# A Meta-Interpreter

Ayda Ghalkhanbaz

Spring Term 2024

## Introduction

The first higher grade assignment is about to create an interpreter for a functional programming language using Elixir. This report covers the implementation details of the program in multiple sections and concludes with some test results indicating the interpreter's functionality.

## Implementation

An interpreter is a program which takes another program as argument, then interprets and executes the instructions of the program without compiling it. Implementing a simple interpreter for this task is broken down into three different parts where each is explained comprehensively in separate sections below.

### Environment

The first step of implementing the interpreter involved creating an environment represented as a list of tuples where each tuple comprises a key and its corresponding value. This environment is mostly similar to the previous task and includes the following functions:

1. `new/0`: creates an environment
2. `add/3`: adds a new pair to the environment
3. `lookup/2`: searches through the environment to find the key, returns `nil` if the key doesn't exist
4. `remove/2`: deletes a key-value pair from the environment

However it follows with some slight modifications in the `add/2` function and also employs two more functions: `closure/2` and `args/3`. The `add/2` function now can handle situations like adding a key-value pair that the key already exists in the list bound to another value. In such a case, the

`add/2` function deletes all the old bindings and adds the new key-value pair to the list. The functionality of the `closure/2` and `args/3` functions will make much more sense when the Eager program is explained but for better clarity, they are explained briefly here.

The `closure/2` function takes a list of variables and an environment as arguments. This function then checks whether all these variables exist in the environment, if not, then it returns `:error`.

The `args/3` function takes 3 arguments: a list of parameters, a list of structures and an environment. It then pairs the parameters and the structures in a tuple and adds these tuples to the environment. The code snippet below demonstrates the `add/3`, `closure/2` and `args/3` functions.

```
def add(id, str, []) do [{id,str}] end
def add(id, str, env) do [{id,str}|remove(id,env)] end

def closure([], env) do env end
def closure([var|tail], env) do
  case lookup(var, env) do
    nil -> :error
    {_,_} -> closure(tail, env)
  end
end

def args([], _, env) do env end
def args([param|params], [struct|structs], env) do
  args(params, structs, [{param, struct}|env])
end
```

## Eager

The second program, Eager, is the heart of the interpreter since it performs all the evaluations and interpretation. This program can be divided into further parts including evaluation of expressions (terms/data structures), pattern matching and evaluating sequences (which can be either an `expression` or in format of `match;expression`).

The program has different evaluation functions depending on the type of expression being interpreted. For instance, the `eval_expr/2` function takes a tuple and an environment. This tuple might indicate an atom, variable, cons, case or a lambda expression. In each case, `eval_expr/2` is designed to evaluate the expression according to the rules provided in the assignment. The implementation of the program was straightforward due to the provided

rules and code skeleton; however, a clear understanding of the rules was crucial for successful implementation.

Table 1 presents the functionality of the `eval_expr/2` function for different expressions.

Table 1: A list of `eval_expr/2` functions for evaluating the different expressions in an arbitrary environment

| **Expression Evaluation** | |
|---|---|
| **Function** | **Functionality** |
| `eval_expr({:atm, id}, _)` | simply returns the atom |
| `eval_expr({:var, id}, env)` | using `Env.lookup/2`, returns `:error` if the variable isn't in the environment otherwise the corresponding structure is returned. |
| `eval_expr({:cons, head, tail}, env)` | calls `eval_expr/2` recursively to evaluate both of the expressions, if it fails in any step, `:error` is returned |
| `eval_expr({:case, expr, cls}, env)` | evaluates the expression using the recursive call of `eval_expr/2`, in case of succession, `eval_cls/3` is called to evaluate the clauses; otherwise returns `:error`. |
| `eval_expr({:lambda, parameter, free, seq}, env)` | evaluates a lambda expression and creates a closure with the given parameters, sequence and the environment. |
| `eval_expr({:apply, expr, args}, env)` | this is for the function application. Evaluates the expression with a recursive call of `eval_expr/2`. In case of succession the arguments will be evaluated by calling `eval_args/2`. |

