# Programming with Algebra

André van Delft

Anatoliy Kmetyuk

LambdaConf

Boulder, Colorado

26 May 2016

www.subscript-lang.org

# Overview

- Introduction
- SubScript Examples
- Semantic Model
  - Algebra of Communicating Processes
  - VM
- Hands on: Debugger

  https://github.com/scala-subscript/examples
  https://github.com/scala-subscript/koans
- Syntax Matters
- Hands on: Koans, Example
- Conclusion

# Programming is Still Hard

Mainstream programming languages: imperative

- good in batch processing
- not good in parsing, concurrency, event handling
- Callback Hell

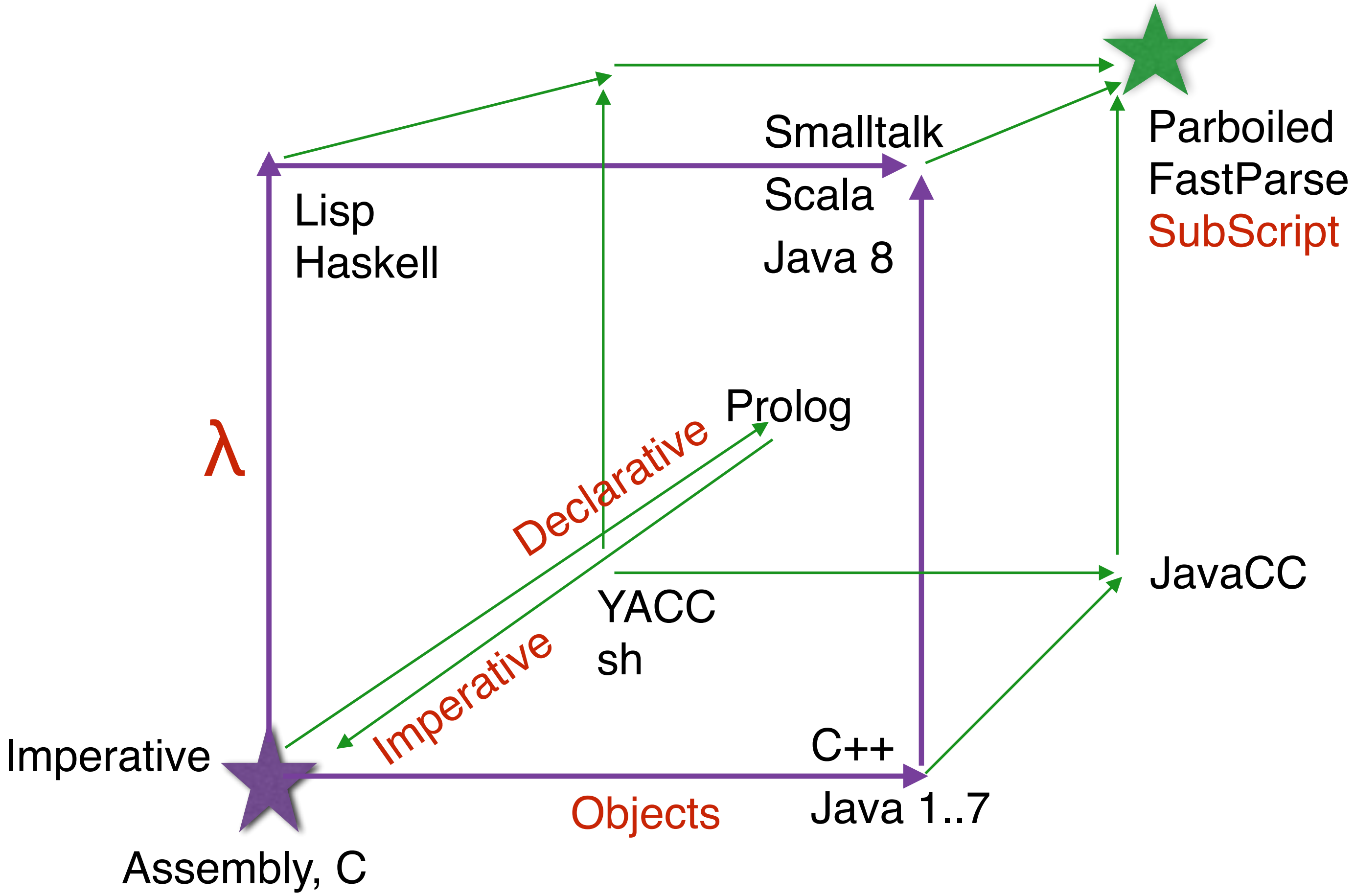
Neglected idioms

- Non-imperative choice: BNF, YACC
- Data flow: Unix pipes

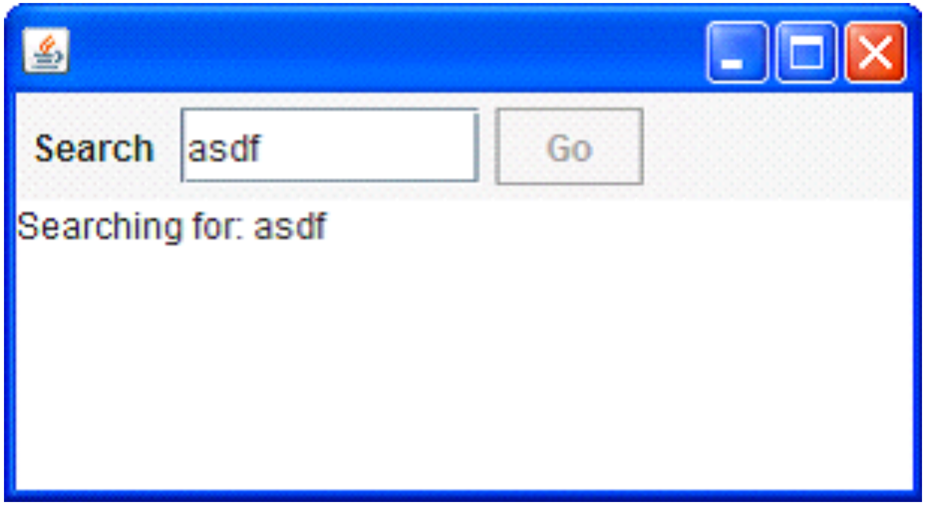
Math!

