Lecture 2: List algorithms using recursion and list comprehensions

Søren Haagerup

Department of Mathematics and Computer Science
University of Southern Denmark, Odense

October 31, 2016

Expressions, patterns and types

	Type	**Value constructors**	**Pattern / expression**
Tuple	(a, b)	$(,)::a \to b \to (a, b)$	(x, y)
List	$[a]$	$[\,] :: [a]$	$[\,]$
		$(:) :: a \to [a] \to [a]$	$(x : xs)$
Bool	$Bool$	$True :: Bool$	$True$
		$False :: Bool$	$False$
Maybe	$Maybe\ a$	$Nothing :: Maybe\ a$	$Nothing$
		$Just :: a \to Maybe\ a$	$(Just\ x)$

- Capitalized words: Specific type
- Lowercase words: Type variable

When “specializing” a type, all occurrences of a type variable in the type expression must be replaced with the same type.

Tuples - $e :: (a, b)$

Value constructor

$$(,) :: a \rightarrow b \rightarrow (a, b)$$

Pattern matching (destructing)

$$\begin{aligned} fst\ (x, \_) &= x \\ snd\ (\_, y) &= y \\ add\ (x, y) &= x + y \end{aligned}$$

Pattern matching is the *only* way to get values out of the tuple - functions from the standard library does this too.

Lists - $e :: [a]$

Value constructors

$$
\begin{aligned}
(:) &:: a \to [a] \to [a] \\
[\,] &:: [a]
\end{aligned}
$$

Pattern matching (destructing)

$$
\begin{aligned}
&sum :: [Integer] \to Integer \\
&sum\ [\,] \qquad\quad = 0 \\
&sum\ (x : xs) = x + sum\ xs
\end{aligned}
$$

$$
\begin{aligned}
&head\ (x : \_)\ = x \\
&tail\ (\_ : xs) = xs
\end{aligned}
$$

How can we define a function $length :: [a] \to Int$ to compute the length of a list?

Lists - $e :: [a]$

You are given two definitions of a function $isEmpty :: [a] \to Bool$
Which is better?

$$isEmpty :: [a] \to Bool$$
$$isEmpty\ [\,] = True$$
$$isEmpty\ \_ = False$$

$$isEmpty' :: [a] \to Bool$$
$$isEmpty'\ xs = length\ xs \equiv 0$$

Booleans

Not language primitives – Booleans and their operations are defined in the standard library! **Constructors**

$$\begin{aligned} &True :: Bool \\ &False :: Bool \end{aligned}$$

Pattern matching (destructing)

$$\begin{aligned} &True \wedge a \quad = a \\ &False \wedge \_ = False \end{aligned}$$

$$\begin{aligned} &False \vee a \;= a \\ &True \vee \_ \;= True \end{aligned}$$

We could define our own inline-if:

$$iif :: Boolean \rightarrow a \rightarrow a \rightarrow a$$

How would the definition look? Haskell also provides special syntax: **if** $(a < 5)$ **then** `"Hello"` **else** `"World"`.

Maybe - $e :: Maybe\ a$

Value constructors

$$\begin{aligned} Just &:: a \rightarrow Maybe\ a \\ Nothing &:: Maybe\ a \end{aligned}$$

Pattern matching (destructing)

$$\begin{aligned} maybeAdd\ Nothing\ \_ &= Nothing \\ maybeAdd\ \_\ Nothing &= Nothing \\ maybeAdd\ (Just\ x)\ (Just\ y) &= Just\ (x + y) \end{aligned}$$

Brush up: Types

- **Monomorphic types:**
  - *Int*, *Integer*, *Bool*, *Char*, *Float*, *Double*, *String*
- **Polymorphic types:**
  - $[a]$, *Maybe a*, (a, b)
  - lowercase letters are *type variables* which can be replaced by any other type to construct a new type
  - $[[a]]$, $[[[a]]]$, $[Maybe\ a]$, $Maybe\ (a, b)$, *Maybe Int* etc. are valid types

Brush up: Function types

An n-argument function is a one-argument function which returns a $(n-1)$-argument function.

$$add :: Int \to (Int \to Int)$$
$$add\ x\ y = x + y$$

Evaluate by calling add 40 2.

The same function in JavaScript would look like this:

```
function add(x) {
    return function(y) {
        return x+y;
    }
}
```

Evaluate by calling `add(40)(2)`.

