Lecture 2: List algorithms using recursion and list comprehensions

Søren Haagerup

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

September 12, 2017
Expressions, patterns and types

Primitive types: \( \text{Int}, \text{Integer}, \text{Double}, \text{Float}, \text{Char} \)

Usage of primitive types:

\[
\begin{align*}
c &:: \text{Int} \\
c &= 42 \\
f &:: \text{Int} \to \text{Int} \\
f \ x &= 1337 - x
\end{align*}
\]
Primitive types:  \( Int, Integer, Double, Float, Char \)

Pattern matching on primitive types:

\[
\begin{align*}
\text{sumTo} &: \text{Int} \rightarrow \text{Int} \\
\text{sumTo} \ 0 &= 0 \\
\text{sumTo} \ n &= n + \text{sumTo} \ (n - 1)
\end{align*}
\]
Expressions, patterns and types

Non-primitive types: Monomorphic: (without type variables)

\textbf{data} \textit{Bool} = \textit{False} | \textit{True}

Pattern matching on non-primitive types:

\textit{invert} \quad :: \textit{Bool} \rightarrow \textit{Bool}

\textit{invert False} = \textit{True}

\textit{invert True} = \textit{False}

\textit{invert} \quad :: \textit{Bool} \rightarrow \textit{Bool}

\textit{invert False} = \textit{True}

\textit{invert _} = \textit{False}

Wildcards are written as \_
Expressions, patterns and types

Non-primitive types: Polymorphic: (with type variables)

```
data Maybe a = Nothing | Just a
```
```
data List a = Nil | Cons a (List a)
```
```
data [a] = [] | (a : [a])
```
```
data (a, b) = (a, b)
```

Example 1/2

```
maybeAdd :: Maybe Int → Maybe Int → Maybe Int
maybeAdd (Just x) (Just y) = Just (x + y)
maybeAdd _ _ = Nothing
```

Example 2/2

```
f :: (Int, Int) → Int
f (0, y) = y
f (x, y) = (x - 1, x * y)
```
### Expressions, patterns and types

<table>
<thead>
<tr>
<th>Type</th>
<th>Value constructors</th>
<th>Pattern / expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tuple</td>
<td>((a, b))</td>
<td>((x, y))</td>
</tr>
<tr>
<td>List</td>
<td>([a])</td>
<td>([\ ])</td>
</tr>
<tr>
<td></td>
<td>((:) :: a \rightarrow [a] \rightarrow [a])</td>
<td>((x : xs))</td>
</tr>
<tr>
<td>Bool</td>
<td>(\text{False})</td>
<td>(\text{False})</td>
</tr>
<tr>
<td></td>
<td>(\text{True})</td>
<td>(\text{True})</td>
</tr>
<tr>
<td>Maybe</td>
<td>(\text{Nothing})</td>
<td>(\text{Nothing})</td>
</tr>
<tr>
<td></td>
<td>(\text{Just})</td>
<td>(\text{Just}\ x)</td>
</tr>
</tbody>
</table>

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

\[
add :: \text{Int} \to (\text{Int} \to \text{Int})
\]
\[
add \ x \ y = x + y
\]

Evaluate by calling \texttt{add 40 2}.

The same function in JavaScript would look like this:

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

Evaluate by calling \texttt{add(40)(2)}.
Brush up: Function types

- **Monomorphic functions:**  
  \( \text{words} :: \text{String} \rightarrow [\text{String}] \)

- **Polymorphic functions:**
  - \( \text{length} :: [a] \rightarrow \text{Int} \)
  - \( (:) :: a \rightarrow [a] \rightarrow [a] \)
    - This type signature ensures that lists can only be constructed with elements of the same type.
  - Can we make the following functions?
    - \( \text{sum} :: [a] \rightarrow a \)
    - \( \text{sort} :: [a] \rightarrow [a] \)
Type classes - Constraining the type of a function