https://github.com/scala-subscript/examples
https://github.com/scala-subscript/koans

# Programming Paradigms

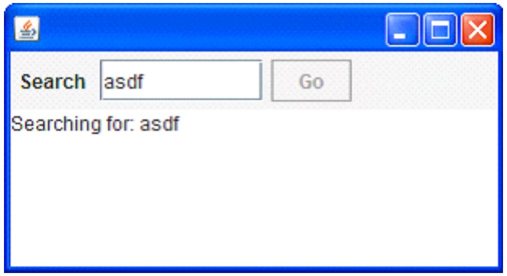


# GUI application - 1



Searching for: asdf

- Input Field
- Search Button
- Searching for…
- Results

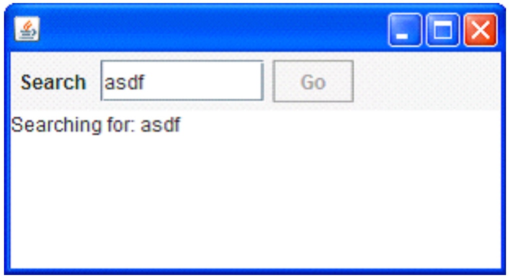


asdf

Searching for: asdf

# GUI application - 2

```
val searchButton = new Button("Go”) {
  reactions.+= {
    case ButtonClicked(b) =>
      enabled = false
      outputTA.text = "Starting search...”
      new Thread(new Runnable {
       def run() {
        Thread.sleep(3000)
        SwingUtilities.invokeLater(new Runnable{
          def run() {outputTA.text="Search ready”
                     enabled = true
        }})
      }}).start
  }
}
```



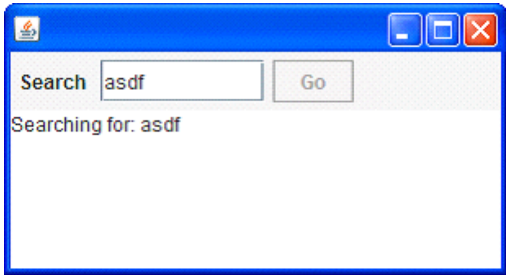
Search: asdf

Searching for: asdf

# GUI application - 3

```
live =       searchButton
        @gui: let outputTA.text="Starting search.."
              do* Thread.sleep(3000)
        @gui: let outputTA.text="Search ready"
              ...
```

- Sequence operator: white space and ;
- gui: code executor for
  - SwingUtilities.InvokeLater+InvokeAndWait
- do* ... : by executor for new Thread

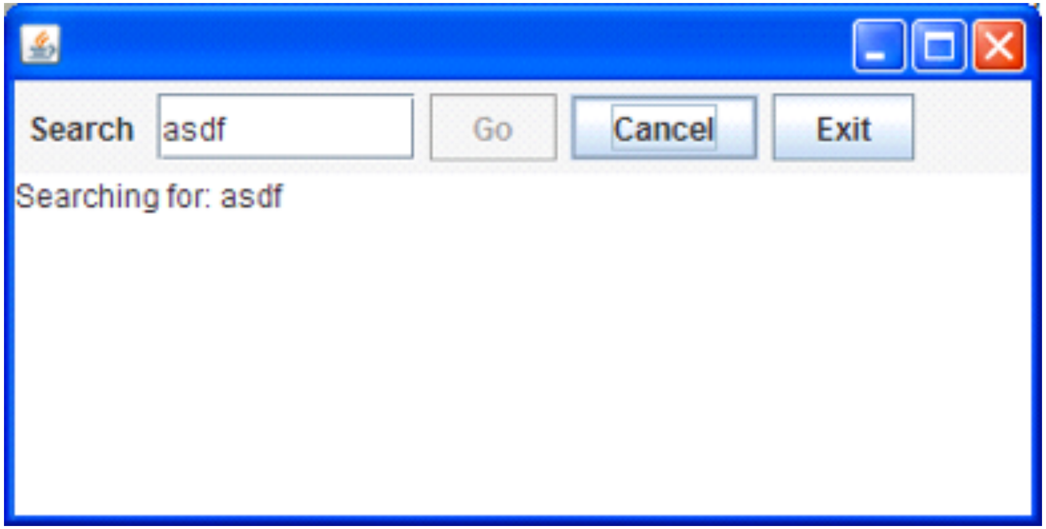


Searching for: asdf

# GUI application - 4

```
live              = searchSequence...

searchSequence    = searchCommand
                    showSearchingText
                    searchInDatabase
                    showSearchResults


searchCommand     = searchButton
showSearchingText = @gui: let outputTA.text = "…"
showSearchResults = @gui: let outputTA.text = "…"
searchInDatabase  =       do* Thread.sleep(3000)
```

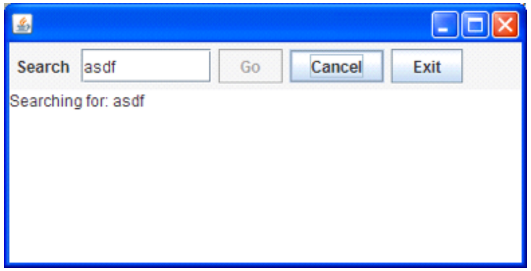
# GUI application - 5



Search asdf

Searching for: asdf

- Search: button or Enter key
- Cancel: button or Escape key
- Exit: button or ☒ ; ; “Are you sure?”…
- Search only allowed when input field **not** empty
- Progress indication


Search asdf

Searching for: asdf

# GUI application - 6

```
live              = searchSequence... || exit

searchCommand     = searchButton + Key.Enter
cancelCommand     = cancelButton + Key.Escape
exitCommand       =   exitButton + windowClosing
exit              =   exitCommand @gui: confirmExit ~~> while !_
cancelSearch      = cancelCommand @gui: showCanceledText

searchSequence    = searchGuard searchCommand
                    showSearchingText searchInDatabase showSearchResults
                    / cancelSearch

searchGuard       = if !searchTF.text.isEmpty then break? anyEvent:searchTF ...

searchInDatabase  = progressMonitor || do* Thread.sleep: 3000
progressMonitor   = do* Thread.sleep: 250
                    @gui: let searchTF.text+=here.pass
                    ...
```

