# A Meta-Interpreter

Programming II - Elixir Version

Johan Montelius

Spring Term 2018

## Introduction

In this assignment you will implement an interpreter for a small functional language. The language could be the functional subset of Elixir but it does not match exactly how Elixir is defined (but maybe it should). The exercise will hopefully give you a better understanding of how a functional programming language is defined and how powerful a functional programming language can be as tool; the size of our interpreter is surprisingly small.

## 1 The overall picture

Before staring the implementation you need to get an overall picture of the goal and what is actually needed. Never start coding one piece in a jigsaw puzzle if you don't know what the final picture should look like or how the piece should fit in.

### 1.1 A meta-interpreter

An interpreter, in computer engineering terminology, is a program that takes a program as input and executes the statements of the program to produce a result. Some programming systems use only a interpreter to execute programs but most systems today use a compiler to produce machine code that is then executed either directly or in a virtual machine. The advantage of an interpreter is that we do not have to wait for a compilation phase, the disadvantage is that the interpreter is much slower (typically a factor ten to

one hundred). Do some reading and find out if the programming languages that you know uses a compiler or an interpreter.

A meta-interpreter, also called self-interpreter, is an interpreter that is implemented in the language that it interprets. The reason for this not obvious but if you have a compiled system and need an interpreter of the same language, why not implement it in the language itself - this is probably the language you know best. We will develop an interpreter for a subset of the Elixir language. Since it is not the complete language it is not a true meta-interpreter but it's close.

## 1.2 Expressions

Our target programming language is a limited functional language and in its first incarnation it will be limited to simple expressions; We will not have any function calls nor any program with function definitions (you might not even call this much of language but it will be a good starting point). This is an example of a *sequence* in the language:

```
x = foo; y = :nil; {z, _} = {:bar, :grk}; {x, {z, y}}
```

Figure 1: a simple sequence

A *sequence* consist of a, possibly empty, sequence of pattern matching expression followed by a single expression. This is a bit more restrictive compared to Elixir sequences but it does not limit the expressiveness of our language. Simple expressions will for now be limited to *terms*. If we use a BNF grammar we can describe a sequence in the following way:

⟨*sequence*⟩ ::= ⟨*expression*⟩
  | ⟨*match*⟩ ';' ⟨*sequence*⟩

⟨*match*⟩ ::= ⟨*pattern*⟩ '=' ⟨*expression*⟩

⟨*expression*⟩ ::= ⟨*atom*⟩
  | ⟨*variable*⟩
  | '{' ⟨*expression*⟩ ',' ⟨*expression*⟩ '}'

## 1.3 Terms, data structures and patterns

Expressions of the form that we have just described are also referred to as *terms*. We will later see examples of expression that are more complicated and we will talk about expressions in general and *term expressions* but for now we will simply call them *terms*.

It is important that we are clear on the terminology when we discuss our language. There is a difference between *terms*, *patterns* and *data structures*. In the sequence previously shown, the following constructs are terms: `:foo`, `:nil`, `{:bar, :nil}` and, `{x, {y, z}}`. Terms are syntactical constructs that, *when evaluated*, will result in *data structures*; data structures are thus something that we handle during execution, the terms are only recipes for how these structures should be constructed.

If we *evaluate* the sequence above we will obtain the data structure {*:foo,* {*:bar, :nil*}}. We will in this text use *italics* when we refer to the data structures and `teletype` when we talk about the terms.

For now, terms are restricted to atoms, variables and the binary compound *cons* structure. Once we understand how to handle this subset it's fairly easy to extend the language.

The patterns in the sequence are: `x`, `y` and, `{z, _}`. The syntax we use for patterns, is the same as the one that we use for terms but we are also allowed to use an underscore (`_`) to represent a *don't-care* variable. This variable acts as a placeholder for a data structure that we have no interest in.

