LLJava Live at the Loop

A Case for Heteroiconic Staged Meta-Programming

Baltasar Trancón y Widemann
Nordakademie
Elmshorn, DE
baltasar@trancon.de

Markus Lepper
semantics GmbH
Berlin, DE

ABSTRACT

This paper investigates the use of staged meta-programming techniques for the transparent acceleration of embedded domain-specific languages on the JAVA platform. LLJAVA-LIVE, the staged API of the low-level JVM language LLJAVA, can be used to complement an interpreted EDSL with orthogonal and extensible compilation facilities. Compiled JVM bytecode becomes available immediately as an extension of the running host program. The approach is illustrated with a didactic structured imperative programming language.

CCS CONCEPTS

• **Software and its engineering → Domain specific languages**; *Software prototyping*; *Extensible languages*; **Dynamic compilers**.

KEYWORDS

embedded domain-specific languages, staged meta-programming, compiler construction, JVM, class loading

ACM Reference Format:
Baltasar Trancón y Widemann and Markus Lepper. 2021. LLJava Live at the Loop: A Case for Heteroiconic Staged Meta-Programming. In *Proceedings of the 18th International Conference on Managed Programming Languages & Runtimes (MPLR '21).* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Embedded domain-specific languages (EDSLs) can greatly enhance the expressivity of a general-purpose programming language and platform. [8] Like all languages, EDSLs can be either interpreted or compiled. The technique of staged meta-programming allows constructing a compiler from an interpreter in a rational, systematic and efficient way. The JAVA language, unlike typical host languages for staged meta-programming, does not have a homoiconic notation of code-as-data. We have equipped the low-level JVM language LLJAVA with a staged meta-programming frontend API, LLJAVA-LIVE. This paper illustrates and evaluates the style of EDSL compiler construction enabled by that approach by application to a didactic but nontrivial embedded programming language.

MPLR '21, September 29–30, 2021, Münster, DE
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-XXXX-X…$0.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

The outline of the paper is as follows: the remainder of this section reviews the involved technologies and paradigms. In the main part, section 2 introduces the example language WHILST. Sections 3 and 4 discuss the interpreter and its metamorphosis into a compiler, respectively. In the conclusion part, the following sections discuss language extensibility and the empirical evaluation of compilation, and wrap up the lessons learned.

The reader is assumed to be reasonably familiar with both the JAVA language and its execution environment, the Java Virtual Machine (JVM).

The main contributions of the present paper are:

- On the didactical side, the design of the extensible imperative programming language WHILST as a showcase for EDSL operationalization, and by means of that example,
- on the methodological side, the characterization of the particular and novel style of staged meta-programming enabled by the LLJAVA-LIVE API, which we name *heteroiconic staged meta-programming*.

1.1 Meta-Programming

Meta-programming as a discipline is concerned with the design and construction of software that operates on other *software as data*, the former being called the *meta-programs* and the latter the *object programs*, respectively. Classical examples for meta-programs are of course found in the area of programming tools, namely interpreters (as semi-meta-programming, where objects programs serve as input only), or assemblers, compilers and linkers (as full meta-programming, where object programs are also output, and thus give rise to the term *object code*).

1.1.1 Staged Meta-Programming. A meta-programming approach is called *staged*, if it is integrated tightly enough into the execution model of a programming language, such that the produced object programs can be accessed as *code* as well as data [18]. Such object programs can extend the codebase of the running meta-program, and be run as subsequent stages of program evolution.

The expressive power of staged meta-programming is particularly enhanced by the fact that object programs-as-data may reference other data of the running meta-program directly, a feature known as *cross-stage persistence* [18]. This is a marked contrast to classical meta-programming where, e.g., the data in internal memory of a compiler is not shared by the compiled application program.

1.1.2 Homoiconic Meta-Programming. Many popular approaches to meta-programming have the intriguing property that meta-program and object program share the same notation. Such a notation is called *homoiconic* [10]. Obviously, if notation applies to both levels, the boundaries must be marked explicitly, which distinguishes this style of meta-programming from implicitly leveled styles, e.g., automatic partial evaluation [9].

At the very least, a homoiconic meta-programming notation requires two operators to switch from meta to object and vice versa. Staged variants are distinguished by a third operator to execute an object program. The former two occur in various typographical variants throughout history, e.g., ‘ and , in Lisp quasiquotation [1], <_> and ~ in MetaML [18], or ''' _ ''' and «_» in Xtend templates [5]. (Apparently there is an evolutionary trend to become fancier with time.)

1.1.3 Homoiconic Staged Meta-Programming and Compilation. The most evidently useful application of staged meta-programming is *runtime program specialization*, where the inputs of a subprogram differ in binding time: Some of these may be more static, i.e., available earlier in the course of the program run, or changing values less frequently, than others. A first stage can take advantage of the contained information, and arrange the code fragments that depend on the more variable inputs accordingly. Obviously the invested effort pays off only if the specialized variant is both more efficient in fact, and used often enough to amortize.

Such techniques have been well studied as optimizations in interpreters and compilers [9]. For example, consider *loop unrolling*, where a loop with dynamically conditional control flow can be turned into a flat sequence of copies of the loop body, if the number of iterations is determined already. In the context of staged-meta programming, this involves concatenating object-level copies of the loop body, driven by a meta-level counting loop.

The connection between monolithic versus staged programs on the one hand, and interpretation versus compilation on the other, goes deeper than that, however. A canonical textbook exercise in staged meta-programming states simply:

> *"Define a language. Write an interpreter for the language. Stage the interpreter to construct a compiler."* [17]

This piece of advice shall be followed in the main part of the present paper, even though some part of the magic hinges on homoiconicity, and is hence lost in translation to our deviant approach.

1.2 The Java Platform and Tools

The Java platform is in principle well-equipped for staged meta-programming, by virtue of its *class loader* facility that turns byte sequences, in precisely specified form, into live classes that seamlessly extend the codebase of the running program. [11]

1.2.1 Meta-Programming Frameworks. There are so many frameworks and tools for meta-programming on the Java platform that a survey is out of scope here. See [19] and [11] for introductions from the compile and runtime perspective, repectively. Only two relevant approaches and their most popular instances should be mentioned.

On the one hand, there are JVM bytecode construction and manipulation libraries, such as BCEL [2] and ASM [3, 13]. Their approach can be characterized as thoroughly machine-centric: the content of the JVM class file format [12, Ch. 4] is modeled directly with little abstraction.

On the other hand, there are comprehensive DSL construction kits, such as Xtext/Xtend [5, 6]. They focus on stand-alone rather than embedded DSLs, and provide tool support for all phases of language processing: syntax/parsing, abstract syntax tree (AST) models, analysis, interpretation, compilation to Java or other target formats.

1.2.2 Where is my Homoiconic Java? Sadly, the Java language is fundamentally ill-equipped for truly integrated homoiconic approaches to meta-programming [24], for various reasons:

Firstly, the provided means for code reification, such as lambda expressions and and anonymous inner classes, are hampered by non-orthogonal constraints (such as shadowed variables and checked exceptions) and complex implementation details (such as name mangling and the infamous invokevirtual instruction); they are far from being simple concepts of the underlying JVM.

