Imports and Modules

By Ana and Kai

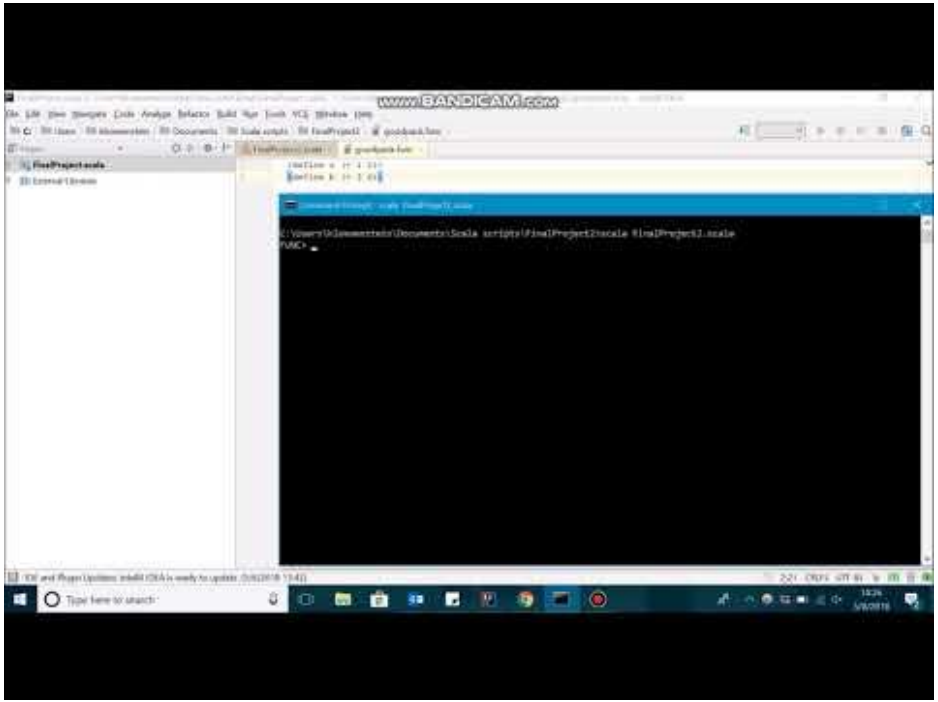
Implemented Features

- Implementing module systems
  - Importing modules
  - Parsing through the modules
  - Referencing modules
  - Keeping the contents of a module separate from the rest of the environment

Our approach to solving

- Started with homework5 code
- Module contents are similar to shell define
- Added functions
  - **EImport /VModule:** EImport reads and parses contents of module, returns a VModule containing the evaluated module contents
  - **EDefine / VDefine:** Creates a tuple containing an ID and a value or expr
  - **EModuleId:** Indexes modules; structured similarly to environment lookup
  - **SEImport:** Pushes a module and its identifier into the environment on import
- Parsing
  - Added stuff to parser
  - Added new parser instance(?) just for imports

Demo



Deep dive or overview

- It works!
- It’s like a mini-environment within the environment

```
def shell_import: Parser[ShellEntry] =
  LP ~ IMPORT ~ ID ~ ID ~ RP ^^ { case _ ~ _ ~ s ~ s2 ~ _ => new SEimport(s, new EImport(s2))}

class SEimport(id:String, es: Exp) extends ShellEntry {

  def processEntry (env:Env) : Env = {
    return env.push(id, es.eval(env))
    }
}
```

```
class EImport (val filename : String) extends Exp {
  override def toString(): String =
    "EImport(" + filename + ")"

  def parseIm (input:String):Exp = {
    val p = new SExpParser
    p.parseAll(p.expr, input) match {
      case p.Success(result,_) => result
      case failure : p.NoSuccess => throw new Exception("Cannot parse "+input+": "+failure.msg)
    }
  }

  def eval(env: Env): Value = {
    var parsed = scala.io.Source.fromFile(filename).getLines().map((line => parseIm(line.mkString)))
    val v = parsed.toList.map((e:Exp) => e.eval(env))
    return new VModule(v, env)

  }
}

class VModule (val ps:List[Value], val env:Env) extends Value {
  println("Module defined")
  override def toString () : String = "VModule(" + ps + ")"
  override def getList () : List[Value] = ps
}
```

```
  def getTup() : (String, Exp) = {
  throw new Exception("Value not of type DEFINE")
  }


class EDefine (val id: String, val e: Exp) extends Exp {
  def eval (env: Env) : Value = {
    println(id + " defined")
    return new VDefine(id,e)
  }
}

class VDefine (val s: String, val es: Exp) extends Value {
  override def toString () : String = "VDefine(" + s + ", " + es + ")"
  override def getTup() : (String, Exp) = {
    return (s, es)
  }
}
```

```
class EModuleId (val m: Exp, id: String) extends Exp {

  override def toString(): String =
    "EModuleId(" + m + id + ")"

  def eval(env: Env): Value = {
    val module = m.eval(env).getList()

    for (entry <-module){
      if (entry.getTup()._1 == id) {
        var v = entry.getTup()._2
        return v.eval(env)
      }
    }

    throw new Exception("Module error: unbound identifier "+id)
  }
}
```

Next Steps

- Importing specific parts of the module
  - Making some things import-able within the package, but not all of it
- Importing within files
- Creating packages
- Parsing through more complex packages