Erlang: getting started

Johan Montelius

October 25, 2013

Getting started

We assume that you have Erlang up and running and an editor to write your first Erlang programs. You don't need a full IDE, rather the less you have to think about the better.

In this tutorial we will write input to the Erlang shall as follow:

```
> x + 2.
```

1 Simple arithmetic

Open up a Erlang shell and try some simple arithmetic. Try these examples:

```
>6 + 2.

>6 div 2.

>7 div 2.

>7 rem 2.

>-1 rem 5.

>7 / 2.
```

Some simple comparisons.

```
>3 == 3

>3 /= 4

>4 < 7
```

2 A first program

Open up a file *test.erl* and create a module called *test*. In this module we define a function called `double` that takes one argument and returns the double of that argument.

```
-module(test).

double(N) -> ...
```

In the Erlang shell you can load the module and try if it works.

```
>double(4).
```

Now in the same module define the following functions:

- a function that converts from Fahrenheit to Celsius (the function is as follows:
$$C = (F - 32)/1.8$$
)
- a function that calculates the area of a rectangle give the length of the two sides
- a function that calculates the area of a square, using the previous function
- a function that calculates the area of a circle given the radius

3 Recursive definitions

Assume that all we have is addition and subtraction but need to define multiplications. How would you do? You will have to use recursion and you solve it by first describing the multiplication functions by words.

The product of m and n is: 0 if m is equal to 0, otherwise the product is m + the product of (m-1) and n.

Once you have written down the definition, the coding is simple.

```
product(M, N) ->
   if
     M == 0 -> ...;
     true -> ...
   end.
```

There are alternative ways of writing this, we could use a *case expression*:

```
product(M, N) ->
   case M of
     0 -> ...;
     _ -> ...
   end.
```

or simply written it as two *clauses*:

```
product(0, _) ->
   0;
product(M, N) ->
   ... .
```

The last definition might be the easiest to read and is often used.

Define a function, `exp/2`, that computes the exponentiation, x^y. Use only the addition and subtraction and the function `product/2`, that you defined.

```
exp(X, ...) ->
   ...;
exp(X, Y) ->
   ... .
```

Use the built-in arithmetic functions *rem*, *div* and multiplication * to implement a much faster exponentiation using the following definition:

- x raised to 1 is x
- x raised to n, if n is even, is x raised to n/2 multiplied by itself
- x raised to n, if n is odd, is x raised to (n-1) multiplied by x

4 List operations

You will do more operations on list than you have ever done before so you might as well get use to them. These are operations that you should know by heart.

```
>[1|[]].

>[1|[2|[]]].

>[1,2].

>[1,2] == [1|[2|[]]].

>[A,B,C] = [1,2,3]

>[H|T] = [1,2,3].

>[_, {X,Y} | _] = [{a,1},{b,2}, {c,3}, {d,4}].

>[Z] = [1,2,3].
```

4.1 Simple functions on list

These are some simple functions that you should implement. They will all use recursion so first try to formulate in words what the definition should look like, then encode it using two or more clauses.

- nth(N, L): return the N't element of the list L
- number(L): return the number of element in the list L
- sum(L): return the sum of all elements in the list L, assume that all elements are integers

These functions take some more thinking. You should return a list as a result of evaluating the function.

- duplicate(L): return a list where all elements are duplicated
- unique(L): return a list of unique elements in the list L, that is `[a,b,d]` are the unique elements in the list `[a,b,a,d,a,b,b,a]`
- reverse(L): return a list where the order of elements is reversed
- pack(L): return a list containing lists of equal elements, `[a,a,b,c,b,a,c,]` should return `[[a,a,a], [b],[c,c]]`

4.2 Sorting

There are several ways to sort a list and you should know them all. We will start with the most basic algorithm and then try some other (more or less good).

4.3 insertion sort

In *insertion sort*, you sort a list of elements by taking them one at a time and *insert* them into an already sorted list. The already sorted list will of course be empty when we start but will when we are done contain all the elements.

Start by defining a function *insert(Element, List)*, that inserts the element at the right place in the list. Think of the two mayor cases, what to do if the list is empty and what to do if the list contains at least one element. Assume that the elements are integers and can be compared using the regular ¡ operator.

Once you have *insert/2* working, implement the sorting function *isort(List, Sorted)*; again what should you do if the list is empty, what should you do if it contains at least one element?

