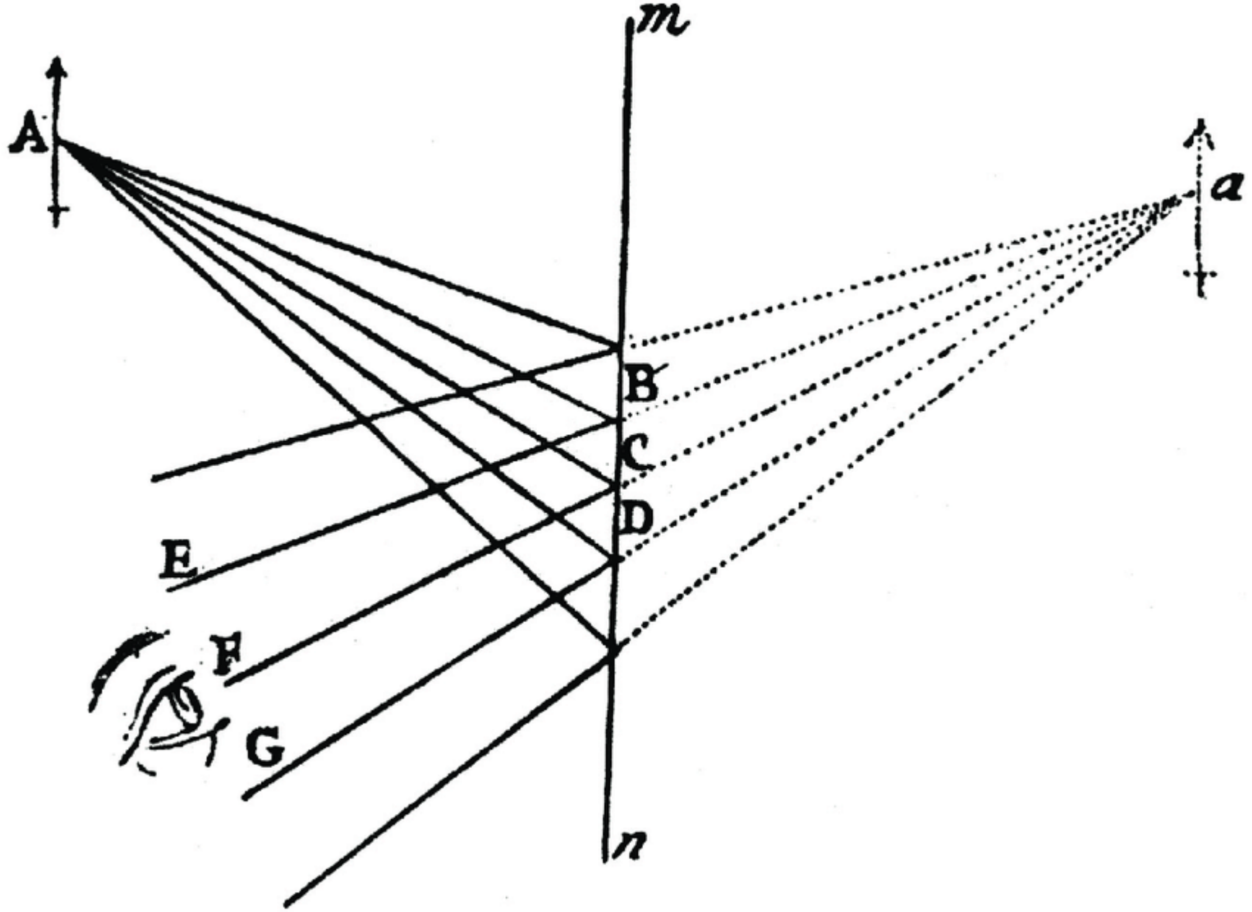
# Code Reflection





Paul Sandoz


**JVM Language Summit**
**August 7–9, 2023**

# Overview

Motivation

Code reflection | noitcelfer edoC

Plan

# Motivation — broaden Java in nontraditional domains

- It should be easy for developers to write and support Java programs that *represent*:
  - GPU kernels, and kernel call graphs
  - Differentiable programs
  - Machine learning models
  - SQL statements (or anything C# LINQ can do)
  - Parallel graph programs (Parallel Graph AnalytiX)
  - Probabilistic programs (Vate: Runtime Adaptable Probabilistic Programming in Java)
  - Secure programs (leveraging CPU secure enclaves)
  - Lane-wise/element-wise vectorizable programs
  - C programs bound to Panama FFM upcalls from native code

# Motivation

- Developers should not have to
  - Embed snippets of non-Java code in text blocks, or string templates
  - Write tedious Java code that builds up data structures to represent their program
  - Use non-standard/internal APIs to access and analyze their program in unsuitable formats that contain too much or too little information
- They should be able to write novel programs *combining* APIs with Java language features
  - Rather than using APIs that poorly emulate language features
- With today's Java platform this is hard to do — let's fix that

