Outline of Lecture 3

- Examples of primitive recursion
- More general forms of recursion
- Local definitions and definition scopes in Haskell
- Basic Haskell types: floating numbers, characters and strings
- Tuples and lists; Pattern matching using tuples

Primitive recursion (reminder)

- Based on primitive induction principle on natural numbers
- General template:

```
fun n
    | n==0 = ...
    | n > 0 = ...  fun(n-1) ...
```

- The first branch defines the base case, stopping the recursion and returning the end result
- The second branch defines the recursive case (inductive case), describing the result in terms of a more simple case `n-1`
- Such template guarantees termination of recursion (for positive integer numbers)

Primitive recursion (cont.)

- Example (factorial):

```
fact ::  Integer -> Integer
fact n
    | n==0 = 1
    | n > 0 = fact(n-1)*n
```

- What happens for `fact (-1)`? The "non-exhaustive pattern" error.

```
fact n
    | n==0 = 1
    | n > 0 = fact(n-1)*n
    | otherwise = 0
```

- Alternatively (by defining own exceptions),
  `| otherwise = error "Only defined for natural numbers"`

Calculation and evaluation of recursive expressions

```
fact :: Integer -> Integer
fact n
     | n == 0    =  1
     | n > 0     =  fact(n - 1) * n
     | otherwise = error "Negative argument!"
```

Evaluation in a step-by-step manner:

$$
\begin{array}{ll}
 & \text{fact } 4 \\
\rightsquigarrow & \text{fact } 3 * 4 \\
\rightsquigarrow & (\text{fact } 2 * 3) * 4 \\
\rightsquigarrow & ((\text{fact } 1 * 2) * 3) * 4 \\
\rightsquigarrow & (((\text{fact } 0 * 1) * 2) * 3) * 4 \\
\rightsquigarrow & \text{rewriting the base case} \\
\rightsquigarrow & (((1 * 1) * 2) * 3) * 4 \\
\rightsquigarrow & ((1 * 2) * 3) * 4 \\
\rightsquigarrow & (2 * 3) * 4 \\
\rightsquigarrow & 6 * 4 \\
\rightsquigarrow & 24
\end{array}
$$

Examples of primitive recursion

Summing up factorials (a single parameter to build recursion on)

```
sumFacts :: Integer -> Integer
sumFacts n
    | n == 0  =  fac 0
    | n > 0   =  sumFacts(n - 1)  +  fac n
```

Summing up function applications (choosing a particular parameter to build recursion on)

```
sumFun :: (Integer -> Integer) -> Integer -> Integer
sumFun f n
    | n == 0  =  f 0
    | n > 0   =  sumFun f (n - 1)  +  f n
    | otherwise = error "Invalid argument!"
```

Examples of primitive recursion (cont.)

Summing up function applications for a range of arguments (combining several parameters to build recursion on)

```
sumFunRange :: (Integer -> Integer) -> Integer -> Integer -> Integer
sumFunRange f m n
    | (n - m) == 0  =  f n
    | (n - m) > 0   =  sumFunRange f m (n - 1) + f n
    | otherwise = error "Invalid range!"
```

Alternatively,

```
sumFunRange2 :: (Integer -> Integer) -> Integer -> Integer -> Integer
sumFunRange2 f m n
    | m == n  =  f n
    | n > m   =  sumFunRange2 f (m + 1) n + f m
    | otherwise = error "Invalid range!"
```

Examples of primitive recursion (cont.)

- Any parameter can be used to build the recursion on
- Parameters can be combined to construct the base and recursive cases, i.e., we can define an expression on their values. We have to ensure though that the recursive call makes the recursive case expression a step closer to the base one
- Special cases can be added to cover the invalid cases leading to non-exhaustive pattern errors

More general forms of recursion

- As long as the expression of the recursive case is simpler and eventually leads to the base case or base cases, different forms of recursion are possible
- Some of simple patterns – below

Fibonacci numbers (two base cases and two different expressions for the recursive case)

```
fib :: Integer -> Integer
fib n
     | n == 0  =  0
     | n == 1  =  1
     | n > 1   =  fib(n - 2) + fib(n - 1)
```

More general forms of recursion (cont.)

