Outline of Lecture 6

- More list examples (text processing)
- General recursion on lists
- Let and case expressions
- User-defined datatypes: enumerated and product types
- User-defined datatypes: general structure and principles

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 =
  (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
- Can be several definition in one line, separated by ;

```
ghci > (let a = 12; b = 33 in a^2 + 3*b + 10) - 200
53
```

Simple pattern matching with a `let` expression:

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

`let` expressions (cont.)

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

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']]
```

User defined datatypes: Enumerated types

- Haskell comes with comprehensive set of basic types and different ways of building complex types from simpler ones (like tuples or lists)
- The user can also to define his/her own datatypes, directly reflecting the problem domain
- The Haskell keyword: `data`
- Enumerated types are simplest examples of such user-defined datatypes

User defined datatypes: Enumerated types

- An enumerated type is defined by simply listing all its members one by one:

```
data TypeName = member1 | member2 ...  | memberN
  deriving (...)
```

- Examples:

```
data Bool = False | True
  deriving (Show, Eq, Ord)
data Temp = Cold | Lukewarm | Hot
  deriving (Show, Eq, Ord)
```

- What is `deriving` for? Allows to inherit some type features like the ability to be printed, compared to be equal, ordered by the way they are defined (from smaller to greater). Will be more explained later

User defined datatypes: Enumerated types (cont.)

Example: the game of rock, paper, scissors. The datatype definition:

```
data Move = Rock | Paper | Scissors
  deriving (Show, Eq)
```

The rules: Rock defeats Scissors, Paper defeats Rock, and Scissors defeat Paper:

```
beat ::  Move -> Move
beat Rock = Paper
beat Paper = Scissors
beat Scissors = Rock
```

Note that the datatype elements can be directly used as literals for simple pattern matching

User defined datatypes: Product types

- A datatype with a number components or fields
- A constructor function is used to combine these fields together into one object

```
data People = Person Name Age
  deriving (Show, Eq)
```

where

```
type Name = String
type Age = Int
```

To construct an object of the type `People`, we use the constructor `Person` with two values of the types `String` and `Int` respectively.

Can be several constructors for the same datatype (separated by `|`)!

User defined datatypes: Product types (cont.)

Example: geometrical shapes

```
data Shape = Circle Float | Rectangle Float Float
  deriving (Show, Ord, Eq)
```

Two constructors to define the datatype!

Calculating the area and checking the roundness property:

```
area ::  Shape -> Float
area (Circle r) = pi*r*r
area (Rectangle h w) = h*w

isRound ::  Shape -> Bool
isRound (Circle _) = True
isRound (Rectangle _ _) = False
```

The constructors and their values are used for pattern matching

Product types vs tuples

- Alternatively, the person type can be defined as
  `type Person = (Name,Age)`
- The advantages of a product datatype:
  - Each object carries an explicit label (the constructor name) of the element purpose
  - It is not possible to treat an arbitrary pair of (String,Int) as a person
  - The datatype name will appear in all typing error messages (in case of `type Person = (Name,Age)`, the type name will be expanded)
- The advantages of a tuple type:
  - The elements are more compact and so definitions are shorter
  - Many polymorphic functions over pairs and tuples in general can be reused

User defined datatypes: General principles

- The general form of datatypes:

$$
\begin{aligned}
&\texttt{data TypeName =}\\
&\qquad \texttt{Con}_1\ \texttt{t}_{11}\ \ldots\ \texttt{t}_{1k_1}\ |\\
&\qquad \texttt{Con}_2\ \texttt{t}_{21}\ \ldots\ \texttt{t}_{2k_2}\ |\\
&\qquad \ldots\\
&\qquad \texttt{Con}_n\ \texttt{t}_{n1}\ \ldots\ \texttt{t}_{nk_n}
\end{aligned}
$$

  where $\texttt{Con}_i$ are constructors and $\texttt{t}_{ij}$ are types names

- Each constructor (function) $\texttt{Con}_i$ takes arguments of the types $\texttt{t}_{i1}\ \ldots\ \texttt{t}_{ik_i}$ and returns a result of the type TypeName
- The element names in enumerated types – nullary constructors, i.e., constructors without arguments ($k_i = 0$)
- As we will see later, the datatypes can be recursive or polymorphic

User defined datatypes: Conjoining different types

- Datatypes allow us to use objects of different types even when a single type is required (like in lists)
- Different types are "separated" or distinguished by different constructors, while on the outside the elements are of the same datatype

```
data MyDatatype = Name String | Number Float
  deriving (Show, Eq)

mylist ::  [MyDatatype]
mylist = [Name "Linas", Number 666.7]
```