MASARYK UNIVERSITY
FACULTY OF INFORMATICS

Java bytecode disassembler

DIPLOMA THESIS

Bc. Jozef Cel'uch

Brno, Spring 2015

Declaration

Hereby I declare, that this paper is my original authorial work, which I have worked out by my own. All sources, references and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Bc. Jozef Cel'uch

Advisor: Mgr. Marek Grác, Ph.D.

Acknowledgement

This work would not be possible without the support of my parents for which I am very grateful. I would like to thank my supervisor Mgr. Marek Grác, Ph.D., for his advice and consultation during the writing of this thesis. And many thanks go also to Claudia who always stubbornly believed that I could finish.

Abstract

The aim of this thesis is to create and describe a working prototype of a bytecode disassembler that can recreate a possible Java source code from the provided bytecode. The tool is able to present both representations of the program side by side in a GUI. On top of that, this work describes the process of compilation to bytecode with the focus on the way of passing operands to instructions.

Keywords

Java, bytecode, decompilation, ASM, Eclipse RCP

Contents

List of Tables

List of Figures

Listings

1 Introduction

Since the creation of the platform in 1995, Java has spread over the wide range of different devices. Ranging from embedded devices, through mobile and desktop computers to servers. The wide applicability of the Java language stems from the fact that it runs on a virtual machine called Java Virtual Machine (JVM). This allows to write platform-independent applications that can run on any platform that is supported by the JVM.

A source code written in Java is compiled to a lower-level language called *bytecode* before it can be executed on the JVM. The standard Java compiler *javac* [7] uses a simple compilation strategy and it performs very few optimizations on the compiled code [9].

The aim of this thesis is to create a tool that can be used to recreate a possible Java source code which can then be viewed alongside the provided bytecode. In order to do that, it is necessary to understand the basic principles of compilation process in Java and also the basic structure of bytecode instructions and their operands. For this reason, an introduction to bytecode and JVM is also a part of the thesis.

Note that this program should be considered mainly as a tool used to improve the understanding of bytecode and compilation process. Therefore, its scope is limited to the *class files* that contain debug information and are compiled with the standard Java compiler *javac*. The tool also focuses only on the latest versions of Java Standard Edition, namely versions 7 and 8. A description of some of the new language features is also a part of this thesis.

1.1 Terminology

This section briefly describes the terms that are used throughout the whole thesis. These expressions can be found in the Java Language Specification[20] and Java Virtual Machine Specification [23].

- `class` *file* compiled code to be executed by the JVM, represented in a hardware- and system-independent binary format
- *bytecodes* instructions for the JVM

- *Constant Pool* table that contains symbolic information about classes and variables which is referred to by the bytecodes
- *opcode* a part of the bytecode instruction which specifies the operation to be performed

For the purposes of this thesis, the terms *disassembly, disassembler* and *decompilation, decompiler* will be used interchangeably and will denote the reverse process to the *javac* compiler process, that is to recreate the Java source code from the provided class file. Therefore, unless specified otherwise, the term *disassembler* will not be used to refer to the *javap*[8] tool which is also known as the "Java class file disassembler". This tool is used to transform class file from binary to human-readable text format.

In Java terminology terms *compilation* and *compiler* can be used for compilation of Java source code to bytecode. Alternatively, they are also used for the compilation of bytecode to native code of the hosting platform that is performed by the JVM[25]. For this reason the compilation performed on the JVM will be always referred to as *just-in-time compilation* and the program performing it will be called *just-in-time (JIT) compiler*.

1.2 Thesis organization

The second chapter of this thesis provides information on JVM, what it is and how it works. In addition to that, descriptions of `class` file and its parts as well as bytecode are provided. A more detailed account of instruction operands concludes this chapter.

In the third chapter, there is a brief overview of the basic principles of compilation and decompilation process. Based on these general ideas, this section provides a more in-dept description of the principles and tools used specifically in Java. The section ends with an overview of existing open-source Java decompilers and bytecode manipulation libraries.

Section four introduces the proposed implementation of the developed decompiler prototype. It contains an in-depth description of the created classes that are used to represent the Java code elements as well as the classes that contain the logic and drive the decompi-

lation process. The chapter concludes with a short description of the graphical user interface (GUI).

Since the objective of the created prototype is to disassemble bytecode and generate Java code, the fifth part provides examples of a few of the Java language structures along with their bytecode. It aims to show some of the new features added to Java in version 7 (and 8), how they are compiled to bytecode and also the Java code created by this disassembler.

The concluding chapter of this thesis evaluates the achieved results of the created tool, its advantages and shortcomings and also some possible use-cases. On top of that, there are ideas of some possible future enhancements and updates.

2 Introduction to JVM

This chapter provides an introduction to the inner workings and structure of JVM. Tightly tied to the specification of the JVM is the definition of class file; this chapter provides therefore basic information about its structure.

2.1 What is JVM?

The Java SE 8 Edition of The Java Virtual Machine Specification defines the JVM as "the component of the technology responsible for its hardware- and operating system-independence, the small size of its compiled code, and its ability to protect users from malicious programs" [23].

It is an abstract machine that is only defined by the specification[23] and there are many implementations of it[1]. However, there is an implementation that is the core component of the Java SE platform, this virtual machine is called HotSpot VM (it identifies at runtime parts of code that are called the most and only optimizes those, these parts are called "hot spots"[13]).

JVM specification describes the virtual machine in terms of data types, memory areas, frames and instructions. It also defines the behavior of exceptions and special methods [2]. A very important role in the working of the JVM plays the class file which is described in the section 2.2.

