SubScript:

Extending Scala with the Algebra of Communicating Processes

André van Delft

20 June 2013

Presentation at Amsterdam.Scala

Overview

- Programming is Still Hard
- Algebra of Communicating Processes
- SubScript Now
  - Examples: GUI controllers
  - Implementation
  - Demonstration
- SubScript when Ready
  - Features
  - Challenges
  - Dataflow Programming, ...
- Conclusion

Programming is Still Hard

Mainstream programming languages: imperative

- good in batch processing
- not good in parsing, concurrency, event handling
- Java threads & event handlers are data
  - boring boilerplate code
  - error-prone: non-responsive GUIs
    - GUI thread
    - background threads
    - event handlers
    - enabling/disabling widgets
- Callback Hell

Neglected idioms

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

Algebra of Communicating Processes - 1

Bergstra & Klop, Amsterdam, 1982 - ...

ACP ~ Boolean Algebra

$+$	choice
$\cdot$	sequence
0	deadlock
1	empty process

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

Algebra of Communicating Processes - 2

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

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

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

Algebra of Communicating Processes - 3

$$
\begin{aligned}
x \parallel y &= x \mathbin{⫡} y + y \mathbin{⫡} x + x \mid y \\
\\
(x+y) \mathbin{⫡} z &= x \mathbin{⫡} z + y \mathbin{⫡} z \\
a \cdot x \mathbin{⫡} y &= a \cdot (x \parallel y) \\
1 \mathbin{⫡} x &= 0 \\
0 \mathbin{⫡} x &= 0 \\
\\
x \mid y &= y \mid x \\
(x+y) \mid z &= x \mid z + y \mid z \\
a \cdot x \mid b \cdot y &= (a \wedge b) \cdot (x \parallel y) \\
1 \mid a \cdot x &= 0 \\
1 \mid 1 &= 1 \\
0 \mid x &= 0
\end{aligned}
$$

Algebra of Communicating Processes - 4

```
(x+y)/ z  =  is0(x+y)·z
            + not0(x)·x/z
            + not0(y)·y/z
 a·x / y  =  a·(x/y) + y
   0 / x  =  x
   1 / x  =  1

 is0(x+y) = is0(x)·is0(y)
 is0(a·x) = 0
 is0(0)   = 1
 is0(1)   = 0
not0(x)   = is0(is0(x))
```

Algebra of Communicating Processes - 5

Less known than CSP, CCS

Specification & Verification

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

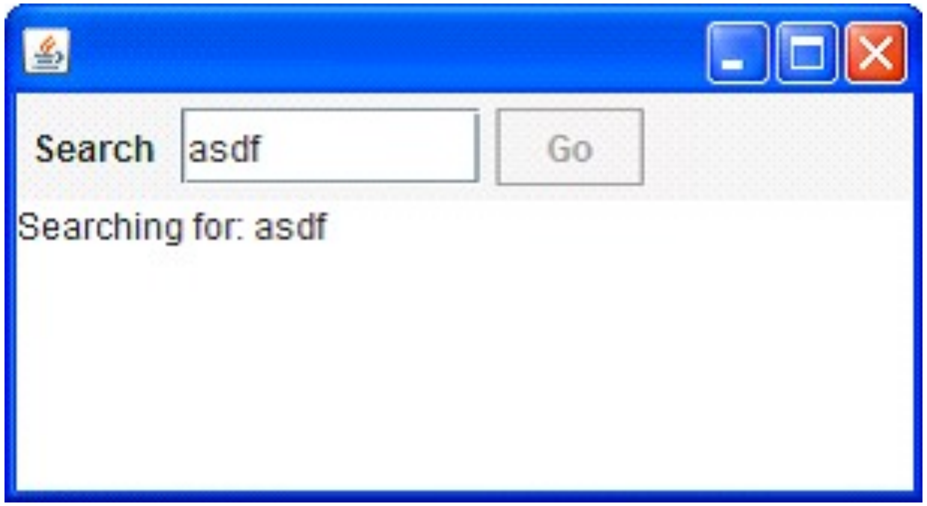
Strengths

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

ACP Language Extensions

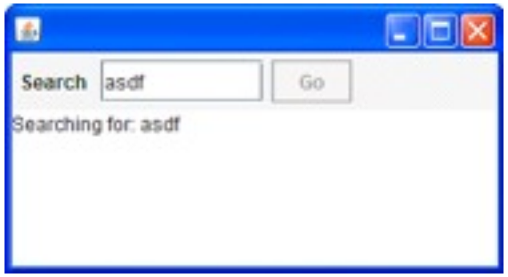
- 1980: Jan van den Bos - Input Tool Model
- 1988-2011: AvD - Scriptic
  - Pascal, Modula-2, C, C++, Java
- 2011-...: AvD - SubScript
  - Scala
  - JavaScript, ... (?)
- Application Areas
  - GUI Controllers
  - Text Parsers
  - Discrete Event Simulation
  - Dataflow Programming (?)
  - Parallel Processing (?)

