

How do you create a programming language for the JVM?

Federico Tomassetti

Hi, I am Federico



tripadvisor®

GROUPON

- Got a PhD in Language Engineering

- Lived here and there

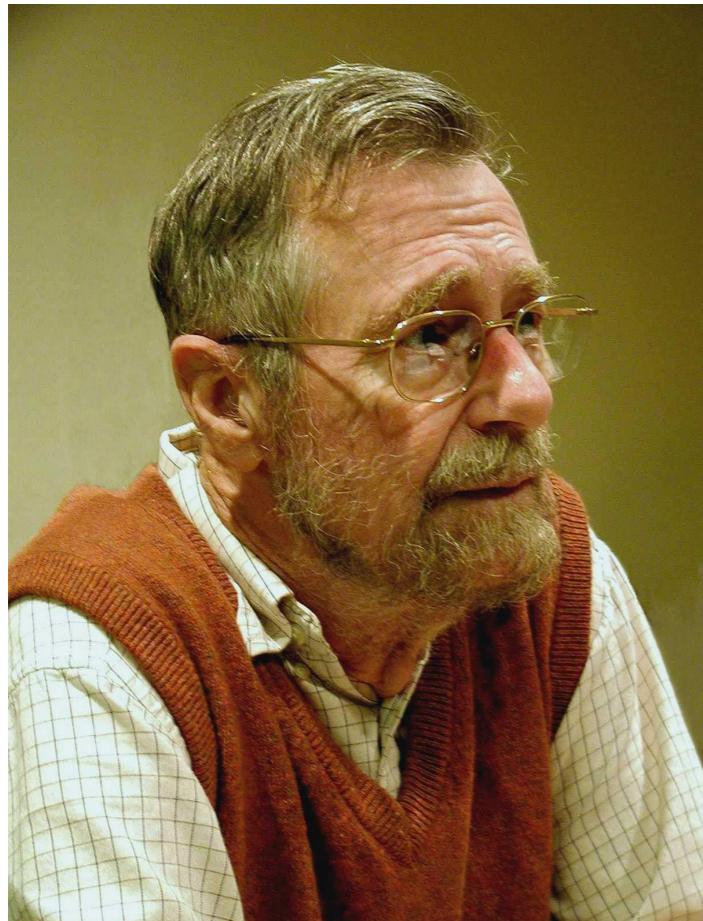
- Now for a living :
 - Create languages
 - Write parsers
 - Transform code
 - Build compilers



strumenta.com



Why creating languages?



The language or notation we are using to express or record our thoughts, are the major factors determining what we can think or express at all!

Edsger W. Dijkstra
The Humble Programmer

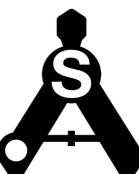
Motivation

Build Domain Specific Languages

To support domain experts or to support working on a very specific kind of applications

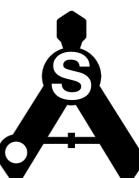
Build your very own programming language

When you want to program differently, because you can think you can do things better or just because having your own language is a lot of fun and something to be proud of

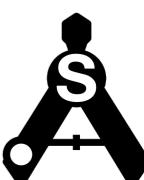
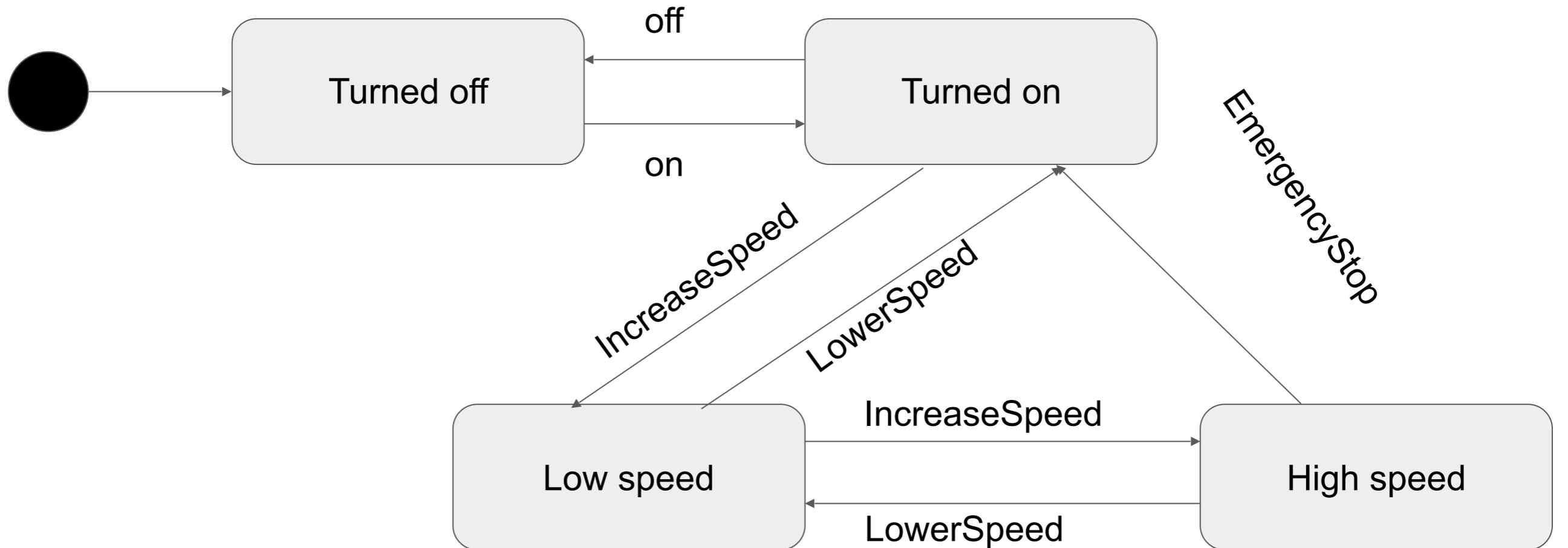


Why targeting the JVM?