Brush up: Function types

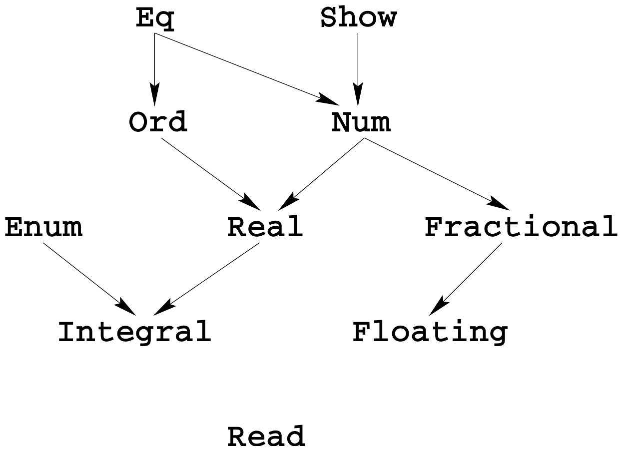
- **Monomorphic functions:** $words :: String \to [String]$
- **Polymorphic functions:**
  - $length :: [a] \to Int$
  - $(:) :: a \to [a] \to [a]$
    - This type signature ensures that lists can only be constructed with elements of the same type.
  - Can we make the following functions?
    - $sum :: [a] \to a$
    - $sort :: [a] \to [a]$

Type classes - Constraining the type of a function

- *Eq a* - all types *a* for which $(\equiv)$ is defined
- *Ord a* - all types *a* for which $(\leqslant)$ is defined
- *Num a* - all types *a* for which $(+), (*), abs, signum, fromInteger, negate$ are defined

$$
\begin{aligned}
sum, product\ &:: Num\ a \Rightarrow [a] \rightarrow a \\
sum\ []\ &= 0 \\
sum\ (x : xs)\ &= x + sum\ xs \\
product\ []\ &= 1 \\
product\ (x : xs)\ &= x * product\ xs
\end{aligned}
$$

Type classes - Constraining the type of a function



RECURSIVE LIST FUNCTIONS

Prelude: *take* and *drop*

$$
\begin{aligned}
&take :: Int \to [a] \to [a] \\
&take\ n\ \_ \mid n \leqslant 0 = [\,] \\
&take\ \_\ [\,] \qquad\qquad = [\,] \\
&take\ n\ (x : xs) \quad\ = x : take\ (n-1)\ xs
\end{aligned}
$$

$$
\begin{aligned}
take\ 3\ [1,2,3,4,5] &\equiv 1 : take\ 2\ [2,3,4,5] \\
&\equiv 1 : (2 : take\ 1\ [3,4,5]) \\
&\equiv 1 : (2 : (3 : take\ 0\ [4,5])) \\
&\equiv 1 : (2 : (3 : [\,])) \\
&\equiv [1,2,3]
\end{aligned}
$$

If the input list has length m, how many reductions are made?

Prelude: *take* and *drop*

$$
\begin{aligned}
&drop :: Int \to [a] \to [a] \\
&drop\ n\ xs \mid n \leqslant 0 = xs \\
&drop\ \_\ [\,] \qquad\qquad\quad = [\,] \\
&drop\ n\ (\_ : xs) \qquad = drop\ (n-1)\ xs
\end{aligned}
$$

$$
\begin{aligned}
drop\ 3\ [1,2,3,4,5] &\equiv drop\ 2\ [2,3,4,5] \\
&\equiv drop\ 1\ [3,4,5] \\
&\equiv drop\ 0\ [4,5] \\
&\equiv [4,5]
\end{aligned}
$$

If the input list has length m, how many reductions are made?

Prelude: takeWhile and *dropWhile*

$$
\begin{aligned}
&\mathit{takeWhile} :: (a \to \mathit{Bool}) \to [a] \to [a] \\
&\mathit{takeWhile}\ \_\ [\,] = [\,] \\
&\mathit{takeWhile}\ p\ (x : \mathit{xs}) \\
&\ \mid p\ x \qquad\qquad = x : \mathit{takeWhile}\ p\ \mathit{xs} \\
&\ \mid \mathit{otherwise} \quad = [\,]
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{dropWhile} :: (a \to \mathit{Bool}) \to [a] \to [a] \\
&\mathit{dropWhile}\ \_\ [\,] = [\,] \\
&\mathit{dropWhile}\ p\ (x : \mathit{xs}) \\
&\ \mid p\ x \qquad\qquad = \mathit{dropWhile}\ p\ \mathit{xs} \\
&\ \mid \mathit{otherwise} \quad = \mathit{xs}
\end{aligned}
$$

How many reductions are made?