Now all you have to do is provide a function *isort(List)*, that calls the function *isort/2* using the right arguments.

```
isort(L) -> isort(L, ...).

isort(.., ...) ->
  ...;
isort(..., ...) ->
  isort(..., ...).

insert(..., ...) ->
  ...;
insert(..., ...) ->
  if
    ... -> ...;
    true -> ...
  end.
```

4.4 merge sort

In *merge sort*, you divide the list into two (as equal as possible) list. Then you merge sort each of these lists to obtain two sorted sub-lists. These sorted sub-lists are then *merged* into one final sorted list.

The two lists are merged by picking the smallest of the elements from each of the lists. Since each list is sorted, one need only to look at the first element of each list to determine which element is the smallest.

The skeleton code below will give you an idea of what the solution will look like.

```
msort(..) ->
    ...;
msort(..) ->
    ...;
msort(...) ->
   {..., ...} = msplit(..., [], []),
   merge(msort(...), msort(...)).

msplit(..., ..., ...) ->
   {..., ...};
msplit(..., ..., ...) ->
   msplit(..., ..., ...).
```

4.5 quick sort

The *quick sort* algorithm sounds even quicker than merge sort but this is not true. The idea is similar but now we "do our sorting on the way down". First split the list into two parts, one containing low elements and one containing

high elements. Then sort the two lists and when you're done append the results.

Splitting the lists is done using the first element in the list as a *pivot element*, all smaller or equal than this is added in one list and all larger in one list. When you're appending the final result, remember to put the pivot element in the middle.

```
qsort(...) -> ...;
qsort(...) ->
   {..., ...} = qsplit(..., ..., [], []),
   SmallSorted = ...;
   LargeSorted = ...;
   append(..., ...).

qsplit(..., ..., ..., ...) ->
  {..., ...};
qsplit(..., ..., ..., ...) ->
  qsplit(..., ..., ..., ...).
```

5 Reverse

One interesting problem to look at is how to reverse a list. The *naive* way to do it is quite straight forward. We do it recursively by removing the first element of the list, reversing the rest and then appending the reversed list to a list containing only the first element.

```
nreverse([]) -> [];
nreverse([H|T]) ->
   R = nreverse(T),
   append(R, [H]).
```

A smarter way to do it, is to use an *accumulator* and add the first element to this accumulator. When we have added all elements in the lists the accumulated list is the reversed list.

```
reverse(L) ->
   reverse(L, []).

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

Ok, so what is so smart by doing this? This is your assignment, you should do some performance analysis of these two functions and describe

what is happening. To have some data lead you in the right direction and to back up your findings you should start by doing some performance measurements. We have here used some library functions and higher order programming that you might not have seen sofar but

```
bench() ->
    Ls = [16, 32, 64, 128, 256, 512],
    N = 100,
    Bench = fun(L) ->
       S = lists:seq(1,L),
       Tn = time(N, fun() -> nreverse(S) end),
       Tr = time(N, fun() -> reverse(S) end),
       io:format("length: ~10w  nrev: ~8w us    rev: ~8w us~n", [L, Tn, Tr])
       end,
    lists:foreach(Bench, Ls).

time(N, F)->
    %% time in micro seconds
    T1 = now(),
    loop(N, F),
    T2 = now(),
    timer:now_diff(T2, T1).

loop(N, Fun) ->
  if N == 0 -> ok; true -> Fun(), loop(N-1, Fun) end.
```

6 Binary coding

As the next exercise, you should implement a function that takes an integer and return its binary representation coded as a list of ones and zeroes. The binary form of 13 is for example *[1,1,0,1]*.

Therea re many ways to solve this problem and you should solve it in two different ways, describing the difference and if one is better than the other.

7 Fibonacci

The Fibonacci sequence is the sequence $0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots$. The two first numbers are 0 and 1 and the following numbers are calcuated by adding the two previous number. To calculate the Fibonacci value for 42, all you have to do is find the Fibonacci number for 40 and 41 and then add them together.

Write simple Fibonacci function *fib/1* and do some performance messurements. Adapt the benchmark program above and run some tests.

```
fibb() ->
    Ls = [8,10,12,14,16,18,20,22,24,26,28,30,32,34,36,38,40],
    N = 10,
    Bench = fun(L) ->
       T = time(N, fun() -> fib(L) end),
       io:format("n: ~4w  fib(n) calculated in: ~8w us~n", [L, T])
       end,
    lists:foreach(Bench, Ls).
```

Find an arithmetic expression that almost describes the computation time for $fib(n)$. Can you justify this arithmetic expression by looking at the definition af the function? How large Fibonacci number do you think you can compute if youstart now and let your machine run until the seminar? First make a guess, don't try to do teh calculation in your head just make a wild guess, then try to estimate how long time that woudl take using your arithmetic function, woudl you be abel to make it? Calculate a high Fibonacci number and bring to the seminar.