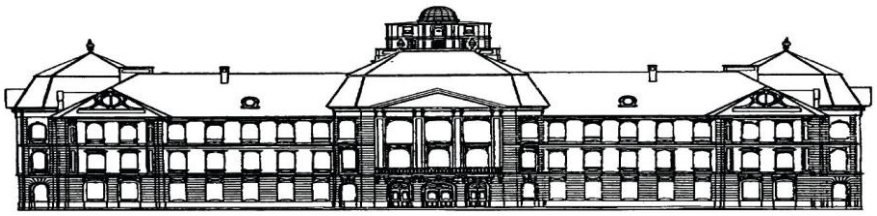
Functional Languages

Functional Languages

Roland Király



EKF • Eger, 2011

Lecture notes sealed: November 11th, 2012.

Nemzeti Fejlesztési Ügynökség
www.ujszechenyiterv.gov.hu
06 40 638 638

MAGYARORSZÁG MEGÚJUL

A projekt az Európai Unió támogatásával, az Európai Szociális Alap társfinanszírozásával valósul meg.

The World of Functional Programing Languages

The world of functional programming languages is not very well known even among programmers. Most of them are skilled in the use of object oriented and imperative languages but they lack any kind of knowledge regarding the ones mentioned above. It is often hard to explain what makes a language functional. This has several reasons, among other things, these languages are either constructed for special purposes and thus they cannot spread widely, or they are so difficult to use that an average programmer does not start using them at all, or even if he does, he cannot measure up to the challenge. Not even in education can you meet this paradigm, except for some precedents to follow. Imperative and OO languages have spread in most institutions of education and it is hard to go off the beaten track. In spite of it all it is worth taking up functional languages such as Haskell [12] , Clean [10], and Erlang [6] more seriously. The enlisted languages are widely spread, easy of attainment, logically constructed and some of them are used in the industry as well. Due to all these attributes we have chosen two of the languages listed in these lecture notes, but we have not forgotten about the currently aspiring languages like F# either. That is why, in each chapter, we show language elements, and the example programs which belong to them, in these three languages. Based on the information above and on the fact that the dear reader is holding this book in his hands, we assume that you know nothing, or only a little, about functional languages but you are eager to learn how to use them. On account of all these we will follow the method of explaining special functional language elements with examples from imperative and OO program constructions and drawing parallels among the paradigms. In case we cannot find analog examples of certain concepts in ”traditional” programing languages we will take examples out of everyday life or other fields of informatics. By this method we will surely accomplish our objective.

Accordingly, this book aims at approaching the above mentioned programing languages from a practical point of view. Obviously, we also put emphasis on the theoretical issues of learning these languages, but that is not our priority. To acquire the proper programming style we suggest that you write the example programs shown in this book; and in order to make it easy for you we explain the source code of these programs step by step, wherever it is possible.

Functional Languages in General

Besides getting to know the theory it is worth it to start learning functional languages by observing the philosophical background of the paradigm. Moreover it is also good to examine the most important properties of functional languages. While teaching this paradigm we also pan out about why and when it is worth it to use these languages. A favorable property of functional languages is their power of expression which means that you can define so much with so little source code. In practice this implies that you can solve relatively complicated problems in a relatively short time using the least amount of energy possible. The language of

functional programing languages is close to the language of mathematics. Mathematical formulas can be almost directly rewritten in functional languages. It is yet another very useful property if you consider that programing does not start with launching the engineering tool and writing the program but with the design phase. First you must create the mathematical model of the program and create its specification and just after that can you start typing the code into a text editor. Let us have a look at a concrete example of using functional languages without knowing any of the languages you would like to learn. Let the calculation of n! be our first example. This problem is so common that you find a version of it in almost every programming book.

```
factorial( 0 ) -> 1;
factorial( N ) -> N * factorial( N- 1 ).
```

Program 1.1. Factorial function - Erlang

You can see that the source code, which was written in Erlang, is pretty simple and does not differ very much from the function defined with a mathematical formula that looks like this:

$$fakt\ n = n * f\ akt\ n - 1$$

If you write this definition in Clean, it is even more similar to the mathematical formula.

```
fakt n = if ( n==0 ) 1
        ( n * fakt( n-1) )
```

Program 1.2. Factorial function - Clean

The next useful property, besides the power of expression, is the effectiveness of creating recursion. Obviously, you can also create recursive programs in imperative languages, but a major disadvantge of such control structures is that the number of recursive calls is limited, meaning that they surely stop after some steps. If the base criteria does not stop the recursion, then stack overflow will. In functional languages it is possible to use a special version of recursion. This construction is tail-recursion, in which the execution of the program is not affected by the stack so if you want, it will never stop.

```
f()->
        ...
        f( )
```

```
g1( ) ->
        g1( )
```

Program 1.3. Tail-recursion calls

The point of this construction is that recursive calls do not use the stack (at least not in a conventional way). Stack introspection and graph rewriting systems always return the value of the last recursive call. The condition, under which the tail-recursion is fulfilled, is that the last command of the recursive function must be calling the function itself and this call must not be part of an expression. (program list 1.3).

```
fact( N ) -> factr( N, 1 ).

factr( 0, X ) -> X;
factr( N, X ) -> factr( N-1, N*X ).
```

Program 1.4. Tail-recursive factorial function - Erlang

```
fact n = factr n 1

factr 0 x = x
factr n x = factr( n-1 ) ( n*x )
```

Program 1.5. Tail.recursive factorial function - Clean

```
let rec fakt n =
        match n wit h
        | 0 -> 1
        | n -> n*fakt(  n-1 )

let rec fakt n = if n=0 then 1 else n*fakt( n-1 )
```

Program 1.6. Tail-recursive factorial function - F#

Observing recursive programs and considering the fact that recursion involves a highly restricted execution, you can wonder why we need these programs at all, since every recursive program has an iterative equivalent. This is only true if stated in a book that is not about functional languages. In functional languages you cannot, or only very rarely, find

control structures that execute N iteration steps, like for or while. The only way of iteration is recursion, so you can understand why we need the tail-recursive version.

```
loop ( Data ) ->
        receive
        {From, stop} -> From ! stop;
        {From, {Fun, Data}} -> From ! Fun( Data );
        loop( Data )
end.
```

Program 1.7. Recursive server - Erlang

In Erlang, as well as in other functional languages – it is not rare for the server part of client-server programs to use recursion for the implementation of busy waiting. The server-side part of client-server applications moves to a new state after serving every single request, namely, it calls itself in a recursive way with the current values. In case you wanted to write the iteration you can see in program list 1.7 with traditional recursion, the server application would stop relatively soon with a version of the stack overflow message used on the particular server. Besides the few characteristics enlisted so far, there are several other attributes of functional languages which will be discussed in this book later. We will show the special elements of functional languages, traditional recursion and its tail-recursion variant, the use of set expressions and the problems of lazy and strict evaluation in detail. As we have mentioned all the chapters are highly practice oriented and they do not merely explain the use of “purely” functional language elements but also deal with how functional languages are used for sending messages, or writing concurrent programs in the industry.
The book explains the programming paradigm in three known functional languages. The first is Erlang [6], the second is Clean [10] and the third one is F# [11]. Since, its industrial use is pretty significant and it possesses all the attributes that make it possible for us to exploit the advantages of the functional paradigm and to write distributed and network programs, we use Erlang as our main language. Clean was chosen as second because it is probably the most expressive among functional languages and it is very similar to Haskell [12], which is a classic functional language. F#, the third language, is spreading rapidly, so we should not forget about it either. The explanations of the example programs mainly refer to the Erlang source code, but where it is possible we also explain the Clean and F# program codes.

Erlang. Erlang language was developed by Ericsson and Ellemtel Computer Science Laboratoriesi. It is a programing language which allows the development of real-time

distributed and highly fault-tolerant systems. Ericsson uses the extension of Erlang Open Telecom Platform to develop telecommunication systems. With the use of OTP, data exchange without shared memory among applications is possible. The language supports the integration of components written in various programming languages. It is not a ”pure” functional language, it has a unique type system and it is perfect for creating distributed systems.

Clean. Clean is a functional language that is accessible on numerous platforms and operating systems and can be used in an industrial environment as well. It has a flexible and stable integrated development environment consisting of an editor and a project manager. Clean IDE is the only developing tool that was completely written in a functional language. The compiler is the most complex part of the system. The language is modular, it consists of definition modules (dcl), and implementation files (icl). The compiler compiles the source code into a platform independent ABC code (.abc files).

FSharp. The functional paradigm will have a significant role in the .NET framework. The new language of the functional paradigm is F#. The language is being developed by Microsoft for the .NET framework. F# is not a pure functional language. It supports object oriented programing, access to the .NET libraries and database management. In F# you can write SQL queries (with meta grammar) and compile them to SQL with an external interpreter. Code written in F# can be used in C# programs, since they can also access F# types.

General Characteristics of Functional Programs

Before beginning to discuss language elements in detail, we must clarify some concepts and get acquainted with the basics of functional languages.

Pure and impure languages. Among functional languages there are pure and not purely functional languages. LISP [13], Erlang and F# are some other well-known functional languages are not purely functional, as their functions contain side effects (and in case of some languages, destructive assignments, too)

1.1 Note: We will return to side effects and destructive assignments later...

Haskell is a pure language that uses the Monad technique for handling input and output. Beside Haskell Clean and Miranda are concerned to be pure. The base of functional languages is calculus and functional programs are usually composed of a sequence of function definitions and an initial expression. When executing the program we can get to the end result by evaluating this initial expression. For the execution we normally use a reduction or a graph rewriting system which reduces the graph, constructed from the program code, with a series of evaluations.

1.2 Note: In mathematical logic and in computing science Lambda calculus is the formal tool of function definitions, function implementations and recursion. Alonzo Church introduced it as part of his research of the basics of mathematics in the 1930s.

Lambda calculus is appropriate for the formal description of calculable functions. This attribute made it the base of the first functional languages. Since every function that can be defined with calculus is computable by a Turing machine [7], every “calculable” function can be defined with calculus. Later, extended Lambda calculus was developed based on these principles, which contains the type, operator and literal concepts regarding data. When a program is being compiled, programs of extended Lambda calculus are first transformed to original Lambda calculus which is then implemented. For example ML, Miranda and Haskell are extended Lambda calculus based languages…

Term rewriting systems. Term rewriting system (TRS) is an alternative version of Lambda calculus. In this model the functions of a functional program correspond rules of rewriting, the initial expression corresponds a reducible term. TRS is a more universal model than Lambda calculus. Nondeterministic operations can also be defined with it. In Lambda calculus the sequence of left-most, outmost derivations lead us to normal form, while in TRS you can grant the achievement of normal form with parallel-outmost reductions. It can be used for implementing functional languages, but you must keep it in mind that it is not secured by the single calculation of equivalent expressions.

Graph rewriting systems. The generalization of TRS is the graph rewriting system (GRS), which consists of constant symbols and rewriting rules interpreted on names of graph nodes. The initial expression of the functional program corresponds a special initial graph, its functions correspond graph rewriting rules. The most important difference between the two rewriting systems is that in GRS equivalent expressions are only calculated once. This makes it much more suitable for implementing functional languages. The general method for implementation is transforming the program to TRS form, then creating a GRS from it and finally reaching normal form with graph reduction. Pattern matching of functional languages and the priority used in it can also be applied in the realization of graph rewriting rules. Such systems using reduction method are called functional GRS (FGRS). By using FGRS in the implementation of functional program languages you can easily define such non-functional characteristics of logical and imperative languages as side effect.

Functional properties

Basic language constructions. **Functional languages - pure or impure -** contain several constructions that cannot be found in OO languages, or only with limited functionality. They also possess properties that only characterize functional languages.

Functions and recursion. **When** writing a functional program we create function definitions and by evaluating the initial expression, which is also a function, we start running the program. In functional languages a function can call itself with tail-recursion as many times as required.

Referential transparency. **The** value of an expression does not depend on where it is located in the program code, so wherever you place the same expression its value remains the same. This is preferably true in pure functional languages where functions do not have side effects, so they do not change the particular expression even during their evaluation.

Non-refreshable variables. **The use** of non-refreshable variables is a technique that is typical in functional languages, but it is the least known among OO programmers. Destructive assignment does not allow multiple binding of variables, such as I = I + 1 type assignments.

Lazy and strict evaluation. **The evaluation of expressions in functional languages can be lazy or strict.** In case of lazy evaluation the arguments of functions are only evaluated if it is absolutely necessary. Clean uses such lazy evaluation, while Erlang belongs to the strict languages. Strict evaluation always evaluates expressions as soon as possible.

Lazy evaluation would first interpret the expression inc 4+1 as (4+1)+1, while strict evaluation would immediately transform it to 5 + 1.

Pattern matching. **Pattern matching applied to functions means that functions can have multiple clauses and various patterns of formal parameters can be assigned to a particular clause.** When calling a function, the clause, whose formal parameters match the actual parameters of the call, is run. In OO languages we call such functions overload functions. Pattern matching can be applied to pattern matching of variables, lists and to select clauses.

Higher order functions. **In functional languages it is possible to pass expressions, namely specially defined functions, to other functions' argument list. Of course, it is not the call that is among the arguments but the prototype of the function. In several programming languages you can bind functions in variables and later use such variables as functions.** You can also place expressions in list expressions. Parameterization with functions helps us to create completely universal functions and programs, which is extremely useful when creating server applications.

The Curry method. **The Curry method is partial function application which means that the return value of functions with multiple parameters can be a function and the remaining parameters. Accordingly, every function can be seen as unary.** In case a function had two variables, only one would be considered as function parameter. The assignment of the parameter creates a new unary function which can be applied to the second variable. This method is also operable with more variables. Erlang and F# lack this language element but it is possible to carry out. In order to do so you can use a function expression.

Static type systems. **In static type systems type declaration is not compulsory, but it is required from the side of interpreter to be always able to conclude the most generic type of the expression. The base of the static system is the polymorphic type system of** Hindley-Milner [7]. Obviously, the optionality of type declaration does not mean that there are no types in a particular language. There are types, indeed, but the type deduction system and the compiler ensures their proper management and the right evaluation of the expressions.

Set expressions. **Besides various language constructions you also find numerous data types in functional languages, such as sorted n-vectors, also called tuples, set expressions and list generators which can be constructed for them.** List data type is based on Zermelo-Fraenkel's [3] set expression. It contains a generator which defines the criteria under which an element belongs to a set and it defines the number of elements and how they should be

constructed. In theory we can generate infinite lists in a given programing language, as we do not give the elements of the list but its definition.

Basic Input-Output

In order to understand and practice the examples in this section you must learn how to create programs in Erlang, Clean and F# and how to handle their input and output when running them. All the tools and runtime environments used for creating these programs are free to access, but you are unbound to choose any other alternative software and operating system that you prefer.

Erlang. Considering Erlang programs, flexibility is an important aspect both in their development and in their application. Erlang does not have a specific development tool. We can write programs with text editors or with general purpose graphical development tools created for other languages. In Linux systems [14] one of the most wide-spread editor programs is Emacs [16]. This, however, cannot be seen as a text editor, since it is capable of handling almost any known language, even Tex [9] based languages. Yet another interesting system is Eclipse [18], which was preferably meant for developing Java [15] programs, but it has a plug-in module that can be used for writing and running Erlang programs. If you do not want to use a development tool, you can directly write programs in a terminal, but in that case you will have a more difficult task with creating more complicated software that is built up of multiple modules.

Clean. The example programs of [10] cause less trouble, since the language has its own unique integrated development tool. Besides text editing and program running capabilities, the system has a built-in debugger, which makes the job of the programmer easier.

Note 2.1: Of course, Erlang has also got a debugger, which is pretty useable, but its error messages may seem a little terrifying at first…

FSharp. Traditionally, you can write F# programs with the latest version of MS Visual Studio or with any editor that is suitable for writing programs. The runtime environment and the debugger of the language are convenient and easy to learn. F# is suited for creating larger projects and when ready they can be integrated into C# and C++ language programs.

Initial Steps of Running Erlang Programs

We will analyze the steps of creating our first program from writing the code to running it. This program implements simple addition of two numbers stored in variables. After creating the command line version, we will write the version which uses a function and is placed in a module, so that we can run it later. The practical benefit of the terminal version is that you do not have to configure a development tool or a runtime environment. Note 2.2: Obviously, to run the program you need the compiler of the language and its runtime environment…
To write the Erlang program let us type the word erl in the terminal and that will launch the Erlang system in terminal mode. Once the system is launched, type the following program lines.

```
> A = 10.
> B = 20.
> A + B.
> 30
```

Program 2.1. Addition of variables in terminal

In the first line of program list 2.1 we assign a value to variable A. Actually, this operation should be called binding because, as we have declared earlier, in functional languages variables are assigned only once, namely data binding occurs only once.

The regular destructive assignment of imperative languages (A = A + 1) does not occur in functional languages. In practice this means that if you repeat the assignment, you will get an error message informing you about the previous binding of the variable. In the second line we bind the value of variable B, and then we add the content of the two variables with the expression A + B. In the last line you can see the result of the expression displayed in the terminal. In case you wanted to repeat these few instructions, you must call a library function named f which will de allocate bound variables to avoid errors.

Note 2.3: The parameter less call of f() (example 2.2) de allocates every variable and all data stored in them is lost…

```
> f(A).
> f(B ).
> f( ) .
```

Program 2.2. De allocating variables

As you can see it in the example programs the end of instructions is marked by a period. In case of modules this is different. After functions there is a period, after their instructions there is a comma. The ; sign is used to separate clauses of functions or branches. If you run the program without an error, prepare its stored version, too, to avoid typing in its instructions again and again before each execution. The functions that belong together are usually saved in modules, the modules are saved in files because this way you can use their content any time. Every module has an identifier which is its name and an export list which makes the functions of a module visible for the world outside the module.

```
-module( mod ).
-export( [ sum/2 ] ).

sum(A, B) ->
        A + B.
```

Program 2.3. Module of the sum function

The order of the functions inside a module is totally unimportant. Their order cannot have the slightest effect on compiling and execution. The name of the module must be placed in -module() brackets and it must be the same as the name of the file containing the module. The export list, as it can be concluded from its name, is a list in which you must place all public functions with a pair derived from their name and arity (f/1, g/2, h/0). Arity is the number of the parameters of a function, so the arity of a function with two parameters is two, while the arity of a parameterless function is zero.