GUI application - 1



Searching for: asdf

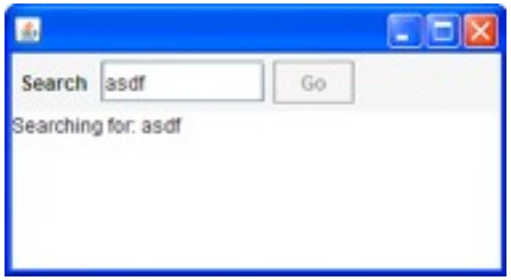
- Input Field
- Search Button
- Searching for…
- Results



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
  }
}
```

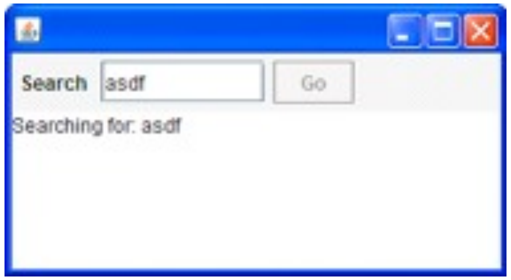


Searching for: asdf

GUI application - 3

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

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

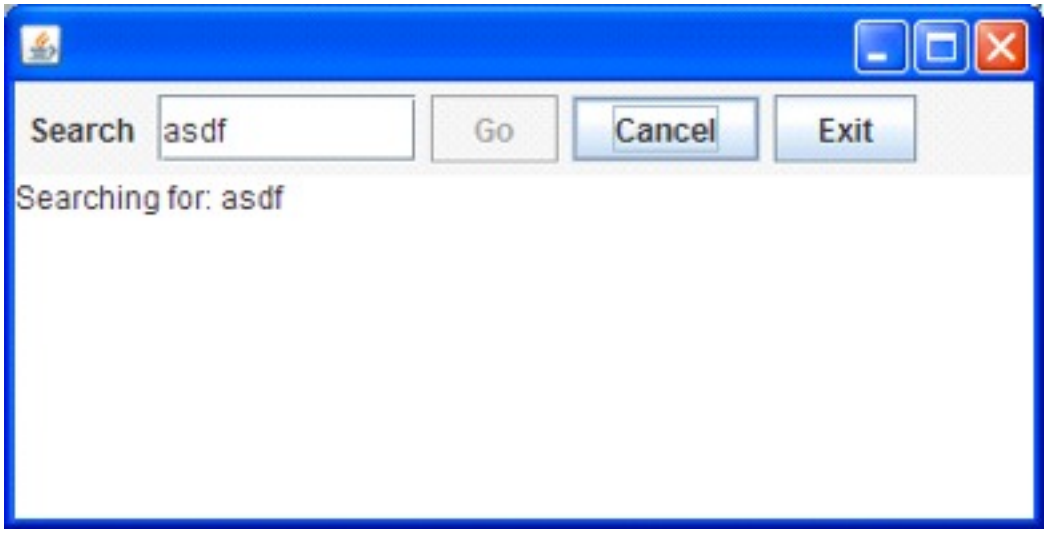


Searching for: asdf

GUI application - 4

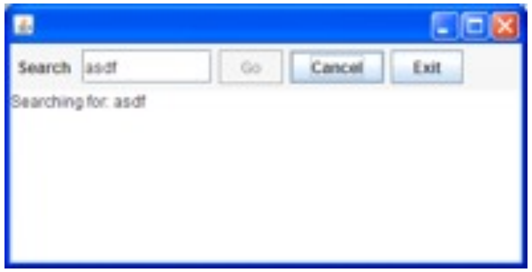
```
live              = searchSequence...

searchSequence    = searchCommand
                    showSearchingText
                    searchInDatabase
                    showSearchResults


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

GUI application - 5



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



Searching for: asdf

GUI application - 6

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

searchCommand      = searchButton + Key.Enter
cancelCommand      = cancelButton + Key.Escape
exitCommand        =   exitButton + windowClosing
exit               =   exitCommand @gui: while(!areYouSure)
cancelSearch       = cancelCommand @gui: showCanceledText

searchSequence     = searchGuard searchCommand;
                     showSearchingText
                     searchInDatabase
                     showSearchResults / cancelSearch

searchGuard        = if(!searchTF.text.isEmpty) . anyEvent(searchTF) ...


searchInDatabase   = {*Thread.sleep(3000)*} || progressMonitor
progressMonitor    = {*Thread.sleep( 250)*}
                     @gui:{searchTF.text+=here.pass} ...
```

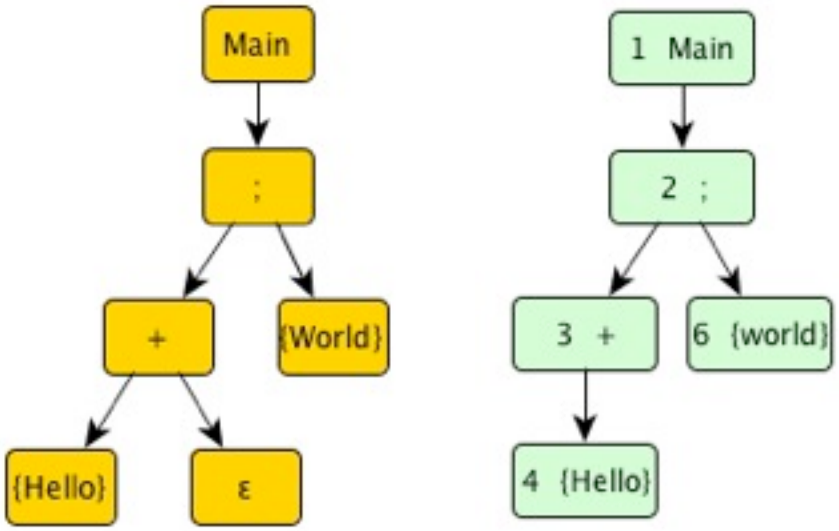
Implementation

- 50% done, communication, data flow due
- Branch of Scalac

