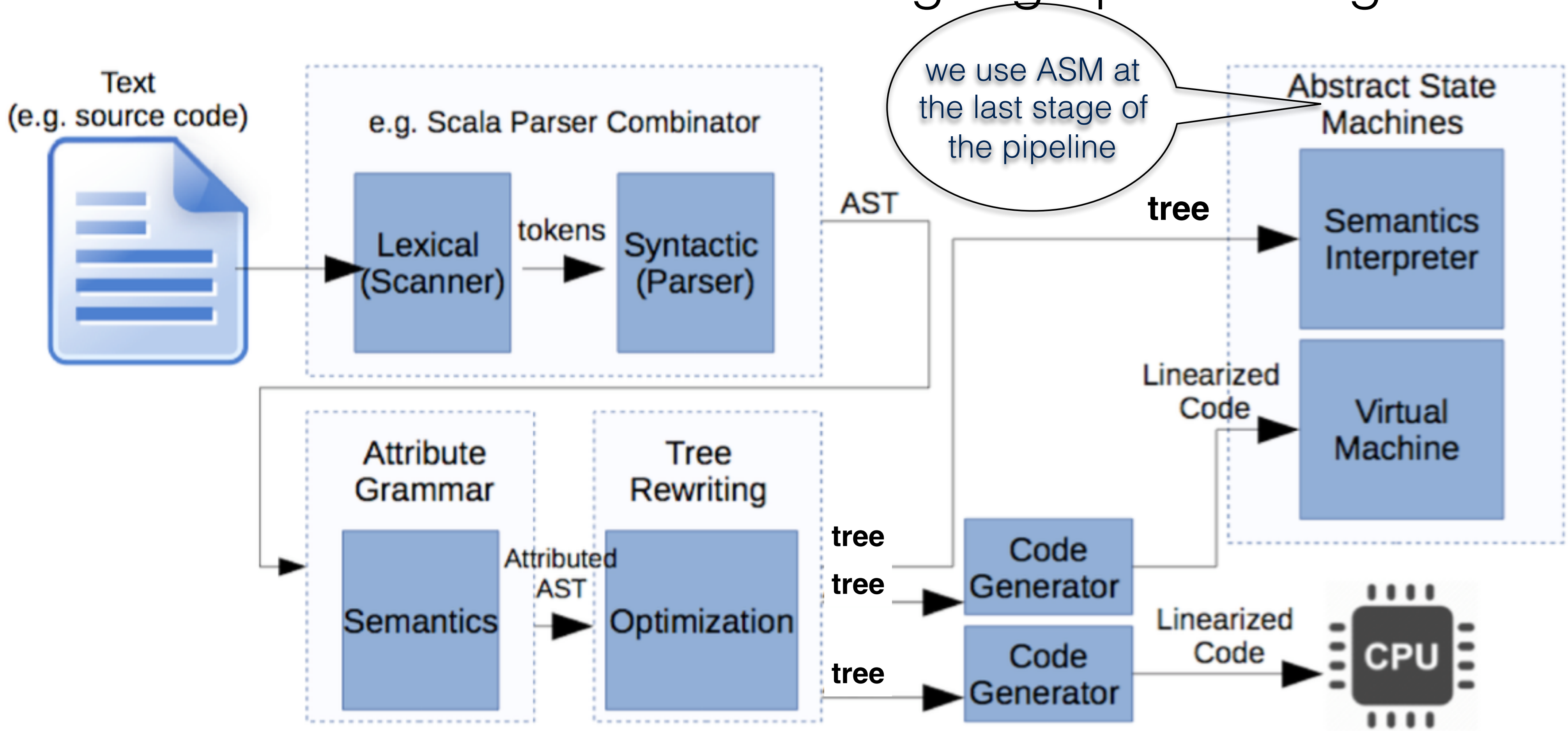
Evaluating Kiama Abstract State Machines for a Java Implementation

Pongsak Suvanpong and Anthony Sloane
Department of Computing, Macquarie University

MACQUARIE
University
SYDNEY · AUSTRALIA

Kiama

A Scala internal DSL for language processing



An example of a language processing pipeline in Kiama

MACQUARIE University SYDNEY·AUSTRALIA

Objectives

- We are interested in using Abstract State Machines (ASM) to execute programming languages.
- We want to develop techniques in Scala, so that we can take ASM definitions and quickly code them in Scala.
- Our aim is to be able to closely replicate the ASM definition written in the JBOOK.

MACQUARIE University SYDNEY·AUSTRALIA

Abstract State Machines(ASM)

- "ASM captures in mathematically rigorous yet transparent form some fundamental operational intuitions of computing, and the **notation is familiar from programming practice and mathematical standards.**"[JBOOK]

- Pseudocode (notation) over abstract data (abstract state).

- Discrete time-step execution model.