Prelude: $(+\!\!+)$, *concat* and *reverse*

- $(+\!\!+) :: [a] \to [a] \to [a]$
  $$\begin{aligned} [\,] +\!\!+ ys &= ys \\ (x : xs) +\!\!+ ys &= x : (xs +\!\!+ ys) \end{aligned}$$

- $concat :: [[a]] \to [a]$
  $$\begin{aligned} concat\ [\,] &= [\,] \\ concat\ (xs : xss) &= xs +\!\!+ concat\ xss \end{aligned}$$

- $reverse :: [a] \to [a]$
  $$\begin{aligned} reverse\ [\,] &= [\,] \\ reverse\ (x : xs) &= reverse\ xs +\!\!+ [x] \end{aligned}$$

$(+\!\!+)$ - running the algorithm

$$
\begin{aligned}
&(+\!\!+) :: [a] \rightarrow [a] \\
&[\,] +\!\!+ ys \qquad\quad = ys \\
&(x : xs) +\!\!+ ys = x : (xs +\!\!+ ys)
\end{aligned}
$$

$$
\begin{aligned}
[1,2,3] +\!\!+ ys &\equiv 1 : ([2,3] +\!\!+ ys) \\
&\equiv 1 : (2 : ([3] +\!\!+ ys)) \\
&\equiv 1 : (2 : (3 : ([\,] +\!\!+ ys))) \\
&\equiv 1 : (2 : (3 : ys))
\end{aligned}
$$

How many reductions?

reverse - running the algorithm

$$
\begin{aligned}
&reverse :: [a] \to [a] \\
&reverse\ [\,] \qquad\quad = [\,] \\
&reverse\ (x : xs) = reverse\ xs +\!\!+ [x]
\end{aligned}
$$

$$
\begin{aligned}
reverse\ [1,2,3] &\equiv reverse\ [2,3] +\!\!+ [1] \\
&\equiv (reverse\ [3] +\!\!+ [2]) +\!\!+ [1] \\
&\equiv ((reverse\ [\,] +\!\!+ [3]) +\!\!+ [2]) +\!\!+ [1] \\
&\equiv (([\,] +\!\!+ [3]) +\!\!+ [2]) +\!\!+ [1] \\
&\equiv \ \dots \\
&\equiv [3,2,1]
\end{aligned}
$$

How many reductions?

Example: *trim*

$$
\begin{aligned}
\mathit{ltrim}\ \mathit{xs} &= \mathit{dropWhile}\ (\equiv \text{' '})\ \mathit{xs} \\
\mathit{rtrim}\ \mathit{xs} &= \mathit{reverse}\ (\mathit{ltrim}\ (\mathit{reverse}\ \mathit{xs})) \\
\mathit{trim}\ \mathit{xs} &= \mathit{rtrim}\ (\mathit{ltrim}\ \mathit{xs})
\end{aligned}
$$

Example: *trim*

$$
\begin{aligned}
\mathit{ltrim}\ \mathit{xs} &= \mathit{dropWhile}\ (\equiv \text{'\ '})\ \mathit{xs} \\
\mathit{rtrim}\ \mathit{xs} &= \mathit{reverse}\ \$\ \mathit{ltrim}\ \$\ \mathit{reverse}\ \mathit{xs} \\
\mathit{trim}\ \mathit{xs} &= \mathit{rtrim}\ \$\ \mathit{ltrim}\ \mathit{xs}
\end{aligned}
$$

Application operator. This operator is redundant, since ordinary application $(f\ x)$ means the same as $(f\ \$\ x)$. However, \$ has low, right-associative binding precedence, so it sometimes allows parentheses to be omitted

Example: *trim*

$$
\begin{aligned}
\mathit{ltrim} &= \mathit{dropWhile}\ (\equiv \texttt{'}\ \ \texttt{'}) \\
\mathit{rtrim} &= \mathit{reverse} \circ \mathit{ltrim} \circ \mathit{reverse} \\
\mathit{trim} &= \mathit{rtrim} \circ \mathit{ltrim}
\end{aligned}
$$

Point-free style. Sometimes it makes the code mode readable. Sometimes it doesn't (this is the reason, that some people call it *pointless style*).