Secondly, the language has rather poor compositional structure: whereas, in a typical functional language, nearly every entity of interest is an expression and thus a first-class citizen, Java program fragments at a granularity that would be useful for staged meta-programming do not have an independent existence of their own; everything is tied to whole classes.

Last but not least, the actual target for staged meta-programming on the Java platform is not the source language, but the JVM bytecode format, which differs in ways that are too substantial to be ignored, and likely to increase with the continuing evolution of both (Consider the many pitfalls and shortcomings of the Java reflection API, and the concept of bridge methods as cases in point.) In the (not unlikely) case that an EDSL's execution model differs from that of its host language Java, low-level control over JVM features can be a valuable asset for compilation.

None of the existing language objects models and bytecode libraries seems ready to address the above issues systematically, let alone effectively. Thus it appears a worthwhile experiment to deviate from the goal of homoiconic staged meta-programming on the platform, and turn the vice of diverging formats and semantics into the virtue of *heteroiconic* staged meta-programming.

1.2.3 The LLJava Language. LLJava [21] is a low-level class-structured, stack-oriented programming language for the JVM. It has both a textual syntax and a public abstract syntax model, for manual and meta-programming, respectively. LLJava does neither aspire to be a full high-level application programming language, nor to map all technical details of the underlying JVM directly. Instead, the design aims at a sweet spot between user and machine orientation, as an intermediate format for JVM-related compilation, experimentation, reasoning and teaching.

LLJava abstracts systematically from machine-centric accidental details of the bytecode format that can be inferred by a compiler pass. For instance, addition instructions with distinct operand types such as iadd and fadd are unified into a single add instruction, and distinct constant loading instructions such as bipush, dconst_1 and ldc are unified into a single load operation.

Tabular sections, such as the constant pool, StackMapTable attributes or exception handlers, and mangling problems, such as method descriptor syntax, are managed automatically. Conversely, the essential properties of the JVM execution model, such as the operand stack, are represented explicitly and faithfully.

The LLJava backend is small and fast, and does not require external resources. Thus it supports the efficient embedded generation of byte code, not only for offline use, but also for online loading in the running application. (See section 6 for corroborating data.)

1.2.4 The LLJava-Live API. LLJava-live is an experimental API in Builder pattern style for the LLJava abstract syntax model and backend. It is geared towards simple compositional code generation, and immediate loading and instantiation of generated classes.

Bytecode is constructed incrementally by invoking API methods that add data to the current context, such as fields to the current class or instructions to the current method body, or navigate between contexts, similar to the usage of ASM.

LLJava-live provides a unified view on variables that subsumes locals, instance fields and static fields, and implicitly selects the appropriate instruction sets: load/store, getfield/putfield, etc. Local variables and code labels are referenced by symbolic handles, their numeric allocation is automatic. For compositional code with complex data flow, LLJava-live has a hierarchical structure of local blocks with designated input and output variables. Where the order of construction events is significant but dynamic, the client code is wrapped in lambda expressions, in Command pattern style, for scheduling at the discretion of the code generator. For instance, the code that loads the operand for a write to an instance field is sandwiched implicitly between aload_0 (**this**) and putfield instructions.

When it comes to staged meta-programming, LLJava-live has full support for cross-stage persistence: Not only primitive values, but arbitrary live objects can be transferred from the meta to the object stage. While the former are easily reproducible in JVM bytecode as "fossile" constants, the latter require more sophisticated treatment. LLJava-live solves the problem by *closure conversion*, in the same way it is also employed by the Java language for the related problem of lexical capture in nested classes. The LLJava-live code generator records all references to cross-stage object references loaded as constants by instructions of the object program, and turns them transparently into synthetic constant fields, to be initialized by additional constructor parameters. The actual references are then passed automatically at instantiation time of the freshly loaded next-stage class. From the user perspective, the mechanism simply offers an overloaded variant of the load operation, with a live operand. (See Figure 8 for an application.)

1.3 Embedded Domain-Specific Languages

Embedded domain-specific languages are languages in a wider sense, having no specific syntax or parsing tools. An EDSL does have an abstract syntax however, and thus a precise notion of well-formed instances. Instances are realized in terms of data types or APIs defined in a host programming language, and can be manipulated and checked in the usual ways.

The relationship is symbiotic in nature: The host language provides general-purpose programming features, semantic constraints, and a runtime environment, whereas EDSLs provide adequately focused expressiveness for specific domains and tasks. The design and use of EDSLs is considered a key concept in the paradigm of language-oriented programming [7].

Compared with traditional textual DSLs, an EDSL has a distinctly more technical appearance, and is certainly less accessible to the non-programmer. On the upside, however, the programmatic construction of EDSL programs as meta-programming in the host language has multiple advantages: EDSL object programs can be constructed automatically by algorithms in addition to just denoted statically, thus covering all needs for parametrization and macro programming in a unified framework; the construction expressions can be checked statically with full integration into the host build process; control and data flow between host and EDSL program are smooth and efficient; and last but not least, the EDSL can inherit runtime environment and compilation features of the host language if properly adapted [20].

LLJava-live has been created in order to accelerate various non-trivial EDSLs by compilation; namely the pattern-matching language Paisley [23], the synchronous data-flow programming language Sig-adLib [22] and the parser combinator language Ramus (unpublished work in progress). The present paper is a summary of the resulting experiences, applied to an artificial but educational subject.

2 THE WHILST LANGUAGE

As the running example for the main part of the present paper, we present Whilst, a simple old-fashioned structured imperative programming language inspired by the *while* construct of computability theory [16], and the example language in [17]. The features of Whilst have been selected for didactic value, not for fitness for particular real-world programming purpose. As a true EDSL, Whilst has neither concrete syntax nor stand-alone programming tools. Its abstract syntax and all aspects of operational semantics are realized as an object model and its APIs, within the host language Java.

2.1 Static Structure

The static structure diagram of the abstract syntax model is depicted in Figure 1: A Program is a collection of global variables and named procedures.

A Procedure is either Foreign or Domestic. The former are implemented in any JVM language, by a method of some live object bound by reflection. The latter are implemented in Whilst, by specifying zero or more parameter variables, a result variable and a body statement. Every procedure has exactly one result, which may of void type. There is no *return* operation; the effective result is the final value of the result variable, which may optionally be one of the parameters.

A Statement is either a Block, which declares a set of zero or more local variables and contains a sequence of zero or more sub-statements, an Assignment, which writes the result of an expression to a variable, or a Loop, which is specified by a head expression and a body statement.

Two variants of loops are predefined in the language (not shown in the diagram): A repeat loop first evaluates its head expression to an integer, and then repeats the body statement a corresponding number of times. A While loop first evaluates its head expression to a boolean, and then either executes the body once and restarts, or terminates.

An Expression is either a Variable or Constant, an Operation with an Operator and one or two operand expressions, or a procedure

Call with zero or more argument expressions. An expression may also stand in for a statement, discarding its result.

