# LLVM-JVM Final Report

*Louis Jenkins*

## Contents

## Introduction

The Java Virtual Machine is a commonly-used backend for many languages, such as Java, Scala, Frege, Kotlin, and Jython, as it offers a portable runtime environment. Modern implementations use a technique called 'Just-In-Time' compilation, where executable code gets compiled on-demand where type-information can be determined.

LLVM is a state-of-the-art compiler infrastructure, offering a plethora of tools, such as its optimizer, hosting over 100 different data analysis and transformation passes, to its backend assembler, offering support for a mulitude of architectural systems.

We introduce LLVM to the Java Virtual Machine as a Just-In-Time compiler by providing a frontend, a mapping from one language to another. This approach is not entirely unique as recently a new competitor to the widely used HotSpot and OpenJDK, both maintained by Oracle, called Zing, produced by Azul Systems, boasts significant performance improvements and owe the results to LLVM. This project seeks to make use of LLVM and explore the relative advantages and tradeoffs to using LLVM.

## Java Virtual Machine

The Java Virtual Machine operates on classfiles, given the appropriate extension '.class', are the basic compilation units defined in the Java Virtual Machine specification [1] in section §4. These files contain constants, fields, methods, interfaces, as well as metadata such as attributes and type descriptors. We employ the usage of a library called *hs-java* [2] which handles parsing into a runtime-equivalent. Of the many fields contained in the classfile, the most interesting to us is the 'Constant Pool'. The constant pool contains not only constants, but the actual metadata themselves, including the actual type and class information, and is even used by some instructions at runtime.

The Java Virtual Machine operates on what is referred to as 'bytecode', which are single-byte instructions. This does not mean that all information can fit within a single byte, as some instructions are in reality multi-width, where the next 'instruction' is used as a hexadecimal constant, normally as some offset or constant value. Each instruction is associated with an instruction pointer, or as it is more commonly referred to as, a program counter. All branch instructions are multi-length, with the following bytes used, not as a constant, but as a destination for a relative or absolute jump in control flow.

The Java Virtual Machine is stack-based, where computations are performed on what is known as an 'operand stack'. Simple integer arithmetic can be seen to be rather intuitive; for example, the expression '1 + 2 + 3' can be expressed below. In each step, we display the operand stack, denoted 'O.S', and its contents.

```
1:  iconst_1 # O.S: {1}
2:  iconst_2 # O.S: {1,2}
3:  iadd # O.S: {3}
4:  iconst_3 # O.S: {3, 3}
5:  iadd # O.S: {6}
```

The operand stack only holds temporary values, the long-term values are held in an array of read/write registers, called 'local variables'. Instructions refer to local variables by their index, and as such the original variable name is discarded during compilation. Values can be loaded onto the operand stack, and the top operand can be stored; for example, we can express a simple load and store, reusing the same operand stack from before and displaying the local variables, denoted 'L.V', and its contents in each step.

```
6: istore_0 # O.S: {}, L.V: [6]
7: iload_0 # O.S: {6}, L.V: [6]
8: iconst_4 # O.S: {6, 4}, L.V: [6]
9: iadd # O.S: {10}, L.V: [6]
10: istore_1 # O.S: {}, L.V: [6, 10]
```

Function arguments are passed through the lowest available local variable slots. For non-static methods, the first slot (L.V#0) is reserved for 'this', the current class instance.

Bytecode is turing-complete, and as such offers basic support for jumps in control flow. Each stack frame maintains a program counter which determines the current instruction. Certain instructions can result in adjustments to the current program counter, resulting in a jump in control flow; these instructions can be conditional, only adjusting the program counter if a certain condition is met, or unconditional, resulting in an immediate jump. As a demonstration, below is an example of a simple loop which employs the usage of both conditional and unconditional jumps. Unlike the other examples, we will not be displaying the actual contents of the operand stack and local variables, as they are non-deterministic.

```
### Java ###
int x = 0;
for(int i = 0; i < 100; i++) {
  x = x + 1;
}

### Bytecode ###
# int x = 0;
0: iconst_0
1: istore_1

# int i = 0;
2: iconst_0
3: istore_2

# if i < 100 then goto end
4: iload_2
5: bipush 100
7: if_icmpge 20

# x = x + 1
10: iload_1
11: iconst_1
12: iadd
13: istore_1

# i++
14: iinc 2, 1

# goto start
17: goto 4

# End of Program
20: ret
```

Beginning with the initialization of local variables ‘i’ and ‘x’, we see our first multi-length instruction at PC#5, `bipush`, which reads the next instruction as an integer and pushes them on the operand stack. At PC#7, we have our first conditional jump instruction, `if_icmpge`, which compares the top two values (‘icmpge’ means ‘integer-compare-greater-equal’) and if the result is true, we adjust the program counter to the value of the next instruction; if the result is false, the program counter is unchanged and we advance to the next instruction. The control flow into the next is implicit and this fact comes with its own complications, explained later. At PC#14, we see our first operation that acts directly on local variables, `iinc`, which is performs a read-modify-write increment. On PC#17, we finally see our unconditional jump, which is a jump back to the loop condition.

Java compilers seldom perform any optimizations, so the it is up to the Java Virtual Machine itself to handle optimizations on the fly, and as such naive interpretation is unfeasible. Some optimizations are obvious, such as the propagation of constants and compile-time expressions, to simplifying instructions. Fortunately Just-In-Time compilation affords us the ability to perform optimizations, data analysis, and so forth. This requires a compiler specific to the Java Virtual Machine’s computational model, which is the ideal use-case for LLVM.

