Outline of Lecture 5

- Primitive recursion on lists (reminder, examples)
- Accumulating function parameters and tail recursion
- Generic functions, polymorphism, and function overloading
- List comprehensions

Primitive recursion on lists (reminder)

- The base case for lists is `[]`, while the recursive case handles a non-empty list `(x:xs)` by a recursive call to a simpler list `xs`
- General template (relying on pattern matching):

```
fun ::  [t]->t1
fun [] = ...
fun (x:xs) = ...  fun xs ...
```

Primitive recursion on lists (examples)

Simple list construction (from the given list):

```
doubleAll [] = []
doubleAll (x:xs) = 2*x : doubleAll xs
```

List filtering (retaining only even numbers):

```
selectEven [] = []
selectEven (x:xs)
  | isEven x =  x : selectEven xs
  | otherwise = selectEven xs
```

where

```
isEven ::  Integer -> Bool
isEven x = mod x 2 == 0
```

Primitive recursion on lists (examples)

List insertion sorting (top-down definition):

```
iSort ::  [Integer] -> [Integer]
iSort [] = []
iSort (x:xs) = ins x (iSort xs)
```

where

```
ins ::  Integer -> [Integer] -> [Integer]
ins x [] = [x]
ins x (y:ys)
  | x <= y = x:(y:ys)
  | otherwise = y:(ins x ys)
```

Helper functions with extra accumulating parameters

- Sometimes it is convenient or necessary to create a *helper* (local) function, which has an extra parameter to accumulate intermediate values that can be passed along with recursive calls
- Example: a function truncating a given integer list by retaining only those first elements that together do not exceed a given number

```
not_exceeding ::  Int -> [Int] -> [Int]
not_exceeding n xs = not_exceed' n xs 0
  where
    not_exceed' _ [] _ = []
    not_exceed' n (x:xs) k
      | (x+k)>n = []
      | otherwise = x :(not_exceed' n xs (x+k))
```

Tail recursion

- Simple recursive function

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

  is fully recursively unfolded into $1 + (1 + (... + 0)...)$ before evaluated

- For a bigger input data structures, it means creating large call stacks, which can lead to a drop in performance and/or stack overflow (especially in GHCI, since compiling a module by GHC and then importing it involves code optimisation)

- One way to improve on this is to rewrite a code by making it *tail recursive*

Tail recursion (cont.)

- A recursive function is **tail recursive** if the final result of the recursive call is the final result of the function itself. If the result of the recursive call must be further processed (say, by adding 1 to it, ...), it is not tail recursive.
- Using extra accumulating parameters (within a helper function) often allows transforming a function into tail recursive
- Example (making len tail recursive):

```
len_tr xs = len' xs 0
  where
    len' [] n = n
    len' (_:xs) n = len' xs (n+1)
```

  Intermediate result is calculated and passed as an extra parameter
- Tail recursion usually means that recursive code can be optimised into a traditional loop (*tail call optimisation*)

Generic functions (polymorphism)

- Polymorphism = 'has many shapes'
- A function is *polymorphic* if it 'has many types', i.e., it can be applied for arguments of many different types
- It is true for many list manipulating functions, which can be used independently of what type elements a list contains, such as `length :: [a] -> Int`, `(++) :: [a] -> [a] -> [a]`
- Here `a` is a type variable, standing for an arbitrary type
- Types like `[Bool] -> Int` or `[(Integer,[Char])] -> Int` are **instances** of `[a] -> Int`
- Different type variables in a function definition mean possibly different types; the same type variables ⇒ the same concrete types

Polymorphic functions on lists (from Prelude)

`:`	`a -> [a] -> [a]`	Adds an element to the list front
`++`	`[a] -> [a] -> [a]`	Joins two lists together
`!!`	`[a] -> Int -> [a]`	Returns n-th list element
`length`	`[a] -> Int`	Returns the list length
`head, last`	`[a] -> a`	Returns the first/last element
`tail, init`	`[a] -> [a]`	All but the first/last element
`replicate`	`Int -> a -> [a]`	Makes a list of n item copies
`take`	`Int -> [a] -> [a]`	Takes n elements from the front
`drop`	`Int -> [a] -> [a]`	Drops n elements from the front
`reverse`	`[a] -> [a]`	Reverses the element order
`zip`	`[a] -> [b] -> [(a,b)]`	Makes a list of pairs from a pair of lists
`unzip`	`[(a,b)] -> ([a],[b])`	Makes pair of lists from a list of pairs

Polymorphism and overloading

- Polymorphism and overloading – two mechanisms by which the same function name can be used with different types
- A polymorphic function: the same function definition, which can be instantiated and applied for different concrete types

```
fst :: (a,b) -> a
fst (x,_) = x
```

Defined for any types `a` and `b`

Polymorphism and overloading (cont.)

- An overloaded function: different function definitions for different types but with the same function name
- Example: the overloaded operator for equality comparison (==) can have very different definitions for different types

```
(==) :: Eq a => Eq b => (a,b) -> (a,b) -> Bool
(==) (x1,y1) (x2,y2)  =  (x1==x2) && (y1==y2)
```

Equality on pairs is defined using equality defined for the corresponding element types

List comprehensions

- One of the distinctive features of a functional language is the list comprehension notation
- In a list comprehension, we define a list in terms of the elements of another list
- From the source list we generate elements which we test (filter) and transform to form elements of the resulting list
- General syntax:

```
[res_expression | source_element <- source_list, guards]
```

Intuition: to create a new list (consisting of `res_expression`), using the elements `source_element` from `source_list`, such that they satisfy the conditions from `guards`

List comprehensions (examples)

Suppose that `input_list == [2,4,15]`

- `[2*n | n <- input_list] == [4,8,30]`
- `[isEven n | n <- input_list] == [True,True,False]`
- `[n*n | n <- input_list, isEven n, n>3] == [16]`

Suppose that `input_list2 == [(2,3),(2,1),(7,8)]`

- `[m+n | (m,n) <- input_list2] == [5,3,15]`
- `[m*m | (m,n) <- input_list2, m<n] == [4,49]`

List comprehensions (examples)

```
digits ::  String -> String
digits st = [ch | ch <- st, isDigit ch]
```

where `isDigit ::  Char -> Bool` (from the module `Data.Char`) returns `True` only for digits characters

```
allEven, allOdd ::  [Integer] -> Bool
allEven xs = (xs == [x | x <- xs, isEven x])
allOdd xs = ([] == [x | x <- xs, isEven x])
```

An example of quick filtering out the list

List comprehensions (cont.)

- A list comprehension expression can have more than one source set
- In that case, all possible combinations of values from all source lists are used to generate the result
- Example:

```
pairs = [(x, y) | x <- [1, 2, 3], y <- "ab"]
```

  contains all six combinations
  `[(1,'a'),(1,'b'),(2,'a'),(2,'b'),(3,'a'),(3,'b')]`
- Another example:

```
powers = [x^y | x <- [1..10], y <- [2, 3], x^y < 200]
```