**Expression Evaluation (continued)**

| Function | Functionality |
|---|---|
| `eval_expr({:fun, id}, _)` | handles the named functions by calling an Elixir function, `apply/3`, that returns a representation of the parameters and its sequences |

Similar to the `eval_expr/2`, the `eval_match/3` function is responsible for handling different types of expressions for pattern matching. The only difference is that `eval_match/3` handles "*don't care*" as well. The code snippet below illustrates the `eval_match/3` function for `:ignore` and `:cons` structures.

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

def eval_match({:cons, pattern1, pattern2},
                              {struct1, struct2}, env) do
    case eval_match(pattern1, struct1, env) do
        :fail -> :fail
        {:ok, env} -> eval_match(pattern2, struct2, env)
    end
end
```

The `eval_seq/2` function is the last important part of the implementation. It takes a sequence of pattern matching expressions as an argument and evaluates each of these expressions by calling the `eval_expr/2`. In case of succession, a new environment without the bindings for the variables will be created and then `eval_match/3` is called to perform the pattern matching with the structures in the new environment.

Below you'll find the implementation for the `eval_seq/2` function. Additionally, table 2 presents other helper functions that have been employed during this program.

```
def eval_seq([exp], env) do eval_expr(exp, env) end
def eval_seq([{:match, pattern, expr} | seq], env) do
    case eval_expr(expr, env) do
      :error -> :error
      {:ok, str} -> new_env = eval_scope(pattern, env)
```

```
        case eval_match(pattern, str, new_env) do
          :fail -> :error
          {:ok, env} -> eval_seq(seq, env)
        end
    end
end
```

Table 2: A list of helper functions for evaluating the different types in an arbitrary environment

| **Helper Functions** | |
|---|---|
| **Function** | **Functionality** |
| `extract_vars(pattern)` | accepts a pattern and calls the `extract_vars/2` and at the end returns a list of all variables found in the pattern list. |
| `extract_vars({:var, var}, list)` | a variable hit results in adding the variable to the list and return the updated list. |
| `extract_vars({:cons, head, tail}, list)` | in case of a cons, `extract_vars/2` is recursively called to check whether the head and tail of the cons expression include any variables. |
| `eval_scope(pattern, env)` | returns an updated environment without any variables by utilising `extract_vars/1` and `Env.remove/2`. |

**Helper Functions (continued)**

| Function | Functionality |
|---|---|
| `eval_cls([{:clause, pattern, seq} \| clauses], struct, env)` | takes in 3 arguments; a list of clauses, an environment and structures. Evaluates the clauses in the case expressions by employing `eval_match/3` and `eval_scope/2`. In case of succession, `eval_seq/2` is called for evaluating the sequence of the clause. Otherwise, the function is recursively called to evaluate the rest of clauses. |
| `eval_args(args, env)` | takes in a list of arguments together with an environment, and calls the `eval_args/3`. |
| `eval_args([arg\|tail], env, structs)` | calls the `eval_expr/2` with one argument from the list at time and evaluates it. In case of succession, it continues evaluating the rest of the list by calling itself recursively and adds the corresponding structure to a new list. Otherwise; returns `:error`. |

### Program

Finally, as the last part of this assignment, a module named *Prgm* was created which contains an already provided code for testing the interpreter's functionality. The results of this test are presented in the next section.

## Result

The interpreter's functionality has been confirmed by running the following test.

The Evaluated Sequence

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

```
{:match, {:var, :y},{:cons, {:atm, :c},
                     {:cons, {:atm, :d}, {:atm, []}}}},
{:apply, {:fun, :append}, [{:var, :x}, {:var, :y}]}]
```

..... Evaluation Result of the Sequence .....

```
iex(70)> Eager.eval_seq(seq, Env.new())
{:ok, {:a, {:b, {:c, {:d, []}}}}}
```

---