# Example: Slick 3

Reactive Streams for Asynchronous Database Access in Scala

http://www.infoq.com/news/2015/05/slick3

```
val q = for (c<-coffees) yield c.name
val a = q.result
val f: Future[Seq[String]] = db.run(a)

f.onSuccess { case s => println(s"Result: $s") }
```

```
val q = for (c<-coffees) yield c.name

q ~~(s)~~> println: s"Result: $s"
```

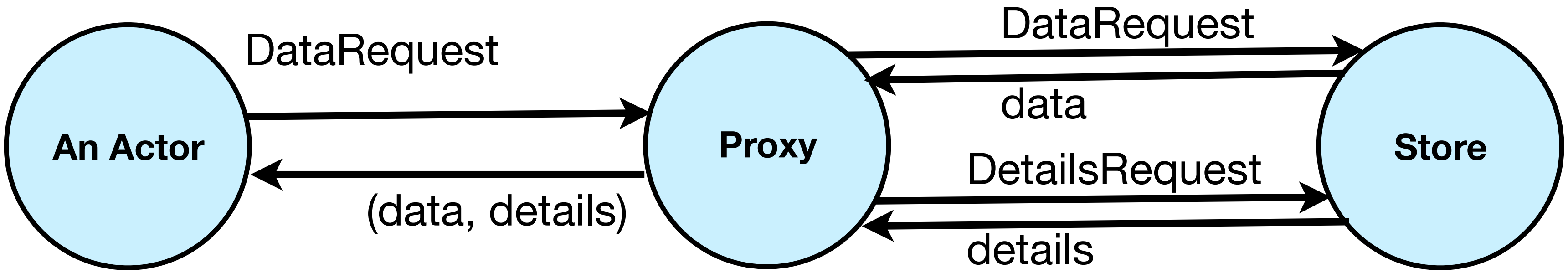
# SubScript Actors: Ping Pong

```
class Ping(pong: ActorRef) extends Actor {
   override def receive: PartialFunction[Any,Unit] = {case _ =>}
     pong ! "Hello"
     pong ! "Hello"
     pong ! "Terminate"
}
```

```
class Pong1 extends SubScriptActor {
  override def receive: PartialFunction[Any,Unit] = {
    case "Hello"     => println("Hello")
    case "Terminate" => println("Done" ); context.stop(self)
  }
}
```

```
class Pong2 extends SubScriptActor {   var ping: ActorRef
  script ..
    live = ping ~~("Hello"    )~~> println: "Hello"  ...
         / ping ~~("Terminate")~~> println: "Done"
}
```

# SubScript Actors: DataStore - 1

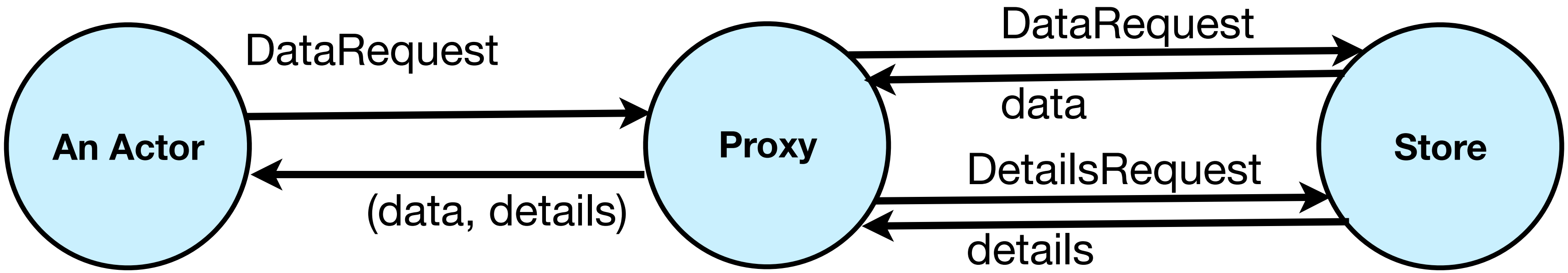


```
class DataStore extends Actor {

  def receive = {
    case     DataRequest(name) => sender ! getData    (name)
    case DetailsRequest(data) => sender ! getDetails(data)
  }

}
```

# SubScript Actors: DataStore - 2

```
class DataProxy(dataStore: ActorRef) extends Actor {

  def waitingForRequest = {
    case req: DataRequest =>
      dataStore ! req
      context become waitingForData(sender)
  }

  def waitingForData(requester: ActorRef) = {
    case data: Data =>
      dataStore ! DetailsRequest(data)
      context become waitingForDetails(requester, data)
  }

  def waitingForDetails(requester: ActorRef, data: Data) = {
    case details: Details =>
      requester ! (data, details)
      context become waitingForRequest
  }
}
```

# SubScript Actors: DataStore - 3



```
class DataProxy(dataStore: ActorRef) extends SubScriptActor {

 script live =

  ?anActor:ActorRef ~~(    req: DataRequest)~~> {dataStore ? req}
                    ~~(   data: Data       )~~> {dataStore ? DetailsRequest:data}
                    ~~(details: Details    )~~> do anActor ! (data, details)
   ...
}
```

# Algebra of Communicating Processes - 1

Bergstra & Klop, Amsterdam, 1982 - ...

ACP ~ Boolean Algebra

| | |
|---|---|
| + | choice |
| · | sequence |
| 0 | deadlock |
| 1 | empty process |

atomic actions a,b,…
parallelism
communication
disruption, interruption
time, space, probabilities
money
...