Variables represent static identifiers in the abstract syntax; they are not carriers of mutable state themselves. A single variable object may be reused in different environments, even in nested (shadowing) or concurrent ones, without conflict. Furthermore, there is no nontrivial equality on variable objects beyond their identity; a new variable object (in the Java sense) always represents a fresh identifier.

2.2 Type System

The Whilst type system is quite primitive. The types void, integer and boolean, on which the language constructs depend, are predefined. Other types may be defined by language extensions (see below), but there is no built-in support for more complex features such as aggregate or enumeration types, subtyping or genericity.

All variables, expressions and assignments are strongly typed, both in the Whilst type system and, via type parameters in the object model, in the underlying Java type system. Type errors are detected at construction time of the abstract syntax objects. As a consequence of the design decision to use Java genericity for expressions of various types, Whilst values can not be of primitive types, but have to be boxed. (A typical performance issue of dynamic languages, to be adressed in compilation.)

Operators are polymorphically overloaded. The language holds an extensible collection of implementations of some operator for a particular type signature. Both argument types and result type are used to distinguish overloaded cases.

By contrast, procedures are monomorphic. Each procedure name in a Whilst program may be associated with only one type signature, and one implementation.

2.3 Recursion

Whilst procedures are recursive. Both domestic and foreign procedures of a program may call themselves or each other without restrictions. Foreign procedurs may also define their own private state, and call themselves or any other foreign code.

In order to construct recursive programs with type safety, procedures must be declared with their type signature before they are defined. Thus calls can be checked at construction time.

2.4 Language Packs

The Whilst language is designed for extensibility. Definitions of types, operator identifiers and typed operator implementations can be organized as *language packs*. Each language pack is an instance of a subclass of LanguagePack, and may provide both operational definitions and meta-level factory methods for the well-typed notation of expressions.

Operational semantics of a Whilst program are relative to an underlying Language object, which is a library of such definitions, obtained by ad-hoc construction or by pasting together the appropriate language packs. The language library stores operator implementations as instances of the standard functional interfaces Function and BiFunction, such that lambda expressions may be used for their concise implementation. Figure 2 shows the relevant excerpt pertaining to integer addition, with a lambda-based operator definition and type-safe expression factory method.

Several language packs have been predefined to demonstrate the possibilities: The BasicLanguagePack defines integer addition, subtraction and the nonzero predicate, barely enough for nontrivial behavior. Variants of the ArithmeticLanguagePack define the other common operations on integer and real numbers (the latter represented as Java double), respectively. The LogicLanguagePack defines boolean operations, and the Iverson bracket that maps **true** to 1 and **false** to 0.

2.5 The Builder API

The Whilst abstract syntax model comes with a Builder API that allows the reasonably concise construction of programs with complex Java expressions.

Figure 9 depicts a worked-out example, the implementation of a simple integer square root algorithm (Whilst Builder expression top left, equivalent Java code top right). The algorithm works by iterated subtraction of the differences d of successive square numbers r^2 from the argument n. Since those differences have a regular form, with the invariant $d = (r + 1)^2 - r^2 = 2r + 1$, only addition, subtraction and comparison operations are required.

2.6 Error Handling

The static semantics of Whilst is subject to many obvious type and context constraints. Violations of these are detected preferrably at construction time of the corresponding abstract syntax objects if feasible, or at interpretation time otherwise, and reported by throwing checked exceptions. This aspect of the language is omitted from API presentations for the sake of focus and brevity.

3 INTERPRETATION

The operational semantics of Whilst program constructs is given in direct form as a decentral interpreter. Each basic category of syntactic entities comes with a mode of interpretation, realized as a method: Procedures can be *called*, statements can be *executed*, expressions can be *evaluated*.

All Whilst abstract syntax objects are context-free; they can be shared between parts of the same or different programs, without affecting the semantics of the abstract syntax graph. Thus their interpretation in context requires additional information, which is bundled into an Environment object that is passed to each interpretation method.

The resulting API is depicted in Figure 3. For better overview over the cross-cutting nature of the API, we adopt the notation of AspectJ [4] extension methods, and prefix the name of a lone method with the name of its conceptually enclosing class.

The language and program underlying an environment are considered constant during each interpreter session. For variables, it is enforced that each variable is declared before a write, and written (including, as a special case, initialized with the type-specific default value) before a read.

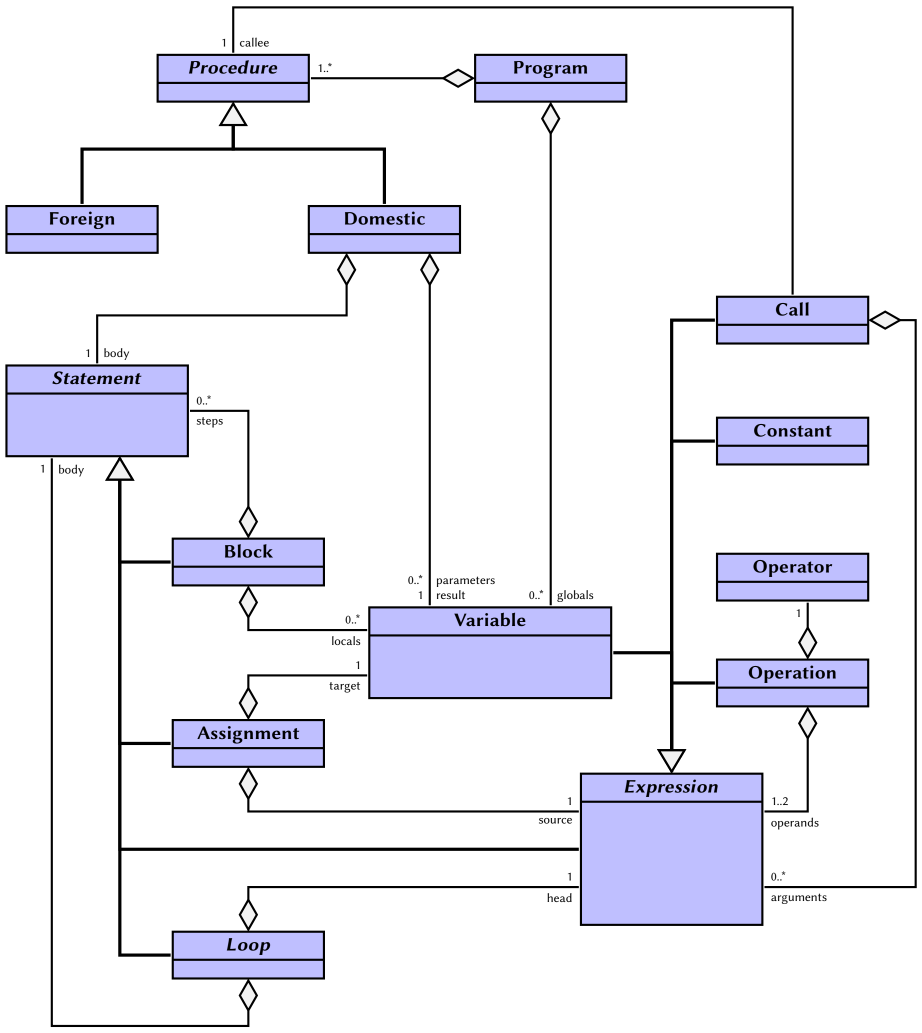

Figure 1: Static structure of the Whilst abstract syntax

```
public class BasicLanguagePack extends LanguagePack {

  public static final PLUS = new Operator();

  public void define(Language target) {
    target.defineBinaryOperator(INTEGER, PLUS,
                                INTEGER, INTEGER,
                                (x, y) → x + y);
  }

  public Expression<Integer> add(Expression<Integer> x,
                                 Expression<Integer> y) {
    return new Binary<>(INTEGER, PLUS, x, y);
  }

}
```

