Outline of Lecture 5

- List comprehensions
- Primitive recursion on lists (reminder, examples)
- Accumulating function parameters and tail recursion
- Generic functions, polymorphism, and function overloading

List comprehensions

- One of the distinctive features of a functional language is the list comprehension notation
- In a list comprehension, we define a list in terms of the elements of another list
- From the source list we generate elements which we test (filter) and transform to form elements of the resulting list
- General syntax:

```
[res_expression | source_element <- source_list, guards]
```

Intuition: to create a new list (consisting of `res_expression`), using the elements `source_element` from `source_list`, such that they satisfy the conditions from `guards`

List comprehensions (cont.)

- Another (quite powerful) list constructor
- Inspired by the notion of mathematical set comprehension
  $\{e \mid e \in S \land P\ e\}$
  (a new set consisting of such elements e of the existing set S satisfying the property P)

List comprehensions (examples)

Suppose that `input_list == [2,4,15]`

- `[2*n | n <- input_list] == [4,8,30]`
- `[isEven n | n <- input_list] == [True,True,False]`
- `[n*n | n <- input_list, isEven n, n>3] == [16]`

Suppose that `input_list2 == [(2,3),(2,1),(7,8)]`

- `[m+n | (m,n) <- input_list2] == [5,3,15]`
- `[m*m | (m,n) <- input_list2, m<n] == [4,49]`

List comprehensions (examples)

```
digits ::  String -> String
digits st = [ch | ch <- st, isDigit ch]
```

where `isDigit ::  Char -> Bool` (from the module `Data.Char`) returns `True` only for digits characters

```
allEven, allOdd ::  [Integer] -> Bool
allEven xs = (xs == [x | x <- xs, isEven x]
allOdd xs = ([] == [x | x <- xs, isEven x]
```

An example of quick filtering out the list

List comprehensions (cont.)

- A list comprehension expression can have more than one source set
- In that case, all possible combinations of values from all source lists are used to generate the result
- Example:

```
pairs = [(x, y) | x <- [1, 2, 3], y <- "ab"]
```

  contains all six combinations
  `[(1,'a'),(1,'b'),(2,'a'),(2,'b'),(3,'a'),(3,'b')]`
- Another example:

```
powers = [x^y | x <- [1..10], y <- [2, 3], x^y < 200]
```

List comprehensions (cont.)

- From the evaluation order standpoint ...
- In general, a list comprehension is an expression of the form

$$[\mathtt{e} \mid \mathtt{q}_1, \mathtt{q}_2, \ldots, \mathtt{q}_\mathtt{k}]$$

  where each $\mathtt{q}_\mathtt{i}$ is either
  - a **generator** of the form `p <- lExp`, where `p` is a pattern and `lExp` is an expression of the list type
  - a **test**, `bExp`, which is a boolean expression
- Multiple generators allow to combine elements from two or more lists. What is the evaluation order?

List comprehensions (cont.)

- Example:

```
num_pairs ::  [a] -> [b] -> [(a,b)]
num_pairs xs ys = [(x,y) | x <-xs, y<-ys]
```

  A call `num_pairs [1,2,3] [4,5]` gives us

  `[(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)]`

- First, the first value from `xs`, 1, is fixed and all possible values from `ys` are chosen. Then, the process is repeated for the remaining values from `xs` (2 and 3)

List comprehensions (cont.)

- This order is not accidental, since we can have the second generator to depend on the value given by the first generator, e.g:

```
triangle ::  Int -> [(Int,Int)]
triangle n = [(x,y) | x <-[1..n], y<-[1..x]]
```

Then calling `triangle 3` gives us

```
[(1,1),(2,1),(2,2),(3,1),(3,2),(3,3)]
```

Thus, the value of `x` restricts how many values are considered for `y`

List comprehensions (cont.)

- Example: Pythagorean triples (where the sum of squares of the first two numbers is equal to square of the third one):

```
pyTriples ::  Integer -> [(Integer,Integer,Integer)]
pyTriples n = [(x,y,z) | x <-[2..n], y<-[x+1..n],
  z <- [y+1..n], x*x + y*y == z*z]
```

Here the test combines the values from the three generators

List comprehensions (cont.)

- If some generator patterns are **refutable**, i.e., may sometimes fail, the corresponding elements are filtered out from (not counted in) the result. For instance,

```
heads ::  [[a]] -> [a]
heads zs = [x | (x:_) <- zs]
```

If we apply

```
> heads [[],[2],[4,5],[]]
```

the result is simply `[2,4]`

Primitive recursion on lists (reminder)

- 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 on lists (examples)

Simple list construction (from the given list):

```
doubleAll [] = []
doubleAll (x:xs) = 2*x : doubleAll xs
```

List filtering (retaining only even numbers):

```
selectEven [] = []
selectEven (x:xs)
  | isEven x =  x : selectEven xs
  | otherwise = selectEven xs
```

where

```
isEven ::  Integer -> Bool
isEven x = mod x 2 == 0
```

Primitive recursion on lists (examples)

List insertion sorting (top-down definition):

```
iSort ::  [Integer] -> [Integer]
iSort [] = []
iSort (x:xs) = ins x (iSort xs)
```

where

```
ins ::  Integer -> [Integer] -> [Integer]
ins x [] = [x]
ins x (y:ys)
  | x <= y = x:(y:ys)
  | otherwise = y:(ins x ys)
```

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))
```

Exercise set 2

- The second assignment for you to solve (exercise set 2) is added to VMA right after these lecture slides
- The solutions should be uploaded to VMA (using the provided submission feature).
- The deadline for uploading (without penalties): October 25th (Monday)