# Algebra of Communicating Processes - 2

Less known than CSP, CCS

Specification & Verification

- Communication Protocols
- Production Plants
- Railways
- Coins and Coffee Machines
- Money and Economy

Strengths

- Familiar syntax
- Precise semantics
- Reasoning by term rewriting
- Events as actions

# Algebra of Communicating Processes - 3

$$
\begin{aligned}
x+y &= y+x \\
(x+y)+z &= x+(y+z) \\
x+x &= x \\
&\boxed{(x+y)\cdot z = x\cdot z+y\cdot z} \\
(x\cdot y)\cdot z &= x\cdot(y\cdot z)
\end{aligned}
$$

$$
\begin{aligned}
0+x &= x \\
0\cdot x &= 0 \\
&\boxed{1\cdot x = x} \\
x\cdot 1 &= x
\end{aligned}
$$

$$
\begin{aligned}
(x+1)\cdot y &= x\cdot y + 1\cdot y \\
&= x\cdot y + y
\end{aligned}
$$

# Algebra of Communicating Processes - 4

$$x \parallel y = x \mathbin{\lfloor\!\!\lfloor} y + y \mathbin{\lfloor\!\!\lfloor} x + x \mid y$$

$$\begin{aligned}
(x+y) \mathbin{\lfloor\!\!\lfloor} z &= \ldots \\
a \cdot x \mathbin{\lfloor\!\!\lfloor} y &= \ldots \\
1 \mathbin{\lfloor\!\!\lfloor} x &= \ldots \\
0 \mathbin{\lfloor\!\!\lfloor} x &= \ldots
\end{aligned}$$

$$\begin{aligned}
(x+y) \mid z &= \ldots \\
\ldots &= \ldots
\end{aligned}$$

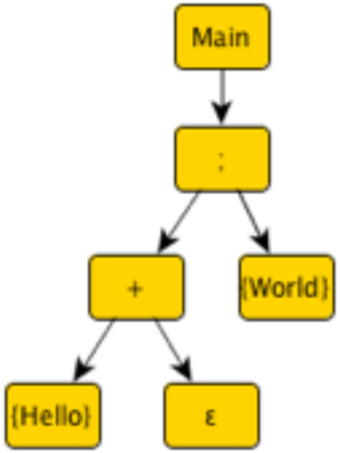
# Implementation - 1

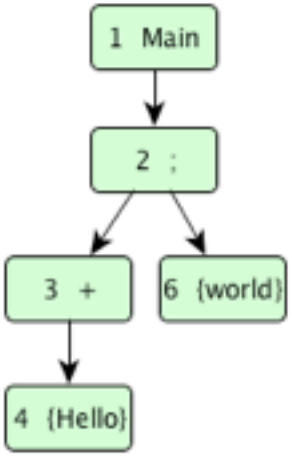
Main = (Hello + 1) · World

```
import subscript.DSL._

def Main = _script('Main) {
             _seq(_alt(_normal{here=>Hello}, _empty),
                       _normal{here=>World}          )
           }


def main(args: Array[String]): Unit = _execute(Main)
```

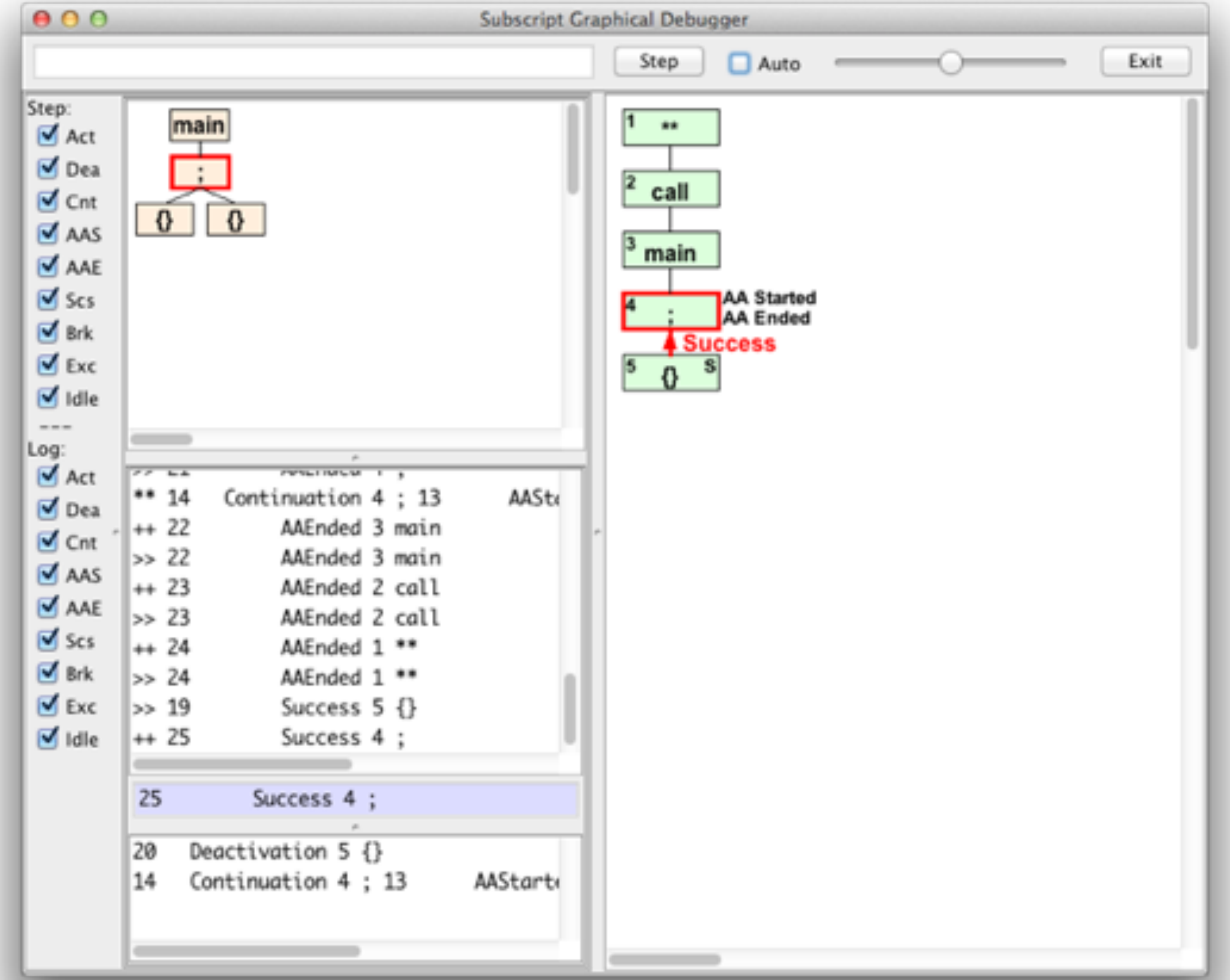
Virtual Machine: 2500 code lines

- static script trees
- dynamic Call Graph
- here there
- onActivate onSuccess





# Debugger - 1



□ Auto

Step:
☑ Act
☑ Dea
☑ Cnt
☑ AAS
☑ AAE
☑ Scs
☑ Brk
☑ Exc
☑ Idle

Log:
☑ Act
☑ Dea
☑ Cnt
☑ AAS
☑ AAE
☑ Scs
☑ Brk
☑ Exc
☑ Idle

```
** 14   Continuation 4 ; 13        AASt
++ 22        AAEnded 3 main
>> 22        AAEnded 3 main
++ 23        AAEnded 2 call
>> 23        AAEnded 2 call
++ 24        AAEnded 1 **
>> 24        AAEnded 1 **
>> 19        Success 5 {}
++ 25        Success 4 ;
```

```
25          Success 4 ;
```

```
20   Deactivation 5 {}
14   Continuation 4 ; 13        AAStart
```

# Hands On - 1

https://github.com/scala-subscript/examples

```
git clone https://github.com/scala-subscript/examples.git
cd examples
```

```
sbt
> project helloworld
> set mainClass in Compile := Some("subscript.example.Hello")
> ssDebug
```

Edit file:
examples/helloworld-example/src/main/scala/subscript/example/Hello.scala

```
Hello
Hello; World
Hello+[+];  World
```

# Syntax Matters - 1

```
ACP:  Main = (Hello + 1) · World
```

```
import subscript.DSL._

def Main = _script('Main) {
             _seq(_alt(_normal{here=>Hello}, _empty),
                       _normal{here=>World}          )
           }
```

Improve with specific syntax; mainly **simple** Sugar

Goals:

- DRY, less Boilerplate code
- Few (Parentheses), {Braces}, [Brackets]
- Few vars
- Refinement support
- Base layer with symbols, not keywords
- Top layer with well readable words
- Clear boundaries Scala <==> SubScript

# Syntax Matters - 2

ACP: Main = (Hello + 1) · World

```
import subscript.DSL._

def Main = _script('Main) {
             _seq(_alt(_normal{here=>Hello}, _empty),
                       _normal{here=>World}          )
           }
```

| Year | Solution |
|---|---|
| 2011 | subscript.DSL |
| 2012 | Scalac branch: scanner, parser, typer |
| 2015 | Parboiled2 preprocessor + macros |
| 2016. | FastParse + Dotty |

# Syntax Matters - 3

ACP: Main = (Hello + 1) · World

```
import subscript.DSL._

def Main = _script('Main) {
             _seq(_alt(_normal{here=>Hello}, _empty),
                       _normal{here=>World}          )
           }
```

Influences

- Scala
- ACP
- YACC
- Prolog, Linda
- Basic
- Smalltalk
- Unix sh
- FastParse

# Syntax Matters - 4

| Construct | ACP | SubScript |
|---|---|---|
| Deadlock process | 0 | [-] |
| Empty process | 1 | [+] |
| Neutral process | 0 or 1 | [] |
| Neutral code | | {: scala :} |
| Atomic actions | a, b, … | {! scala !}   {* *}   {. .} |
| Choice | x+y | x+y |
| Sequence | x·y | x y   x;y |
| Expression parentheses | (x+y)·z | [x+y] z   x+y;z |
| Parallelism | x‖y | x&y   x\|y   x&&y   x\|\|y |
| Sequential Iteration | x*y | ..? x; y |
| Iterators | ∑ ∏ ‖ | ..?   ...   while   for |
| Break from expression | | break?   break |
| Process launching | cr(x) | [*x*] |
| Communication | a,b = c | shared scripts: multiple callers |

# Syntax Matters - 5

| Construct | SubScript |
|---|---|
| N-ary Operators | whitespace ; + & && \| \|\| / |
| Grouping | […] |
| Special terms | [+] [-] [] ..? ... while for break? break |
| Code fragments | {@ scala @} for @ in :, !, ?, *, ., ... |
| Annotations, call graph node | @there.onDeactivate{...}: here.pass |
| Declarations | val, var |
| Output parameters | s( ?i:Int) s(?i) ?i ?j:Int |
| Constrained parameters | t(??i:Int) t(?i) t(?i ?if(_>3)) t(5) |
| Control | if-then-else do-then-else |
| Dataflow map | ~~^ ~/~^ ~~^ +~/~^ |
| Dataflow flatmap | ~~> ~/~> ~~> +~/~> |
| Result values | Script[T] x^ x^^ x^^1 ^x |
| **Scala terms** | true 1 'a' "A" p p.q p.q(r) (..) {…} |

# Syntax Matters - 6

```
resolve(termType) =

  termType match {
    case t: Unit       => neutralCodeFragment
    case t: Script[_]  => scriptCall
    case other         => findImplicitConversionsFor(other) match {
                            case List(c) if c isInstanceOf[Unit]
                                         || c.isInstanceOf[Script[_]]
                                   => resolve(c.type)
                            case _ => error
                         }
  }
```