Figure 2: An operator syntax and semantics definition

```
public abstract Object Procedure.call(Environment env,
                                      Object... args);

public abstract void Statement.execute(Environment env);

public abstract A Expression<A>.evaluate(Environment env);

public class Environment {
  public Language getLanguage();
  public Program getProgram();
  public void declare(Variable<?> variable);
  public <A> A read(Variable<A> variable);
  public <A> void write(Variable<A> variable, A value);
}
```

Figure 3: Whilst Interpretation API

```
public R Constant<R>.evaluate(Environment env) {
  return value;
}

public R Variable<R>.evaluate(Environment env) {
  return env.read(this);
}

public R Unary<T, R>.evaluate(Environment env) {
  final T arg = operand.evaluate(env);
  return env.getLanguage()
    .getUnaryOperator(getType(), getOperator(),
                      operand.getType())
    .apply(arg);
}

public R Binary<T, U, R>.evaluate(Environment env) {
  final T arg1 = operand1.evaluate(env);
  final U arg2 = operand2.evaluate(env);
  return env.getLanguage()
    .getBinaryOperator(getType(), getOperator(),
                       operand1.getType(), operand2.getType())
    .apply(arg1, arg2);
}

public R Call<R>.evaluate(Environment env) {

  final Object[] argValues = evaluateToArray(arguments, env);
  final Procedure callee = env.getProgram()
    .getDefinedProcedure(procedureName);
  final Object result = callee.call(env, argValues);
  return getType().cast(result);
}
```

```
public void Constant<R>.compileEvaluate(CodeGenerator cg) {
  getType().compileConstant(cg, value);
}

public void Variable<R>.compileEvaluate(CodeGenerator cg) {
  cg.load(cg.getCompiledVariable(this));
}

public void Unary<T, R>.compileEvaluate(CodeGenerator cg) {
  operand.compileEvaluate(cg);
  cg.getLanguage()
    .compileUnaryOperator(cg, getType(), getOperator(),
                          operand.getType());
  // implicit data flow via operand stack
}

public void Binary<T, U, R>.compileEvaluate(CodeGenerator cg) {
  operand1.compileEvaluate(cg);
  operand2.compileEvaluate(cg);
  cg.getLanguage()
    .compileBinaryOperator(cg, getType(), getOperator(),
                           operand1.getType(), operand2.getType());
  // implicit data flow via operand stack
}

public void Call<R>.compileEvaluate(CodeGenerator cg) {
  cg.loadThis();
  for (Expression<?> arg : arguments) arg.compileEvaluate(cg);
  final Class<?> rtype = cg.getCompiledType();
  final Class<?>[] ptypes = cg.getCompiledTypes(arguments);
  cg.invokeVirtualSelf(rtype, procedureName, ptypes);
  // static typing, no cast
}
```

Figure 4: Interpreted (left) and compiled (right) evaluation of expressions; synopsis

3.1 Evaluation of Expressions

Evaluation of expressions is defined decentrally case by case (see Figure 4, left; also cf. Figure 1 for part names in aggregates):

A constant has a value that has been fixed at construction time. Evaluation returns that value, regardless of the environment. A variable evaluates to its current binding in the environment, which may fail if the variable is undeclared or uninitialized.

An operation is evaluated by recursively evaluating the operand(s) in the same environment first, then looking up the operator definition for the pertinent type signature in the language library, and then invoking that definition. By analogy, a procedure call is evaluated by recursively evaluating the arguments in the same environment first, then looking up the procedure definition in the program, and then invoking that definition. Finally, the result needs to be type-checked, because procedures are not statically typed.

3.2 Execution of Statements

Execution of statements is defined decentrally case by case (see Figure 5, left): An expression is executed as a statement by evaluating it in the given environment, and discarding the result. An assignment is executed by evaluating its source (rhs) expression, and then writing the result to its target (lhs) variable in the same environment. A while loop is executed in metacircular fashion, by evaluating its head and executing its body, in the same environment, in a JAVA **while** loop. A block is executed by creating a new nested environment, declaring all local variables, then executing its subepressions iteratively in that environment, and finally discarding the local environment.

3.3 Calling Procedures

Calling of procedures is defined decentrally case by case (see Figure 6, left): A domestic procedure is called by creating a new top-level environment (stack frame), setting each parameter variable to the corresponding value, initializing the result variable if necessary, then executing the procedure body in that environment, and finally discarding the environment. A foreign procedure is called by simply invoking a suitably prepared JVM method handle. Foreign procedure implementations are required to use unboxed values of primitive type at their interface.

3.4 Discussion

The interpreter, as outlined above, has three evident performance bottlenecks that need to be adressed by any useful compiler: The lookup of variable, operator and procedure bindings; the dynamic

```
public void Expression<R>.execute(Environment env) {
  evaluate(env);
  // result is discarded implicitly
}

public void Assignment<R>.execute(Environment env) {
  env.write(target,
            source.evaluate(env));
}

public void While.execute(Environment env) {

  while (head.evaluate(env))

    body.execute(env);
}

public void Repeat.execute(Environment env) {
  int n = head.evaluate(env);
  while (n−− > 0) body.execute(env);
}

public void Block.execute(Environment env) {
  env = env.newBlock(this);
  for (Variable<?> v : locals) env.declare(v, true);
  for (Statement s : steps) s.execute(env);
  // env is discarded implicitly
}
```

```
public void Expression<R>.compileExecute(CodeGenerator cg) {
  compileEvaluate(cg);
  if (getType() != VOID) cg.pop();
}

public void Assignment<R>.compileExecute(CodeGenerator cg) {
  cg.store(cg.getVariableForWrite(target),
           () → source.compileEvaluate(cg));
}

public void While.compileExecute(CodeGenerator cg) {
  cg.betweenLabels((before, after) → {
    getHead().compileEvaluate(cg);
    cg.branchIfZero(after);
    getBody().compileExecute(cg);
    cg.branch(before);
  });
}

public void Block.compileExecute(CodeGenerator cg) {
  cg.newBlock(this, () → {
    for (Variable<?> v : locals) cg.declare(v, true);
    for (Statement s : steps) s.compileExecute(cg);
  });
}
```

Figure 5: Interpreted (left) and compiled (right) execution of statements; synopsis

```
public Object DomesticProcedure.call(Environment env,
                                     Object... arguments) {
  env = env.newCall(this);
  env.substituteAll(parameters, arguments);
  if (!parametersContainResult) env.declare(result, true);
  body.execute(env);
  return env.read(result);

}

public Object ForeignProcedure.call(Environment env,
                                    Object... arguments) {



  return handle.invokeWithArguments(args);

}
```

```
public void DomesticProcedure.compile(String name,
                                      CodeGenerator cg) {
  cg.methodForProcedure(this, () → {
    cg.bindAllInputs(parameters);
    if (!parametersContainResult) cg.bindOutput(result);
    body.compileExecute(cg);
    if (parametersContainResult) cg.copyOutput(result);
  });
}

public void ForeignProcedure.compile(String name,
                                     CodeGenerator cg) {
  cg.methodForProcedure(this, () → {
      owner.ifPresent(cg::loadEnv);
      cg.getInputs().forEach(cg::load);
      cg.invoke(method);
  });
}
```

Figure 6: Interpreted (left) and compiled (right) procedure calls; synopsis

traversal of the abstract syntax tree; and the boxed representation of data.