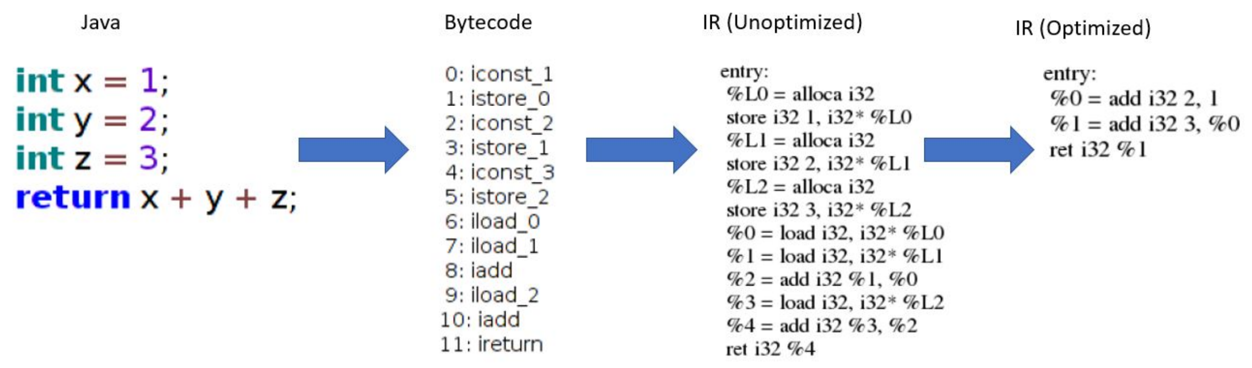
## LLVM Frontend

While we provide the frontend[3], LLVM offers both an optimizer, capable of performing advanced data-flow analysis and transformation similar to that other modern compilers, and a backend, providing portability and architecture-specific optimizations to a vast amount of systems. These are provided for free, as LLVM is not only an open-source project, but a competitive state-of-the-art compiler infrastructure. We employ the use of the library called *llvm-hs* [4], which provides a runtime interface with LLVM's native bindings, which are based on the original language references [5]. We also save time by using a modified version of the runtime and frontend of an preexisting Java Virtual Machine called MateVM[6], which provided Just-In-Time compilation to 32-bit x86 assembly using Hoopl[7], a compiler generation tool based heavily on lattices, and Harpy[8], an x86 code generator.

Like the Jave Virtual Machine, LLVM has its own virtual instruction set called Intermediate Representation. The LLVM, however, is a register-based machine, and its instruction set is similar to a strictly-typed three-address-code assembly language, except that it is also a 'single static assignment' language, meaning that all registers must be written to exactly once. The intermediate representation, unlike assembly, have 'infinite virtual' registers, where each virtual register is mapped to a real register using a register allocation[9] algorithm that utilizes a combination of liveness analysis and graph coloring.

Local variables in the Java Virtual Machine are different from the registers in LLVM in that we must only assign to the local variable exactly once, consequence of being a single static assignment language. To remedy this, we allocate space on the stack for our local variable and perform loads and stores directly on the location of the stack itself. We may also stimulate the operand stack at compile-time by treating registers as operands, which is possible in a single forward pass; thanks to the simplicity and lack of optimizations, this is always safe, and thanks to register allocation we can theoretically have an infinite number of operands.

The obvious downside in our design is in how naive our solution is: Loads and stores to memory are magnitudes slower than those to registers, but LLVM offers an optimizations called 'mem2reg' which converts memory allocations to pruned single-static assignment form using a combination of liveness analysis and dominance frontiers[10]. After the program is in pruned SSA form, other optimization passes and data-analysis can be used.



```
int x = 1;
int y = 2;
int z = 3;
return x + y + z;
```

```
0: iconst_1
1: istore_0
2: iconst_2
3: istore_1
4: iconst_3
5: istore_2
6: iload_0
7: iload_1
8: iadd
9: iload_2
10: iadd
11: ireturn
```

```
entry:
 %L0 = alloca i32
 store i32 1, i32* %L0
 %L1 = alloca i32
 store i32 2, i32* %L1
 %L2 = alloca i32
 store i32 3, i32* %L2
 %0 = load i32, i32* %L0
 %1 = load i32, i32* %L1
 %2 = add i32 %1, %0
 %3 = load i32, i32* %L2
 %4 = add i32 %3, %2
 ret i32 %4
```

```
entry:
 %0 = add i32 2, 1
 %1 = add i32 3, %0
 ret i32 %1
```

Figure 1: Mem2Reg

Beyond the basic metadata and global variables, we have functions, which are essentially control flow graphs. A Control Flow Graph is a directed graph $G = (V, E)$, where $V$ is the set of nodes $\{v_1, v_2, ..., v_n\}$, and $E$ is the set of directed edges where each edge is represented by an ordered pair, $(v, v') \in V \times V$, meaning that there is an edge from $v$ to $v'$. For convenience, we also define a relation $\rightarrow: V \times V$, where $v \rightarrow v' \iff (v, v') \in E$. As well, we define $In, Out : V \rightarrow \mathcal{P}(V)$, where $In(v) = \{w | w \rightarrow v\}$, the set of all nodes that have an outgoing edge to $v$, and $Out(v) = \{w | v \rightarrow w\}$, the set of all nodes that $v$ has an outgoing edge. We define $Path : V \times V \rightarrow \mathcal{P}(\mathcal{P}(V))$ to be set of possible sets of nodes on the path from $v$ to $v'$; for example, if $a, b, c, d \in V$ where $a \rightarrow b \rightarrow d$ and $a \rightarrow c \rightarrow d$, then $Path(a, d) = \{\{a, b, d\}, \{a, c, d\}\}$. Finally, we define a reachability relation $\rightarrow^*: V \times V$ where $v \rightarrow^* v' \iff Path(v, v') \neq \emptyset$.

In the context of control flow graphs, each node is a basic block, which is a linear sequence of program instructions having one entry point and one exit point, and each edge represents a branch in control flow. In the particular context of LLVM's Intermediate Representation, the concept of basic blocks are extended to include a single terminator instruction, either a branch or function return, as well as a distinct label. Edges in the control flow graph are identified from branch instructions, but unlike bytecode which make heavy use of program counters, the destination node is identified by it's label. For clarity, if $v \rightarrow v'$ then $v$ has the label of $v'$ as one of its possible destinations.