Note 2.4: In the documentation and in also in the description of programs, functions are referred to based on their name and arity because that way we can identify them unambiguously.

In case of using more modules the name of a function's module is also part of the identifier. The name consisting of the name of the module and the identifier of the function is called qualified name. (module:function/arity, or mod:sum/2)

Names of functions always start with lowercase, names of variables start with capital letters. Functions are introduced with their name, followed by the parameter list, which can be followed by various guard conditions. After the conditions, comes the body of the function, which, in case of one clause, is closed with a period.

Note:2.5: We will examine creating and using of functions in detail later …

If you are ready with the module, save it with the name used inside the module, as identifier. Then, you are ready to compile and run it.

```
> c( mod )
> { ok, sum}
> mod:sum( 10, 20 ).
> 30
```

Program 2.4 Compilation in terminal

The easiest way of executing the compilation of a module in a terminal is with the c(sum) formula, where c refers to the word compile, and between the brackets you can find the name of the module (2.4). The compiler immediately informs you about the success of compilation or about the errors made. If you succeed, you can call the functions, which are in the module and enlisted in the export list, at once. In order to run and qualify it, you must use the name of the module. The name of the module is followed by the name of the function and then come the actual parameters based on the formal parameter list. If you do everything right, the result of function sum/2, namely the sum of the two numbers given, appears on the screen.

```
> f( A ).
> f( B ).
> f( ).
```

Program 2.5. Deallocation of variables

Introduction to Clean

Creating Clean programs with the integrated development system is significantly easier than creating programs in those functional languages which do not have IDE. After launching the system, you can write the source code of programs in the code editor and then, after debugging you can run them. In order to write a Clean code the following steps must be followed:

- Run Clean IDE. Create a new file with icl extension, with any name, in the File menu (New File...). This becomes your implementation module
- Create a new project, also in the File menu (New Project...). Its name must be the name of our module. By doing so, you have created a project, which contains a reference to the .icl file created earlier. At this point you should see two windows. The first is the project manager window, the other one is the .icl file, in which you type the source code. If you try to carry out these steps in a reverse order, you will see the "Unable to create new project there’s no active module window." error message.
- The next step is to type in the line with the name of the import module in the window showing the content of the .icl file. It is compulsory to write the name of the .icl file,

without its extension, in place of the name of the module! This is how you inform the compiler about the name of your module and about the file that contains its content.

- If you do everything right, then, you can start writing the source code. In order to be able to use the example codes of this book (including the basic operators, lists and types. etc.) you will need the Standard Environment library. You can access it by typing import StdEnv after the name of the module.
-

```
module ownmodul
import StdEnv

// h e r e y.o.u.t.y.p.e.i.n.t.h.e.c.o.d.e
…
..
.
```

Program 2.6. Clean example program.

Writing and running F# programs

A simple option is to use Visual Studio 2010. This is the first Visual Studio that contains F#. Let us launch the tool and create a new F# project in the file menu. Open the File/New/Project dialogue window, then, choose F# Application in the Other Languages/Visual F# category. Open the „Interactive" window from the View/Other Windows/F# Interactive menu (you can also use the Ctrl+Alt+F keyboard shortcut). Commands can be directly entered in the interactive window, or in the automatically created Program.fs file through the editor. In this latter case you can run the desired part of the code by right clicking the particular line or the selected area and choosing the Send Line to Interactive – or the Send to Interactive – menu items. You can run the whole program by choosing the Debug/Start Debugging menu item (shortcut: F5). For practice, enter the lines of list 2.7 into the interactive window.

```
let A = 10;;
let B = 20;;
A + B;;
```

Program 2.7. F# Example program

The line from list 2.8 appears in the interactive window

```
val it:int = 30
```

Program 2.8. The interactive window

However, there is a much simple way of writing F# programs. You have the option of using the command line interpreter of the language, which can be downloaded from www.fsharp.net for free.

```
module ownModul

let a = 10
let b = 20
let sum a b = a + b
```

Program 2.9. F# module

When using the command line interpreter, you must place the program in a file created based on the example that you can see in list 2.9. (in case of using multiple modules the use of keyword module is compulsory)

Handling side effects

Let us observe the sum/2 function written in Erlang once again (program list 2.3). You can see that the function does not do anything besides adding the two values in the parameter list. It returns to the place of call with the sum, but nothing is displayed on the screen, there are no messages sent or received. It only gets data from its parameters. We can declare that these sorts of functions have no side effect. In many languages, functions with side effects are called dirty, which refers to the fact that their operation is not completely regular. In imperative languages functions with side effects are called methods (void), the ones without side effects are called functions. In these languages functions without side effects have a type, while methods do not have a type or their type can be considered to be void. Clean, Erlang and F# are not pure functional languages, unlike Haskell [12] in which IO operations are based on so called Monads. To see the result of the example programs, sometimes we have to depart the principle of not creating dirty functions. Also, side effects cannot be dispensed because output commands are predestined to have side effects. Such functions not only return their value, but also carry out output operation...

An output command of Erlang is the format function found in the IO library module. We can display data on standard input or on input defined by the programmer, in a formatted way with it. For our second tryout let us create a version of our summing program which displays a message on standard input and returns with the value. This function is a typical example of side effects, which can make the functions of functional programs difficult to use. By the way, this statement is true in case of any other paradigm as well.

```
-module (mod ).
-export ( [ sum/2 ] ).

sum(A, B) ->
    io:format( "according to me ~w~n",
                                [random:uniform( 100 ) ] ),
A + B.
```

Program 2.10. Function with side effect - Erlang

In example 2.10 function sum/2 first displays the result of the function according to it. The random module generates a random number with its uniform/1 function and uses it as a guess obviously, with a highly limited chance of guessing it right. After displaying, it returns with the true result. The function in this form does not make too much sense but it properly illustrates how side effects and format work. Programmers often use IO operations as a tool of debugging. In programming environments where debugging is difficult or there is no alternative solution, output commands of the particular language can be used for displaying variables, lists, other data and for finding errors.

Data Management

Variables

Handling of variables in functional languages is a lot different from the way you have got accustomed to in imperative and OO environment. Variables can be assigned only once, namely you can only bind a value to them once. The lack of destructive assignment rules out constructions which are based on consecutive multiple assignment (I = I + 1). Naturally, there is a functional equivalent to every single form used in OO and imperative languages. Destructive assignment can be replaced with the introduction of a new variable, A = A0 + 1. Iteration (loops, such as for, while and do-while) is carried out with recursion and multiple clause functions. This will be discussed in the section about functions.

Storing data. In functional languages you can store your data in variables; variables can be stored in lists or in sorted n-vectors (tuple), and naturally in files and tables of data base managers.

Note 3.1: You can find database management modules or distinct systems optimized to the particular language in several functional languages. The known database management system of Erlang is MNESIA [17], and its query language is QLC [17]…

We will not deal with file and database based storage in detail, but it is essential to examine the three previously mentioned data structures in order to be able to manage data efficiently. Variables, tuples and lists characterize data storage of functional languages the best, however, there is yet another special element, that can be found in many of these languages, called atom, used mostly as identifier. The name of functions is also an atom, and it is common to find atoms in parameter lists which identify the particular clause of functions (program list 3.2).

Simple variables. Integers, real numbers, characters and strings can be stored in variables. In most functional languages character strings are “syntactic candy” created for lists.
Note 3.2: The phrase „syntactic candy” refers to the fact that string is not a real data type but a list storing codes of characters and has a mechanism, a few functions and operators which make its use, display and formatting easy …

```
A = 10, %integer
B = 3.14, %real
C = apple, %atom stored in variable
L = [ 1, 2, A, B, a, b ], %list with mixed content
N = { A, B, C = 2}, % tuple with three elements
LN = [ N, A, B, { a, b } ] %list with four elements
```

Program 3.1. Erlang variables

```
let A = 10 //int
let B = 3.14 //float
let L1 = [ 1, 2, A, B, ’a’, ’b’ ] //list with mixed content
let L2 = [ 1; 2; 3; 4 ] // int list
let N = ( A, B, C ) //tuple with three elements
let LN = [ N, A, B, ( ’a’, ’b’ ) ] //list with four elements
```

Program 3.2. F# variables

Expressions

Operation arithmetics. Functional languages also possess the same simple, numeric and logical operators as their imperative and OO counterparts. The only difference in how they evaluate expressions is that in functional languages you can find both lazy and strict evaluation. Despite the differences there is a relevant similarity in writing simple operations and in the structure of expressions. List 4.1 illustrates, though not at its full extent, the use of operators both numeric and logical.

```
> hello
hello
> 2 + 3 * 5.
17
> ( 2 + 3 ) * 5.
25
> X = 22
22
> Y = 20.
20 > X + Y.
42
> Z = X + Y
42
> Z == 42.
true
> 4/3.
1.3333333333333333
> { 5 rem 3, 4 div 2}.
{2 , 2}
```

Program 4.1. Use of operators - Erlang

List 4.2 contains operators which you can use when writing various expressions.

Note 4.1: If you observe this list thoroughly, you can realize that it does not only contain operators but also their precedency. The first row contains the strongest expressions and below that you can find ”weaker” and ”weaker” ones as you scroll down the list...

(unary) +, (unary) -, bnot, not

/, *, div, rem, band, and (left associatives)

+, -, bor, bxor, bsl, bsr, or, xor (left associatives)

++, -- (right associatives)

==, /=, =<, <, >=, >, =:=, =/=

andalso

orelse

Program 4.2. Operator precedency - Erlang

!

o !! %

^

* / mod rem

+ - bitor bitand bitxo r

++ +++

== <> < <= > >=

&&

||

:=

‘bind‘

Program 4.3. Operator precedency - Clean

-, +, %, %%, &, &&, !, ~ (prefix operators, left associatives)

*, /, % (left associatives)

-, + (binary , left associatives)

<, >, =, |, & (left associatives)

&, && (left associatives)

o r, || (left associatives)

Program 4.4. Operator precedency - F#

Note 4.2: Besides numeric and logical types you also find operators of the string type. These are ++ and -- . In fact these expressions are list handling operators. We know that these two types are basically the same...

Pattern matching

The use of patterns and pattern matching is a well-known and commonly used operation in functional languages. You can find similar concepts, such as operator and function overload, in imperative programing languages and in OO programs as well. Before starting to deal with pattern matching in functional languages, let us examine the functioning of the overload method, often used in OO classes. The essence of the technique is that you can create methods with the same name but with different parameter lists within the same class

```
class cppclass
{
        double sum( int x, int y )
        {
                return x + y;
        }
        string sum( string x, string y )
        {
                return toint( x ) + toint( y );
        }
}
```

Program 4.5. Overloading in OO languages

You can access the methods of the class defined in source text 4.5 with a qualified name of the instance after instantiation. (PName -> sum(10, 20)). Thus, you can call two different version of the same method. In case method sum/2 is called with two integers, the first version that works with ints will be run. If you have string parameters, the second version will be run (4.6).

```
cppclass PName = new PName( );
int result1 = PName -> sum( 10, 20 );
int result2 = PName -> sum( "10", "20" );
```

Program 4.6. Using overload method

When overload methods are called, the branch with “matching” formal and actual parameters will run.

Note 4.3: Pattern matching in OO programs follow different principles from the ones used in functional languages. The difference originates from the fact that the functional paradigm uses a different mentality for solving various problems. …

Pattern matching is a technique similar to overload but it works in a different way from the functions of functional languages using patterns. In these languages any function can have multiple clauses, and the running of a clause always depends on the parameters.

```
patterns( A ) ->
        [ P1, P2 | Pn ] = [ 1, 2, 3, 4 ],
        { T1, T2, T3 } = { data1, data2, data3 },
        case A of
                { add, A, B } -> A + B;
                { mul, A, B} -> A * B;
                    { inc, A} -> A + 1;
                  { dec, B } -> A - 1;
                                _ -> {error , A}
        end.
```

Program 4.7. Pattern matching in case - Erlang

```
let patterns a =
        match a with
        | ( "add", A, B) -> A + B
        | ( "mul", A, B) -> A * B
        | ( "inc", A) -> A + 1 //HIBA!
                // Unfortunately, you cannot do this in F#
                // this language is not lenient enough
        | _ -> ( "error", A) // This also means trouble
        …
```

Program 4.8. Pattern matching in case - F#

Case exploits pattern matching (program list 4.7), and when you use if or send messages they are also patterns that the reaction of the recipient routine is based on. We will discuss sending messages later on.

```
receive
        { time, Pid } -> Pid ! morning;
        { hello, Pid} -> Pid ! hello;
        _ -> io:format( "~ s~n", "Wrong message format" )
end
```

Program 4.9. Receiving messages - Erlang

Not only can you use pattern matching in functions, selection or receiving messages but also in handling variables, tuple, list and record structures. Program list 4.10 contains examples of the further uses of patterns and their explanation.

```
{ A, abc } = { 123, abc }
                %A = 123,
                %abc matches the abc atom
{ A, B, C } = { 123 , abc, bca }
                %A = 123, B = abc, C = bca
                %successful pattern matching
{ A, B } = { 123, abc, bca }
                %wrong match,
                %because there are not enough elements
                %on the right side
```

```
A = true
                %you bind value true to A
{ A, B, C } = { { abc, 123 }, 42, { abc, 123 } }
                %Proper match,
                %variables get the
                %{ abc, 123 }, the 42, and the { abc, 123 }
                %elements in order
[ H|T ] = [ 1, 2, 3, 4, 5 ]
                %Variable H is assigned value 1 ,
                %T is assigned the end of the list: [ 2, 3, 4, 5 ]
[ A, B, C|T ] = [ a, b, c, d, e, f ]
                %atom a gets into variable A ,
                %b gets into B, c gets into C, and T gets
                %the end of the list: [ d, e, f ]
```

Program 4.10. Pattern matching - Erlang

Use of guard conditions. Clauses of functions, if and case statements, the try block and the various message sending expressions can be extended with a guard condition, thus regulating their run. The use of guard conditions is not compulsory but they are an excellent option to make program codes easier to read.

```
max( X, Y ) when X > Y -> X;
max( X, Y) -> Y.
```

Program 4.11. Guard condition for a function clause - Erlang

```
maximum x y
| x > y = x
        = y
```

Program 4.12. Guard condition for a function clause - Clean

```
let max x y =
        match x, y w ith
        | a, b when a >= b -> a
        | a ,b when a < b -> b
```

Program 4.13. Guard condition for a function clause - F#

In example 4.11 function max/2 uses the guard condition of the first clause to choose the bigger one of its parameters. Obviously, this problem could have been solved with an if or a case statement, but this solution is far more interesting.

```
f( X ) when ( X == 0 ) or ( 1/X > 2 ) ->
…
g( X ) when (X == 0) orelse (1/X > 2 ) ->
…
```

Program 4.14. Complex guard conditions - Erlang

```
f x
| x == 0 || 1/x > 2 =
…
```

Program 4.15. Complex guard conditions - Clean

You can use operators at will in guard conditions but always ensure the result of the guard to be a logical value (true/false). The condition can be complex and its parts can be separated with logical operators from each other 4.14.

The If statement

As you know, every single algorithmic problem can be solved with the help of sequence, selection and iteration. It is no different in functional languages either. Sequence is obvious, since instructions in functions are executed sequentially. Iteration steps are carried out by means of recursion, while branches are realized in programs as functions with multiple clauses or as if, or case control structures. If uses guard conditions to ensure its proper branch to be selected. However, its functioning is slightly different from the routine one because it does not only have one true and one false branch, but it can have as many branches as the number of guard conditions. In every single branch, a sequence containing expressions can be placed which are executed if their guard is fulfilled. (list 4.16)

```
if
  Guard1 -> Expr_seq1;
  Guard2 -> Expr_seq2;
  …
end
```

Program 4.16. General form of an if statement - Erlang

```
if Guard1 Expr_seq1 if Guard2 Expr_seq2
…
```

Program 4.17. General form of an if statement in Clean

```
let rec hcf a b =
        if a = 0 then b
        elif a < b then hcf a ( b - a )
        else hcf ( a-b ) b
```

Program 4.18. F# if example (greatest common divisor)

In the majority of functional languages programmers prefer using case statements and many languages tend to be in favor of selections with multiple branches.

The Case statement

The other version of selection also has multiple branches. In C based programing languages it is common to use the switch keyword in the instruction of selections with multiple branches (program list 4.19), with case introducing the particular branches.

```
switch ( a )
{
        case 1:    …  break;
        case 2:    …  break;
        …
        default:  …break;
}
```

Program 4.19. The case statement in C based languages

In Erlang and Clean the selection after the case keyword contains an expression and then after the word of, an enumeration where you can make sure, based on the possible outputs, that the proper branch is selected. Each particular branch contains a pattern which is to be matched by the possible output (Expr in list 4.20) of the expression (at least it is supposed to match the pattern of one of the branches). Then comes the -> symbol followed by the

instructions that should be executed in the given branch. You can also set a default branch in the selection introduced by the _ -> formula. Any value matches the symbol _ . This means that the value of the expression after the case statement – no matter what it is, even including errors – will always match this branch. Note 4.4: If you do not want your function, containing a selection, to run partially, and you intend to write your programs with care, being mindful of errors during the evaluation of the expression, then you should always give a default branch at the end of selections...

```
case Expr of
        Pattern1 -> Expr_seq1;
        Pattern2 -> Expr_seq2;
        …
end
```

Program 4.20. General form of a case statement in Erlang

```
patterns Expr
| Expr == Pattern1 = Expr_seq1
| Expr == Pattern2 = Expr_seq2
| otherwise = Expr_seq3
```

Program 4.21. General form of a case statement in Clean

```
match Expr w ith
| Pattern1 -> Expr_seq1
| Pattern2 -> Expr_seq2
```

Program 4.22. General form of a case statement in F#

Branches are closed with ;, except for the last one, the one before the word end. Here you do not write any symbol, thus informing the compiler that it will be the last branch. Sticking to this rule is really important in nested case statements, because when nesting, instead of the branches of case, you have to use (or sometimes you do not have to) the ; symbol after the word end. (example program 4.23).

```
...
case Expr1 of
        Pattern1_1 -> Expr_seq1_1;
```

```
        Pattern1_2 -> case Expr2 of
                        Pattern2_1 -> Expr_seq2_1;
                        Pattern2_2 -> Expr_seq2_2;
                        end
end,
…
```

Program 4.23. Nested case statements in Erlang

```
module modul47
import StdEnv
patterns a
| a == 10 = a
| a > 10 = patterns2 a
patterns2 b
| b == 11 = 1
| b > 11 = 2
Start = patterns 11
```

Program 4.24. Nested case statements in Clean

```
match Expr w ith
| Pattern1_1 -> Expr_seq1_1
| Pattern1_2 -> match Expr2 w ith
        | Pattern2_1 -> Expr_seq2_1
        | Pattern2_2 -> Expr_seq2_2
```

Program 4.25. Nested case statements in F#

Note 4.5: Example in list 4.25 works, but it is not identical to the Erlang and Clean versions. In case of the nested case statements of this example the F# compiler handles pattern matching based on the indentions...

