Meta Interpreter

Aditya Gupta

February 2023

Introduction

In this assignment, we implement an interpreter for a language that can work like a functional subset of an elixir with some differences from how Elixir is defined. This interpreter is not a proper meta-interpreter as it doesn't actually interpret a complete language but we implement enough functionality in it for it to be considered close to one.

Implementation

Our interpreter takes "*terms*" as inputs and turns them into data structures upon evaluation. These *terms* are restricted to just atoms, variables and the binary structure called *cons*. We first write programs that can evaluate terms which we then extend for more functionality. We also have patterns which used the same syntax as terms but it also includes the use of an underscore (_) to depict a "don't care variable" which is just used as a placeholder for data structures we don't care about. Even though the syntax for both is the same, they can be distinguished based on the grammatical rules that we define for the language.

Environment

We implement an environment using a list of Tuples. We use atoms to represent variables and use unique names. We define the following functions which are used to build up the environment and access the mappings of variables inside it:

- **new():** This function returns an empty list.
- **add():** We take the key and the value as arguments and add them to the environment as a key-value pair in the form of a tuple.
- **remove():** We take the list of keys we want to remove from the environment as an argument. We then use *List.foldr()* and *List.keydelete()* to remove them from it.

- **lookup():** We call *List.keyfind* to search for the key in the environment.

Evaluating Expressions

When we evaluate expressions for atoms, we do not need to extend the environment. If we try and evaluate a variable then we search for the variable in the environment and return their value if we find one in the environment. For the *cons* structure we start at the first element and then keep going through the structure until we reach the tail and evaluate each element we encounter.

Pattern Matching

When pattern matching against :atm or :ignore we just return the unchanged environment. When we pattern-match variables in the environment then we lookup the variable in the environment and if we find a match against the provided structure then we also return the environment. If we match against a different structure then we return :fail instead. If we do not find the variable in our environment, we add it to the environment and return it. For the *:cons, head, tail* structure we call the eval_match() function on the head. If that doesn't give us a :fail then it would return a new environment. After this we call the eval_match(tail,new_env); this allows for the function to be called recursively until the head and tail arguments have been reduced down to just atoms or variables.

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

      {:ok, env} ->
        eval_match(tp, ts, env)
    end
end
```

Sequences

The sequence we pass as the argument for the eval_seq(seq,env) function is comprised of some pattern matching expressions followed by the expression we want to return at the end. This function evaluates all the pattern matching expressions, and alters the scope or the environment at the same time. At the end we just evaluate the last expression after the final changes to the environment have been made.

Evaluating the pattern match

To do this we call the eval_match(pattern, str, env) function and go through the following steps:

- **Evaluate the expression**
- **Alter the scope/Environment**: We change the scope using the eval_scope() method which removes the all the variables from the environment which occur in the pattern.

  ```
  def eval_scope(pattern, env) do
      Env.remove(extract_vars(pattern), env)
  end
  ```

- **Match the pattern to the expression in the new environment**

We use the function extract_vars(pattern) to extract all the variables from the pattern given to it as an argument. In this report I show the case of using this function on a :cons structure.

```
def extract_vars({:cons, h, t}, env) do
    h_vars = extract_vars(h, env)
    extract_vars(t, h_vars)
end
```

The extract_vars/1 calls extract_vars/2 with the pattern and an empty list as an argument. This is then called recursively and we keep appending the extracted variables to the empty list given as the second argument.

Case Expressions

In *Case* we have the expression and the list of clauses, we will need to evaluate all of them. We want to first evaluate the expression in the given environment/scope using the function eval_expr(:case, expr, cls, env). If that gives us a favourable result then we start evaluating the list of clauses using the eval_cls() function, given below.

```
def eval_cls([{:clause, ptr, seq} | cls], str, env) do
    new_env = eval_scope(ptr,env)
    case eval_match(ptr,str,new_env) do
        :fail -> eval_cls(cls, str, env)
        {:ok, env} -> eval_seq(seq, env)
    end
end
```

The str parameter is the structure obtained by evaluating the expression expr from the case statement. For each clause we have a pattern and a sequence. If we manage to match the pattern to the structure then we call eval_seq() on the sequence. We call the eval_cls() function recursively. If the match fails then try the next clause. But before evaluating each pattern matching, we need to create a new environment. If we cannot find a match for the pattern with the structure then we just return an error message.

Lambda Expressions

We represent lambda expressions in the following way $\rightarrow$: $lambda, paramaters, free, sequence$. To be able to evaluate such a statement we need a value for all the free variables in the environment. Before evaluating such statements we create a closure which is acts like a smaller environment that contains only the free variables and their values.

```
def closure(keyss, env) do
    List.foldr(keyss, [], fn key, acc ->
      case acc do
        :error ->
          :error

        cls ->
          case lookup(key, env) do
            {key, value} ->
              [{key, value} | cls]

            nil ->
              :error
          end
        end
    end)
  end
```

The function closure(ids,env) goes through the list of free variables one by one, checks their values in the environment and if its found, then the tuple of {id, str} gets appended to the closure or we return an :error.

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

      closure ->
        {:ok, {:closure, par, seq, closure}}
    end
  end
```

Named Function

The output for the sequence given in the assignment is obtained upon running the following test function:-

```
    def test4 do
        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}]}
        ]

        Eager.eval_seq(seq, Env.new())
    end
```

The Output:

```
iex(4)> Eager.test4
{:ok, {:a, {:b, {:c, {:d, []}}}}}
```