Have you ever...

Wanted to add/multiply a constant to an entire vector/table?

Wanted to have an easier way to deal with vectors?

Wanted to do more with vectors/matrices?

J Functionalities

Hong Giap Tee
Zhengyang Feng

Overview

- Some background of J
  - What is J
  - Basic of J
- Implementation
- Demo
- Something interesting
- To do

What is J?

According to Wikipedia:

- J is a programming language
- J is a synthesis of APL, FP, and FL
- J is a very concise array programming language
- J is free and open-source

Basics of J

Nouns:

- Data types
  - Scalars
    - Single numbers/characters
  - Vectors
    - List of scalars
  - Matrices
    - List of Vectors
  - Higher-dimensional arrays
    - List of Matrices/Higher-dimensional arrays
  - Compatible with each other
    - Eg. multiply scalar to vectors as long as they are reasonable

Basics of J

Verbs:

- Operators
  - Monadic
    - Takes in 1 argument
  - Dyadic
    - Takes in 2 arguments
  - Overloaded
    - Symbols have 2 different (usually related, but sometimes inverse) functions depending on the number of arguments
  - Does not have an order of operation
    - Computed from right to left
    - Eg. 3*2+1=9

Jala!

J-ish Interpreter in Scala

Implementation

- Multi-dimensional arrays!

```
class VVector (val l:List[Value]) extends Value {
  ...
  override def isVector () : Boolean = true
  override def getList () : List[Value] = l
}
```

Implementation

JALA > 1 2 + 3

[4 5]

- Compatible operations!

```
def operPlus (vs:List[Value]) : Value = {
  val v1 = vs(0) val v2 = vs(1)
  if (v1.isVector() && v2.isVector()) { // both vector
    if (v1.getList().length == v2.getList().length) {
      var result : List[Value] = List()
      for ((entry1,entry2) <- v1.getList().zip(v2.getList())) {
        result = result :+ operPlus(List(entry1,entry2))
      }
      return new VVector(result)
    } else { runtimeError("vectors of different length") }
  } else if (v1.isVector() && !(v2.isVector())) {  // single vector
    return new VVector(v1.getList().map((v:Value) => operPlus(List(v,v2))))
  } else ...
}
```

Implementation

- Monadic and Dyadic operations!

```
val dyadicOpt = Map(
  "+" -> DyadicOps.operPlus _,
  "-" -> DyadicOps.operMinus _,
  "*" -> DyadicOps.operTimes _,
)

val monadicOpt = Map(
  "-" -> MonadicOps.operMinus _,
  "+/" -> MonadicOps.operSum _
)
```

Implementation

- Monadic and Dyadic operations!

```
def MONADIC : Parser[String]  =
  ( SUM | MINUS ) ^^ { s => s }

def DYADIC : Parser[String]  =
  ( PLUS | MINUS | TIMES ) ^^ { s => s }

def mexpr : Parser[Exp] = MONADIC ~ dexpr ^^ {
  case e1 ~ e2 =>  new EApply(new ELiteral(
    new VPrimOp(Shell.monadicOpt(e1))),List(e2)
  )
}
```

Implementation

JALA > 3 * 2 + 1

9

- Monadic and Dyadic operations!

```
def dexpr : Parser[Exp] = fexpr ~ rep(DYADIC ~ fexpr) ^^ {
  case e ~ Nil => e
  case e1 ~ e2 => expandDexpr(e1, e2.map(x => (x._1, x._2)))
}

def expandDexpr(e1: Exp, e2 :List[(String, Exp)]) : Exp = {
 if (e2.length == 1) new EApply(new ELiteral(new VPrimOp(
    Shell.dyadicOpt(e2.head._1))), List(e1, e2.head._2))
 else {
   val er = e2.dropRight(1)
   expandDexpr(e1, e2.dropRight(2) :+
     (er.last._1, new EApply(new ELiteral(
       new VPrimOp(Shell.dyadicOpt(e2.last._1))),
     List(er.last._2, e2.last._2))))
 }
}
```

Implementation

- Infix operators!

