Expression

Simon Yllmark

January 2024

1 Introduction

In this task I implemented a program that evaluates expression and takes an environment to that expression as a additional arguments. An environment in this case is just an key value database that maps keys to values for example:

```
environment = new([{:x, {:num, 2}}, {:y, {:num, 3}}])
```

Thus, it enables the program to evaluate expressions like $2 * x + 5$ to be equal to 9. Just regular one variable math to put it simply.

2 Expressions

In the assignment description it say that I should give examples of that the program actually work. Hence, here is a test() module example of the expression $2x + 3 + 1/2$ written in Elixir code and the environment for x is specified.

```
  def test() do
  env = new([{:x, {:num, 2}}])
  expression = {:add, {:add, {:mul, {:num, 2}, {:var, :x}}, {:num, 3}}, {:q, {:nu
  e = eval(expression, env)
  IO.write("Evaluated Expression: #{pprint(e)}\n")
  f = lookup(env, :x)
  IO.write("Lookup: #{pprint(f)}\n")
end
```

This test() function will the print out in the iex terminal:

- Evaluated Expression: (15/2)
- Lookup: 2
- :ok

This result is correct and verified through calculations done in my head. Then I will go into how the eval() function works.

3 Evaluation

The eval() function is supposed to evaluate an expression by taking in to arguments where one is the expression and the other it's environment. Then what my program does it pattern matches the input and uses recursive programming to solve the expression by breaking it down into smaller pieces but the environment is preserved. The eval() functions are listed down below:

```
def eval({:num, n}, _) do {:num, n} end
def eval({:q, {:num,n1}, {:num,n2}}, _) do {:q, {:num,n1}, {:num,n2}} end
def eval({:var, var}, env) do lookup(env, var) end
def eval({:add, e1, e2}, env) do
  add(eval(e1, env), eval(e2, env))
end
def eval({:sub, e1, e2}, env) do
  sub(eval(e1, env), eval(e2, env))
end
def eval({:mul, e1, e2}, env) do
  mul(eval(e1, env), eval(e2, env))
end
def eval({:div, e1, e2}, env) do
  divide(eval(e1, env), eval(e2, env))
end
```

What I mean by that is for example it takes an expression like:

```
{:add, {:mul, {:num, 2}, {:var, :x}}, {:num, 3}}
```

and break it down to:

```
e1 = {:mul, {:num, 2}, {:var, :x}}
e2 = {:num, 3}
```

and e1 then get broken down to:

```
e1 = {:num, 2}
e2 = {:var, :x}
```

and the environment is given as the second argument meaning that it is preserved through all the steps. These recursions are then placed on a stack and when the program reach the last recursive call then all the stacked operation gets executed.

4 Environment

It is stated that I should use my previous experience from two earlier assignments when solving this and thus I would argue that this assignment is an extension of the derivative assignment. One of the things that makes this assignment different from the derivative is that instead of assigning a value to variable like x the program has an environment that does that which then enables the program to have multiple variables as parameters.

5 Conclusion

The environment in itself is an injective function that maps keys to value. In the beginning of this course I was kind of scared of the environment and thought it was like a super complicated thing but it's just a mapping function. After some web-searching about the subject most sources mentions two subject which are environment and scope like for example in this source. The scope of the environment in my program would be the program if the variable declaration is not set to be global.

According to this source the environment in my program is not global because I would have to create a cache which saves the specified variables and this concept is apparently called memorisation. Maybe we will discuss how to make global variables between differnt processes in this course