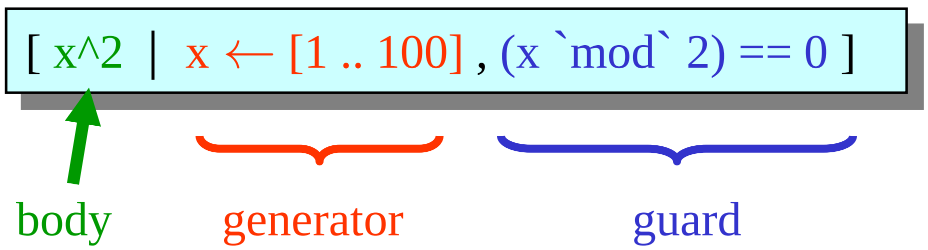
$$\{ \ x^2 \ | \ x \in \{1, ..., 100\} \ \wedge \ (x \bmod 2) = 0 \ \}$$

- In Haskell, analogous concept for lists:

```
[ x^2 | x ← [1 .. 100] , (x `mod` 2) == 0 ]
```

List comprehensions (2)

- A list comprehension general consists of three “ingredients”:



- The body represents list elements and is an expression, typically containing at least one variable, whose possible values are produced by the generator.

- The generator is an expression of the form variable ← list, which successively binds that variable to all elements of the list (in list order).

- The guard is a Boolean expression, which restricts the generated values to those for which that expression gives the value True.

- Additionally possible: local definitions with let.

List comprehensions (3)

- The parts are optional, e.g.,

```
[ x^2 | x ← [1 .. 10] ]
```

- A list comprehension may contain several variables with several generators, e.g.,

```
>  [ (x, y) | x ← [ 1, 2, 3 ], y ← [1 .. x] ]
[ (1, 1), (2, 1), (2, 2), (3, 1), (3, 2), (3, 3) ]
```

- Every variable (that is not known from outer context) needs a generator:

```
[ (x * y) | x ← [ 1, 2, 3 ], y ← [ 1, 2, 3 ] ]
```

but also

```
[ x ++ y | (x, y) ← [ ("a", "b"), ("c", "d") ] ]
```

List comprehensions (4)

- The order in which generators are given influences output order:

```
> [ (x, y) | x ← [ 1, 2, 3 ], y ← [ 4, 5 ] ]
[ (1, 4), (1, 5), (2, 4), (2, 5), (3, 4), (3, 5) ]
```

vs.

```
> [ (x, y) | y ← [ 4, 5 ], x ← [ 1, 2, 3 ] ]
[ (1, 4), (2, 4), (3, 4), (1, 5), (2, 5), (3, 5) ]
```

(like nested loops)

List comprehensions (5)

- "Later" generators can depend on "earlier" ones, e.g.,

```
> [ (x, y) | x ← [ 1, 2, 3 ],  y ← [1 .. x] ]
[ (1, 1), (2, 1), (2, 2), (3, 1), (3, 2), (3, 3) ]
```

- In particular, a variable bound via a generator can itself serve as a generator source, e.g.,

```
fun :: [[Int]] → [Int]
fun xss  =  [ x | xs ← xss, x ← xs ]
```

```
>  fun [ [ 1, 2, 3 ], [ 4, 5 ], [ 6 ], [ ] ]
[ 1, 2, 3, 4, 5, 6 ]
```

List comprehensions (6)

- Also guards can only depend on earlier generators, e.g.,

```
> [ x | x ← [1 .. 10], even x ]
[ 2, 4, 6, 8, 10 ]
```

- Yet another example:

```
factors :: Int → [Int]
factors n = [ x | x ← [1 .. n], n `mod` x == 0 ]
```

```
> factors 15
[ 1, 3, 5, 15 ]
```