2.1.1 Data types

JVM supports three primitive data types: *numeric types*, `boolean` type, `returnAddress` type, and three reference types. Their structure can be seen in the figure 2.1 from Inside the Java Virtual Machine by Bill Venners[37]

Numeric types consist of *integral* and *floating-point* types. Integral types are `byte`, `short`, `int`, `long` and `char`. Except for `char`,

1. http://en.wikipedia.org/wiki/List_of_Java_virtual_machines
2. instance initialization method called <init> and class or interface initialization method called <clinit>

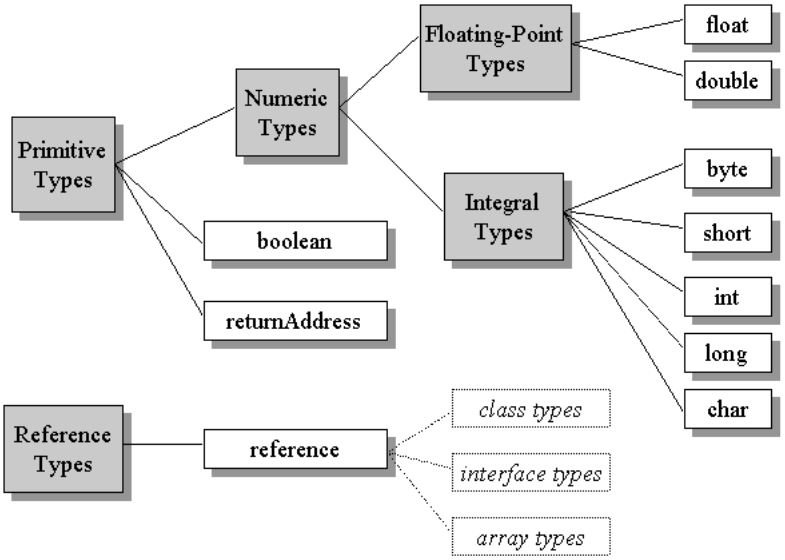

Figure 2.1: Data types of the JVM, Inside the Java Virtual Machine by Bill Venners[37]

which is a 16-bit unsigned integer that represents UTF-16 encoded Unicode character, all integer types are signed two's-complement integers with the default value of zero. Floating-point types are `float` and `double`.

JVM does not provide any dedicated instructions for operations on `boolean` type. Instead, `true` and `false` values of this type are compiled to 1 and 0, respectively, and are handled as `int` values. However, JVM does support `boolean` arrays directly. More information about instruction set is in the section 2.1.4 of this chapter.

`returnAddress` type is used by the JVM as a pointer to the opcodes of JVM instructions and does not correspond to any Java programming language type, nor can it be modified by the running program.

A reference type can be one of these three types: class, array or interface. Its value references to a dynamically created class instance,

array, or an class or array instance implementing an interface, respectively. Default value for reference type is `null` which has initially no run-time type but can be cast to any type.

2.1.2 Frames

Frames are used to hold method-specific data and partial results, they also perform dynamic linking, return values for methods and dispatch exceptions.

Local variables are stored in the frame in an array indexed from zero. Instructions can access them through the index in this array. Variables of type `long` and `double` occupy two places in this array and are addressed using the lesser index, all the other types occupy one position. Local variables are used to pass parameters on method invocation. For invocation of instance methods, in the first parameter (position zero), there is always a reference to the instance on which the method is being invoked (in Java it is the keyword `this`). Static, or class methods, do not have any fixed variable in the first position, therefore the parameters passed on to the method are indexed from zero. It is worth noting that the length of the local variable array is determined at compile-time and supplied in the class file (2.2).

Operand stack is a last-in-first-out (LIFO) stack contained in every frame, the depth of which is determined at compile-time and provided in the class file (2.2). The stack is empty when the frame is first created and it is used to load constants and local variables or store partial results of some instructions. Values on the stack are then used as arguments of other instructions. JVM is stack-based because all operands are passed on to the instructions through the operand stack. Another approach would be a register-based virtual machine, which could result in a faster-executing machine with a bigger code-size of both the VM and bytecode[31]. A more detailed account of how instructions modify the operand stack is provided later in this chapter.

2.1.3 Run-time data areas

JVM specifies six memory areas that are used during execution of the program, *the pc register*, *JVM stacks*, *native method stacks heap*,

method area, *run-time constant pool*. Some are created when the JVM starts and destroyed when it exits, others are specific per thread.

The pc register (or program counter) contains an address of the instruction that is currently being executed. But it is not defined in case the method is native [3]. This register is specific per thread since a thread always executes the code of one method at a time.

JVM stacks hold local variables and partial results and play a part in the method invocation and return. The stacks hold frames and are never accessed directly, except to push or pop frames. The specification does not impose any restrictions on the size of the stack and its memory does not have to be continuous either. Each thread has its own stack.

Native method stacks are an optional feature of a JVM implementation and they provide the stack capabilities for the native methods. Implementations that cannot load native methods or do not rely on conventional stacks do not need to provide this feature. If provided, these stacks are usually allocated per thread.

Heap is the data area from which memory for all class instances and arrays is allocated. JVM does not provide any instructions to deallocate objects, instead, JVM uses an automatic storage management system called *garbage collector*, the specification does not enforce any specific implementation of the garbage collector. Heap is created on JVM start-up.

Method area stores per-class structures, such as run-time constant pool, field and method data and the code for methods and constructors, including the special methods. This area is shared among all JVM threads.

Run-time constant pool is an area that corresponds to the constant pool table that is present in a class file. As well as constants of the class or interface, it contains references to methods and fields and is used when JVM searches for methods or fields. It is present in the method area and it is created for each class and interface.

3. a method implemented in platform-dependent code, typically written in another programming language such as C

2.1.4 Instruction set

Each bytecode instruction is encoded as a byte-long unsigned integer, therefore the maximum number of possible instructions is 256. Every instruction has a mnemonic textual representation which describes what the instruction does. The following sections shortly describe all available instructions.

2.1.4.1 Arithmetic instructions

Families of instructions that do the same operation for different types usually follow the pattern *<T><operation>* where T is the type. Instructions for arithmetic operations, provided in table 2.1 follow this pattern and thanks to their mnemonic representation they are very easy to understand.

Operation \ Type	`int`	`long`	`float`	`double`
Addition	iadd	ladd	fadd	dadd
Subtraction	isub	lsub	fsub	dsub
Multiplication	imul	lmul	fmul	dmul
Division	idiv	ldiv	fdiv	ddiv
Remainder	irem	lrem	frem	drem
Negation	ineg	lneg	fneg	dneg
Bitwise AND	iand	land		
Bitwise OR	ior	lor		
Bitwise XOR	ixor	lxor		
Arithmetic shift right	ishr	lshr		
Arithmetic shift left	ishl	lshl		
Logical shift right	iushr	lushr		
Local variable increment	iinc			
Comparison		lcmp	fcmp{l,g}	dcmp{l,g}

Table 2.1: Bytecode instructions for arithmetic operations

These arithmetic instructions, with the exception of `iinc`, are binary operations. They take two values from the top of the stack and push back the result. The `iinc` instruction, that increments a value

to a local variable, does not pop any values from the stack. All the necessary information is provided to it by compiler at compile time.

