Programming Paradigms

Unit 16 — Erlang Modules, Functions and Control Structures

J. Gamper

Free University of Bozen-Bolzano
Faculty of Computer Science
IDSE

Outline

Outline

Functions and Modules

- Erlang is a functional language, so functions are the main building blocks
- Functions are defined within modules
- Function and module names must be atoms
- Functions must be exported before they can be called from outside the module where they are defined
- The standard library has a lot of predefined modules, e.g., the `lists` module to work with lists
- When calling a function from another module you need a qualified name, i.e., module name and function name separated by a colon

```
> lists:reverse([1,2,3,4]).
[4,3,2,1]
```

- You've already seen the module `io` for printing "Hello World!"

```
> io:format("Hello World!\n").
Hello World!
ok
```

Creating and using Modules

- To create your own module with some functions, you need to do the following:
  - Write a source file that contains function definitions and has extension `.erl`
  - Compile the module, which generates an executable file with extension `.beam`
  - Use the module
- Let's do this for a very simple piece of code:
  - A function that returns its input value

Creating a Module

- This is what the module file basic.erl looks like:

```
-module(basic).
-export([mirror/1]).

mirror(Anything) -> Anything.
```

- The first line defines the name of the module, an atom
- The second line tells Erlang that the function mirror
  - should be visible outside of the module and
  - has one parameter (that is the meaning of /1)
- The third line defines the function mirror with one argument
  - The symbol -> separates the function head and the function body
  - Notice the similarity to Haskell functions and Prolog-style rules

Compiling a Module

- After starting the Erlang shell from the directory of the code file you can compile it with the command c

  ```
  > c(basic).
  {ok,basic}
  ```

- This will create an executable file `basic.beam`

Using a Module

- Now you can run the functions defined in the module like this:

```
> basic:mirror(1).
1
> basic:mirror(abc).
abc
> basic:mirror("string").
"string"
> mirror(1).
** exception error:  undefined shell command mirror/1
```

- Notice that the parameter Anything was bound to different types in each call (number, atom, string)
- This means that Erlang uses dynamic typing
- The function name must be qualified with the module name, otherwise you get an error

Local Functions

- A module can define several functions
- Some functions might be local, i.e., not visible outside

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

double(X) -> times(X, 2).

times(X, N) -> X * N.
```

- This module defines 2 functions
  - `double` can be called from outside the module
  - `times` is local to the module

Function Declaration Revisited

- So far, each function consisted of a single line
- In general, a function declaration consists of a sequence of function clauses separated by semicolons, and terminated by period (.).

```
Name(Arg11, ..., Arg1N) [when GuardSeq1] ->
    Expr11, ..., Expr1M;
...;
Name(ArgK1, ..., ArgKN) [when GuardSeqK] ->
    ExprK1, ..., ExprKM.
```

- Each clause represents a different matching possibility (cf. Prolog, Haskell)
- Clauses are separated by a semicolon (;)
- A function clause consists of a head and a body, separated by `->`.
- A clause head consists of the function name, an argument list, and an optional guard sequence beginning with the keyword `when`.
- A clause body consists of a sequence of expressions separated by comma (,)

Function with Multiple Matching Possiblities

- This is a simple example with multiple matching possibilities/clauses

```
-module(matching_function).
-export([number/1]).

number(one) -> 1;
number(two) -> 2;
number(three) -> 3.
```

- You can execute it like this

```
> c(matching_function).
{ok,matching_function}
> matching_function:number(one).
1
> matching_function:number(four).
** exception error:  no function clause matching ...
```

Recursive Functions

- Similar to Prolog, recursion plays a big role in Erlang
- Erlang is optimized for tail recursion

```
-module(yam).
-export([fac/1]).
-export([fib/1]).

fac(0) -> 1;
fac(N) -> N * fac(N-1).

fib(0) -> 1;
fib(1) -> 1;
fib(N) -> fib(N-1) + fib(N-2).
```

Function Evaluation

- Function evaluation works essentially in the same way as in Prolog
- The function clauses are scanned sequentially (from top to down) until a clause is found that fulfills the following two conditions:
  - the pattern in the clause head can be successfully matched
  - the guard sequence, if any, is true
- If such a clause cannot be found, a runtime error occurs
- If such a clause is found, the corresponding clause body is evaluated:
  - the expressions in the body are evaluated sequentially (from left to right)
  - the value of the last expression is returned