```
script a = b;{c}   ⟹    def _a = _script('a) {
                          _seq(_call{here=>_b}, _normal{here=>c})
                        }
```

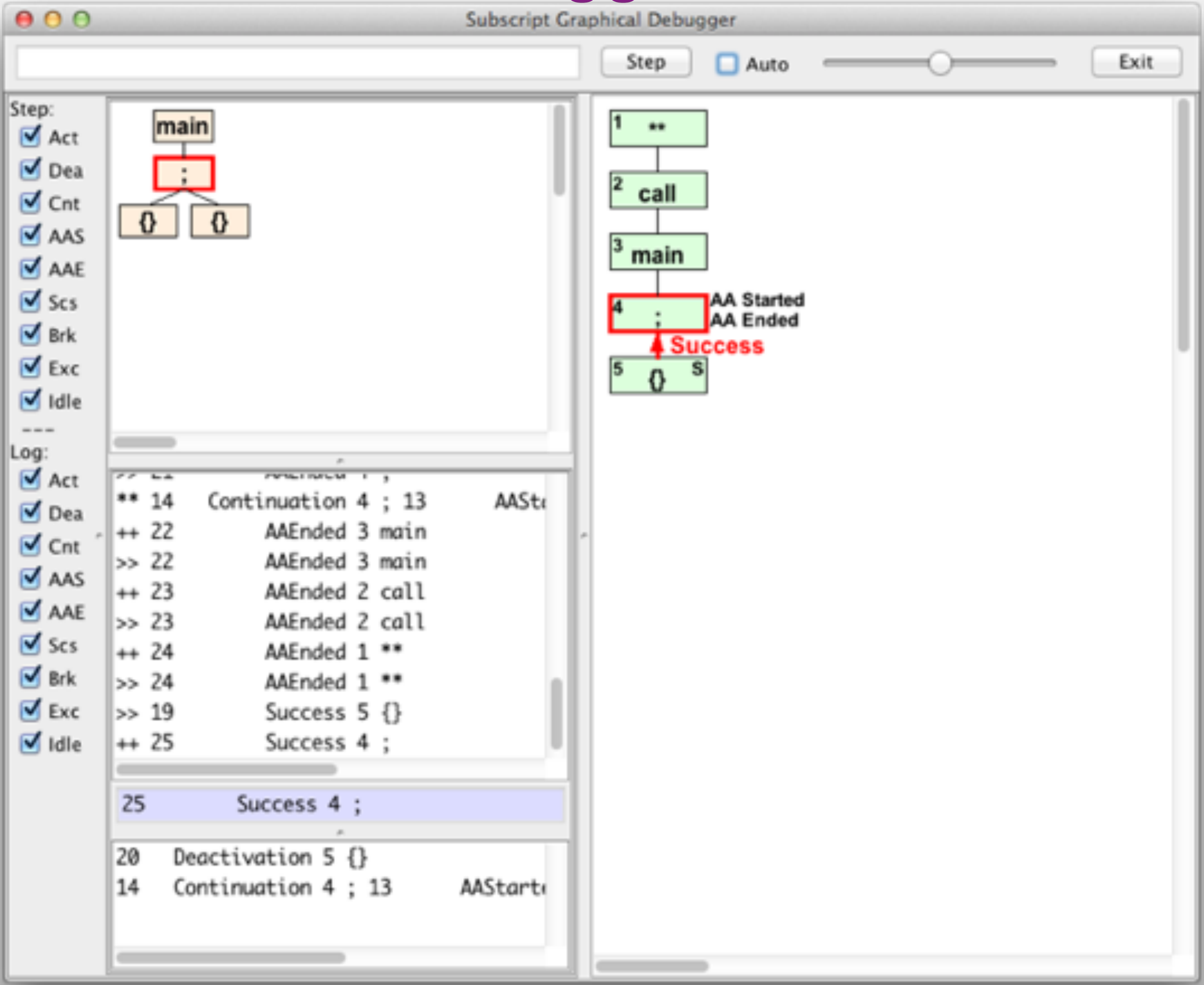
  - lines: scanner 100, parser 1000, typer 200

- Virtual Machine
  - lines: 2000
  - static script trees
  - dynamic Call Graph



- Swing event handling scripts
  - lines: 260
- Graphical Debugger
  - lines: 550   (10 in SubScript)

Debugger - 1



```
** 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
```

Debugger - 2

built using SubScript