E = F { D F }
F = v | M E | “(” E “)”

D = “+” | “-” | “*” // Dyadic
M = “-” | “+/” // Monadic

Implementation

- Element construction!
- Dyadic operator $

```
JALA > 3 4 $ 5 6 7

[ [ 5 6 7 5 ] [ 6 7 5 6 ] [ 7 5 6 7 ] ]
```

```
def operShape(vs: List[Value]) : Value
```

Implementation

- Assignment!

JALA > A =. 3 4 $ 5 6 7

JALA > A

[[5 6 7 5] [6 7 5 6] [7 5 6 7]]

```
def assign_entry : Parser[ShellEntry] =
  ID ~ ASSIGN ~ expr ^^ {
    case id ~ _ ~ e => new SEdefine(id, e)
  }
```

Implementation

- Verb train!

```
JALA > (+ +/) 2 3

[ 7 8 ]

JALA > 5 (% +/) 2 3      NB. % = dyadic; divide

1

JALA > (+/ % #) 1 2 3    NB. # = monadic; No. of

2
```

Implementation

```
  def mhexpr : Parser[Exp] = HOOK ~ dexpr ^^ {
      case h ~ e => new EApply(new ELiteral(new VPrimOp(Shell.dyadicOpt(h(0)))),
          List(e, new EApply(new ELiteral(new VPrimOp(Shell.monadicOpt(h(1)))),List(e))))
  }

  def mfexpr : Parser[Exp] = FORK ~ dexpr ^^ {
      case f ~ e => new EApply(new ELiteral(new VPrimOp(Shell.dyadicOpt(f(1)))),
          List(new EApply(new ELiteral(new VPrimOp(Shell.monadicOpt(f(0)))),List(e)), new EApply(new ELiteral(new VPrimOp(Shell.monadicOpt(f(2)))),List(e))))
  }

  def mcexpr : Parser[Exp] = COMB ~ dexpr ^^ {
      case f ~ e => new EApply(new ELiteral(new VPrimOp(Shell.monadicOpt(f(0)))),
          List(new EApply(new ELiteral(new VPrimOp(Shell.monadicOpt(f(1)))),List(e))))
  }

  def expandDexpr(e1: Exp, e2 :List[(List[String], Exp)]) : Exp = {
      e2.last._1.length match {
          case 1 =>
              if (e2.length == 1)
                  new EApply(new ELiteral(new VPrimOp(Shell.dyadicOpt(e2.head._1.head))), List(e1, e2.head._2))
              else {
                  val er = e2.dropRight(1)
                  expandDexpr(e1, e2.dropRight(2) :+ (er.last._1, new EApply(new ELiteral(new VPrimOp(Shell.dyadicOpt(e2.last._1.head))), List(er.last._2, e2.last._2))
))
              }
          case 2 =>
              if (e2.length == 1)
                  new EApply(new ELiteral(new VPrimOp(Shell.dyadicOpt(e2.head._1.head))), List(e1,
                      new EApply(new ELiteral(new VPrimOp(Shell.monadicOpt(e2.head._1.last))), List(e2.head._2))
                  ))
              else {
                  val er = e2.dropRight(1)
                  expandDexpr(e1, e2.dropRight(2) :+ (er.last._1, new EApply(new ELiteral(new VPrimOp(Shell.dyadicOpt(e2.last._1.head))), List(er.last._2,
                      new EApply(new ELiteral(new VPrimOp(Shell.monadicOpt(e2.head._1.last))), List(e2.last._2))
                  ))))
              }
      }
```

- Surface syntax

Demo

Something Interesting

- More than J-style array processing
- Functional programming
- Implementation of J programming language framework

To do

- Nested verb train

```
J > (*:@-+/ % #) 1 2 3 4 5
2
```

- Name operators

```
J > stddev = *:@-+/ % #
J > stddev 1 2 3 4 5
2
```

Thank you :)

Hong Giap Tee
Zhengyang Feng

Questions?