```
fac(2)                  % match clause 2
    2 * fac(1)          % match clause 2
        1 * fac(0)      % match clause 1
            1
        1
    2
2
```

- This function is not safe. What happens if you call, e.g., `yam:fac(-3)`?

Functions with Guards

- A guard (sequence) is an optional condition that can be placed in the head of a function clause
- They can be used to make functions more robust

```
-module(yam).
-export([fac2/1]).

fac2(0) -> 1;
fac2(N) when N > 0 -> N * fac2(N-1).
```

- Now you get the following:

```
> yam:fac2(3).
6
```

```
> yam:fac2(-3).
** exception error: no function clause matching yam:fac2(-3)
```

- We will see later on how to avoid even the runtime error

Outline

Control Structures: case

- Control structures look similar to functions
- The case statement in Erlang looks like this

```
> Animal = "dog".
"dog"
> case Animal of
>     "dog"        -> underdog;
>     "cat"        -> thundercat;
>     "elephant"   -> dumbo;
>     _            -> something_else
> end.
underdog
```

- The case statement uses pattern matching to distinguish various cases
  - A pattern might optionally be followed by a guard
- The underscore (_) matches anything
- The pattern can be more complex, e.g., lists or tuples

Control Structures: if

- In contrast to the case statement, the if statement uses guards
  - i.e., boolean conditions that must be satisfied for a match to succeed

```
> if
>     X > 0 -> positive;
>     X < 0 -> negative;
>     true  -> zero
> end.
```

- One of the branches of a case or if statement has to be true, otherwise a runtime error will occur if no match is found
- The reason is that case and if are functions that have to return a value
- In the above example, the last guard is simply set to the expression true
  - true always succeeds
  - alternatively, X == 0 could be used

Outline

Higher-order Functions

- A function that returns functions or takes functions as arguments is called a higher-order function
  - In Ruby this was achieved passing code blocks
  - In Prolog, you can pass as argument a list that contains functor and arguments of a predicate (which then can be composed to a predicate using the univ operator `=..`)
  - In Haskell, we have seen functions such as `map` that take a (anonymous) function and a list and apply the function to the list
  - In Erlang arbitrary functions can be assigned to variables and be passed around like any other data type
- Let's start with anonymous functions (functions without a name)

Anonymous Functions

- Let's define a function that negates its argument

```
> Negate = fun(I) -> -I end.
#Fun<erl_eval.6.111823515>

> Negate(1).
-1

> Negate(-3).
3
```

- The keyword `fun` defines an anonymous function that has no name
- This function is assigned to variable `Negate`
- `Negate` actually is the function, not the value returned by the function
- By assigning the function to a variable, the function
  - can be passed as a parameter like any other data, and
  - it can even be called with different parameters
- We will show this next

Lists and Higher-order Functions

- A large number of higher-order functions are defined for lists in the module `lists`
- `lists:foreach` takes a function and a list as arguments and iterates over the list applying the function to each element

```
> Numbers = [1,2,3,4].
[1,2,3,4]
> Print = fun(X) -> io:format("~p~n", [X]) end.
> lists:foreach(Print,Numbers).
1
2
3
4
ok
```

- `io:format/2` is a function for formated printing
- `~p` pretty prints an argument, `~n` prints a newline

Mapping Lists

- The map function is another function that takes a function and a list as arguments
- However, it applies the function to each element and builds a new list with the results
- The following example increases each element of a list by 1

```
> Numbers = [1,2,3,4].
[1,2,3,4]
> lists:map(fun(X) -> X+1 end,Numbers).
[2,3,4,5]
```

- This example shows also that an anonymous function can be defined directly when a function is called

Filtering Lists

- Lists can also be filtered using a boolean function.
- The function `lists:filter` builds a new list with all elements that satisfy a given function

```
> Small = fun(X) -> X < 3 end.