- Performance: the JVM can perform optimisations through dynamic analysis and even outperform native code
- Libraries availability: whatever you want to do, there is a library for it
- Platforms supported: the JVM runs in a ton of places



State Machine



State Machine

**Software
Developers**



Write the infrastructure
code in Java/Kotlin



Class files

System

**Mechanical
Engineers**



Write the logic controlling
the machinery



Class files

An example of State Machine

```
statemachine mySm

input lowSpeedThroughput: Int
input highSpeedThroughput: Int

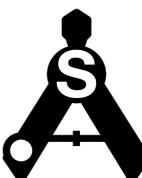
var totalProduction = 0

event turnOff
...
event emergencyStop

start state turnedOff {
    on turnOn -> turnedOn
}

state turnedOn {
    on turnOff -> turnedOff
    on speedUp -> lowSpeed
}

state lowSpeed {
    on entry {
        totalProduction = totalProduction
            + lowSpeedThroughput
        print("Producing " + lowSpeedThroughput
            + " elements (total
"+totalProduction+")")
    }
    on speedDown -> turnedOn
    on speedUp -> highSpeed
    on doNothing -> lowSpeed
}
```



General picture

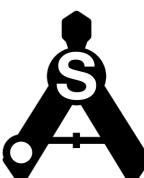
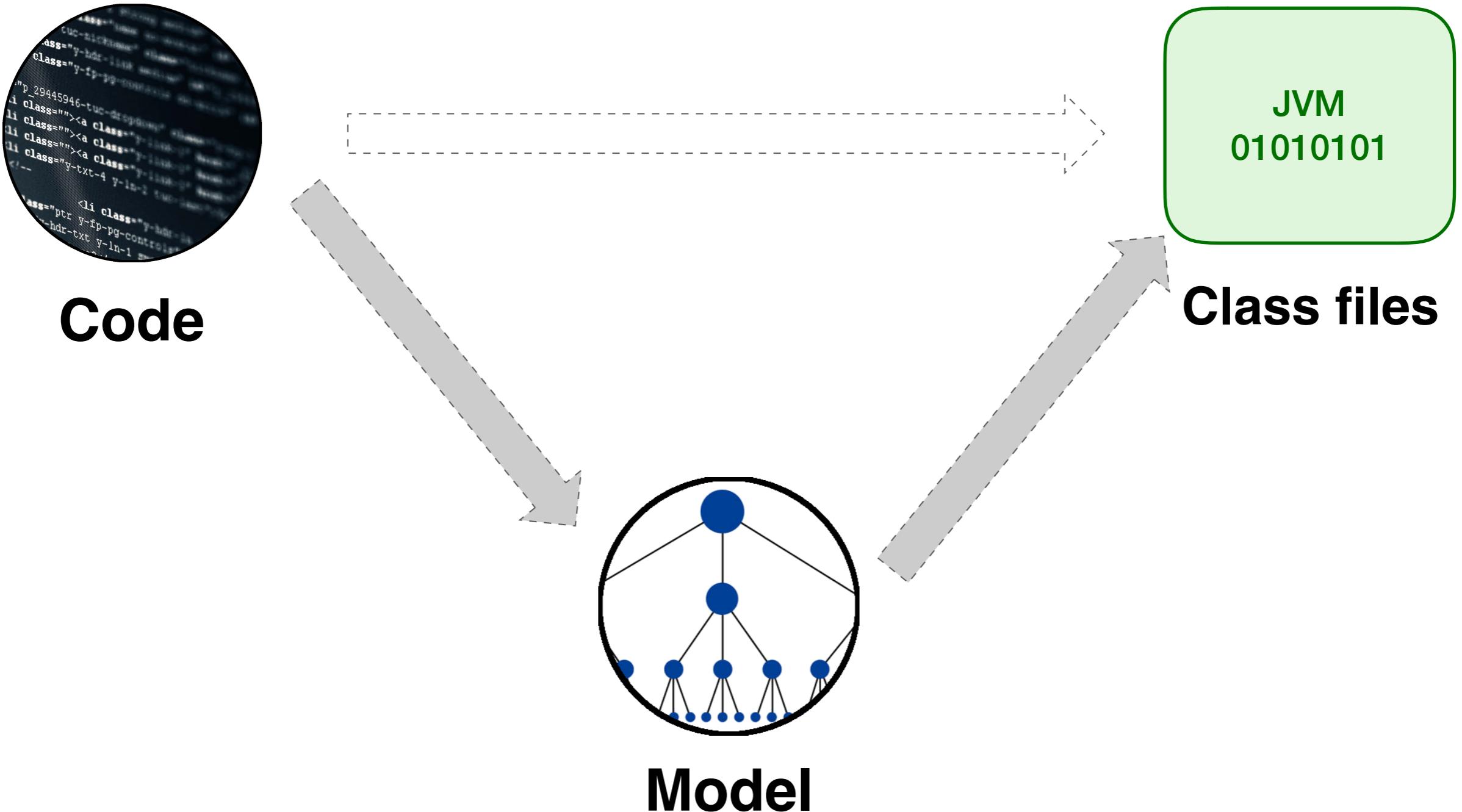


Code



Class files

General picture



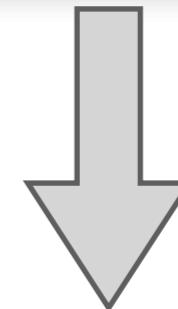
Parsing: from code to model

```
statemachine mySm  
  
input lowSpeedThroughput: Int  
input highSpeedThroughput: Int  
  
var totalProduction = 0  
  
event turnOff  
...  
event emergencyStop
```

```
start state turnedOff {  
    on turnOn -> turnedOn  
}  
  
state turnedOn {  
    on turnOff -> turnedOff  
    on speedUp -> lowSpeed  
}  
  
state lowSpeed {  
    on entry {  
        totalProduction = totalProduction  
            + lowSpeedThroughput  
        print("Producing " + lowSpeedThroughput  
            + " elements (total  
"+totalProduction+")")  
    }  
    on speedDown -> turnedOn  
    on speedUp -> highSpeed  
    on doNothing -> lowSpeed  
}
```



```
stateMachine preamble (states+=state)* EOF ;  
  
preamble : SM name=ID (elements+=preambleElement)* ;
```



```
data class StateMachine(  
    val name: String,  
    val inputs: List<InputDeclaration>,  
    val variables: List<VarDeclaration>,  
    val events: List<EventDeclaration>,  
    val states: List<StateDeclaration>)  
: Node
```



Parsing: from code to model

```
statemachine mySm
```

```
    input lowSpeedThroughput: Int  
    input highSpeedThroughput: Int
```

```
    var totalProduction = 0
```

```
    event turnOff
```

```
    ...  
    event emergencyStop
```

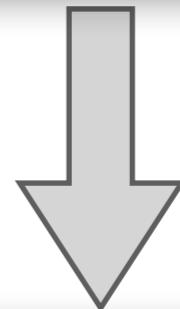
```
start state turnedOff {  
    on turnOn -> turnedOn  
}
```

```
state turnedOn {  
    on turnOff -> turnedOff  
    on speedUp -> lowSpeed  
}
```

```
state lowSpeed {  
    on entry {  
        totalProduction = totalProduction  
            + lowSpeedThroughput  
        print("Producing " + lowSpeedThroughput  
            + " elements (total  
"+totalProduction+")")  
    }  
    on speedDown -> turnedOn  
    on speedUp -> highSpeed  
    on doNothing -> lowSpeed  
}
```



```
preambleElement : EVENT name=ID # eventDecl  
| INPUT name=ID COLON type # inputDecl  
| VAR name=ID (COLON type)? ASSIGN initialValue=expression # varDecl
```



```
data class VarDeclaration(  
    override val name: String,  
    val explicitType: Type?,  
    val value: Expression)  
    : ValueDeclaration
```

```
data class InputDeclaration(  
    override val name: String,  
    override val type: Type)  
    : ValueDeclaration
```