In example 4.26 the case statement is combined with a function that has two clauses. It sorts the elements of the input list of sum/2 based on their format in the list. If a certain element is a tuple, then it adds the first element in it to the sum (Sum). If it is not a tuple but a simple variable, then it simply adds it to the sum calculated up to this point. In every other case, when the following element of the list does not match the first two branches of the case, the element is ignored and not added to the sum.

Note 4.6: Mind that English terminology uses the expression clause for branches of functions and the expression branch for branches in case…
The two clauses of the function are used to ensure that summing will stop after processing the last element of the list and the result will be returned.

```
sum( Sum, [ Head | Tail ] ) ->
        case Head of
                { A, _} when is_ integer( A ) ->
                                sum( Sum + A, Tail );
                B when is_ integer( B ) ->
                                 sum( Sum + B, Tail );
                _ -> sum( Sum, Tail )
        end;
sum( Sum, [] ) ->
        Sum
```

Program 4.26. Using selections - Erlang

Exception handling

Similarly to other paradigms, handling of errors and exceptions caused by errors has a key role in functional languages. However, before starting to learn how to handle exceptions, it is worth thinking about what we call an error. Error is such an unwanted functioning of a program which you do not foresee when you write the program but can occur during execution (and most of the time it does occur, naturally not during testing but during presenting it…).
Note 4.7: Syntactic and semantic errors are filtered by the compiler in runtime, so these cannot cause exceptions. Exceptions are mainly triggered by IO operations, file and memory management and user data communication....
In case you are prepared for the error and handle it at its occurrence, then it is not an error any more but an exception that our program can handle.

```
try function / expression of
        Pattern [ when Guard1 ] -> block;
        Pattern [ when Guard2 ] -> block;
        …
catch
        Exceptiontype:P attern [ when ExGuard1 ] -> instructions;
        Exceptiontype:P attern [ when ExGuard2 ] -> instructions;
        …
```

```
after
        instructions
end
```

Program 4.27. Try-catch block - Erlang

```
let function [parameter, …] =
        try
                try instructions
                wit h
                        | :? Exceptiontype as ex
                                [ when Guard1 ] -> instructions
                        | :? Exceptiontype as ex
                                [ when Guard2 ] -> instructions
        finally
                instructions
```

Program 4.28. Exception handling, try-finally block in F#

One known means of handling exceptions is the use of functions with multiple clauses, another is the use of the exception handler of the particular language 4.27.

Functions with multiple clauses solve some errors, as you can see in program list 4.29 but their potential is limited. Function sum/1 can add two numbers received from a tuple or a list. In all other cases it returns 0 but stops with an error if the type of the input data is not proper (e.g. string or atom)

Note 4.8: This problem could be handled with introducing a guard condition or with adding newer and newer function clauses but it cannot be done infinitely. Maybe sooner or later you might find all the inputs which produce unwanted functioning but it is more probable that you could not think of every possibility and you would leave errors in the program even with high foresight. …

Exception handling is a much safer and more interesting solution than writing function clauses. The point of the technique is to place the program parts that possibly contain an error into an exception handling block and handle them with the instructions in the second part of the block when they occur.

Note 4.9: You can throw exceptions intentionally with the throw keyword but you must handle them by all means or otherwise the operating system will do so and that option is not too elegant…

```
sum ( { A, B } ) ->
        A + B;
sum ( [ A, B ] ) ->
        A + B;
sum( _ ) ->
        0.
```

Program 4.29. Variations of handling parameters - Erlang

Obviously, it is not sufficient to simply catch errors, you must handle them. Most of the time it is enough to catch the messages of the system and display them on the screen in a formatted way so that the programmer can correct them or to inform the user not to try to make the same mistake again and again (4.30).

```
display( Data ) ->
        try
                io:f ormat( "~ s~n", [ Data ] )
        catch
                H1:H2 ->
                io:fo rmat ( "H1:~w~n H2:~w~n Data:~w~n",
                                                [H1 , H2 , Data ] )
        end
```

Program 4.30. Finding the cause of an error in a catch block - Erlang

In program 4.30 function display/1 displays the text in its input on the screen. The type of the parameter is not defined but it can only display string type data due to the "~s" in the format function. With the use of exception handling the function will not stop even in case of a wrong parameter but it will show the error messages of the system that match pattern VAR1:VAR2.

Note 4.10: If the contents of the system messages are irrelevant, you can use pattern _:_ to catch them and thus the compiler will not call your attention to the fact that you do not use the variables in the pattern, despite they are assigned, at each run…

Exception handling in program 4.30 is equivalent to the one in program 4.31. In this version there is an expression or instruction that can cause an error after try, then comes keyword of. After of, just like in case, you can place patterns with which you can process the value of the expression in try.

```
try funct( Data ) of
        Value -> Value
        _ -> error1
catch _:_ -> error2
end
```

Program 4.31. Exception handling with patterns – Erlang

```
let divide x y =
        try
          Some( x / y )
        wit h
          | :? System.DivideByZeroException as ex ->
                printf n "Exception! %s " ( ex.Message ); None
```

Program 4.32. Exception handling example in F#

Keyword after is used when you want to run a particular part of a program by all means disregarding exception handling. The parts within the after block will be executed following either the flawless or flawy run of the instructions in the try block.

```
try
        raise Invalid Process ( “Raising_Exn" )
wit h
        | InvalidProcess( str ) -> printf "%s \n" str
```

Program 4.33. Throwing exceptions in F#

This mechanism can be used well in situations where resources (file, database, process, memory) must be closed or stopped even in case of an error

Complex Data

Sorted n-vectors

Tuple. The first unusual data structure is the sorted n-vector also called tuple. N-vectors can contain an arbitrary number of inhomogeneous elements (expression, function, practically anything). The order and number of elements in a tuple is bound after construction but they can contain an arbitrary number of elements.

Note 5.1: The restriction on the number of tuple elements is very similar to the restrictions used in the number of static array elements. Simply, they can contain an arbitrary but predefined number of elements. …

It is worth using the tuple data structure when a function has to return with more than one data and you have to send more than one data in a message. In such and similar situations data can be simply packed (program list 5.1).

– {A, B, A + B} is a sorted n-vector where the first two elements contain the two operands of the adder expression and the third element is the expression itself.

– {query, Function, Parameters} is a typical message in case of distributed programs where the first element is the type of the message, the second is the function to be executed and the third element contains the parameters of the function.

Note 5.2: The phenomena that the function seems to be a single variable will be explained in the section on calculus and higher order functions. This is a commonplace technique in functional languages and in systems supporting mobile codes....

– {list, [Head|Tail]} is a tuple that allows a list to be split. In fact it is not the tuple that splits the list into a first element and a list that contains the tail of the list. The tuple simply describes the data structure that contains the list pattern matching expression.

Let us have a look at program 5.1 that uses sorted n-vectors to display data.

```
-module( sum ).
-export ( [ sum / 2, test / 0 ] ).
        sum ( A, B ) ->
                { A, B, A + B }.

test( ) ->
        io:format ( "~d~n", [ sum ( 10, 20 ) ] ).
```

Program 5.1. Using Tuple in Erlang

```
module sum
import StdEnv

sum a b = ( a, b, a+b )

test = sum 10 20

Start = test
```

Program 5.2. Using Tuple in Clean

```
let sum a b = ( a, b, a + b )
```

Program 5.3. Creating a Tuple in F#

Function sum/2 not only returns the result but also the input parameters which are displayed on the screen by function test/0.
Let us observe that function sum/2 does not have a regular single element return value. Obviously, it is possible to define functions that return lists or other complex data types in other programming technologies as well, but data structures like tuple, where elements do not have a type, are very rare. Maybe an example to that is the set but there the order and number of elements is not bound and it is more difficult to use than sorted n-vectors. Another means of using tuple structure is when, for some reason, in clauses of functions with multiple clauses you wish to keep arity, namely the number of parameters. Such reasons can be error handling and the possibility of writing more universally useable functions.

```
-module( sum ).
-export( [ sum /1 ] ).

sum( { A, B } ) -> A + B;
sum( [ A, B ] ) -> A + B;
sum(_) ->
        0.
test( ) -> io:fo rmat( "~d~n~d~n", [ sum ( { 10, 20 } ), sum ( [ 10, 20 ] ) ] ).
```

Program 5.4. Tuple in multiple function clauses - Erlang

In program list 5.4 in this version of function sum/1 we can process data received both in tuple and list formats. The third clause of the function handles wrong parameters, (_) means that the function can receive any data that differs from the ones specified in its previous clauses.
In this version the function can only have multiple clauses if the arity does not change. Compile the program and run it with various test data. You can see that with data matching the first two clauses it returns the right value, but with any other parameterization it returns 0. As we have mentioned previously, in the section on exception handling, this is a remarkably simple and never-failing method of handling errors. In case of functions with multiple clauses it is common to use the last clause and the "_" parameter to secure the proper execution of the function under any circumstances.

Record

Record is a version of the tuple data structure with a name, endowed with library functions which help the access of data of fields, namely of records.

{person, Name, Age, Address}

Program 5.5. Named tuple – Erlang

In program list 5.6 we can see a typical record structure that, besides it name, has two fields. The record is similar to a tuple structure in which the first element of the tuple is an atom. If in this tuple, the first element is an atom, then the record matches the tuple and vice versa. This means that in the parameter list of functions one type matches the other as formal parameter and the other matches it as actual parameter. Records must be declared (program list 5.6), so at the beginning of modules a type must be introduced. There, you must give the name of the record and its fields. Unlike in tuples, the order of fields is not bound. Value assignment of fields in the record can occur separately. This mechanism is called record update (5.14).

-record(Name, { field1, field2, ..., field }).
-record(realestate, {price = 120000, county, location = "Eger" })

Program 5.6. Record – Erlang

```
:: Realestate = { price :: Int,
                  county :: String,
                     location :: String
                     }
realestate1 :: Realestate
realestate1 = { price = 120000,
                county = "Heves",
                   location = "Eger" }
```

Program 5.7. Record in Clean

```
type name = { field1 : type;
                  field : type;
               …   field : type }

type realestate = { price : int;
                    county : string;
                       location : string }
```

Program 5.8. Record definition in F#

In program 5.6 the first field of the second record definition, price has a default value, while field named county is not assigned. It is a routine method when defining records, since this way the fields, which must be assigned, are secured to get a value. Of course, the default values can be reassigned any time. Records can be bound into variables, as you can see it in program list 5.9 . In function rec1/0 record realestate is bound into variable X. The fields of the record with default values keep their value, while the fields which are not assigned during definition remain empty.

Note 5.3: Fields without a value are similar to the NULL value elements of records and lists used in OO languages…

In the assignment found in function rec2/0 we refresh the price field of the record and we assign a value to the field named county, thus this function illustrates both assignment and record update operations. In function rec3/3, which is a generalization of the previous ones, you can fill fields of the records with data received from the parameter list.

```
-module( rectest ).
-export( [ rec1 / 0, rec2 / 0, rec3 / 3 ] ).
-record( realestate, { price = 120000, county, location = "Ege r" } ).
```

```
rec1() -> R1 = #realestate{}.

rec2() ->
        R2 = #realestate{ price = 110000, county = "Heves" }.

rec3( Price, County, Location ) ->
        R3 = #realestate{ price = Price, county = County, location = Location }.
```

Program 5.9. Tuple and record - Erlang

```
type realestate = { price : int; county : string; location : s t r i ng }

letrec1 =
        let R1 = { price = 12000; county = "Heves"; location = "Eger" } R1

letrec2 price county location =
        letR2 = { price = price; county = county; location = location } R2
```

Program 5.10. Tuple and record - F#

Note 5.4.: Records make functions and data storage much more universal since with their help you can extend the number of parameters.
When the parameter of a function is a record, the number of parameters of the function is not bound. The record can be extended with new fields in the definition, thus extending the number of parameters…

```
-module ( rectest ).
-export( [ writerec / 1 ] ).
-record{ favorite, { name = "Bloki", owner } }.

writerec( R ) ->
        io:format( "~s", [ R#favorite.name ] ).
```

Program 5.11. Record update - Erlang

```
module rectest
```

```
import StdEnv

:: Owner = { oname :: String,
           age :: Int }
:: Favorite = { fname :: String, owner :: Owner }

owner1 :: Owner
owner1 = { oname = "An_Owner", age = 10 }

favorite1 :: Favorite
favorite1 = {fname = "Bodri", owner = owner1 }

rectest r = r.oname
Start = rectest favorite1.owner
```

Program 5.12. Record update - Clean

```
type favorite = {  name : s t r in g; owner : string }
let writerec ( r : favorite ) = printfn "%s" r.name
```

Program 5.13. Record update - F#

Naturally, fields of records can be referred to individually, as seen in program list 5.11.. Function writerec/1 gets a record as its parameter and displays its first field on the screen.

```
-module( rectest ).

-export( [ set / 2, caller / 0 ] ).

-record( realestate, { price, county , location } ).


set(# realestate{  price = Price, county = County } = R ) ->

        R#realestate{ location = Location}.


caller() -> set( { 1200000, "BAZ" }, "Miskolc" ).
```

Program 5.14. Record update - Erlang

Program list 5.14 belongs to the “magic” category even in the world of functional languages. The parameter list of function set/1 seemingly contains an assignment, but it is in reality a pattern matching, creating a record in function caller/0 and supplying it with data. The second parameter of function set/2 serves the purpose of enabling field location in the function body to be simply assigned.

Functions a Recursion

Writing functions

As we have mentioned earlier, we define functions and an initial expression in the modules of functional programs and start evaluation with the initial expression. Of course, this is not true to library modules, in which you want to make the call of several, or all of the functions, possible for the outside world. Since every functional language element can be traced back to functions, we have to be familiar with all the variants and their usage. Functions must have a name, a parameter list and a function body.

```
funName( Param1_1, …, Param1_N )
        when Guard1_1, …, Guard1_N ->
    FunBody1;
funName( Param2_1 , …, Param2_N )
        when Guard2_1, …, Guard2_N ->
    FunBody2 ;
…
funName( ParamK_1, ParamK_2, …, ParamK_N ) ->
    FunBodyK
```

Program 6.1. General definition of Erlang functions

```
function args
| guard1 = expression1
| guard2 = expression2
where
    function args = expression
```

Program 6.2. General definition of Clean functions

These parts are compulsory, but can be amended with new ones. Functions can have a guard criterion and multiple clauses. Branches are called clauses. Of course, clauses can have guards, too.

```
//Non-recursive
let funName Param1_1 ...
        Param1_N [ : return-ty pe ] = FunBody1

//Recursive
let rec funName Param2_1 …
        Param2_N [ : return-ty pe ] = FunBody2
```

Program 6.3. General definition F# functions

The function is identified with its name. The type of the name is atom, it starts with lowercase and it cannot contain space. A function can have multiple actual parameters, however, you can also write functions without parameters. In this case, you still need the brackets guarding the parameter list in Erlang, while in Clean and in F# there is no such restriction. The function body defines what to execute when the function is called. The body can contain one or multiple instructions, divided by a separator that is peculiar to the particular language. The separator is usually a comma (,), or a semicolon (;).

In Erlang, function bodies are closed with a '.', or in case of multiple clauses, clauses are separated with it.

```
f( P1, P2 ) ->
```

```
        P1 + P2.


g( X, Y ) ->
        f( X, Y ).


h () ->
        ok.
```

Program 6.4. Erlang functions

```
f p1 p2 = p1 + p2.
g x y = f x y
h = "ok"
```

Program 6.5. Clean functions

```
let f p1 p2 = p1 + p2
let g x y = f x y
let h = "ok"
```

Program 6.6. F# functions

Recursive functions

Recursion. Naturally, functions can call other functions or themselves. This phenomenon, when a function calls itself, is called recursion. recursion can be found in OO languages as well, but its use can cause several problems, since the number of recursive calls is rather

limited. In functional languages, the execution of functions, namely the stack management system, is totally different from the ones in the compilers of non-functional languages. If the last instruction of a function calls the function itself and that call is not in an expression, then the number of recursive calls is unlimited. If there are several recursive functions calling each other and all recursive calls are tail-recursive, then these calls do not ”consume” stack space. Most functional programming languages allow unlimited recursion and their evaluation is still Turing complete.

```
int n = 10;
do
{
    Console.WriteLine( "{0}", n )
    n--;
}
while( n >= 0 )
```

Program 6.7. Imperative do-while loop

Recursion is a very important tool, because there is no real alternative to the implementation of iteration in functional languages, since they do not have loops at all.

```
let f =
        let mutable n = 10
        while( n >= 0 ) do
                printfn "%i" n
                n <- n - 1
```

Program 6.8. Do-while loop in F#

Recursive iterations. In program 6.9 you can see how you can create iteration similar to the C# do-while loop shown in program 6.7. Function dowhile/1 has two clauses. The first clause reduces the value of the number it got as its parameter in each run. This way we approximate zero, until the second clause is executed returning the zero value. In the first clause the displaying instruction was introduced to be able to see the result of every single run. Normally, we do not use such iterations; the example merely illustrates the similarity of loops and recursive iteration. In practice, we rather use this kind of iteration structure for implementing busy waiting of servers and lists processing. The following sections will show examples of list management and how to create simple server applications.

```
-module( reciter ).

-export( [ dowhil e / 1 ] ).


dowhile( N ) ->

    N0 = N - 1,

    io:fo rmat( "~w~n", [ N0 ] ),

    repeat( N0 );

dowhile( 0 ) ->

    0.
```

Program 6.9. Iteration with recursion – Erlang

Recursion can be used for the implementation of basic control structures and programming theorems of imperative languages.

```
let rec do while n =
```

```
        match( n - 1 ) with
        | 0 -> printfn "0"
        | a -> printf n "%i " a
                dowhile( n - 1)
```

Program 6.10. Iteration with recursion - F#

This is a rather unique but not unprecedented approach to recursion, so let us show its implementations (program list 6.11).

```
sum = 0;
for( i = 0; i < max; i++ )
{
    sum += i;
}
```

Program 6.11. Sum in C

```
for( I, Max, Sum )
        when I < Max ->
    for( I + 1, Max, F, Sum + I );
for( I, Max, Sum ) ->
    Sum.
```

Program 6.12. Sum in Erlang

```
let sum i max =
    let mutable sum0 = 0
    for j = i to max do
        sum0 <- sum0 + j
    sum0
```

Program 6.13. Sum in F#

```
let rec sum i max =
    match i with
    | i when i < max -> ( sum ( i + 1 ) max ) + i
    | _ -> i
```

Program 6.14. Recursive sum in F#

Higher order functions. Higher order functions are probably the most interesting constructions in the toolkit of functional languages. With their help, we can pass function prototypes, in fact expressions, as parameters in the formal parameter list of functions. In Erlang it is also possible to pass functions bound into variables and to use variables constructed that way as functions.