Integer division by recursion (another implementation of `div`)

```
divide :: Integer -> Integer -> Integer
divide m n
      | m < n  =  0
      | otwerwise  =  divide (m - n) n + 1
```

- Many possible values satisfying the base case
- divide 7 0 or divide 4 (-4) would lead to infinite looping and stack overflow $\Rightarrow$ separate base cases with error messages or special returned values are needed

Local definitions in Haskell

- Sometimes, we want to store an intermediate result to, e.g., avoid calculating the value over and over
- In (traditional) programming, we store an intermediate value in a separate variable
- In Haskell, within a definition we are working on, we can make a number of local definitions (bindings) under the `where` clause
- Both values and, if needed, (local) function definitions can be declared there and used within the whole body of the encompassing global definition

Local definitions in Haskell (cont.)

Example:

```
cylinderArea :: Float -> Float -> Float
cylinderArea r h = sideArea + 2*topArea
      where
         sideArea = 2 * pi * r * h
         topArea = pi * r^2
```

- The local definitions under the `where` clause must be aligned (similarly like the top definitions) to avoid Haskell confusion where a definition starts and stops
- **Jumping forward**: there are also more general local definitions (starting with `let` keywords) that can be used within practically any Haskell expression. Will be explained later!

Local definitions in Haskell (cont.)

Can be combined with guards:

```
ff :: Integer -> Integer -> Integer -> Integer
ff x y z
     | sqsum >= z = z
     | sqsub >= z = sqsub
     | otherwise = sqsum
     where
       sqsum = x*x + y*y
       sqsub = x*x - y*y
```

- Local definitions allow solving the problem in steps (e.g., imitate sequential composition)
- The local definitions under the `where` clause must be aligned (similarly like the top definitions) to avoid confusion

Definition scopes (revisited)

- A Haskell script (module) consists of a number definitions. The scope defines the program part a particular definition can be used
- All the top definitions has the whole module file as their scope

```
isOdd, isEven :: Int −> Bool
isOdd n
      | n ≤ 0     = False
      | otherwise = isEven (n-1)

isEven n
      | n < 0     = False
      | n == 0    = True
      | otherwise = isOdd (n-1)
```

An example of mutually dependent definitions

Definition scopes (cont.)

- Local definition (under `where`) are only visible within the definition they occur
- Local definitions can be used in guards and other local definitions
- Function parameters are visible only within the function body
- If there are several visible variables with the same name defined, the *most local* one is used

Definition scopes (cont.)

```
maxsq :: Int -> Int -> Int
maxsq x y
      | sqx ≥ sqy  =  sqx
      | otherwise = sqy
      where
         sqx = sq x
         sqy = sq y
         sq :: Int -> Int
         sq x = x*x
```

- The local definition `sq` is used before it is defined (the order does not matter, provided it is within the same `where` clause)
- The local function parameter `x` in the `sq` definition overrides the global parameter `x` of `maxsq`

Basic Haskell types: floating point numbers

- Standard Haskell type for such numbers – `Float`. Other (not covered in this course types) – `Double`, `Rational`
- Literals: 0.3478, -23.12, 231.56e9, ...
- Operations: many overloaded operations from integers (like `+`, `-`, `>=`, `...`)
- Also `**` (float exponentiation), `floor`, `round`, `log`, `exp`, `fromIntegral` (converting any integer to float), etc.

Basic Haskell types: characters and strings

- The built-in type for characters – `Char`
- Literals: 'a', 'f', 'z', '\t' (tab), '\n' (new line) ...
- Can be checked for equality, compared, used in patter matching, ...
- Some pre-defined functions:
  `fromEnum :: Char -> Int` (returns the character numerical code),
  `toEnum :: Int -> Char` (returns a character for the given code),
  `toUpper :: Char -> Char` (makes the upper case character),
  `isDigit :: Char -> Bool` (checks whether a character represents a digit), etc.