Parsing: from code to model

```
statemachine mySm

input lowSpeedThroughput: Int
input highSpeedThroughput: Int

var totalProduction = 0

event turnOff
...
event emergencyStop

start state turnedOff {
    on turnOn -> turnedOn
}

state turnedOn {
    on turnOff -> turnedOff
    on speedUp -> lowSpeed
}

state lowSpeed {
    on entry {
        totalProduction = totalProduction
            + lowSpeedThroughput
        print("Producing " + lowSpeedThroughput
            + " elements (total
"+totalProduction+")
    }
    on speedDown -> turnedOn
    on speedUp -> highSpeed
    on doNothing -> lowSpeed
}
```

```
state : (start=START)? STATE name=ID LBRACKET (blocks+=stateBlock)* RBRACKET ;

stateBlock : ON ENTRY LBRACKET (statements+=statement)* RBRACKET # entryBlock
            | ON EXIT LBRACKET (statements+=statement)* RBRACKET # exitBlock
            | ON eventName=ID ARROW destinationName=ID # transitionBlock
            ;
```

```
data class StateDeclaration(
    override val name: String,
    val start: Boolean,
    val blocks: List<StateBlock>
) : Node, Named
```

```
data class OnEventBlock(
    val event: ReferenceByName<EventDeclaration>,
    val destination: ReferenceByName<StateDeclaration>
) : StateBlock
```

```
data class OnExitBlock(
    override val statements: List<Statement>
)
data class OnEntryBlock(
    override val statements: List<Statement>
) : StatementsBlock
```



Parsing: from code to model

```
statemachine mySm
```

```
input lowSpeedThroughput: Int
input highSpeedThroughput: In
var totalProduction = 0
event turnOff
...
event emergencyStop
```

```
start state turnedOff {
    on turnOn -> turnedOn
}
```

```
state turnedOn {
    on turnOff -> turnedOff
    on speedUp -> lowSpeed
}
```

```
state lowSpeed {
    on entry {
        totalProduction = totalProduction
            + lowSpeedThroughput
        print("Producing " + lowSpeedThroughput
            + " elements (total
"+totalProduction+")
    }
    on speedDown -> turnedOn
    on speedUp -> highSpeed
    on doNothing -> lowSpeed
}
```

```
expression : left=expression operator=(DIVISION|ASTERISK) right=expression # binaryOperation
| left=expression operator=(PLUS|MINUS) right=expression # binaryOperation
| value=expression AS targetType=type # typeConversion
| LPAREN expression RPAREN # parenExpression
| ID # valueReference
| MINUS expression # minusExpression
| INTLIT # intLiteral
| DECLIT # decimalLiteral
| STRINGLIT # stringLiteral ;
```



```
interface Expression : Node
```

```
data class SumExpression(
    override val left: Expression,
    override val right: Expression
) : BinaryExpression
```

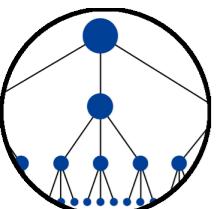
```
interface BinaryExpression
    : Expression {
    val left: Expression
    val right: Expression
}
```

```
data class IntLit(
    val value: String
) : Expression
```

```
data class ValueReference(
    val symbol: ReferenceByName<ValueDeclaration>
) : Expression
```



Parsing: from code to model



Model

```
data class StateMachine(  
    val name: String,  
    val inputs: List<InputDeclaration>,  
    val variables: List<VarDeclaration>,  
    val events: List<EventDeclaration>,  
    val states: List<StateDeclaration>)  
: Node
```

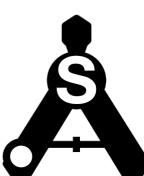
```
data class VarDeclaration(  
    data class VarDeclaration(  
        override val name: String,  
        val explicitType: Type?,  
        val value: Expression)  
    : ValueDeclaration
```

```
data class StateDeclaration(  
    override val name: String  
    data class StateDeclaration(  
        override val name: String,  
        val start: Boolean,  
        val blocks: List<StateBlock>)  
    : Node, Named
```

```
data class OnEventBlock(  
    val event: ReferenceByName<EventDeclaration>,  
    val destination: ReferenceByName<StateDeclaration>)  
: StateBlock
```

```
data class SumExpression(  
    override val left: Expression,  
    override val right: Expression)  
: BinaryExpression
```

```
data class IntLit(  
    val value: String)  
: Expression
```



Validation

```
fun StateMachine.validate() : List<Error> {
    val errors = LinkedList<Error>()

    // check a symbol or input is not duplicated
    val valuesByName = HashMap<String, Int>()
    this.specificProcess(ValueDeclaration::class.java) {
        checkForDuplicate(valuesByName, errors, it)
    }

    // check references
    this.specificProcess(ValueReference::class.java) {
        if (!it.symbol.tryToResolve(this.variables) && !it.symbol.tryToResolve(this.inputs))
    {
        errors.add(Error("A reference to symbol '${it.symbol.name}' cannot be resolved",
    it.position!!))
    }
}

// check the initial value is compatible with the explicitly declared type
this.specificProcess(VarDeclaration::class.java) {
    val valueType = it.value.type()
    if (it.explicitType != null && valueType != null && !
it.explicitType.isAssignableBy(valueType)) {
        errors.add(Error("Cannot assign ${it.explicitType} to variable of type ${
    it.value.type()}", it.position!!))
    }
}
...

return errors
}
```



How the JVM works: classes

```
public class Person {  
    private String name;  
    private int age;  
}
```

class file

```
// class version 52.0 (52)  
// access flags 0x21  
public class Person {  
  
    // access flags 0x2  
    private Ljava/lang/String; name  
  
    // access flags 0x2  
    private I age  
  
    ...  
    // Here we have the table, etc  
    ...  
}
```

The Java® Virtual Machine Specification

Java SE 9 Edition

Tim Lindholm

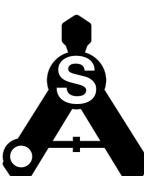
Frank Yellin

Gilad Bracha

Alex Buckley

2017-08-07

strumenta.com

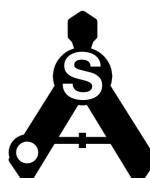


How the JVM works: math operations

```
public static int sum(int a, int b) {  
    return a + b;  
}
```

Bytecode

```
public static sum(II)I  
L0  
LINENUMBER 4 L0  
ILOAD 0  
ILOAD 1  
IADD  
IRETURN  
L1  
LOCALVARIABLE a I L0 L1 0  
LOCALVARIABLE b I L0 L1 1  
MAXSTACK = 2  
MAXLOCALS = 2
```



How the JVM works: math operations

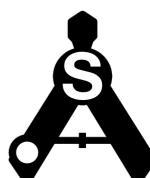
```
public static int sum(int a, int b) {  
    return a + b;  
}
```