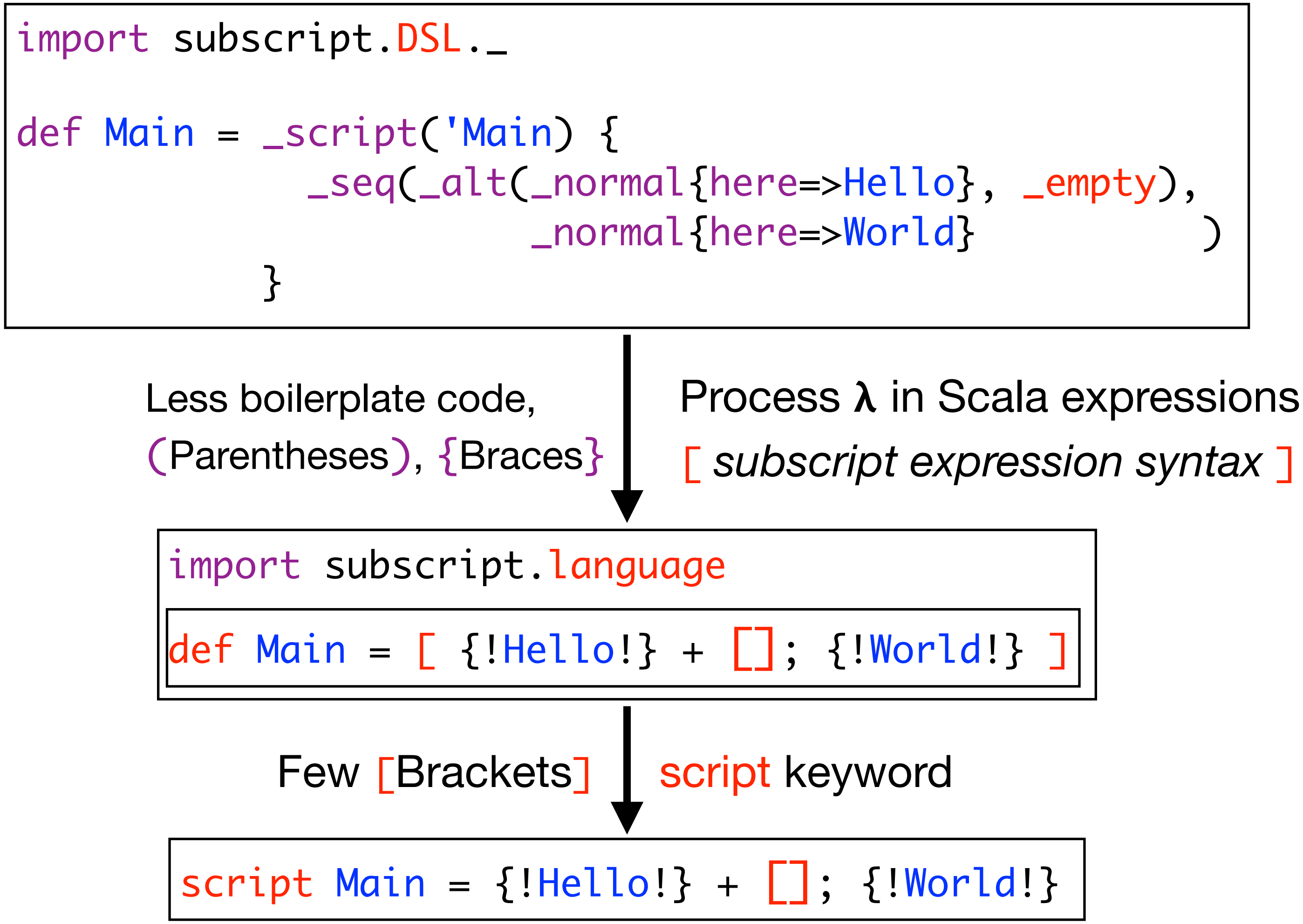
# Syntax Matters - 7

```
resolve(termType) =

  ^termType
      ~~(t: Unit     )~~^ neutralCodeFragment
     +~~(t: Script[_])~~^ scriptCall
     +~~(other       )~~^ ^findImplicitConversionsFor: other
                              ~~(List(c) if c isInstanceOf[Unit]
                                         || c.isInstanceOf[Script[_]]
                                )~~^ resolve: c.type
                             +~~^    error
```

([x]) = ???

# Syntax Matters - 8

ACP: Main = (Hello + 1) · World



```
import subscript.DSL._

def Main = _script('Main) {
             _seq(_alt(_normal{here=>Hello}, _empty),
                       _normal{here=>World}          )
           }
```

Less boilerplate code,
(Parentheses), {Braces}

Process λ in Scala expressions
[ *subscript expression syntax* ]

```
import subscript.language

def Main = [ {!Hello!} + []; {!World!} ]
```

Few [Brackets]

script keyword

```
script Main = {!Hello!} + []; {!World!}
```

# Syntax Matters - 9

```
script searchCommand      = searchButton
script showSearchingText = @gui: {: outputTA.text = "…" :}
script showSearchResults = @gui: {: outputTA.text = "…" :}
script searchInDatabase  =       {* Thread.sleep(3000)  *}
```

Top layer with well readable words

Use `let` and `do`

```
script searchCommand      = searchButton
script showSearchingText = @gui: let outputTA.text = "…"
script showSearchResults = @gui: let outputTA.text = "…"
script searchInDatabase  =       do* Thread.sleep(3000)
```

DRY

`script ..` section

```
script ..                  // .. also for Scala (trait, class, def, val, var, ...)?

  searchCommand      = searchButton
  showSearchingText = @gui: let outputTA.text = "…"
  showSearchResults = @gui: let outputTA.text = "…"
  searchInDatabase  =       do* Thread.sleep(3000)
```

# Syntax Matters  - 10

| Construct | Base form | Less {Braces} |
|---|---|---|
| Neutral code | `{:   scalaCode   :}` | `let   scalaCode` |
| Atomic action | `{!   scalaCode   !}` | `do!   scalaCode` |
| Threaded code | `{*   scalaCode   *}` | `do*   scalaCode` |
| Event handling code | `{.   scalaCode   .}` | `do.   scalaCode` |
| Persistent event handler | `{... scalaCode ...}` | `do... scalaCode` |

# Hands On - 2

https://github.com/scala-subscript/koans

Download; unzip to koans/

```
cd koans

sbt
> koans
```

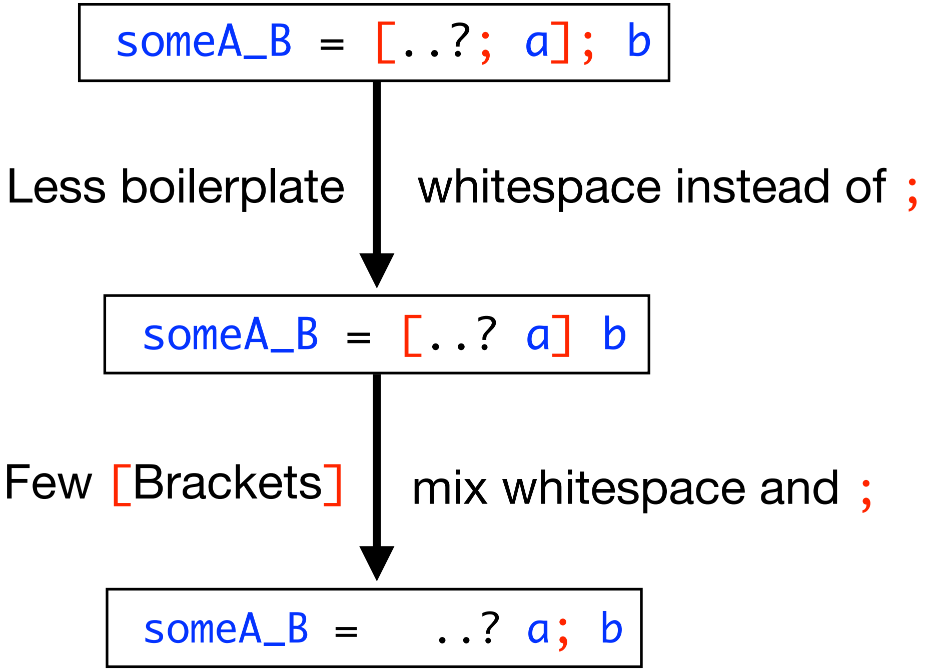
Edit; retry; ...

# github.com/scala-subscript/koans

```
package subscript.koans

import subscript.language
import subscript.Predef._

import subscript.koans.util.KoanSuite

class AboutSubScript extends KoanSuite {
  koan(1)(
    """
    | Imports, scripts and atomic actions:
    |
    | To use SubScript in a file, it should have these two import statements:
    |
    | `import subscript.language`
    | ...
    """
  ) {
    var    flag = false
    script foo  = {! flag = true !}

    test(1) { runScript(foo); flag shouldBe __ }
  }
```

# Syntax Matters  - 11

```
ACP: a*b
```



```
someA_B = [..?; a]; b
```

Less boilerplate

whitespace instead of ;

```
someA_B = [..? a] b
```

Few [Brackets]

mix whitespace and ;

```
someA_B =    ..? a; b
```

# Syntax Matters - 12

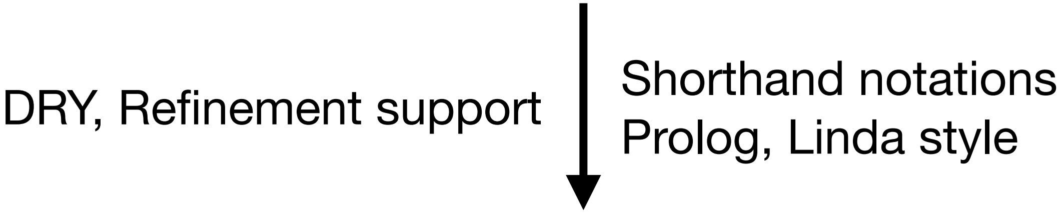
```
searchCommand = clicked(searchButton) + pressed(Key.Escape)
cancelCommand = clicked(cancelButton) + pressed(Key.Escape)
  exitCommand = clicked(  exitButton) +  windowClosing
```

DRY ↓ Implicit Conversions

```
searchCommand = searchButton + Key.Escape
cancelCommand = cancelButton + Key.Escape
  exitCommand =   exitButton + windowClosing
```

# Syntax Matters - 13

```
key(c: FormalConstrainedParameter[Int]) =
                        key(top, AdaptingParameter(c, {c = _}))

var c = 0
key(ActualOutputParameter(    c, {c = _}))
key(ActualConstraintParameter(c, {c = _}, {_<64}))
key(ActualValueParameter('x'))
```

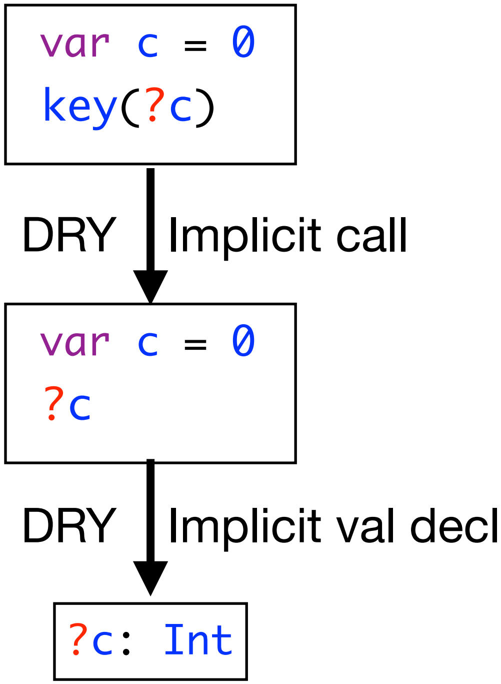


DRY, Refinement support

Shorthand notations
Prolog, Linda style

```
key(??c: Int) = key(top, ??c)

var c = 0
key(?c)
key(?c ?if(_<64))
key('x')
```

# Syntax Matters  - 14



```
var c = 0
key(?c)
```

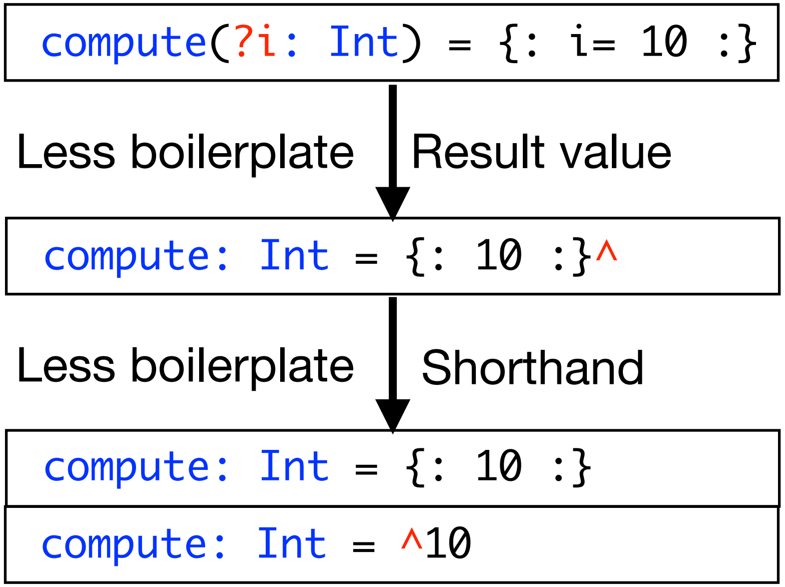
DRY Implicit call

```
var c = 0
?c
```

DRY Implicit val decl

```
?c: Int
```

# Syntax Matters - 15



```
compute(?i: Int) = {: i= 10 :}
```

```
compute: Int = {: 10 :}^
```

```
compute: Int = {: 10 :}
```

```
compute: Int = ^10
```

```
compute^ println:"Ok"
```

```
naturalsUpTo(n: Int) = times:n ^pass^^
```

# Syntax Matters - 16

```
naturalsUpTo(n: Int) = times:n ^pass^^
```

```
([x]) = x
```

in a λ

```
[x]^ =def= ([x])^
```

```
naturalsWithSquaresUpTo(n: Int) = times:n [ ^ pass      ^^1
                                            ^(pass*pass)^^2 ]^^
```

# Syntax Matters  - 17

```
x ~~(b:Boolean    )~~> y1
 +~~(i:Int if i<10)~~> y2
 +~~( _           )~~> y3
+~/~(e:IOException)~~> z1
+~/~(e:  Exception)~~> z2
+~/~(e:  Throwable)~~> z3
```

```
x ~~> case b:Boolean     => [y1]
      case i:Int if i<10 => [y2]
      case _             => [y3]

+~/~> case e:IOException => [z1]
      case e:  Exception => [z2]
      case e:  Throwable => [z3]
```

```
def x ~~> y +~/~> z  =
{
  var    x_node: N_call[Any] = null
  [ do @{x_node = there.asInstanceOf[N_call[Any]]}:
         x
    then y:x_node.$success ^
    else z:x_node.$failure ^
  ]
}
```

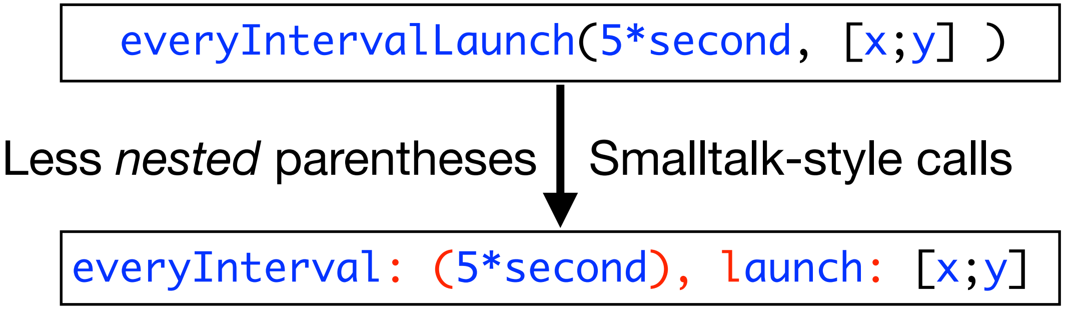
match+catch
flatmap

map:

`x ~~^ toString`

# Syntax Matters - 17

```
everyIntervalLaunch(d: Duration, p: Script[_]) = wait:d [*p*] ...
```



```
everyIntervalLaunch(5*second, [x;y] )
```

Less *nested* parentheses

Smalltalk-style calls

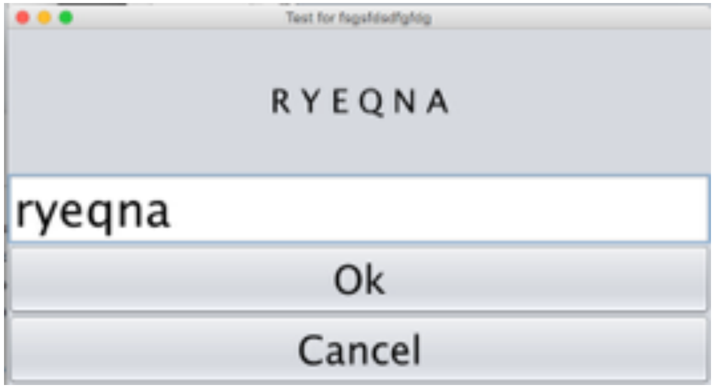
```
everyInterval: (5*second), launch: [x;y]
```

# github.com/scala-subscript/examples

- helloworld
- lookupframe
- life
- filedownloader
- pingpong
- storage
- subscript-twitter-search
- taskprocessor

# github.com/scala-subscript/eye-test

```
script..
   live = mainTestProcess^ / cancelBtn

   mainTestProcess = eyeTest("Right")^^1
                     eyeTest("Left" )^^2

   eyeTest(eyeName: String)
   = let testArea.font          = new Font("Ariel", java.awt.Font.PLAIN, 20)
     let testArea.text          = s"<html>Look with your $eyeName eye.</html>"
     let answerField.enabled = false
     sleep: 250
     Key.Enter + okBtn
     doTest( if(eyeName=="Right") previousScoreRight else previousScoreLeft )^
```



Test for fsgsfdsdfgfdg

R Y E Q N A

ryeqna

# Hands On - 3

https://github.com/scala-subscript/examples

```
cd examples
sbt
> projects
> project lookupframe
> run

Multiple main classes detected, select one to run:

 [1] subscript.example.LookupFrame
 [2] subscript.example.LookupFrame2
 [3] subscript.example.LookupFrame2TBD

Enter number: 3
```

Edit file according to guidelines:

```
lookup-example/src/main/scala/subscript/example/LookupFrame2TBD.scala
        storage/src/main/scala/subscript/example/StorageTBD.scala
subscript-twitter-search/src/main/scala/subscript/twitter/app/controller/
                                   SubScriptControllerTBD_Futures.scala
```

# Open Source Project

- subscript-lang.org
  github.com/scala-subscript
- 10^4...10^5 actions per second
- Simple implementation: 6000 lines, 50%
  - Scalac branch ~~> Parboiled + Macro's
  - VM
  - scripts for actors, swing

- Jetbrains - IntelliJ Plugin
- ScalaParse + Dotty

# FastParse & ScalaParse

- http://www.lihaoyi.com/fastparse/
- Better error messages than Parboiled2

- Inspiration for SubScript:
    - ^ - normal result value
    - ^^ - result values into List
    - ^^1, ^^2 - result values into tuple

```
script..
   s = var i= 0
       var j=10
       while(i<3) [^i^^1 ^j^^2]^^  {! i+=1; j-=1 !}

 test(1) {runScript(s).$ shouldBe Success(List((0,10),(1,9),(2,8)))}
```

# Conclusion

- Programming great again with Algebra
- Still much to do:
  - ScalaParse & Dotty
  - JS, NodeJS
  - ACP style communication
  - ...
- and to discover:
  - programming patterns
  - arXiv paper "Some New Directions in ACP Research"
- To join the project: andre.vandelft@gmail.com
- Sponsors also welcome