⟨*pattern*⟩ ::= ⟨*atom*⟩
| ⟨*variable*⟩
| '_'
| '{' ⟨*pattern*⟩ ',' ⟨*pattern*⟩ '}'

The reason why we can use the same syntax for terms and patterns without confusion is that it is clear from the grammatical rules of the language if we refer to a pattern or a term.

## 1.4 Evaluation

Our interpreter will take a sequence and evaluate the pattern matching expressions one after the other; the result of the last expression is the result of the whole sequence. When the evaluation starts the interpreter will have

en empty *environment* i.e. it knows of no variable bindings. Each pattern matching expression will add variable binding to the environment and the following expressions are then evaluated in the new environment.

If we evaluate the sequence in figure 1 we gradually build an environment, first we add the binding *x/foo*, then *y/nil* and then *z/bar*. The final expression is thus evaluated in the following environment: {*x/foo, y/nil, z/bar*}. In this environment the term `{x, {z, y}}` is evaluated to the data structure {*foo,* {*bar, nil*}}.

### 1.5 The architecture

In order to implement our interpreter we need to solve the following problems: we need to represent expressions and we need to implement an environment. Once we have these pieces in place we can start to define the rules for the interpreter.

Before you proceed, you should think this problem through. What do sequences look like; assume that we will only handle sequences as the one shown above? What are the elements of a pattern matching expression, what is on the right side and what is on the left side? How should we represent a term and does it have to be different from the representation of a pattern? How should data structures be represented?

Read this section through one more time, then start to sketch on your representation. Write it down and then later compare it to the representation proposed in this exercise.

## 2 The implementation

We will build the interpreter starting with the environment, that will be implemented in a separate module. Then we will handle evaluation of expressions, pattern matching and finally a sequence of expressions.

### 2.1 The environment

Implementing an environment will be the simplest task that we have; an environment is simply a mapping from variables to data structures. If we assume that environments will be small, we can represent an environment as a list of key-value tuples. The environment {*x/foo, y/bar*} could be

represented as: `[{:x, :foo}, {:y, :bar}]` . The variables are represented by atoms, and we have here chosen to name them `:x` and `:y` but we could as well have chosen other atoms (`x12`, `:variable_x`) or integers (`1` and `2`), the important thing is that they all have unique names.

In a module `Env`, you should now define the following functions:

- `new()` : return an empty environment
- `add(id, str, env)` : return an environment where the binding of the variable `id` to the structure `str` has been added to the environment `env`.
- `lookup(id, env)` : return either {`id, str`}, if the variable `id` was bound, or `nil`
- `remove(ids, env)` : returns an environment where all bindings for variables in the list `ids` have been removed

These are all operations we need from the environment module. Test the environment by calling the functions from the Elixir shell to see that it produces the expected results. The following call should return {`:foo, 42`}:

```
Env.lookup(:foo, Env.add(:foo, 42, Env.new()))
```

## 2.2 Terms and patterns

If we only needed to represent terms consisting of atoms and cons-cells, things would be trivial. The term {`:a, :b`} could simply be represented by the Elixir term {`:a, :b`}. The problem is that we also need to represent variables. We will of course not be able to represent a variable in our target language with a variable in Elixir; we have to find a way to represent variables and make sure that we can separate them from atoms.

One solution is to represent atoms with the tuple {`:atm, a`} and variables with the tuple {`:var, v`}. The good things is then that we only have to make sure that the identifiers of variables are all different and that the identifiers of atoms are all different. An atom could of course have the same identifier as a variable with out causing any problems; the atom {`:atm, 123`} is different from the variable {`:var, 123`}.

A cons cell could be represented by a tuple {`:cons, head, tail`}. We could of course have chosen to represent cons cells as Elixir cons cells but we want to make a distinction between the representation of terms in our target language and terms in Elixir (our target language now happens to be Elixir look-alike but that is of course not always the situation).

As an exercise you can write down the representation of the term:

```
{:a, {x, :b}}
```

You will have to choose identifiers for the atoms `:a`, `:b` (why not `:a`, `:b`) and the variable `x` (why not `:x`).