MACQUARIE
University
SYDNEY · AUSTRALIA

ASM
State and Rule

- A state in ASM is an n-arity function $f(a_1,a_2,\ldots,a_n)$ where $a_1, a_2, \ldots, a_n$ are called *locations and f* is the state name

- A state can be thought as a memory unit of ASM which allows the read/write operations. The location abstracts away the memory addressing.

- In each time-step, all rules are executed which may update states.

- An update to a state is not visible until the next time-step.

- Rules are the control logic of ASM.

MACQUARIE
University
SYDNEY·AUSTRALIA

ASM execution model

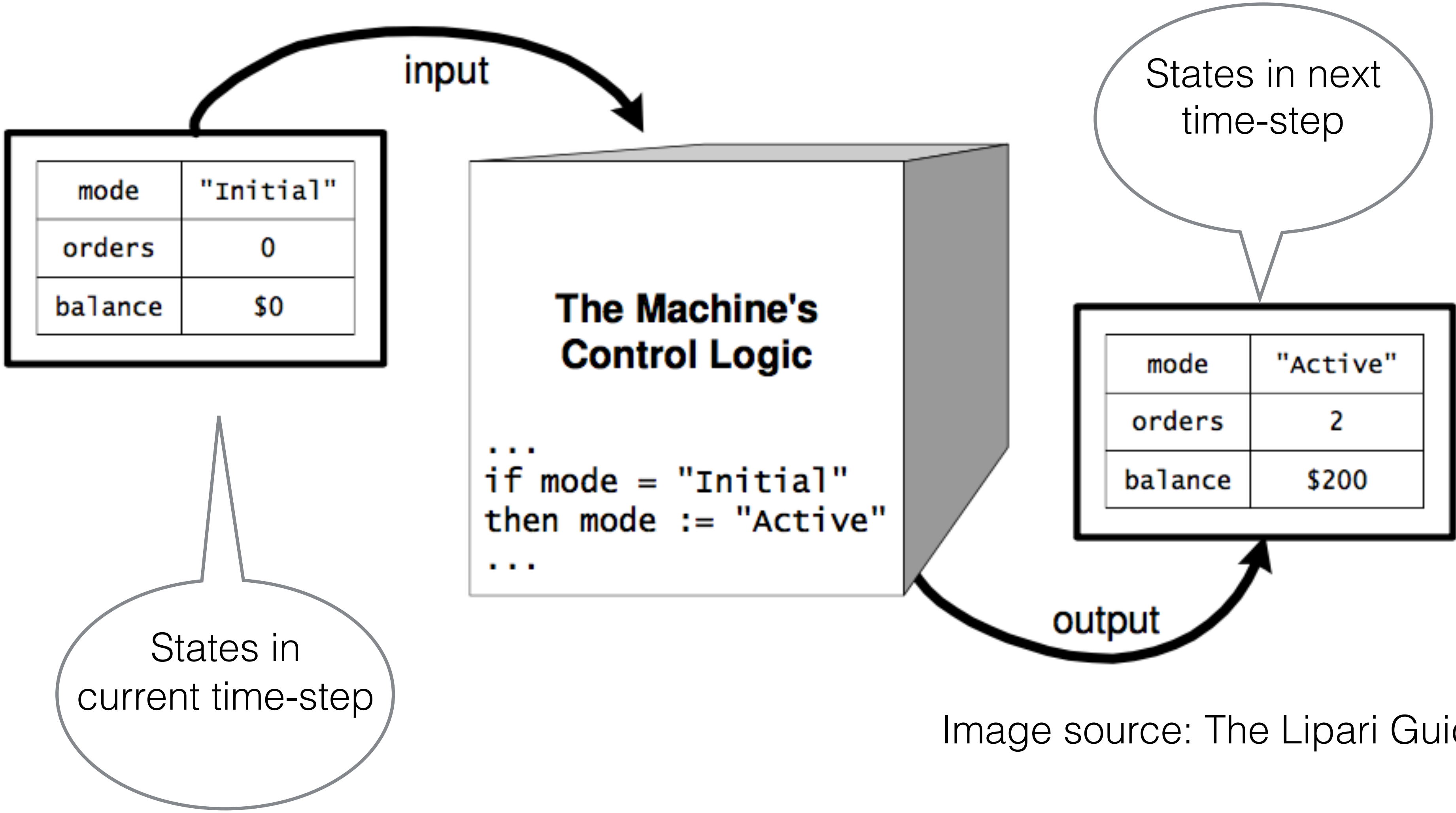


Image source: The Lipari Guide

ASM
example: estimate $\log_2(9)$

nullary state

```
Function N: ⇒ Number
InitRule =
  N := 9

MainRule =
  if N > 1 then N := N / 2
  else skip
```