# Method to be differentiated — tedious code

```java
// Developers should not have to write this code, the compiler should do that
var fModel  = func("f", methodType(double.class, double.class, double.class))
        .body(entry -> {
            var x = entry.parameters().get(0);
            var y = entry.parameters().get(1);

            var r = entry.op(mul(
                    entry.op(mul(
                            x,
                            entry.op(add(
                                    entry.op(neg(
                                            entry.op(call(MATH_SIN,
                                                    entry.op(mul(
                                                            x,
                                                            y))))))),
                                    y)))),
                    entry.op(constant(DOUBLE, 4.0))));
            entry.op(_return(r));
        });
```

```java
// Developers should write ordinary Java code
static double f(double x, double y) {
    return x * (-Math.sin(x * y) + y) * 4.0d;
}
```

# Method to be differentiated — difficult access and format

```
// Developers should write ordinary Java code
static double f(double x, double y) {
    return x * (-Math.sin(x * y) + y) * 4.0d;
}


// Library developers should not have to use non-standard access to code
// in a format unsuitable for analysis

// Find class file bytes for class file with method f
ClassLoader l = ...
byte[] classbytes = l.getResourceAsStream("....class")
        .readAllBytes();

// Parse the class file bytes and obtain the (byte) code model for method f
CodeModel fMethodModel = Classfile.of().parse(classbytes).methods().stream()
        .filter(methodModel -> methodModel.methodName().equalsString("f"))
        .flatMap(methodModel -> methodModel.code().stream())
        .findFirst().orElseThrow();
List<CodeElement> fCodeModel = fModel.elementList();

// Transform the (byte) code elements into a suitable format for analysis
// - Manage the stack
// - Reconstruct structure and type information
// - Reverse engineer the source compiler's translation strategy
```

# Observation — domain specific programming models

- Those Java programs are domain specific
  - Not all Java programs can or should execute on a GPU
  - Not all Java programs are differentiable
- In many cases program meaning may differ from that specified by the Java platform
  - The byte code is never intended to be executed

# How to grow a language, not!

- Why don’t we “just” enhance the Java platform to support these programming models?
- This is a terrible way to grow the Java language
  - Complicated and costly process to update the Java specifications and implement
  - Does not scale as new programming models are requested
  - Does not compose — models will surely conflict

# Observation — Java programs transforming Java programs

- A domain specific programming model can be implemented as a Java program that
  - **A**ccesses the code of the domain specific Java program
  - **A**nalyzes that program; and then
  - **T**ransforms it to a new program
- The transformation need not preserve Java program meaning
  - The new program might not be a Java program
- The transforming program could be “just” an ordinary Java library
  - We don’t need to add new programing domains to Java’s programming model

# Limited **P**rogram **A**ccess, **A**nalysis, and **T**ransformation

- Today's Java platform features supporting **PAAT** are limited and hard to use
- There are two choices, each available at two distinct phases in the life cycle of the program
    1. At source compile time, with access to the unspecified **A**bstract **S**yntax **T**ree (**AST**), derived from the specified grammar, and produced by the Java compiler
    2. At run time, with access to the specified bytecode of the class files produced by the Java compiler
- Both are insufficient to meet the needs of PAAT

# Limited PAAT

- Neither the AST nor bytecode is fully accessible using public Java APIs
  - The compile time mirror API and run time reflection API only reflect the "surface" details
- Neither provide a suitable program model for PAAT
  - AST is designed to be processed by the source compiler
    - Containing surface syntax details and grammatical idiosyncrasies
    - Specific to each Java compiler implementation
  - Bytecode is designed for "shipping" to and execution by the Java run time
    - Types are erased, program structure is stripped away
    - Arrangement is specific to the Java compiler's translation strategy
- Operating on the program at both compile time and run time requires the use of two APIs and two models

# Code reflection — *deepen* and *broaden* Java reflection

1. Modeling Java programs as *code models*
   - Suitable for access, analysis, and transformation
   - Preserving program structure and type information
2. Enhancements to Java reflection
   - Identifying areas of Java source code to reflect over and give access to as code models at compile time and run time
   - e.g., code of method bodies and lambda bodies
3. API to build, analyze, and transform code models
   - For use at compile time and run time
   - e.g, domain-specific errors can be reported at compile time

# Example — Automatic differentiation

```java
// Identify that f has a code model
@CodeReflection
static double f(double x, double y) {
    return x * (-Math.sin(x * y) + y) * 4.0d;
}

...

// Reflect on method f using existing Reflection API
Method mf = ClassWithF.class.getDeclaredMethod("f", double.class, double.class);
// Get the code model of f's body using new Reflection API
var cmf = mf.getCodeModel().orElseThrow();

// Differentiate f using an AD library
// Code models in, code models out -- code model *composition*
var d_cmf = AutoDiff.differentiate(cmf);

// Compile the differentiated code method to a method handle
MethodHandle mh_d_cmf = d_cmf.compile();

// Execute to obtain the gradient at a particular point
double a = ...
double b = ...
double[] gv = (double[]) mh_d_cmf.invoke(a, b);
```