The representation of patterns will be exactly the same as for terms with the only difference that we need to represent the special *don't-care* pattern. We choose the atom `:ignore` which will be exactly what we will do when we encounter the symbol.

## 2.3 Expressions

In our simple language an expression is simply a term expression. Expressions should be evaluated to data structures and the question is of course how these data structures should be represented.

The nice thing with a meta-interpreter is that the data structures of the interpreted language could be mapped directly to the data structures of the implementation language. An atom will thus be represented by an Elixir atom, and a cons structure of the Elixir tuple i.e. {`:a, :b`}.

Create a module called `Eager` (for reasons that will be given later) and implement a function `eval_expr/2` that takes an expression and an environment and returns either {`:ok, str`}, where `str` is a data structure, or `:error`. An error is returned if the expression can not be evaluated. This should be a quite simple task. The following skeleton code will get you started:

```
def eval_expr({:atm, id}, ...) do ... end

def eval_expr({:var, id}, env) do
  case ... do
    nil ->
```

```
      ...
    {_, str} ->
      ...
  end
end

def eval_expr({:cons, ..., ...}, ...) do
  case eval_expr(..., ...) do
    :error ->
      ...
    {:ok, ...} ->
      case eval_expr(..., ...) do
        :error ->
          ...
        {:ok, ts} ->
          ...
      end
  end
end
```

Here are some examples that you should be able to handle.

- `eval_expr({:atm, :a}, [])` : returns `{:ok, :a}`
- `eval_expr({:var, :x}, [{:x, :}])` : returns `{:ok, :a}`
- `eval_expr({:var, :x}, [])` : returns `:error`
- `eval_expr({:cons, {:var, :x}, {:atm, :b}}, [{:x, :a}])` : returns `{:a, :b}`

Note that `eval_expr/2` returns a data structure if successful i.e. *:a*, *:foo*, or {*:a, :b*}. The last test is an effect of representing the binary tuples in our source program `{a, b}`, with the internal representation `{:cons, {:atm, :a}, {:atm, :b}}` that when evaluated will return the data structure {*:a, :b*}.

## 2.4 Pattern matching

A pattern matching will take a pattern, a data structure and an environment and return either `{:ok, env}`, where `env` is an extended environment, or the atom `:fail`.

Implement a function `eval_match/3` that implements the pattern matching. Some examples will give you an idea of what we're looking for.

- `eval_match({:atm, :a}, :a, [])` : returns `{:ok, []}`
- `eval_match({:var, :x}, :a, [])` : returns `{:ok, [{:x, :a}]}`
- `eval_match({:var, :x}, :a, [{:x, :a}])` : returns `{:ok, [{:x, :a}]}`
- `eval_match({:var, :x}, :a, [{:x, :b}])` : returns `:fail`
- `eval_match({:cons, {:var, :x} {:var, :x}}, {:cons, {:atm, :a} {:atm, :b}}, [])` : returns `:fail`

Solving the cases where the pattern is an atom or variable is quite straight forward, especially since we already have the environment module. The slightly more problematic case is when the pattern is a cons structure. Note that we first would add a binding for `:x` to `:a` and then try to match `:x` with `:b`. This will of course fail, the variable $x$ can not have two values.

The following skeleton code should lead you in the right direction. We start with the simple cases, we can ignore the case where the atoms do not match for now.

```
def eval_match(:ignore, ..., ...) do
  {:ok, ...}
end

def eval_match({:atm, id}, ..., ...) do
  {:ok, ...}
end
```

Matching a variable is only slightly more complicated, we check if it has a value and if not we add it to the environment. This skeleton code uses a special construct that you might not have seen before, the `ŝtr` variable. This is to indicate that we do not want a new variable `str` but rather use the existing variable.

```
def eval_match({:var, id}, str, env) do
  case ... do
    nil ->
      {:ok, ...}
    {_, ^str} ->
      {:ok, ...}
    {_, _} ->
      :fail
  end
end
```

Now the complicated (not so complicated) case where we match a cons pattern with a cons structure. This is where we must make sure that a variable binding in one branch is transferred to the pattern matching of the other branch.

```
def eval_match({:cons, hp, tp}, ..., env) do
  case eval_match(..., ..., ...) do
    :fail ->
      ...
    ... ->
      eval_match(..., ..., ...)
  end
end
```

And last but not least, if we can not match the pattern to the data structure we fail.

```
def eval_match(_, _, _) do
  :fail
end
```

Complete the implementation and try the examples give before.