Step	Value of State N
0 (Init)	9
1	4
2	2
3	1

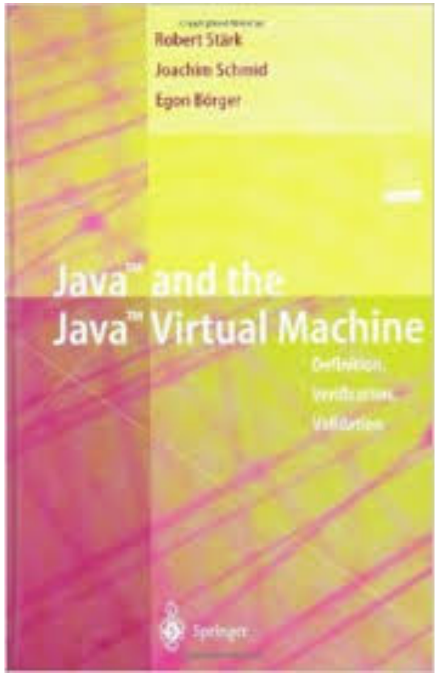
The number of steps is the result.

```
Function N: ⇒ Number
InitRule =
  N := 9

MainRule =
  if N > 1 then N := N / 2
  else skip
```

Log2 machine in standard ASM notation

Log2 machine in Kiama

```
class Log2ASM(n:Int) extends Machine("Log2ASM")
{
  val N = new State[Int]("N")

  override def init(): Unit = N := n
  override def main(): Unit =
  {
    if(N > 1)
      N := N / 2
  }
}
```



Java and The Java Virtual Machine (JBOOK).

- It defines ASM definitions of the semantics of the Java language, the compiler and the JVM.
- It mathematically proves that the execution of the semantics ASM and the JVM ASM is equivalent.

$$
\begin{array}{l}
\mathit{execJavaExp}_I = \textbf{case } \mathit{context}(\mathit{pos}) \textbf{ of} \\
\quad \mathit{lit} \rightarrow \mathit{yield}(\mathit{JLS}(\mathit{lit})) \\
\\
\quad \mathit{loc} \rightarrow \mathit{yield}(\mathit{locals}(\mathit{loc})) \\
\\
\quad \mathit{uop}\ {}^{\alpha}\mathit{exp} \rightarrow \mathit{pos} := \alpha \\
\quad \mathit{uop}\ {}^{\blacktriangleright}\mathit{val} \rightarrow \mathit{yieldUp}(\mathit{JLS}(\mathit{uop}, \mathit{val})) \\
\\
\quad {}^{\alpha}\mathit{exp}_1\ \mathit{bop}\ {}^{\beta}\mathit{exp}_2 \rightarrow \mathit{pos} := \alpha \\
\quad {}^{\blacktriangleright}\mathit{val}\ \mathit{bop}\ {}^{\beta}\mathit{exp} \quad \rightarrow \mathit{pos} := \beta \\
\quad {}^{\alpha}\mathit{val}_1\ \mathit{bop}\ {}^{\blacktriangleright}\mathit{val}_2 \rightarrow \textbf{if } \neg(\mathit{bop} \in \mathit{divMod} \wedge \mathit{isZero}(\mathit{val}_2)) \textbf{ then} \\
\qquad\qquad\qquad\qquad \mathit{yieldUp}(\mathit{JLS}(\mathit{bop}, \mathit{val}_1, \mathit{val}_2)) \\
\\
\quad \mathit{loc} = {}^{\alpha}\mathit{exp} \rightarrow \mathit{pos} := \alpha \\
\quad \mathit{loc} = {}^{\blacktriangleright}\mathit{val} \rightarrow \mathit{locals} := \mathit{locals} \oplus \{(\mathit{loc}, \mathit{val})\} \\
\qquad\qquad\qquad\quad \mathit{yieldUp}(\mathit{val})
\end{array}
$$

The
rules use pattern (left hand side of ->)
matching on concrete syntax.
The := symbol updates a state with the
right hand side value