# Example — Automatic differentiation

```
// Hide the code model

// Pass a method reference to f
GradientFunction gf = AutoDiff.differentiate(ClassWithF::f);
double[] gv = gf.apply(a, b);

...

// Pass lambda expression, passes code model for lambda body
GradientFunction gf = AutoDiff.differentiate((double x, double y) ->
        x * (-Math.sin(x * y) + y) * 4.0d
);
double[] gv = gf.apply(a, b);

...

@FunctionalInterface
interface GradientFunction {
    double[] apply(double... args);
}
```

# Example — What is the Java compiler and run time doing?

- At source compile time the Java compiler
  - Transforms the AST of f to a code model (using the API to build)
  - Serializes the code model (using the API to traverse)
  - Stores the serialized code model in the class file ClassWithF.class
- Accessing the code model of f at run time
  - Checks that the caller has permission to access the code model
  - Loads the serialized code model from the class file ClassWithF.class
  - Deserializes the code model (using the API to build)
  - Returns the code model to the caller

# Example — **P**arallel **G**raph **A**nalytiX (**PGX**) Algorithm

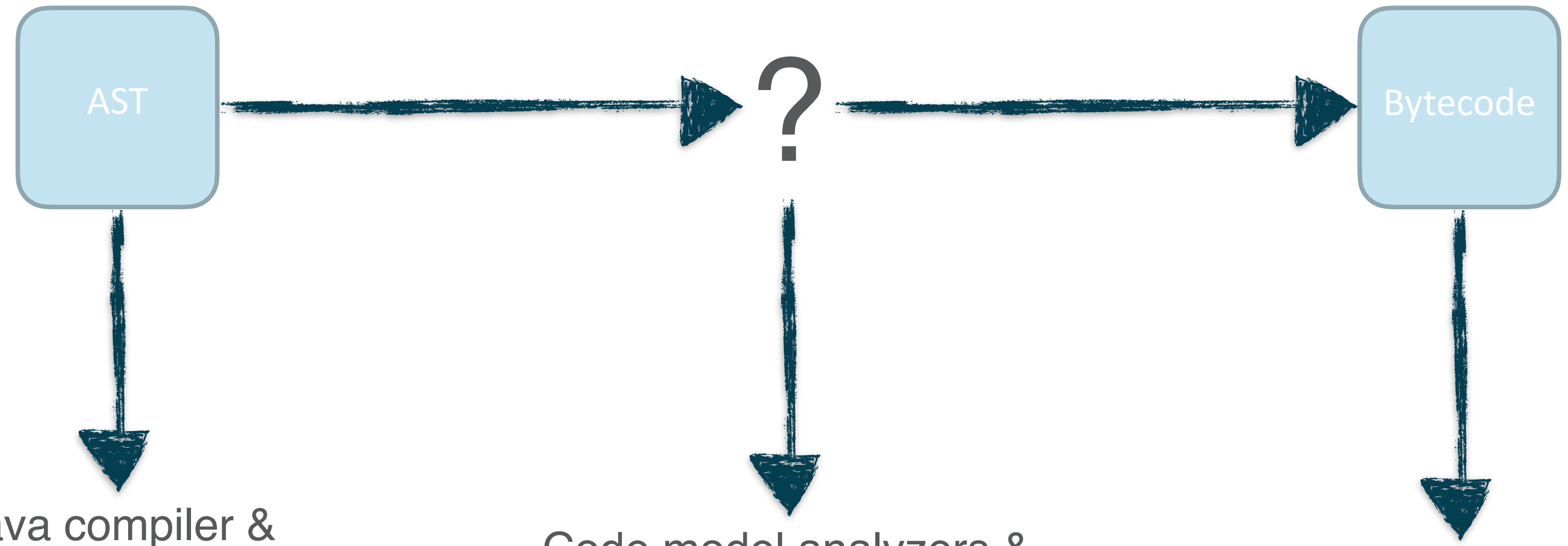
```java
@CodeReflection
public void pagerank(PgxGraph g, double tol, double damp,
                     @Out VertexProperty<Double> rank) {
    Scalar<Double> diff = Scalar.create();
    double n = g.getNumVertices();

    rank.setAll(1 / n);
    do {
        diff.set(0d);

        g.getVertices().forEach(v -> {
            double inSum = v.getInNeighbors().sum(w -> rank.get(w) / w.getOutDegree());
            double val = (1 - damp) / n + damp * inSum;
            diff.reduceAdd(Math.abs(val - rank.get(v)));
            rank.setDeferred(v, val);
        });
    } while (diff.get() > tol);
}
```