4 COMPILATION

The former two shortcomings of interpretation are dealt with routinely by the general principle of compilation as staged interpretation, by shifting work from the (runtime) object stage to the (compile time) meta-stage. For an example of lookup elimination, compare the object-level lookup of interpretation method

Variable.evaluate and the meta-level lookup of its compilation counterpart Variable.compileEvaluate in Figure 4. For an example of traversal elimination, compare the object-level **for** loops of interpretation method Block.execute and the meta-level **for** loops of its compilation counterpart Block.compileExecute in Figure 5.

By contrast, the last shortcoming is an artefact of the irregular JAVA type system, and requires more ad-hoc treatment, where each WHILST type is optionally assigned a second, primitive representation, and the canonical (un)boxing conversions are applied transparently.

4.1 Code Generator

Following the idea that a compiler is just a staged interpreter, the compilation API of the WHILST model is analogous to the interpretation API. Instead of directly producing the effect of a program construct, however, a fragment of code is emitted that can produce the effect at a later stage.

Thus dependency on context arises not just at the object stage on the input side as in the interpretation case, but also at the meta-stage on the output side, where code fragments have to be assembled in a meaningful way. In order to synchronize meta and object context, both are bundled into a CodeGenerator object that is passed around.

In accordance with the structure of JVM bytecode, the compilation strategy is split into two layers: an upper layer that maps whole WHILST programs to JVM classes and is implemented centrally in the code generator, and a lower layer that maps statements and expressions to instruction sequences, and is implemented decentrally, alongside the interpreter clauses, in the constituent abstract syntax objects.

An interesting secondary aspect of staging is that error-checking diagnostics represented by checked exceptions are to be concentrated at the meta-stage, thus giving a concrete metric of the degree of type and context safety achieved for the object stage.

4.2 Layers of Compilation

The upper layer is determined by basic object-oriented conventions: a WHILST program is compiled into a single JVM class, with each global variable mapped to a private field, and each procedure to a public method, respectively. All of these are non-static, such that multiple instances of the same program can coexist concurrently without interference, just like multiple interpretation environments.

The upper layer tasks are implemented in the CodeGenerator object that is passed around, and amount to some 200 lines of JAVA code, mainly adapter boilerplate for the underlying LLJAVA-LIVE API.

The lower layer is determined by the above principle of compilation as staged interpretation, and duplicates the structure of the interpretation API exactly. It makes both direct (generic) and indirect (WHILST-specific) use of the LLJAVA-LIVE APIs inherited by the CodeGenerator class; the direct uses are underlined in the code shown in Figures 4–8.

At this lower layer, staged metaprogramming tactics and the impact of heteroiconicity come into play. The meta and object level are encoded as Java source and JVM bytecode, respectively. Since these formats are typically mediated by a compiler, the programmer who stages an interpreted has to think in compiler construction concepts. As a helpful guideline, one may first compile the interpreter with `javac` and observe the isolated bytecode pattern, and then generalize to a modular code building block.

4.3 Compiling Evaluation of Expressions

Evaluation of expressions is compiled to instruction sequences that loads (pushes) the result value onto the JVM operand stack (see Figure 4, right):

Evaluation of a variable compiles to a LLJAVA-LIVE load operation, which in turn is compiled into a JVM load instruction for local variables, or a getfield instruction for fields, respectively. Evaluation of a constant compiles to a type-specific constant-loading operation, of which the JVM has various.

Evaluation of composite expressions compiles recursively to instruction sequences that load the values of subexpressions on the operand stack, followed by an operator or method invocation instruction that consumes (pops) these intermediate values and loads a result.

4.4 Compiling Execution of Statements

Execution of statements is compiled into instruction sequences that leave the operand stack invariant, and thus work by side effect (see Figure 5, right):

Execution of an assignment compiles to a LLJAVA-LIVE store operation for the target variable, which in turn is compiled into a JVM store instruction for local variables, or a putfield instruction for fields, respectively. The value to be stored is loaded onto the operand stack by inserting the instruction sequence obtained by compiling the evaluation of the source expression, in the appropriate place.

Execution of a while loop is compiled in the way recommended by the JVM specification [12, Ch. 3]. For illustration purposes, the code generator is shown here instruction by instruction, rather than wrapped up in a convenience method. Whether to use the basic LLJAVA-LIVE APIs directly, or to aggregate them into more problem-specific high-level operations is a controversial matter of design. The adequate amount of boilerplate may vary with EDSL types and user context.

Execution of a block is compiled by declaring the local variables in the code generator, and then compiling the substatements sequentially, thus concatenating their instruction sequences. Note how both the variable management and the iterative traversal of substatements occur at the meta stage, and are thus eliminated from the object stage.

4.5 Compiling Procedures and Calls

As shown in Figure 4, procedure calls are always compiled to JVM invocations of methods of the same compiled program object (see Figure 6, right).

For domestic procedures, there is a corresponding target method, by design of the high-level compilation scheme. The code generator for the procedure populates the body of that method, and handles the special case of the result variable being contained in the parameter list.

For foreign procedures, a bridge method is created that calls the actual target. The method to be invoked may be either static, or non-static and requiring an owner object fixed at construction time.

The dynamic method reference, stored in the abstract syntax as reflection data, is compiled to a hard-coded JVM method reference in both cases. The owner object may be any live object of the meta-level JAVA program, and is included into the compiled program by cross-stage persistence.

4.6 Putting Things Together

As a practical example, Figure 9 (bottom left) depicts the JVM bytecode (disassembled with JAVAP) generated for the integer square root program. Since the code has been generated in decentral fashion and not post-processed, the contributions of individual abstract syntax entitites can be clearly discerned.

Note the redundant operations between adresses 8–16, due to the double role of boolean values as data and branch conditions. However, since performance-relevant code is eventually JIT-compiled anyway, there may be no need for further optimization at this stage.

Compare the resulting machine code depicted in Figure 9 (bottom right). Only instructions related to stack administration and synchronization points have been omitted. The synopsis suggests that the JIT compiler is quite able to optimize the redundancy away. (See also section 6 for further evidence.)