- **Eq a** - all types `a` for which `(≡)` is defined
- **Ord a** - all types `a` for which `(≤)` is defined
- **Num a** - all types `a` for which `(+)`, `(*)`, `abs`, `signum`, `fromInteger`, `negate` are defined

```
sum, product :: Num a ⇒ [a] → a
sum [] = 0
sum (x : xs) = x + sum xs
product [] = 1
product (x : xs) = x * product xs
```
Type classes - Constraining the type of a function

- **Eq**
- **Ord**
- **Show**
- **Num**
- **Enum**
  - **Integral**
- **Real**
- **Fractional**
  - **Floating**
- **Read**
RECURSIVE LIST FUNCTIONS
Prelude: take and drop

\[
\text{take} \colon \text{Int} \rightarrow [a] \rightarrow [a]
\]

\[
\text{take}\ n\ _\mid\ n\ \leq\ 0\ =\ []
\]

\[
\text{take}\ _\ [\ ]\ =\ []
\]

\[
\text{take}\ n\ (x:xss)\ =\ x:\text{take}\ (n-1)\ xss
\]

\[
\text{take}\ 3\ [1,2,3,4,5]\ \equiv\ 1: \text{take}\ 2\ [2,3,4,5]
\]

\[
\equiv\ 1: (2: \text{take}\ 1\ [3,4,5])
\]

\[
\equiv\ 1: (2: (3: \text{take}\ 0\ [4,5]))
\]

\[
\equiv\ 1: (2: (3: []))
\]

\[
\equiv\ [1,2,3]
\]

If the input list has length \( m \), how many reductions are made?
Prelude: take and drop

\[
\text{drop} :: \text{Int} \rightarrow [a] \rightarrow [a]
\]

\[
\text{drop } n \ x s \mid n \leq 0 = x s
\]

\[
\text{drop } _\_ [\ ] = [\ ]
\]

\[
\text{drop } n \ (\_ : x s) = \text{drop } (n - 1) x s
\]

\[
\text{drop } 3 \ [1, 2, 3, 4, 5] \equiv \text{drop } 2 \ [2, 3, 4, 5] \\
\quad \equiv \text{drop } 1 \ [3, 4, 5] \\
\quad \equiv \text{drop } 0 \ [4, 5] \\
\quad \equiv [4, 5]
\]

If the input list has length \( m \), how many reductions are made?
Prelude: `takeWhile` and `dropWhile`

\[
\text{takeWhile :: } (a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a]
\]
\[
\text{takeWhile } \_ \ [\ ] = [\ ]
\]
\[
\text{takeWhile } p \ (x : xs)
\]
\[
| \ p \ x \ = \ x : \text{takeWhile } p \ xs
\]
\[
| \ \text{otherwise} \ = \ [\ ]
\]

\[
\text{dropWhile :: } (a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a]
\]
\[
\text{dropWhile } \_ \ [\ ] = [\ ]
\]
\[
\text{dropWhile } p \ (x : xs)
\]
\[
| \ p \ x \ = \ \text{dropWhile } p \ xs
\]
\[
| \ \text{otherwise} \ = \ xs
\]

How many reductions are made?
Prelude: (+ +), concat and reverse

- \((+ +) :: [a] \rightarrow [a] \rightarrow [a]\)
  \([\ ] + ys \quad = \quad ys\)
  \((x : xs) + ys = x : (xs + ys)\)

- \(concat :: [[[a]]] \rightarrow [a]\)
  \(concat [\ ] \quad = \quad [\ ]\)
  \(concat (xs : xss) = xs + concat xss\)

- \(reverse :: [a] \rightarrow [a]\)
  \(reverse [\ ] \quad = \quad [\ ]\)
  \(reverse (x : xs) = reverse xs + [x]\)
(++) - running the algorithm

(++) :: [a] → [a]

[] ++ ys = ys

(x : xs) ++ ys = x : (xs ++ ys)

[1, 2, 3] ++ ys ≡ 1 : ([2, 3] ++ ys)
  ≡ 1 : (2 : ([3] ++ ys))
  ≡ 1 : (2 : (3 : ([] ++ ys)))
  ≡ 1 : (2 : (3 : ys))

