List comprehension

Like notation in Set Theory:

- $\{ x^2 \mid x \in \{1,\ldots,10\},\ even(x) \}$
- `[ x^2 | x <- [1..10] , x `mod` 2 == 0 ]`
- `[ (x,y) | x <- [1..6] , y <- [1..6] , y > x ]`
- `[ x+y | (x,y) <- zip [1..6] [1..6] ]`

Permutations

```
perms [] = [ [] ]
perms xs = [ x:p | x <- xs , p <- perms (xs\\[x]) ]
```

Higher order functions

List !! index

Gives the lement at index place starting at 0

- `map f xs`

```
map (^2) [2,3,4,5]          ⇒ [4,9,16,25]
map (2^) [2,3,4,5]          ⇒ [4,8,16,32]
map ord "apple"             ⇒ [97,112,112,108,101]
```

```
map :: (a->b) -> [a] -> [b]
```

- `filter f xs`

```
filter (<4) [2,3,4,5]       ⇒ [2,3]
filter (>3) [2,3,4,5]       ⇒ [4,5]
```

```
filter :: (a->Bool) -> [a] -> [a]
```

- `zipWith f xs ys`

```
zipWith (+) [2,3,4,5] [2,3,4,5]   ⇒ [4,6,8,10]
zipWith (*) [2,3,4,5] [2,3,4,5]   ⇒ [4,9,16,25]
```

```
zipWith :: (a->b->c) -> [a] -> [b] -> [c]
```

Matrix transpose

```
transp ([]:xss) = []
transp  xss     = map head xss : transp (map tail xss)
```

```
transp' []      = repeat []
transp' (r:rs)  = zipWith (:)  r rs'
                  where
                   rs' = transp' rs
```

Some standard Haskell operators and functions

Remark. The list below presents the standard Haskell types, *without* the type `Number` as used in this course.

`negate, abs,`
`signum` :: `Num a => a -> a`
`+, -, *` :: `Num a => a -> a -> a`
`div, mod` :: `Integral a => a -> a -> a`
`/` :: `Fractional a => a -> a -> a`
`^` :: `(Num a, Integral b) => a -> b -> a`
`abs, exp, log,`
`sqrt, sin,`
`cos` :: `Floating a => a -> a`
various arithmetical operations and functions

`min, max` :: `Ord a => a -> a -> a`
gives the minimum, maximum of two arguments

`not` :: `Bool -> Bool`
`&&, ||` :: `Bool -> Bool -> Bool`
boolean operations negation, conjunction, disjunction

`isLower,`
`isUpper` :: `Char -> Bool`
says whether a letter is lower-case or upper-case

`isAlpha` :: `Char -> Bool`
says whether a character is a letter

`isDigit` :: `Char -> Bool`
says whether a character is a digit

`isAlphaNum` :: `Char -> Bool`
says whether a character is a letter or a digit

`ord` :: `Char -> Int`
converts a character to its Unicode number

`chr` :: `Int -> Char`
converts a Unicode number to the corresponding character

`show` :: `Show a => a -> String`
Converts a showable datatype (e.g. Int, Float) to a String

`read` :: `Read a => String -> a`
Converts a String to readable datatype (e.g. Int, Float)

`toLower,`
`toUpper` :: `Char -> Char`
converts a letter to lower-case, upper-case

`==, /=` :: `Eq a => a -> a -> Bool`
`>, >=,`
`<, <=` :: `Ord a => a -> a -> Bool`
various comparison operations

`even, odd` :: `Integral a => a -> Bool`
says whether a (integral) number is even or odd

`:` :: `a -> [a] -> [a]`
adds element to the front end of a list (*cons*)

`length` :: `[a] -> Int`
length of a list

`!!` :: `[a] -> Int -> a`
list indexing

`++, \\` :: `[a] -> [a] -> [a]`
list concatenation, list subtraction

`␣` :: `(a->b) -> a -> b`
function application

`$` :: `(a->b) -> a -> b`
function application operator, that has a low right-associative binding precedence

`.` :: `(b->c) -> (a->b) -> (a->c)`
function composition

`head, last` :: `[a] -> a`
`tail, init,`
`reverse` :: `[a] -> [a]`
`elem` :: `Eq a => a -> [a] -> Bool`
tests whether a list contains a given element

`concat` :: `[[a]] -> [a]`
concats a list of lists into one list

`sort` :: `Ord a => [a] -> [a]`
`merge` :: `Ord a => [a] -> [a] -> [a]`
merges two sorted list into a single, sorted whole

`sum` :: `Num a => [a] -> a`
`minimum,`
`maximum` :: `Ord a => [a] -> a`
`take, drop` :: `Int -> [a] -> [a]`
`takeWhile,`
`dropWhile` :: `(a->Bool) -> [a] -> [a]`
various functions on lists

`splitAt` :: `Int -> [a] -> ([a],[a])`
splits a list in the first n elements and the rest

`span` :: `(a -> Bool) -> [a] -> ([a],[a])`
splits a list at the first positon where property p is not satisfied

`insert` :: `Ord a => a -> [a] -> [a]`
inserts an element into an ordered list

`and, or` :: `[Bool] -> Bool`
yields the conjunction, disjunction of a list of booleans

`lines` :: `String -> [String]`
breaks a string at newlines (`'\n'`) into a list of strings

`unlines` :: `[String] -> String`
glues a list of strings with `'\n'`

`fst` :: `(a,b) -> a`
yields the first element of a pair

`snd` :: `(a,b) -> b`
yields the second element of a pair

`zip` :: `[a] -> [b] -> [(a,b)]`
turns two lists into a list of pairs

`unzip` :: `[(a,b)] -> ([a],[b])`
turns a list of pairs into a pair of lists

`zipWith` :: `(a->b->c) -> [a] -> [b] -> [c]`
zips two lists and applies a function to the corresponding elements

`map` :: `(a->b) -> [a] -> [b]`
applies a function to all elements in a list

`filter` :: `(a->Bool) -> [a] -> [a]`
selects those elements from a list which satisfy a property

`foldl` :: `(a->b->a) -> a -> [b] -> a`
"folds" a list with a function, starting with a given value. Works from left to right through the list

`foldr` :: `(a->b->b) -> b -> [a] -> b`
like `foldl`, but works from right to left

`foldl1,`
`foldr1` :: `(a->a->a) -> [a] -> a`
like `foldl`, `foldr`, with first, last element of the list as starting value. Error for empty list

`scanl` :: `(a->b->a) -> a -> [b] -> [a]`
like `foldl`, but yielding intermediate results too

`scanr` :: `(a->b->b) -> b -> [a] -> [b]`
like `foldr`, but yielding intermediate results too

`seq` :: `a -> b -> b`
partially evaluates first argument, and delivers the second

`error` :: `String -> a`
causes error with given string as error message

Comparison Functional – Imperative

Observations:

- Specification — Implementation
- Rewriting — State transformations
- Problem oriented — Computer oriented

Consequencies:

- Higher order functions, Polymorpism, Pattern matching, List comprehension
- Further abstraction mechanisms
- Manage efficiency

Fundamental:

- Provability of correctness
- Bounden to time and space
- Parallellisability
- Suitability for various architectures

- `'p' < 'q'`
- `(3,4) < (5,6)`
- `(3,'a') > (2,'p')`
- `[1,2,3] < [4,5,6,7]`
- `False < True`
- `:t (<)` `(<) :: Ord a => a -> a -> Bool`