> Small(4).
false

> Small(2).
true

> lists:filter(Small,Numbers).
[1,2]
```

Testing Lists

- The function `Small` can also be used to test whether
  - all elements (`lists:all`) of a list are less than 3 with , or
  - any element (`lists:any`) of a list is less than 3

```
> lists:all(Small, [0,1,2]).
true
```

```
> lists:all(Small, [0,1,2,3]).
false
```

```
> lists:any(Small, [0,1,2,3]).
true
```

```
> lists:any(Small, [3,4,5]).
false
```

- The two functions applied on empty lists

```
> lists:any(Small, []).
false
> lists:all(Small, []).
true
```

Extract and Skip Prefix of Lists

- You can also use a filter condition to
  - extract a prefix of a list (`lists:takewhile`), i.e., make a list of all elements at the head of a list that satisfy the filter or
  - to skip that elements (`lists:dropwhile`)

```
> lists:takewhile(Small, [1,2,3,4]).
[1,2]
```

```
> lists:dropwhile(Small, [1,2,3,4]).
[3,4]
```

```
> lists:takewhile(Small, [1,2,1,4,1]).
[1,2,1]
```

```
> lists:dropwhile(Small, [1,2,1,4,1]).
[4,1]
```

Fold Functions on Lists/1

- Fold functions are a family of functions that accumulate a return value while processing a data structure, such as a list
- A fold function takes as argument:
  - a function
  - an initial accumulator value
  - and a list
- Lets sum up the elements of our list of numbers

```
> Adder = fun(Item,SumSoFar) -> Item + SumSoFar end.
> InitSum = 0.
> lists:foldl(Adder,InitSum,[1,2,3,4]).
10
```

Fold Functions on Lists/2

- Fold functions for lists come in two flavors: `foldl` and `foldr`
- Both of them take the same parameters: a function, initial accumulator value, and a list
- The difference is the order in which they combine the list elements with the accumulator
- `foldl` accumulates elements from left to right
- `foldr` accumulates elements from right to left

```
> P = fun(A, AccIn) -> io:format(" p ", [A]), AccIn end.

> lists:foldl(P, void, [1,2,3]).
> 1 2 3 void

> lists:foldr(P, void, [1,2,3]).
> 3 2 1 void
```

- Notice that the function assigned to variable `P` has side-effects, hence is not pure functional!

Outline

Example: Length of Lists

- Let's write a function to compute the length of a list (i.e., `length/1`)

```
-module(mylists).
-export([len/1]).

len([]) -> 0;
len([_|T]) -> 1 + len(T).
```

- This is how this function is executed:

```
len([1,2,3]) = len([1 | [2,3]])
             = 1 + len([2 | [3]])
             = 1 + 1 + len([3 | []])
             = 1 + 1 + 1 + len([])
             = 1 + 1 + 1 + 0
             = 1 + 1 + 1
             = 1 + 2
             = 3
```

- This is function is not tail-recursive

Example: Length of Lists – Tail-recursive

- To come up with a tail-recursive version, we use an accumulator

```
-module(mylists).
-export([tail_len/1]).

tail_len(L) -> tail_len(L, 0).

tail_len([], Acc) -> Acc;
tail_len([_|T], Acc) -> tail_len(T, Acc+1).
```

- This is how this function is executed:

```
tail_len([1,2,3])        = tail_len([1,2,3], 0)
tail_len([1|[2,3]], 0)   = tail_len([2,3], 0+1])
tail_len([2|[3], 1])     = tail_len([3], 1+1])
tail_len([3|[]], 2])     = tail_len([], 2+1])
tail_len([],3])          = 3
```

- This function is tail-recursive

Example: Reverse a List

- We write a function reverse/1 to reverse a list

```
-module(mylists).
-export([reverse/1]).

reverse(List) -> reverse(List, []).

reverse([H|T], Acc) -> reverse(T, [H|Acc]);
reverse([], Acc) -> Acc.
```

- This is how this function is executed:

```
reverse([1,2,3])         = reverse([1,2,3], [])
reverse([1|[2,3]], []) = reverse([2,3], [1|[]])
reverse([2|[3]], [1])  = reverse([3], [2|[1]])
reverse([3|[]], [2,1]) = reverse([], [3|[2,1]])
reverse([], [3,2,1])   = [3,2,1]
```