How many reductions?
reverse - running the algorithm

\[
\begin{align*}
\text{reverse} :: [a] & \rightarrow [a] \\
\text{reverse} \; [] & = [] \\
\text{reverse} \; (x : xs) & = \text{reverse} \; xs \; \mathbin{\#} \; [x] \\
\end{align*}
\]

\[
\text{reverse} \; [1, 2, 3] \equiv \text{reverse} \; [2, 3] \; \mathbin{\#} \; [1] \\
\equiv (\text{reverse} \; [3] \; \mathbin{\#} \; [2]) \; \mathbin{\#} \; [1] \\
\equiv ((\text{reverse} \; [] \; \mathbin{\#} \; [3]) \; \mathbin{\#} \; [2]) \; \mathbin{\#} \; [1] \\
\equiv ((([] \; \mathbin{\#} \; [3]) \; \mathbin{\#} \; [2]) \; \mathbin{\#} \; [1] \\
\equiv ... \\
\equiv [3, 2, 1]
\]

How many reductions?
Example: *trim*

\[
\begin{align*}
\text{ltrim } xs & = \text{dropWhile } (\equiv ' ') \; xs \\
\text{rtrim } xs & = \text{reverse } (\text{ltrim } (\text{reverse } xs)) \\
\text{trim } xs & = \text{rtrim } (\text{ltrim } xs)
\end{align*}
\]
Example: \textit{trim}

\begin{align*}
ltrim\;xs &= \text{dropWhile}\;\left(\equiv\,'\;\right)\;xs \\
rtrim\;xs &= \text{reverse} \;\$\;\text{ltrim} \;\$\;\text{reverse} \;xs \\
\text{trim}\;xs &= \text{rtrim} \;\$\;\text{ltrim}\;xs
\end{align*}

\textbf{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*

\[
ltrim = \text{dropWhile} \ (\equiv \ ' \ ')
\]
\[
rtrim = \text{reverse} \circ ltrim \circ \text{reverse}
\]
\[
trim = rtrim \circ ltrim
\]

**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)

\[
\begin{align*}
    \text{left } n &= \text{take } n \\
    \text{right } n &= \text{reverse} \circ \text{take } n \circ \text{reverse} \\
    \text{mid } s \ n &= \text{take } n \circ \text{drop } s
\end{align*}
\]

Examples:

\[
\begin{align*}
    \text{left } 3 \ "abcde" &= "abc" \\
    \text{right } 3 \ "abcde" &= "cde" \\
    \text{mid } 2 2 \ "abcde" &= "cd"
\end{align*}
\]
Example: *substr* (inspired by PHP)

**Description**

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

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

```
substr :: [a] → Int → Maybe Int → [a]
substr xs s Nothing  = drop s xs
substr xs s (Just l) = take l (substr xs s Nothing)
```

```
substr "abracadabra" 5 Nothing  = "adabra"
substr "abracadabra" 5 (Just 4) = "adab"
```
Example: \texttt{substr} (inspired by PHP)

But \textsf{substr} should work with negative offsets/lengths as well.

\begin{verbatim}
substr "abcdef" (-1) Nothing = "f"
substr "abcdef" (-2) Nothing = "ef"
substr "abcdef" (-3) (Just 1) = "d"
substr "abcdef" 0 (Just (-1)) = "abcde"
substr "abcdef" 2 (Just (-1)) = "cde"
substr "abcdef" 4 (Just (-4)) = ""
substr "abcdef" (-3) (Just (-1)) = "de"
\end{verbatim}
Example: substr (inspired by PHP)