# Example — Parallel Graph AnalytiX (PGX)

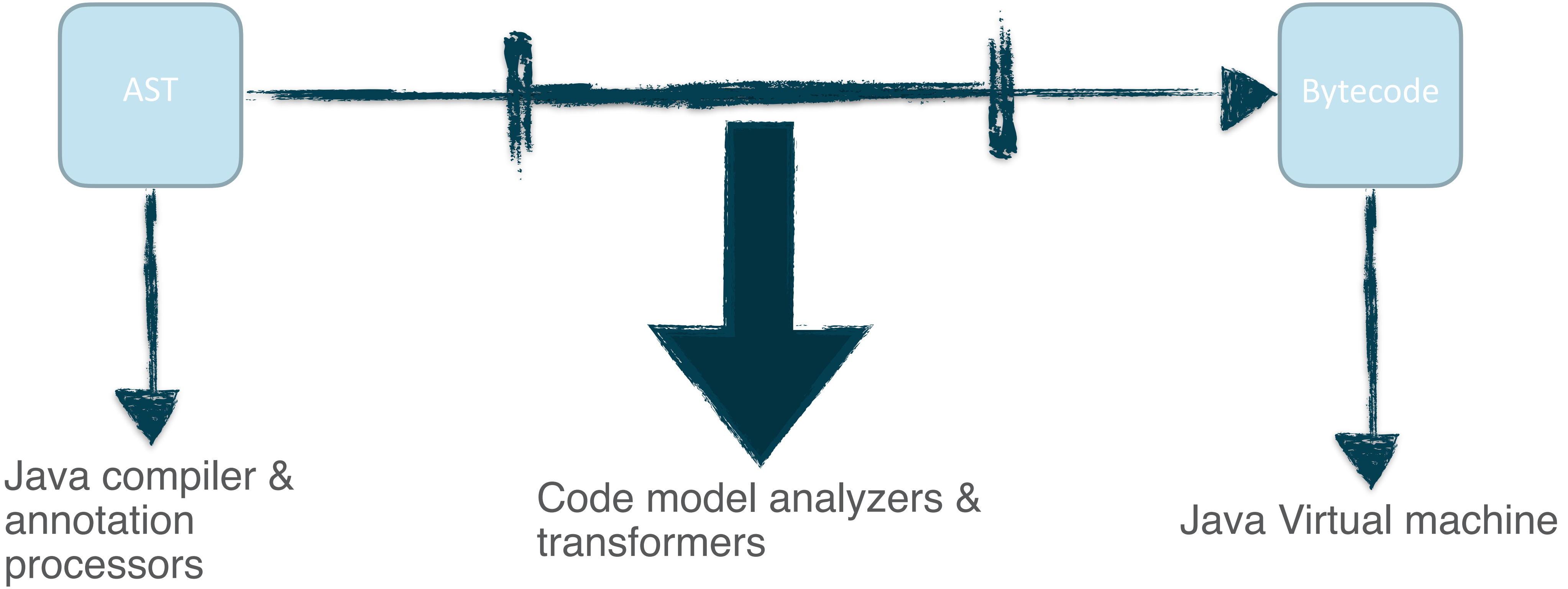
- The PGX Algorithm compiler is an OpenJDK compiler plugin operating on the AST
- We have implemented a prototype PGX compiler that operates on the code model — which is easier to develop, maintain, and aligns with the Java platform
- The PGX runtime can then be enhanced to use the new compiler

```java
var data = ...
try (PgxSession session = Pgx.createSession("pgx-algorithm-session")) {
    // Transform the PGX Algorithm to executable Java code
    CompiledProgram program = session.compileProgram(oracle.pgx.PgxAlgorithm.Pagerank::pagerank);

    // Create the input graph
    PgxGraph graph = session.readGraphWithProperties(createGraphConfig(data));
    VertexProperty<Object, Object> rank = graph.createVertexProperty(PropertyType.DOUBLE);

    // Run the compiled program
    program.run(graph, TOLERANCE, DAMPING, rank);

    // Process rank result
    ...
}
```

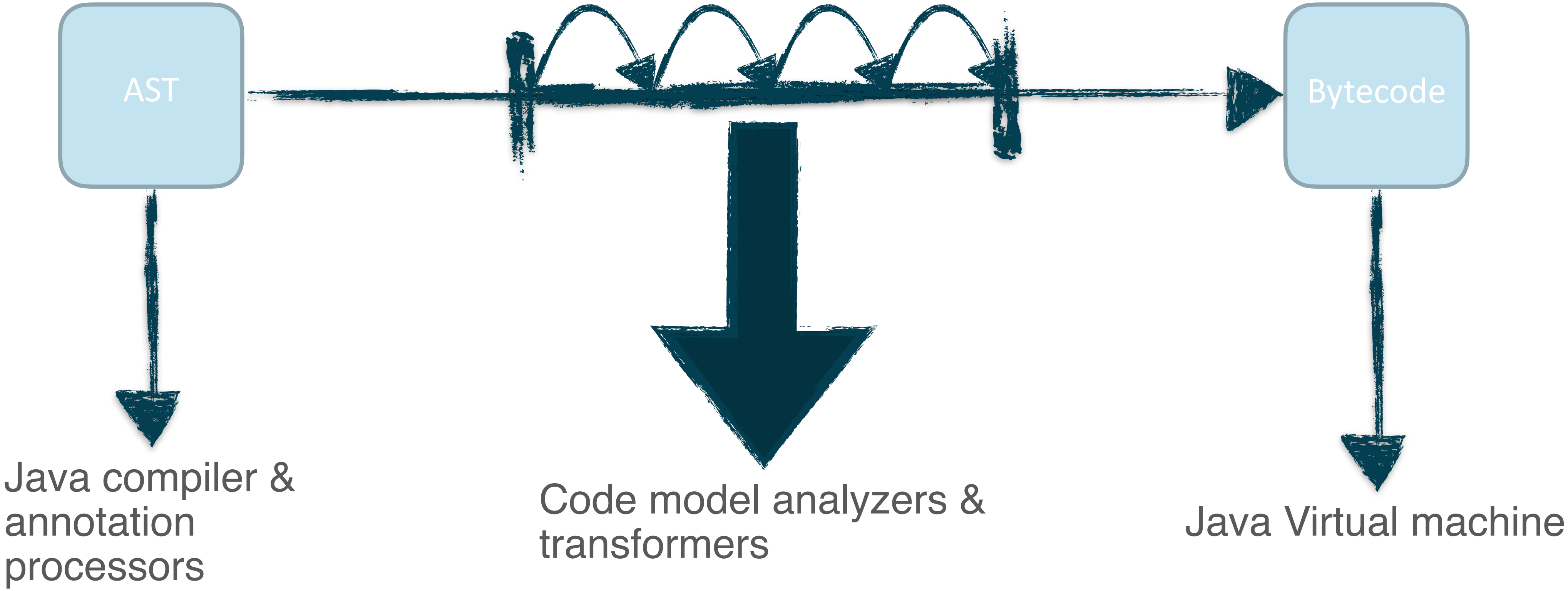
# Modeling Java programs — spectrum of possibilities