```
live = {* awaitMessageBeingHandled *}
        if (shouldStep)
        (  @gui: {!updateDisplay!} stepCommand
        || if (autoCheckBox.isChecked) waitForStep
        )
        { messageBeingHandled=false }
        ...
     || exit

exit = exitCommand
       var exitConfirmed = false
       @gui: { exitConfirmed=confirmExit }
       while (!exitConfirmed)
```

SubScript Features - 1

"Scripts" – process refinements as class members

- Called like methods from Scala
  - with a ScriptExecutor as extra parameter
- Call other scripts
- Parameters: in, out?, constrained, forcing

Formal	`implicit key(c??: Char) = ...`		
Actual Calls	Output	Constrained	Forcing
Conventional	`key(c?)`	`key(c? if? c.isDigt)`	`key('1')`
No parentheses	`key,c?`	`key,c? if? c.isDigt`	`key,'1'`
Using `implicit`	`c?`	`c? if? c.isDigt`	`'1'`

SubScript Features - 2

ACP Atomic Actions ~ Scala Code {…} start/end

`{ … }`	Normal
`{? … ?}`	Unsure
`{! … !}`	Immediate
`{* … *}`	New thread
`@gui: { … }`	GUI thread
`@dbThread: { … }`	DB thread
`@reactor: {. … .}`	Event handler
`@reactor: {... … ...}`	Event handler, permanent
`@startTime: { … }`	Simulation time + real time
`@processor=2: {* … *}`	Processor assignment
`@priority=2: {* … *}`	Priority
`@chance=0.5: { … }`	Probability

SubScript Features - 3

N-ary operator	Meaning
`;` `ws`	Sequence
`+`	Choice
`&`	Normal parallel
`|`	Or-parallel
`&&`	And-parallel
`||`	Or-parallel
`==>`	Network/pipe
`/`	Disrupt
`%/`	Interrupt

Unary operator	Meaning
`x*`	Process launch

Construct	Meaning
`here`	Current position
`@ … :`	Annotation
`if-else`	
`match`	
`try-catch-`	
`for`	
`while`	
`break`	
`...`	`while(true)`
`..`	Both `...` and `.`
`.`	Optional break
`(-), (+),`	Neutral: 0, 1-like

SubScript Features - 4

Process Communication

Definitions: Shared Scripts
`send(i:Int), receive(j??:Int) = {j=i}`
`send(i:Int), receive(i??: _ ) = {}`
`ch<-(i:Int),     ch->(i??:Int) = {}`
`ch<-->(i??:Int)                = {}`
`  <-->(i??:Int)                = {}`
`  <==>(i??:Int)                = {}`

Usage: Multicalls	
`send(10) & receive(i?)`	Output param
`send(10) & receive(10)`	Forcing
`ch<-(10) &     ch->(10)`	Channel
`<-10 & ->i?`	Nameless
`*<-10 ; ->i?`	Asynchronous send

Feedback (EPFL, Scala Workshop)

- “Get rid of the vars”
- “The GUI Client is dead”
- Potential for Akka programming
- Terseness ≠ Simplicity

Data Flow Support

- Script Lambda's
- Split Scripts
- Script Result Values
- One-time flow
- Lasting flow
- Partial receive scripts - Akka

Script Lambdas

- Henk Goeman 1989:

  (Self) Applicative Communicating Processes

- Robin Milner 1989:

  π-calculus

`< a; b >`   λ - anonymous script

Split Scripts - 1

```
header:            do~ s:script ~while~ b: =>Boolean ~end

same as:           do~~while~~end(s:script, b: =>Boolean)


define:  do~ s:script ~while~ b: =>Boolean ~end = s while(b)


usage:   test = do~<  a;b   >~while~ !found ~end
```

Split Scripts - 2

```
progressMonitor = sleep_ms(250) updateStatus ...
               || sleep_ms(5000)


progressMonitor = during_ms~ 5000
                  ~every_ms~ 250
                  ~do~< updateStatus >~end


during_ms~ duration:Int
~every_ms~ interval:Int
~do~ task:script ~end = sleep_ms(interval) task...
                     || sleep_ms(duration)
```

Script Result Values - 1

```
expr   = term   .. "+"
term   = factor .. "*"
factor = number + "(" expr ")"
```

```
expr   : expr PLUS term  { $$ = $1 + $3; }
                 |  term  { $$ = $1; } ;
term   : term MUL factor { $$ = $1 * $3; }
                 | factor { $$ = $1; } ;
factor : LPAR expr RPAR  { $$ = $2; }
                 | NUMBER { $$ = $1; };
```

Script Result Values - 2

```
~ tsk:script ~~ f:Unit ~:Int = @onDeactivateWithSuccess{f}: tsk


expr(?r:Int) = {!r=0!}; var t:Int ~<  term(?t)>~~r+=t~  .. "+"
term(?r:Int) = {!r=1!}; var t:Int ~<factor(?t)>~~r*=t~  .. "*"

factor(?n:Int) = ?n + "(" expr,?n ")"


implicit num(??n:Int) = @expNum(_n): {?accept?}
```

Script Result Values - 3

```
~ task: script[Int] ~~ f: Int=>Int ~ : Int
=
    @onDeactivateWithSuccess{$ = f($task)}: task


expr  : Int = {!0!}^; ~< term >~~ $ + _ ~^  .. "+"
term  : Int = {!1!}^; ~<factor>~~ $ * _ ~^  .. "*"

factor: Int = ?$ + "(" expr^ ")"
```

One-time Flow

```
~[T,U]s:script[T]~~t:T=>script[U]~: U = if<s> t($s)^

~<a^>~~<b^>~
  a ==> b

clickHandler = click ==> handleClick(_); ...
keyHandler   = key   ==> handleKey(  _); ...