Example: *left*, *right*, *mid* (inspired by VBScript)

$$
\begin{aligned}
\mathit{left}\ n &= \mathit{take}\ n \\
\mathit{right}\ n &= \mathit{reverse} \circ \mathit{take}\ n \circ \mathit{reverse} \\
\mathit{mid}\ s\ n &= \mathit{take}\ n \circ \mathit{drop}\ s
\end{aligned}
$$

Examples:

$$
\begin{aligned}
\mathit{left}\ 3\ \texttt{"abcde"} &= \texttt{"abc"} \\
\mathit{right}\ 3\ \texttt{"abcde"} &= \texttt{"cde"} \\
\mathit{mid}\ 2\ 2\ \texttt{"abcde"} &= \texttt{"cd"}
\end{aligned}
$$

Example: *substr* (inspired by PHP)

Syddansk Universitet
University of Southern Denmark

Description

```
string substr ( string $string , int $start [, int $length ] )
```

Returns the portion of **`string`** specified by the **`start`** and **`length`** parameters.

$$
\begin{aligned}
&\mathit{substr} :: [a] \to \mathit{Int} \to \mathit{Maybe\ Int} \to [a]\\
&\mathit{substr}\ \mathit{xs}\ s\ \mathit{Nothing} = \mathit{drop}\ s\ \mathit{xs}\\
&\mathit{substr}\ \mathit{xs}\ s\ (\mathit{Just}\ l) \quad = \mathit{take}\ l\ (\mathit{substr}\ \mathit{xs}\ s\ \mathit{Nothing})
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{substr}\ \texttt{"abracadabra"}\ 5\ \mathit{Nothing} = \texttt{"adabra"}\\
&\mathit{substr}\ \texttt{"abracadabra"}\ 5\ (\mathit{Just}\ 4) = \texttt{"adab"}
\end{aligned}
$$

Example: *substr* (inspired by PHP)

SYDDANSK UNIVERSITET
UNIVERSITY OF SOUTHERN DENMARK

But *substr* should work with negative offsets/lengths as well.

$$
\begin{aligned}
&\mathit{substr}\ \texttt{"abcdef"}\ (-1)\ \mathit{Nothing} &&= \texttt{"f"}\\
&\mathit{substr}\ \texttt{"abcdef"}\ (-2)\ \mathit{Nothing} &&= \texttt{"ef"}\\
&\mathit{substr}\ \texttt{"abcdef"}\ (-3)\ (\mathit{Just}\ 1) &&= \texttt{"d"}\\
&\mathit{substr}\ \texttt{"abcdef"}\ 0\ (\mathit{Just}\ (-1)) &&= \texttt{"abcde"}\\
&\mathit{substr}\ \texttt{"abcdef"}\ 2\ (\mathit{Just}\ (-1)) &&= \texttt{"cde"}\\
&\mathit{substr}\ \texttt{"abcdef"}\ 4\ (\mathit{Just}\ (-4)) &&= \texttt{""}\\
&\mathit{substr}\ \texttt{"abcdef"}\ (-3)\ (\mathit{Just}\ (-1)) &&= \texttt{"de"}
\end{aligned}
$$

Example: *substr* (inspired by PHP)

But *substr* should work with negative offsets/lengths as well.

$$
\begin{aligned}
&\mathit{substr} :: [a] \to \mathit{Int} \to \mathit{Maybe}\ \mathit{Int} \to [a] \\
&\mathit{substr}\ \mathit{xs}\ s\ \mathit{Nothing} = \mathit{drop}\ (\mathit{nonneg}\ \mathit{xs}\ s)\ \mathit{xs} \\
&\mathit{substr}\ \mathit{xs}\ s\ (\mathit{Just}\ l) \ \ = \mathit{take}\ (\mathit{nonneg}\ \mathit{xs}'\ l)\ \mathit{xs}' \\
&\quad \mathbf{where}\ \mathit{xs}' = \mathit{substr}\ \mathit{xs}\ s\ \mathit{Nothing}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{nonneg} :: [a] \to \mathit{Int} \to \mathit{Int} \\
&\mathit{nonneg}\ \mathit{xs}\ n \\
&\quad \mid n < 0 \qquad\ \ = \mathit{max}\ 0\ (\mathit{length}\ \mathit{xs} + n) \\
&\quad \mid \mathit{otherwise} = n
\end{aligned}
$$

Prelude: *zip*

$$
\begin{aligned}
&zip :: [a] \to [b] \to [(a,b)] \\
&zip\ [\,]\ \_ && = [\,] \\
&zip\ \_\ [\,] && = [\,] \\
&zip\ (x:xs)\ (y:ys) && = (x,y) : zip\ xs\ ys
\end{aligned}
$$