Note 6.1.: The base of higher order functions and functional programming languages is Lambda-calculus...

Note 6.2.: The packaging and passing of higher order functions in messages sent to distant systems ensures such a dynamism for functional languages with messaging, that is rare to find in imperative systems.

We can send data and the functions that will process the data, packed in the same message, to another party or program through the network using shared memory.

Thus, we can fully generalize client-server applications enabling them to be parameterized with a function. In functional languages the meaning of the sentence beginning as ”Every function...” is revealed for those who have only worked in OO and imperative language environments so far.

```
caller() ->
    F1 = fun( X ) -> X + 1 end

    F2 = fun( X, inc ) -> X + 1;
         ( X, dec ) X - 1 end

F2( F1( 10 ), inc ).
```

Program 6.15. Function expressions - Erlang

```
let caller() =
    let f1 = ( fun x -> x + 1 )
    let f2 = ( fun x exp ->
                match exp with
                | "inc" -> x + 1
                | "dec" -> x - 1 )

f2 ( f1 10 ) "i nc"
```

Program 6.16. Function expressions - F#

In example 6.15 the first function is assigned to variable F1. In variable F2 we bind a function with multiple clauses, similarly to the above mentioned, which reduces or increases the value it got in its parameter by one, depending on whether the first parameter is the dec or the inc atom.

Note 6.3.: Higher order functions can have multiple clauses, which are executed if the parameter with which the function is called matches the particular clause. Clauses are separated with ; and they are selected with pattern matching …

Let us examine program list 6.15. Calling function caller/0 of the module will return 12, since F1/(10) increases the value to 11 and it is used in the call of F2(11, inc) which increases the value to 12

Note 6.4: Of course, this program does not require the use of higher order functions, in fact, the ”traditional” version is much more transparent.

The only reason why it is shown is that it expressively illustrates the use of this special language construction. Higher order functions can have other functions as their parameters – here we parameterize with function prototype - , or they can be placed in list expressions which is yet another interesting means of their use. Naturally, the use of list expressions will be discussed later on. Parameterization with functions helps us to develop completely general functions or even modules. Program 6.17 shows a simple example of this.

```
-module( lmd ).

-export( [ use / 0, funct /2 ] ).


funct( Fun, Data ) ->

    Fun( Data );

funct( Fun, DataList ) when is_list( DataList ) ->

    [ Fun( Data ) || Data <- DataList ].


use() ->

    List = [ 1, 2, 3, 4 ],
```

```
    funct( fun( X ) -> X + 1 end, List ),
    funct( fun( X ) -> X - 1, 2 ).
```

Program 6.17. Module of higher order functions - Erlang

```
// "use" is an occupied keyword, instead of it we use "funct"

let funct1 fn data = fn data

let funct2 fn data List =
    match dataList with
    | h :: t -> List.map fn dataList

let funct() =
    let List = [ 1; 2; 3; 4 ]
    ( funct2( fun x -> x + 1 ) List ), ( funct1 ( fun x -> x + 1 ) 2 )

//alternative solution:
let funct() =
    let List = [ 1; 2; 3; 4 ]
    funct2( fun x -> x + 1 ) List
    funct1( fun x -> x + 1 ) 2
```

Program 6.18. Module of higher order functions - F#

Higher order functions, just like ”simple” functions, can be nested together (example 6.20), and they can have multiple clauses as we have seen it with simple functions.

```
fun( fun( X ) -> X - 1 end ) -> X + 1 end.
```

Program 6.19. Nested functions - Erlang

```
let caller = ( fun x -> ( fun x -> x - 1 ) x + 1 )
```

Program 6.20. Nested functions - F#

Function expressions. Functions can be referred to with a pair derived from their name and arity. Such function expressions can be seen (program list 6.23) as a reference pointing to the function.

```
-module( funcall ).
-export( [ caller / 0, sum / 2 ] ).

sum( A, B ) ->
        A + B.
caller() ->
        F = fun sum / 2,
        F( 10, 20 ).
```

Program 6.21. Function reference - Erlang

```
summa a b = a+b


caller = f 10 20
        where f x y = summa y x / 2
```

Program 6.22. Function reference - Clean

In example program 6.23 function sum/2 is referred to in function caller/0 using a variable in which we bind the function and we can call the name of the variable with the right parameters.

```
let sum a b = a + b


let caller =
        letf = sum
        f 10 20
```

Program 6.23. Function reference - F#

This solution can also be used in parameter lists of functions, and in cases where we must select the function to be called from a set of functions. In order to fully understand the parallelism between the two versions, let us rewrite the previous (6.23) version, which used a function expression, in a way that we unroll the expression F = sum/2 in function caller/0.

```
-module( funexp ).
-export( [ caller1 / 0, caller2 / 0 ] ).
```

```
sum( A, B ) ->
        A + B.


caller1()->
        F = fun( V1, V2 ) ->
                f (V1 , V2) end,
        F( 10, 20 ).


caller2() ->
        sum( 10, 20 ).
```

Program 6.24. Unrolled function expression – Erlang

The first version of the function in example 6.24 we unrolled the function expression with a higher order function.

In the second version we simply called function sum/2. This is the simplest solution but its result and functioning is the same as the original one and as the first version after rewriting it. It is common in server applications that the server does not contain the programs to be executed. It gets functions and data in messages, executes the task and returns the value to the client. A big advantage of systems based on this generalized principle is that the server spares resources for the clients and they can work completely generally. Since, it gets the source code of the tasks to be executed, in form of higher order functions, from the clients, its source code can remain relatively simple no matter how complicated the task is. (program list 6.25).

```
loop( D ) ->
        receive
                { exec, From, Fun, Data } ->
```

```
                From ! Fun( Data );

        …

    end
```

Program 6.25. Calling a function with messaging - Erlang

In other cases the server contains the functions to be executed but the client decides which ones to run. Here the server is parameterized with the name-arity pair and it only gets the data that is necessary for the calculations. In such applications, function expressions and higher order functions in general are very useful, because there is no real alternative to simply send and execute mobile codes.

Lists and Set Expressions

List data structure is significantly more complicated than tuple, however, in return for its complexity it empowers the programmer with possibilities that no other data structure is capable of doing. List expression is a construction for creating lists, tied strongly to them. Its base is the Zermelo-Fraenkel set expression [3]. The list expression is in fact a generator, which, strangely, does not define elements of the list but gives the criteria of being part of it and regulates the number of elements and how they should be created.

V = {x |x X, x > 0} 2

Note 7.1: This construction is called list-comprehension or set expression. The first element of list-comprehension is the generator. In Hungarian the name list generator is used for the whole expression, so for easier understanding let us use the name list expression. …

With this technology we can define lists of infinite elements, since we do not give the elements of the list but its definition. Lists can be used in pattern matching or as return value of functions and they can appear in any part of a program where data can be used. Lists can be split (with pattern matching) to a head, the first element, and to a tail which is the rest of the list without the head. Lists can be defined with the general syntax shown in program list 7.1.

[E || Qualif ier_1, Qualifie r_2, …]

Program 7.1. List syntax - Erlang

Program text 7.2 shows how to define and handle static lists. This program part binds lists in variables and reuses them in other lists.

Data = [{ 10, 20 }, { 6,4 }, { 5, 2 }],

List = [A, B, C, D],

L = [Data, List] …

Program 7.2. Binding lists in variables – Erlang

```
let Data = [ ( 10, 20 ); ( 6, 4 ); ( 5, 2 ) ]

let List = [ A, B, C, D ]

let L = [ Data, List ]
```

Program 7.3. Binding lists in variables F#

There are several tools to create (generate) lists in functional languages. For this purpose we can use recursive functions, list expressions or the library functions of the given language.

Handling Static Lists

Processing lists. Every L list can be split to a Head element and the rest of the list the Tail. This decomposition and its recursive iteration on the second part of the list ([Head|T ail] = T ail) enables us to traverse the list recursively. Pattern matching in example 7.4 uses only one element from the beginning of the list but if iterated, it would match the current first element and it would sooner or later process the whole list. However, for the iteration and for the full processing of the list we would need a recursive function.

```
L = [ 1, 2, 3, 4, 5, 6 ],

[ Head | Tail ] = L …
```

Program 7.4. Pattern matching in lists – Erlang

```
let L = [ 1; 2; 3; 4; 5; 6 ]

match L with

| Head :: Tail -> …
```

Program 7.5. Pattern matching in lists - F#

Note 7.2: Notice that variable Head in program 7.4 contains a data (a number), while Tail contains a list. This is an important detail regarding the further processing of the list, as the beginning and the end of the list should be handled in a different way. …

So, the head of the list is always the actual first element and the end is always the list of the remaining elements which can be separated from the beginning and passed on for further processing at the next recursive call of the function with pattern matching.

```
listtraversal( Acc, [ H | T ] ) ->

        Acc0 = Acc + H,
```

```
        listtraversal( Acc0, T );
listtraversal( Acc, [] ) ->
        Acc.
```

Program 7.6. List traversal with recursive function - Erlang

```
listtraversal [h : t ] = h + listtraversal t
listtraversal [] = 0
```

Program 7.7. List traversal with recursive function - Clean

```
let rec lista traversal acc list =
        match list with
        | h :: t -> listtraversal ( acc + h ) t
        | [] -> acc
```

Program 7.8. List traversal with recursive function - F#

When reaching a single element list Head gets the only element of the list, while Tail gets the rest, namely an empty list. The empty list is handled by the second clause of the function stopping the recursive execution. In fact, the empty list is the base criterion of the recursion. List elements can be processed, summed or displayed arbitrarily in. Properly written recursive functions do not stop in functional languages, so they can process lists of indefinite length no matter whether we generate or traverse the list. In order to understand the point of recursive processing better, let us write this function summing the elements of a list containing arbitrary numbers.

```
-module( list1 ).
-export( [ sum / 2 ] ).

sum( Acc, [ Head | Tail ] ) ->
        Acc0 = Acc + Head
        sum( Acc0, Tail );
sum( Acc, [] ) ->
        Acc.
```

Program 7.9. Summing list elements – Erlang

```
module list1
import StdEnv
sum [ head : t a i l ] = head + sum tail
sum [] = 0
```

Program 7.10. Processing the list - Clean

```
let rec sum acc list =
        match list with
        | h :: t ->
                let mutable acc0 = acc + h
                sum acc0 t
        | [] -> acc
```

Program 7.11. Processing the list - F#

Function sum/2 in program 7.9 is a multiple clause function, regarding its structure. The task of the first clause is to split the first element of the list, it got as its parameter, from the rest of the list and bind it in variable Head. Then, it calls itself with the actual sum, which is the rest of the list.

The second clause stops the recursion when the list runs out of elements, namely when the actual (second) parameter is an empty list. We sign the empty list with the [] formula. The return value of the function is the sum of the numbers in the list, which is stored in the Acc and Acc0 variables during the recursive run.

Note 7.3.: We need variable Acc0 because ass Acc = Acc + H is a destructive assignment and as such it cannot be used in functional languages…

Note 7.4.: If the list given as parameter does not contain numbers, the function still works, but it must contain elements on which the + operator can be interpreted…

Let us create the module implementing function sum/2, compile it and call it from the command line (program list 7.12).

```
> c( list1 ).

> {ok, list1 }

> List = [ 1, 2, 3, 4 ].

> List.

> 1, 2, 3, 4

> list1 : sum( 0, List ).

> 10
```

Program 7.12. Running program sum - Erlang

```
modul list1
```

```
import StdEnv

sum [ he ad : tail ] = head + sumtail

sum [] = 0

L= [1, 2, 3, 4 ]

Start = sum L
```

Program 7.13. Running sum – Clean

```
val sum : int -> int list -> int

> let List = [ 1; 2; 3; 4 ];;

val List : int list = [ 1; 2; 3; 4 ]

> sum 0 List;;

val it : int = 10
```

Program 7.14. Running sum in F# interactive window

The program of sum implements a simple operation. Recursion runs on one clause and its return value is a single element. In order to comprehend recursive processing better, let us create some more complicated list handling applications.

```
-module( functions ).
```

```
-export ( [ sum / 2, max /1, avg /1 ] ).


sum ( Acc, [ Head | Tail ] ) ->
        Acc0 = Acc + Head,
        sum( Acc0, Tail );
    sum( Acc, [] ) ->
        Acc.


max( List ) ->
        lists : max( RList ).


avg ( List ) ->
        LList = [ Num || Num <- List,
                is_ integer(Num), Num =/= 0 ],
        NData = lists : foldl(
                fun( X, Sum ) -> X + Sum end, 0, List ),
        NData / length( LList ).
```

Program 7.15. Handling lists - Erlang

```
module functions
import StdEnv


sum [ he ad : tail ] = head + sum tail
sum [] = 0
```

```
maximum [ x ] = x

maximum [ head : tail ]

        | head > res = head

        | otherwise = res

                where res = maximum tail


average lst = toReal( foldl sum 0 lst )

                / toReal( length lst )

                where

                sum x s = x + s
```

Program 7.16. Handling lists – Clean

```
let rec sum acc list =

        match list w ith

        | h :: t ->

                let mutable acc0 = acc + h

                sum acc0 t

        | [] -> acc


let max list = List.max list


let avg( list : float list ) = List.average list
```

Program 7.17. Handling lists - F#

We can generate new lists with recursive functions or transform existing ones bound into a new variable. The parameter of functions carrying out such tasks can be a list (or a construction “producing elements”), and their return value can be another list containing the generated elements (program list 7.18).

```
-module( functions ).

-export( [ comp1 / 1, comp2 / 2, comp3 / 3 ] ).


comp1( { A, B, C } ) ->

        [ A, B,C ].


comp2( List, [ Head | Tail ] ) ->

        List1 = List ++ add( Head, 1),

        comp2( List1, Ta il );

comp2( List, [] ) ->

        List.


comp3( Fun, List, [ Head | Tail ] ) ->

        List1 = List ++ Fun( Head ),

        comp3( Fun, List1, Tail );

comp3(_, List, [] ) ->

        List.
```

Program 7.18. Generating and concatenating lists - Erlang

```
module functions

import StdEnv


comp1abc = [ a, b, c ]


comp2 [ head : tail ] = [ head + 1 ] ++ comp2 tail

comp2 [] = []


comp3 fun [ head : tail ] = fun [ head ] ++ comp3 fun tail

comp3 fun [] = []
```

Program 7.19. Generating and concatenating lists - Clean

Example module 7.18 contains three functions. The single line of comp1/2 puts the elements of the triplet passed as parameters into a list.

```
let comp1 ( a, b, c ) = [ a; b; c ]


let rec comp2 list1 list2 =
        match list2 with
        | head :: tail ->
                comp2 ( list1 @ [ head ] ) tail
        | [] -> list1
```

```
let rec comp3 fn list1 list2 =
        match list2 wit h
        | head :: t a i l -> comp3 fn ( list1 @ [ fn head ] ) tail
        | [] -> list1
```

Program 7.20. Generating and concatenating lists - F#

Function comp2/2 is typical recursive list processing with multiple clauses. The first clause increases the value of the first element in the list in the third parameter by one, then, puts it into the new list which is passed on for the next call. When the list is empty, it stops and returns the result list. Function comp3/3 works similar to function comp2/2. This function can be considered as the general version of comp3/3 as its first parameter is a function expression which is called with every element of the list. This version is more general because it can not only call a certain function with the elements of the list but any kind of a function. Program list 7.21 contains the implementation of the fully generalized version of the program. This example is in this section to show a common use of higher order functions.

```
-module( list3 ).
-export( [ comp3 / 3, use / 0 ] ).

comp3( Fun, List, [ Head | Tail ] ) ->
        List1 = List ++ Fun( Head ),
        comp3( Fun, List1, Tail );
comp3(_, List, [] ) ->
        List.

use() ->
        List = [ 1, 2, 3, 4 ],
```

```
comp3 ( f un( X ) -> X + 1 end, [], List ).
```

Program 7.21. Generalization of a function - Erlang

```
module list3

import StdEnv


comp3 funct [ head : tail ] = funct head
                              ++ comp3 funct tail
comp3 funct [] = []


use = comp3 plus list
      list = [ 1, 2, 3, 4 ]
      where plus n = [ n+1 ]
```

Program 7.22. List with a higher order function – Clean

We can see that the only role of function use/0 is to call function comp3/3 and generate the list.

```
let rec comp3 fn list1 list2 =
      match list2, fn with
      | head :: tail, fn -> comp3
            fn ( list1 @ [ fn head ] ) tail
      | [], _ -> list1
```

```
let funct =
        let List = [ 1; 2; 3; 4 ]
        comp3 ( fun x -> x + 1 ) [] List
```

Program 7.23. List with a higher order function - F#

If you want to run this program, compile it and call the use/0 function with the module qualifier (program list 7.24).

```
> c( list3 ).
> { list3, ok}
> list3:use().
> […]
```

Program 7.24. Calling use/0 - Erlang

```
module list3
import StdEnv

comp3 funct [ head : tail ] = funct head
++ comp3 funct tail
comp3 funct [] = []

use = comp3 plus list
```

```
        list = [ 1, 2, 3, 4 ]

        where plus n = [n + 1 ]


Start = use
```

