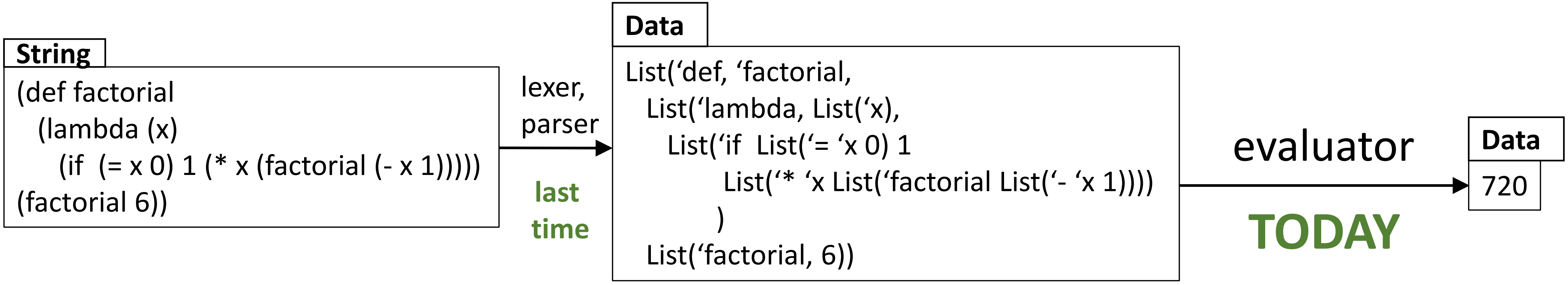
Growing an Interpreter: From Expressions to Recursive Functions

Goals

We finish presenting an interpreter for Scheme-- , a LISP-like language

- Simple LISP syntax: nested lists
- No static types: every value carries its dynamic type
- Few constructs, but sufficient to see how things work
- Higher-order and anonymous functions are supported (lambda expression)!

We work with our representation using nested lists:



Growing an Interpreter from the Simplest One

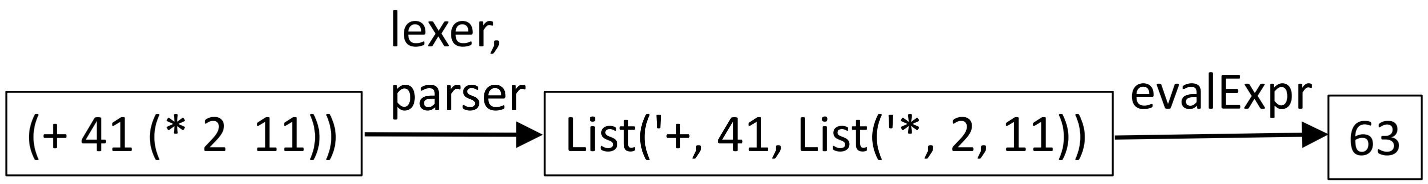
A sequence of interpreters, for increasingly more general expressions:

- **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, +, *




```
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

$$\boxed{\text{(+ 42 (* 2 q))}} \xrightarrow{\text{lexer, parser}} \boxed{\text{List('+, 42, List('*, 2, 'q))}} \xrightarrow{\text{evalSym}} \boxed{62}$$

Map("q" -> 10)

```scala
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

(f 42 (* 2 q) 6) → lexer, parser → List(‘f, 42, List('*, 2, 'q), 6) → evalFun → Value of f applied to 42, 20 and 6

Map("q" -> 10,
“f” -> <someFunction>)

```
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

```scala
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)

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

evalVal: non-recursive definitions

```scala
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

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

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

$\underbrace{\text{((lambda (x) (+ x x))}}_{\text{opE}}\ \underbrace{\text{7)}}_{\text{argsE}}$

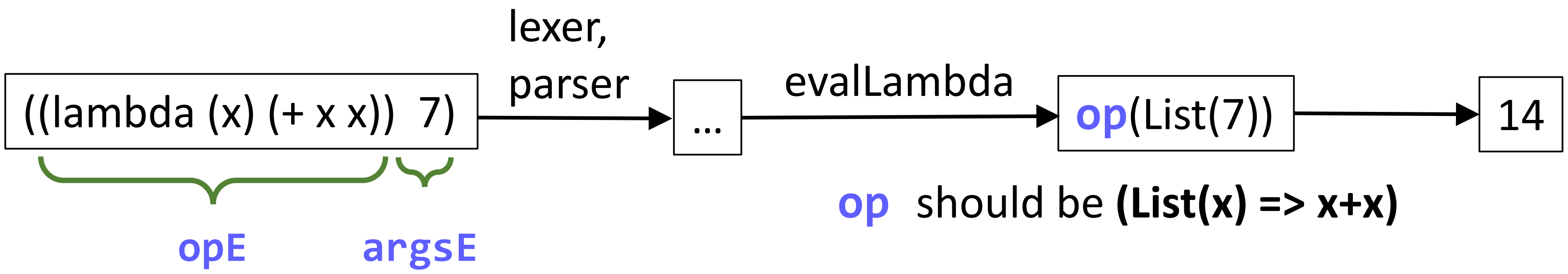
Towards anonymous functions (lambda expressions)

```
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



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

evalLambda: 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]) => {
        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('lambda, List(‘x, ‘y),
         body)
```

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

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

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

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

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

Meta-circular interpreter

- We implemented an interpreter for Scheme-- in a language Scala, which has essentially all the features of Scheme--
- To implement anonymous functions in Scheme-- we used anonymous functions in Scala
- If we mostly understood Scala, we now understand meaning of Scheme-- programs very well, because we know what the interpreter would do
- But this could be considered cheating
  - If we do not understand Scala higher-order functions, we are lost
  - If we do not have a Scala implementation, we still do not know how to build Scheme– interpreter
- Can we remove explicit use of functions passed as arguments?

How environments work: always evaluate original body

def incrementer(x:Int) = (y: Int) => x + y

val inc = incrementer(100)

inc -> [y => x+y, (x -> 100)]

inc(3)

[inc(3), **inc -> [y => x+y, (x -> 100)]**]

[**x+y**, (y -> 3, **x -> 100**)]

103

How environments work: always evaluate original body

def g = (x: Int => Int) => x(2)

def f = (x : Int) => g((y:Int) => x + y)

[f(1),	(g -> x=>x(2), f -> x=>g(y=>x+y)]
[g(y => x + y),	(x -> 1, g ->x=>x(2), f ->x=>g(y=>x+y))]
[x(2),	(x -> [y => **x + y, (x -> 1, g ->..., f ->...)**], g ->..., f ->...)]
[**x+y**,	**(y -> 2, x -> 1, g ->..., f ->...)**]
3	

A pair [e, (k1->v1,...,kn->vn)] of expression and environment is called a **closure**

Implementing closures: Example

```
def incrementer(x:Int) = (y:Int) => x + y   > incrementer: (x: Int) Int => Int
val inc = incrementer(100)                  > inc   : Int => Int = <function1>
    // inc knows by how much to increment, it stored it in environment
inc(3)                                      > res0: Int = 103
inc(5)                                      > res1: Int = 105
```

```
abstract class HasApply {         // in Scala: Function[A,B]
  def applyMe(arg: Int): Int
}
class IncrementerBody(envX: Int) extends HasApply {
  def applyMe(argY: Int):Int = envX + argY    // in Scala: apply
}
def myIncrementer(x:Int) = new IncrementerBody(x) // Scala would do it for us
val myInc = myIncrementer(100)
myInc.applyMe(3)                              > res2: Int = 103
myInc.applyMe(5)                              > res3: Int = 105
```

Now let's look at the code – see Lisp.scala

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