Characters and strings (cont.)

- Strings – a sequences (lists) of characters, syntactically represented as "...", e.g., "qwerty", "Hurray!\n"
- The `String` type is just a shorthand for `[Char]`, which denotes a list of characters
- 'a' :: `Char` is not comparable with "a" :: `String` (a string which happens to be a list consisting of a single character 'a')
- The overloaded functions `show` and `read` convert a value to `String` and vice versa
- More about lists – below

Simple aggregate/collection types in Haskell: tuples and lists

- Both tuples and lists are built up by combining a number of data elements into a single object
- In a tuple (denoted `(v1,v2,...,vn)`), we combine a fixed number of values of fixed, possibly different types into a single object
- In a list (denoted `[v1,v2,...,vn]`), we combine an arbitrary number of values of the same type into a single object

Tuples and lists (cont.)

Example: a shopping basket in a supermarket

- A single item in the shopping basket consists of its name (of the type `String`) and its price in cents (of the type `Int`) ⇒ we can represent as a tuple of the type `(String,Int)`
- Item examples: `("Salt:1kg",119)`, `("Plain crisps",35)`
- A shopping basket – a collection of items (represented in the same way) ⇒ we can represent as a list of the type `[(String,Int)]`
- `shopping_basket = [("Salt:1kg",118), ("Plain crisps",40), ("Gin:1lt",1299)]`

Type checking and naming in Haskell

- A strict type system allowing the types to be inferred and checked at the compilation time
- While tuples can be defined over the fixed number of diverse types, lists can contain a varying number elements of the same type
- Different lists however can include elements of different types
- For readability, we can give our own names to types in Haskell (using the keyword `type`), e.g.,

```
type ShopItem = (String,Int)
type ShopBasket = [ShopItem]
type String = [Char]                defined in Prelude
```

Tuples

Creating a tuple (using the `(...)` constructor):

```
minAndMax :: Integer -> Integer -> (Integer,Integer)
minAndMax x y
     | x ≥ y  =  (y, x)
     | otherwise = (x,y)
```

Pattern matching a tuple as an argument:

```
addPair :: (Integer,Integer) -> Integer
addPair (x,y) = x+y

name :: ShopItem -> String
price :: ShopItem -> Int

name (n,p) = n
price (n,p) = p
```

Tuples (cont.)

Pattern matching (with literals).

```
multPair :: (Integer,Integer) -> Integer
multPair (0,_) = 0
multPair (_,0) = 0
multPair (x,y) = x*y

name :: ShopItem -> String
price :: ShopItem -> Int

name (n,_) = n
price (_,p) = p
```

Special symbol (wildcard) _ is used instead of an arbitrary value. In other words, it matches any concrete value of the argument

Tuples (cont.)

In general, the type of a tuple is of the form $(\text{Type}_1, \text{Type}_2, ..., \text{Type}_n)$.
Each type can be any valid type, including a tuple type again.
In the latter case we have nested tuples, e.g.,

```
shift :: ((Integer,Integer),String) -> (Integer,(Integer,String))
shift ((x,y),s) = (x,(y,s))
```

which can be generalised to such a polymorphic function:

```
shift :: ((a,b),c) -> (a,(b,c))
shift ((x,y),s) = (x,(y,s))
```

Pattern matching on nested tuples:

```
ff ((False,_),_) = 0
ff ((_,x),y) = x *10 + y
```

One pair or two arguments?

It is important to distinguish between

```
addPair :: (Integer,Integer) -> Integer
addPair (x,y) = x+y
```

and

```
addTwo :: Integer -> Integer -> Integer
addTwo x y = x+y
```

In the first case, we have a single argument which is a pair of numbers. In the second case, we have two arguments, each of which is a number.

As we see later, the second case is more flexible, since we can apply arguments one by one (and not necessarily all of them)!