Index	Variable
0	a
1	b

Bytecode

```
public static sum(II)I  
L0  
LINENUMBER 4 L0  
ILOAD 0  
ILOAD 1  
IADD  
IRETURN  
L1  
LOCALVARIABLE a I L0 L1 0  
LOCALVARIABLE b I L0 L1 1  
MAXSTACK = 2  
MAXLOCALS = 2
```

Stack



How the JVM works: math operations

```
public static int sum(int a, int b) {  
    return a + b;  
}
```

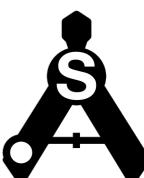
Index	Variable
0	a
1	b

Bytecode

```
public static sum(II)I  
L0  
LINENUMBER 4 L0  
ILOAD 0  
ILOAD 1  
IADD  
IRETURN  
L1  
LOCALVARIABLE a I L0 L1 0  
LOCALVARIABLE b I L0 L1 1  
MAXSTACK = 2  
MAXLOCALS = 2
```

Stack

a



How the JVM works: math operations

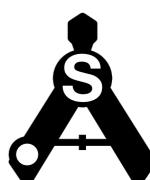
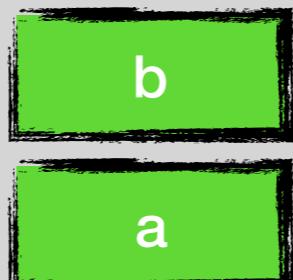
```
public static int sum(int a, int b) {  
    return a + b;  
}
```

Index	Variable
0	a
1	b

Bytecode

```
public static sum(II)I  
L0  
LINENUMBER 4 L0  
ILOAD 0  
ILOAD 1  
IADD  
IRETURN  
L1  
LOCALVARIABLE a I L0 L1 0  
LOCALVARIABLE b I L0 L1 1  
MAXSTACK = 2  
MAXLOCALS = 2
```

Stack



How the JVM works: math operations

```
public static int sum(int a, int b) {  
    return a + b;  
}
```

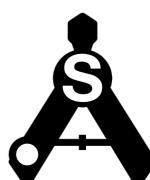
Index	Variable
0	a
1	b

Bytecode

```
public static sum(II)I  
L0  
LINENUMBER 4 L0  
ILOAD 0  
ILOAD 1  
IADD  
IRETURN  
L1  
LOCALVARIABLE a I L0 L1 0  
LOCALVARIABLE b I L0 L1 1  
MAXSTACK = 2  
MAXLOCALS = 2
```

Stack

a + b



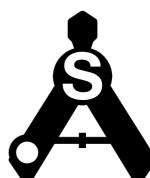
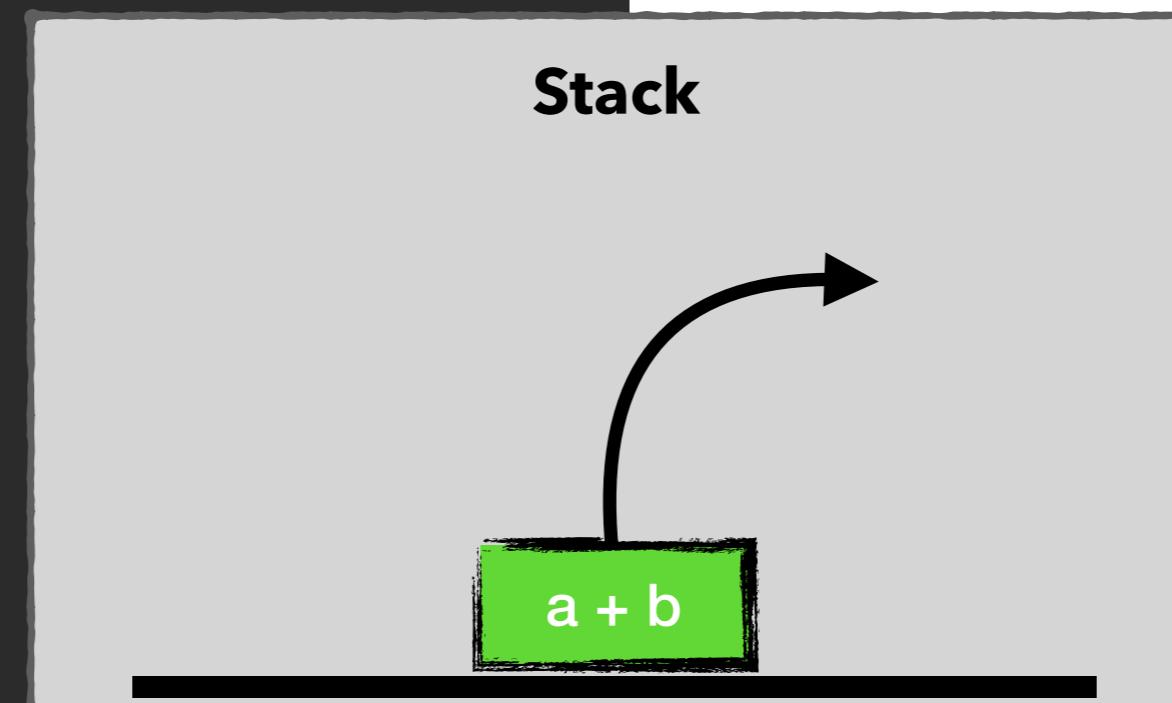
How the JVM works: math operations

```
public static int sum(int a, int b) {  
    return a + b;  
}
```

Index	Variable
0	a
1	b

Bytecode

```
public static sum(II)I  
L0  
LINENUMBER 4 L0  
ILOAD 0  
ILOAD 1  
IADD  
IRETURN  
L1  
LOCALVARIABLE a I L0 L1 0  
LOCALVARIABLE b I L0 L1 1  
MAXSTACK = 2  
MAXLOCALS = 2
```

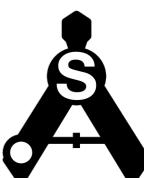


How the JVM works: math operations

```
public static float sum(float a,  
                      float b) {  
    return a + b;  
}
```

Bytecode

```
public static sum(FF)F  
L0  
LINENUMBER 8 L0  
FLOAD 0  
FLOAD 1  
FADD  
FRETURN
```

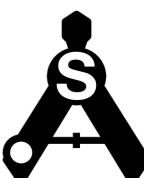


How the JVM works: math operations

```
public static int tripleSum(int a,  
    int b, int c) {  
    return a + b + c;  
}
```

Bytecode

```
public static tripleSum(III)I  
L0  
  LINENUMBER 8 L0  
  ILOAD 0  
  ILOAD 1  
  IADD  
  ILOAD 2  
  IADD  
  IRETURN
```



How the JVM works: math operations

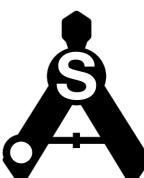
```
public static int tripleSum(int a,  
    int b, int c) {  
    return a + b + c;  
}
```

