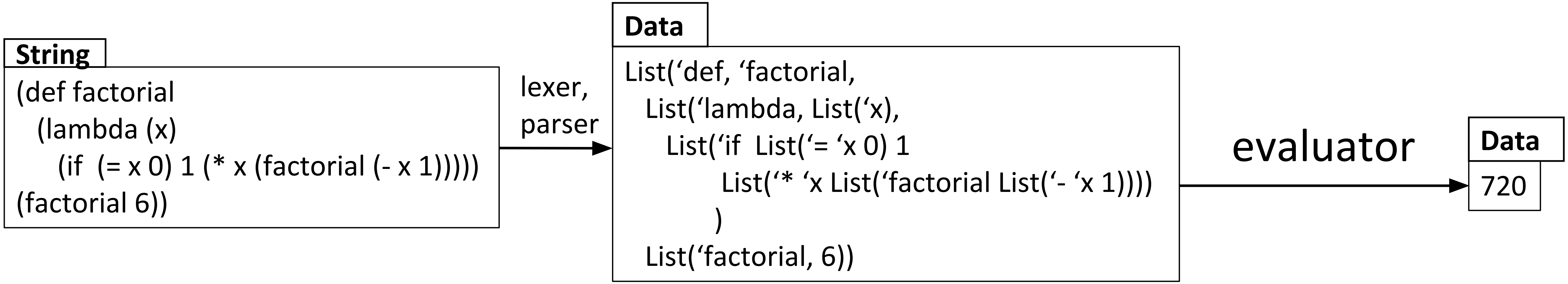
Interpreting Intermediate Representation of Scheme--

We have seen how to transform strings into nested lists representing programs

Now we write functions that run evaluate (run) program by recursively traversing these nested lists



String

```
(def factorial
   (lambda (x)
      (if  (= x 0) 1 (* x (factorial (- x 1)))))
(factorial 6))
```

lexer, parser

Data

```
List(‘def, ‘factorial,
   List(‘lambda, List(‘x),
      List(‘if  List(‘= ‘x 0) 1
              List(‘* ‘x List(‘factorial List(‘- ‘x 1))))
             )
   List(‘factorial, 6))
```