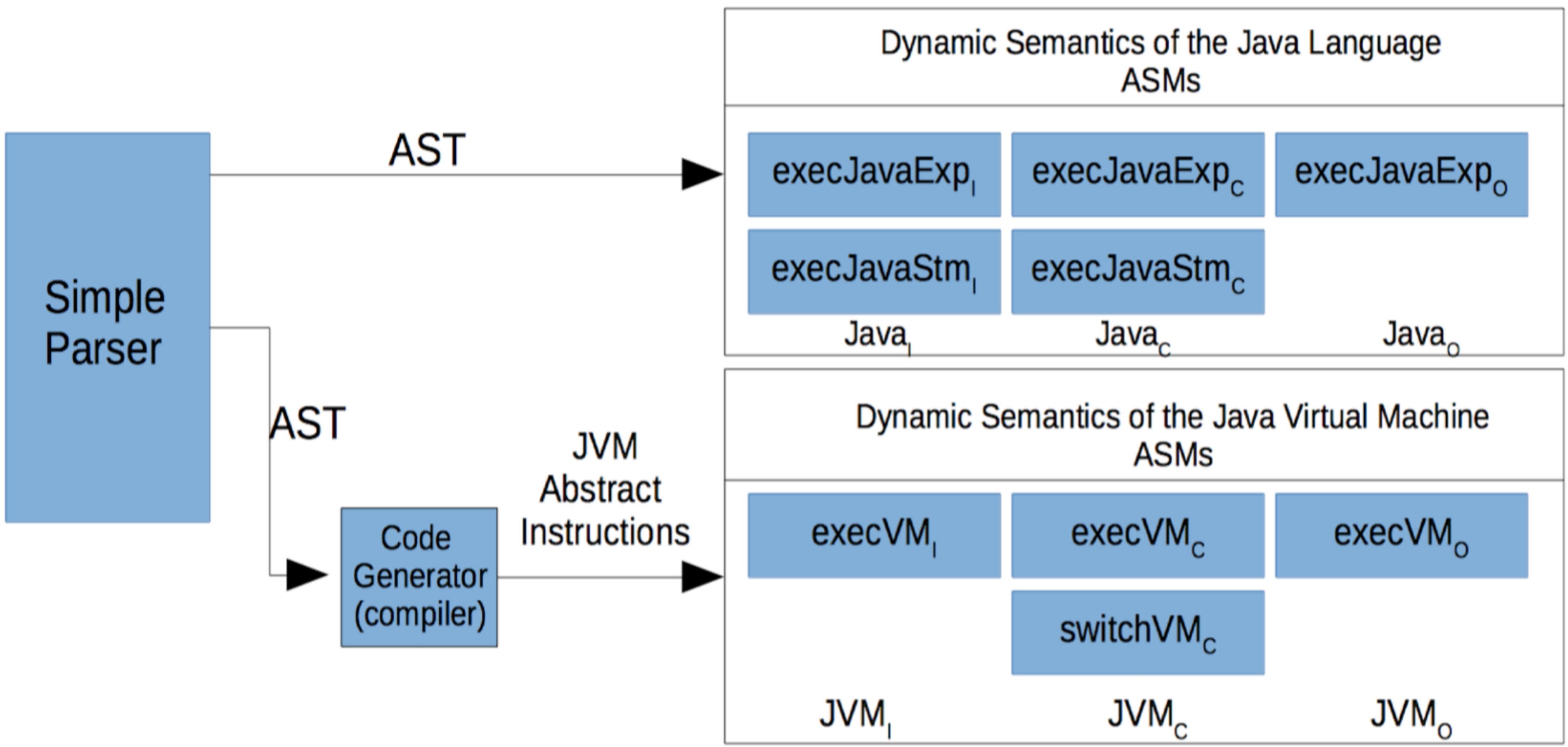
```
a == 1 ? b + 2 : b - 2;
```

$$
\boxed{
\begin{array}{l}
{}^{\alpha}\mathit{exp}_0\ ?\ {}^{\beta}\mathit{exp}_1\ :\ {}^{\gamma}\mathit{exp}_2 \rightarrow \mathit{pos} := \alpha \\
{}^{\blacktriangleright}\mathit{val}\ ?\ {}^{\beta}\mathit{exp}_1\ :\ {}^{\gamma}\mathit{exp}_2 \rightarrow \textbf{if } \mathit{val} \textbf{ then } \mathit{pos} := \beta \textbf{ else } \mathit{pos} := \gamma \\
{}^{\alpha}\mathit{True}\ ?\ {}^{\blacktriangleright}\mathit{val}\ :\ {}^{\gamma}\mathit{exp} \rightarrow \mathit{yieldUp}(\mathit{val}) \\
{}^{\alpha}\mathit{False}\ ?\ {}^{\beta}\mathit{exp}\ :\ {}^{\blacktriangleright}\mathit{val} \rightarrow \mathit{yieldUp}(\mathit{val})
\end{array}
}
$$