2.1.4.2 Load and store instructions

Another group of instructions that provide a specific version for each type are instructions for loading and storing values, provided in table 2.2. There are specific instructions for array items and local variables.

Op / Type	Array load	Array store	Load var	Store var
`byte`	baload	bastore		
`short`	saload	sastore		
`int`	iaload	iastore	iload	istore
`long`	laload	lastore	lload	lstore
`float`	faload	fastore	fload	fstore
`double`	daload	dastore	dload	dstore
`char`	caload	castore		
`ref`	aaload	aastore	aload	astore

Table 2.2: Bytecode instructions for data operations

Instructions `<T>load_<n>` and `<T>store_<n>`, where T can be of type `int`, `long`, `float`, `double`, or reference denote a group of load and store instructions that encode their operand in them. The number *n* can range from 0 to 3 and represents the local variable that is to be loaded onto the stack. More general versions of these instructions (`<T>load` and `<T>store`) provide the same functionality but in a less efficient way as they contain the variable number as an *implicit operand*[4] provided at compile-time.

There are a few instructions that are used for loading constants. `Bipush` and `sipush` are used to load `byte` (8-bit signed) and `short` (16-bit signed) constants, respectively.

For pushing a one-word constant (`int`, `float` or `String`) on the stack, the instruction `ldc` is used. There are two other modifications

4. operand provided at compile time as a value or a constant pool reference

of this instruction, `ldc_w` which uses a 16-bit index instead of the 8-bit used by `ldc` and `ldc2_w` which is used for loading constants of types `long` and `double`.

And finally it is a group of instructions that push a specific constant onto the stack. They follow the format `<T>const_<n>`. Possible types for these instructions are `int`, `float`, `long` and reference (in this case the n represents `null`).

2.1.4.3 Type conversion instructions

Next group are type conversion instructions, they allow conversion between numeric types of JVM and they follow the *<T1>2<T2>* naming convention where T1 and T2 are types and number 2 is used in the sense of "to". Conversions can be divided into two categories, *widening* (table 2.3) and *narrowing* (table 2.4) instructions. It is worth noting here that there are no instructions for widening `byte`, `char` and `short` types to `int` as these types are all implicitly widened to `int` by the JVM. For a more detailed account of how the instructions perform these operations please refer to the chapter 2 of the JVM specification[24].

Type	`long`	`float`	`double`
`int`	i2l	i2f	i2d
`long`		l2f	l2d
`float`			f2d

Table 2.3: Instructions for widening numeric conversions

Type	`byte`	`char`	`short`	`int`	`long`	`float`
`int`	i2b	i2c	i2s			
`long`				l2i		
`float`				f2i	f2l	
`double`				d2i	d2l	d2f

Table 2.4: Instructions for narrowing numeric conversions

2.1.4.4 Object creation and manipulation

To create a new class instance, the instruction `new` is used. For creating arrays, however, there are three distinct instructions: `newarray` (arrays of numbers or booleans), `anewarray` (arrays of objects) and `multianewarray` (multi-dimensional arrays). JVM also contains a specific instruction called `arraylength` that is used to get the length of an array (in case of multi-dimensional arrays it is the length of the first dimension).

Access to instance fields is provided through a pair of instructions `getfield` and `putfield` for loading and storing values to fields, respectively. Analogically, there is a similar pair of instructions `getstatic` and `putstatic` for accessing static fields.

JVM specification also defines a pair of instructions to check the type of object currently loaded on top of the operand stack. The first instruction, `instanceof` behaves as a test which pushes 1 to the stack if the reference on top of the stack belongs to the required type (or subtype) or implements the required interface, otherwise it pushes 0. Instruction `checkcast`, on the other hand, throws a `ClassCastException` if the object on top of the stack is not an instance of the specified class. These instructions also differ in the way they handle `null` references, the former fails if the reference is `null`, whereas the latter passes. This is due to the fact that `null` can be cast to any type but it is not an instance of any class.

2.1.4.5 Operand stack management

The instructions `pop`, `dup` and `swap` give the program that is being executed on the JVM a way to directly modify the operand stack. All their alternatives and explanations are given in the form of examples in figure 2.2 (top of the stack is denoted by the arrow "$\rightarrow$").

2.1.4.6 Exception throwing

Some JVM instructions (such as `checkcast`) throw an exception if they detect an abnormal condition. Exceptions can also be thrown programmatically with the `throw` statement. This statement is translated directly to `athrow` instruction which expects a reference to an

- `pop` ..., value2, value1→
..., value2→
- `pop2` ..., value3, value2, value1→
..., value3→
- `swap` ..., value3, value2, value1→
..., value3, value1, value2→
- `dup` ..., value2, value1→
..., value3, value1, value1→
- `dup_x1` ..., value2, value1→
..., value1, value2, value1→
- `dup_x2` ..., value3, value2, value1→
..., value1, value3, value2, value1→
- `dup2` ..., value2, value1→
..., value2, value1, value2, value1→
- `dup2_x1` ..., value3, value2, value1→
..., value2, value1, value3, value2, value1→
- `dup2_x2` ..., value4, value3, value2, value1→
..., value2, value1, value4, value3, value2, value1→

Figure 2.2: Stack manipulation instructions

exception object on the stack.

2.1.4.7 Method invocation

There are various instructions to invoke different types of methods. All of them receive their method descriptor (the information about method in the constant pool 2.2.1)) at compile time as their operand.

`Invokestatic` invokes a static method and pushes the result on the stack. Method arguments are already prepared on the stack and they are popped and used by this instruction. An example of a method call with this instruction is in listing 2.1. Note that this instruction does not require reference to any instance because it invokes a class method.

