

Scheme-- Interpreter Worksheet:

git clone git@github.com:vkuncak/SchemeMinusInterpreter.git

or


git clone <https://github.com/vkuncak/SchemeMinusInterpreter.git>

sbt launchIDE

open file: src/main/scala/Main.sc

Just define Env, updateEnv, updateEnvRec

```
def evalRec(x: Data, env: Env): Data = {  
  x match {  
    case i: Int => i  
    case Symbol(s) => env(s) match { case Some(v) => v }  
    case List('lambda, params: List[Data], body) =>  
      ((args: List[Data]) => {  
        val paramBinding = params.map(_.asInstanceOf[Symbol].name).zip(args)  
        evalRec(body, updateEnv(env, paramBinding))  
      })  
    case List('val, Symbol(s), expr, rest) =>  
      evalRec(rest, updateEnv(env, List(s -> evalRec(expr, env))))  
    case List('def, Symbol(s), expr, rest) => {  
      evalRec(rest, updateEnvRec(env, s, expr))  
    }  
    case List('if, bE, trueCase, falseCase) =>  
      if (evalRec(bE, env) != 0) evalRec(trueCase, env)  
      else evalRec(falseCase, env)  
    case opE :: argsE => {  
      val op = evalRec(opE, env).asInstanceOf[List[Data] => Data]  
      val args: List[Data] = argsE.map((arg: Data) => evalRec(arg, env))  
      op(args) }}}}
```

 s will have value `expr` inside both `expr` and `rest`

Env, updateEnv, updateEnvRec

```
type Env = String => Option[Data]

val recEnv : Env = ((id:String) => stdEnv.get(id))

def updateEnv(env: Env, bindings: List[(String,Data)]): Env = bindings match {
  case Nil => env
  case (id,d)::rest => ((x:String) =>
    if (x==id) Some(d)
    else updateEnv(env,rest)(x))           // non-recursive - val
}

def updateEnvRec(env: Env, s: String, expr: Data) : Env = {
  def newEnv: Env = ((id:String) =>
    if (id==s) Some(evalRec(expr, newEnv)) // recursive - def
    else env(id)
  )
  newEnv
}
```

The “Lab Version” of Interpreter - eval

sbt launchIDE

open file: `src/main/scala/LabLikeInterpreter.sc`

A variation of the interpreter presented in slides so far:
also supports recursive definitions

What are the differences?

Representing Environment

First versions: Map[String, Data]

evalRec: String => Option[Data]

eval:

```
abstract class Environment {  
  def lookup(n: String): Data  
  def extend(name: String, v: Data): Environment  
  def lookup(n: String): Data  
  def extendRec(name: String, expr: Environment => Data)  
}
```

Environment is Given by Updates

```
abstract class Environment {  
  def extend(name: String, v: Data): Environment = new Environment {  
    def lookup(n: String): Data = {  
      if (n == name) v else Environment.this.lookup(n)    // outer class call  
    }  
  }  
}
```

Above is an “OOP way” of doing the following:

```
type Env = String => Option[Data]  
def extend(env: Env, name: String, v: Data): Env = {  
  (n: String) => if (n==name) v else env(name)  
}
```

Recursive Update

```
def extendRec(name: String, expr: Environment => Data) = new Environment {  
  def lookup(n: String): Data =  
    if (n == name) expr(this)           // expr evaluated in extended environment  
    else Environment.this.lookup(n)  
}
```

the 'def' case in 'eval':

```
case 'def :: Symbol(name) :: body :: rest :: Nil =>  
  eval(rest, env.extendRec(name, env1 => eval(body, env1)))
```

Mutable Global Environment

```
evaluate("(def giveNumber (lambda (x) 42))")
```

```
evaluate("(giveNumber 0)") // 42
```

```
evaluate("(def giveNumber (lambda (x) 72))")
```

```
evaluate("(giveNumber 0)") // 72
```


Mutable Global Environment

Global definition supports read-eval-print loop (REPL) using a 'def' that mutates a global environment

We write this as a 'def' that has no 'rest' expression:

```
case 'def :: Symbol(name) :: body :: Nil => // 'rest' expression is Nil
  if (env == globalEnv) { // definition is global, not nested
    globalEnv = env.extendRec(name, env1 => eval(body, env1))
```

As for Scala REPL, the behavior of eval(expr) then depends on which definitions were evaluated in the past, and in which order

Logic Programming and Prolog

Beyond Functional Programming

Functional programming is

- declarative: we can think of functions as mathematical functions
- has simple execution model, which is basis of implementation
- we can estimate the running time of functions

More ambitious approach: state **properties** of what we want to compute as opposed to directly stating **how** to compute them

One approach is followed in logic programming and its implementation - Prolog

- state facts and rules using first-order logic formulas
- **computation** reduces to **deduction** (logical inference)

Prolog Implementations

SWI-prolog - download, install, run 'prolog' from terminal
to load a file, use
`consult('filename.pl')`.

Interpreter written in Scala

git clone <https://github.com/vkuncak/SchemeMinusInterpreter.git>
sbt launchIDE

open workshet: `src/main/scala/Prolog.sc`