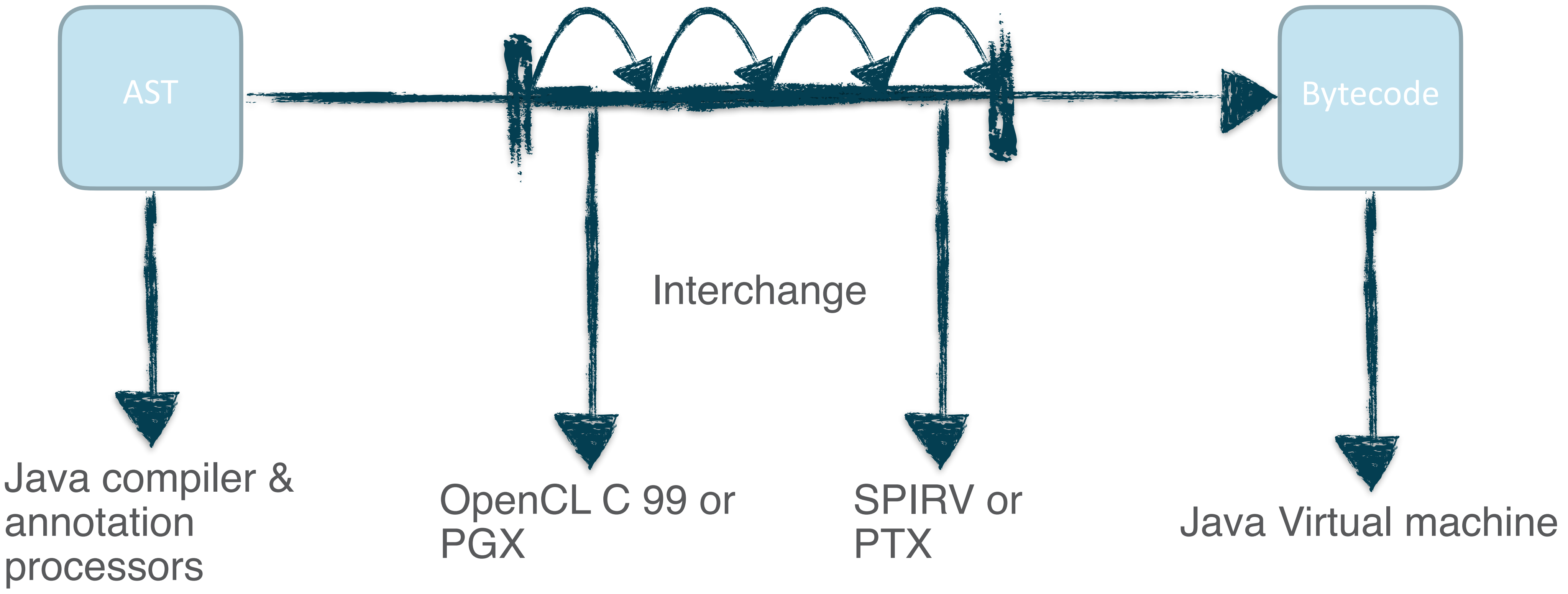
# Modeling Java programs — an interval



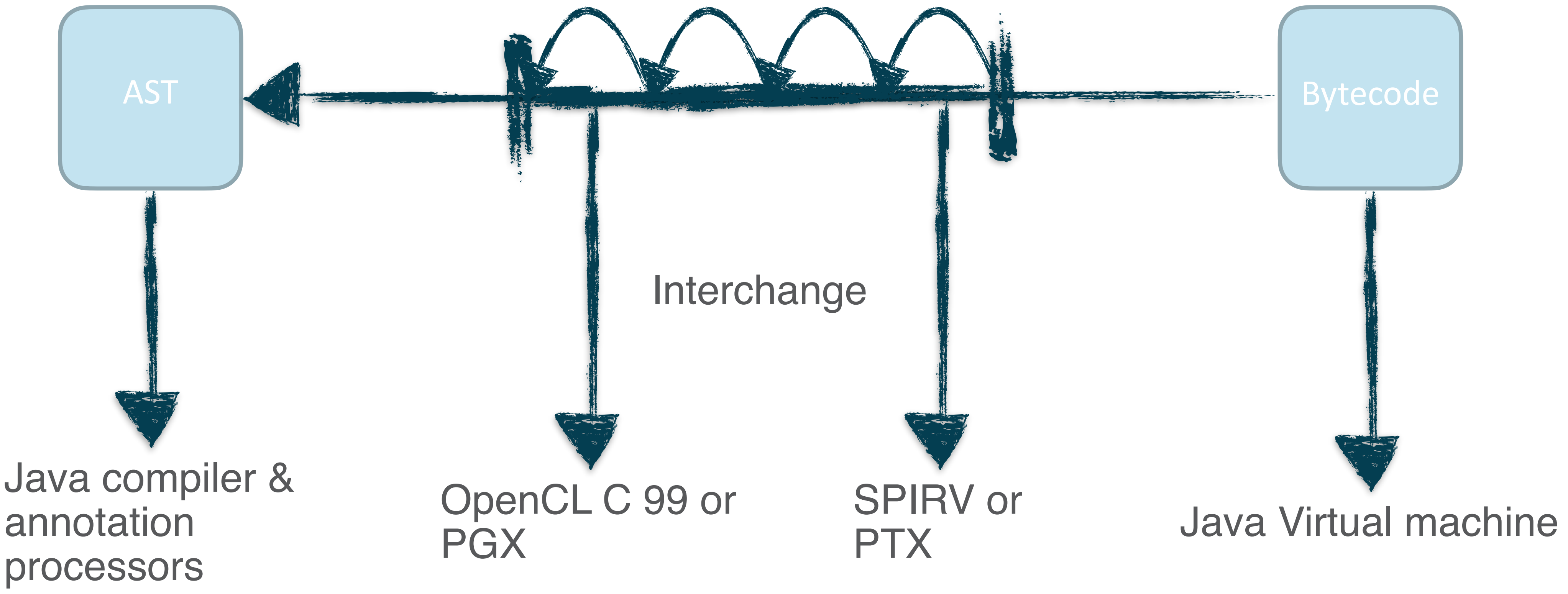
# Modeling Java programs — progressive lowering



# Modeling Java programs — progressive lowering



# Modeling Java programs — progressively harder lifting



# Code meta-model

- We need to devise a code *meta-model* that is flexible to model a broad set of Java programs as code models
  - At a high level closer to the AST; and be transformed (using the API) to
  - A lower level closer to the bytecode
  - Where program meaning is preserved
- Some programming domains are more suited to higher levels, where as others to lower levels
  - The same modeling capabilities and API should apply
- The meta-model should be comprehensible to many Java developers

# Code meta-model — op → body* → block+ → op+

- Drawing inspiration from MLIR we can design a meta-model with the following properties
  - Decomposition of a program into operations, bodies, and (basic) blocks
  - An operation is comprised of bodies; a body is comprised of blocks, that may form the vertices of a control-flow graph; and a block is comprised of operations
  - An operation produces a result and a block has parameters (equiv. to phi nodes), both values in **S**tatic **S**ingle **A**ssignment (**SSA**) form; values have types
  - An operation has operands (value use), a terminal operation may reference successor blocks with arguments (value use)
- A code model is a shallow tree structure
  - Control flow graphs and data dependency graphs are emergent properties
- This meta-model is extremely flexible and is capable of modeling many Java language constructs at high and lower levels