Program 7.25. Calling use - Clean

If we examine programs 7.21 and 7.18 more closely, we can see that functions comp2/3 and comp3/3 return the same result if the same thing “happens” in function add as in the function expression, however, function comp3/3 can be used for multiple purposes as we can change the function’s first parameter arbitrarily.
Note 7.5.: Generalization of functions makes our programs more effective and more widely applicable. When there is chance, we should create as generally applicable functions as possible…

List expressions

In the toolkit of functional languages there are several interesting constructions, which we can not find in traditional imperative or OO languages. Such tool is the list expression, which can be used for processing as well as generating lists. List expression generates lists, or set expressions if you wish, dynamically in runtime. In practice this means that we do not give the elements of lists but the criteria based on which they should be generated. In theory, this dynamism enables the generation of infinite lists. In spite of all, most of the time list expressions generate new lists from existing ones (program list 7.26). We can apply functions in list expression to any element of the list in a way that there is no need for writing recursive functions; and we can concatenate or split lists even based on complex criteria.

```
FList = [ 1, 2, 3, 4, 5, 6 ],
Lista = [ SelectedItems | A <- FList, A /=  0 ] …
```

Program 7.26. List processing with list expression - Erlang

The list expression in example 7.26 selects the elements from the list with a value other than zero. The list expression consists of two parts. The first element generates the list, so it is the generator. The second gives a sequence of elements from which the new list is generated. (this list can be called source list). There is a third, optional, part where you can give criteria

for the processing of list elements. You can have multiple criteria and all of them affect the processing of the source list. The enlisted criteria are evaluated as if they had the AND operator in between.

```
let fList = [ 1; 2; 0; 3; 4; 5; 6 ]
let list = List.filter( f un a -> a <> 0 ) fList
```

Program 7.27. List processing with a list expression - F#

If you wanted to have multiple function calls or expressions in the generator part, you have to use keywords begin and end to create groups. Keywords begin and end form a block, in which instructions are executed sequentially (program list 7.28).

```
List = [ begin f1( Element ), f2( Element ) end
                || Element <- KList, Element >= 0, Element < 10 ]
```

Program 7.28. Use of begin-end block in a list expression - Erlang

```
let list =
        let kList = List.filter
                ( fun element -> element >= 0 && element < 10 ) kList
        let kList = List.map f1 kList
        let kList = List.map f2 kList
        kList
```

Program 7.29. Begin-end block - F# ”the closest solution”

In example 7.28 the result of the list expression is bound in a variable. The first element between [] defines the actual element of the new list in a begin-end block. We must place the expression defining the way the list is generated here. After the || signs, the actual element of the original list can be found or the expression that “unpacks” the element in case of more complex data types. The line is finished with the original list following <- and after another comma the criteria which regulate which elements can be part of the list. The criterion is in fact a filtering mechanism letting proper elements to be part of the new list, ”rejecting” the rest.

```
List2 = [ begin filter ( Element ), add( Element ) end ||
                Element <- List1, Element > 0, Element < 100 ]
```

Program 7.30. Complex criteria in a list expression – Erlang

We place two criteria in example program 7.30, which filter the elements of the input list between 1 and 100. The enlisted criteria is executed sequentially and is connected with AND.

```
let list2 = List.filter
        ( fun element -> ele ment > 0 && element < 100)
```

Program 7.31. Complex criteria in a list expression - F#

Note 7.6.: When applying criteria the length of the original list does not always match the length of the result list…
Based on the structure of the construction it is easy to imagine how list-comprehension works. We evaluate the elements of the original list one by one, and based on the criteria we add them to the new list applying the expression in the generator part.

```
listmaker() ->
        List1 = [ { 2, element1 }, 3, 0, { 4, { pos, 2 } } ],
        List2 = [ Add(Element, 1 ) || Element <- List1 ],
        List2.

add( Element, Value ) - >
        Element + Value;
add( { Element, _}, Value ) ->
        Data = { Element, _},
        Data + Value;
add(_ ) ->
        0.
```

Program 7.32. Generating a list with a function - Erlang

```
let add element value = element + value

let listmaker =
        let list1 = [ 1; 2; 3; 4; 5 ]
        let list2 = List.map
                ( fun element -> add ele ment 1 ) list1
```

```
        list2

//output:
val listmaker : int list = [ 2; 3; 4; 5; 6 ]
```

Program 7.33. List processing with a function - F# brief version

If you do not want to use begin-end block when generating, you can place expressions you would put in them in a function. Then, you can call the function to every element in the generator, as shown in program 7.32.

The elements of List1 are given, but their format is not proper (inhomogeneous). When processing we generate the new list from these elements by adding an arbitrary value to every element using function add/2. We built a little trick in function add/2. The function has two clauses, which is useful if the original list contains n-vectors and numbers in turns.

Note 7.7.: In practice it is not rare to get inhomogeneous data for processing. In theses cases you can use the overload technology known in OO languages as well, or you can use pattern matching to select elements…

So, when function add gets sorted n-vectors as parameter it “unpacks” the actual element and increases it with the number stored in variable Value. However, when it is called with a simple value parameter, it “merely” executes the addition. The third clause of the function returns zero if called with wrong parameters. If you properly write and run the program, then it generates the list you can see in text 7.34 based on the parameters in List2, since the first element of the original list is an n-vector containing a number and an atom. The second clause of add/2 “unpacks” the number and adds one to it resulting in number three.

```
[ 3, 4, 5 ]
```

Program 7.34. Output of program 7.32

The second element is generated simply by the first clause of the function in the generator increasing the value in the parameter with one. The third element of the original list is zero and it is simply left out due to the condition set in the generator. Finally, the third element of the new list is generated from the fourth element of the original one and its value is five.

Complex and nested lists

Nesting list expressions. List expressions, just like like lists can be nested together in arbitrary depth, but you must pay attention that too deep nesting makes our programs illegible and can slow down their run.

```
List1 = [ { one, 1}, { two, 2}, ..., { lot, 1231214124 } ],
List2 = [ Element + 1 || Element
                          <- [ E || { E, _ } <- List1 ] ]
```

Program 7.35. Nested list - Erlang

In program 7.35 the innermost list expression takes effect first. Then comes the next expression which handles the elements it gets as if they were the elements of a simple list.

Note 7.8.: A basic constructing element of functional languages is the list and the list expression, so probably all such languages have a library module which enables us to use list handling procedures through the functions of the language.

However, the decision how to generate, process and handle lists lies with the programmer. Many use list expressions while others trust list handling library functions. Neither is better than the other and most of the time the actual problem defines which way is easier to follow.

Industrial Use of Functional Languages

The spreading of functional languages in the industry and in telecommunication is not negligible. Erlang is used for creating communication and highly fault tolerant systems and Clean appears in several industrial projects. Systems carrying out mathematical calculations are programmed in functional languages and people involved in other branches of science, like physics and chemistry are also empowered to work quickly and effectively with them because mathematical formulas can be easily transcribed to the dialect of the particular programming language.

Creating client-server applications

To prove our earlier claims regarding servers and to show the possibilities of messaging and writing concurrent programs, we implement a server capable of sending and receiving messages in list 8.1.

```
-module( test_server ).
-export( [ loop / 0, call / 2, start /0 ] ).

start() ->
        spawn( fun loop / 0 ).

call( Pid, Request ) ->
        Pid ! { self() , Request},
         receive
            Data ->
                Data
        end.

loop() ->
        receive
            { From, hello } ->
                From ! hello,
                loop();
            { From, { Fun, Data } } ->
                From ! Fun( Data ),
                loop();
            { From, stop} ->
                From ! stopped;
            { From, Other} ->
```

```
            From ! {error , Other},
            loop()
    end.
```

Program 8.1. Code of a client-server application - Erlang

However, before starting to analyze the code, we must acquire the necessary background knowledge of running servers. Various server applications run on so-called node()-s. To simplify things node is a running application that can be addressed through its name. The nodes that communicate with each other can either run on distinct computers or on the same one. If you know the identifier of a particular node, you can send a message to it, or more precisely put, you can call functions implemented in its code. Calls in program list 8.2 address the node identified as node1@localhost. Variable Mod contains the name of the node's module, Fun contains the function in the module that we want to call, and variable Param contains the parameters of the function. If the function has no parameters, then an empty list should be passed instead of the parameters. Obviously, before using the server application, it must be launched with the spawn(Mod, Fun, Params) call, in which the variables contain the name of the module, the function to be executed and the proper parameters. Spawn runs nodes on separate threads which can be named.This name will be the identifier of the node. If you do not want to assign a name to it, you can also use its process ID, which is the return value of the spawn capable of identifying the running application.

```
> rpc : call( node1@localhost, Mod, Fun, [] ).
> rpc : call( node1 ( @localhost, Mod, Fun, Params ).
```

Program 8.2. Remote procedure call from command line - Erlang

Example program 8.1 shows the simplest way of messaging and launching processes. The server application shown here is capable of receiving data from clients, running functions it received to process data and returning the results of calculations to the process it got the request from. The loop/0 function of the server implements busy waiting and serves incoming requests, then calls itself recursively to be able to serve new requests.
Function start/0 launches the server by calling function spawn which ”launches” the server process. The return value of this function gives the process identifier of the server to the clients, namely its accessibility. When the server is launched, this value should be stored in a variable to be able to address the application any time. (program list 8.3).

```
> c( test_server ).
> { ok, test_ server }
> Pid = test_server : start().
> <0.58.0 >
```

Program 8.3. Launching the server - Erlang

Function loop/0 is capable of four things and it can be addressed by calling function call/2 which is also in the module (program list 8.4). In this example the server is parameterized with a function expression and a number. When requested, the server executes the function expression on the number and returns the result to the process that sent the request. This process is the Erlang node that sent the request, and its identifier is generated by function call/2 by calling function self/0. By the way, the result is returned by this function, too.

```
> test_server : call( Pid, { fun( X ) -> X + 2 end, 40 } ).
> 42
```

Program 8.4. Requesting the server - Erlang

Functions of loop/0 based on the parameters after launching the server:
– If {From, hello} tuple parameter is sent – where variable From contains the identifier of the requesting process – the answer is atom hello, which is immediately received by the requester.
– If the request contains data matching pattern {From, Fun, Params}, From is the identifier of the requester, Fun is the function expression to be executed and Params is the data that contains the parameter. The result is returned to the requester in this case also.
– The result of request {From, stop} is that the server does not call itself, namely it stops after returning atom stopped, informing the requester about the fact (program list 8.5).
– The last clause of the iteration is responsible for handling errors in runtime. Its job is solely to inform the requester about the error if none of the requests match the patterns and to call itself again (program list 8.6).

```
> test_server : call ( Pid, stop ).
> stopped
```

Program 8.5. Stopping the server – Erlang

```
> test_server : call( Pid, abc 123 ).
```

> { error, abc 12 3 }

Program 8.6. Handling wrong request - Erlang

Receive control structure implements waiting in function loop/0, while messaging is implemented with operator !, which has the identifier of the addressed on its left and the content of the message on its right. The message must only have one element that is why we use tuple data structure to pack the data. Patterns in the particular clauses can be seen as the protocol of the server application with which we can communicate with it. The server only understands data matching these patterns.

The code of the server application and the clients is placed in the same module. This technique is not unique in distributed programs, since this way both client and server computers can execute each others tasks if necessary.

Note 8.1.: Client-server programs are only illustrated in Erlang because the F# version is long and the part implementing distribution is not as functional as one would expect and Clean was not developed to implement distributed programs either…

After writing and testing the program we can think about the fact that in this language we can write client-server applications no matter how complex, FTP servers or even chat programs with a few lines of code and run them with using the least possible resources. This program can also be the basis for distributed running of complex programs executing calculations requiring a lot of resources, but you can also use the capacity of stronger computers to send functions and data to them and only process the returned result on weaker machines as shown earlier. As you could see in the sections above, functional languages can be used at almost any field of programming, let us consider data base management, writing servers or writing user programs of simple calculations. These languages have a rather strong power of expression and their various language elements empower the programmer with the freedom of creativity.

Functional Languages in Practice

The functionality of library modules in imperative languages and the functionality of OO programs organised in namespaces mean a set of functions grouped by their type, similarly the functionality of functional languages means that functions are organised in modules.
In Erlang functions are organised in modules, modules of functions are exported to be accessible from outside the module. This technology is similar to the classes of OO languages and their protection levels, private, public and protected. (In Erlang, functions which are not exported are local regarding the module. If we want to define global data, we create records.)

Besides all these, functional programing languages, especially Erlang, contain several special program constructions which can not be found in OO and imperative languages.

- Use of list expressions
- Strict or lazy expression evaluation
- Tail recursion
- Binding of variables (lack of destructive assignment)
- Lack of loop type iterations
- Referential transparency of functions
- Pattern matching
- Currying
- Higher order functions

The items of the enumeration above make functional languages different, these properties make them interesting or special. In the following pages you can find exercises almost to every section and you can get to know almost all of these properties through creating the solutions. The solutions of the exercises are only shown where inevitably necessary. To tell you the truth the exercises contain the solutions but only in a textual form (they just explain the solutions, it is up to the reader to exploit all the possibilities and benefit as much from them as possible…).

These exercises can be seen as a textual description before formal specification, which is an unavoidable tool in planning larger programs if you do not want to end up in a dead-end. It is done this way so that the beginner programer can learn how to analyze textual descriptions he gets from users, clients or, as it happens during most program developing, from the documentations of discussions with other programmers. Due to this method, the description of exercises is long, however, they supply thorough explanation and help to create the programs in each case.

Program developing in Erlang – settings of the development tool

In order to be able to create the following program in all three languages we must install and configure their development tools. Ad-hoc developments may work in smaller programs even if we write the code in an arbitrary text editor and compile it with a command line compiler and run the program in the command line of the operating system. However, in

case of well-considered developments, where a program plan, specification, an implementation study and several other important elements of the program are prepared, the use of a well-prepared program development environment is unavoidable.

Clean has a development tool, written in Clean, for writing programs (IDE – Integrated Development Environment) which, with its several extensions and debug function, can make the work of the programmer easier. In case of F# the situation is similar, since F# is among the programing languages that can be used with the rather professional Visual Studio tool. Thus, F# programmers can use all the high level services of this IDE. When writing Clean programs, you must pay attention that the development, text editor and runtime systems of the language operate in a rather unique way. When you start a new project, first, you must open or create the module file and only after that phase can you request a previously written or brand new project. Unless you do so, you will end up getting numerous error messages during compiling or running and your program will not run due to this. So, you must pay attention to order, but besides this you will not experience too many surprises.

In Visual Studio and in Clean IDE it is enough to activate menu items compile and run and the tool automatically saves, compiles and runs the program, and in case of errors it warns the programmer and shows the type and location of the error.

Using Erlang is a completely different matter. Erlang programs can be written with an arbitrary text editor and after compiling, you have multiple options to run them. By text editor we do not mean MS Word type of software. Not even Wordpad type editors are used for developing because they not only save the text into files but have their own formats and save the form definition tags and signs into the text as well. Due to this method they can be problematic regarding program development. By the way, too many text editing functions would only set the programmer back.

If you develop Erlang programs with this simple method, you must choose a simpler, easy to use tool lacking special formatting tags. NotePad can be a good choice. Save the programs prepared this way to a location where the Erlang compiler can access them or give the full path during compiling and ensure that the compiler has write privilege in the particular folder where it should create the (.beam) files. Erlang saves the compiled program text into files with .beam extension. An important momentum of compiling is that the character encoding of files must be proper. Use latin 1 encoding in Unix and ANSII in Windows or else the compiler throws a syntax error even in an otherwise completely right program text.

A much more effective way is to choose an editor program that supports scripting, namely one in which you can set the keyboard shortcuts to any executable file extensions. This enables us in case of Erlang programs to run the Erlang command line compiler with one keyboard shortcut and to compile the current file in focus. Unfortunately, most editors can only execute one instruction with activating one menu item or button but this problem can be solved with a well-written script or batch file. Obviously, we must choose a tool that is versatile and easy to configure. Emacs, which runs both in Unix and in Windows and has several facilities which make its use possible in any known programming language, can be a very good choice for Erlang programers. Emacs uses various methods, to which it can be taught with downloading and configuring pre-prepared plug-ins or with writing your own

ones in Elisp, to manage file types. Auto installing plug-ins, menus and other tools for editing and compiling C, C++ or even Latex and more importantly Erlang files can be found in Emacs. Download the version that can be run in our operating system and install it. Launch it when you are ready and observe immediately that the program does not react to files with erl extension at all.

In order for Emacs to get into Erlang mode you must do some minor settings. Emacs stores settings in a file named (.emacs). If you can not find it, you can easily create it in your user home folder or in Windows in the root of drive C. (mind privileges). If you already have an Emacs editor, you should use the existing (.emacs) file. Simply type the next few Lisp program lines into it:

```
((setq load-path (cons "C:/Program Files/erl5.7.5/lib/tools-2.6.5.1/emacs"
load-path))
(setq erlang-root-dir "C:/Program Files/erl5.7.5")
(setq exec-path (cons "C:/Program Files/erl5.7.5/bin" exec-path))
(require 'erlang-start)).
```

The first setq part in variable load-path tells Emacs where it can find the Erlang Emacs plug-in, the second one shows the Erlang folders and the third one defines the path required for running the binary files. These settings are compulsory to set up the development tool

If you do not have an Erlang runtime environment yet, download it from the erlang.org website and install it in a way that is proper in your operating system. In Unix systems you should search somewhere around folder (/usr/lib), in Windows search in folder Program Files under the name (erlangx.y.z), where x.y.z refers to the version, which is important because that is what you need in configuring Emacs. Within this folder there is a tool folder where you can find the Erlang extension of Emacs, namely the plug-in which is required by Erlang programs. The paths in the Lisp code must be changed to these.

When you are done, save the (.emacs) file and open a file with erl extension. If you have set everything right the menu should be extended with a new erlang menu item, where you can find several useful functions like syntax highlight or insert skeleton. This function inserts pre-defined controlling structures parameterized in the most general way into the code, helping beginner and advanced programmers as well. This menu helps beginners not to think about the syntax of language elements and spares advanced programmers a lot of typing. You can find codes of branches, simple instructions or even complete client-server applications here, which you are free to change after insert.