evaluator

Data

720

Worksheet Project

```
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

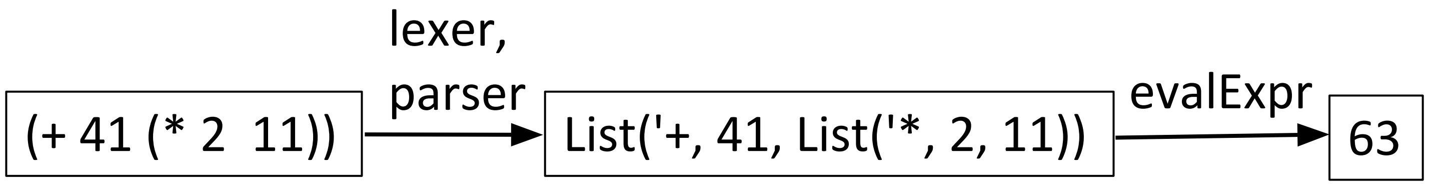
Growing an Interpreter from the Simplest One

A sequence of interpreters, for increasingly more general expressions:

- **evalExpr**: constant numbers, +, *

  (+ 41 (* 2 11))
- **evalSym**: symbols and environment
- **evalFun**: general function application; **if** special form
- **evalVal**: non-recursive definitions
- **evalLambda**: anonymous functions (lambda expressions)
- **evalRec**: recursion
- **eval1**: alternative definition of environment, better checks
- **eval**: debug output of evaluator

evalExpr: constant numbers, +, *



(+ 41 (* 2 11))

lexer,
parser

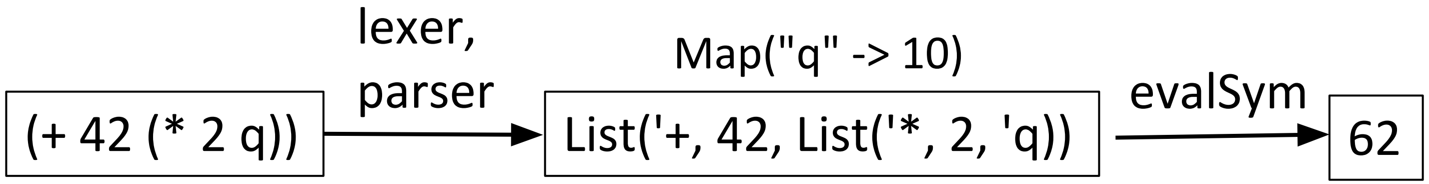
List('+, 41, List('*, 2, 11))

evalExpr

63

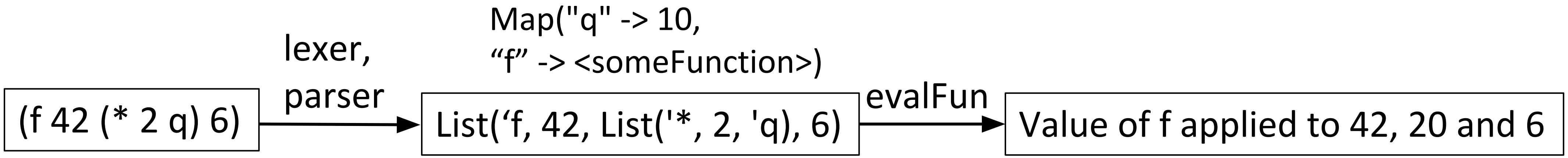
```
def evalExpr(x: Data): Data = {    // (+ (+ 2 5) 8)
   x match {
     case i: Int => i
     case List('+, arg1, arg2) => (evalExpr(arg1), evalExpr(arg2)) match {
       case (x1: Int, x2: Int) => x1 + x2
       case (v1, v2) => sys.error("+ takes two Ints, invoked with " + v1 + " and " + v2)
     }
     case List('*, arg1, arg2) => (evalExpr(arg1), evalExpr(arg2)) match {
       case (x1: Int, x2: Int) => x1 * x2
       case (v1, v2) => sys.error(“* takes two Ints, invoked with " + v1 + " and " + v2)
     }
     case _ => sys.error("Did not know how to evaluate " + x)
   }
 }
```

evalSym: symbols and environment



```
def evalSym(x: Data, env: Map[String, Data]): Data = {
   x match {
     case i: Int => i
     case Symbol(s) => env.get(s) match {
       case Some(v) => v
       case None    => sys.error("Could not find " + s + " in the environment.")
     }
     case List('+, arg1, arg2) => (evalSym(arg1, env), evalSym(arg2, env)) match {
       case (x1: Int, x2: Int) => x1 + x2
     }
     case List('*, arg1, arg2) => (evalSym(arg1, env), evalSym(arg2, env)) match {
       case (x1: Int, x2: Int) => x1 * x2
     }
   }
 }
```

Function Application



```
case Symbol(s) => env.get(s) match {
  case Some(v) => v
}
```

s can also be +, *

old case

```
~~case List('+, arg1, arg2) => (evalSym(arg1, env), evalSym(arg2, env)) match {~~
~~  case (x1: Int, x2: Int) => x1 + x2~~
}
```

general

```
~~case List(fExp, arg1, arg2,..., argN) => {~~
~~val f = evalSym(fExp, env)~~
~~  f(evalSym(arg1, env), evalSym(arg2, env),...,evalSym(argN, env))~~
}
```

informal: types, dots

```
case fExp :: argsE => {
   val f = evalFun(fExp, env).asInstanceOf[List[Data] => Data]
   val args: List[Data] = argsE.map((arg: Data) => evalFun(arg, env))
   f(args)
}
```

Standard Environment

```
val stdEnv : Map[String,Data] = {
   val plus = (args: List[Data]) => (args match {
     case List(x: Int, y: Int) => x + y
     case _                    => sys.error("plus expects two integers, applied to " + args)
   })
   val times = (args: List[Data]) => args match {
     case List(x: Int, y: Int) => x * y
   }
   val minus = (args: List[Data]) => args match {
     case List(x: Int, y: Int) => x - y
   }
   val equality = (args: List[Data]) => args match {
     case List(x, y) => if (x == y) 1 else 0
   }
   Map("+" -> plus, "*" -> times, "-" -> minus, "=" -> equality)
 }
```

evalFun(List(‘=, 30, List('*, 2, 'q)),
         stdEnv + (“q” -> 15))

if Cannot be a function in the environment: it does not always evaluate all arguments!

