Elixir: getting started

Johan Montelius

January 12, 2018

Getting started

We assume that you have Elixir up and running and an editor to write your first Elixir 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 Elixir shell as follow:

```
>  x + 2
```

You will actually see a number `iex(1)>` etc, but we will simply show a single `>`.

1 Simple arithmetic

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

```
> 6 + 2

> 6 / 2

> div(7,2)

> rem(7,2)

> rem(-1,5)
```

Some simple comparisons.

```
>3 == 3

>3 != 4

>4 < 7
```

Try this:

```
>h()
```

2 A first program

Open up a file *test.ex* and create a module called *Test*. In this module we define a function called `double/1` that takes one argument and returns the double of that argument.

```
defmodule Test do

 def double(n) do
        :
 end

end
```

In a regular shell you can now compile this program using the stand alone compiler:

```
$ elixirc test.ex
```

You can also compile and load the program from within the elixir shell.

```
$ c("test.ex")
```

In the Elixir shell do try this:

```
> Test.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

Sooner or later you will have to think about which programming environment that you are going to use. We will not write very large programs in the course, but you will need to be able to quickly re-compile programs and shift between the editor and the Elixir shell. There are many environments to choose from and you are of course free to use whichever you find best.

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 n plus the product of $m - 1$ and n.*

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

```
def product(m, n) do
  if  m == 0 do
      ..
  else
      ..
  end
end
```

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

```
def product(m, n) do
 case  m do
   0 ->
       ...
   _ ->
       ...
 end
end
```

We could also have used the a style that sometimes is very handy. Here we break the definition up into two *clauses* that are tried one after the other.

```
def product(0, _) do 0 end
def product(m, n) do
  product(.., ..)
end
```

This should be read: if we call product, and the first argument matches *0*, then the result is *0*. If we can not use the first clause then we try the second.

Sometimes the code becomes easier to understand, especially if we have many conditions that should be tested. Remember though that the clauses of a function need to be after each other. You can not spread the clauses around in a program.

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.

```
def exp(x,y) do
  case ...   do


  end
end
```

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 used to them. These are operations that you should know by heart.

```
>[1|[]]

>[1|[2|[]]]

>[1,2]

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

>[x,y,z] = [1,2,3]

>[head|tail] = [1,2,3].

>[_, {x,y} | _] = [{:a,1},{:b,2}, {:c,3}, {:d,4}].

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

In the above examples, what is the value of the variables after the pattern matching expressions?

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 implement it.

- nth(n, l): return the *n'th* element of the list l
- len(l): return the number of elements 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
- add(x,l): add the element x to the list l if it is not in the list
- remove(x,l): remove all occurrences of x in l
- 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]`
- 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]]`
- reverse(l): return a list where the order of elements is reversed

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.

```
def isort(l) do
  isort(l, ...)
end

def isort(x, l) do
  case ...  do
    [] ->
       ...
    [h|t] when h < x ->
       ...
    [h|t] ->
       ...
  end
end
```

Try also to rewrite the *isort* function using the clause syntax; same-same but different.

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. Here we do use the clause syntax when defining *merge*, you can try to define it using *case expressions* but it becomes a bit messy.

```
def msort(l) do
  case ..  do
    .. ->
      ...
    .. ->
      {.., ..} = msplit(l, [], [])
      merge(msort(..), msort(..))
  end
end

def merg(.., ..) do ... end
def merg(.., ..) do ... end
def merg(.., ..) do
```

```
  if ...
    merge(.., ..)
  else
    merge(.., ..)
  end
end

def msplit(.., .., ..) do
  case ..  do
     .. ->
       {.., ..}
     ..  ->
       msplit(..., ..., ...)
  end
end
```

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.

```
def qsort(...) do ... end

def qsort([p|l]) do
  {..., ...} = qsplit(p, l, [], [])
  small = ...
  large = ...
  append(small, [p|large])
end


def qsplit(_, [], small, large) do ....  end
def qsplit(p, [h|t], small, large) do
  if ...  do
       ...
  else
       ...
```

```
  end
end

def append(.., ..) do
  case .. do
    [] -> ..
    [h|t] -> ...
  end
end
```

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.

```
def nreverse([]) do [] end
def nreverse([h|t]) do
  r = nreverse(t)
  append(r, [h])
end
```

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.

```
def reverse(l) do
  reverse(l, [])
end

def reverse([], r) do  .. end
def reverse([h|t], r) do
  reverse(t, [h|r])
end
```

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 so far but don't worry, you will get use to it.

```
def bench() do
  ls = [16, 32, 64, 128, 256, 512]
  n = 100
  # bench is a closure - function with an environment
  bench = fn(l) ->
    seq = :lists.seq(1,l)
    tn = time(n, fn() -> nreverse(seq) end)
    tr = time(n, fn() -> reverse(seq) end)
    :io.format("length: ~10w  nrev: ~8w us    rev: ~8w us~n", [l, tn, tr])
  end
  # We use the library function Enum.each that
  # will call bench(l) for each element l in ls
  Enum.each(ls, bench)
end

def time(n, fun) do
  start = System.monotonic_time(:milliseconds)
  loop(n, fun)
  stop = System.monotonic_time(:milliseconds)
  (stop - start)
end

def loop(n, fun) do
  if n == 0 do
    :ok
  else
    fun.()
    loop(n-1, fun)
  end
end
```

6 More challenges

You should now be up and running to take on some new challenges. When you try these challenges, first try to express your algorithm using recursion. Think about the simplest case and have this as your base case. Then formulate a rule that will take you from a more complex form one step closer to the base case.

6.1 integer to binary

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]*. Converting 0 should be trivial so the base case should be simple.

In the recursive case we can calculate the binary representation of *div(n,2)* and the append it to either a 0 or 1 depending on if the number is even or odd.

```
def to_binary(0) do ...  end
def to_binary(n) do
    apped( ..., ...)
  end
end
```

This could be written in a better way by using an accumulator. The accumulator will hold the binary sequence that we have determined so far. We start with a empty list, [] and add binary digits as we go.

```
def to_better(n) do to_better(n,[]) end

def to_better(0, b) do b end
def to_better(n, b) do
   to_better(div(n/2), [rem(n,2) | b])
end
```

Why is this better than the previous one? Can you notice the difference? Try the following:

```
> Test.time(1000, fn() -> Test.to_binary(12348987980980980980987) end)
```

Now try the better version - any difference? Why?

6.2 binary to integer

Now try the revers and implement a function that takes a binary representation of a number and returns an integer. To solve this it is a lot easier to use an accumulator that holds the number of what we have seen so far.

```
def to_integer(x) do
  to_integer(x, ..)
end

def to_integer([], n) do  ..  end
def to_integer([x|r], n) do
  to_integer(.. , ....)
end
```

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 calculated 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 measurements. Adapt the benchmark program above and run some tests.

```
def bench() ->
    ls = [8,10,12,14,16,18,20,22,24,26,28,30,32],
    n = 10,
    bench = fn(l) ->
         t = time(n, fn() -> fib(l) end),
         :io.format("n: ~4w  fib(n) calculated in: ~8w us~n", [l, t])
       end
    Enum.each(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 of the function? How large Fibonacci number do you think you can compute if you start now and let your machine run until the seminar? First make a guess, don't try to do the calculation in your head just make a wild guess, then try to estimate how long time that would take using your arithmetic function, would you be able to make it?

Calculate a Fibonacci number as high as you possibly can.