However, this is not the most important part of the new menu for us. Under insert there is a (start a new shell) and a (view shell) item (these open from the run menu by default). First, activate the first one. This divides the window of the editor to two parts. By, the way, windows here are called buffers and there is a separate menu to manage them. Our source code appears in the top buffer (if you have written it…), a command line which is similar to the terminal of Linux systems is shown in the bottom one. You can write instructions, (c(erlangfile)) where erlangfile is the name of the file you created but you must exclude the erl extension, which are necessary for compiling, here. Since Erlang knows the path to our files, you do not have to deal with it during compiling and running. When you compiled the

syntactically correct code, the compiler informs you about it in the command line: ({ok, erlangfile}). Now, you are ready to run the program by entering the name of the module, a colon and calling the function with the proper parametrization (e.g.: erlfile:f(10).), where f is a function with a single parameter and can be found in a module called erlfile (exported). You must export it to be able to call the functions from outside the module
If the program is executed, it informs us that we have done a good job. In the future, if you open Emacs with Erlang files, it will always operate as a development tool for Erlang.
To learn how to use Emacs can cause some trouble at first because the common keyboard shortcuts Ctrl X/C/V do not work here or they have a completely unexpected effect. In Emacs functions can be activated by pressing Alt X or Ctrl and other keys simultaneously. For example, to open files you must press the Ctrl + X + F keys. To copy something it is enough to simply select the text to be copied and use the Ctrl + Y keys. Cut does not have a shortcut, you can use this functions from the menu. In Erlang you use the Ctrl + C + K keys to compile programs immediately in a way that first you press the Ctrl and the C keys together, then release C and press K. If you do so, the same thing happens as if you activated the shell in the erlang menu and entered text (c(modulename).) with a full-stop in the end and hit enter. If you find this method too complicated, you should use the menu or find a software with similar functions which you find easier to use. Shortcut Ctrl + X + B is used to change between buffers
Many programmers use the pretty versatile IDE called Eclipse which is capable of developing Java, C, C++ and Erlang programs, among others, with applying the plug-ins developed for the particular language. Netbeans is similar to Eclipsehez and it is also appropriate for Erlang programmers and these two softwares can be used in a familiar way. (they are standard in Windows operating system and Ctrl X/C/V works).

To tell you the truth, it does not matter which tool you use for developing, the point is to be able to work effectively and quickly with them, using the least possible human, time and other resources even in case of bigger projects to accomplish our goals.

To make things easy, some steps of configuration and installation have been recorded to help the reader establish his own toolkit. If you still fail to succeed, the best solution is to look for solutions in Internet documentations and forums. Software developers and members of the Internet community are usually open and friendly and willing to help beginners. Do not be afraid to ask questions.

Implementing the first exercise

1. exercise

If your Emacs tool is ready, you can start editing, compiling and running a simple module. If you have doubts about initial success, you can watch the video tutorial of this phase, in which all steps are documented.
The first task is to launch Emacs. You can do so by activating either the emacs.exe in the program bin folder or the runemacs.exe file. Foreseeing future projects to come, let us create a shortcut icon placed on the desktop. If you have done that, open a new file with the menu of Emacs or with the Ctrl + X + F hotkeys and save it with .erl extension (in the menu or with Ctrl + X, Ctrl + S).

Write (-module()) to the beginning of the file and the name of the file inside the brackets. It is important that the file name is the same as the name of the module, otherwise you get an error message when compiling. By the way, if you leave that empty, Emacs asks you if you want it to fill the space with the right name. You should say yes. Then comes the export list with formula (-export([])). Here you must give the functions to be exported and to be accessible from outside the module. In this list you give the name and arity (the number of their parameters) of functions in the following form: (name/arity, or f/1), where f is the name of the function and 1 is the number of its parameters. Since export contains a list do not forget about the square brackets

The export list should be empty now, and you should concentrate on creating your functions. The first function to be written is named f and it has one parameter which is also the return value of the function: (f(A) -> A.). Mind that the name of the function is an atom so it can not start with capital letters but variables start with capitals since Erlang is a weakly typed language and it follows that protocol in identifying labels, functions, records, record field names and variables. The name of the function is followed by the list of parameters. You must write the brackets even if there is no formal parameter list (it is a custom form in C based languages). The name of the variable is (A). The parameter list is followed by (->) which introduces the function body. The body only contains the identifier of the variable in the formal parameter list and as it is the last instruction of the function it is also its return value. The function body is closed with a period.

If you are ready, press Ctrl + C, Ctrl + K (or find start a new shell in the Erlang men and in the shell type the (c(modulname)) call). Then your system compiles the module. If it finds an error, it informs you in the command line, if the source text is flawless, you see the following sorted couple: ({ok, modulname}).

Now, you can calmly run the the program. Enter the following function call in the command line: (modulname:f(10).), where modulname is the name of your compiled module and following the colon comes the name of the function and in brackets its actual parameter list matching the formal one perfectly. The line is closed by a period and on enter the result is displayed.

Now back to the source code. If it is not in focus, find it in the buffer list (menu) or with hotkeys Ctrl + X + B and arrows. Let us write another function in the module calling the first and increasing the return value with a constant. You must also export this function just like the first one. You only have to expand the export list with the name of the new function and its arity separated with a slash to do so. You can also write a new export list but it is not necessary at this point. Name the new function g. G also has a parameter and it contains the (f(B) + B) expression in its body. In order to be able to try more instructions in the body, you must bind the return value of f/1 in an arbitrary variable and this value is added to the value of the actual parameter passed in g/1. At this point the body looks like this: (C = f(B), F + B). You bind the return value of f/1 which is derived from the parameter of g/1 and the result is added to the value passed in the parameter of g/1. Since addition is the last expression of the function it returns this result. After modifying the program, you must compile the source text again to place the modification in the (.beam) file (it is the same as with the (.exe) files compiled from the source codes of any other programming languages). Run the new function: (modulname:g(10).), which results in twenty due to the addition.

Source code of the program:

```
-module(test1).
```

```
-export([f/1, g/1]).
   f(A) -> A.

g(B) ->
   C = f(B),
   C + B.
```

By following this method you can write Erlang programs of arbitrary complexity. The phases are the same, only the source text, module and function names differ. If you stuck at any point, use the help files of the Erlang runtime environment (you can find the manual in one of the folders of Erlang, usually in a folder called doc, depending on the version) or navigate to the official website of Erlang, where you can find full reference of the language.

Media help to our solution

Videos have been recorded and pictures have been taken illustrating the solutions, configurations and installations of this section

The content of the video: the attached video material shows the download of the tools used in this section, the installation of Erlang runtime environment (video/video1.avi, and video4.avi), the installation and configuration of Emacs (video/video3.avi and video4.avi), the running of Clean programs (video/video2.avi)

The pictures of the appendix show how to implement the example programs and exercises in the appendix. The first picture always shows the source code in the text editor, the second shows the output of the program in runtime in order to help the reader in all the phases of solving a problem. The steps are in the appendix of the book, so they are not necessarily in order.

Pictures

Practice exercises

1. exercise

Let us create an Erlang program that displays Hello World on the screen. For the solution we use the io:format function and place the following elements in its format string: ”~s~n”. The ~s marks the format of string and ~n marks the line break. As you could see earlier function io:format expects a list as its parameter and displays the elements of the list on the screen having the right formatting set in the format string. If the number of parameters is wrong, you get an error message while compiling. The IO module contains several other functions as well for handling input and output. In Erlang programs, however, console input is not typical because Erlang is not a language designed for creating GUI (graphical user interface). Low level Erlang routines almost always have interface functions which are called by a graphical

user interface written in some other language and their task is to ensure proper data exchange between the two languages and the services of the IO module are used for proper data conversion.

Function FORMAT (deliberately with capital letters to highlight it within the sentence, but obviously in programs it is used normally) has many services which can be used for converting data to string. You can transform lists, sorted n-vectors, atoms and numbers to a form extended with textual explanation. There are two different ways to do this exercise. The first option is not to use a format string in the display instruction, since the text ”Hello World” is string type by default and it can be displayed without formatting. The second option is to bind the text in a variable and format it with ~s shown earlier and pass it in the second parameter of the displaying function. No matter which option you choose, it is worth using the ~n formatting at the end of the line so that the next display would start in the next line after the line break. If you omit that, the displays appear in one line.

If you use an atom type instead of a string in the display, you only need to change the formatting to be proper. In this type the single parameter version of the format can not be used because it only works with the string type. If you cannot decide on the type when you want to display something, then you still do not have to find another displaying instruction. This can happen when the type of the parameter is revealed in runtime. Since, Erlang is a weakly typed language it is common that the concrete type is revealed only when the actual parameters are passed to the formal parameter list. If the parameterization of the function is such and you want to display the results of operations you can use ~w formatting as the first parameter of format which can be used with any type but its use makes string type hardly legible. It happens because string type is in fact a list and the function sees this data as a real list resulting in a list of numbers when displayed. If you want to convert string to character, this solution is ideal.

Keep in mind that when IO:format is the only or last parameter of a function it causes a side effect, namely the side effect is the display and its return value is the ok atom, since that is the return value of function format. Let us create the Clean and F# versions of the function based on the Erlang one. Naturally, you must also find the proper language variants of IO:format to be able to do the exercise. In these solutions you need other displaying instructions with other formatting capabilities. In F# it is an easy task for those who know C# language because the console works the same in F# programs as in OO variants. In Clean the language has its own IDE and it offers numerous possibilities, or at least as many as the two languages above, to display data

2. exercise

In the following exercises you can practice displaying on a console and they introduce the programmer to managing and displaying complex data as well. The parts of this exercise are worth being implemented in the same module using one or more functions in each of them. The first task is to write a program which selects the lowest and second lowest element of an arbitrary set. For the solution you should use the list data structure, thus you will not have to deal with generating the input data. If you want to work elegantly, you should write a function which returns a list and pass it as a parameter for other functions in the module. The list created that way can be replaced any time by changing the body of the function. If you wish to use a simpler solution, then after creating the module, you should bind the list, used for testing, to a variable with an arbitrary name in runtime. The particular exercise is to traverse the list, passed as a parameter, with a recursive function and bind the actual lowest element in a variable declared for this purpose and pass it on for the next call.

If during recursion, you find a lower element than the actual lowest element it should be put into the second lowest category. In practice it means that this element should also be passed for the next call. This sequence must be iterated with every element of the list. When the list, traversed with pattern matching, is empty, namely the function clause with the empty list in its parameter list is coming up, the variables with the lowest and second lowest elements should be displayed or simply returned as the return value of the function.

Let us write a program which displays the multiplication table on screen. Obviously by screen we mean the console window of Erlang where formatting can only be solved with tricks of the io:format function. To get the multiplication table, you need to generate a sequence of numbers where the actual element is the product of two elements. The first factor is the imaginary row index which is increased by one at each call of the iteration and the second factor is the column index which must be increased from 1 to 10 with each row index. So the first element is 1*1, the second is 1*2, then 1*3, and so on. When the second factor reaches 10 the first should be increased by one and you can start the next sequence. This operation is easy to implement in imperative languages, since all that you need are two nested for loops, but in functional languages the lack of loops completely rules out this solution.

The only tool for implementing iteration is recursion (unless you consider list generators). Recursive solution should be implemented in a way that the rows and columns are given with separate functions (as a beginner you should stick to this solution). The version using a list generator is somewhat easier, since the problem can be solved with embedding two lists together in this solution. While the outer list generator gives an element, the inner one generates elements from one to ten. In fact, you get the best solution by combining the two, the solution using function and the solution using list generator. This time you use a function in the list generator, which counts from 1 to 10 recursively and returns the elements one by one or in a list for the list generator which uses them to create the products. In order for the

display to have the matrix like format of the multiplication table you must place a line break at the end of each sequence (1*1, 1*2, … n*n+1) which involves the use of function IO:FORMAT.

Let us create a function in the module which gets an integer as its parameter and displays the equivalent day of the week on the screen. Value 1 means Monday, value 2 means Tuesday and value 7 means Sunday. For implementing the function you need the case control structure which can divide the function to multiple clauses based on the result of a particular expression. The solution is simple. A number between one and seven is the actual parameter of the function (let us call it N-nek, where N = (1..7), and the type of N is integer). Using the number as the expression of case you have to create seven plus one branches. The first seven branches displays the day of the week that belongs to the number using one of the versions of format. The last branch is required because the function can get a value other than N, and in that case the user, namely the user interface, must be informed about the error. You can make the function even more general and more fault tolerant by defining the type in a guard expression in the header (when is_integer(N)), and the interval (and N > 0, N< 8). Another way of handling exceptions here can be the use of an exception handler block which supplies serious protection against any kind of error.

The following function of the module should be able to select the minimum or maximum element of an arbitrary set and display it on the console screen. The solution is rather simple considering the previous exercises The task differs from the classic minimum and maximum selection only in that it gets which element to return as its parameter. Its first parameter is the min or max atom which informs the function as a label which operation to execute (in fact, it only gives the direction of the relation in the condition). To implement this function you can use the case control structure or a multiple clause function where it is easy to change the direction of the relation exploiting the properties of pattern matching or overload type function calls. The first clause of the function contains pattern matching which splits a list to a first element and the rest of the list. We decide whether the actual element is bigger or smaller than the previously stored one (initially you compare the first element with itself, but you can easily rule this efficiency problem out by not evaluating the first element and supposing that it is the smallest and start evaluating from the second element). If you find an element that is smaller than the previous one, you change it (pass it to the next recursive call).

Based on the first parameter of the function you must do the same in case of selecting the biggest element. The parameter of the second clause of the function is not important as its value is not used at this phase (If you want to display whether you selected the minimum or maximum element then it is not entirely true). So here the underscore key can be used which is the equivalent of does not matter in Erlang programs.The second parameter is an

empty list which is to sign that the list passed as a parameter has been completely split and we got to the empty list. You must stop at that phase and display the result for the user. This result can be the min or max element if you managed to create a general function without side effects but it can also be the displaying of the last instruction of the function. (in this latter case we are not talking about a real function but rather a method used in imperative languages).

You should try out all the functions listed in this exercise and obviously you must compile and run the module in the Erlang command line for this purpose. Where necessary, use constant parameters or create the right data in the command line because asking for data in the console is not too easy.

3. exercise

Write a program that generates integer numbers until their sum exceeds 100. After finishing input, you should display how many of the numbers were even and how many were odd. To solve the task you should use a list generator or a data storer structure that supplies the integer numbers. Just like in any other programing language you can generate random numbers in Erlang with the rand library module. The random value can be stored in a variable or can be immediately added to the variable used as accumulator which is passed in each phase of the recursion. The task can be solved most effectively with recursion. The recursive function will have two clauses. The first clause runs with every value less than a hundred and it calls itself in a way that it adds the actual randomly generated number to the previous sum (initially the value of its first parameter is zero) then it calls itself with the new value.

The situation with the second clause is a little bit more complicated because you should use the value 100 in the pattern matching and it would only be true if the sum could not exceed 100. This however would end up in infinite recursion (it is almost possible with functional, tail recursive function). It is true that the solution is not trivial but it is far from being impossible. You do not necessarily use an integer type parameter. It is a good solution for the summing clause to call itself with the atom ‘less’ after generating the random number and after summing or else it passes atom ‘more’ for the next call which results in the run of the second clause. This way the proper functioning of pattern matching is solved with the introduction of a simple selection and all the nuisances of exceeding the value 100 are ruled out. Besides pattern matching you can also use case control structure to stop recursion and display the result on the screen. The implementation of this solution is up to the dear reader.

4. exercise

Write the well-known program giving the value of n factorial in Erlang, Clean and in F#. The output of the program should appear on the console screen in all three cases and it should not contain anything else besides the calculated value in the given exercise. To calculate the factorial of n you have to enumerate the numbers and multiply them. The task is absolutely recursive and if you had any doubts just examine the mathematical definition of the factorial function which is also recursive. Obviously there is an iterative solution to the problem as well, but it is not an option in a functional language due to the complete lack of iteration language primitives. The solution is a function with two clauses in which the clauses differ in that the first runs in case of value zero, the second runs with any N value that is not zero and multiplies it with the value of N-1. In fact, it evaluates the expression N * fact(N-1) where fact(N - 1) is the recursive call of the function itself. As the value is decreased by one with every call the zero value of the actual parameter is reached soon and the first clause is called giving us the result.

The problem can be solved with this method but it has a drawback of having the recursive call in an expression so the function is not tail recursive which is a problem because this way it needs a stack to store the actually calculated data. The good solution is to somehow pass the (N * …) expression in the parameter list of the function extending the program with a new clause or with another function.

5. exercise

Let us suppose that there is a completely separated tunnel which is L units long. There is a mouse at both ends of the tunnel. To a signal one of the mice starts running with U speed, the other one with V speed, towards the other end of the tunnel. When they reach it, they turn and run facing each other again (they run from wall to wall, if you do not like mice, you can use bugs or dogs but then do not run them in a tunnel because they can get stuck.). The running creatures can meet in three ways. Sometimes face to face and sometimes the faster will catch up on the slower one or sometimes they get to the wall at the same time. Create a program that can display, using constant data, how many times the animals meet in a given period. Implement the program as a function of a module to be able to try it out. One possible solution is to use a list generator to simulate the run or you can write the function body as a simple expression. The times of meeting, namely the constants where a meeting happens can be stored in a list and they are returned by the function if the length of the tunnel and the period make it possible. If you are smart, the list generator can be substituted with two clauses of a recursive function. You need two clauses for the recursion to stop and for the result to be displayed.

6. exercise

Write a program that converts an arbitrary base 10 number to base 2. The number to be converted should be read from the keyboard and the result should be displayed. The conversion can be most easily carried out if you divide the number with the base of the number system, namely 2.

The remainder of the division is a number in base 2 system and the whole number result must be divided again and again until the remainder is zero. Create the program in a module named (sr) and export the function named (convert) to be able to call it after compilation. Convert must get two parameters. The first is the base of the number system and the second is the number to be converted. Conversion can be implemented with the iteration of recursive calls in a way that with each call you get the remainder and the whole number result of the division. The remainder is stored in a list created for this purpose or it is concatenated to a string. If you convert to a higher base system, you must pay attention that characters are used instead of numbers. To convert the appropriate number character couple use conversion instructions placed in a case control structure or create a function which is also implemented in the module for conversion. Recursion should stop if the result of the division is zero regarding the whole number result. Wherever it stops, there is nothing else to do but display the result.