```
def evalFun(x: Data, env: Map[String, Data]): Data = {
   x match {
     case i: Int => i
     case Symbol(s) => env.get(s) match {
       case Some(v) => v
     }
     case List('if, bE, trueCase, falseCase) =>
       if (evalFun(bE, env) != 0) evalFun(trueCase, env)
       else evalFun(falseCase, env)
     case opE :: argsE => {
       val op = evalFun(opE, env).asInstanceOf[List[Data] => Data]
       val args: List[Data] = argsE.map((arg: Data) => evalFun(arg, env))
       op(args)
     }
   }
 }
```

only zero is
treated as false

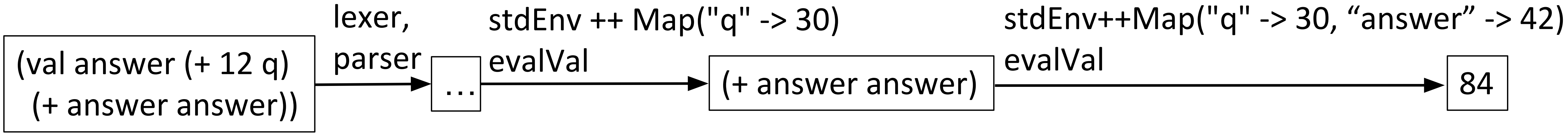
for function application,
all arguments always evaluated
(call by value)

evalVal: non-recursive definitions

```
def evalVal(x: Data, env: Map[String, Data]): Data = {
   x match {
     case i: Int => i
     case Symbol(s) => env.get(s) match {
       case Some(v) => v
       case None    => sys.error("Unknown symbol " + s)
     }
     case List('val, Symbol(s), expr, rest) =>
      evalVal(rest,
                      env + (s -> evalVal(expr, env)))
   ...
     case opE :: argsE => {
       val op = evalVal(opE, env).asInstanceOf[List[Data] => Data]
       val args: List[Data] = argsE.map((arg: Data) => evalVal(arg, env))
       op(args)
```

Corresponds to this in Scala:
{ **val** s = expr;
 rest }

`s` is known to have value `expr` inside `rest`


(val answer (+ 12 q)
 (+ answer answer))

lexer,
parser

...

stdEnv ++ Map("q" -> 30)
evalVal

(+ answer answer)

stdEnv++Map("q" -> 30, “answer” -> 42)
evalVal

84

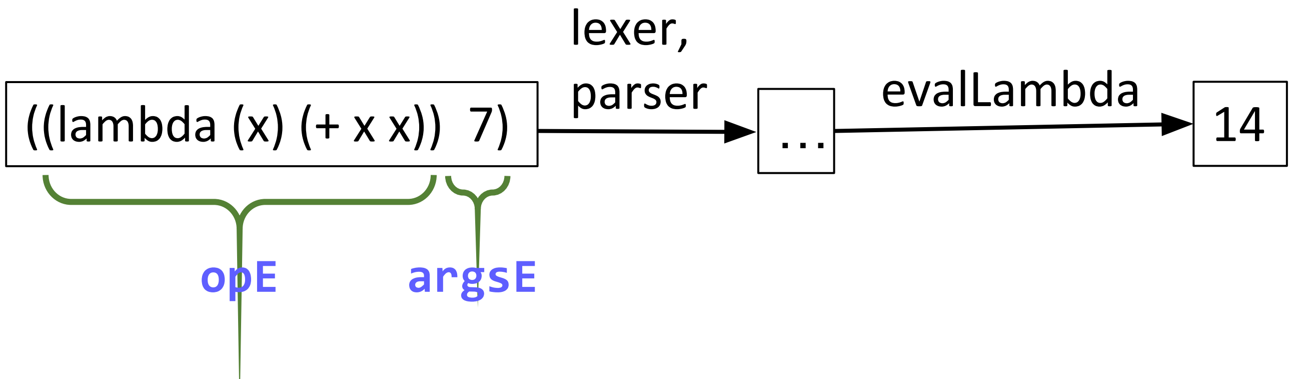
Anonymous Functions