## 2.5 Sequences

We now have all the pieces of the puzzle to implement the evaluation of a sequence. We represent a sequence as list, the first elements will of course

be pattern matching expressions but the last element is of course a regular expression. The evaluation starts with an empty environment that is extended as we proceed down the list.

Each pattern matching expressions is evaluated in two steps, first the expression on the right hand side is evaluated returning a data structure. The pattern on the left hand side is then match to the data structure resulting in an extended environment.

It is important to understand how the environment is extended. We first need to remove all bindings of variables that occur in the pattern. The evaluation of the following sequence should result in $\{c, b\}$.

```
x = a; y = b; x = :c; {x, y}
```

Here is some skeleton code that Will get you started. You need to implement the function `extract_vars/1` that returns a list of all variables in the pattern.

```
def eval_seq([exp], env) do
  eval_expr(..., ...)
end

def eval_seq([{:match, ..., ...} | ...], ...) do
  case eval_expr(..., ...) do
    ... ->
      ...
    ... ->
      vars = extract_vars(...)
      env = Env.remove(vars, ...)

      case eval_match(..., ..., ...) do
        :fail ->
          :error
        {:ok, env} ->
          eval_seq(..., ...)
      end
  end
end
```

When you have everything in place you should define a function `eval/1`, that takes a sequence and returns either `{:ok, str}` or `:error`. You should then be able to run the following query:

```
seq = [{:match, {:var, :x}, {:atm,:a}},
       {:match, {:var, :y}, {:cons, {:var, :x}, {:atm, :b}}},
       {:match, {:cons, :ignore, {:var, :z}}, {:var, :y}},
       {:var, z}]

Eager.eval(seq)
```

The query is the representation of the following expression:

```
x = :a; y = {x, :b}; {_, z} = y; z
```

# 3 Extensions

We now have an interpreter that can handle sequences of expressions but the expressions are rather simple. You should now extend the language and the interpreter to handle: *case expressions*, *lambda expressions* and named functions. In each case you need to think about how expressions are represented before thinking about how the `eval_expr/2` function is extended.

## 3.1 Case expressions

A case expression consists of an expression and a list of clauses where each clause is a pattern and a sequence. We of course need to be able to tell a case expression from any other expression so why not represent it as a tuple with a `:case` key word as the first element. A clause is simply a tuple with the key word `:clause`.

Now we extend the evaluation with a clause that can handle the case expressions.

```
def eval_expr({:case, expr, cls}, ...) do
  case eval_expr(..., ...) do
    ... ->
      ...
    ... ->
      eval_cls(..., ..., ...)
  end
end
```

The function `eval_cls/1` will take a list of clauses, a data structure and an environment. It will select the right clause and continue the execution.

```
def eval_cls([], _, _, _) do
  :error
end

def eval_cls([{:clause, ptr, seq} | cls], ..., ...) do
  ...
  ...
  case ... do
    :fail ->
      eval_cls(..., ..., ...)

    {:ok, env} ->
      eval_seq(..., ...)
  end
end
```

That's it, you should now be able to evaluate something like this:

```
seq = [{:match, {:var, :x}, {:atm, :a}},
       {:case, {:var, :x},
          [{:clause, {:atm, :b}, [{:atm, :ops}]},
           {:clause, {:atm, :a}, [{:atm, :yes}]}
          ]}
       ]

Eager.eval_seq(seq, [])
```