7. exercise

Write an Erlang module which implements min, max, sum, and avg, operations. The operations, where possible, should work with sorted n-vectors, lists and two variables. The example below shows a possible multiple clause implementation of function sum.

```
sum({A, B}) ->
   A + B;
sum([A| B]) ->
   sum(A, B);
sum(Acc, [A| B]) ->
   sum(Acc + A, B);
sum(Acc, []) ->
   Acc.
```

The sum/1 is a function where 1 is the number of its parameters (arity). This is because the number of parameters of the clauses handling lists and the ones used with sorted n-vectors must be the same. Functions with the same name but different number of parameters do

not count as clauses of the same function. The first clause of sum gets two numbers in a sorted n-vector and returns their sum. In this example you can see that multiple clause functions can not only be used in recursive calls but also with “traditional” overload operations. The second clause contains a pattern matching as its formal parameter, which pattern splits the list in the actual parameter list to a first element and to the rest of the list, where the first element is a scalar type and the second is a list that can be used again as actual parameter to a clause of the function. This second clause calls a function with the same name but with two parameters in its body. This function is also called sum but its arity is two, so it is not the same function and it must be listed separately in the export list if you wish to access it from outside the module. The function gets the first element of the previously split list and the rest of the list which it splits further using pattern matching but meanwhile, it calls itself with the sum of the original first element and the previously split one. When the list is empty, when it runs out of elements to be summed, the second clause of sum/2 is called returning the result.

Based on the sum function you can write the avg function which returns the average of all the elements, or you can use sum/1 to execute the summing and implement a division in avg to get the result. However, for the division you need the number of elements in the list which you can get in two ways. The first solution is to count recursive calls but to do so you have to change the sum/1 function which is not a good idea if you intend to keep its original functionality. The second solution is to call length/1 function in avg to get the number of elements in the list it got as a parameter then to divide the result of sum/1 with this value. So avg/1 is not too complicated. For the elegant solution, let us create the local, self-implemented version of function length in the module. For that you only need to traverse the list recursively and increase the value of the register, which is initially set to zero, by one in each step. While counting, pay attention to the fact that functional languages lack destructive assignment just as they lack loops. Instead you should apply the accumulating solution you saw in the parametrization of sum/1. However, here you store the number instead of the sum. The implementation of functions min and max is up to the reader. To try out the module you should create a function (do not forget to export it), which returns a list containing numbers, to reduce testing time.

8. exercise

Write an Erlang program which selects integer type elements from the list below and sums them in a variable used as an accumulator. The list can contain integer type and sorted n-vectors, whose first element is an atom and the second is int, in turns, but it can not contain nested lists only one level deep ones. With these lists you should use the previously created function summing list elements, thus you can add an integer type element to the actual sum. The sublists in the list can only contain integer data. The list can contain elements other than

the above mentioned ones, namely atoms. Obviously, these elements cannot be added to the sum due to the difference in their type.

[{apple, 1}, 2, apple, {apple, apple}, {pear, 3}, 4,5, 3, {walnut, 6}, [1,2,3,4], 5].

The result of the summing of this list used for testing is 39. The summing function must have multiple clauses, since the list must be traversed recursively. To traverse the list you can place a case control structure in the first clause which can select the proper elements via pattern matching and can reveal the integer type elements in the complex data structures. Pattern matching contains the following patterns: ({E1, E2} when is_integer(E2)), which pattern gets the second element, integer type elements only, from a two element tuple. Sorted n-vector {apple, apple} does not match the pattern. The following pattern is: (A when is_integer(A)) which is matched by integer type elements. Of course, the guard condition is not part of any of the patterns but can make their effect stricter. The next pattern filters the (not nested) list type data on which you must call function sum/1 to be able to add the result to the sum. This clause looks like the following: (L when is_list(L) -> Acc1 = Acc + sum(L);), where Acc0 is the variable of the particular clause used for summing and Acc contains the value of the sum passed from the previous call. You need two variables because of the lack of destructive assignment. This clause stops the execution of the function with an error in case of a nested list. To avoid that, you must somehow prevent sublists to be passed as parameters for sum/2 or you must catch the error with an exception handler. This second solution might be somewhat simpler. You must place a default branch in case, a universal pattern which will catch any data that has a different type or form from the ones above. Unlike in the other branches you can not use a guard here, and you should use the underscore symbol which matches any pattern. It can be seen as the Joker card in card games, so underscore takes all. In this clause (_ -> …) the function calls itself but does not increase the value of the sum. This clause is the equivalent of SKIP (empty) instruction. In the second clause of the function, which runs in case of an empty list, you only have to display the result of the sum. This exercise is very interesting and you can learn a great deal while writing it. The solution involves list handling, shows pattern matching in functions and in case control structure, explains recursion and the use of multiple clauses. We highly recommend you to write it.

9. exercise

Create an application which calculates the following sum to any K positive integer constants: 1*2 + 2*3 + 3*4 + … + K* K+1. The program should be written in Erlang, Clean and in F#. The solution is very similar to the one of the factorial function but here, besides multiplication you need to implement addition as well. The similarity suggests that this problem should also be solved with recursive functions. Since you do not have to use a list, the stopping

condition in pattern matching can not be an empty list. The parameter of the first clause of the function is the K number which is decreased by one in each step and added to the actual sum to produce the required sequence. As you do not have to display the results of calculations in each step, including the actual value of K it does not matter whether you do it from 1 to 10 or vica versa If you needed the partial results, you could store them in a list or could calculate them using the original and actual value of K in every step (K - K'). So the first clause of the function gets the actual K and the actual sum (Acc), then produces the (K * K + 1) value and adds it to Acc. Due to the lack of destructive assignment it is simpler to use the multiplying and adding expression in the actual parameter list of the next call (func(Acc + (K* (K+1))…). The task of the second clause is only to return and display the result after reaching the empty list parameter. Here, the second solution, the display of the result as return value is better, since functional languages try to avoid side effects. Side effect is an event when the function not only returns the result but executes other IO operations as well

In imperative languages functions with side effects are called methods, functions without side effects are called functions. In order to display the return value in a formatted way you can use a display function, implemented in the module, enlisted in the export list, which stylishly displays the results of functions if the io:format is parameterized correctly

10. exercise

Write the tail-recursive version of the factorial exercise as well. The original program can be changed by changing the functions of the factorial program and adding a new clause to them. The condition of tail-recursion is that the last instruction of the function calls the function itself and this call should not be part of an expression as (N * fact(N – 1)). Instead, formula (fact(N-1, N*X)) should be used where X is the previous partial sum. In this case our program functions properly even with high values because the unique runtime environment of functional languages makes it possible with high N values, too. In fact the limit of the number of recursive calls is the highest value you can store in a variable. The calculation of the factorial is similar to the previous one except for the changes mentioned above, and it can be familiar to everyone from math lessons. The value of N is decreased while the value of the product is increased until N reaches zero. Then you stop and display the result, namely the value of n!.

11. exercise

Recursion is an important element of functional programming languages. Practicing it is unavoidable, so let us create a module which implements known programming problems recursively. Basic programming problems should be applied on list data structure, since this solution comes handy in storing sequences for example: sorting or selecting the maximum is implemented on a list data structure not on an array. We write those known problems in this

module which we have not implemented previously, so we skip the selection of maximum and minimum here. First we should create the decision problem. If you have already learnt about it, think back how you had to solve the problem of decision on imperative languages, but if you have not, then pay focus on the steps of the solution. The decision problem examines an arbitrary vector of N elements and decides whether it contains an element (or elements) with a given attribute. The result is always a yes or no decision. This is the classical solution which we have to change slightly due to the characteristics of Erlang. The principle is the same but the vector is changed to a list. List tr aversion is changed to recursion and effectiveness, that we can stop when finding the first T attribute element, is not yet considered. This can be added to our program later.

First, we need to write a recursive function, with two clauses, which traverses the list and in case of an empty list it displays the result. The list should be split with pattern matching, as usual, to a first element and to the rest of the list. Examining the first element you can tell whether it has a T attribute or not. At this phase you can immediately run into two problems. The first problem is which language elements to use to define the T attribute. The second is how to pass the result of the decision and what this decision means and how it affects the result of the final decision and how to write it in Erlang or in other functional programing languages.

The solution to the first problem can be to wire the decision, namely to write an if statement to a constant attribute but this requires the writing of a new function to each attribute. This solution is not too general. For a general solution you must use a higher order function in which you define the evaluation of the required attribute and this should be passed as a function parameter of the function implementing the decision, which now has one more parameter. This is the right solution and you only need to write the function expression defining the given attribute to solve the actual problem.

So the function must have three parameters, the first is the list (with the elements to be evaluated), the second is your D decision variable, the last one is the element containing the function that defines the attribute. The first clause of the function evaluates if the attribute is matched by the first element split in that call, then, calls itself with the rest of the list placing the true or false value of the decision in D. The second clause of the function is called with the empty list and returns the result of the final decision. Based on all of this, the solution to the second problem is very simple because at the first run of the function you just have to place the false value in D and in the second clause you have to check if it changes. Of course, once the value is true you can not let it change to false again. At this point you can introduce the restriction regarding effectiveness. You have several options to do so. For example if the value of D is once changed to true, no further evaluation is necessary, but you simply extend the second clause with a new one so that in case of true

this new clause must run no matter how many elements remain in the list. The solving of the problem is up to the dear reader. Do the same with the other programming problems as well. Prepare a textual description or even a specification with one of the solutions to the problem and start implementation only after deliberate planning.

12. exercise

In functions with multiple clauses a so called default clause can be introduced which works with any actual parameter or at least produces an acceptable result. The solution with a default clause can not always be implemented and it does not work with functions with side effect. To avoid this problem and to learn the technology, create a sum function which works properly with one, two or three parameters, then, introduce a clause in this function which works with any number of parameters. It should work using N parameters (where (N > 3)). The function should return the atom: too_many_parameters. To format the result of the function properly, create a function that formats the output and the result properly no matter which clause is executed. Do not use case in this solution. Unfortunately, if you do not use complex data structures like tuple (sorted n-vector), the numbers of parameters will not be the same in every clause and the compiler stops with an error message. To solve this problem, you should pack the formal parameter list in sorted n-vectors in all of the clauses. The parameter of the first clause is: (sum({A}) ->). All the others are similar, except for the clause with N parameters (sum({A, B}), sum({A, B, C})). The default clause should be used with a simple variable parameter. This clause must be put after the others, or else its parameter would swallow all the other clauses, since this clause would be matched by any actual parameter list. (sum(A)).

Unfortunately, this solution does not work properly with list data type because its summing with scalar type is impossible. If you want to prepare for this possibility, you must use an exception handler block or plan solving the task with list data structure from the beginning. In the second part of the exercise you have to plan the display in a way that it should work with a single atom or an integer return value as well. This also requires the implementation of a multiple clause function. The first or even both clauses can be used with a guard which filters the parameter list it gets, even in case of attributes that can not be decided with pattern matching. The first clause can have a form like (display(A) when is_integer(A)), the second can look like (display(A) when is_atom(A)). If you only had two different types of data, one guard would be enough, but it is not a problem if you prepare for more types of output from the beginning and create a proper clause for all of them and place a default clause at the end of the function. If the functionality of decision is ready, the only thing left is to format the display instruction the right way in each clause and to create the format string to the particular data type.

13. exercise

An important part of programs with user interface is error handling and displaying the error messages on screen paralytically. To practice error handling, create a multiple clause function that is able to handle and display special error messages of an arbitrary module in a format that matches the type and format of the error. For this solution, create a multiple clause function with pattern matching in which the first parameter is the cause of the error in atom type, the second parameter is the message that belongs to the error in string type and finally, its third parameter is the name of the function initiating the display.

e.g.: error_message(errorType, Message, FunctionName) -> …

The function should have a default clause which handles the output even in case of wrong parameterization and displays a general error message on screen. To try out our error handling routine, create an Erlang module with three functions. The first function counts the number of elements in a list of arbitrary length, the second sums the elements it gets as its parameters and the third mirrors the list it gets as a parameter. All three functions use the previously created error handling routine in a way that the function body contains exception handling and it calls the error handling and displaying functions based on parameterization.

In order to ensure proper functioning, export all the functions, then, compile the module and run the functions with flawless and also with wrong parameterization. To handle exceptions you can use the simplest form of try where errors can be caught using the following pattern: (Error:Mesg -> …), where variable Error contains the cause of the error, variable Mesg contains the message supplied by the runtime environment of Erlang. All exception handling messages match this pattern but only in Erlang. In Clean and in F# programs you have to act differently. The error handling routine should display those two variables and the name of the function that contains exception handling which is given as a constant in all three functions. Based on this, you call error handling in the following way: (error_message(Error, Mesg, functionname)), where functionname is atom type. Function error_message/3 must be implemented in a way to include clauses for catching the most possible errors, which clauses are called if they are parameterized with the error type they define. The display instructions of the clauses should be formatted based on the messages that belong to the error. Besides the Erlang solution you can also implement the Clean and the F# versions. Naturally, the functions, error handling and the display should be changed according to the possibilities of the particular language.

14. exercise

Write an Erlang module which implements a server application capable of running higher order functions. The core of the server should be a function named loop without parameters

which calls itself recursively. Of course, recursion must be implemented with tail-recursion so that the server would not stop too early. In order for the server to be easily operable by the users, plan the proper communication protocol. The protocol always contains the identifier of the client in a sorted n-vector (process ID, or node identifier), the function expression of the task to be executed (higher order function), and the data sequence on which the function expression should be executed. The protocol in the server should also contain a stopping triplet ({Pid, stop, []}), and an operation similar to start ping, with which you can check the status of the server ({Pid, ping, []}). On stop message the server should send the stopped message for the client and the answer to status should be running. (Of course, if the server is not running you will not get the proper message.) The server should be addressed in a syncron way and the client must wait for the answer. For messaging you should use a remote procedure call type function, namely the call function of module rpc. You can find further help in the section about client-server applications. The server must contain client side functions as well (which is common in Erlang programs), and should work as follows:

rpc:call(server, remote, {self(), fun(X) -> X + 1 end, [12]})

Where function (rpc:call) is a library function of module rpc , remote is the name of the interface function of the server application, and function (fun(X) …) is the higher order function to be executed by the server. Obviously the function can be substituted with any higher order function. List [12], is the parameter of the function, which is used by the server as follows: If the request comes with tupple pattern {Pid, F, Params} and is caught with the proper receive structure,then function call F(Params) always works and can be sent back to the caller with formula Pid ! F(Params). To stop the server you just have to skip the recursive call in the particular clause of function loop.

Before starting to implement the server, try running the higher order function in a module to avoid correcting basic functionality mistakes when implementing the much more complicated server application. The function of the test module must have two parameters. The first parameter is the function to be executed, the second is the list. With this simple function you only need to pay attention that the second parameter must be a list (caller(F, Params) when is_list(Params) ->). The function body contains the following simple instruction: (F(Params)), because there is a higher order function, in fact a function prototype in, variable F. Here, you have to handle the list passed as a parameter, but do not use any other data type no matter how complex it is, because this is the only way to secure different numbers of parameters. Rather, you should solve list handling in the parameter functions. This method is common in functional languages. Erlang contains library functions with similar operations named apply. Function caller/2 can be parameterized based on its header: (caller(fun(X) -> X * X end, [1,2])). Notice, that our higher order function does not

deal with the list data structure. In order for function caller/2 to work properly, you have to use function apply/3 in its body, which library function runs the function it gets automatically with any list (apply(mod, F, Params)). This solution can be used for implementing the server version. To integrate it into the server you only have to place caller/2 in the proper clause of loop (the function implementing the inner cycle of the server) and send the result back to the client initiating the operations.

15. exercise

Implement an Erlang module with which you can check and compare the use of list generators and list traversing. The module should contain the recursive and list generator solutions of some known list handling library functions. The library function to be implemented is the following: lists:members/2 which functions decides whether an element is part of an arbitrary list (this is similar to the decision problem, but here the attribute is whether the element is part of the list or not). For the recursive solution you need to create a recursive function which checks the elements of the list one by one and compares the element with the other elements. It stores the match in recursive runtime and informs the world about it.

16. exercise

Create the version of the exercise using a list generator. This solution is a few instructions shorter as you can place every single element to another list under the condition that it matches the element you are looking for. If you get an empty list after running the generator the element you were looking for is not part of the list, otherwise it is. You can have multiple matches as condition unic was not used (List = [Element|| Element <- EList, Element == Wanted]), where variable Elist contains the list to be checked, List contains result z, and in variable Wanted you can find the element that you are looking for, received as the parameter of the function. If List remains empty, the element is not part of the set, otherwise it is. This fact can be concluded by evaluating the result of function call length(List), in a way that the return value of function members/2 will be the result of the following expression: (length(List) == 0).

The second library function to solve the task highly effectively can be length/1, which returns the number of elements it got as its parameter. This recursive function can be written with a body which counts the elements but if you are smart you can write a much shorter version using pattern matching. The version using list generator can be a little bit more complicated than the one you saw in members/2 but you can use a better solution here, too which is up to you to find. The point of the solution in all three cases is that somehow you have to check and count the elements of the list one by one just like in the task implementing summing.

The third and last function to be implemented is the map which runs the function expression, which it gets as its parameter, to all the elements of an arbitrary list. In the recursive version of this function the list it gets must be traversed and the higher order function, which is also a parameter, must be run to every element. In case of a list generator the solution is very simple as in the first part of the generator you can simply call the function expression to the elements ([F(Element) || Element <- List]), where F is the parameter function. In the version which does not use list generator you must traverse the function recursively but in this solution you must store the whole list or store it in a data storage invented for this purpose(ETS, MNESIA), if you need its elements. Calling a higher order function to the elements is not problematic here either since you can use the operations you used in the server. If the list is split to a first element and to the rest of the list at each function call, and the type of list elements is homogeneous and you know the type, the following call solves the problem: (F(First)), where F is the parameter function and First is the first element of the split list.

17. exercise

Write an Erlang module which can concatenate lists. This is not the same as the known merge problem in which you create one ordered list out of two ordered lists with the number of elements being the sum of the number of elements of the original lists. Here you will merge two lists but your task will be to form couples from the elements of the first list containing atoms and the elements of the second list containing numbers and place these into a new list. Let us add another twist to it. If the function receives a list that consist of couples it should divide it into two separate lists, the first containing the atoms, the first element of a couple, and the second containing the numbers, the second element of a couple. These two lists should be returned in a sorted n-vector as the function can only have one return value.