This Java inline
condition matches
with these set of
rules

$$
\begin{array}{l}
\mathit{execJavaExp}_I = \textbf{case } \mathit{context}(\mathit{pos}) \textbf{ of} \\
\quad \mathit{lit} \rightarrow \mathit{yield}(\mathit{JLS}(\mathit{lit})) \\
\\
\quad \mathit{loc} \rightarrow \mathit{yield}(\mathit{locals}(\mathit{loc})) \\
\\
\quad \mathit{uop}\,{}^{\alpha}\mathit{exp} \rightarrow \mathit{pos} := \alpha \\
\quad \mathit{uop}\,{}^{\blacktriangleright}\mathit{val} \rightarrow \mathit{yieldUp}(\mathit{JLS}(\mathit{uop}, \mathit{val})) \\
\\
\quad {}^{\alpha}\mathit{exp}_1\ \mathit{bop}\ {}^{\beta}\mathit{exp}_2 \rightarrow \mathit{pos} := \alpha \\
\quad {}^{\blacktriangleright}\mathit{val}\ \mathit{bop}\ {}^{\beta}\mathit{exp} \quad \rightarrow \mathit{pos} := \beta \\
\quad {}^{\alpha}\mathit{val}_1\ \mathit{bop}\ {}^{\blacktriangleright}\mathit{val}_2 \quad \rightarrow \textbf{if } \neg(\mathit{bop} \in \mathit{divMod} \wedge \mathit{isZero}(\mathit{val}_2)) \textbf{ then} \\
\qquad\qquad\qquad\qquad\qquad \mathit{yieldUp}(\mathit{JLS}(\mathit{bop}, \mathit{val}_1, \mathit{val}_2)) \\
\\
\quad \mathit{loc} = {}^{\alpha}\mathit{exp} \rightarrow \mathit{pos} := \alpha \\
\quad \mathit{loc} = {}^{\blacktriangleright}\mathit{val} \rightarrow \mathit{locals} := \mathit{locals} \oplus \{(\mathit{loc}, \mathit{val})\} \\
\qquad\qquad\qquad\quad \mathit{yieldUp}(\mathit{val}) \\
\\
\quad {}^{\alpha}\mathit{exp}_0\,?\,{}^{\beta}\mathit{exp}_1 : {}^{\gamma}\mathit{exp}_2 \rightarrow \mathit{pos} := \alpha \\
\quad {}^{\blacktriangleright}\mathit{val}\,?\,{}^{\beta}\mathit{exp}_1 : {}^{\gamma}\mathit{exp}_2 \quad \rightarrow \textbf{if } \mathit{val} \textbf{ then } \mathit{pos} := \beta \textbf{ else } \mathit{pos} := \gamma \\
\quad {}^{\alpha}\mathit{True}\,?\,{}^{\blacktriangleright}\mathit{val} : {}^{\gamma}\mathit{exp} \quad \rightarrow \mathit{yieldUp}(\mathit{val}) \\
\quad {}^{\alpha}\mathit{False}\,?\,{}^{\beta}\mathit{exp} : {}^{\blacktriangleright}\mathit{val} \quad \rightarrow \mathit{yieldUp}(\mathit{val})
\end{array}
$$

- This is just mathematical notation of an ASM, it is not executable.
- We want to be able to write Scala code as close as possible to notation (to reduce translation effort) and execute it in computers.


Simple Parser
AST
AST
Code Generator (compiler)
JVM Abstract Instructions
Dynamic Semantics of the Java Language ASMs
execJavaExp$_I$
execJavaExp$_C$
execJavaExp$_O$
execJavaStm$_I$
execJavaStm$_C$
Java$_I$
Java$_C$
Java$_O$
Dynamic Semantics of the Java Virtual Machine ASMs
execVM$_I$
execVM$_C$
execVM$_O$
switchVM$_C$
JVM$_I$
JVM$_C$
JVM$_O$

$$
\begin{array}{l}
execJavaExp_I = \textbf{case}\ context(pos)\ \textbf{of} \\
\quad lit \rightarrow yield(JLS(lit)) \\
\quad loc \rightarrow yield(locals(loc)) \\
\\
\quad uop\ {}^{\alpha}exp \rightarrow pos := \alpha \\
\quad uop\ {}^{\blacktriangleright}val \rightarrow yieldUp(JLS(uop, val)) \\
\\
{}^{\alpha}exp_1\ \mathrm{bop}\ {}^{\beta}exp_2 \rightarrow pos := \alpha \\
{}^{\blacktriangleright}val\ \mathrm{bop}\ {}^{\beta}exp_2 \rightarrow pos := \beta \\
{}^{\blacktriangleright}val_1\ \mathrm{bop}\ {}^{\blacktriangleright}val_2 \rightarrow \textbf{if}\ \neg(bop \in divMod \wedge isZero(val_2))\ \textbf{then}\ yieldUp(JLS(bop, val_1, val_2)) \\
\\
\quad loc = {}^{\alpha}exp \rightarrow pos := \alpha \\
\quad loc = {}^{\blacktriangleright}val \rightarrow locals := locals \oplus \{(loc, val)\} \\
\qquad\qquad\qquad\qquad\qquad yieldUp(val) \\
\\
{}^{\alpha}exp_0\ ?\ {}^{\beta}exp_1 : {}^{\gamma}exp_2 \rightarrow pos := \alpha \\
{}^{\blacktriangleright}val\ ?\ {}^{\beta}exp_1 : {}^{\gamma}exp_2 \rightarrow \textbf{if}\ val\ \textbf{then}\ pos := \beta\ else\ pos := \gamma \\
{}^{\alpha}true\ ?\ {}^{\blacktriangleright}val : {}^{\gamma}exp_2 \rightarrow yieldUp(val) \\
{}^{\alpha}false\ ?\ {}^{\beta}exp1 : {}^{\blacktriangleright}val \rightarrow yeildUp(val)
\end{array}
$$