# Code meta-model — operations

- Operations specify program behavior — we define two sets
  - A set of **core** operations, that model a broad set of Java programs
  - A set of **auxiliary** operations, that model certain Java language constructs
- Auxiliary operations model higher level Java language constructs
  - e.g., loops, try statements, switch expressions, patterns, conditionals
  - With fewer constraints than the modeled language constructs specified by the JLS
  - Each auxiliary operation can lower itself (using the API) into a substructure of core operations
- A code model supplied by the platform is comprised of auxiliary and core operations
  - A code model is never supplied for invalid Java source code, since it will fail to compile
  - It may be transformed into a model comprised only of core operations, while preserving program meaning

# Example — serialized code model as text*

- Java method to differentiate

```java
@CodeReflection
static double f(double x, double y) {
    return x * (-Math.sin(x * y) + y) * 4.0d;
}
```

- Serialized code model

```
func @"f" (%0 : double, %1 : double)double -> {
    %2 : Var<double> = var %0 @"x";
    %3 : Var<double> = var %1 @"y";
    %4 : double = var.load %2;
    %5 : double = var.load %2;
    %6 : double = var.load %3;
    %7 : double = mul %5 %6;
    %8 : double = call %7 @"java.lang.Math::sin(double)double";
    %9 : double = neg %8;
    %10 : double = var.load %3;
    %11 : double = add %9 %10;
    %12 : double = mul %4 %11;
    %13 : double = constant @"4.0";
    %14 : double = mul %12 %13;
    return %14;
};
```

*Lets not get too distracted by the syntax!

