Outline of Lecture 4

- Tuples (reminder)
- Lists in Haskell
- Different list constructors (':', ranges, concatenation, strings)
- Pattern matching with lists
- Pattern matching: general principles
- Primitive recursion with lists

Simple aggregate/collection types in Haskell: tuples and lists (reminder)

- 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 (reminder)

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

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

Pattern matching a tuple as an argument:

```haskell
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 $(\mathrm{Type}_1, \mathrm{Type}_2, \ldots, \mathrm{Type}_n)$. 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))
```

Lists in Haskell

- A collection of items from a given type
- For every type `t`, there is a Haskell type `[t]` of lists of elements from `t`
- Examples:
  `[1,2,3,4,9,77] :: [Integer]`
  `[False] :: [Bool]`,
  `[fac,sumFacs] :: [Integer->Integer]`
  `[] :: [t]` empty list (element of any list type)
- The order in a list is significant, as is the number of times that an item appears

Lists in Haskell (cont.)

Some basic operations on lists (from Prelude):

- `':'` – adding an element to the beginning of a list
- `head` – extracting the first element (head) of a non-empty list
- `tail` – returning the list with its first element removed (also only for non-empty lists)
- `length` – returning the number of list elements
- `++` – concatenating (merging) two lists
- `null` – checking whether a list is empty
- `elem` – checking whether a given element belongs to a list
- ...

Constructing lists

- Explicitly listing its elements: `[2,17,999]`, `['c','d']`, `[True]`
- Adding an element to the beginning of a list e.g., `(-23.45):lst` for some `lst::[Float]`
- Using list concatenation operation ++
- Using ranges of the form `[n .. m]` (the default step by one):

  `[2 .. 7]` $\leadsto$ `[2,3,4,5,6,7]`
  `[3.1 .. 7]` $\leadsto$ `[3.1,4.1,5.1,6.1,7.1]`
  `['a' .. 'm']` $\leadsto$ `"abcdefgijklm"`
- Using ranges of the form `n,p .. m` (with the given step):

  `[13,11 .. 3]` $\leadsto$ `[13,11,9,7,5,3]`
  `['a','c' .. 'n']` $\leadsto$ `"acegikm"`

Constructing lists – Strings

- Strings as a special case of lists
  `type String = [Char]`
- `"valio!" == ['v','a','l','i',''o,'!']`
- The functions `show` and `read` convert to strings and vice versa.
  `read` typically requires the typing information to work, e.g.,
  `(read "5")::Integer`
  `(read "[True,False]")::[Bool]`
- All standard operations on lists apply to strings. More specialised functions can be found in a library (module) `Data.String`

List patterns

- Every list is either empty, `[]`, or is non-empty
- If a list is non-empty, it can be written in the form `x:xs`, where `x` is the first element and `xs` is the remainder (tail) of the list, for instance, `[4,2,12] == 4:[2,12]`
- Moreover, every list can be built from the empty list by repeatedly applying `':'`, e.g.,
  `[4,2,12] == 4:[2,12] == 4:(2:[12]) == 4:(2:(12:[]))`
- `':'` is the primary constructor for lists
- Two standard list patterns: `[]` and `(x:xs)`

Pattern matching on lists

Essentially, it is distinguishing between the empty and non-empty cases as well as using variables, literals or _ to match lists or list elements, for example,

```
hd :: [Integer] -> Integer
hd [] = error "Empty list!"
hd (x:_) = x

isEmpty :: [Float] -> Bool
isEmpty [] = True
isEmpty (_:_) = False
```

Nested patterns of arbitrary complexity (e.g, `(q:(p:xs))`) are also allowed.
Note that `[2,3]` will match `(q:(p:xs))`, but `[5]` will not

Pattern matching: general principles

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
- A **tuple pattern** $(\mathtt{p}_1, \mathtt{p}_2, ..., \mathtt{p}_\mathtt{n})$. To match this, an argument must of the form $(\mathtt{v}_1, \mathtt{v}_2, .., \mathtt{v}_\mathtt{n})$ and each $\mathtt{v}_\mathtt{k}$ must match $\mathtt{p}_\mathtt{k}$
- In general, a **constructor** applied to a number of patterns $(\mathtt{C}\ \mathtt{p}_1\ \mathtt{p}_2 \ldots \mathtt{p}_\mathtt{n})$. To match this, the argument must be constructed by `C` to arguments $\mathtt{v}_1,\ \mathtt{v}_2,\ \ldots, \mathtt{v}_\mathtt{n}$ and each $\mathtt{v}_\mathtt{k}$ must match $\mathtt{p}_\mathtt{k}$

Pattern matching: general principles (cont.)

- The **tuple pattern** can be seen a special case of the **constructor pattern** because `(,)` is the primary constructor for tuples
- Similarly, `[]` and `(:)` are primary constructors for lists
- Literals can be also considered as (nullary) constructors – constructors without parameters
- Later, we will see how to define new datatypes with our own constructors. Haskell will then automatically support pattern matching on the newly defined constructors

Pattern matching and guards

- Matching literal values: can be easily done with both pattern matching and guards
- More complex value comparisons (not just checking for equality or inequality): only by guards
- Matching against the argument structure (e.g., a tuple of three elements, a singleton list): only by pattern matching
- The good news: we can combine both of them (first pattern matching and then extra guards within each case)

Pattern matching and guards (cont.)

An example: summing elements of a list. Special treatment of the last list element: if it is 0, it is replaced by 100, if it is negative, it is ignored (i.e., replaced by 0)

```
ff ::  [Integer] -> Integer
ff [] = 0
ff [x] =
 | x==0 = 100
 | x<0 = 0
 | otherwise = x
ff (x:xs) = x + ff xs
```

Pattern matching (with lists and pairs)

Another example: conditional summing elements of a list of (number,bool) pairs. The value of the second pair element determines whether to add the first pair element or not.

```
condSum ::  [(Integer,Bool)] -> Integer
condSum [] = 0
condSum ((x,True):xs) = x + condSum xs
condSum ((_,False):xs) = condSum xs
```

Pattern matching (with lists and pairs), cont.

Pattern matching when binding a global or local identifier (i.e., when defining / introducing a new variable):

```
(x,y) = (10,(True,"abc"))    -- x = 10, y = (True,"abc")

mk_triple ::  a -> b -> c -> (a,b,c)
mk_triple x y z = (x,y,z)
(_,_,w) = mk_triple True "Hurray!" 999

(z:rest) = [1,2,3]            -- z = 1, rest = [2,3]
(first:_) = [4,6..24]         -- first = 4
(_:second:_) = [4,6..24]      -- second = 6
(e1:e2:other) = [10..20]      -- e1 = 10, e2 = 11,
                              -- other = [12..20]
```

The constituent tuple or list elements (from the given pattern) get "assigned" accordingly during pattern matching

Primitive recursion with lists

- The way lists are constructed by using the ':' constructor, starting from [], suggests how (primitive) recursive functions on lists can be written
- 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 with lists (cont.)

Examples:

```
mylength [] = 0
mylength (_:xs) = mylength xs + 1

myelem x [] = False
myelem x (y:ys) = (x==y) || (myelem x ys)

remove [] _ = []
remove (y:ys) x
  | x==y = remove ys x
  | otherwise = y:(remove ys x)
```