Outline of Lecture 7

- User-defined datatypes: enumerated and product types
- User-defined datatypes: general structure and principles
- Patterns of computation
- Higher order functions (map, filter, fold)

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

Pattern matching: general principles (reminder)

A pattern can be one of a number of things:

- A **literal value** such as `24`, `'c'`, `True`, `...` An argument matches this pattern if it is equal to the value
- A **variable** such as `x`, `z`, `n`, `longVarName`, `...` Any argument value will match this and the variable "gets assigned" the value within the function definition
- A **wildcard** `'_'`. Any argument value will match this
- In general, a **constructor** applied to a number of patterns (`C` p_1 p_2 ... p_n). To match this, the argument must be constructed by `C` to arguments v_1, v_2, ... , v_n and each v_k must match p_k

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]
```

Patterns of computation

One mechanism to generalise computations in Haskell – to define polymorphic functions that work for all or many different concrete types

Another one is to implement reusable patterns of computations.
For instance, examples of such patterns for working with lists would be:

- Transform every element of a list in some way;
- Select/ filter/ break up a list into smaller parts using some criteria;
- Combine the elements of a list using some operator;
- A mixture of the above.

This can be achieved via creating higher-order functions, i.e., functions that take other functions (implementing a particular kind of computations) as parameters

Higher-order functions: Mapping

Mapping (transforming every element of a list in some way)

- The `map` function:

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

- The first parameter – a transformation function (from arbitrary type `a` to a possibly different type `b`)
- The second parameter – the list to be transformed
- General definition of `map`:

```
map f xs = [f x | x <- xs]
```

- Convenient to have such a definition in the explicit function form to possibly combine it with other functions

Mapping examples

- Doubling list elements:

```
doubleAll ::  [Integer] -> [Integer]
doubleAll xs = map double xs

double ::  Integer -> Integer
double x = 2*x
```

- Converting characters into their codes:

```
convertChrs ::  [Char] -> [Int]
convertChrs xs = map fromEnum xs
```

Mapping examples (combining `zip` and `map`)

- Polymorphic function `zip`:

```
zip ::  [a] -> [b] -> [(a,b)]
```

- What if we want to do something different (other to just pairing them up) to two corresponding list elements:

```
zipWith' ::  (a->b->c) -> [a] -> [b] -> [c]
zipWith' f (x:xs) (y:ys) = f x y :  zipWith' f xs ys
zipWith' _ _ _ = []
```

- A version relying on `map`:

```
zipWith'' ::  ((a,b)->c) -> [a] -> [b] -> [c]
zipWith'' f xs ys = map f (zip xs ys)
```

Higher-order functions: Filtering

Filtering a list according to the given property

- How a property (e.g., of being a digit, an even number, etc.) is expressed in Haskell?
- In general, a property is given as a function of the type `t -> Bool`, where `t` is a concrete type we are dealing with, e.g., `isDigit :: Char -> Bool`, `isEven :: Integer -> Bool`
- The `filter` function:

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

  where the first argument is a property function and the second argument is the list to be filtered

Higher-order functions: Filtering (cont.)

- General definition of `filter`:

```
filter p xs = [x | x <- xs, p x]
```

- Examples:

```
digits ::  String -> String
digits xs = filter isDigit xs

evenNumbers ::  [Integer] -> [Integer]
evenNumbers xs = filter isEven xs

sortedLists ::  [[Integer]] -> [[Integer]]
sortedLists xs = filter isSorted xs

isSorted ::  [Integer] -> Bool
isSorted xs = (xs == iSort xs)
```

Variations of filtering

- Skipping or continuing until/while some property holds

```
getUntil ::  (a->Bool) -> [a] -> [a]
getUntil p [] = []
getUntil p (x:xs)
     | p x = []
     | otherwise = x :  getUntil p xs

takeWhile ::  (a->Bool) -> [a] -> [a]
takeWhile p [] = []
takeWhile p (x:xs)
     | p x = x :  takeWhile p xs
     | otherwise = []
```

- `takeWhile (dropWhile)` – predefined in Haskell

Higher-order functions: Folding

- Combining (*folding*) elements according to the given function
- The folding function is applied for each list element, combining its value with the previously accumulated result. The calculation is performed either from the list beginning or the list end
- Many primitive recursive functions over lists can be defined via folding
- Several varieties of higher-order functions based on folding
- Folding (to the right) of a non-empty list:

```
foldr1 ::  (a->a->a) -> [a] -> a
foldr1 f [x] = x
foldr1 f (x:xs) = f x (foldr1 f xs)
```

Higher-order functions: Folding (cont.)

- The first argument: a binary function over an arbitrary type a
- The second argument: the list to be combined (folded) together into one value
- What does it mean "folding to the right"?

```
foldr1 f [e1,e2, e3, ..., en]
    = e1 'f' (e2 'f' ( e3 'f' (...  'f' en) ...)))
```

Folding (by parentheses) happens on the right first

Higher-order functions: Folding (cont.)

Folding examples (with `foldr1`):

```
foldr1 (+) [3,98,1] = 102
foldr1 (||) [False,True,False] = True
foldr1 min [6] = 6
foldr1 (*) [1 ..  6] = 720
```

A drawback: `foldr1` is not defined for `[]`

Higher-order functions: Folding (cont.)

- (Slightly) generalised folding to the right:

```
foldr ::  (a->a->a) -> a -> [a] -> a
```

- The first argument: a binary function over an arbitrary type a
- The second argument: the starting (default) value of the type a
- The third argument: the list to be folded
- The definition of `foldr`:

```
foldr f s [] = s
foldr f s (x:xs) = f x (foldr f xs)
```

Higher-order functions: Folding (cont.)

Using `foldr`, we can define now some standard functions of Haskell, like concatenating a list of lists or calculating conjunction over a list of boolean values:

```
concat ::  [[a]] -> [a]
concat xs = foldr (++) [] xs

and ::  [Bool] -> Bool
and bs = foldr (&&) True bs
```

`foldr1` can be then defined in terms of `foldr`:

```
foldr1 f xs = foldr f (last xs) (init xs)
```

where `last` gives the last element of a list, and `init` removes that element