In the case where we have an instruction $i$ in basic block $b$ have as a destination an instruction $i'$ in basic block $b'$, if $i'$ is not the first instruction then it becomes reachable from more than one instruction, and as such is no longer linear. In this case, if $b' = \{i_1, ..., i_{k-1}, i_k, ..., i_N\}$ where $i' = i_k$, then $b'$ is split into two blocks, $b'_1 = \{i_1, ...i_{k-1}\}$ and $b'_2 = \{i_k, ..., i_N\}$, with a new edge $b'_1 \rightarrow b'_2$ and $b \rightarrow b'_2$. This case is fairly common when you have a backedge, and would significantly increase the complexity of algorithm if we had to do so in a single pass. Instead, we make a first pass to collect all basic blocks so splitting is not necessary, and then create the required edges.

We maintain a mapping of program counters to basic blocks, as the program counter is the beginning of a new block. We initialize this mapping, $O$ (line 1), with a mapping of the first instruction as our entry block. The notation $X \mapsto Y$ is used to denote a mapping from $X$ to $Y$; as well the value $\perp$ is the 'default' value for any given type, in this case an empty basic block. We maintain a naive program counter, disregarding the complexity of multiple-length instructions for sake of simplicity. In a first pass through all instructions if the instruction is a branch instruction, in this case is apart of the set of all branch instructions, $Branch$, then the instruction is the last in the current basic block, meaning the next instruction is the beginning of a new basic block (line 8). As the destination of any branch is itself the start of a new basic block, we add one for all destinations, obtainable from the function $Dest$ (line 6).

We add all basic blocks found to our graph (line 13) and make our second pass to locate the edges in the graph. Our current basic block is determined by our program counter, updated on each iteration (line 16). After some out-of-bounds checking, we add an edge from from our current block's successor (line 19) if the last instruction is not unconditional (line 18) to simulate the 'fall-through' nature of bytecode. We add the current instruction to our current basic block (line 23), and as before, if it is a branch, we add an edge to all outgoing edges (line 24 - 28).

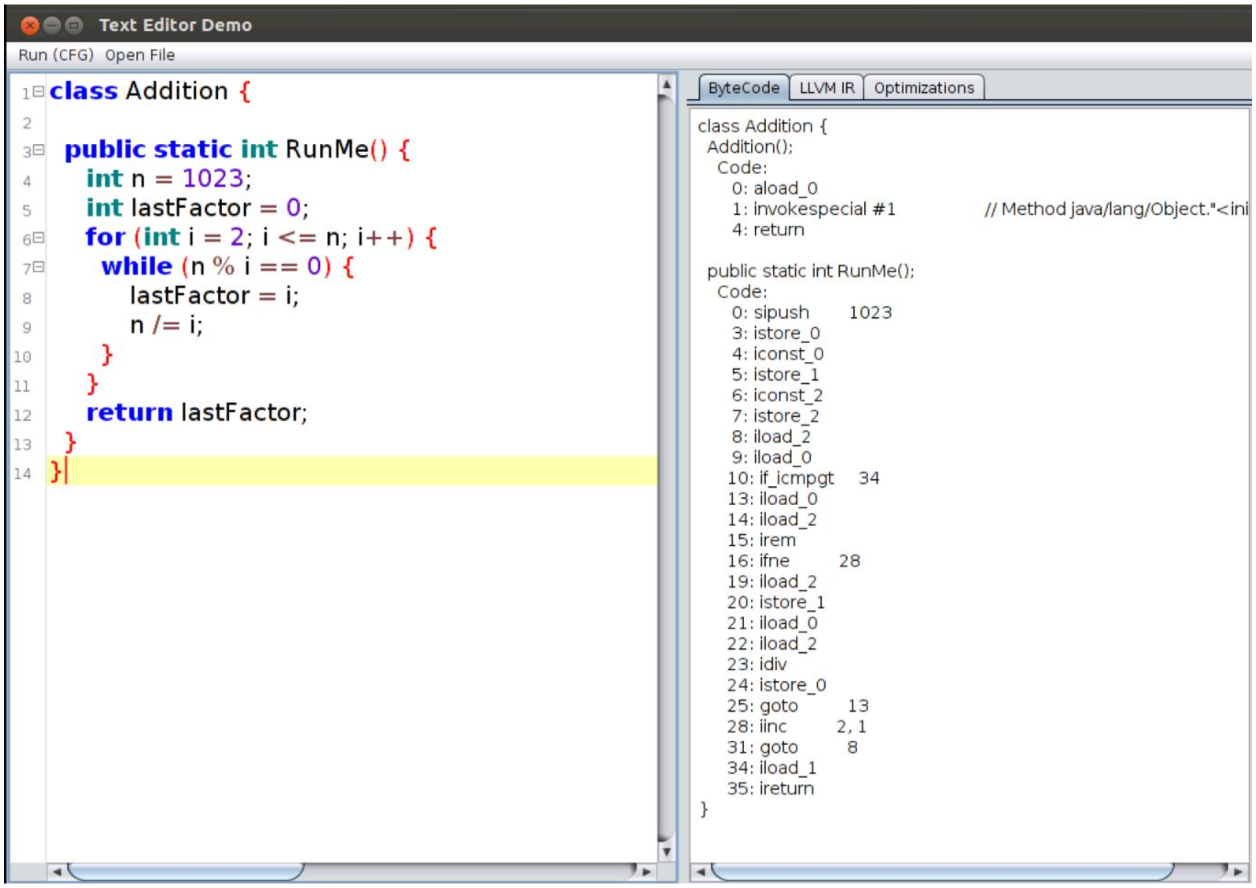
The resulting control flow graph is not much different from LLVM intermediate representation. As control flow is properly mapped, the only thing left to do is to perform instruction-level mappings which is relatively straight forward. It should be noted that the optimization tool, 'mem2reg', will handle places phi-functions where appropriate in its conversion to prune single-static assignment form. Once all mapping is complete, our naive frontend is now fully functional, with very little performance concerns.

```
Input: List, $I$, of instructions
Output: Control Flow Graph, $G = (V, E)$
$O := \{0 \mapsto \bot\}$ ;                                  /* $i \in Z, v \in V, i \mapsto v$ */
$pc := 0$ ;                                                  /* Current program counter */
/* First Pass:  Construct basic blocks                                                     */
foreach $i \in I$ do
    if $i \in Branch$ then
        foreach $d \in Dest(i)$ do
            $O.insert(d)$ ;                                 /* Add destination index */
        end
        $O.insert(pc + 1)$ ;                                /* Add successor node */
    end
    $pc \leftarrow pc + 1$;
end
$B := \bot$ ;                                                /* Current basic block */
$V \leftarrow V \cup O.values$ ;                             /* Append basic blocks to $G$ */
$pc \leftarrow 0$ ;
/* Second Pass:  Construct $E$ in $G$                                                      */
foreach $i \in I$ do
    if $pc \in O.keys$ then
        $B' := O[pc]$ ;
        /* Add edge to successor where last instruction is not a 'goto'   */
        if $pc \neq 0 \wedge I[pc - 1] \neq GOTO$ then
            $E \leftarrow E \cup (B, B')$ ;                 /* Add edge $B \rightarrow B'$ to $G$ */
        end
        $B \leftarrow B'$ ;                                 /* Update current basic block */
    end
    $B.add(i)$ ;                                            /* Append instruction */
    if $i \in Branch$ then
        foreach $d \in Dest(i)$ do
            $B' := O[d]$ ;
            $E \leftarrow E \cup (B, B')$ ;                 /* Add edge $B \rightarrow B'$ to $G$ */
        end
    end
    $pc \leftarrow pc + 1$ ;
end
```