4.7 Special Considerations

4.7.1 The Void Type. The JVM is plagued by an inherited illness running in the C-like language family: **void** is nearly a type, but not quite, giving rise to annoying irregularities, such as distinct unary and nullary **return** instructions. The issue is further confounded by the fun fact that the JAVA reflection framework considers **void.class** a proper primitive type reification.

By contrast, in WHILST the void type is completely regular, albeit without constants or operators, such that void variables are doomed to retain their default value forever. They do not have a proper JVM counterpart, except for void result variables that map to void (non-)results. Hence the theoretically nicest compilation tactic is to eliminate them from code generation, and thus also from variable and parameter lists, altogether. Most of the dirty work can be accomplished by extending the concept of LLJAVA-LIVE virtual variables by a new case, whose load and store operation do nothing. Some situations require special care, e.g., see the explicit conditional pop instruction in Figure 5 (right).

4.7.2 The Boolean Type. The JVM does have a proper **boolean** type, but emulates truth values with **int** words. The specification is slightly ambiguous about the supposed encoding, namely whether **true** is denoted by any nonzero value (as in C), or just by the value one. Each case has some slight advantages in compilation: The former simplifies the nonzero check (to a no-operation), whereas the latter simplifies the Iverson bracket (also to a no-operation) and the and and not operations.

The decision for either encoding can be realized as a configurable property of the code generator, with the compilation code for each affected operation performing a case distinction at the meta stage, thus incurring no object-level penalty at all.

4.7.3 Falling Back to Interpretation. In the staged meta-programming approach, the dynamically constructed object program is completely integrated into its meta-level host. There is no reason why the call relationship should be asymmetrical; the host evidently wishes to call the object program, but the object program might call back just as well.

```
public void BasicLanguagePack.define(Language target) {
  // ...
  target.defineCompileBinaryOperator(INTEGER, PLUS,
                                     INTEGER, INTEGER,
                                     cg → cg.add());
}
```

Figure 7: Compilation in basic language pack

```
final Function operator = cg.getLanguage()
  .getBinaryOperator(getType(), getOperator(),
                     operand1.getType(), operand2.getType());
cg.loadEnv(operator);
operand1.compileEvaluate(cg); /*[B]*/
operand2.compileEvaluate(cg); /*[B]*/
cg.invoke(METHOD_BiFunction_apply); /*[U]*/
```

Figure 8: Staged eta expansion of binary operators

This simple observation has far-reaching implications for compilation: If a compiled program may call back to interpretation, then compilation support, which is by design decentral as argued above, need not be complete in order to be effective.

This does not mean that any old fragment of a program can be excluded from compilation, however, as obvious from the API. Since interpretation depends on an environment, only certain *articulation points* of the control flow graph, where the appropriate environment can be reconstructed from compiled information, are suitable for switching back. In the WHILST setting, two categories of such articulation points can be found, namely domestic procedures and operator implementations. For the former a top-level environment can be crafted, for the latter no environment is required at all.

LLJAVA-LIVE has a generic pattern for falling back to the preceding stage. Depending on the theoretical background, it can be understood as the staged version of *eta expansion*, or as an instance of the *reverse stub* construct for virtual machines. The pattern can be described concisely as:

> "By default, implement the staged counterpart of an operational API method *m* as the generation of code to invoke *m* on the cross-stage persistent owner object."

For example, consider again the case of the integer addition operator. The basic language pack contains an additional definition, naturally in terms of the corresponding JVM instruction (see Figure 7 and cf. Figure 2). If this definition were absent, then the fall-back mechanism would emit code to call the functional interface method BiFunction.apply of the object created by the lambda expression in the interpreted definition of the operator (see Figure 8).

In the resulting object code, the lambda object that reifies the operator is loaded as a cross-stage persistent reference. (The lookup has been performed at the meta-level exactly as in the interpreter, except that the actual invocation of the apply method is of course omitted.) The operand values are loaded onto the operand stack by the compiled code for the evaluation of the operand expressions. Finally, the apply method that is bound to the lambda expression is invoked. The code has been simplified at the positions marked with [B] and [U], where boxing and unboxing of primitive values, respectively, has to be performed in a type-specific way, since the lambda expression works on boxed types.

4.7.4 Exploiting Staged Code Lifetime. The lifetime of LLJava-live-generated code for later stages is limited to the host JVM. It is intended to be executed right away without persistent storage of generated class bytecode. While this appears wasteful, there are some distinctive and intriguing advantages, just like for JIT compilation to machine code. In the following, we describe a particular illustrative example application in the Whilst compiler; the general potential of the technique is much wider.

An interpreted Whilst program comprises global variables and procedures with dynamic scope and type, which can hence be organized simply via hashtables. By contrast, compilation turns variables and procedures into statically scoped and typed fields and methods, respectively. These require heterogenous access code, rather a **switch** statement than a hashtable lookup. For the purpose, we adopt a technique from the reference implementation of **switch**-on-String, introduced to Java in Version (1.)7.

The compilation tactic uses ordinary **int**-based **switch** cases as hash buckets. This requires that the hash code of a case label constant at compilation agrees with the hash code of the matching key value at execution time. For objects of class String, the hash code is specified explicitly, hence the tactic is always applicable. In the case of staged code, compilation and execution time share the same JVM session, hence any object whose hash code is *stable for lifetime*, even if not specified any further, will do. This includes the default implementation Object.hashCode(). The Whilst compiler exploits the behavior to generate **switch** statements on objects of class Variable for efficient, statically safe implementations of Environment.read/write.

5 EXTENDING THE LANGUAGE

The Whilst language can be extended in various ways. Depending on their nature, the burden of compilation support can be felt with very different impact. Analogous to the case of the infamous expression problem [25], there is an axis that cross-cuts interpretation and compilation, as well as an axis from which they can be separated, and dealt with in an agile manner.

The abstract syntax is a class hierarchy with public APIs only, and hence open for extension by new subclasses for new language constructs. Several kinds of obvious extensions that illustrate common interpretation and compilation tactics come to mind: IfThenElse statements, short-circuiting boolean operators, a Comma constructs that prefixes an expression with a statement, concurrency primitives, etcetera.

There is no simple way to make compilation support optional, and transparently fall back to interpretation, for such new language constructs. By the self-similar nature of statements and expressions, this would require reconstructing an environment for interpretation, in the middle of things anywhere in compiled code. That might not be impossible, but is hardly a trivial task. Thus, compilation is only feasibly available in the presence of new language constructs if they all support it.

For the articulation points designed into the language, things are different however. For operator implementations defined ad-hoc or in language packs, there is a clear and transparent fall-back mechanism, such that their compilation support is nice to have for optimal performance, but otherwise completely optional. A user who extends the language in this way needs not even to be aware of the possibility of compilation, and can still provide definitions that function correctly in compiled programs. This lowers the bar of expertise required to contribute to language expressivity, and enables rapid prototyping of experimental features.

6 BENCHMARKS

The performance of the Whilst interpreter and compiler has been evaluated empirically.

All measurements have been carried out on a dual Core i5-10210U CPU at 1.6 GHz with 8 GiB of RAM, running Ubuntu 20.04LTS and the OpenJDK 11.0.10 64-bit Server VM. Times are wallclock times measured with System.nanoTime precision. To mitigate the erratic influence of JIT compilation and GC pauses, we report median and median absolute deviation of a sample of $N = 100$ repetitions, after a warm-up of the same magnitude.