### 3.2 Lambda expressions

Adding lambda expressions might look very complicated but it turns out to be quite simple. We first have to find a representation but this is by now not a problem. We know that a lambda expression (unnamed function) consists of a sequence of parameter variables, a sequence of free variables and a sequence expression. If we agree to represent the parameters as well as the free variables as lists of identifiers we are done.

```
{:lambda, parameters, free, sequence}
```

Now when we evaluate a lambda expression we need to represent a closure but this is equally simple. A closure simply consists of a sequence of parameter variables, a sequence expression together with an environment.

```
{:closure, parameters, sequence, environment}
```

To evaluate a lambda expression we need to add a function to the `Env` module that creates a new environment from a list of variable identifiers and an existing environment. If we can do this the rest is simple.

```
def eval_expr({:lambda, par, free, seq}, ...) do
  case Env.closure(free, ...) do
    :error ->
      :error
    closure ->
      {:ok, {:closure, ..., ..., ...}}
  end
end
```

The only thing we now have left is to function application i.e. when we apply a closure to a sequence of argument expressions. We need a way to represent this and the most natural way is the best. Note that we have an expression in the structure. The closure is something we will hopefully have as a result of evaluating the expression.

```
{:apply, expression, arguments}
```

The evaluation should first evaluate the expression. If this is indeed a closure we can evaluate the arguments and apply the closure to the resulting list of data structures.

```
def eval_expr({:apply, expr, args}, ...) do
  case ... do
    :error ->
      :error
    {:ok, {:closure, par, seq, closure}} ->
      case ... do
        :error ->
          :foo
```

```
        strs ->
          env = Env.args(par, strs, closure)
          eval_seq(seq, env)
      end
  end
end
```

You have to extend the `Env` module to include a function `args/3` that is given a list of variable identifiers (`par`) , a list of data structures (`strs`) and an environment (that includes the values of all free variables). The environment that is returned should now include bindings for all the variables in the sequence of the closure. If you get it right we should be able to evaluate the following.

```
seq = [{:match, {:var, :x}, {:atm, :a}},
       {:match, {:var, :f},
          {:lambda, [:y], [:x], [{:cons, {:var, :x}, {:var, :y}}]}},
       {:apply, {:var, :f}, [{:atm, :b}]}
      ]

Eager.eval_seq(seq, [])
```

### 3.3 Named functions

You're now a small step from being able to handle named functions i.e. a program. What we need is a key-value store that given a function identifier (an atom), returns a structure that holds a list of parameters and a sequence. We store this in a list and can use the library function `List.keyfind/3` to retrieve the right function.

Since we now have a program we need to give each function access to this data structure. This means that we need to change the `eval_expr/2` function to take a third argument, the program. This value must also be passed to `eval_seq/2` and `eval_cls/3`. If you think this is a tedious task you have just encountered the downside of not having global data structures.

Change the program and add the following clause to handle named functions.

```
def eval_expr({:call, id, args}, env, prg) when is_atom(id) do
  case List.keyfind(prg, id, 0) do
    ... ->
      ...
    {_, par, seq} ->
      case eval_args(..., ..., prg) do
        :error ->
          :error

        strs ->
          env = Env.args(..., ..., ...)
          eval_seq(..., ..., prg)
      end
  end
end
```

If everything works you should now be able to run the following program:

```
prgm = [{:append, [:x, :y],
           [{:case, {:var, :x},
               [{:clause, {:atm, []}, [{:var, :y}]},
                {:clause, {:cons, {:var, :hd}, {:var, :tl}},
                   [{:cons,
                      {:var, :hd},
                      {:call, :append, [{:var, :tl}, {:var, :y}]}}]}
                }]
            }]
         }]

seq = [{:match, {:var, :x},
       {:cons, {:atm, :a}, {:cons, {:atm, :b}, {:atm, []}}}},
     {:match, {:var, :y},
       {:cons, {:atm, :c}, {:cons, {:atm, :d}, {:atm, []}}}},
     {:call, :append, [{:var, :x}, {:var, :y}]}
    ]

Eager.eval_seq(seq, [], prgm)
```