Lab 4 - Interpreter

Weichen Chai

Spring 2023

Introduction

This paper seeks to answer the questions presented in"Interpreter" for ID1019. In this assignment, we will explore how the functional language Elixir can be utilized to implement an interpreter for a functional language.

Tasks

The introduction to this assignment was quite long, but fast forward to the implementation of the algorithm, we are first required to build an environment for the interpreter

Environment

Similar to previous tasks, we are only required to map variables to data-structures using key-value. We will define functions in the environment module shown below:

```
defmodule Env do
  def new() do [] end
  def add(id, str, env) do [{id, str} | env] end
  //id to str in environment
  def lookup(_, []) do nil end
  def lookup(id, [{id, str} | tail]) do {id, str} end
...
  def remove(_, nil) do [] end
  def remove(id, [{id, _}|tail]) do tail end
...
end
```

Eager

The second task is to implement the Eager module, where we deal with evaluating expressions, pattern matching and sequences. Where we are given a skeleton code to fill in where the result should either be :error or give an ok. Some instances of the completed code are shown below:

```
 def eval_expr({:atom, id}, _) do {:ok, id} end
  def eval_expr({:var, id}, env) do
    case Env.lookup(id, env) do
    //perform lookup in environment, if we dont find we retunr nil
      nil ->
        :error

    // if we find something, we return tuple with {_,str}
    //as we already know what we are looking for
      {_, str} ->
         {:ok, str}
    end
```

Pattern-Matching

A similar instance for a compound structure is also implemented. An important part of the Eager module is pattern matching, which includes :fail and gives an ok if it is an extended environment. completing the skeleton code gives:

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

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

When matching a variable, we follow the skeleton code and give the following code.

```
def eval_match({:var, id}, str, env) do
    case Env.lookup(id, env) do
    //when variable has value, if not add to environment
      nil ->
        {:ok, Env.add(id, str, env)}
      {_, ^str} -> //id doesn't matter.
      // ^str is to indicate that it is not a new str variable.
        {:ok, env}
...
def eval_match({:cons, hp, tp}, {hs, ts}, env) do
    case eval_match(hp, hs, env) do
```

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

As shown, all three cases for pattern matching have been implemented. We then move forth to look at sequences in the Eager module,

Sequence

Something we need to consider is the scope, which removes variables in the pattern from environment,

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

When implementing a sequence, it should return a data structure or an error, filling in the skeleton code given we have:

```
...
def eval_seq([{:match, ptr, exp} | tail], env) do
    case eval_expr(exp, env) do
...
      {:ok, str} ->
        env = eval_scope(ptr, env)
        // scoping for removeing knowledge of variables.
        case eval_match(ptr, str, env) do
...
          {:ok, env} ->
            eval_seq(tail, env)
        end
    end
```

Testing with test code given in the assignment outputs the correct result which indicates that the code runs properly.

Extensions

Case expression

In this section we will be looking at clauses, in order the evaluation with clauses, we finish the code given to us in the assignment, for clauses, we want to go with the first, do scoping where we forget the variables in the environment, then do pattern matching, if that fails, we go for other clauses in the environment. If succed we use it when evaluating sequences.

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

Running the testing code given gives the reply : :ok,yes showing it works as intended.

Lambda expression

Since there are no way of recursion, looping or defining a function in the current interpreter apart from the ability to pattern match, we introduce lambda expressions. We introduce free variables to an environment and ask for the return of a closure (Env.closure). the closure variable contains information which gives values of all free variables. If it is indeed a closure, we evaluate all arguments in the environment closure, if works we evaluate the sequence.

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

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

The code given fills in blanks for the code given in the assignment. And it runs properly, giving the result ok, a, b

Named functions

Finally for named functions, we can make a module named : Prgm, which contains code given to us in the assignment. Then we would need to add a term which shows that the identifier is a name of a function. The code for all the above were already give in the task, therefore we can just plug it in directly with out filling in any blanks. result:

```
{:ok, {:a, {:b, {:c, {:d, []}}}}}
//note
```