Index	Variable
0	a
1	b
2	c

Bytecode

```
public static tripleSum(III)I  
L0  
LINENUMBER 8 L0  
ILOAD 0  
ILOAD 1  
IADD  
ILOAD 2  
IADD  
IRETURN
```

Stack



How the JVM works: math operations

```
public static int tripleSum(int a,  
    int b, int c) {  
    return a + b + c;  
}
```

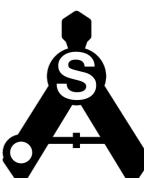
Index	Variable
0	a
1	b
2	c

Bytecode

```
public static tripleSum(III)I  
L0  
  LINENUMBER 8 L0  
  ILOAD 0  
  ILOAD 1  
  IADD  
  ILOAD 2  
  IADD  
  IRETURN
```

Stack

a



How the JVM works: math operations

```
public static int tripleSum(int a,  
    int b, int c) {  
    return a + b + c;  
}
```

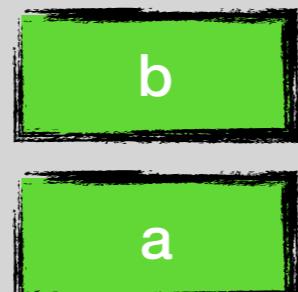
Index	Variable
0	a
1	b
2	c

Bytecode

```
public static tripleSum(III)I  
L0  
LINENUMBER 8 L0  
ILOAD 0  
ILOAD 1  
IADD  
ILOAD 2  
IADD  
IRETURN
```



Stack



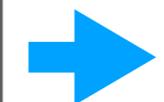
How the JVM works: math operations

```
public static int tripleSum(int a,  
    int b, int c) {  
    return a + b + c;  
}
```

Index	Variable
0	a
1	b
2	c

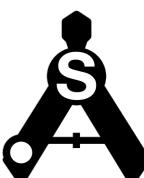
Bytecode

```
public static tripleSum(III)I  
L0  
LINENUMBER 8 L0  
ILOAD 0  
ILOAD 1  
IADD  
ILOAD 2  
IADD  
IRETURN
```



Stack

a + b



How the JVM works: math operations

```
public static int tripleSum(int a,  
    int b, int c) {  
    return a + b + c;  
}
```

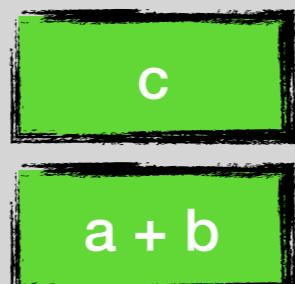
Index	Variable
0	a
1	b
2	c

Bytecode

```
public static tripleSum(III)I  
L0  
LINENUMBER 8 L0  
ILOAD 0  
ILOAD 1  
IADD  
ILOAD 2  
IADD  
IRETURN
```



Stack



How the JVM works: math operations

```
public static int tripleSum(int a,  
    int b, int c) {  
    return a + b + c;  
}
```

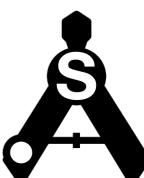
Index	Variable
0	a
1	b
2	c

Bytecode

```
public static tripleSum(III)I  
L0  
LINENUMBER 8 L0  
ILOAD 0  
ILOAD 1  
IADD  
ILOAD 2  
IADD  
IRETURN
```

Stack

a + b + c



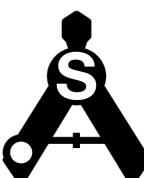
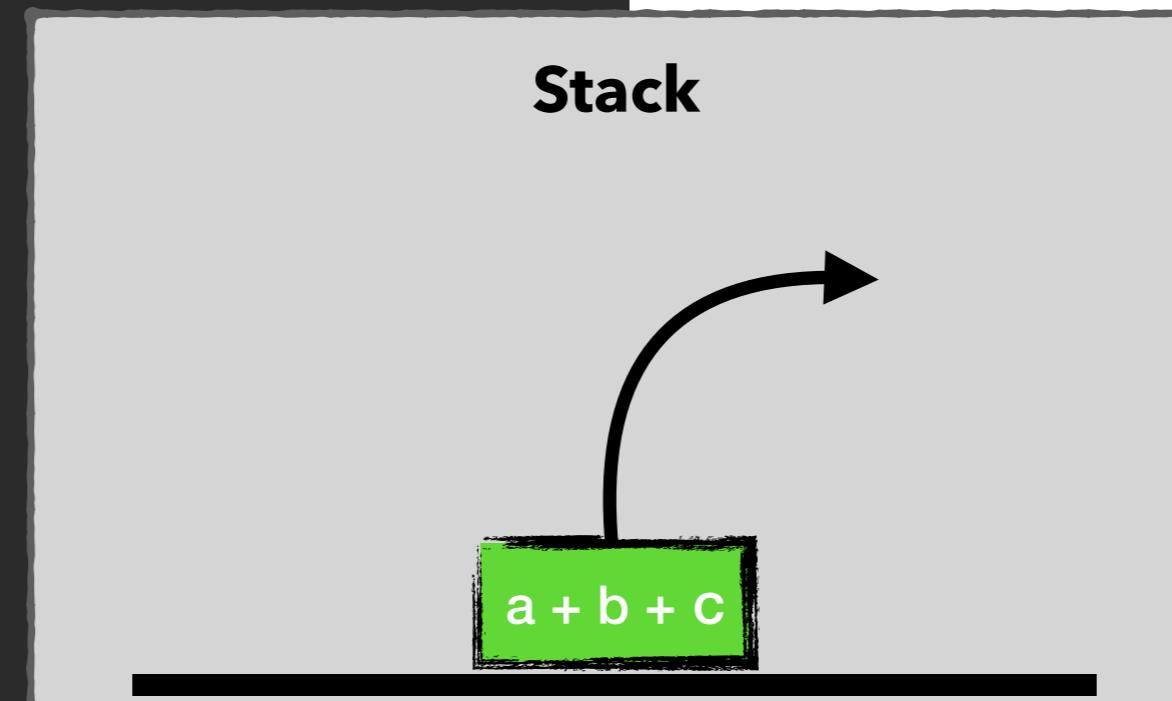
How the JVM works: math operations

```
public static int tripleSum(int a,  
    int b, int c) {  
    return a + b + c;  
}
```

Index	Variable
0	a
1	b
2	c

Bytecode

```
public static tripleSum(III)I  
L0  
LINENUMBER 8 L0  
ILOAD 0  
ILOAD 1  
IADD  
ILOAD 2  
IADD  
IRETURN
```

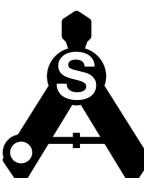


How the JVM works: math operations

```
public static int calculateMax(int a,  
    int b) {  
    return Math.max(a, b);  
}
```

Bytecode

```
public static  
    calculateMax(II)I  
L0  
    LINENUMBER 8 L0  
    ILOAD 0  
    ILOAD 1  
    INVOKESTATIC java/lang/  
Math.max (II)I  
    IRETURN  
L1  
    LOCALVARIABLE a I L0 L1 0  
    LOCALVARIABLE b I L0 L1 1  
    MAXSTACK = 2  
    MAXLOCALS = 2
```



Bytecode generation with ASM



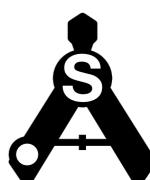
ASM is an open-source library to:

- ***Generate classes dynamically***
- ***Edit class files***
- ***Create class files***

What is out compiler?