doExit = var sure=false
         exitCommand @gui:{sure=areYouSure} while(!sure)

doExit = exitCommand; @gui:areYouSure ==> while(!_)
```

Lasting Flow - 1

```
def copy(in: File, out: File): Unit = {
  val  inStream = new  FileInputStream(in)
  val outStream = new FileOutputStream(out)
  val eof = false
  while (!eof) {
    val b = inStream.read()
    if (b==-1) eof=true else outStream.write(b)
  }
   inStream.close()
  outStream.close()
}
```

Lasting Flow - 2

```
fileCopier(in:File, out:File)    =    reader(in) &==> writer(out)


reader(f:File)    = val inStream = new FileInputStream(f);
                    val b = inStream.read()  <=b   while (b!=-1);
                    inStream.close()


writer(f:File)    = val outStream = new FileOutputStream(f);
                    =>?i: Int   while (i != -1)    outStream.write(i);
                    outStream.close()



   <==>(i:Int)  = {}
```

Lasting Flow - 3

```
fileCopier  (in:File, out:File) = reader,in &==> writer,out

fileCrFilter(in:File, out:File) = reader,in &==> crFilter &==> writer,out

                                              crFilter  =  =>?c:Int  if(c!='\r') <=c  ...
```

Akka Receive: Partial scripts - 1

```
def receive = {
  case Request   (r) => sender ! calculate(r)
  case Shutdown      => context.stop(self)
  case Dangerous (r) => a.tell(Work(r),sender)
  case OtherJob  (r) => a!JobRequest(r,sender)
  case JobReply(r,s) => s ! r
}

live = .. <<
  case Request   (r) => {sender ! calculate(r)}
  case Dangerous (r) => {a.tell(Work(r),sender)}
  case OtherJob  (r) => {a!JobRequest(r,sender)}
  case JobReply(r,s) => {s!r}
  >> ;
  << Shutdown >>
```

Akka Receive: Partial scripts - 2

```
var initializationReady = false
var activeActors        = 0
var sum: Double         = 0

def receive = {
  case context: Context =>
    sum = 0 //reset the instance variables
    activeActors = 0

    for(task <- context.tasks) {
      val actor = actorOf[Delegate].start
      actor ! DoTask(task)
      activeActors += 1
    }
    initializationReady = true

  case delegateResult : Double =>
    sum += delegateResult sender.get.stop
    activeActors -= 1

    if(initializationReady && activeActors<=0) {
      clientActor ! sum
    }
}
```

Akka Receive: Partial scripts - 3