```
def evalLambda(x: Data, env: Map[String, Data]): Data = {
  x match {
```

want to create our own values to be used as **opE**

evaluate to **op**, a function from a list of arguments

```
    case opE :: argsE => {
      val op = evalFun(opE, env).asInstanceOf[List[Data] => Data]
      val args: List[Data] = argsE.map((arg: Data) => evalLambda(arg, env))
      op(args)
```



((lambda (x) (+ x x)) 7)

lexer, parser

. . .

evalLambda

14

opE

argsE

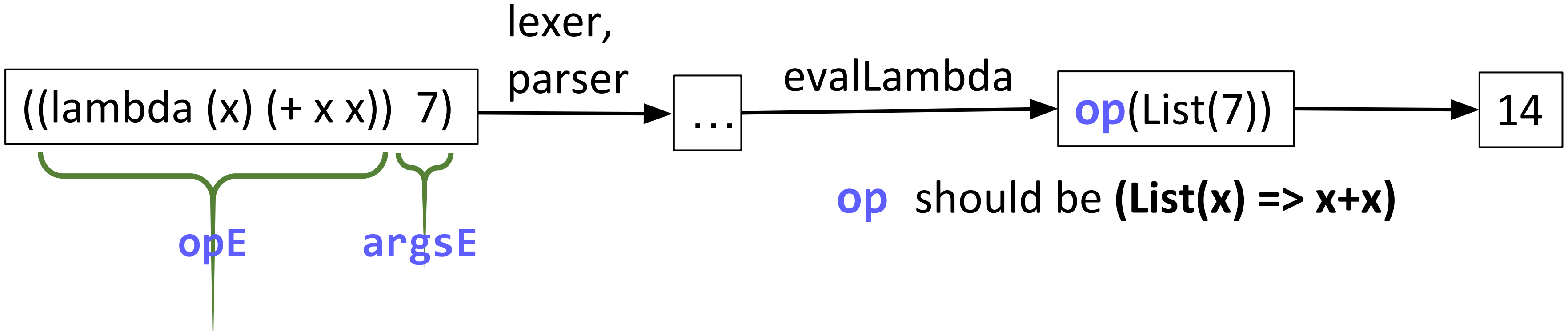
Towards anonymous functions (lambda expressions)

```
def evalLambda(x: Data, env: Map[String, Data]): Data = {
   x match {




     case List('lambda, params: List[Data], body) =>
       ((args: List[Data]) => {
         evalLambda(body,???)
       })
     case opE :: argsE => {
       val op = evalLambda(opE, env).asInstanceOf[List[Data] => Data]
       val args: List[Data] = argsE.map((arg: Data) => evalLambda(arg, env))
       op(args)
```

when evaluating **body**, it must know that **params** are bound to args



((lambda (x) (+ x x)) 7)

lexer, parser

…

evalLambda

op(List(7))

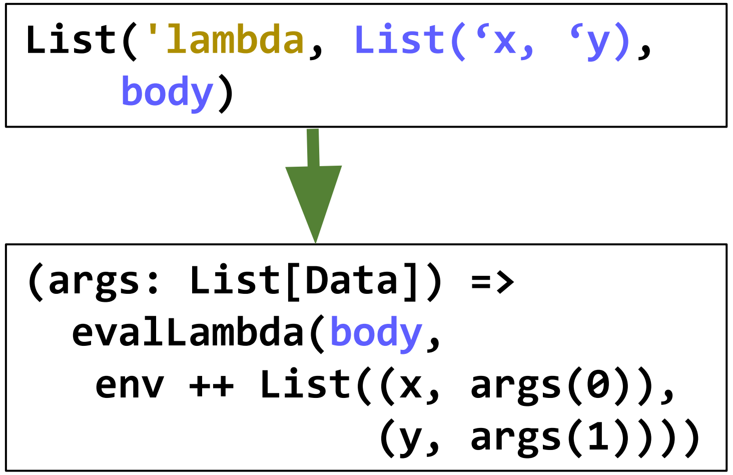
14

opE

argsE

op should be **(List(x) => x+x)**

evalLambda: anonymous functions (lambda expressions)