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

\[
\text{substr} :: [a] \rightarrow \text{Int} \rightarrow \text{Maybe Int} \rightarrow [a]
\]

\[
\text{substr } xs \, s \, \text{Nothing} = \text{drop} \ (\text{nonneg } xs \, s) \, xs
\]

\[
\text{substr } xs \, s \, (\text{Just } l) = \text{take} \ (\text{nonneg } xs' \, l) \, xs'
\]

where \( xs' = \text{substr } xs \, s \, \text{Nothing} \)

\[
\text{nonneg} :: [a] \rightarrow \text{Int} \rightarrow \text{Int}
\]

\[
\text{nonneg } xs \, n
\]

| \( n < 0 \) \quad = \max 0 \ (\text{length } xs + n) \\
| \text{otherwise} \quad = n
Prelude: zip

\[ \text{zip} :: [a] \rightarrow [b] \rightarrow [(a, b)] \]
\[ \text{zip} \ [\] \_ = [] \]
\[ \text{zip} \ _ \ [\] = [] \]
\[ \text{zip} \ (x : xs) \ (y : ys) = (x, y) : \text{zip} \ xs \ ys \]

\[ > \text{zip} \ [1..5] \ "abcd" \]
\[ [(1,'a'),(2,'b'),(3,'c'),(4,'d')] \]
Prelude: zipWith

\[ \text{zipWith} :: (a \to b \to c) \to [a] \to [b] \to [c] \]
\[ \text{zipWith} \_ [] \_ = [] \]
\[ \text{zipWith} \_ \_ [] = [] \]
\[ \text{zipWith} f (x:xs) (y:ys) = f x y : \text{zipWith} f xs ys \]

\[ \text{zip} = \text{zipWith} (,) \]

\[ > \text{zipWith} (+) [1..5] [5,4..1] [6,6,6,6,6] \]
**Insertion sort**

\[\text{insert} :: \text{Ord } a \Rightarrow a \rightarrow [a] \rightarrow [a]\]

\[\text{insert } x \ [\] = [x]\]

\[\text{insert } x \ (y : ys) \mid x \leq y = x : y : ys\]

\[\mid \text{otherwise} = y : \text{insert } x \ ys\]

\[\text{isort} :: \text{Ord } a \Rightarrow [a] \rightarrow [a]\]

\[\text{isort } [\] = []\]

\[\text{isort } (x : xs) = \text{insert } x \ (\text{isort } xs)\]
Merge sort

\[\text{merge} :: \text{Ord } a \Rightarrow [a] \rightarrow [a] \rightarrow [a]\]

\[\text{merge } xs \; [] = xs\]

\[\text{merge } [] \; ys = ys\]

\[\text{merge } (x : xs) \; (y : ys) \mid x \leq y = x : \text{merge } xs \; (y : ys)\]

\[\mid \;	ext{otherwise} = y : \text{merge } (x : xs) \; ys\]

\[\text{msort} :: \text{Ord } a \Rightarrow [a] \rightarrow [a]\]

\[\text{msort } [] = []\]

\[\text{msort } [x] = [x]\]

\[\text{msort } xs = \text{merge } (\text{msort } ys) \; (\text{msort } zs)\]

\[\text{where}\]

\[(ys, zs) = \text{splitAt } (\text{length } xs \div 2) \; xs\]
Lab this Friday: Polynomials

A polynomial $p : \mathbb{R} \rightarrow \mathbb{R}$ 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

```
  type Poly a = [a]
```

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

```
  p :: Num a => 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 $[]$
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.

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

Modify `zip` to implement `add`. 

---

**Lab this Friday: Polynomials**
LIST COMPREHENSIONS
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

$$[x \times x \mid x \leftarrow [1..5]]$$

We say

- $\mid$ “such that”
- $\leftarrow$ “is drawn from”
- $x \leftarrow xs$ is a “generator”
Cartesian product

\[
\text{cartesian } xs \ ys = [(x, y) \mid x \leftarrow xs, y \leftarrow ys]
\]

\[
> \text{cartesian } [1\ldots3] \ "abc"
\]
\[
[(1,'a'), (1,'b'), (1,'c'),
(2,'a'), (2,'b'), (2,'c'),
(3,'a'), (3,'b'), (3,'c')]
\]

Ordering matters!

\[
\text{cartesian'} \ xs \ ys = [(x, y) \mid y \leftarrow ys, x \leftarrow xs]
\]

\[
> \text{cartesian'} [1\ldots3] \ "abc"
\]
\[
[(1,'a'), (2,'a'), (3,'a'),
(1,'b'), (2,'b'), (3,'b'),
(1,'c'), (2,'c'), (3,'c')]
\]
Finding index of elements in a list

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

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

\[> \, \text{elemIndices} \, [3,4,2,1,4,5] \, 4\]
\[[2,5]\]
\[ pythags \, n = \left[ (x, y, z) \right. \]
\[ | \, z \leftarrow [1..n], \]
\[ | \, x \leftarrow [1..z], \]
\[ | \, y \leftarrow [x..z], \]
\[ \left. | \, x \times x + y \times y \equiv z \times z \right] \]

\[ > pythags \, 15 \]
\[ \left[ (3, 4, 5), (6, 8, 10), (5, 12, 13), (9, 12, 15) \right] \]