```
live = ...
       << context: Context =>
             var sum: Double = 0
             ( for(task <- context.tasks)
             & {!val actor=actorOf[Delegate].start
               actor ! DoTask(task) !}
               << d:Double => {sum+=d; sender.get.stop} >>
             )
             {clientActor ! sum}
       >>
```

Challenges

- Implementation: compiler, vm, debugger
- Unit tests
- vms for simulations, parallel execution, ...
- New features
  - split scripts
  - process lambdas
  - return values
  - data flow
  - disambiguation
- Documentation, papers, ...

Conclusion

- Easy and efficient programming
- Support in Scalac branch
- Simple implementation: 5000 lines
- Still much to do and to discover
- Open Source:
  subscript-lang.org
  github.com/AndreVanDelft/scala
- Help is welcome
  - Participate!

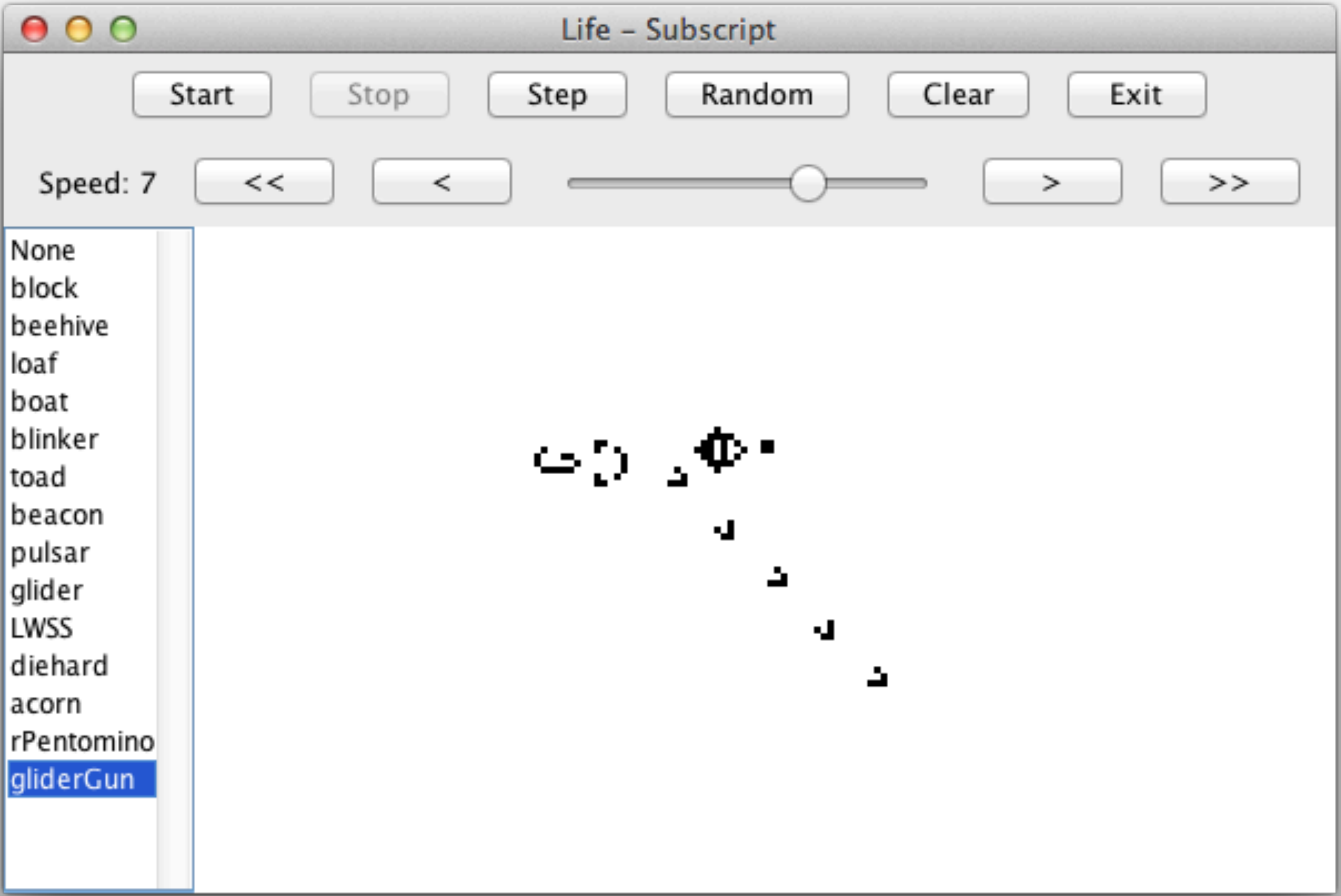
The End

- Spare Slides next

Challenge: Disambiguation

```
  a b  +  a c
..a b  ;  a c


  a b  ||  a c


  a b  |+|  a c
..a b  |;|  a c
```

Game of Life - 1



Game of Life - 2

```
  live              = ||  boardControl mouseInput speedControl doExit

 boardControl       = ...; (..singleStep) multiStep || clear || randomize

  doExit            = exitCommand var r=false @gui:{r=areYouSure} while(!r)

 randomizeCommand = randomizeButton + 'r'
     clearCommand =     clearButton + 'c'
      stepCommand =      stepButton + ' '
      exitCommand =      exitButton + windowClosing,top
multiStepStartCmd =     startButton + Key.Enter
 multiStepStopCmd =      stopButton + Key.Enter