Therefore you have to create a function with minimum two clauses for the transformations. To make things simple let us create the version implementing concatenation. The first clause of the function gets the two lists to be concatenated (listhandler(L1, L2)). The first list contains atoms, the second numbers. Both lists must be split to a first element and to the rest of the list ([F1| R1], [F2| R2]). You have to create the ({F1, F2}) sorted n-vector, and add this to the list containing the result (…, F ++ [{F1, F2}]). Notice, that we used the list concatenation operator to produce the result so the sorted n-vector had to be transformed to a list as well. We must pass the rest of the two lists for the next call (R1, R2, …). If you run out of list elements, the second clause should take over control and return the result. It is worth paying attention to have two lists with equal number of elements. This can be ensured with the introduction of a guard: (when length(L1 == length(L2)), but this can cause problems in pattern matching (try to solve it…).

The second point should be the disjunction of the list. For the sake of simplicity you can use a function with the same name but different number of parameters in this task, which gets a list and returns two lists or you can use atoms for labeling the function clauses. The point of both solutions is to check in the header of the function, in the pattern matching of the lists if they have the right format. Only in this case should you run the function. (listhandler([{A, Int}, R], F1, F2)), where the ([{A, Int}]) pattern is interesting because with its use the clause only runs on lists which contain sorted couples. If you wish to refine the solution further, use a guard condition to check the type of data (when is_atom(A) and is_integer(Int)). The module can be tested after compilation with constant lists (listhandler(list1(), List2(), []).), to which you create functions in the module: (list1() -> [a, b, c, d]. list2() -> [1, 2, 3, 4].). This test data should return the following list: ([{a, 1}, {b, 2}, {c, 3}, {d, 4}]). Running the second clause the original two lists should be returned in tupple form: ({[a, b, c, d], [1, 2, 3, 4]}).

18. exercise

Write an application which gets a constant value and generates a list with the same length. It generates random numbers to fill the list and displays it on the screen in a formatted way then counts how many different values it got.

19. exercise

Write a console program which calculates the value of an expression. The program gets the expression from a function in string format or it can contain it as a constant.
There are some restrictions for the expression:

It can not contain space or any other white-space character.

It can only contain one operator and two operands.

Operands can only be integers and they are placed on the two sides of the operator.

For the solution you need to divide the string, which is simple in imperative languages, but in Erlang, where string type is just a so-called syntactic candy, our only help is the use of list expressions. For the division of the string you can use the conversion functions of the proper library module. If you can not cope with this solution then allow the operator and the operands to be passed in a list and the operator to be an atom while the operands are integers. If the parameters are divided the right way, you can easily decide with a case which real operation is the equivalent of the operator. For example: in case of atom ’+’ you must execute addition and so on. In the case you might need a default branch to avoid improper output even in case of wrong input and it is also a good idea to use an exception handling block to handle possible fatal errors. If you want to play safe, place a complex guard condition in the header of the function which is responsible for type compatibility: (expr([A, B, C]) when is_integer(A), is_atom(B), is_integer(c)). As you can see parts of the complex guard condition are divided with a comma which is the same as the use of the and operator.

In the function body you get the three elements from the list with pattern matching but in this case the list parameter can be omitted or can be changed to a sorted n-vector. Then, the parameter list looks like this: (expr({A, B, C}) when …). You can create multiple clauses for the function which provide proper result in case of using various data types, e.g.: they can concatenate lists. The module should be extended with a function that implements the following exercise

Let us have a look at another example similar to the previous one: Generate random integer numbers, then, tell which is the longest sequence in the list, that contains zeroes. The solution seems easy here but if you take your time you will realize that counting zeroes is problematic. All the generated values are numbers and the number of zeroes can only be counted if we convert them to another type. Conversion to string is the first version that comes up. The length of string can be measured and the character codes of zeroes can be found in the sequence. The second conversion type is the atom but its name suggests that it can not be divided to elements and you can not count zeroes in them. You can also choose a way to try to give the number of zeroes using mathematical operations. You are free to choose any of the options. After completing the task the output of the function should be displayed on screen using the previously applied error handling and displaying functions.

20. exercise

Create a module based on function lists:flatten/1 which so to say flattens nested lists: it unpacks nested lists until you get data that is different from a list. In this exercise a unique way of pattern matching of lists should be used. The list must be split to the first element and to the rest of the list here as well, but for unpacking further examination of the first element is necessary and if you still find list data structure the pattern matching must be carried out again involving the first element. It is a little bit like running a pattern matching function clause to the beginning and to the end of the list too.

Function flatten/1 called on the following list : ([[1, 2], {a, 1}, [{b, 2}, [c, 1]]]) returns the following result: ([1, 2, {a, 1}, {b, 2}, c, 1]). In order for our function to return the same result we must create a multiple clause function. During the usual pattern matching in the clauses you must separate the first element and pattern matching must be repeated with another function (or with the introduction of another clause) until you still get list data structure from the nested elements (or an empty list). If you find the innermost element, you have to return it and add it to a list in which you collect unpacked elements. This step is followed by the examination of the other elements of the list until you get an empty list. Obviously, the task can be solved with the use of a list generator with which you can rule out recursive calls. Due to varied data types and the use of various return values in Erlang, flattening is a very useful operation.

21. exercise

Erlang is capable of creating parallel, distributed programs. The simultaneous running of multiple threads is controlled by the system automatically. We only need to write the function which executes the particular calculations and passes them to thread handling routines as a parameter. To launch the thread handler the following expression must be evaluated: (Result1= [rpc:async_call(node(), ?MODULE, function, [Params] || <- List])). It is the async_call function of the rpc library module which asynchronously runs the function, which it gets as parameter, in as many threads as the number of elements in the list of the list generator (List). This list is not equivalent to the one passed as parameter which has as many elements as many parameters the paralytically run function expects. The next expression is ([_ = rpc:yield(K) || K <- Result1]), which processes the elements of list Result1, namely the data passed in the output of the previous expression. Of course, besides the two expressions you also need to write the function implementing real calculations which is named function in this example and has one parameter.

Now, that you can create distributed programs, write a module whose interface function can run an arbitrary function parallel on the elements of a list also passed as a parameter. To solve the task, you must use your knowledge on higher order functions and the functionality of parallelism. The only exported function of the module should have two parameters. The first of them is a function which is to be run parallel, the second is a list which contains the actual parameters of the function, one to each parallel running thread, and every element is a list which contains as many parameters as many functions it expects. Let us see an example to this parameterization. In case you wanted to run a summing function parallel which has two parameters (A, B), and you wish to implement distribution to 10 elements, you must create a list with double nested depth where each nested list is a possible parameter list of the summing ([[1,3], [6, 8], …, [n, m]]).

The function to be executed is sum(A, B) -> A + B, and we only have to give its name to the function (sum). Then, the interface has the following form: (sum, [[1, 2], [4, 6]…])). The formal parameter list, based on the actual one, is very simple, since it needs to contain only two variables. If you wish to write a nice program, the second parameter can be checked with a guard which filters list data structure (inerf(F, Params) when is_list(params)). You have to place the following expression in the function body: (KRes = [rpc:async_call(node(), ?MODULE, F, Params]), where macro ?MODULE refers to the currently used module. After that, you only have to process the result in the KRes variable with the (Res = [_ = rpc:yield(K) || K <- KRes]) expression(the meaning of the underscore sign in the list generator is that we do not want the ok atom of function yield, returned in the end of the run).

22. exercise

Proplist is an interesting element of Erlang programming language. We can create data structures similar to dictionary or enumeration type with it, which are common in imperative languages. In fact, proplist is a list containing a sequence of sorted couples. The first element of the couples always serves as a key, while the second element is the value that belongs to the key. It is a little bit like a telephone dictionary where there is a telephone number to every name.

A huge advantage of proplist is that there is a library function to handle them which implements plenty of list handling (proplist handling) functions, some of which we are going to use, or even write, ourselves.

The first such useful function is proplists:get_value/3, whose first parameter is the key, the second is the proplist in which we search for the value and the third parameter optionally contains the default value which is to be returned in case of an unsuccessful search. It is necessary for the search to always have a return value and to avoid extending the call routine with error handling. The get_value returns a single value. If the key is found among the list elements, the value which belongs to it is returned. If it is not found, the default parameter is returned (if you have given one). The next example program chooses a value from a three element list whose key is atom apple. If it fails to find such a key, it returns an empty list: (proplists:get_value(apple, List, [])). The list passed as parameter is the following: (List = [{apple, 1}, {pear, 2}, {wallnut, 3}]). Now, that we have tried how function get_value works, let us write our own, having the same function as the original, but implemented by us. We have already created something similar to concatenate lists. The only difference is that there we had to create a new list while here we have to find one element.

The first solution is based on list generators and finds the element that belongs to the key with their help. The parameter list is in the previously created List variable. This list must be traversed, the first element of each couple in it must be compared with the wanted element and if a match is found, it must be put into another list. It is not a problem if you have multiple matches as you can easily separate the first element of the list with pattern matching. At this phase the generator looks like this: (E = [Element || {Key, Element} <- List, Element = WantedElement]). WantedElement is the parameter of the function and it contains the key which we are looking for. List is also a parameter and you can also apply a third, default parameter as you could see in function get_value. The generator separates each couple in the list with a tupple pattern and checks in its condition if the first element (Key) and the wanted element (WantedElement) are equivalent. By the way, it also uses pattern matching for this check and exploits the fact that every element matches itself. If they are equivalent, then the value in variable Element must be put in the result list and the

first element must be the result of the function. ([KElement| _End]). If there is no match, you simply return the default parameter. The header of the function is like this: (getvalue(Wanted Element, List, []) when is_list(List) -> …). Its body contains the following instructions: (E = [Element || {Key, Element} <- List, Element = WantedElement, [KElement| _End] = E, KElement.). You can place a case statement to the end of the function checking if E contains any elements because if it does not, the previous pattern matching stops with an error. In case it contains something, you match the pattern, otherwise you return the default value.

23. exercise

The functioning of ETS tables is very similar to the functioning of proplist. Ets is a database management application that operates in the memory and helps us use global data in Erlang programs. Data transfer between functions of Erlang modules, except for passing parameters, is otherwise impossible as global variables are predestined to cause side effects in functions. Handling of shared data in servers is also implemented with messaging therefore global variables can not be used in that case either. Thus, our only option is the ETS table. ETS tables must be created and they will remain in the memory until the program that created them runs or an error occurs in the table handling. In case an error occurs we say our table crashed. Tables can be created with function ets:new(…) where you can name the table (atom type), and configure it to be accessible through its name. It is necessary to avoid handling the identifier of the table as a global variable if you wish to access it in various functions (not calling one another) (named_table). Besides this you can also configure the set property to rule out identical names, ensuring so-called unic functioning. The generating of our table is like this: (ets:new(table1, [set, named_table])), which instruction generates table table1 which, by default, contains key-value couples similarly to proplist. From this you can easily conclude that ETS tables are perfect for storing proplists and executing global data transfer between functions of the particular module. To delete the table, call (ets:delete(table1)) can be used, which not only deletes the content but also the table itself. If you want to insert elements into the table, use function (ets:insert/2) (ets:insert(table1, {key, 1})). In insert, the first parameter is the name of the function and obviously it only works if property named_table is set, or else you must use the identifier of the table which you get when it is generated (Identifier = ets:new(…)). The second parameter is the couple, whose first element is the key, identifying the data, and the second is the data. These can be defined in the form of sorted couples but of course you can create complex data with the nesting of n-vectors or lists.

The lookup is based on the keys similar to the way used in proplist but here you use functions capable of handling ETS tables. The simplest way is the (ets:lookup(table, key)) call,

where key is the wanted key and the return value of the function is the data. ETS tables can be easily transformed to lists with function call (ets:tab2list(table1)).

Let us try how to use the ETS tables. Create a table in Erlang command line with an arbitrary name, insert data into it, display it and finally delete it. After you succeed with all this, create your own database management module in the most platform independent way possible. Independence means that functions of the module have standard interfaces which do not change even if you change the database management module so that programs using the module do not recognize the change. This technique is very useful in every data storage program as you can never know when you have to change your database manager due to effectiveness issues. At times like that, it is very good that you do not have to change the whole program only the body of functions in the database management module. It is common for the programs using the module to have varied parameterization or for the new version to require new parameterization, still having older versions of your program running. In such cases you must introduce a conversion function keeping compatibility, compensating function calls with old parameters, but it can only be done if you have created a platform independent module.

Therefore, the module interface to be created contains the equivalents of function calls used in ETS and should simply call them implementing the necessary changes. The plan is the following: You need a create function as the synonym of ets:new. Its parameters must be wisely chosen but it might be best to use the original parameterization.: (create(TableName, Params) when is_list(Params)), where TableName contains the name of the table, Params contains the settings ([set, named_table]). The function body is the following: (ets:new(TableName, Params)). The when condition is introduced to filter non-list parameters and you may collect them in a list helping users that way. The return value of the function must be the identifier of the table as you must be able to work with non-named tables as well. This will work properly even if you use MNESIA tables or SQL instead of ETS. Writing function delete is really easy based on create. Its parameter is the name of the table, it calls function ets:delete/1 and its return value can be changed to be logical type but you can also return the return value of ets:delete/1.

Insert and lookup works similarly. You can choose more telling names too, e.g.: find or search might work. Keep its parameterization, so use the same parameters as in the original function: (find(Table, Key)), where Table is the name of the table and Key is the key to find and the return value is the value that belongs to the key. Now, that we can change parameterization, you have the chance to introduce a default value you could see in proplist (find(Table, Key, [])), which you need to give as the third parameter of the function (an empty list [] can be appropriate, but you can also let the user to decide). In this case you can

check with a case in the function body if there is a return value and if there is not, you return the default value. Insert calls ets:insert and works the same as delete.

You can name your own version implementing delete drop, based on SQL (drop(Table)), and you only need to call function ets:delete/1 in it and return its return value. Functionality is almost complete now, the only function remaining is to delete data from the table, or what is even better, a delete function that can be combined with the deletion of the whole content, which can be named delete due to the use of the name drop earlier. If delete does not have other parameter than the name of the table, deletes every data from it. The simplest way to do so is to drop the table and create it again. After that, you can also create a version with the same name but with different number of parameters, whose first parameter is the name of the table, the second is the key, namely the first element of the couples to be deleted. That version converts the content of the table into a list with function ets:tab2list and creates a new list not containing the wanted elements. When it is ready, it deletes the table and creates it again filling it with the elements of the new list (you can use the interface function of the module created for this purpose).

If you have completed the module, try it out. In order to do so, you only need to repeat the function calls that you have just entered into the command line using your own functions. If the result is the same as previously, or at least equivalent (due to the changes) then you have finished this exercises.

24. exercise

Write the erlang implementation of the well-known quicksort program, illustrated in the example program below.

```
quicksort([H|T]) ->
     {Smaller_Ones,Larger_Ones} = split(H,T,{[],[]}),
      lists:append( quicksort(Smaller_Ones),
      [H | quicksort(Larger_Ones)]
);
quicksort([]) -> [].
   split(Pivot, [H|T], {Acc_S, Acc_L}) ->
    if Pivot > H -> New_Acc = { [H|Acc_S] , Acc_L };
        true -> New_Acc = { Acc_S , [H|Acc_L] }
     end,
split(Pivot,T,New_Acc);
split(_,[],Acc) -> Acc.
```

Implement the program in an own module or integrate it where you implemented basic programming problems. Create an interface function to this module whose first parameter is the programming problem to be run, the second is the set or scalar type variable with which the program should be run. Add error handling routines to the module and to its exported functions. At this phase you can use the previously implemented error handling function which, besides the error message, also shows which function call caused it. The error handler is worth extending with a clause which does not display errors but results of the particular functions formatted according to the type of the particular return value.

Bibliography

[1] Programozási nyelvek, Volume.: Programozási nyelvek, Nyékyné Ga- izler Judit(ed.) Funkcionális programozási nyelvek elemeiZoltán HorváthProgramozási nyelvek, pages 589-636 ISBN 9-639-30147-7

[2] Horváth Zoltán, Csörnyei Zoltán, Lövei László, Zsók Viktória Funkcio-nális programozás témakörei a programtervező képzésben, Volume.: Informatika a Felsőoktatásban 2005., Pethő Attila, Herdon M.(ed.) Debrecen, Hungary , ISBN 963-472-9096

[3] Horváth Zoltán Funkcionális programozás nyelvi elemei, in.:Technical University of Kosice, Kassa, Slovakia

[4] Lövei László, Horváth Zoltán, Kozsik Tamás, Király Roland, Víg Anikó, Nagy Tamás Refactoring in Erlang, a Dynamic Functional Language, Dig Danny, Cebulla Michael(ed.) in.: Proc of 1st Workshop on Refactoring Tools (WRT07), ECOOP 2007 , ISSN 1436-9915

[5] Kozsik, T., Csörnyei, Z., Horváth, Z., Király, R., Kitlei, R., Lövei, L., Nagy, T., Tóth, M., and Víg, A.: Use cases for refactoring in Erlang In Central European Functional Programming School, volume 5161/2008, Lecture Notes in Computer Science, pages 250-285, 2008

[6] Joe Amstrong Programming Erlang Software for a Concurrent World armstrongonsoftware United States of America 2007 ISBN-10: 1- 9343560-0-X, ISBN-13: 978-1-934356-00-5

[7] Csörnyei Zoltán Lambda-kalkulus Typotex, Budapest, 2007

[8] Csörnyei Zoltán Típuselmélet (kézirat), 2008

[9] Wettl Ferenc, Mayer Gyula, Szabó Péter: L TEX kézikönyv, Panem A Könyvkiadó, 2004

[10] Functional Programming in Clean http://wiki.clean.cs.ru nl/Functional_Programming_in_Clean

[11] F# at Microsoft Research http://research.microsoft.com/en- - us/um/cambridge/projects/fsharp/

[12] The Haskell language http://www.haskell.org/onlinereport/

[13] Patrick Henry Winston, Bertold Klaus, Paul Horn Lisp Massachhussetc Institute of Technology - ISBN: 0-201-08329-9 Addison Wesley 8329 USA 1981

[14] Sikos László Bevezetés a Linux használatába 2005 ISBN: 9789639425002

[15] Imre Gábor Szoftverfejlesztés Java EE platformon Szak Kiadó 2007 ISBN: 9789639131972

[16] GNU Emacs http://www.gnu.org/manual/manual.html

[17] Mnesia Database Management System http://www.erlang.org

[18] Eclipse - The Eclipse Foundation open source community http://www.eclipse.org/