```
start

|___________________________...|
|                           |
|                           |
|                           |
|                           |
|                           |
|                           |
|                           |
|                           |
|                           |
|                           |
|                           |
|                           |
|                           |
|                           |
|                           |
|                           |
|                           |
|                           |
|                           |
|                           |
|                           |
|                           |
|                           |
|                           |
|                           |
|                           |
|                           |
|                           |
|                           |
|                           |
|                           |
|                           |
|                           |
|                           |
|                           |
|                           |
|                           |
|                           |
|                           |
|                           |
|                           |
|                           |
|                           |
|                           |
```
Prelude functions
- here implemented using list comprehensions

- \( \text{zipWith } f \ [x_1, x_2, \ldots, x_n] \ [y_1, y_2, \ldots, y_n] = [f(x_1, y_1), f(x_2, y_2), \ldots, f(x_n, y_n)] \)
  Example: \( \text{zipWith } (+) \ [2, 1, 3] \ [3, 1, 2] = [5, 2, 5] \)

- \( \text{concat } [x_1, \ldots, x_n] = [x_1, \ldots, x_n] \)
  Example: \( \text{concat } [[1], [1, 2], [1, 2, 3]] = [1, 1, 2, 1, 2, 3] \)

- \( \text{map } f \ [x_1, x_2, \ldots, x_n] = [f(x_1), f(x_2), \ldots, f(x_n)] \)
  Example: \( \text{map } (\ast 3) \ [1, 2, 3, 4] = [3, 6, 9, 12] \)

- \( \text{filter } p \ [x_1, x_2, \ldots, x_n] = [x_1, x_2, \ldots, x_n] \)
  Example: \( \text{filter even } [6, 2, 7, 5, 2] = [6, 2, 2] \)
Checking if a list is sorted

\[
\text{sorted } xs = \text{and } [x \leq y \mid (x, y) \leftarrow \text{zip } xs \ (\text{tail } xs)]
\]

\[
\text{sorted } [2, 3, 1] \equiv \text{and } [\text{True}, \text{False}]
\]
\[
\equiv \text{False}
\]
Pascal’s triangle

\[
\begin{array}{cccc}
\binom{0}{0} & 1 \\
\binom{1}{0} & (\binom{1}{1}) & 1 & 1 \\
\binom{2}{0} & (\binom{2}{1}) & (\binom{2}{2}) & 1 & 2 & 1 \\
\binom{3}{0} & (\binom{3}{1}) & (\binom{3}{2}) & (\binom{3}{3}) & 1 & 3 & 3 & 1 \\
\end{array}
\]

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

\[(a + b)^3 = \binom{3}{0}b^3 + \binom{3}{1}ab^2 + \binom{3}{2} + a^2b + \binom{3}{3}a^3\]

\[= b^3 + 3ab^2 + 3ba^2 + a^3\]
Pascal’s triangle

\[
\begin{array}{c}
\binom{0}{0} = 1 \\
\binom{1}{0} \binom{1}{1} = 1 1 \\
\binom{2}{0} \binom{2}{1} \binom{2}{2} = 1 2 1 \\
\binom{3}{0} \binom{3}{1} \binom{3}{2} \binom{3}{3} = 1 3 3 1 \\
\end{array}
\]

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

\[
pascal [1] = [1, 1] \\
pascal [1, 1] = [1, 2, 1] \\
pascal [1, 2, 1] = [1, 3, 3, 1] \\
pascal [1, 3, 3, 1] = [1, 4, 6, 4, 1] \\
pascal [1, 4, 6, 4, 1] = [1, 5, 10, 10, 5, 1]
\]
A prime number \( p \) is a number where its only divisors are 1 and \( p \).

\[
\text{divisors } n = [x \mid x \leftarrow [1 \ldots n], n \mod x \equiv 0]
\]

\[
\text{prime } n = \text{divisors } n \equiv [1, n]
\]

\[
\text{primes } n = [x \mid x \leftarrow [2 \ldots n], \text{prime } x]
\]
Caesar cipher