The JBOOK's ASM definition to execute the semantics of the imperative core Java expressions

```
private def execJavaExpI: Unit=
{
 val node = context(pos)
 node match
 {
  case lit:Lit                                     => yieldResult(JLS(lit))
  case Local(name)                                 => yieldResult(locals(name))
  case UnaryOp(op, Value(v))                       => yieldResultUp(JLS(op, v))
  case UnaryOp(_, exp)                             => pos := exp
  case BinaryOp(op, Value(left), Value(right))     => yieldResultUp(JLS(op, left, right))
  case BinaryOp(_, Value(_), exp2)                 => pos := exp2
  case BinaryOp(_, exp1, _)                        => pos := exp1
  case Asgn(loc, Value(v))                         => locals(loc) := v
                                                      yieldResultUp(v)
  case Asgn(_, exp)                                => pos := exp
  case InlineCond(BooleanValue(_), Value(v), _)    => yieldResultUp(v)
  case InlineCond(BooleanValue(_), _, Value(v))    => yieldResultUp(v)
  case InlineCond(BooleanValue(v), exp1, exp2)     => if(v.value) pos := exp1 else pos := exp2
  case InlineCond(exp0, _, _)                      => pos := exp2
 }
}
```

Our Scala/ Kiama code

```scala
private def execVMi(inst:Instruction): Unit =
{
 inst match
 {
  case Prim(p) =>
   val (opdP, ws) = split(opd, argSize(p))
   opd := opdP ::: JVMS(p, ws)
   pc := pc + 1
  case Dupx(s1, s2) =>
   val (opdP, ws1::ws2::_) = splits(opd, List(s1, s2))
   opd := opdP ::: ws2 ::: ws1 ::: ws2
   pc := pc + 1
  case Pop(s) =>
   val (opdP, ws) = split(opd, s)
   opd := opdP
   pc := pc + 1
  case Load(t, x) =>
   if(1 == size(t))
    opd := opd :+ reg.value(x)
   else
    opd := opd :+ reg.value(x) :+ reg.value(x + 1)
   pc := pc + 1
  case Store(t, x) =>
   val (opdP, ws) = split(opd, size(t))
   if(1 == size(t))
    reg(x) := ws(0)
   else
   {
    reg(x) := ws(0)
    reg(x + 1) := ws(1)
   }
   opd := opdP
   pc := pc + 1
  case Goto(offset) => pc := offset
  case Cond(p, offset) =>
   val (opdP, ws) = split(opd, argSize(p))
   opd := opdP
   if(1 == JVMS(p, ws).head)
    pc := offset
   else
    pc := pc + 1
  case Halt() => halt := "Halt"
  case _ =>
 }
}
```

Our Scala/ Kiama code

```
execVM_I(instr) = case instr of
  Prim(p) → let(opd', ws) = split(opd, argSize(p))
             if p ∈ divMod → sndArgIsNotZero(ws) then
               opd := opd' · JVMS(p, ws)
               pc  := pc + 1

  Dupx(s1,s2) → let(opd',[ws_1,ws_2])=splits(opd,[s_1,s_2])
                  opd :=  opd' · ws_2 · ws_1 · ws_2
                  pc  := pc + 1

  Pop(s) → let(opd', ws) = split(opd, s)
             opd :=  opd'
             pc := pc + 1

  Load(t,x) → if size(t)=1 then opd := opd · [reg(x)]
                else opd := opd · [reg(x), reg(x + 1)]
                pc := pc + 1

  Store(t,x) → let(opd',ws) = split(opd,size(t))
                 if size(t) = 1 then
                   reg := reg ⊕ {(x, ws(0))}
                 else
                   reg := reg ⊕ {(x,ws(0)),(x+1,ws(1))}
                 opd := opd'
                 pc := pc + 1

  Goto(o) → pc := o

  Cond(p,o) → let(opd', ws) = split(opd,argSize(p))
                opd := opd'
                if JVMS(p,ws) then
                  pc := o
                else
                  pc := pc + 1

  Halt → halt := "Halt"
```