Figure 2: Constructing Control Flow Graph

## The GUI: Frontend to the Frontend

A GUI frontend for the virtual machine has been constructed to explore the effects of optimizations on various programs. The application features a simple syntax-highlighted text editor that allows the user to run Java code; this code gets passed ‘javac’, the Java compiler, and the ‘.class’ file gets passed to ‘javap’ to produce the bytecode, shown in the ‘ByteCode’ panel below.

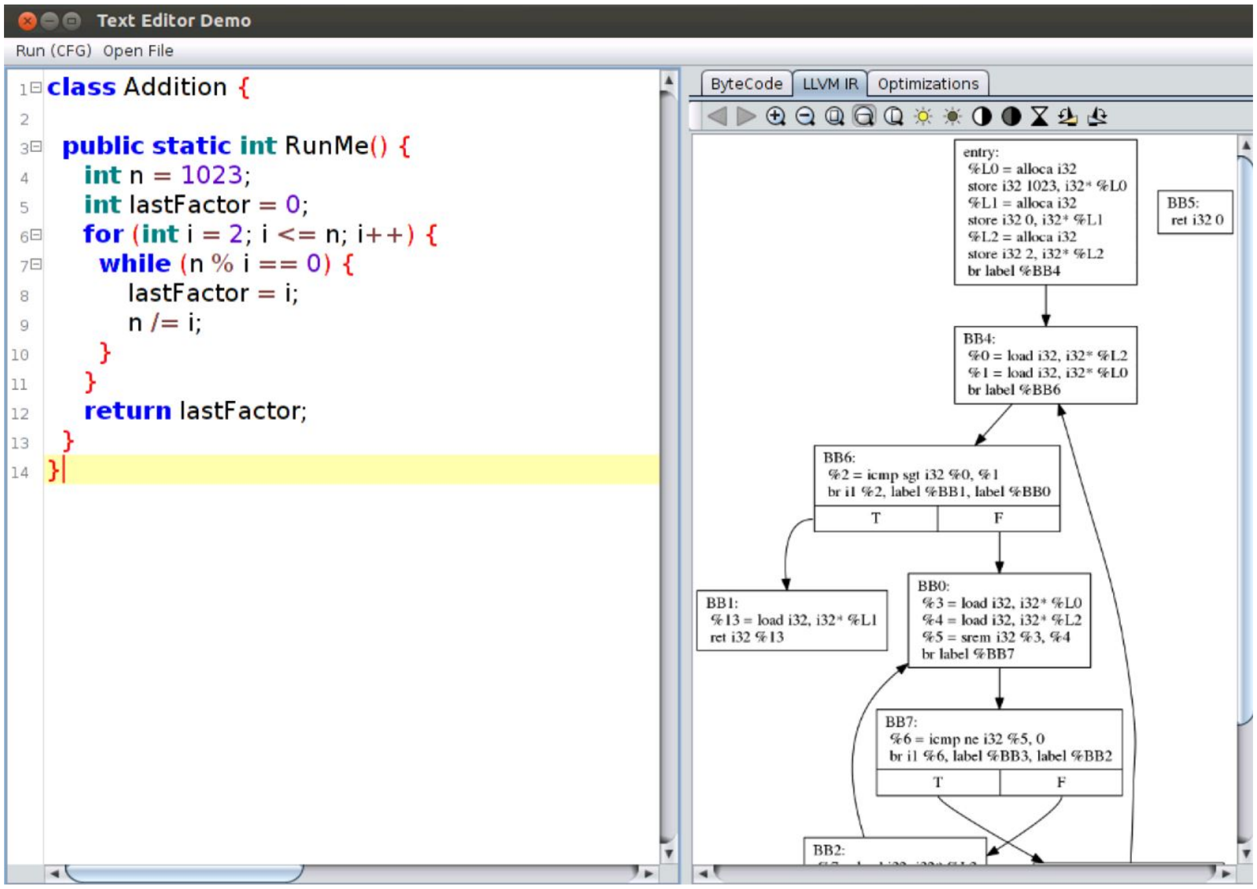


```
class Addition {

  public static int RunMe() {
    int n = 1023;
    int lastFactor = 0;
    for (int i = 2; i <= n; i++) {
      while (n % i == 0) {
        lastFactor = i;
        n /= i;
      }
    }
    return lastFactor;
  }
}
```

```
class Addition {
  Addition();
    Code:
       0: aload_0
       1: invokespecial #1                  // Method java/lang/Object."<ini
       4: return

  public static int RunMe();
    Code:
       0: sipush        1023
       3: istore_0
       4: iconst_0
       5: istore_1
       6: iconst_2
       7: istore_2
       8: iload_2
       9: iload_0
      10: if_icmpgt     34
      13: iload_0
      14: iload_2
      15: irem
      16: ifne          28
      19: iload_2
      20: istore_1
      21: iload_0
      22: iload_2
      23: idiv
      24: istore_0
      25: goto          13
      28: iinc          2, 1
      31: goto          8
      34: iload_1
      35: ireturn
}
```

Figure 3: Text Editor and ByteCode

The same '.class' file gets passed to our Java Virtual Machine, which after being handled by the frontend, outputs the unoptimized LLVM assembly. Our GUI frontend feeds that to LLVM's assembler, 'llvm-as', which outputs bitcode, and that bitcode is fed to the optimizer, 'opt', which performs any analysis and transformations and outputs a GraphViz DOT graph, which then gets fed to 'dot' to produce a visual graph, displayed in the 'LLVM IR' panel.



```
class Addition {

  public static int RunMe() {
    int n = 1023;
    int lastFactor = 0;
    for (int i = 2; i <= n; i++) {
      while (n % i == 0) {
        lastFactor = i;
        n /= i;
      }
    }
    return lastFactor;
  }
}
```

```
entry:
 %L0 = alloca i32
 store i32 1023, i32* %L0
 %L1 = alloca i32
 store i32 0, i32* %L1
 %L2 = alloca i32
 store i32 2, i32* %L2
 br label %BB4
```

```
BB5:
 ret i32 0
```

```
BB4:
 %0 = load i32, i32* %L2
 %1 = load i32, i32* %L0
 br label %BB6
```

```
BB6:
 %2 = icmp sgt i32 %0, %1
 br i1 %2, label %BB1, label %BB0
```

```
BB1:
 %13 = load i32, i32* %L1
 ret i32 %13
```

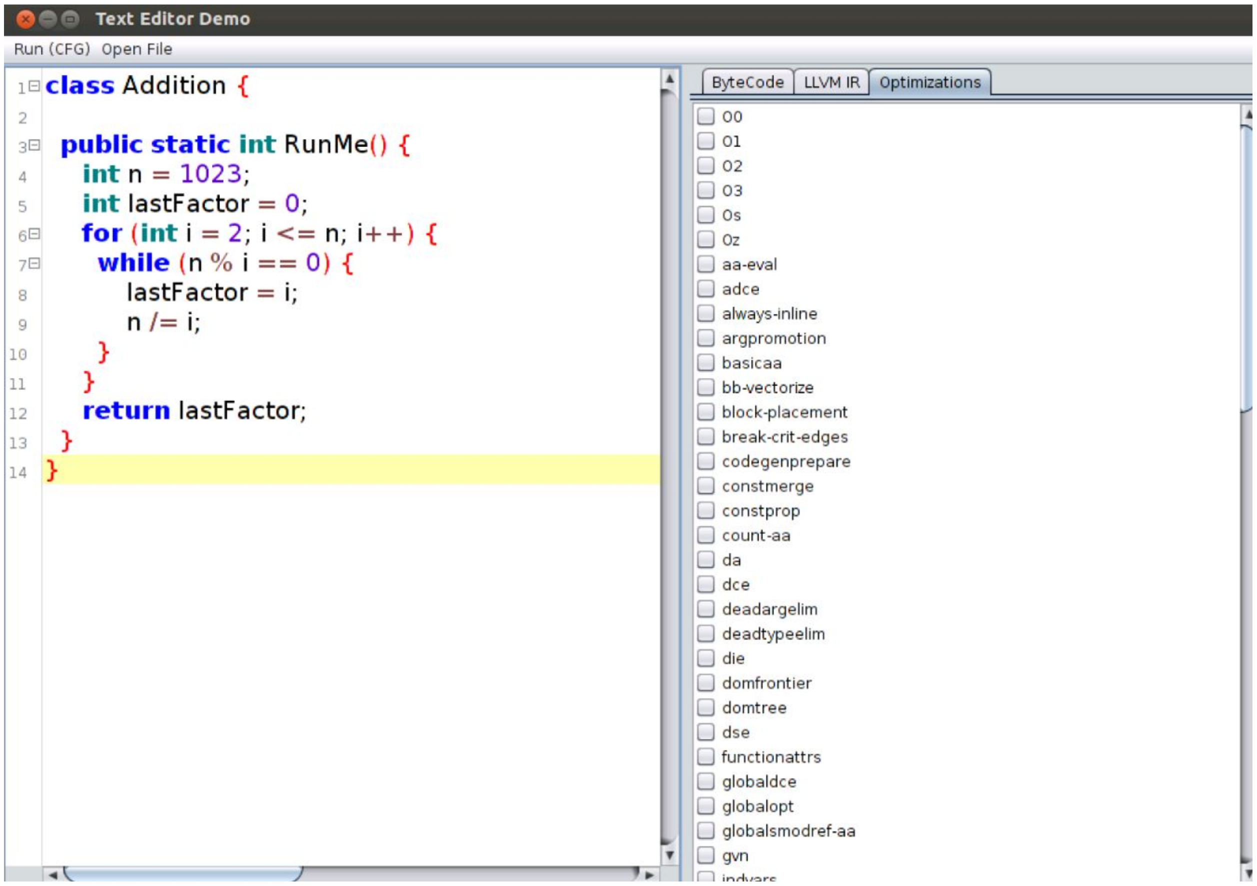
```
BB0:
 %3 = load i32, i32* %L0
 %4 = load i32, i32* %L2
 %5 = srem i32 %3, %4
 br label %BB7
```

```
BB7:
 %6 = icmp ne i32 %5, 0
 br i1 %6, label %BB3, label %BB2
```

```
BB2:
```

Figure 4: Intermediate Representation (Unoptimized)

In the ‘Optimizations’ panel, we allow the user to select various optimizations, in which is applied in the order selected.