```haskell
import Data.Char (ord, chr, isLower)

char2int :: Char → Int
char2int c = ord c - ord 'a'  -- a=0, b=1 ...

int2char :: Int → Char
int2char n = chr (ord 'a' + n)  -- 0=a, 1=b ...

shift n c | isLower c = int2char ((char2int c + n) `mod` 26)
           | otherwise = c

encode n xs = [shift n x | x ← xs]
decode n xs = [shift (−n) x | x ← xs]

encode 3 "haskell er fantastisk"
≡ "kdvnhoo hu idqwdvwlvn"
```
Generating bitstrings

bitstrings 0 = [[]]
bitstrings n = [b : bs | b ← [0, 1], bs ← bitstrings (n − 1)]

bitstrings 0 ≡ []
bitstrings 1 ≡ [[0, 1]]
bitstrings 2 ≡ [[0, 0], [0, 1], [1, 0], [1, 1]]
bitstrings 3 ≡ [[0, 0, 0], [0, 0, 1], [0, 1, 0], [0, 1, 1],
                 [1, 0, 0], [1, 0, 1], [1, 1, 0], [1, 1, 1]]
Finding the transpose of a matrix

\[\text{transpose} :: [[a]] \to [[a]]\]
\[\text{transpose } [] = []\]
\[\text{transpose } ([] : xss) = \text{transpose } xss\]
\[\text{transpose } xss \quad = \quad [x \mid (x: _) \leftarrow xss]\]
\[\quad : \text{transpose } [xs \mid (_: xs) \leftarrow xss]\]

\[A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}\]
\[A^T = \begin{pmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{pmatrix}\]
Finding the transpose of a matrix

\[ \text{transpose} :: [[a]] \rightarrow [[a]] \]

\[ \text{transpose} [] = [] \]

\[ \text{transpose} ([]:xss) = \text{transpose} xss \]

\[ \text{transpose} xss = [x | (x: _) \leftarrow xss] : \text{transpose} [xs | (_: xs) \leftarrow xss] \]

\[ \text{transpose} [[1, 2, 3], [4, 5, 6]] \]
\[ \equiv [1, 4] : \text{transpose} [[2, 3], [5, 6]] \]
\[ \equiv [1, 4] : [2, 5] : \text{transpose} [[3], [6]] \]
\[ \equiv [1, 4] : [2, 5] : [3, 6] : \text{transpose} [[]], [[]] \]
\[ \equiv [1, 4] : [2, 5] : [3, 6] : \text{transpose} [[]] \]
\[ \equiv [[1, 4], [2, 5], [3, 6]] \]
Generating permutations

permutations [] = [[]]
permutations (x:xs) = [ys' ++ x:ys'' |
  ys ← permutations xs,
  i ← [0..length ys],
  let (ys',ys'') = splitAt i ys]

> 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 $8 \times 8$ 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.
validExtensions n qs = [q : qs | q ← [1..n] \ qs, q ‘notDiag‘ qs]

where

q ‘notDiag‘ qs = and [ abs (q – qi) ≠ i | (qi, i) ← qs ‘zip‘ [1..n]]

queens' n 0 = [[]]
queens' n i = [ qs’ | qs ← queens' n (i – 1),
                     qs’ ← 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]...]