The JBOOK's ASM definition to execute the **semantics** of the imperative core **JVM** expressions

$$
\begin{aligned}
\mathcal{E}(\mathtt{lit}) &= \mathtt{Prim(lit)}\\
\mathcal{E}(\mathtt{loc}) &= \mathtt{Load}(\mathtt{T(loc)}, \overline{\mathtt{loc}})\\
\mathcal{E}(\mathtt{loc = exp}) &= \mathcal{E}(\mathtt{exp})\cdot\mathtt{Dupx(0, size(T(exp)))}\cdot\mathtt{Store}(\mathtt{T(exp)}, \overline{\mathtt{loc}})\\
\mathcal{E}(\mathtt{!exp}) &= \mathcal{B}_1(\mathtt{exp}, \mathtt{una}_1)\cdot\mathtt{Prim(1)}\cdot\mathtt{Goto}(\mathtt{una}_2)\cdot\mathtt{una}_1\cdot\mathtt{Prim(0)}\cdot\mathtt{una}_2\\
\mathcal{E}(\mathtt{uop\ exp}) &= \mathcal{E}(\mathtt{exp})\cdot\mathtt{Prim(uop)}\\
\mathcal{E}(\mathtt{exp}_1\ \mathtt{bop}\ \mathtt{exp}_2) &= \mathcal{E}(\mathtt{exp}_1)\cdot\mathcal{E}(\mathtt{exp}_2)\cdot\mathtt{Prim(bop)}\\
\mathcal{E}(\mathtt{exp}_0\ ?\ \mathtt{exp}_1 : \mathtt{exp}_0) &= \mathcal{B}_1(\mathtt{exp0}, \mathtt{if}_1)\cdot\mathcal{E}(\mathtt{exp}_0)\cdot\mathtt{Goto}(\mathtt{if}_2)\cdot\mathtt{if}_1\cdot\mathcal{E}(\mathtt{exp1})\cdot\mathtt{if}_2
\end{aligned}
$$

The JBOOK's definition of the **compiler** for the imperative core Java

```scala
private def E(node:Node): List[Instruction] =
{
 node match
 {
  case Lit(lit)                     => Prim(lit)
  case loc:Local                    => Load(T(loc), Bar(loc))
  case Asgn(loc, exp)               => E(expr) ::: Dupx(0, size(T(exp))) ::: Store(T(exp), Bar(loc))
  case UnaryOp(Op.NOT, exp)         => val una1 = LabelDef("una1") val una2 = LabelDef("una2")
                                       B1(exp, una1) ::: Prim(1) ::: Goto(una2) ::: una1 :::
                                       Prim(0) ::: una2
  case uop@UnaryOp(_, exp)          => E(exp) ::: Prim(uop)
  case bop@BinaryOp(_, exp1, exp2)  => E(exp1) ::: E(exp2) ::: Prim(bop)
  case InlineCond(exp0, exp1, exp2) => val if1 = LabelDef("if1") val if2 = LabelDef("if2")
                                       B1(exp0, if1) ::: E(exp2) ::: Goto(if2) ::: if1 :::
                                       E(exp1) ::: if2
 }
}
```

Our Scala/ Kiama code

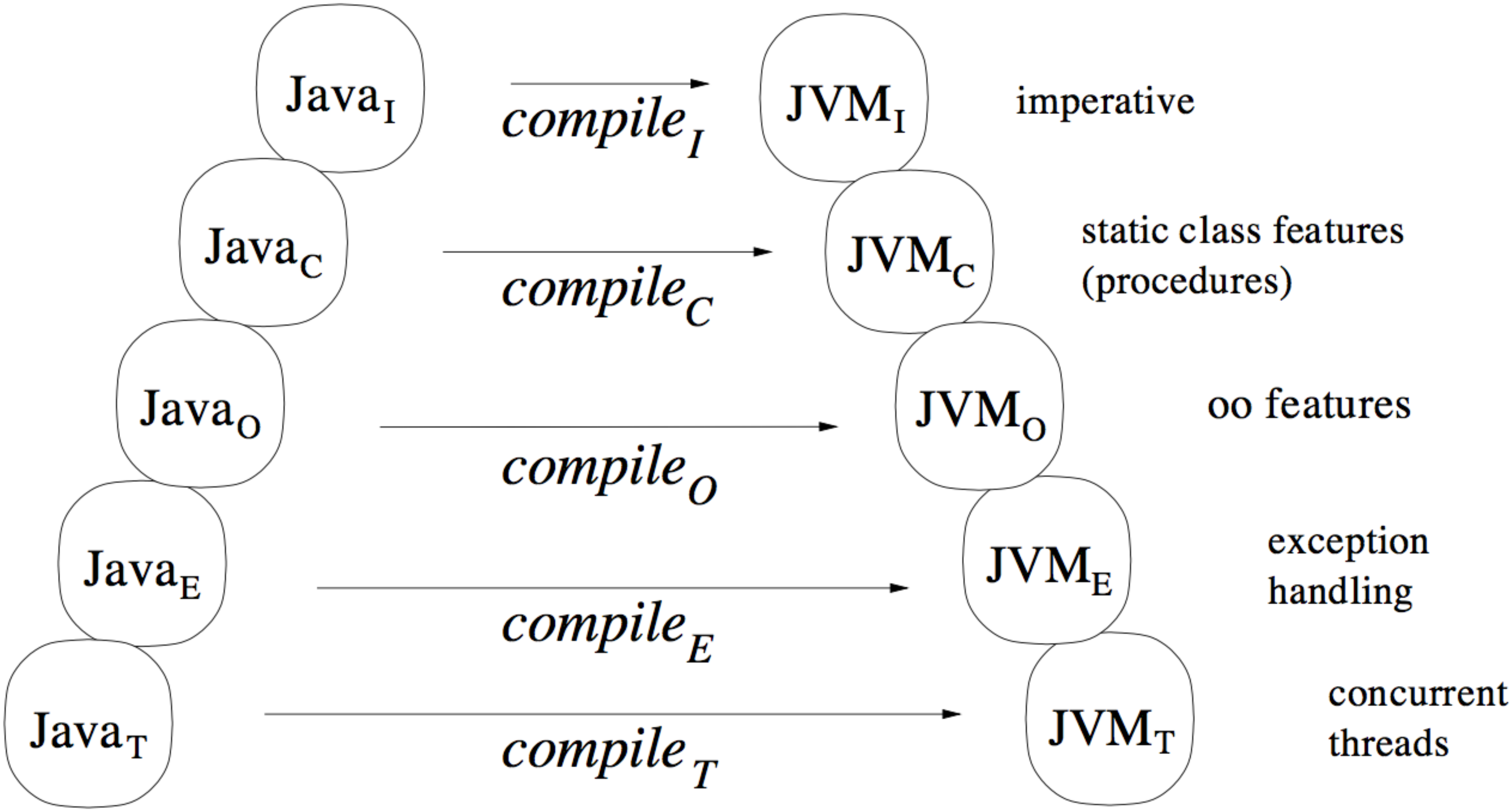
- Scala has many features which allow us to closely replicate the JBOOK ASM definitions.
- Case classes
- Pattern matching
- Extractor pattern
- Implicit functions
- Functional programming
- Kiama provides basic execution model and definition of states.

