Outline of Lecture 6

- Accumulating function parameters and tail recursion
- Generic functions, polymorphism, and function overloading
- More list examples (text processing)
- General recursion on lists
- Let and case expressions

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 $1 + (1 + (\ldots + 0)\ldots)$ 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 $\Rightarrow$ the same concrete types

Polymorphic functions on lists (from Prelude)

`:`	`a -> [a] -> [a]`	Adds an element to the list front
`elem`	`a -> [a] -> Bool`	An element belongs to the list?
`++`	`[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 examples (text processing)

The goal: split a string into a list of words (smaller strings). Whitespaces and punctuation should not be taken into account.

Preliminaries:

```
whitespaces = ['\n', '\t', ' ']
punctuation = ['.', ',', ';', '-', ':']

spaces = whitespaces ++ punctuation
```

A preparatory (helper) function – returning the first word:

```
getWord ::  String -> String
getWord [] = []
getWord (x:xs)
  | elem x spaces = []
  | otherwise = x :  getWord xs
```

List examples (text processing, cont.)

A preparatory function – returning a string without the first word:

```
dropWord ::  String -> String
dropWord [] = []
dropWord (x:xs)
  | elem x spaces = (x:xs)
  | otherwise = dropWord xs
```

Both functions (`getWord` and `dropWord`) work incorrectly for leading spaces $\Rightarrow$ the leading spaces must be removed first:

```
dropSpaces ::  String -> String
dropSpaces [] = []
dropSpaces (x:xs)
  | elem x spaces = dropSpaces xs
  | otherwise = (x:xs)
```

List examples (text processing, cont.)

The first version of a word splitting function:

```
splitWords ::  String -> [String]
splitWords [] = []
splitWords st = if new_st == "" then []
  else
     (getWord new_st) : splitWords(dropWord new_st)
  where
     new_st = dropSpaces st
```

Can we simplify this function by relying on a new function that returns both the first word and the remainder of the string, after removing the leading spaces first?

List examples (text processing, cont.)

A preparatory function – returning a pair of the first word and the remainder of the string:

```
splitFirstWord ::  String -> (String,String)
splitFirstWord st = (firstWord,rem_st)
  where
     new_st = dropSpaces st
     firstWord = getWord new_st
     rem_st = drop (length firstWord) new_st
```

Note how local definitions allow us to code sequential composition of bindings/assignments (relying on the previous ones) in Haskell

List examples (text processing, cont.)

The second version of a word splitting function:

```
splitWords2 ::  String -> [String]
splitWords2 [] = []
splitWords2 st = first : splitWords2 rest
  where
     (first,rest) = splitFirstWord st
```

Note the use of pattern matching in "multiple declaration" (first,rest) = ...
This works for any declarations and data constructors, e.g.,
[x,y,z] = "abc" assigns x, y, and z the corresponding letters

Also note that both `splitWord` and `splitWord2` does not follow the technique of primitive recursion on lists, since the recursive case is not defined on a list tail. Instead, a smaller list is used in recursive call(s)

General recursion on lists

- A recursive definition of a function does not need to always use a recursive call on the list tail (as prescribed by the primitive recursion pattern)
- Any recursive call to the value on a simpler (smaller) list will be legitimate and will lead to function termination
- A general question: **In defining `f xs` (where `xs` is non-empty), which values of `ys` that is a sublist of `xs` would help us to work out the answer?**
- Many patterns of general recursion over lists: filtering a list before a recursive call, partitioning a list into several and recursively handling those partitions, defining a recursion over multiple list arguments, etc.

General recursion on lists (examples)

Filtering a list before a recursive call. Example – a function calculating how many times numbers occur in a list:

```
nOccurs ::  [Integer] -> [(Integer,Int)]
nOccurs [] = []
nOccurs (x:xs) = (x, length onlyX + 1) : (nOccurs withoutX)
  where
     onlyX = [xx | xx <- xs, xx == x]
     withoutX = [xx | xx <- xs, xx /= x]
```

General recursion on lists (examples, cont.)

Partitioning a list before recursive call(s). Example – list sorting (qsort):

```
qsort ::  [Integer] -> [Integer]
qsort [] = []
qsort (x:xs) =
  qsort [y | y<-xs, y <= x] ++ [x] ++
  qsort [y | y<-xs, y > x]
```

Recursion over several lists. Example – zipping two lists together:

```
zip ::  [a] -> [b] -> [(a,b)]
zip [] _ = []
zip _ [] = []
zip (x:xs) (y:ys) = (x,y) : zip xs ys
```

`let` expressions

- A variation of local definitions
- Contrary to `where` definitions, `let` expressions can be used within almost any Haskell expression
- Wrapping the function result with a local definition block:

```
mylength2 xs =
  let
    length' [] n = n
    length' (_:xs) n = length' xs (n+1)
  in
    length' xs 0
```

`let` expressions (cont.)

Simple pattern matching with a `let` expression:

```
ghci > (let (a,b,c) = (1,2,3) in a+b+c) * 100
600
```

A `let` expression within list comprehensions:

```
calculateBMIs ::  [(Float,Float)] -> [Float]
calculateBMIs xs = [bmi | (w,h) <- xs, let bmi = w / h^2]
```

Calculating the BMI index for given weight and height pairs

Note a slightly different syntax (no `in` keyword afterwards)! The local definition scope is the whole list comprehension block `[...]`

A `case` expression

- So far, pattern matching was performed only over function arguments or declaration variables
- The `case` construction allows us to define a result by pattern matching over an arbitrary Haskell expression
- The general form of a `case` expression:

```
case e of
  p1 -> e1
  p2 -> e2
  ...
  pn -> en
```

Here `e` is an input expression, `p1,p2, ... pn` are patterns, and `e1,e2, ... en` are the resulting expressions

The case expression (cont.)

Example: finding the first digit for a given string

```
firstDigit ::  String -> Char
firstDigit st =
  case (digits st) of
      [] -> '\0'
      (x:xs) -> x
```

where

```
digits :: String -> String
digits st = [ch | ch <- st, elem ch ['0'..'9']]
```