The following microbenchmarks have been investigated: Totient calculates Euler's totient function, $\varphi(n) = \sum_{i=1}^{n}[\gcd(n,i) = 1]$, where the subprocedure gcd is implemented with Euclid's algorithm, for $n = 100\,000$. Fibonacci calculates the n-th Fibonacci number, with the naïve recursive definition $F_{n+1} = F_n + F_{n-1}$, and **if** emulated by **while**, for $n = 31$. Spin calculates a full-circle 2D rotation in 1 296 000 small steps, by iterated multiplication of a 2×2 rotation matrix corresponding to an angle of one arcsecond, where matrix elements are held individually as global variables.

For each benchmark, the interpreted and compiled Whilst programs are investigated, together with a hand-coded baseline Java implementation, compiled statically with javac. The results are consistent with expectations: Compilation increases the speed of Whilst programs by up to three orders of magnitude, which appears reasonable for tight loops. The speedup is particularly dramatic for the Spin benchmark, where the JVM JIT compiler has been observed to select the appropriate AVX [14] extension instructions with good effect.

Running times for compiled Whilst are indistinguishable from the baseline given the observed uncertainties, except in the case of Fibonacci, where javac does a better job at branch condition optimization than our simplistic code generator, resulting in a 10% penalty. However, the benefits of simplicity are reaped in the compilation phase, which performs LLJava-live AST construction and checking, bytecode serialization to a heap array, class loading and instantiation of object code, in about a millisecond for each example.

Bytecode size for compiled Whilst is generally slightly bigger than for the baseline. This is due to the emission of some redundant instructions that would require a simple but non-local pass to clean up. (The need to do so is not pressing, however, since the runtime measurements show that the JVM JIT is mostly able to get rid of them anyhow.) The exception here is the Spin case, where the our compiler producess *less* code than javac. This is due to the

Table 1: Benchmark results

Time (ms)	**Totient**	**Fibonacci**	**Spin**
interpreted	374.37±3.45	3 549.63±24.13	2 359.49±10.56
compiled	7.64±0.13	10.03± 0.11	3.18± 0.05
baseline	7.73±0.09	8.95± 0.10	3.11± 0.04
compilation	1.35±0.11	1.29± 0.13	1.45± 0.22
# AST Nodes	60	31	115
compiled (byte)	530	373	853
baseline (byte)	463	343	882

explicit initialization code for the rotation matrix with invocations of Math.sin and Math.cos, which are naturally shifted to the meta-level in the Whilst compiler.

7 CONCLUSION

The concept of heteroiconic staged meta-programming, as implemented by the LLJava-live API, has been illustrated with the construction of a competitive modular compiler for the Whilst language from a trivial interpreter. The presentations of interpretation and compilation of course employ different idioms of the host language, for all the reasons given in section 1.2.2. But they are sufficiently similar for the individual translation tasks to be feasible and well-understood. In fact, the JVM specification has a whole chapter [12, Ch. 3] entitled *"Compiling to the JVM"* devoted to recommended solutions for similar problems. The design of the LLJava-live API aspires to the same level of intuitive appeal and constructive fit, but with executable modular code generator building blocks rather than illustrative particular pairs of high-level and bytecode fragments.

The alternative possibility of compilation is a game-changer for interpreter implementation; the need for complex optimizations in the interpreter is greatly reduced. By keeping things simple, a clean staged compiler that produces clean code is obtained. Many of the stereotypical performance issues of interpreters, such as dynamic scoping and typing, and abstraction barriers, vanish in the process by virtue of standard compilation techniques, either in the compilation step from the EDSL to JVM bytecode, or in the subsequent JIT compilation step to machine code. Simple reference implementations of interpreters are of course cheaper to design, prototype, implement, test and debug. We conjecture also that their technical documentation, and even code, may serve as effective user training material in the process of empowerment for language-oriented programming.

In places where the EDSL has been designed for extensibility, fallback mechanisms from compiled to interpreted code can be provided by a canonical technique. Thus the implementation of compilation support is a transparent, fine-grained, optional optimization task. It need not even be undertaken by the same person or delivered in the same unit of deployment, giving the whole compilation business a distinctly "agile" and "democratic" feel.

Thus the stakeholders of a DSL can have the best of both worlds: A simple language design with intention-centric operational semantics in terms of an interpreter reference implementation, and a transparent acceleration strategy with flexible modular case-based compilation to JVM bytecode. The latter smoothly extends the "drainage basin" of the underlying JVM JIT compilation capabilities and puts high-level DSLs in effective direct contact with state-of-the-art machine code realizations.

7.1 Related Work

The most popular DSL construction framework on the JVM is Xtext [6], together with the backend language Xtend [5]. These set the standard for maturity and IDE integration. However, they are largely concerned with textual, stand-alone DSLs that integrate into a multilingual project at build time rather than at runtime.

The existing work closest in spirit to our approach is *Lightweight Modular Staging* (LMS) [15], which has been implemented in Scala. It demonstrates how close one can get to homoiconic metaprogramming on the JVM, making heavy use of the sophisticated Scala type system, overloading and mixins. However, their code generation strategy involves generating Scala source code and invoking the compiler, thus the predicate "lightweight" certainly does not extend to the backend, in particular in comparison with LLJava-live. The description in [15] does not clarify to what extent cross-stage persistent live objects are supported.

7.2 Future Work

The LLJava-live API is novel and experimental, hence it is not yet clear which features are the truly indispensable ones for staged meta-programming on the JVM. The case studies that have been carried out so far are inconclusive, owing to their very different nature: the Whilst and Sig-adLib languages are dominated by structured control and data flow, respectively, where as Paisley is a logical language characterized by data matching and backtracking. A consolidation and validation of the interfaces developed so far is needed.

The bytecode resulting from a compiler obtained in the way described here has a notable fine-grained compositional structure. All case studies so far indicate that this matches well with the heuristics of the JVM JIT, resulting in well-optimized machine code. The limitations and caveats of this beneficial situation are not yet well-understood, and require future research.

