1 Functional programming

Lists are well suited for recursive functions and pattern matching with, e.g., `match-with` as illustrated in the next example:

Listing 1.1 listPatternMatching.fsx: Examples of list concatenation, indexing.

```
let rec printListRec (lst : int list) =
  match lst with
    elm::rest ->
      printf "%A " elm
      printListRec rest
    | _ ->
      printfn ""

let a = [1; 2; 3; 4; 5]
printListRec a
```

```
$ fsharpc --nologo listPatternMatching.fsx
$ mono listPatternMatching.exe
1 2 3 4 5
```

The pattern `l::rest` is the pattern for the first element followed by a list of the rest of the list. This pattern matches all lists except an empty list, hence `rest` may be empty. Thus the wildcard pattern matching anything including the empty list, will be used only when `lst` is empty.

Pattern matching with lists is quite powerful, consider the following problem: · pattern matching

Problem 1.1

Given a list of pairs of course names and course grades, calculate the average grade.

A list of course names and grades is `[("name1", grade1); ("name2", grade2); ...]`. Let's take a recursive solution. First problem will be to iterate through the list. For this we can use pattern matching similarly to Listing 1.1 with `(name, grade)::rest`. The second problem will be to calculate the average. The average grade is the sum all grades and divide by the number of grades. Assume that we already have made a function, which calculates the `sum` and `n`, the sum and number of elements, for `rest`, then all we need is to add `grade` to the `sum` and `1` to `n`. For an empty list, `sum` and `n` should be `0`. Thus we arrive at the following solution,

Listing 1.2 avgGradesRec.fsx:
Calculating a list of average grades using recursion and pattern matching.

```
let averageGrade courseGrades =
  let rec sumNCount lst =
    match lst with
      | (title, grade)::rest ->
        let (sum, n) = sumNCount rest
        (sum + grade, n + 1)
      | _ -> (0, 0)

  let (sum, n) = sumNCount courseGrades
  (float sum) / (float n)

let courseGrades =
    ["Introduction to programming", 95;
    "Linear algebra", 80;
    "User Interaction", 85;]

printfn "Course and grades:\n%A" courseGrades
printfn "Average grade: %.1f" (averageGrade courseGrades)
```

```
$ fsharpc --nologo avgGradesRec.fsx && mono avgGradesRec.exe
Course and grades:
[("Introduction to programming", 95); ("Linear algebra", 80);
 ("User Interaction", 85)]
Average grade: 86.7
```

Pattern matching and appending is a useful combination, if we wish to produce new from old lists. E.g., a function returning a list of squared entries of its argument can be programmed as,

Listing 1.3 listSquare.fsx:
Using pattern matching and list appending elements to lists.

```
let rec square a =
  match a with
    elm :: rest -> elm*elm :: (square rest)
    | _ -> []

let a = [1 .. 10]
printfn "%A" (square a)
```

```
$ fsharpc --nologo listSquare.fsx && mono listSquare.exe
[1; 4; 9; 16; 25; 36; 49; 64; 81; 100]
```

This is a prototypical functional programming style solution, and which uses the "`::`" for 2 different purposes: First the list `[1 .. 10]` is first matched with `1 :: [2 .. 10]`, and then we assume that we have solved the problem for `square rest`, such that all we need to do is append `1*1` to the beginning output from `square rest`. Hence we get, `square [1 .. 10]` ↷ `1 * 1 :: square [2 .. 10]` ↷ `1 * 1 :: (2 * 2 :: square [3 .. 10])` ↷ `...1 * 1 :: (2 * 2 :: ... 10 * 10 :: [])`, where the stopping criterium is reached, when the `elm :: rest` does not match with a, hence it is empty, which does match the

wildcard pattern "_". More on functional programming in Section 1

Arrays only support direct pattern matching, e.g.,

Listing 1.4 arrayPatternMatching.fsx:
Only simple pattern matching is allowed for arrays.

```
let name2String (arr : string array) =
  match arr with
    [| first; last|] -> last + ", " + first
    | _ -> ""

let listNames (arr :string array array) =
  let mutable str = ""
  for a in arr do
    str <- str + name2String a + "\n"
  str

let A = [|[|"Jon"; "Sporring"|]; [|"Alonzo"; "Church"|];
   [|"John"; "McCarthy"|]|]
printf "%s" (listNames A)
```

```
$ fsharpc --nologo arrayPatternMatching.fsx
$ mono arrayPatternMatching.exe
Sporring, Jon
Church, Alonzo
McCarthy, John
```

The given example is the first example of a 2-dimensional array, which can be implemented as arrays of arrays and here written as `string array array`. Below further discussion of on 2 and higher dimensional arrays be discussed.