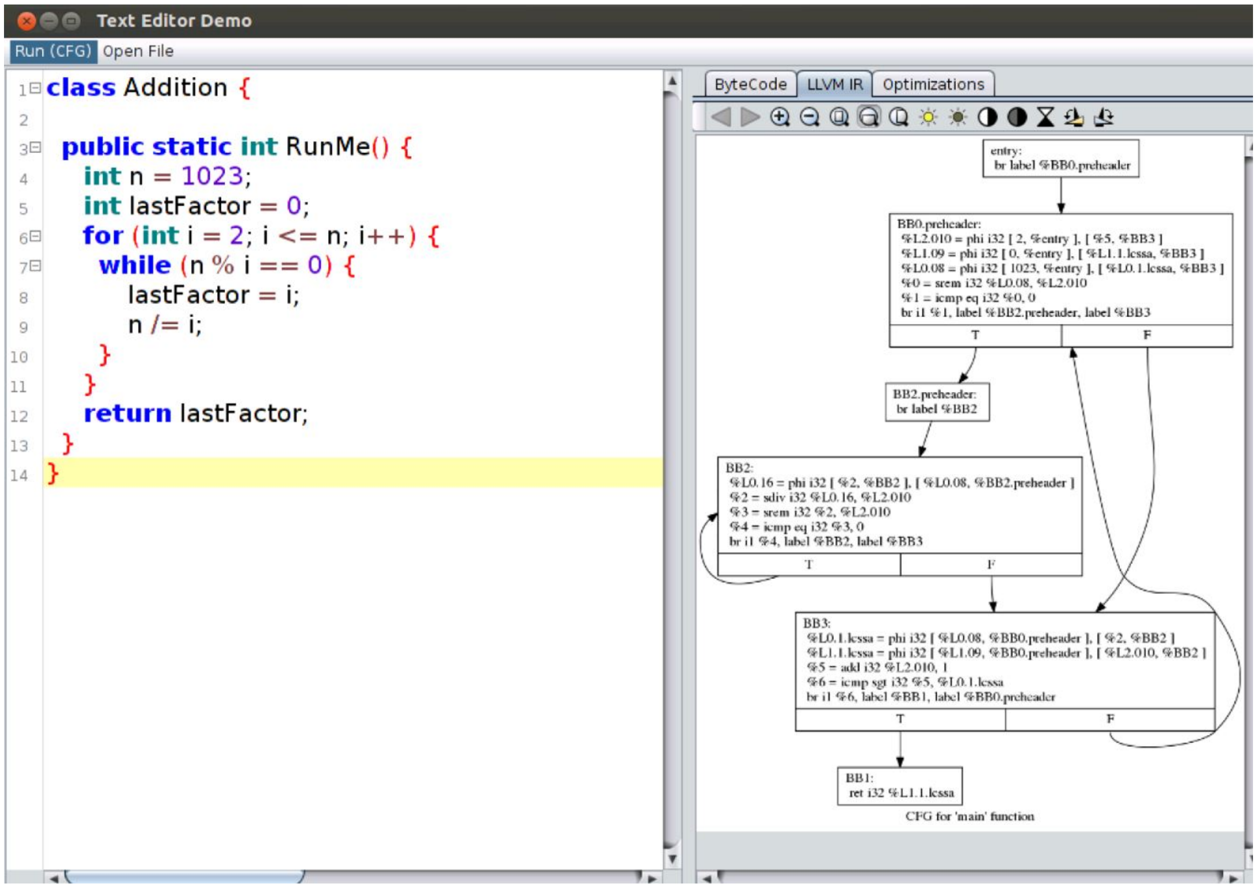


```java
class Addition {

 public static int RunMe() {
   int n = 1023;
   int lastFactor = 0;
   for (int i = 2; i <= n; i++) {
    while (n % i == 0) {
      lastFactor = i;
      n /= i;
    }
   }
   return lastFactor;
 }
}
```

- [ ] O0
- [ ] O1
- [ ] O2
- [ ] O3
- [ ] Os
- [ ] Oz
- [ ] aa-eval
- [ ] adce
- [ ] always-inline
- [ ] argpromotion
- [ ] basicaa
- [ ] bb-vectorize
- [ ] block-placement
- [ ] break-crit-edges
- [ ] codegenprepare
- [ ] constmerge
- [ ] constprop
- [ ] count-aa
- [ ] da
- [ ] dce
- [ ] deadargelim
- [ ] deadtypeelim
- [ ] die
- [ ] domfrontier
- [ ] domtree
- [ ] dse
- [ ] functionattrs
- [ ] globaldce
- [ ] globalopt
- [ ] globalsmodref-aa
- [ ] gvn

Figure 5: Optimization Menu

Running the program a second time with the optimization selected, in this case '-O3', it will result in a new graph in the 'LLVM IR' panel.