An aspect-oriented notation has been used in this paper, half-heartedly for solely illustrative reasons, for the synopsis of interpreter and compiler code. The question whether interpretation and staged compilation of EDSLs are suitable for implementation by aspect-oriented Java programming in the narrow sense is intriguing, and should be investigated in the future.

```
final BasicLanguagePack basic = new BasicLanguagePack();
final IntegerArithmeticLanguagePack arith = new IntegerArithmeticLanguagePack();
return new Builder()
  .withLanguagePacks(basic, arith)
  .buildProgram(prog → {
      final Variable<Integer> n = Type.INTEGER.newVariable("n");
      final Variable<Integer> r = Type.INTEGER.newVariable("r");
      prog
        .procedure(r, "isqrt", n)
        .define(isqrt → {
            final Variable<Integer> d = Type.INTEGER.newVariable("d");
            isqrt.withLocal(d)
              .assign(d, basic.constant(1))
              .assign(r, basic.constant(0))
              .whileLoop(arith.greaterOrEqual(n, d), l → {
                              l .assign(n, basic.subtract(n, d))
                                .assign(d, basic.add(d, basic.constant(2)))
                                .assign(r, basic.add(r, basic.constant(1)));
                            });

          });
  });
```

```
class IntegerSqrt {



  int isqrt(int n) {
    int r;

    int d;
    d = 1;
    r = 0;
    while (n >= d) {
      n = n − d;
      d = d + 2;
      r = r + 1;
    }
    return r;
  }
}
```

```
public int isqrt(int);
  Code:
      0: iconst_0
      1: istore_3
      2: iconst_1
      3: istore_3
      4: iconst_0
      5: istore_2
      6: iload_1
      7: iload_3
      8: if_icmplt 15
     11: iconst_1
     12: goto 16
     15: iconst_0
     16: ifeq 34
     19: iload_1
     20: iload_3
     21: isub
     22: istore_1
     23: iload_3
     24: iconst_2
     25: iadd
     26: istore_3
     27: iload_2
     28: iconst_1
     29: iadd
     30: istore_2
     31: goto 6

     34: iload_2
     35: ireturn
```

```


0x00007fa63c3bb25c: mov $0x1,%r11d

0x00007fa63c3bb262: xor %eax,%eax





0x00007fa63c3bb264: jmp 0x7fa63c3bb283
0x00007fa63c3bb266: nopw 0x0(%rax,%rax)
0x00007fa63c3bb270: mov 0x108(%r15),%r10
0x00007fa63c3bb277: sub %r11d,%edx



0x00007fa63c3bb27a: add $0x2,%r11d



                    ...

0x00007fa63c3bb283: cmp %r11d,%edx
0x00007fa63c3bb286: jnl 0x7fa63c3bb270
                    ...
0x00007fa63c3bb297: retq
```

Figure 9: Four views on the integer square root algorithm: WHILST Builder construct hosted in JAVA (top left); hand-coded JAVA equivalent (top right); JAVAP disassembly of JVM bytecode from LLJAVA-LIVE compilation (bottom left); disassembly of machine code from JVM JIT-compilation (bottom right).

REFERENCES

[1] Alan Bawden. 1999. Quasiquotation in Lisp. In *Proc. ACM SIGPLAN Workshop on Partial Evaluationand Semantics-Based Program Manipulation (PEPM '99)*, Olivier Danvy (Ed.), Vol. NS-99-1. BRICS, 4–12.

[2] Apache Software Foundation 2020. *Apache Commons BCEL*. Apache Software Foundation. https://commons.apache.org/proper/commons-bcel/

[3] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. 2002. ASM: A code manipulation tool to implement adaptable systems. (2002).

[4] Eclipse Foundation 2020. *AspectJ*. Eclipse Foundation. https://www.eclipse.org/aspectj/

[5] Eclipse Foundation 2020. *Xtend*. Eclipse Foundation. http://www.eclipse.org/xtend/

[6] Eclipse Foundation 2020. *Xtext*. Eclipse Foundation. http://www.eclipse.org/xtext/

[7] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy, and Sam Tobin-Hochstadt. 2018. A Programmable Programming Language. *Commun. ACM* 61, 3 (Feb. 2018), 62–71. https://doi.org/10.1145/3127323

[8] Paul Hudak. 1996. Building Domain-Specific Embedded Languages. *ACM Comput. Surv.* 28, 4es (Dec. 1996), 196–es. https://doi.org/10.1145/242224.242477

[9] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. 1993. *Partial Evaluation and Automatic Program Generation*. Prentice Hall International.

[10] Alan Kay. 1969. *The Reactive Engine*. Ph.D. Dissertation. University of Utah.

[11] Ignacio Lagartos, Jose Redondo, and Francisco Ortin. 2019. Efficient Runtime Metaprogramming Services for Java. *Journal of Systems and Software* 153 (04 2019), 220–237. https://doi.org/10.1016/j.jss.2019.04.030

[12] Tim Lindholm, Frank Yellin, Gilad Bracha, Alex Buckley, and Daniel Smith. 2018. *The Java Virtual Machine Specification* (Java SE 11 ed.). JSR, Vol. 384. Oracle. https://docs.oracle.com/javase/specs/jvms/se11/jvms11.pdf

[13] OW2 2021. *ASM*. OW2. https://asm.ow2.io/

[14] James Reinders. 2013. *Intel AVX-512 Instructions*. Intel. https://software.intel.com/content/www/us/en/develop/articles/intel-avx-512-instructions.html

[15] Tiark Rompf and Martin Odersky. 2012. Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs. *Commun. ACM* 55 (2012), 121–130. https://doi.org/10.1145/2184319.2184345

[16] Uwe Schöning. 2008. *Theoretische Informatik – kurz gefasst* (5th ed.). Oxford University Press.

[17] Timothy Sheard. 1999. Using MetaML: A Staged Programming Language. In *Advanced Functional Programming*, Doaitse Swierstra, Pedro Henriques, and Jose Oliveira (Eds.). LNCS, Vol. 1608. Springer, 207–239. https://doi.org/10.1007/10704973_5

[18] Walid Taha and Timothy Sheard. 2000. MetaML and multi-stage programming with explicit annotations. *Theoretical Computer Science* 248, 1 (2000), 211–242. https://doi.org/10.1016/S0304-3975(00)00053-0

[19] James Thompson. 2019. *Metaprogramming Java: introduction*. Inkblot Software. http://inkblotsoftware.com/articles/metaprogramming-java-intro/

[20] Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. 2011. Languages as Libraries. *SIGPLAN Not.* 46, 6 (June 2011), 132–141. https://doi.org/10.1145/1993316.1993514

[21] Baltasar Trancón y Widemann and Markus Lepper. 2016. LLJava: Minimalist Structured Programming on the Java Virtual Machine. In *Proc. Principles and Practices of Programming on the Java Platform (PPPJ 2016)*. ACM, 1–6. https://doi.org/10.1145/2972206.2972218

[22] Baltasar Trancón y Widemann and Markus Lepper. 2018. Sig adLib – Mostly Compositional Clocked Synchronous Data-Flow Programming in Java. In *Tagungsband des 35ten Jahrestreffens der GI-Fachgruppe Programmiersprachen und Rechenkonzepte (IFI Reports, Vol. 482)*. University of Oslo, 31–48. http://urn.nb.no/URN:NBN:no-65294

[23] Baltasar Trancón y Widemann and Markus Lepper. 2019. Improving the Performance of the Paisley Pattern-Matching EDSL by Staged Combinatorial Compilation. In *Declarative Programming and Knowledge Management (LNAI, Vol. 12057)*. Springer, 268–285. https://doi.org/10.1007/978-3-030-46714-2

[24] u/jstuartmill. 2015. *Clojure is Homoiconic, Java is Not*. Reddit. https://www.reddit.com/r/Clojure/comments/3g4rnw/clojure_is_homoiconic_java_is_not/

[25] Philip Wadler. 1998. *The Expression Problem*. Bell Labs. https://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt