Note slide Distributed Systems 4

What about Strings?

Actually, in Erlang there is no special data type for strings, but they can be represented as Lists of integers.

What about the value of each element ?
It corresponds to a UNICODE codepoint --> we have the coding of the character (an integer) in each position in the list.

Anyway, there exists a special syntax for them, and a string can be indicated by special delimiters, the double quotes “ ”” ”.

See the examples:

Line 1: define a string. Of course it is a pattern matching.
Line 2: we obtain another string, using a special operator for the concatenation of strings (++). What you have in the right side is just an expression:

the evaluation of this expression is the concatenation of the 2 lists, then we apply pattern matching: S2 is an unbounded variable, so it will be bound to “Hello World!”.

Line 3: concatenate the list indicated by variable L2 and another list made of 1 integer (10). 10 in ASCII is the codepoint for new line.

Line 4: we concatenate to S2 an integer which cannot be interpreted as a codepoint, f.i. a negative integer ? S2 is just a list of integer, so we are just adding to the list yet another integer. Since -3 is not a codepoint, every element of the list is showed as an integer

Line 5: we can define S3 as a special string and the 4th character is indicated as “\x{2200} …” this is just a representation as a string of a Unicode codepoint and, in this example 2208 is the code for the special character in.

Line 6: we print S3 in a proper way.
Format(): in the first argument we put the template (“-ts” is for the string, “-n” is for “new line”), then we supply this function another argument, that is a list of the actual values to be substituted in the template string. This kind of string formatting works exactly as printf() in C.

Modules

What about real programs ?

They are made by different pieces of code that contains all the functionalities to be exploited in the app. These parts of SW are named modules.

A module is contained in a file whose extension is ".erl", and it is the basic unit of code in Erlang and it contains basically 2 different sections:

- Metadata:
  special description for what is the content of module and what should be done with the content w.r.t. external SW

- The core of the module:
  it contains the functions.

We cannot write functions directly on the shell, only in modules.

What are the most important metadata?

The module name: it has to be the same as the file name

Line 1: "-module";
the "-" means that this line belong to the metadata part.

The specification of what functions that are defined inside the module can be called out of that module.

Line 3: "-export" it presents the API of the module.

In Erlang we do not have public or private stuff, but instead we have another mechanism to make the invocation of functions available from the outside or not:
functions that are not exported from the module can be called only inside the module, instead what is in the export corresponds to the list of the functions you want to make all the other SWs be able to call, so you are declaring the API for your module.
Inside the export there is a LIST of all the functions that have to be exported.
Look f.i. at the first element "sayhello/0", what does it mean?

Each of this element has to identify one single function; a function in Erlang is identified by a name and the number of arguments, that is the arity of the function.
So, you can have 2 functions with the same name but with a different number of arguments.

Remember: Erlang make use of an intermediate language, the bytecode, so this module has to be compiled to obtain the corresponding bytecode to be stored in yet another file, with the same name but a different extension, ".beam".

After the metadata, you can see the specification of different functions:

- add/2 (not present in the export list, so it is a sort of helper function which can be called only inside this module),

- addinc/1 (not present in the export list since the arity is 1),

- addinc/2 (it is present in the export list)

- sayhello/0 (present in the export list).

Compiling & Using Modules

How to compile a module and how to use it ?

We can do it just from the command line. There is a special command, erlc, where you can specify some calculation for the compilation of the files, and then you have to provide the name of the file.

We can compile the module also from inside another module, or maybe from the shell, or repl, through a special function named compile:file() where you have to provide the name of the file as an argument, and it will get compiled. Most of the time, we will do the following, just operating from the inside of the repl.

There is a very simple command: it is a function named c() and you have to provide the name of the module indicated as an atom.

Once a module has been compiled, it can be used by external SWs. The most trivial external SW that can use the functions exported by the module is the Erlang shell, the repl.
Every time you want to call them, you have to specify both the name of the module and the name of the function.

Sometimes it could be convenient importing some specific functions, so that in a module that imports some functions defined in a specific module, such functions could be called without specifying the module name.

When you start your Erlang shell from one point in your file system, such directory will be taken as the working directory by the Erlang system. We can launch command like ls():
why, when launching it, at the end it prints “ok” ?
because when calling ls() actually it is just an expression, and it is made by an evaluation of a function, ls;
this function has as a side effect:
the printing on the screen of the name of the files in the current directory;
but, as any expression in Erlang, it has to correspond to a sort of output value, so the result of the evaluation of this command is the atom “ok”, and it was chosen to inform the programmer and the user that everything has been done in the correct way.

Often functions in Erlang outputs a tuple containing all the information we are interested in, like the compile:file() about what it has been done.

Functions - Basics

A function declaration can be inserted only in a module.

It is made of a sequence of function clauses, separated by “;” and terminated by “.”.

For each function clause:

- The arguments will be patterns;

- then there is an optional part [when GuardSeqJ] ,
- then there is “->”
- then the actual body for the clause.

What is the function name?
It is an atom

What about the arguments?
They are patterns.

What about the body of a clause?
It is a sequence of expression, each separated by “,” and one single clause evaluates to the value of the last expression in its body.

See the following example

Functions - Example Def/Call

We specify the module name “shape”, and what is exported is a function named “area” whose arity is 1.

We then have a definition of the function “area” and this definition is made of 3 different clauses.
When a clause terminates, I put a “;”.
When the definition of the entire function is terminated, I have to insert “.”.
If the body of a single clause contains more than one expression, we have to separate them with “, ”.

First clause: it takes only 1 argument, which is actually a pattern, made this way:
curly brackets --> we assume that here what is actually passed when the function is called is a tuple, likely with 2 different elements: the first one “square” is an atom, the second one is “Side”, this will be a variable (uppercase first letter) --> so in case area() will be called with an argument which is a tuple where the first element is the atom “square”, let’s consider the other argument that will be passed, let’s take it and let’s use it within the body, performing this calculation “Side*Side”.

If I want to compute the area of a circle, I pass as argument to the function “area” a tuple whose first element is the atom “circle” and second element is a variable named “Radius” which will be used in the body of the function to compute the area.

The same for the triangle.

Guards

There is something else that could be inserted in clauses to make the pattern matching more expressive, it is something that could be added to the basic pattern matching to better fits the needs of the programmer: guards.

A guard is just a Boolean expression that affects the way the program is executed.
Guards are especially used in the heads of the function clauses, just after the parenthesis containing the arguments.

A guard is a sequence of guard expressions. The set of guard expressions is a subset of valid Erlang expressions.

One important point:
it is not possible to insert a user-defined function in a guard. This has been decided to be sure to avoid side effects.

See the example:

Line 1: sign(X) (is a function).
Suppose you want to do different things according to what it is passed as an argument. We can write several different clauses.
The first clause can be written this way:

sign(X) <:– I want to choose this particular clause for the execution of the function when X is a number greater than 0, so I will execute the body of this clause, only when X is a number and (",") X>0. When all these conditions are met, the body will be evaluated.

Last line: it is as a default case (in all the other cases returns an atom saying "argerror") with the anonymous variable as argument, whose value is not important and we do not care the value placed there.

"Case" Expressions...

Note that in Erlang we have to do always with expressions, you can imagine the computation as the evaluation of subsequent expressions.

Even the "case" construct corresponds actually to a special way to specify an expression and sometimes, instead of relying on pattern matching on many clauses, you can choose to adopt this expression.

See in the example the structure of the "case" construct:

You have case, an expression Expr that will be evaluated, "of" and then different clauses. The case is terminated by the keyword end.

Inside each clause you have first a pattern (P1, ..., Pn), and in case the value of the expression Expr at the first line matches the pattern, this clause body E1 (, .., En) will be executed.
Moreover, in each clause, as it happens for functions, you can also add yet another condition using "when" (so a guard).

The patterns P1…Pn are sequentially matched, we stop at the first match.
In case the guard is not satisfied the other clauses are tried.
In case we have no matching pattern with a true guard sequence, we have a run time error, "case_clause".

To avoid this situation, at the end we place typically a "catch all" clause.
This is not a way to send the control flow to one direction or another as in other languages, this is just an expression, so the expression has always to be evaluated to a value, and if you do not have any match we do not know what will be the value associated with this specific expression and that is a run time error.
The use of case and the use of if is one of the most difficult point to be understood in the language because it is difficult to think at them as expressions.

… and “If” Expressions

It has to be used in case the resulting term has to depend only on guards, so we do not have something special but only a list of guards, and each of them will have a corresponding body.

We consider the different guards, they are sequentially checked, as soon as one guard succeeds, then the corresponding body is evaluated, and such a value will be a value returned as the evaluation of the if expression.

In case no guard succeeds, we will have a run time error because it must be evaluated to something.
Typically, in an if expression, the last guard is the guard TRUE which always evaluates to true, so the corresponding body will be the code that will be evaluated to the value to be returned.

Example: Looping over a List

The idea is that I want to print the content of a list.
We can go through the content of the list this way:
in case we have a list that contains nothing we want just to return back something;
otherwise, suppose you have a generic list with some elements, let’s take a look of the definition of scanl/1, which corresponds to calling scanl/2 with the same first argument, and with another helper value (0) that enables us to perform this scanning of the list.

So let’s focus on the second clause of scanl()/2, since the first one corresponds let’s say to the base case:

we have a list and we can specify that the list has to be identified as made of an Head H and a Tail T, so in the body I can access the Head through the variable H and the tail through the variable T.
Moreover, I will have an index in “Index”.

I want to write on the screen the value of the Head (see the template: “~w” is the first placeholder, than we have “: “ and then another placeholder ”~w”, and then the next line place holder).

What to substitute to the first place holder and to the second one ?
the arguments indicated in the list after the template, [Index, H]. --> “Index: H”.

The Index variable can be seen as a state variable.

Having Fun with Funs

In python funs are named lambdas.

Higher order functions typically work with function as parameters and they also return functions as the result of the execution.

To support the handling of functions as values to be passed from one function to the other and so on, in Erlang a special data type has been introduced: FUN, or anonymous function, because the syntax of FUN let us describe the value of a function without giving a name to such a value; it is like the value 10, I can imagine the value 10 even if no variable takes such a value.

You can see on the right the syntax:
you can specify different arguments that can be used in the function body.

As it regards the scope rules, please remember that here we are not in the context of imperative languages --> it does not make sense to think about variables that are outside our scope.
Here it makes sense using variables in the current expression to be evaluated, once you evaluated it the evaluated expression represents a value that might be bound to yet another value.
So, talking about variable scope in the same meaning we are used to think of it in imperative languages it's not appropriate.
So, always think of everything as evaluation of an expression.

The result of the FUN will be the value in which the last expression in the body of the FUN is evaluated to, and it will be returned.

It is a way to encapsulate behavior.

In the simplest case you may insert in the body a simple expression that makes use of the variables coming from the passed arguments, but there is the possibility to organize a FUN also providing multiple clauses

See the example:
A single fun is used.
It ends to the closing statement “end”.

Just to be able to handle this value, in this example I take this fun and I made the variable TempConv access to it.
So even if this is an anonymous function, I have a variable that refers to this fun.
This is a simple anonymous function to convert from Celsius to Fahrenheit and the vice versa.
These 2 operations can be done using different formulas, so I can imagine to have 2 different clauses in the definition of this fun, to deal with the 2 different situations.

How to discriminate the first situation from the second ? That is from C to F and from F to C ?
I can imagine to have a tuple as an argument whose first element indicates just what I want to be done --> cel if I want from C to F, far for the opposite.

The second element of the tuple will be the value I want to convert.

So now I have a fun which is accessible via the TempConv variable.
We know that we cannot define functions in the shell. Here we are making use of a sort of function and a variable to refer to it, so we are using kind of a trick to use a simple function without using external modules.

What if I want to have a fun that corresponds to an existing function?
The syntax is the one bottom right.

Funs for Building Control Abstractions

One important way to use funs is for building control abstractions.

This is important because we do not have so many control structures (like for), since this is a functional language, but we often need to create abstraction to work with some operation that could be repeated, or carried out in different ways, or mixed and so on, in abstract ways so that we will be able to specify on one side the way the operation have to be exploited and on the other side what specific operation you want to operate on in that particular way.

So, a control abstraction has to refer to an operation (that could be specified using a fun) which has to be executed according to some given rules.
The way to apply the rules can be specified by a function, and this generic operation you want to use could be a fun used as a parameter.

Look at the module in the example:

We have a specific function named myfor(), with 3 parameters which works with some operation indicated by the parameter named Oper, and this has to be executed several times, referring to an arithmetic progression starting from one number up to another number and at each iteration I want to get the following succeeding number:
say, myfor() from 2 up to 8 and you have to perform this operation. <:-- This is a control abstraction.

Suppose I call this function specifying that the starting index and the final index are just the same, I have to perform this operation just once, using the only index I passed, so the first clause can be considered as a sort of base case for recursion. So if I have the first case, I have to return a list consisting of one single element, which is calculated by passing the index Max as a parameter to the operation Oper.

General case: I have the initial value I, the final value Max and the operation Oper.
I have to fill up a list and the head of this list will be the value obtained by calling Oper(I). The rest of the list can be obtained by calling myfor() with the next first index (I+1), the same last index Max and the same operation Oper.
What if I want to perform the calculation of a certain function using the indexes whose values are from 1 up to 5 ?
We have to do a in the test() function.
See the last parameter passed there to myfor(): This will be a fun(X), and in this case we are doubling the value of X.

Example: Filtering a List

Here the idea is that I want to look at the list and filter some element in the list that have to be removed, because they are not good for my purposes.
I can decide what ones have to be removed just by applying a predicate:

it is just a function whose co-domain is made of 2 elements, true and false.

So, the idea is that I check element by element whether it has to be excluded because it doesn’t fit (the predicate will return false), all the elements that have to be kept are the ones where the predicate will return true.
The output will be the same as the output but those elements where the predicate returned false.

See the example:

The base case (first line) is very trivial:
suppose that you have an empty list, there is nothing to list --> no matter what the predicate is, what will be returned will be the empty list.

Ordinary case:

I can consider the list as made of a head (H) and a tail (T), so in the pattern I have the predicate (Pred) and a list sub-divided in head and tail.
Now, I have to check whether the head has or not to be kept, so I calculate Pred(H). This can be used as a guard of a case expression.
So, if it turns to be true, this expression will be computed and will result in the following: a list where H is kept, and I check whether the list which corresponds to the tail contains some elements to be excluded, so I make a recursive call to myfilter() where the predicate is the same, of course, but I have to work only on the tail T.

If I find that the Head has to be excluded, I only make a recursive call to myfilter() where the predicate is the same of course, but I have to work only on the tail T.

In the test() function we choose as predicate one that checks whether a number is even or not.

We defined a function (actually a predicate since its co-domain is only made of true and false) is_even() that makes the check.

Every time we have to work on lists, we do not have to write our own filter function, map function and so on, the most used functions to operate on lists has been already implemented in a very efficient way within a module named lists.

Example: Returning a Function

We know that we have in Erlang the Higher order functions --> it is possible to return a function as well --> there might be functions returning functions and getting functions as arguments.

See the example:
The idea is just to write a function that makes the composition of function in the mathematical meaning: suppose you have 2 functions, F and G, and you can consider them as functions from R -->R, taking one number and returning a number.
We first pass an argument to the first function and the result will be passed to the second one.
fcomp() gets 2 functions, F and G, and returns a fun.
How this new function is made ?

every time you pass a value X to this function, you will get as a result F(G(X)).