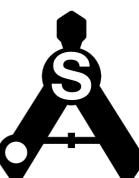
A piece of code that reads other code and depending on what it reads either:

1. Tell us what is **wrong**
2. **Generate** one or more class files.



Plan

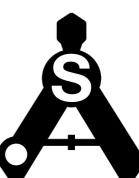
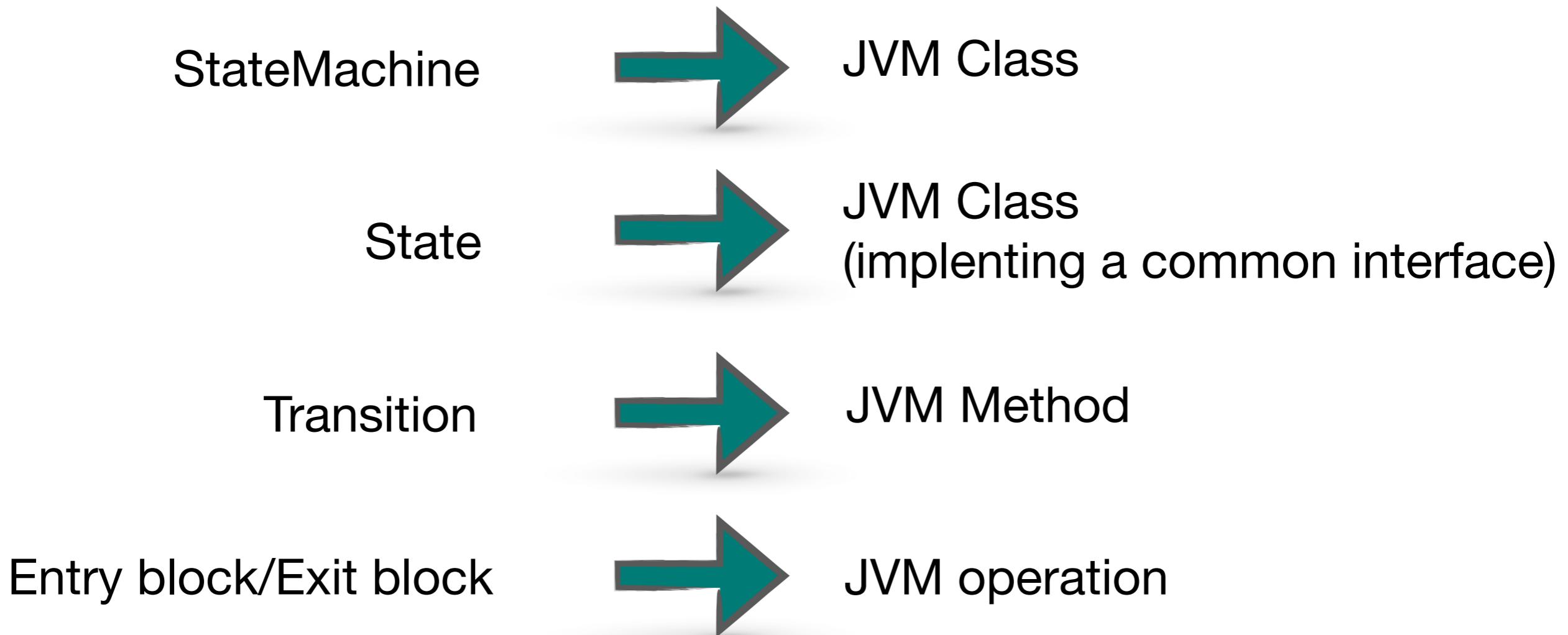
- Structuring the code
- Compile expressions
- Convert control structures



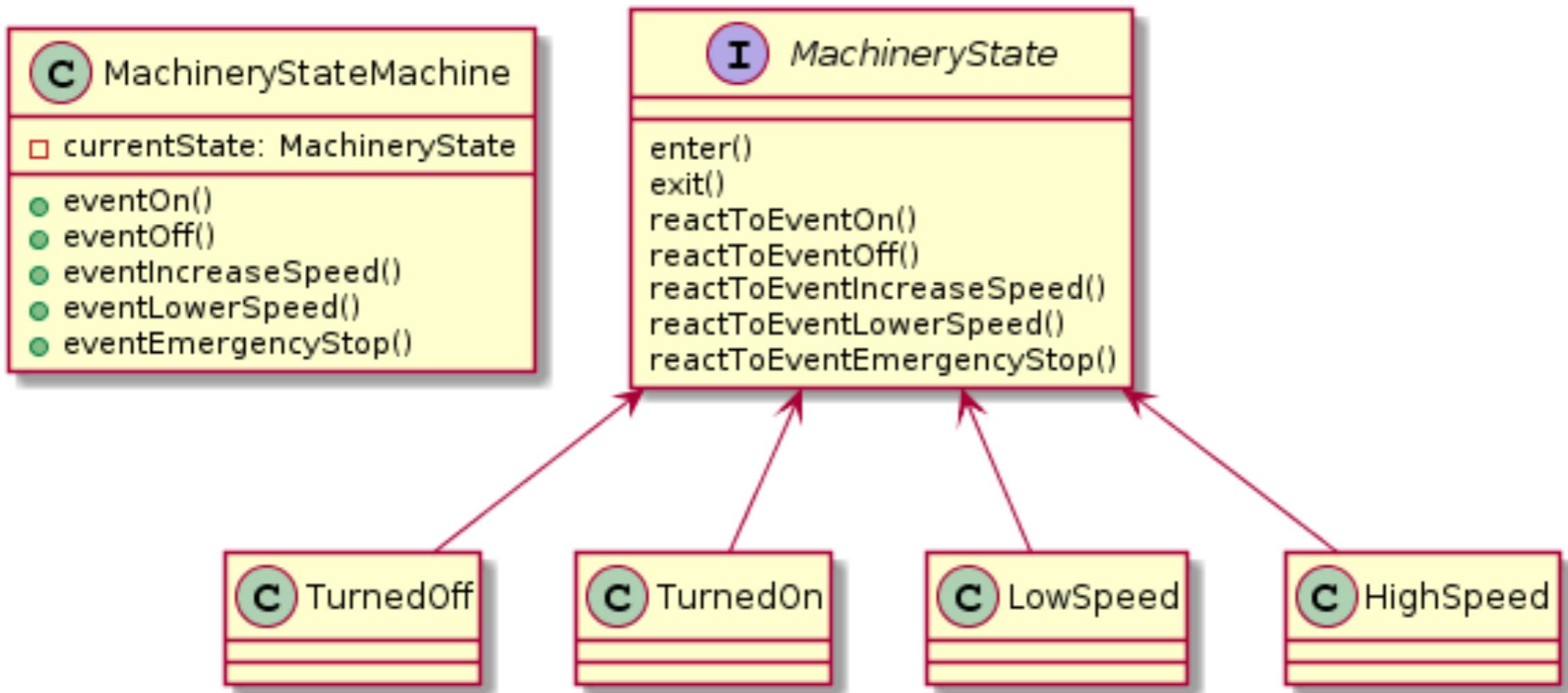
Structuring the code

The code (statements, expressions) stays in methods.