Next Step

- Develop a library based on the techniques that we've used in this study.
- —> the techniques can be reused.
- —> ASM definitions may be directly written using Kiama/Scala.
- Try using the concrete syntax in pattern matching, instead of using syntax tree (case classes)

MACQUARIE
University
SYDNEY · AUSTRALIA

Fin


$Java_I$
$compile_I$
JVM_I
imperative
$Java_C$
$compile_C$
JVM_C
static class features (procedures)
$Java_O$
$compile_O$
JVM_O
oo features
$Java_E$
$compile_E$
JVM_E
exception handling
$Java_T$
$compile_T$
JVM_T
concurrent threads

step	transition rule	Source matched	pos	source at pos
1	$^{\alpha}exp_0 \ ?\ ^{\beta}exp_1 : {}^{\gamma}exp_2 \rightarrow pos := \alpha$	1 == 1 ? 1 + 2 : -3	$^{\alpha}exp_0$	1==1
2	$^{\alpha}exp_1$ bop $^{\beta}exp_2 \rightarrow pos := \alpha$	1==1	$^{\alpha}exp_1$	1
3	$lit \rightarrow yield(JLS(lit))$	1	Val(1)	1
4	$\vdash val$ bop $^{\beta}exp_2 \rightarrow pos := \beta$	Val(1) == 1 ? 1 + 2 : -3	$^{\beta}exp_2$	1
5	$lit \rightarrow yield(JLS(lit))$	1	Val(1)	1
6	$\vdash val_1$ bop $\vdash val_2 \rightarrow$	Val(1) == Val(1)	Val(true)	1==1
	if $\neg(bop \in divMod \wedge isZero(val_2))$			
	then $yieldUp(JLS(bop, val_1, val_2))$			
7	$\vdash val \ ?\ ^{\beta}exp_1 : {}^{\gamma}exp_2 \rightarrow$	val(1) == val(1) ? 1 + 2 : -3	$^{\beta}exp_1$	1+2
	if val **then** $pos := \beta$ else $pos := \gamma$			
8	$^{\alpha}exp_1$ bop $^{\beta}exp_2 \rightarrow pos := \alpha$	1 + 2	$^{\alpha}exp_1$	1
9	$lit \rightarrow yield(JLS(lit))$	1	Val(1)	1
10	$\vdash val$ bop $^{\beta}exp_2 \rightarrow pos := \beta$	Val(1) + 2	$^{\beta}exp_2$	2
11	$lit \rightarrow yield(JLS(lit))$	2	Val(2)	2
12	$\vdash val_1$ bop $\vdash val_2 \rightarrow$	Val(1) + Val(2)	Val(3)	1 + 2
13	$^{\alpha}true\ ?\ \vdash val : {}^{\gamma}exp_2 \rightarrow yieldUp(val)$	Val(true) ? Val(3) : -3	Val(3)	

Table 2: Step by step evaluation of an inline condition 1 == 1 ? 1+2 : -3.