# Example — lower with pure SSA transform

```
func @"f" (%0 : double, %1 : double)double -> {
    %7 : double = mul %0 %1;
    %8 : double = call %7 @"java.lang.Math::sin(double)double";
    %9 : double = neg %8;
    %11 : double = add %9 %1;
    %12 : double = mul %0 %11;
    %13 : double = constant @"4.0";
    %14 : double = mul %12 %13;
    return %14;
};
```

# Example — translate to bytecode operations*

```
func @"f" (%0 : double, %1 : double)double -> {
    Tload @index=0 @type="D";
    Tload @index=2 @type="D";
    Tmul @type="D";
    invoke @kind="STATIC" @desc="java.lang.Math::sin(double)double";
    Tneg @type="D";
    Tload @index=2 @type="D";
    Tadd @type="D";
    Tstore @index=2 @type="D";
    Tload @index=0 @type="D";
    Tload @index=2 @type="D";
    Tmul @type="D";
    ldc @type="double" @value="4.0";
    Tmul @type="D";
    Treturn @type="D";
};
```

*Speculative modeling of other domains

# Example — translate to bytecode

```
- class name: f
  version: 66.0
  flags: [PUBLIC]
  superclass: java/lang/Object
  interfaces: []
  attributes: []
  fields:
  methods:
    - method name: f
      flags: [PUBLIC, STATIC]
      method type: (DD)D
      attributes: [Code]
      code:
          max stack: 4
          max locals: 4
          attributes: []
          //stack map frame @0: {locals: [double, double2, double, double2], stack: []}
          0: {opcode: DLOAD_0, slot: 0}
          1: {opcode: DLOAD_2, slot: 2}
          2: {opcode: DMUL}
          3: {opcode: INVOKESTATIC, owner: java/lang/Math, method name: sin, method type: (D)D}
          6: {opcode: DNEG}
          7: {opcode: DLOAD_2, slot: 2}
          8: {opcode: DADD}
          9: {opcode: DSTORE_2, slot: 2}
          10: {opcode: DLOAD_0, slot: 0}
          11: {opcode: DLOAD_2, slot: 2}
          12: {opcode: DMUL}
          13: {opcode: LDC2_W, constant value: 4.0}
          16: {opcode: DMUL}
          17: {opcode: DRETURN}
```

# Example — modeling conditional operator ? :

```
@CodeReflection
static String f(boolean v, String a, String b) {
    return v ? a : b;
}
```

- The Java language specification states (in section 15.25)

  “The conditional operator has three operand expressions. ? appears between the first and second expressions, and : appears between the second and third expressions."

- We can model this as an operation comprised of three bodies, each comprised of operations modeling the operand expressions (in order)

# Example — serialized code model as text

```
func @"f" (%0 : boolean, %1 : java.lang.String, %2 : java.lang.String)java.lang.String -> {
    %3 : Var<boolean> = var %0 @"v";
    %4 : Var<java.lang.String> = var %1 @"a";
    %5 : Var<java.lang.String> = var %2 @"b";
    %6 : java.lang.String = java.cexpression
        ^cond()boolean -> {
            %7 : boolean = var.load %3;
            yield %7;
        }
        ^truepart()java.lang.String -> {
            %8 : java.lang.String = var.load %4;
            yield %8;
        }
        ^falsepart()java.lang.String -> {
            %9 : java.lang.String = var.load %5;
            yield %9;
        };
    return %6;
};
```

# Example — lower with auxiliary operation transform

```
func @"f" (%0 : boolean, %1 : java.lang.String, %2 : java.lang.String)java.lang.String -> {
    %3 : Var<boolean> = var %0 @"v";
    %4 : Var<java.lang.String> = var %1 @"a";
    %5 : Var<java.lang.String> = var %2 @"b";
    %7 : boolean = var.load %3;
    cond_br %7 ^then ^else;

  ^then:
    %8 : java.lang.String = var.load %4;
    br ^exit(%8);

  ^else:
    %9 : java.lang.String = var.load %5;
    br ^exit(%9);

  ^exit(%3_1 : java.lang.String):
    return %3_1;
};
```

# Example — lower with pure SSA transform

```
func @"f" (%0 : boolean, %1 : java.lang.String, %2 : java.lang.String)java.lang.String -> {
    cond_br %0 ^then ^else;

  ^then:
    br ^exit(%1);

  ^else:
    br ^exit(%2);

  ^exit(%3 : java.lang.String):
    return %3;
};
```