```
List('lambda, List(‘x, ‘y),
    body)
```

```
(args: List[Data]) =>
  evalLambda(body,
   env ++ List((x, args(0)),
               (y, args(1))))
```

```
def evalLambda(x: Data, env: Map[String, Data]): Data = {
   x match {




     case List('lambda, params: List[Data], body) =>
       ((args: List[Data]) => {
         val paramBinding = params.map(_.asInstanceOf[Symbol].name).zip(args)
         evalLambda(body, env ++ paramBinding)
       })
     case opE :: argsE => {
       val op = evalLambda(opE, env).asInstanceOf[List[Data] => Data]
       val args: List[Data] = argsE.map((arg: Data) => evalLambda(arg, env))
       op(args)
     }}}
```

```
List(“x”, “y”).zip(List(10,5)) == List((“x”,10), (“y”,5))
```

Interpreter so far: numbers, names, ifs, lambda calculus

```
(val dup (lambda (x) (+ x x))
  (dup (dup 7))
)
```

```
28
```

```
(val dup (lambda (x) (if (= x 10) 100 (+ x x)))
  (dup (dup 10))
)
```

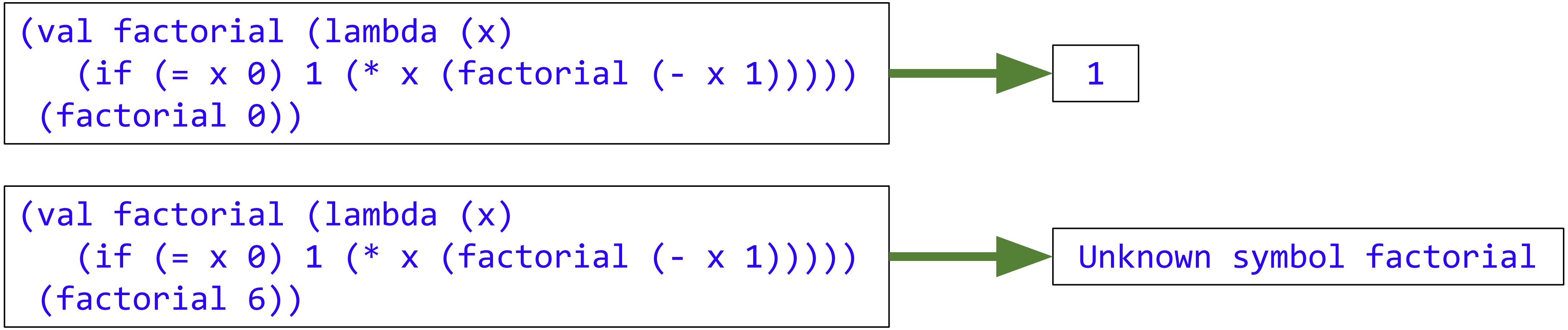
```
200
```

```
(val Z (lambda (f)
    (val comb (lambda (x)
              (f (lambda (v)
                   ((x x) v))))
    (comb comb)))
(val factorial (lambda (fact) (lambda (x)
   (if (= x 0) 1 (* x (fact (- x 1))))))
((Z factorial) 6) ))
```

```
720
```

Z is slightly more complex version of Y; works for call by value
Recursion through Z is possible, but painful and inefficient.

Interpreter so far: direct recursion does not work



```
(val factorial (lambda (x)
   (if (= x 0) 1 (* x (factorial (- x 1)))))
 (factorial 0))
```

```
1
```

```
(val factorial (lambda (x)
   (if (= x 0) 1 (* x (factorial (- x 1)))))
 (factorial 6))
```

```
Unknown symbol factorial
```

val does not support recursion

```
def evalVal(x: Data, env: Map[String, Data]): Data = {
   x match {
     case i: Int => i
     case Symbol(s) => env.get(s) match {
       case Some(v) => v
       case None    => sys.error("Unknown symbol " + s)
     }
     case List('val, Symbol(s), expr, rest) =>
      evalVal(rest,
                     env + (s -> evalVal(expr, env)))
```

Corresponds to this in Scala:
{ **val** s = expr;
 rest }

s is known to have value expr inside rest

s is **not** known to have value expr inside expr itself because expr is evaluated in the original **env**

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))
}

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

Alternative: mutable environment