Methods live inside classes.



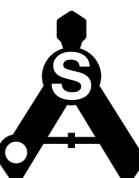
Structuring the code



Class files generation with ASM

```
val classWriter = ClassWriter(ClassWriter.COMPUTE_FRAMES  
                           or ClassWriter.COMPUTE_MAXS)  
...  
// put stuff in my class or interface  
...  
classWriter.visitEnd()  
val bytes = classWriter.toByteArray()  
// time to store those bytes into a file
```

Not that magic, eh?



Class files generation with ASM

```
val classWriter = ClassWriter(ClassWriter.COMPUTE_FRAMES  
                           or ClassWriter.COMPUTE_MAXS)  
...  
mv = classWriter.visitMethod(ACC_PUBLIC,  
                           "myMethodName", "()V", null, null);  
mv.visitCode();  
...  
// writing the method code  
...  
mv.visitMaxs(-1, -1);  
mv.visitEnd();
```

And finally in methods goes the code

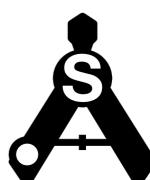


Compiling expressions

Expressions use operands from the stack and put results on the stack

Boolean	Bytecode
FALSE	ICONST_0
TRUE	ICONST_1

Byte/Short/Int	Bytecode
0	ICONST_0
1	ICONST_1
2	ICONST_2
3	ICONST_3
4	ICONST_4
5	ICONST_5
6+	BIPUSH/SIPUSH/SIPUSH x



Compiling expressions

Expressions use operands from the stack and put results on the stack

Reading variables	Bytecode	Writing variables	Bytecode
Type byte	ILOAD var_index	Type byte	ISTORE var_index
Type int	ILOAD var_index	Type int	ISTORE var_index
Type float	FLOAD var_index	Type float	FSTORE var_index
Type double	DLOAD var_index	Type double	DSTORE var_index
Type object	ALOAD var_index	Type object	ASTORE var_index

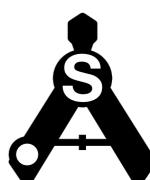


How to compile control structures

```
public void count(int n) {  
    int i = 0;  
    while (i < n) {  
        i = i + 1;  
    }  
}
```

Bytecode

```
// access flags 0x1  
public count(I)V  
L0  
ICONST_0  
ISTORE 2  
L1  
ILOAD 2  
ILOAD 1  
IF_ICMPGE L2  
L3  
ILOAD 2  
ICONST_1  
IADD  
ISTORE 2  
GOTO L1  
L2  
RETURN
```



How to compile control structures

```
public void count(int n) {  
    int i = 0;  
    while (i < n) {  
        i = i + 1;  
    }  
}
```

Index	Variable	Value
0	this	this
1	n	3
2	i	-

Stack

Bytecode

```
// access flags 0x1  
public count(I)V  
L0  
ICONST_0  
ISTORE 2  
L1  
ILOAD 2  
ILOAD 1  
IF_ICMPGE L2  
L3  
ILOAD 2  
ICONST_1  
IADD  
ISTORE 2  
GOTO L1  
L2  
RETURN
```



How to compile control structures

```
public void count(int n) {  
    int i = 0;  
    while (i < n) {  
        i = i + 1;  
    }  
}
```

Index	Variable	Value
0	this	this
1	n	3
2	i	-

Stack



Bytecode

```
// access flags 0x1  
public count(I)V  
L0  
    ICONST_0  
    ISTORE 2  
L1  
    ILOAD 2  
    ILOAD 1  
    IF_ICMPGE L2  
L3  
    ILOAD 2  
    ICONST_1  
    IADD  
    ISTORE 2  
    GOTO L1  
L2  
    RETURN
```



How to compile control structures

```
public void count(int n) {  
    int i = 0;  
    while (i < n) {  
        i = i + 1;  
    }  
}
```

Index	Variable	Value
0	this	this
1	n	3
2	i	0

Stack

Bytecode

```
// access flags 0x1  
public count(I)V  
L0  
ICONST_0  
ISTORE 2  
L1  
ILOAD 2  
ILOAD 1  
IF_ICMPGE L2  
L3  
ILOAD 2  
ICONST_1  
IADD  
ISTORE 2  
GOTO L1  
L2  
RETURN
```



How to compile control structures

```
public void count(int n) {  
    int i = 0;  
    while (i < n) {  
        i = i + 1;  
    }  
}
```

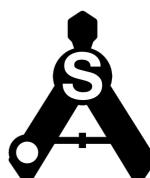
Index	Variable	Value
0	this	this
1	n	3
2	i	0

Stack



Bytecode

```
// access flags 0x1  
public count(I)V  
L0  
ICONST_0  
ISTORE 2  
L1  
ILOAD 2  
ILOAD 1  
IF_ICMPGE L2  
L3  
ILOAD 2  
ICONST_1  
IADD  
ISTORE 2  
GOTO L1  
L2  
RETURN
```

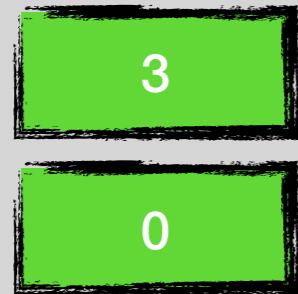


How to compile control structures

```
public void count(int n) {  
    int i = 0;  
    while (i < n) {  
        i = i + 1;  
    }  
}
```

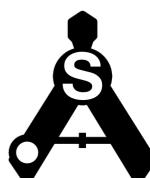
Index	Variable	Value
0	this	this
1	n	3
2	i	0

Stack



Bytecode

```
// access flags 0x1  
public count(I)V  
L0  
ICONST_0  
ISTORE 2  
L1  
ILOAD 2  
ILOAD 1  
IF_ICMPGE L2  
L3  
ILOAD 2  
ICONST_1  
IADD  
ISTORE 2  
GOTO L1  
L2  
RETURN
```



How to compile control structures

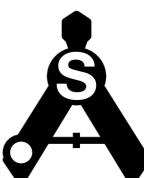
```
public void count(int n) {  
    int i = 0;  
    while (i < n) {  
        i = i + 1;  
    }  
}
```