# Example — translate to bytecode operations

```
func @"f" (%0 : boolean, %1 : java.lang.String, %2 : java.lang.String)java.lang.String -> {
    Tload @index=0 @type="I";
    ifC ^br_T ^br_F @cond="EQ";

  ^br_T:
    goto ^then;

  ^then:
    Tload @index=1 @type="A";
    Tstore @index=0 @type="A";
    goto ^exit;

  ^br_F:
    goto ^else;

  ^else:
    Tload @index=2 @type="A";
    Tstore @index=0 @type="A";
    goto ^exit;

  ^exit:
    Tload @index=0 @type="A";
    Treturn @type="A";
};
```

# Example — translate to bytecode

```
- class name: f
  version: 66.0
  flags: [PUBLIC]
  superclass: java/lang/Object
  interfaces: []
  attributes: []
  fields:
  methods:
    - method name: f
      flags: [PUBLIC, STATIC]
      method type: (ZLjava/lang/String;Ljava/lang/String;)Ljava/lang/String;
      attributes: [Code]
      code:
          max stack: 1
          max locals: 3
          attributes: [StackMapTable]
          stack map frames:
              9: {locals: [int, java/lang/String, java/lang/String], stack: []}
              11: {locals: [java/lang/String, java/lang/String, java/lang/String], stack: []}
          //stack map frame @0: {locals: [int, java/lang/String, java/lang/String], stack: []}
          0: {opcode: ILOAD_0, slot: 0}
          1: {opcode: IFEQ, target: 9}
          4: {opcode: ALOAD_1, slot: 1}
          5: {opcode: ASTORE_0, slot: 0}
          6: {opcode: GOTO, target: 11}
          //stack map frame @9: {locals: [int, java/lang/String, java/lang/String], stack: []}
          9: {opcode: ALOAD_2, slot: 2}
          10: {opcode: ASTORE_0, slot: 0}
          //stack map frame @11: {locals: [java/lang/String, java/lang/String, java/lang/String], stack: []}
          11: {opcode: ALOAD_0, slot: 0}
          12: {opcode: ARETURN}
```

# Enhancements to Java reflection

- Identify parts of a program to be deeply and broadly reflected over
- Grant access to those parts as code models at compile time and run time
  - With appropriate access control restrictions
- At a minimum identify individual methods and lambda expressions
  - e.g., annotate methods, target type lambda expressions
- Perhaps as a convenience broaden the scope to that of all methods of class and its nest

# Identifying lambda expressions

```
int c = 42;
IntUnaryOperator f = (Quotable & IntUnaryOperator) z -> {
    return z + c;
};

Quotable quotableF = (Quotable) f;
Quoted quotedF = quotableF.quoted();
quotedF.op().writeTo(System.out);
System.out.println(quotedF.capturedValues());
```

- Target lambda expressions as being *quotable*
  - Similar to them being serializable
- An instance of a quotable functional interface encapsulates the code model of the lambda expression and any captured values
- We now have the means to experiment with C# LINQ-like APIs

# API to *build*, *analyze*, and transform code models

- A run time instance of a code model should have the following desirable properties
  - Immutable
  - Easily traversed down and up its tree structure
  - Dominance relationships are easily queryable
  - Values report their users, from which data dependency graphs can be constructed
  - Blocks in a body are sorted topologically in reverse postorder
- A code model is built using a builder
  - During building a code model is in a *larval* state, when building is complete it is frozen, and thereafter is unmodifiable
- Code models can be serialized to and deserialized from text
  - Primarily for debugging and testing, but also very convenient for storage and transfer

# API to build, analyze, and *transform* code models

- Transformation is an *emergent* property of traversal combined with building
  - Inspired by the transformation pattern supported by the Classfile API
- We can traverse an input code model and flat map its contents into a builder of the output code model
  - Flat mapping supports a zero to many transformation enabling removal, copying, or replacement
- Alternative forms of transformation can perform their own traversal and building

# API to build, analyze, and *transform* code models

- Each auxiliary operation implements its own transformation
- We can implement the lowering of a code model by *composing* the transforms of all the auxiliary operations in the model
  - Thereby lowering the code model in a *single* transformation pass
  - (This includes lowering loops with labeled break and continue statements and nested try statements)

```
var lf = f.transform((blockBuilder, op) -> {
    if (op instanceof Op.Lowerable lop) {
        return lop.lower(blockBuilder);
    } else {
        blockBuilder.apply(op);
        return blockBuilder;
    }
});
```

# Known unknowns and risks

- How many Java language constructs do we need to directly model?
  - Do we need to model class declarations?
- Can the code model design evolve with evolution of the language?
  - Skating to where the language puck will be on an N-dimensional ice rink
  - Core set of operations evolves slowly (like byte code)
  - Auxiliary set of operations evolves faster — lower to core if unrecognized
- Increases the incremental cost of adding new language features
- How performant do we need to be when building and transforming?
  - Some costs may be offset by shifting when transformations are performed

# Plan* — some time this year

- Submit an OpenJDK project proposal
  - Project name TBD
  - Scope will also include exploration of GPU programming domains
- Open source the prototype code reflection JDK
  - Enable and further experimentation
  - Needs tidying up first, but generally in good shape

*The plan is the plan until the plan changes