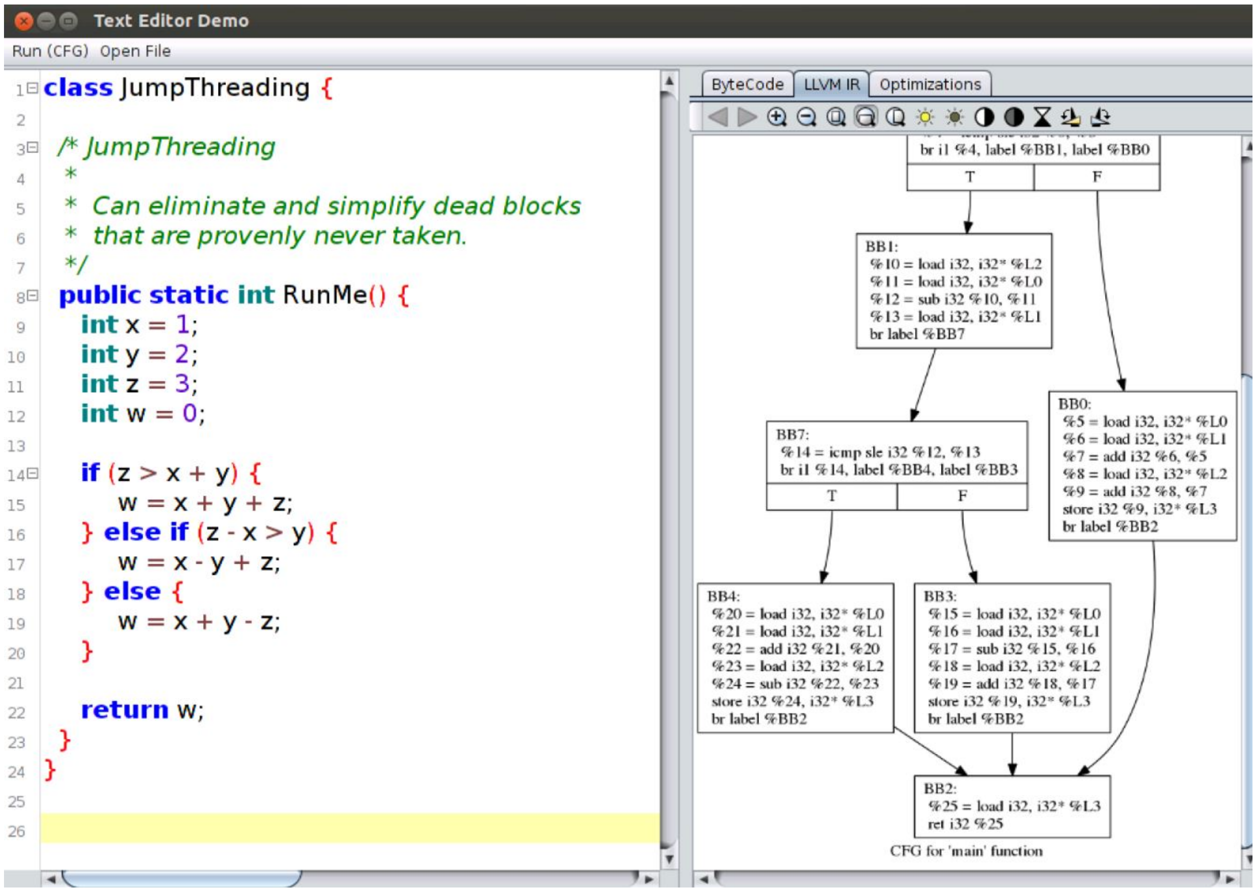


```
class Addition {

  public static int RunMe() {
    int n = 1023;
    int lastFactor = 0;
    for (int i = 2; i <= n; i++) {
      while (n % i == 0) {
        lastFactor = i;
        n /= i;
      }
    }
    return lastFactor;
  }
}
```

```
entry:
 br label %BB0.preheader

BB0.preheader:
 %L2.010 = phi i32 [ 2, %entry ], [ %5, %BB3 ]
 %L1.09 = phi i32 [ 0, %entry ], [ %L1.1.lcssa, %BB3 ]
 %L0.08 = phi i32 [ 1023, %entry ], [ %L0.1.lcssa, %BB3 ]
 %0 = srem i32 %L0.08, %L2.010
 %1 = icmp eq i32 %0, 0
 br i1 %1, label %BB2.preheader, label %BB3

BB2.preheader:
 br label %BB2

BB2:
 %L0.16 = phi i32 [ %2, %BB2 ], [ %L0.08, %BB2.preheader ]
 %2 = sdiv i32 %L0.16, %L2.010
 %3 = srem i32 %2, %L2.010
 %4 = icmp eq i32 %3, 0
 br i1 %4, label %BB2, label %BB3

BB3:
 %L0.1.lcssa = phi i32 [ %L0.08, %BB0.preheader ], [ %2, %BB2 ]
 %L1.1.lcssa = phi i32 [ %L1.09, %BB0.preheader ], [ %L2.010, %BB2 ]
 %5 = add i32 %L2.010, 1
 %6 = icmp sgt i32 %5, %L0.1.lcssa
 br i1 %6, label %BB1, label %BB0.preheader

BB1:
 ret i32 %L1.1.lcssa

CFG for 'main' function
```

Figure 6: Intermediate Representation (Optimized)

Furthermore, to demonstrate the power that LLVM has to offer, next we use a program that is not irreducible.

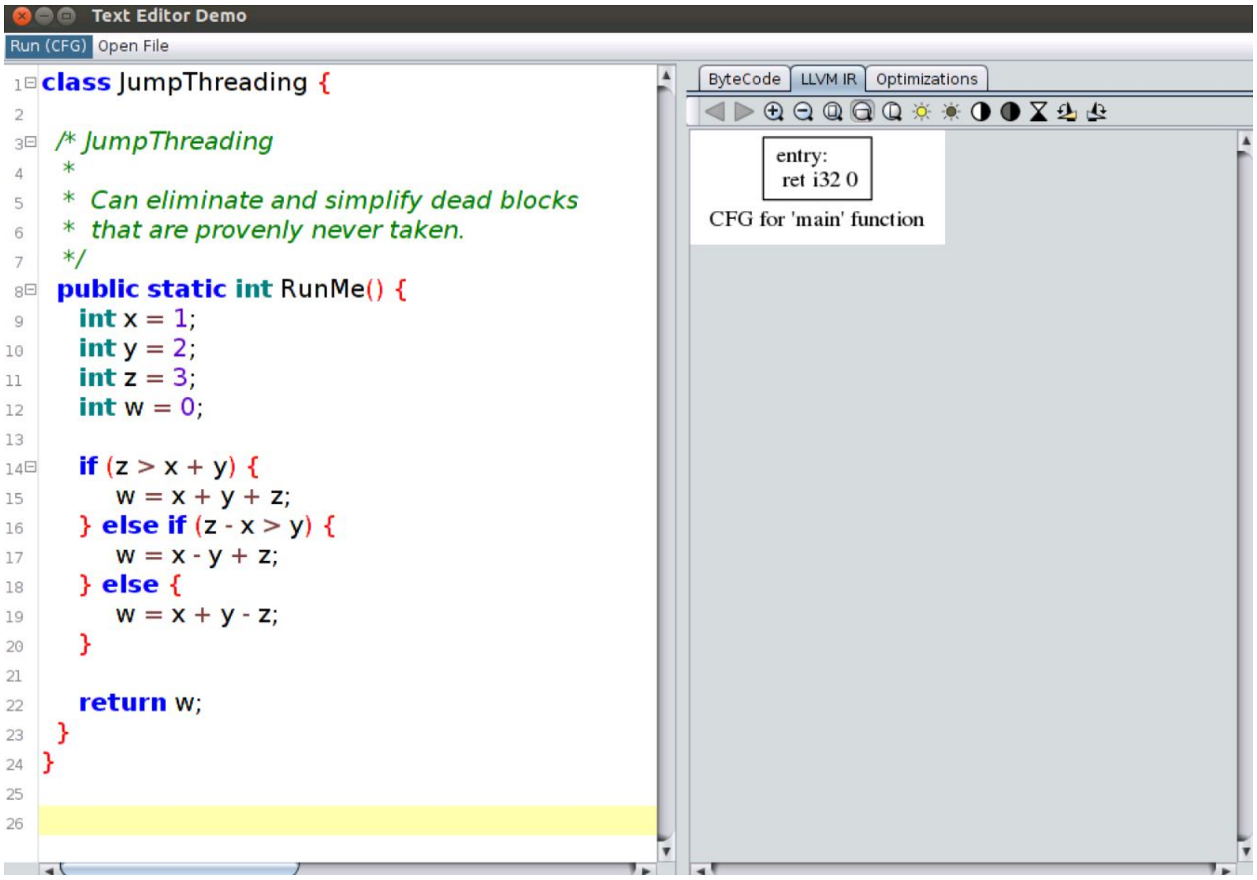


```
class JumpThreading {

 /* JumpThreading
  *
  * Can eliminate and simplify dead blocks
  * that are provenly never taken.
  */
 public static int RunMe() {
   int x = 1;
   int y = 2;
   int z = 3;
   int w = 0;

   if (z > x + y) {
       w = x + y + z;
   } else if (z - x > y) {
       w = x - y + z;
   } else {
       w = x + y - z;
   }

   return w;
 }
}
```

```
br i1 %4, label %BB1, label %BB0

BB1:
 %10 = load i32, i32* %L2
 %11 = load i32, i32* %L0
 %12 = sub i32 %10, %11
 %13 = load i32, i32* %L1
 br label %BB7

BB0:
 %5 = load i32, i32* %L0
 %6 = load i32, i32* %L1
 %7 = add i32 %6, %5
 %8 = load i32, i32* %L2
 %9 = add i32 %8, %7
 store i32 %9, i32* %L3
 br label %BB2

BB7:
 %14 = icmp sle i32 %12, %13
 br i1 %14, label %BB4, label %BB3

BB4:
 %20 = load i32, i32* %L0
 %21 = load i32, i32* %L1
 %22 = add i32 %21, %20
 %23 = load i32, i32* %L2
 %24 = sub i32 %22, %23
 store i32 %24, i32* %L3
 br label %BB2

BB3:
 %15 = load i32, i32* %L0
 %16 = load i32, i32* %L1
 %17 = sub i32 %15, %16
 %18 = load i32, i32* %L2
 %19 = add i32 %18, %17
 store i32 %19, i32* %L3
 br label %BB2

BB2:
 %25 = load i32, i32* %L3
 ret i32 %25

CFG for 'main' function
```

Figure 7: Intermediate Represention (Unoptimized)

After applying '-03', we see something more interesting.



```
class JumpThreading {

 /* JumpThreading
  *
  * Can eliminate and simplify dead blocks
  * that are provenly never taken.
  */
 public static int RunMe() {
   int x = 1;
   int y = 2;
   int z = 3;
   int w = 0;

   if (z > x + y) {
       w = x + y + z;
   } else if (z - x > y) {
       w = x - y + z;
   } else {
       w = x + y - z;
   }

   return w;
 }
}
```

```
entry:
 ret i32 0
```

CFG for 'main' function

Figure 8: Intermediate Represention (Optimized)

Many various optimizations and effects can be observed. By selecting your own sequence of data-analysis and transformations, you get varying and interesting results.

## Conclusion

The end result of this research is a very interesting proof-of-concept Java Virtual Machine with very basic Just-In-Time compilation support.We provided a linear-time algorithm of mapping the Java Virtual Machine's bytecode to a control flow graph, and from the control flow graph to LLVM's intermediate representation. We explored various optimizations that are available at little cost other than providing a frontend, and we provide a GUI for convenience.

While majority of time has been invested into becoming familiar with LLVM and the Java Virtual Machine, surprisingly little time has been required to construct the frontend itself. The frontend is a very naive and simplistic model, yet the LLVM optimizer is still able to do its job. This shows that, once you get past the learning-curve, LLVM becomes an invaluable tool for speeding up development time for any language.

## References

1. T. Lindholm GB F. Yellin, Buckley A (2015) The java virtual machine specification

2. (2012) HS-java, hackage api. https://hackage.haskell.org/package/hs-java

3. (2017) LLVM-jvm, github repository. https://github.com/LouisJenkinsCS/LLVM-JVM

4. (2012) LLVM-hs, hackage api. https://hackage.haskell.org/package/llvm-hs

5. (2017) LLVM language reference. https://llvm.org/docs/LangRef.html

6. (2012) MateVM, github repository. https://github.com/MateVM/MateVM

7. N. Ramsey J. Dias, Jones S (2010) Hoopl: A modular, reusable library for dataflow analysis and transformation. ACM SIGPLAN Notices 45:121. doi: 10.1145/2088456.1863539

8. (2012) Harpy, hackage api. https://hackage.haskell.org/package/harpy

9. Chaitlin G (1982) Register allocation & spilling via graph coloring. SIGPLAN '82 Proceedings of the 1982 SIGPLAN symposium on Compiler construction 98–105. doi: 10.1145/800230.806984

10. Ramalingam G (2002) On loops, dominators, and dominance frontiers. ACM Transactions on Programming Languages and Systems (TOPLAS) 24:455–490. doi: 10.1145/570886.570887