Index	Variable	Value
0	this	this
1	n	3
2	i	0

Stack

Bytecode

```
// access flags 0x1  
public count(I)V  
L0  
ICONST_0  
ISTORE 2  
L1  
ILOAD 2  
ILOAD 1  
IF_ICMPGE L2 not jumping  
L3  
ILOAD 2  
ICONST_1  
IADD  
ISTORE 2  
GOTO L1  
L2  
RETURN
```



How to compile control structures

```
public void count(int n) {  
    int i = 0;  
    while (i < n) {  
        i = i + 1;  
    }  
}
```

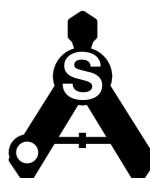
Index	Variable	Value
0	this	this
1	n	3
2	i	0

Stack



Bytecode

```
// access flags 0x1  
public count(I)V  
L0  
ICONST_0  
ISTORE 2  
L1  
ILOAD 2  
ILOAD 1  
IF_ICMPGE L2  
L3  
→ ILOAD 2  
ICONST_1  
IADD  
ISTORE 2  
GOTO L1  
L2  
RETURN
```

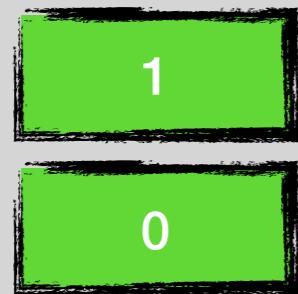


How to compile control structures

```
public void count(int n) {  
    int i = 0;  
    while (i < n) {  
        i = i + 1;  
    }  
}
```

Index	Variable	Value
0	this	this
1	n	3
2	i	0

Stack



Bytecode

```
// access flags 0x1  
public count(I)V  
L0  
ICONST_0  
ISTORE 2  
L1  
ILOAD 2  
ILOAD 1  
IF_ICMPGE L2  
L3  
ILOAD 2  
ICONST_1  
IADD  
ISTORE 2  
GOTO L1  
L2  
RETURN
```



How to compile control structures

```
public void count(int n) {  
    int i = 0;  
    while (i < n) {  
        i = i + 1;  
    }  
}
```

Index	Variable	Value
0	this	this
1	n	3
2	i	0

Stack



Bytecode

```
// access flags 0x1  
public count(I)V  
L0  
ICONST_0  
ISTORE 2  
L1  
ILOAD 2  
ILOAD 1  
IF_ICMPGE L2  
L3  
ILOAD 2  
ICONST_1  
IADD  
ISTORE 2  
GOTO L1  
L2  
RETURN
```



How to compile control structures

```
public void count(int n) {  
    int i = 0;  
    while (i < n) {  
        i = i + 1;  
    }  
}
```

Index	Variable	Value
0	this	this
1	n	3
2	i	1

Stack

Bytecode

```
// access flags 0x1  
public count(I)V  
L0  
ICONST_0  
ISTORE 2  
L1  
ILOAD 2  
ILOAD 1  
IF_ICMPGE L2  
L3  
ILOAD 2  
ICONST_1  
IADD  
→ ISTORE 2  
GOTO L1  
L2  
RETURN
```



How to compile control structures

ASM

```
mv.visitLabel(l0)
mv.visitInsn(ICONST_0)
mv.visitVarInsn(ISTORE, 2)

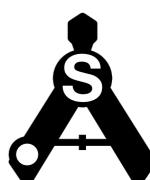
mv.visitLabel(l1)
mv.visitVarInsn(ILoad, 2)
mv.visitVarInsn(ILoad, 1)
mv.visitJumpInsn(IF_ICMPGE, l2)

mv.visitLabel(l3)
mv.visitVarInsn(ILoad, 2)
mv.visitInsn(ICONST_1)
mv.visitInsn(IADD)
mv.visitVarInsn(ISTORE, 2)
mv.visitJumpInsn(GOTO, l1)

mv.visitLabel(l2)
mv.visitInsn(RETURN)
```

Bytecode

```
// access flags 0x1
public count(I)V
L0
ICONST_0
ISTORE 2
L1
ILoad 2
ILoad 1
IF_ICMPGE L2
L3
ILoad 2
ICONST_1
IADD
ISTORE 2
GOTO L1
L2
RETURN
```

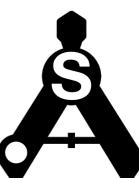


Recap

Building a language means:

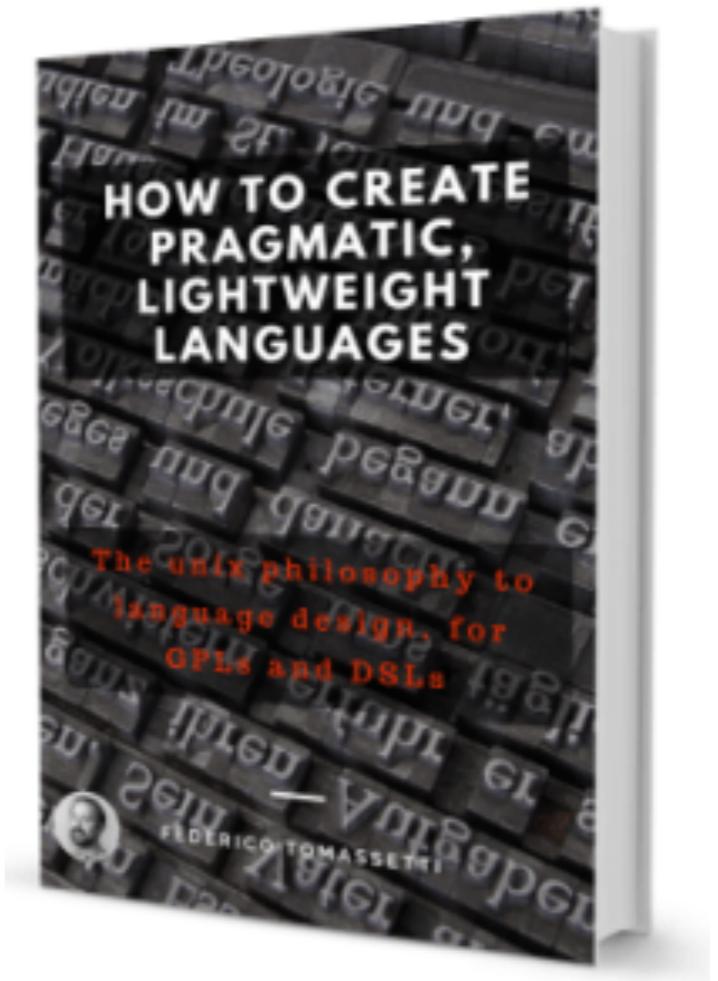
- 1) Parsing: use ANTLR
- 2) Validating: simple code navigating the model
- 3) Generating class files: use ASM

You need to familiarize with how a stack-based machine works and with the instructions available.



Where to find out more

- **Slides**
- **Free course** on building languages (8 articles)
- **Coupon** for my book on building languages



<https://tomassetti.me/berlin>