$> zip\ [1\,..\,5]$ `"abcd"`

$[(1,'\texttt{a}'),(2,'\texttt{b}'),(3,'\texttt{c}'),(4,'\texttt{d}')]$

Prelude: zipWith

$$
\begin{aligned}
&\mathit{zipWith} :: (a \to b \to c) \to [a] \to [b] \to [c] \\
&\mathit{zipWith}\ \_\ [\,]\ \_ &&= [\,] \\
&\mathit{zipWith}\ \_\ \_\ [\,] &&= [\,] \\
&\mathit{zipWith}\ f\ (x : xs)\ (y : ys) &&= f\ x\ y : \mathit{zipWith}\ f\ xs\ ys
\end{aligned}
$$

$$
\mathit{zip} = \mathit{zipWith}\ (,)
$$

$> \mathit{zipWith}\ (+)\ [1\,..\,5]\ [5, 4\,..\,1]$

$[6, 6, 6, 6, 6]$

Insertion sort

$$
\begin{aligned}
&\mathit{insert} :: \mathit{Ord}\ a \Rightarrow a \rightarrow [a] \rightarrow [a]\\
&\mathit{insert}\ x\ [\,] \qquad\qquad\qquad\qquad\ = [x]\\
&\mathit{insert}\ x\ (y : \mathit{ys})\ \mid x \leqslant y \qquad = x : y : \mathit{ys}\\
&\qquad\qquad\qquad\qquad \mid \mathit{otherwise} = y : \mathit{insert}\ x\ \mathit{ys}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{isort} :: \mathit{Ord}\ a \Rightarrow [a] \rightarrow [a]\\
&\mathit{isort}\ [\,] \qquad\ \ = [\,]\\
&\mathit{isort}\ (x : \mathit{xs}) = \mathit{insert}\ x\ (\mathit{isort}\ \mathit{xs})
\end{aligned}
$$

Merge sort

$$
\begin{aligned}
&\mathit{merge} :: \mathit{Ord}\ a \Rightarrow [a] \rightarrow [a] \rightarrow [a]\\
&\mathit{merge}\ \mathit{xs}\ [\,] && = \mathit{xs}\\
&\mathit{merge}\ [\,]\ \mathit{ys} && = \mathit{ys}\\
&\mathit{merge}\ (x : \mathit{xs})\ (y : \mathit{ys})\ \mid x \leqslant y && = x : \mathit{merge}\ \mathit{xs}\ (y : \mathit{ys})\\
&\qquad\qquad\qquad\qquad\quad \mid \mathit{otherwise} && = y : \mathit{merge}\ (x : \mathit{xs})\ \mathit{ys}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{msort} :: \mathit{Ord}\ a \Rightarrow [a] \rightarrow [a]\\
&\mathit{msort}\ [\,] = [\,]\\
&\mathit{msort}\ [x] = [x]\\
&\mathit{msort}\ \mathit{xs} = \mathit{merge}\ (\mathit{msort}\ \mathit{ys})\ (\mathit{msort}\ \mathit{zs})\\
&\quad \textbf{where}\\
&\qquad (\mathit{ys}, \mathit{zs}) = \mathit{splitAt}\ (\mathit{length}\ \mathit{xs}\ `\mathit{div}`\ 2)\ \mathit{xs}
\end{aligned}
$$

Lab this Friday: Polynomials

A polynomial $p : \mathbb{R} \to \mathbb{R}$ with degree n is a function

$$p(x) = a_0 x^0 + a_1 x^1 + \ldots + a_n x^n$$

where $a_0 \ldots a_n$ are constants in $\mathbb{R}$, $a_n \neq 0$.
In Haskell we define a type synonym

$$\textbf{type}\ Poly\ a = [a]$$

and let a polynomial be defined by the list of its coefficients

$$p :: Num\ a \Rightarrow Poly\ a$$
$$p = [a0, a1\ ...\ an]$$

Lab this Friday: Polynomials

Examples:

- $5 + 2x + 3x^2$ is represented by $[5, 2, 3]$
- $-2 + x^2$ is represented by $[-2, 0, 1]$
- 0 is represented by $[]$

Lab this Friday: Polynomials

Think about this in the break:

1. We discover that $-2 + x^2$ can be represented by infinitely many lists:
   $[-2, 0, 1], [-2, 0, 1, 0], [-2, 0, 1, 0, 0], [-2, 0, 1, 0, 0, 0] \ldots.$

   Inspired by *trim*, write a function *canonical* that converts a polynomial to its smallest representation.
2. We want to define addition of polynomials, such that

   $$(5 + 2x + 3x^2) + (-2 + x) = 3 + 3x + 3x^2$$

   i.e.

   $$add\ [5, 2, 3]\ [-2, 1] = [5 + (-2), 2 + 1, 3] = [3, 3, 3]$$

   Modify *zip* to implement *add*.

LIST COMPREHENSIONS

Introduction

In mathematics, the set of square numbers up to 5^2 is

$$\{x^2 \mid x \in \{1, \ldots, 5\}\}$$

In Haskell, the list of square numbers up to 5^2 can be written

$$[x * x \mid x \leftarrow [1 \mathinner{..} 5]]$$

We say

- $\mid$ "such that"
- $\leftarrow$ "is drawn from"
- $x \leftarrow xs$ is a "generator"

Cartesian product

$$cartesian\ xs\ ys = [(x, y) \mid x \leftarrow xs, y \leftarrow ys]$$

$> cartesian\ [1\,..\,3]$ `"abc"`
$[(1, \text{'a'}), (1, \text{'b'}), (1, \text{'c'}),$
$(2, \text{'a'}), (2, \text{'b'}), (2, \text{'c'}),$
$(3, \text{'a'}), (3, \text{'b'}), (3, \text{'c'})]$

Ordering matters!

$$cartesian'\ xs\ ys = [(x, y) \mid y \leftarrow ys, x \leftarrow xs]$$

$> cartesian'\ [1\,..\,3]$ `"abc"`
$[(1, \text{'a'}), (2, \text{'a'}), (3, \text{'a'}),$
$(1, \text{'b'}), (2, \text{'b'}), (3, \text{'b'}),$
$(1, \text{'c'}), (2, \text{'c'}), (3, \text{'c'})]$

Finding index of elements in a list

$$elemIndices :: Eq\ a \Rightarrow a \rightarrow [a] \rightarrow [Int]$$
$$elemIndices\ xs\ y = [i \mid (i,x) \leftarrow zip\ [0\,..\,length\ xs]\ xs, x \equiv y]$$

The boolean expression $x \equiv y$ is called a **guard**.

$$> elemIndices\ [3,4,2,1,4,5]\ 4$$
$$[2,5]$$

$$
\begin{aligned}
\mathit{pythags}\ n = \ & [\ (x, y, z) \\
& \mid z \leftarrow [1\mathinner{..}n], \\
& \quad x \leftarrow [1\mathinner{..}z], \\
& \quad y \leftarrow [x\mathinner{..}z], \\
& \quad x * x + y * y \equiv z * z]
\end{aligned}
$$

$> \mathit{pythags}\ 15$

$[(3,4,5),(6,8,10),(5,12,13),(9,12,15)]$

```
     start
     |______________________...
     |  |       |
z    1  2       3
     |  |___    |_________
     |  |   |   |     |   |
x    1  1   2   1     2   3
     |  |__ |   |___  |_  |
     |  | | |   | | | | | |
y    1  1 2 2   1 2 3 2 3 3
```

Prelude functions
- here implemented using list comprehensions

- $zipWith\ f\ xs\ ys = [f\ a\ b \mid (a,b) \leftarrow zip\ xs\ ys]$
  Example: $zipWith\ (+)\ [2,1,3]\ [3,1,2] = [5,2,5]$
- $concat\ xss = [x \mid xs \leftarrow xss, x \leftarrow xs]$
  Example: $concat\ [[1],[1,2],[1,2,3]] = [1,1,2,1,2,3]$
- $map\ f\ xs = [f\ x \mid x \leftarrow xs]$
  Example: $map\ (*3)\ [1,2,3,4] = [3,6,9,12]$
- $filter\ p\ xs = [x \mid x \leftarrow xs, p\ x]$
  Example: $filter\ even\ [6,2,7,5,2] = [6,2,2]$

Checking if a list is sorted

$$sorted\ xs = and\ [x \leqslant y \mid (x, y) \leftarrow zip\ xs\ (tail\ xs)]$$

$$\begin{aligned} sorted\ [2, 3, 1] &\equiv and\ [True, False] \\ &\equiv False \end{aligned}$$

Pascal's triangle

$$
\begin{array}{ccc}
\binom{0}{0} & \qquad & 1 \\
\binom{1}{0} \; \binom{1}{1} & & 1 \; 1 \\
\binom{2}{0} \; \binom{2}{1} \; \binom{2}{2} & & 1 \; 2 \; 1 \\
\binom{3}{0} \; \binom{3}{1} \; \binom{3}{2} \; \binom{3}{3} & & 1 \; 3 \; 3 \; 1
\end{array}
$$

$$
(a+b)^n = \sum_{i=0}^{n} \binom{n}{i} a^i b^{n-i}
$$

$$
\begin{aligned}
(a+b)^3 &= \binom{3}{0} b^3 + \binom{3}{1} ab^2 + \binom{3}{2} + a^2 b + \binom{3}{3} a^3 \\
&= b^3 + 3ab^2 + 3ba^2 + a^3
\end{aligned}
$$

Pascal's triangle

$$
\begin{array}{c}
\binom{0}{0} \\
\binom{1}{0} \ \binom{1}{1} \\
\binom{2}{0} \ \binom{2}{1} \ \binom{2}{2} \\
\binom{3}{0} \ \binom{3}{1} \ \binom{3}{2} \ \binom{3}{3}
\end{array}
\qquad
\begin{array}{c}
1 \\
1\ 1 \\
1\ 2\ 1 \\
1\ 3\ 3\ 1
\end{array}
$$

$$\mathit{pascal}\ \mathit{xs} = [1] \mathbin{+\!\!+} [x + y \mid (x, y) \leftarrow \mathit{zip}\ \mathit{xs}\ (\mathit{tail}\ \mathit{xs})] \mathbin{+\!\!+} [1]$$

$$
\begin{aligned}
\mathit{pascal}\ [1] &= [1, 1] \\
\mathit{pascal}\ [1, 1] &= [1, 2, 1] \\
\mathit{pascal}\ [1, 2, 1] &= [1, 3, 3, 1] \\
\mathit{pascal}\ [1, 3, 3, 1] &= [1, 4, 6, 4, 1] \\
\mathit{pascal}\ [1, 4, 6, 4, 1] &= [1, 5, 10, 10, 5, 1]
\end{aligned}
$$

Prime numbers

A *prime number* p is a number where its only divisors are 1 and p.

$$\begin{aligned}
\mathit{divisors}\ n &= [x \mid x \leftarrow [1\,..\,n], n\ \text{`}\mathit{mod}\text{`}\ x \equiv 0] \\
\mathit{prime}\ n &= \mathit{divisors}\ n \equiv [1, n] \\
\mathit{primes}\ n &= [x \mid x \leftarrow [2\,..\,n], \mathit{prime}\ x]
\end{aligned}$$

Caesar cipher

$\textbf{import}\ \mathit{Data.Char}\ (\mathit{ord}, \mathit{chr}, \mathit{isLower})$

$\mathit{char2int} :: \mathit{Char} \to \mathit{Int}$

$\mathit{char2int}\ c = \mathit{ord}\ c - \mathit{ord}\ \texttt{'a'}$ -- a=0, b=1 ...

$\mathit{int2char} :: \mathit{Int} \to \mathit{Char}$

$\mathit{int2char}\ n = \mathit{chr}\ (\mathit{ord}\ \texttt{'a'} + n)$ -- 0=a, 1=b ...

$$\begin{aligned} \mathit{shift}\ n\ c\ &|\ \mathit{isLower}\ c = \mathit{int2char}\ ((\mathit{char2int}\ c + n)\ \text{`}\mathit{mod}\text{`}\ 26) \\ &|\ \mathit{otherwise} = c \end{aligned}$$

$\mathit{encode}\ n\ \mathit{xs} = [\mathit{shift}\ n\ x \mid x \leftarrow \mathit{xs}]$

$\mathit{decode}\ n\ \mathit{xs} = [\mathit{shift}\ (-n)\ x \mid x \leftarrow \mathit{xs}]$

$\mathit{encode}\ 3$ `"haskell er fantastisk"`

$\equiv$ `"kdvnhoo hu idqwdvwlvn"`

Generating bitstrings

$$
\begin{aligned}
&\mathit{bitstrings}\ 0 = [[\,]] \\
&\mathit{bitstrings}\ n = [b : bs \mid b \leftarrow [0,1], bs \leftarrow \mathit{bitstrings}\ (n-1)]
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{bitstrings}\ 0 \equiv [\,] \\
&\mathit{bitstrings}\ 1 \equiv [[0,1]] \\
&\mathit{bitstrings}\ 2 \equiv [[0,0],[0,1],[1,0],[1,1]] \\
&\mathit{bitstrings}\ 3 \equiv [[0,0,0],[0,0,1],[0,1,0],[0,1,1], \\
&\qquad\qquad\quad [1,0,0],[1,0,1],[1,1,0],[1,1,1]]
\end{aligned}
$$

Finding the transpose of a matrix

$$
\begin{aligned}
&\mathit{transpose} :: [[a]] \to [[a]] \\
&\mathit{transpose}\ [\,] &&= [\,] \\
&\mathit{transpose}\ ([\,] : \mathit{xss}) &&= \mathit{transpose}\ \mathit{xss} \\
&\mathit{transpose}\ \mathit{xss} &&= \quad [x \mid (x{:}\ \_) \leftarrow \mathit{xss}] \\
& && \qquad : \mathit{transpose}\ [\mathit{xs} \mid (\_ : \mathit{xs}) \leftarrow \mathit{xss}]
\end{aligned}
$$

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} \qquad A^T = \begin{pmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{pmatrix}$$

Finding the transpose of a matrix

$$
\begin{aligned}
&\mathit{transpose} :: [[a]] \to [[a]]\\
&\mathit{transpose}\ [\,] && = [\,]\\
&\mathit{transpose}\ ([\,] : \mathit{xss}) && = \mathit{transpose}\ \mathit{xss}\\
&\mathit{transpose}\ \mathit{xss} && = \quad [x \mid (x{:}\ \_) \leftarrow \mathit{xss}]\\
& && \quad : \mathit{transpose}\ [\mathit{xs} \mid (\_ : \mathit{xs}) \leftarrow \mathit{xss}]
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{transpose}\ [[1,2,3],[4,5,6]]\\
&\quad \equiv [1,4] : \mathit{transpose}\ [[2,3],[5,6]]\\
&\quad \equiv [1,4] : [2,5] : \mathit{transpose}\ [[3],[6]]\\
&\quad \equiv [1,4] : [2,5] : [3,6] : \mathit{transpose}\ [[\,],[\,]]\\
&\quad \equiv [1,4] : [2,5] : [3,6] : \mathit{transpose}\ [[\,]]\\
&\quad \equiv [1,4] : [2,5] : [3,6] : \mathit{transpose}\ [\,]\\
&\quad \equiv [1,4] : [2,5] : [3,6] : [\,]\\
&\quad \equiv [[1,4],[2,5],[3,6]]
\end{aligned}
$$

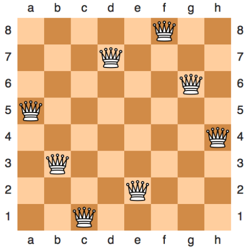
Generating permutations

$$
\begin{array}{l}
permutations\ [\,] = [[\,]] \\
permutations\ (x : xs) = [ys' +\!\!+ x : ys'' \mid \\
\quad ys \leftarrow permutations\ xs, \\
\quad i \leftarrow [0 \mathinner{..} length\ ys], \\
\quad \textbf{let}\ (ys', ys'') = splitAt\ i\ ys]
\end{array}
$$

$> permutations\ [\,]$
$[[\,]]$
$> permutations\ [1]$
$[[1]]$
$> permutations\ [1,2]$
$[[1,2],[2,1]]$
$> permutations\ [1,2,3]$
$[[1,2,3],[2,1,3],[2,3,1],[1,3,2],[3,1,2],[3,2,1]]$

Solving the n-queens problem

The **eight queens puzzle** is the problem of placing eight chess queens on an 8×8 chessboard so that no two queens threaten each other. Thus, a solution requires that **no two queens share the same row, column, or diagonal**. The eight queens puzzle is an example of the more general **n-queens problem** of placing n queens on an $n \times n$ chessboard.



Backtracking - n-queens problem

SYDDANSK UNIVERSITET
UNIVERSITY OF SOUTHERN DENMARK

$$validExtensions\ n\ qs = [q : qs \mid q \leftarrow [1 \mathinner{..} n] \setminus\setminus qs, q\ \text{`}notDiag\text{`}\ qs]$$

where

$$q\ \text{`}notDiag\text{`}\ qs = and\ [\ abs\ (q - qi) \not\equiv i \mid (qi, i) \leftarrow qs\ \text{`}zip\text{`}\ [1 \mathinner{..} n]]$$

$$queens'\ n\ 0 = [[\,]]$$

$$queens'\ n\ i = [\ qs' \mid qs \leftarrow queens'\ n\ (i-1),\ qs' \leftarrow validExtensions\ n\ qs]$$

$$queens\ n = queens'\ n\ n$$

$> queens\ 8$

$[[4,2,7,3,6,8,5,1],[5,2,4,7,3,8,6,1],[3,5,2,8,6,4,7,1]...]$