Listing 2.1: Instance method with static method invocation

```
method(Ljava/lang/String;)Ljava/lang/Integer;
  ALOAD 1
  INVOKESTATIC java/lang/Integer.valueOf
          (Ljava/lang/String;)Ljava/lang/Integer;
  ARETURN
```

`Invokespecial` invokes instance initialization methods (`<init>`), private instance methods and methods of the superclass. The instance on which the method is being invoked is pushed on the operand stack first followed by method arguments. An example is provided in listing 2.2, which shows a creation of an `Integer` instance from an `int` constant. Note the use of `dup` instruction which duplicates the reference to the new `Integer` instance on top of the stack. This is done in order to have a reference to the newly created and initialized object on top of the stack after the `invokespecial` instruction finishes.

Listing 2.2: Creation of a new instance with constructor invocation

```
  NEW java/lang/Integer
  DUP
  ICONST_1
  INVOKESPECIAL java/lang/Integer.<init> (I)V
```

`Invokeinterface` invokes interface methods. The implementation is resolved at runtime and it is the object that is below the method arguments on the operand stack. In case the example in listing 2.3, the instance is passed as an argument of the enclosing method.

Listing 2.3: Invocation of an interface method

```
method(Ljava/util/List;)Ljava/util/List;
  ALOAD 1
  ICONST_3
  ICONST_4
  INVOKEINTERFACE java/util/List.subList
          (II)Ljava/util/List;
  ARETURN
```

`Invokevirtual` invokes instance methods except for the ones invoked by the instructions above. The instruction behaves in the same way as `invokeinterface`, as can be seen in the listing 2.4.

Listing 2.4: Invocation of an interface method

```
method(Ljava/lang/String;)I
  ALOAD 1
  ICONST_3
  ICONST_4
  INVOKEVIRTUAL java/lang/String.codePointCount (II)I
  IRETURN
```

`Invokedynamic` invokes methods linked dynamically at runtime. It was added in Java SE 7 to support dynamically typed languages[30] which led to addition of *Lambda expressions* in Java SE 8 [19]. As it is a relatively new instruction, it is described in more detail in section 5.2.

2.1.4.8 Control flow

Next big group of instructions are flow control instructions. These are used to conditionally or unconditionally cause JVM to continue execution with an instruction other than the one following. There are three smaller groups and they are described separately in the following tables.

First group are the unconditional jump and subroutine instructions, shown in table 2.5. The subroutine instructions were used for exception handling to implement the `try-finally` construct. Note that according to the JVM specification [23], instructions `jsr` and `ret` (and their wide alternatives) cannot be used in any class file of version 51.0 (Java SE 7) and higher. Therefore, they are only supported for the backward compatibility.

Instruction	Description
`goto`	Jump to address
`goto_w`	Jump to address using wide offset
`jsr`	Jump to subroutine
`jsr_w`	Jump to subroutine using wide offset
`ret`	Return from subroutine
`ret_w`	Return from subroutine using wide offset

Table 2.5: Unconditional jumps and subroutines

Second group are conditional jumps contained in table 2.6. These

instructions either jump to a different location in code or let the execution continue normally depending on the values that are on top of the stack.

Instruction	Jump condition
`ifeq`	value == 0
`ifne`	value != 0
`iflt`	value < 0
`ifle`	value <= 0
`ifgt`	value > 0
`ifge`	value >= 0
`ifnull`	value == null
`ifnonnull`	value != null
`if_icmpeq`	integer1 == integer2
`if_icmpne`	integer1 != integer2
`if_icmplt`	integer1 < integer2
`if_icmple`	integer1 <= integer2
`if_icmpgt`	integer1 > integer2
`if_icmpge`	integer1 >= integer2
`if_acmpeq`	reference1 == reference2
`if_acmpne`	reference1 != reference2

Table 2.6: List of conditional jumps

Last two instructions falling into the control flow category are `tableswitch` and `lookupswitch`. Both, as their names suggest, are used to implement `switch` statements. More detailed description and examples for these two instructions can be found in section 5.1.

2.1.4.9 Synchronization

To conclude this section, there are two instructions that provide synchronization, `monitorenter` and `monitorexit`. Together with a `try--finally` block,they are used to implement the `synchronized` block of the Java language.

This section provided only a brief overview of the instructions available on JVM platform. More information and examples of how some of the instructions are used by `javac` is in the chapter 5.

2.2 Class file

Class file is the single input file that is provided to a JVM when a class is being executed. All the information that JVM needs to execute the class is contained within the single class file. Additional class files are loaded as necessary when the class contains calls to static methods or creates instances of other classes.

The list below summarizes all fields specified in the class file together with their descriptions.

- *magic*: The magic constant `0xCAFEBABE`, every class file must start with these 4 bytes.
- *minor_version, major_version*: Numbers that together determine the version of the class file format (in the form `major_version.minor_version`). Each new version of JVM is mostly backward compatible with the previous version[10].
- *constant_pool_count*: Number of items in the constant pool table plus one.
- *constant_pool[]*: Heterogeneous array that contains class specific information such as names of classes, methods and fields but also string and integer constants. It is described in more detail in section 2.2.1 below.
- *access_flags*: Mask of flags that indicates the permissions and properties of the class or interface.
- *this_class*: Name of this class, stored as an index of a `CONSTANT_Class_info` structure in the constant pool table.
- *super_class*: Name of the direct superclass, stored as an index of a `CONSTANT_Class_info` structure in the constant pool table. It can be zero, in which case the class must represent the class `Object` which is the only class or interface that has no direct superclass.
- *interfaces_count*: Number of direct superinterfaces of this class or interface

- *interfaces[]*: Array of indexes of `CONSTANT_Class_info` structures in the constant pool that represent the interfaces of this class.
- *fields_count*: Number of both class and instance fields in the class
- *fields[]*: Array of `field_info` (2.2.2) structures representing the fields declared by this class or interface, it does not contain the inherited fields
- *method_count*: Number of all methods declared in this class or interface
- *methods[]*: Array of `method_info` (2.2.2) structures that keeps all the information about a method declared in this class or interface including instance methods, class methods but also instance and class initialization methods. The array does not store methods inherited from the superclass or interface.
- *attribute_count*: Number of attributes defined in this class
- *attributes[]*: Array of `attribute_info` (2.2.2.1) structures

2.2.1 Constant pool

Constant pool is the structure that contains all class and interface names, field names and constants that are referred to by instructions and other structures within the class file. All entries in the constant pool have the same general structure shown in listing 2.5.