 do1Step            = {*board.calculateGeneration*} @gui: {!board.validate!}

 randomize          =   randomizeCommand @gui: {!board.doRandomize()!}
 clear              =       clearCommand @gui: {!board.doClear      !}
 singleStep         =        stepCommand do1Step
  multiStep         = multiStepStartCmd; ...do1Step {*sleep*}
                    / multiStepStopCmd
```

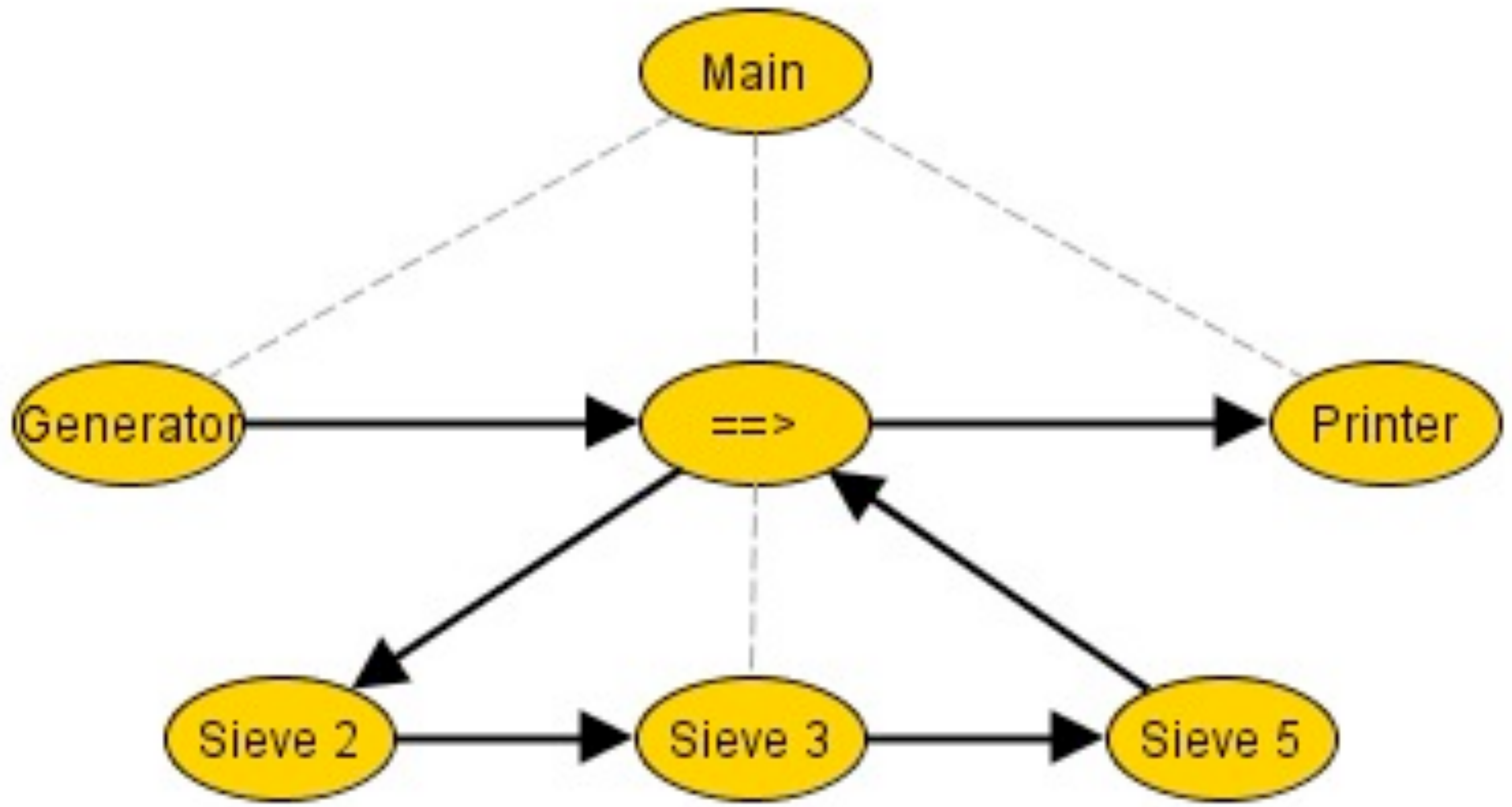
Game of Life - 3

```
 speedControl      = ...; speedKeyInput+speedButtonInput+speedSliderInput

 setSpeed(s: Int) = @gui: {!setSpeedValue(s)!}

 speedKeyInput     = times(10)
                   + val c = chr(pass_up1+'0') key(c)
                     setSpeed(digit2Speed(c))

speedButtonInput = if (speed>minSpeed) speedDec
                 + if (speed<maxSpeed) speedInc

speedDec         = minSpeedButton setSpeed,minSpeed
                 +   slowerButton setSpeed(speed-1)
speedInc         = maxSpeedButton setSpeed,maxSpeed
                 +   fasterButton setSpeed(speed+1)

speedSliderInput = speedSlider setSpeed,speedSlider.value
```

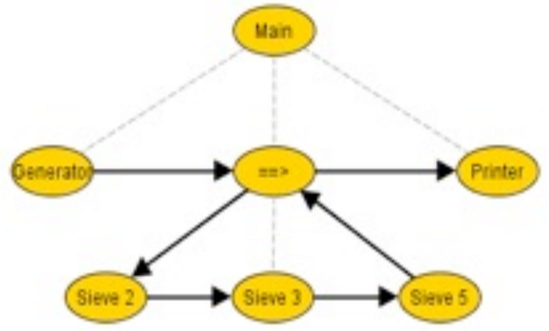
Game of Life - 4

```
   mouseInput      = (mouseClickInput & mouseDragInput)
                   /  doubleClick
                      (mouseMoveInput / doubleClick {!resetLastMousePos!}); ...

  mouseClickInput  = var p:java.awt.Point=null
                   ; var doubleClickTimeout=false
                     mouseSingleClick, board, p?
                     {! resetLastMousePos !}
                     ( {*sleep_ms(220); doubleClickTimeout=true*}
                     / mouseDoubleClick, board, p? )
                     while (!doubleClickTimeout)
                   ; {! handleMouseSingleClick(p) !}
                   ; ...