Listing 2.5: General constant pool item structure

```
cp_info {
    u1 tag;
    u1 info[];
}
```

The 1-byte tag at the beginning of each item specifies the type of the constant pool entry and it is the only required field in all entry types. Based on the tag, the structures follow the naming convention `<tag>_info` and each specifies its own fields. That is why in the general structure `cp_info` the second field is defined as a general array

of bytes. All constant pool tags are listed in figure 2.3 together with the usage of constant pool entry types they represent.

CONSTANT_Utf8 String encoded in modified UTF-8 format

CONSTANT_Integer 4-byte `int` constant

CONSTANT_Float 4-byte `float` constant

CONSTANT_Long 8-byte `long` constant

CONSTANT_Double 8-byte `double` constant

CONSTANT_Class Class or interface name encoded in internal form

CONSTANT_String Contents of a `String` object

CONSTANT_Fieldref Symbolic reference to a field

CONSTANT_Methodref Symbolic reference to a method

CONSTANT_InterfaceMethodref Symbolic reference to an interface method

CONSTANT_NameAndType Method or field name and type descriptor

CONSTANT_MethodHandle Unresolved method handle

CONSTANT_MethodType Unresolved method type

CONSTANT_InvokeDynamic Information required by the `invokedynamic` instruction to load a method dynamically

Figure 2.3: Constant pool tags and their usage

To minimize the size of the constant pool table the structures only store references to other constant pool structures so everything is only defined once. For instance a `CONSTANT_Fieldref_info` contains, apart from the tag, two additional fields called `class_index` and `name_and_type_index` which refer to a `CONSTANT_Class_info` and a `CONSTANT_NameAndType_info`, respectively. Thanks to this approach, the size of the `CONSTANT_Fieldref_info` is only 5 bytes.

2.2.2 Remaining class file structures

The constant pool contains data which is then referenced from every other part of the class file. Parts that are missing in the description of class file are *Fields*, *Methods* and *Attributes*.

Fields and methods represent their respective Java counterparts. Fields contain all the information about static and instance fields.

Methods hold together information about all methods (including class and instance initialization methods). Their general structure is virtually identical, but these structures differ in the attributes they store.

2.2.2.1 Attributes

Attributes can be seen as named pieces of data that appear in class files. There are 23 predefined attributes in Java SE 8 and they can be a part of structures such as `ClassFile`, `Method`, `Field` and `Code` (which is an attribute itself). They follow the same general structure, provided in the listing 2.6.

Listing 2.6: General attribute item structure

```
attribute_info {
    u2 attribute_name_index;
    u4 attribute_length;
    u1 info[attribute_length];
}
```

A correct implementation of Java virtual machine must support 5 attributes described in figure 2.4. From the remaining 18 attributes, 12 are critical to correct implementation of the class file by the class libraries of the Java SE 8 platform and remaining 6 are optional but can be useful for various tools [23]. Compilers can define their own attributes, and if the JVM cannot recognize them it should simply ignore them without throwing any exception.

ConstantValue Value of a constant expression (i.e. primitive type or a `String`)

Code JVM instructions and other auxiliary information for a method

StackMapTable Information used during the process of verification by type checking defined in section 4.10.1 of JVMS [23]

Exceptions Information that indicates which checked exceptions a method can throw

BootstrapMethods Specifiers of bootstrap methods referenced by `invokedynamic` instructions (more information on `invokedynamic` is in section 5.2)

Figure 2.4: Required attribute types

A knowledge of all predefined attributes is not necessary to understand the class file structure or process of decompilation. It suf-

fices to know that all bytecode instructions are in the `Code` attribute and a method that is not `native` or `abstract` must have exactly one `Code` attribute. A detailed account of all attributes is in the chapter 4.7 of JVM specification [23].

2.2.3 Instruction operands

Section 2.1.4 contains the instructions but a little information about how they get their operands. The following section summarizes all possible ways of providing operands to bytecode instructions.

The first and simplest possibility is that instructions do not take any operands whatsoever. For example, this is the case for instructions of the two following groups: `<T>const_n`, `<T>load_n`. Neither do they have any implicit operands nor do they take any operands from the operand stack. These instructions only push either a constant or a local variable to the stack and they have their values encoded in them.

Second possibility is that instructions do not have any operands and they only take values from the top of the operand stack, do something with them and (possibly) push back the result. This is the case for the opcodes representing arithmetic operations or type conversion instructions or instructions for manipulating arrays (`<T>astore`, `<T>aload`).

With the increasing complexity of the instructions come opcodes that have some implicit operands but also take additional operands from the stack. Perhaps the most natural example of this would be the instructions for method invocation, such as `invokevirtual`. This instruction is used to call instance methods. Therefore, its implicit operand is a reference to a constant pool entry of type `CONSTANT_Methodref`. This contains references to the owning class of the method and its name and type which effectively defines a method in a class. Operands it takes from the stack represent the method arguments. They are popped from the stack and passed on to the invoked method as its local variables. After all arguments are popped there is a reference to the instance on which the method is to be called. The instruction pops this reference too and calls the required method on it. Result of the method call is pushed on the stack after the method returns.

The final and the most complex group consists of instructions that have variable length. Examples of these are the instructions that encode the Java `switch` statement: `lookupswitch` and `tableswitch`. These instructions contain bytecode locations of cases as their implicit operands. In addition to that they take an item from the top of the stack and use it to decide which case block to choose. Closer description, as well as the differences between these two instructions are a part of the chapter 5.

3 Decompilation of Java programs

The following chapter provides information about the decompilation process in Java. Nevertheless, in order to grasp a firm understanding of that it is rather beneficial to have a basic understanding of the compilation process. Mainly because it is a very well-defined process, but also because compilation and decompilation share some common features [36].

3.1 Compilation in a nutshell

3.1.1 General compilation

Compiler is a computer program that translates programs from the source language to equivalent programs in the target language [1] which can then be executed by a computer. Another way of executing a program would be using a tool called interpreter, the purpose of which is to take the source code and execute the operations defined in it directly. This approach is usually much slower compared to the machine-language program generated by a compiler. Java combines both these approaches by first translating the source program to bytecode that is then executed on the JVM, more detail on this process is provided in section 3.1.2.

From the architectural point of view, compilers generally consist of two parts, *front end* and *back end*[15]. Compilation starts at the front end where the compiler processes the source code and ends at back end where the resulting target is produced. The whole process can be described in more detail by the following stages[1].

Lexical analysis During this phase the *scanner* reads source code as a stream of characters and converts them to a stream of tokens that are passed on to the second compilation stage. All meaningful sequences of characters are stored into a *symbol table*[1].

Syntax analysis The *parser* converts the stream of tokens to a tree-like intermediate representation called *syntax tree*. Each inte-

1. a data structure containing a record for every variable name and its attributes

rior node in this tree represents an operation and the children of the node represent the arguments of the operation.

Semantic analysis The *semantic analyzer* checks semantic consistency of the syntax tree and the symbol table with the language specification. Compiler checks types during this phase and can even perform type conversions that are allowed by the language specification.

Intermediate code generation A number of intermediate code representations of the source program can be generated during compilation. In this phase, a low-lever representation that can be easily translated to machine code is created.

Code optimization This phase attempts to improve the intermediate code in order to produce a better result in the target code. There is a great variety of possible optimizations that compilers can perform. Some compilers may leave out this phase entirely, such as `javac`.

Code generation Compiler generates the target code here by translating the intermediate code sequences to the sequences of machine instructions.

3.1.2 Compilation of Java programs

As was previously mentioned, `javac` compiles the Java source code to bytecode that is executed on the JVM. On one hand, this approach enables execution of platform-independent class files on different platform-specific virtual machines. On the other hand, however, the virtual machine causes a certain overhead which makes the execution slower compared to the languages compiled to machine code. To address this problem the Java HotSpot VM[26] uses so-called just-in-time (JIT) compilation which dynamically detects parts of the code that are executed the most (i.e. hot spots) and compiles them at run-time into native machine code[25]. Note that the JIT compilation is performed at run-time and is not in the scope of this thesis.

The compilation of Java classes using the Java compiler `javac` roughly conforms to the model presented in the previous section but there are also some significant differences. Whole process can be di-

vided into three top-level stages: *Parse and Enter*, *Annotation Processing* and *Analyse and Generate*[11].

3.1.2.1 Parse and Enter

This phase can be thought of as the front end. The compiler reads the set of the provided `*.java` source files and turns them into a stream of tokens (i.e. scanning) that is then transformed into a syntax tree. In the *enter* step of this phase the symbol table is created, which contains all symbols encountered in the source code together with their attributes.

3.1.2.2 Annotation processing

This phase is the preparation step for the compilation. Its goal is to scan the parsed and entered files for annotations, determine if annotation processing is needed and then invoke any appropriate annotation processors. As Java provides an API for custom annotation processing[2], this phase can cause the need to parse and enter another files required by the processed annotations. Therefore the runtime determines after each run of annotation processors if another round of annotation processing is required and repeats the process for the new files.

3.1.2.3 Analyse and Generate

The last phase of the Java compilation process is responsible for analysis of syntax trees and subsequent generation of class files. Before the code generation itself, a various analyses are performed on the syntax tree, such as name resolution, type checking, type parameter inference and constant folding. If these pass without any errors, they are followed by dataflow analysis. This includes checks for statement reachability and proper exception handling. In addition to that, there is assignment analysis to ensure that each variable is assigned when used and `final` variables are assigned only once. And finally, a local variable capture analysis which ensures that variables accessed in inner classes or lambdas are `final`. Before the code generation

2. JSR 269: Pluggable Annotation Processing API

itself, the compiler transforms the code that uses generic types to code without generic types and and also replaces the complex syntax tree structures with the simpler ones (i.e. processes the "syntactic sugar"). The compiler does not perform any optimizations of the created code after all the analyses and simplifications only transforms the resulting structures into bytecode and generates the resulting class files.

3.2 Decompiling Java bytecode

3.2.1 Decompilation in general

A decompiler is a program that attempts to reverse the process of a compiler, i.e. to create a program in a high-level language from a program in a low-level machine language[6]. Tasks and analyses performed by decompilers are very similar to those performed by compilers even though the process is essentially reversed.

On the input, a decompiler receives a binary file which needs to be decoded (or parsed) into a more general intermediate representation. Then it can perform various analyses and modifications on this representation, similar to those of a compiler. In the final stage, the high-level code representation is produced.

The task of decompilation is generally not an easy one. This is due to the fact that during compilation a program loses a lot of information that was present in the high-level language. Problems that decompilers need to solve can be divided into a number of categories[36]:

1. Separate code from data
2. Separate pointers from constants
3. Separate original from offset pointers
4. Declare data
5. Recover parameters and returns
6. Analyse indirect jumps and returns
7. Type analysis
8. Merge instructions

9. Structure loops and conditionals

A general machine code decompiler needs to be able to solve all of these problems. In order to do that it uses various analyses and heuristics[36]. In case of Java bytecode the decompilation is simpler due to the structure and information contained in the class file.

3.2.2 Decompilation of Java

To decompile a standard[3] class file to Java source code is relatively easy, at least compared to the decompilation from machine code. One reason is that the data is already separated from the code in the constant pool. Secondly, the references are already separated from constants due to the usage of distinct instructions for different types. Finally, it is the fact that the class file for one Java class is self-contained and retains a lot of information compared to other binary formats.

Even with the obvious advantages, class file decompilers still need to overcome some real challenges. From the problems presented in the previous section, there are the following: type analysis, merging of stack-based instructions into expressions and structuring of loops and conditionals.

General class files contain the type information about fields and parameters and return values of methods. Thanks to that the type analysis is needed only for local variables[4]. The JVM even contains specific instructions for different types, such as `int`, reference, `float`, `long` and `double`. Nevertheless, it does not distinguish between the types of `boolean`, `byte`, `short`, `char` and treats them as `int`. Therefore, a decompiler must be able to infer the specific subtype of integer and the type of the object that is referred to by the reference [18].

Code generation strategy used by `javac` is quite straightforward. Each simple statement is compiled into a series of bytecode instructions with the assumption that the Java evaluation stack is empty before and after the statement executes [28]. On one hand, this strategy makes the decompilation of simple statements compiled by `javac` fairly easy and is quite commonly used by a number of decompilers

3. compiled from a Java source file without optimizations or obfuscation
4. class files with debug information contain also `LineNumberTable`, `LocalVariableTable` and `LocalVariableTypeTable`

(see section 3.3). On the other hand, even basic optimizations make bytecode indecipherable for decompilers using this strategy.

A more general approach would be to create some intermediate representation of the bytecode and generate the Java code from that [35]. This technique can be used to decompile optimized class files or class files generated by different compilers.

The final problem that bytecode decompilers face is recreation of control flow constructs such as loops and conditionals but also exception handling and `switch` statements. In bytecode, every conditional branching is represented by `goto` jump to a specific position in bytecode (using additional structures in case of exceptions and switch statements). However, in Java, there are no unconditional jumps[5] and the control flow is organized by higher-level constructs that must be generated from the simple `goto` jumps. This is generally not an easy task and can be achieved by creating a control flow graph and reducing it down to recognizable patterns [22].

3.3 Overview of decompilers

Java decompilers have been present almost as long as the Java language itself. One of the first Java decompilers ever created was *Mocha* [32] and it appeared as early as 1996, one year after the release of Java. This decompiler, along with many others that appeared after it over the years is unmaintained and obsolete now. But it goes to show that decompilation of Java has been an interesting problem since the very beginning of the platform.

A survey released in 2003 [16] tests a group of decompilers available at the time. It uses a set of 9 tests, 6 of which are compiled by `javac`.The remaining 3 are either generated by a bytecode generation tool or by an optimizer to check how decompilers cope with arbitrary bytecode. Measuring the improvements of the decompilers in the original survey, an evaluation released in 2009 [22] uses the same tests and metrics to survey decompilers available in 2009. In addition to the original group it also tests two new decompilers that became available since the original survey. None of the tested tools was able to decompile all the tests correctly.

5. with the partial exception of `break` and `continue` statements

Many of the tested decompilers have been unmaintained for a long time. Therefore they cannot be used to decompile bytecode generated from the latest versions of Java programming language. On the other hand, since 2009 some new decompilers have been created. The following list contains decompilers that are available today. Only the first two of them took part in the surveys.

Dava [28] is a part of the Soot Java Optimization Framework [29]. The framework provides an API and a set of internal representations on top of which Dava is developed. Dava is able to generate Java code from arbitrary bytecode by using more general techniques than simple pattern matching [27]. It was present in both surveys and it was one of the decompilers with best results with 4 correctly decompiled programs. Due to its design and the fact that it attempts to reconstruct the control flow it outperformed the other decompilers with arbitrary class files. However, with class files generated by `javac` it was not as successful.

Java Decompiler (JD) is free for non-commercial use [14]. It was also present in the survey where it performed very well with the `javac` generated bytecode. This decompiler supports Java from version 5 up to version 7 with some known limitations. Lambda expressions or default methods added in Java 8 are not supported. Being written in C++, it allows fast decompilation and there are even plug-ins for popular IDEs such as Eclipse[6] and IntelliJ IDEA[7].

CFR (Class File Reader) is a free decompiler but its source is not publicly available yet [2]. The first version was released in 2011 so it is not included in either of the surveys. It is written entirely in Java 6 and it supports lambda expressions added in Java 8 or `switch` with strings added in Java 7.

Procyon [33] is an open-source decompiler written in Java 7. It is an experimental part of a suite of Java metaprogramming tools focused on code generation and analysis. Focusing on

6. https://eclipse.org/ide/
7. https://www.jetbrains.com/idea/

the class files generated by `javac` it can successfully decompile enums, `switch` statements with strings or enums, local classes, annotations and Java 8 lambda expressions. It shares test suite with CFR so it is expectable that their functionality will be very similar.

Fernflower "is the first actually working analytical decompiler for Java and probably for a high-level programming language in general" [34]. The decompiler has become a part of IntelliJ IDEA in the version 14. There is very little information available about it apart from the source code.

Krakatau [21] is a set of bytecode tools that contain a decompiler, a disassembler for Java class files and an assembler to create class files. It is written in Python 2.7. The decompiler takes arbitrary bytecode and generates a possible Java code. Thanks to this approach it can cope with obfuscated code. On the other hand, the generated code is not as readable as it would be with a pattern-matching approach for unobfuscated class files.

3.4 Bytecode manipulation libraries

Tools for bytecode manipulation are used for numerous purposes in JVM ecosystem. From various Java tools and applications to different JVM languages that can use them to provide additional language features.

3.4.1 ASM

ASM[8] is a Java bytecode manipulation framework designed to dynamically generate and manipulate Java classes [5]. The main idea of ASM is to be as fast and efficient as possible. This is achieved by using only a small number of classes compared to other tools. To perform fast transformations, it uses the *Visitor*[9] design pattern without

8. http://asm.ow2.org/
9. Allows to add new functionality to the existing object structure without the need to change the structure

the need to represent the visited class and bytecode instructions with objects. Visitors transform the visited code by changing call chains.

All the complexity of serialization and deserialization of class files is hidden from users of the API [4] and is done automatically. On top of that, ASM provides automatic management of constant pool, therefore users do not have to manipulate its indexes manually. `ClassReader` class makes a preliminary analysis of the code in order to detect jump instructions and it uses *labels* to mark jump destinations.

In addition to the event-based visitor approach the API also contains a *Tree API*. Using the tree representation of a class file takes more time and consumes more memory. On the other hand, it allows for making certain transformations more easily since the tree representation does not have any constraints regarding the order in which nodes are accessed.

3.4.2 Apache Commons BCEL

BCEL[10] (Byte Code Engineering Library) is a framework for creation, analysis and modification of binary Java class files [17]. Classes and all their attributes are represented as objects when the class file is deserialized from the byte array. All interaction with the loaded class file is done through the created objects. This approach provides a higher level of abstraction from the bytecode details.

The library provides serialization and deserialization of the class files. It also contains a verifier named *Justice* for the verification of the generated bytecode. Apart from that, it contains classes to represent every aspect of a class file from the top-level `JavaClass` to the classes representing each bytecode instruction.