 mouseMoveInput    = mouseMoves(    board,(e:MouseEvent)=>handleMove(e.point))
 mouseDragInput    = mouseDraggings(board,(e:MouseEvent)=>handleDrag(e.point))
                   / (mouse_Released  {!resetLastMousePos!})
                   ; ...
```

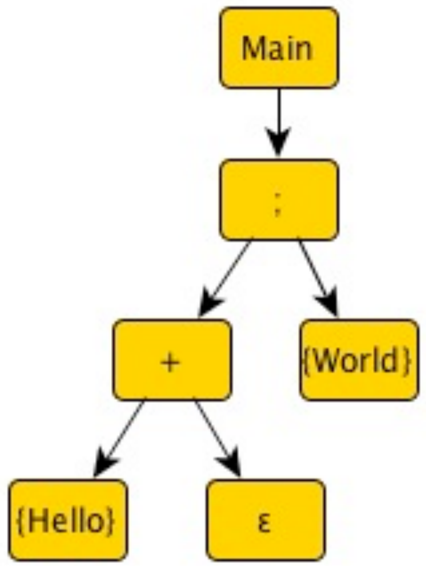
Sieve of Eratosthenes - 1



Sieve of Eratosthenes - 2

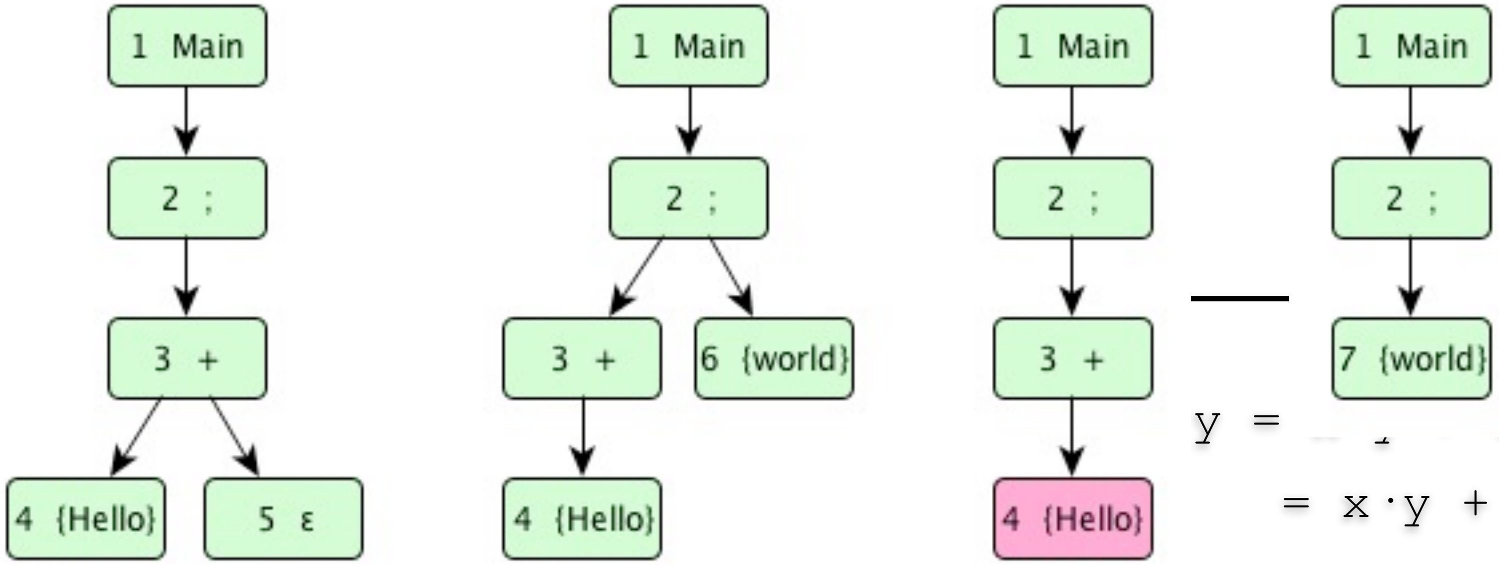


```
main = generator(2,1000000)
                 ==> (..==>sieve)
   =={toPrint}==> printer

generator(s:Int,e:Int) = for(i<-s to e) <=i

sieve              =   =>?p:Int    @toPrint:<=p;
                     ..=>?i:Int if (i%p!=0) <=i

printer            = ..=>?i:Int println,i

<==>(i:Int)        = {}
```



Templates & Call Graphs

```
{Hello}+ε; {World}
```



$$y = \quad \cdot y$$
$$= x \cdot y + y$$

Experience - 1

- Scriptic: Java based predecessor
- In production since 2010
- Analyse technical documentation
- Input: ODF ~ XML Stream
- Fun to use mixture of grammar and 'normal' code
- Parser expectations to scanner

```
implicit text(??s: String) = @expect(here,  TextToken(_s): {?accept(here)?}
implicit number(??n:  Int) = @expect(here,NumberToken(_n): {?accept(here)?}
```

- 30,000 accepted of 120,000 expected tokens per second

Experience - 2

Low level scripts

```
anyText         = ?s: String
anyLine         = anyText endOfLine

someEmptyLines = ..endOfLine
someLines       = ..anyLine
```

Experience - 3

For-usage

```
tableRow(ss: String*) = startRow; for(s<-ss) cell(s); endRow
```

```
oneOf(r?: String, ss: String*) = for(s<-ss) + s {! r=s !}
```

Experience - 4

If-usage

```
footnoteRef(n?: Int) = "(" n? ")"



footnote(n?: Int,
         s?: String) =  if (fnFormat==NUMBER_DOT) (n? ".")
                        else (footnoteRef,n? "-")
                        s?
                        endOfLine
```

Experience - 5

Grammar ambiguity

```
  var s: String
  var n: Int

  startCell s? endCell  +  startCell n? endCell
  startCell s? endCell  ||  startCell n? endCell
  startCell s? endCell |+| startCell n? endCell
```

```
xmlTag(t: XMLTag),.. = @expect(here, t) {?accept(here)?}
```