3.4.3 Serp

Serp[11] is an open-source framework for manipulating Java bytecode [38]. It provides high-level APIs for all normal bytecode modification functionality along with a large set of convenience methods. Addi-

10. http://commons.apache.org/proper/commons-bcel/
11. http://serp.sourceforge.net/

tionally, it provides low-level APIs in order to provide the widest range of functionality possible.

Similarly to BCEL, it creates an object representation of the class file and all bytecode instructions. The difference is that Serp does not have a class for each instruction but rather it represents similar instructions with one class. This approach can lead to smaller memory consumption.

4 Description of the prototype

The main goal of this thesis was to create a working prototype of a decompiler that could be used to translate bytecode to a possible Java source code. Even with all the information contained in class files this is generally not a trivial task. Therefore, the scope of this work is limited to class files generated by the standard Java compiler *javac*. This restriction allows to use a simpler decompilation strategy based on matching of bytecode patterns to Java code.

Another simplification is that the decompiler prototype is intended to be used with class files that contain debug information. Thanks to that, there is no need to do any form of type inference since all type information is contained within the class file. This restriction, however, could be removed by adding a form of type inference mechanism.

The whole prototype consists of two separate parts, decompiler and graphical user interface (GUI). This chapter provides an overview of both parts separately. Main focus is on the decompiler itself since the GUI is very simple and pretty much self-explanatory.

4.1 Used libraries

To start the description, we first introduce libraries that were chosen to implement the tool.

4.1.1 ASM

ASM bytecode manipulation framework presented in the previous chapter (section 3.4.1) is used to transform bytecode to a higher-level abstraction. There is a number of reasons for this choice.

Firstly, the framework is under active development and it is used in various projects and number of JVM-based languages (such as Groovy[1] or Jython[2]). Which means that it is up to date with the newest bytecode versions.

1. http://groovy.codehaus.org/
2. http://www.jython.org/

Secondly, it is a relatively small library with a smaller number of classes compared to its competitors. This allows to create faster tools with smaller memory footprint.

The main advantage of ASM is the fact that it allows to work with class files on the fly and modify them using a visitor. Additionally, it also provides a visitor that constructs a tree representation of visited class file. This tree structure is very useful for reconstructing Java code from the bytecode. It already provides a certain level of abstraction from the underlying instructions so it simplifies the pattern matching procedure. On top of that, the tree structure enables additional movement to check what are the neighboring nodes. This can provide additional context for processing certain instructions (typically jump instructions).

Here it is also fair to note that using another library, such as BCEL, for this type of application would most likely yield very similar results. Therefore, the choice of using the ASM library is to some extent due to personal preference. This is based on the fact that ASM seems like a lower-level library and provides a better opportunity to learn more about bytecode. Additionally, according to the BCEL project website, "there hasn't been much development going on over the past few years" [17].

4.1.2 Testing libraries

A very important part of developing a computer program is testing. It enables to ensure with a certain level of confidence that the developed software works correctly. For this reason, testing was a vital part of the development process. Two frameworks were used to achieve this, JUnit[3] and JUnitParams[4]. Former is a very popular open source framework to write and run repeatable tests. The latter is a library built on top of JUnit that allows to write simple and readable parametrized tests.

3. http://junit.org/
4. https://github.com/Pragmatists/junitparams

4.1.3 Eclipse 4/RCP

Eclipse 4 is the latest release of the platform for building Eclipse-based tools and client desktop applications. It has a model-based user interface with declarative mechanism for application styling based on CSS[5]. These qualities make it easy to design and modify graphical user interfaces. The programming model is based on services. This provides a better modularity which is a key feature of the Eclipse framework. The platform also provides an annotation-based dependency injection framework which does not require the client to know how to access services. Instead, clients describe service they require and the platform provides it.

This approach seems ideal for making simple but also complex desktop applications. For this reason, it was chosen as the platform for creating the GUI. Even despite the steep learning curve caused by the loose coupling of components and the size of the platform.

4.2 Design overview

Design of the decompiler is partly based on the design of the ASM library. The `org.objectweb.asm.tree` package represents each part of the class file as a node in a tree. On the top level it is a `ClassNode` that provides an adapter for the class file and contains all the information from the processed class in its fields. Fields are stored in a list of `FieldNode` objects, methods in `MethodNode` objects and inner classes in `InnerClassNode` objects. All these nodes are processed by their respective classes of the decompiler, classes that extend the abstract `Block` class.

ASM tries to keep the number of classes low, therefore, not every bytecode instruction has its own class. Instead, a logical group of bytecode instructions is handled similarly, wrapped into a class that extends `AbstractInsnNode`. A similar approach was chosen for the decompiler classes that transform nodes into higher-level structures. All these classes extend an abstract `Expression` class and each specific subclass represents an expression in Java language.

Expressions generated from bytecode are further enclosed by classes

5. Cascading Style Sheets

extending an abstract `Statement` class. These statements represent the created Java code and are written out into a Java source file.

4.2.1 Expression

The `Expression` abstraction is created to process bytecode instructions and transform them into higher-level language structures. Each specific expression represents a certain Java code structure. The list in figure 4.1 summarizes all `Expression` subtypes in alphabetical order together with a brief description of what Java language expression they represent.

General `Expression` class contains only very basic information that is common for all expressions. It implements an interface called `Writable`. This interface provides a unified way for all code elements to be written out into Java code. Using this interface, expressions keep the responsibility for printing out relevant parts of themselves which simplifies the `Statement` abstraction.

Class diagram with class members can be seen in figure 4.2. Properties of the `Expression` class include expression type, cast type (if there is any), opcode of the bytecode instruction, line from the original source where the expression occurred and a flag `virtual` that indicates whether the expression should be converted to a `Statement` and written to Java code or not. An example of a virtual expression is a `PrimaryExpression` because a single constant or variable does not form a valid Java statement, or a general `UnconditionalJump` because Java does not allow a `goto` jump.

Bytecode instructions are turned into expressions with the help of a class called `ExpressionStack`. It is used to store expressions and facilitate their creation. Expressions have access to the stack through two methods that are called before and after expression is pushed onto the stack: `prepareForStack()` and `afterPush()`. More information about this process is in section 4.2.4.

4.2.2 Statement

`Statement` is a higher-level abstraction from the bytecode instructions than the `Expression` in the sense that in its most general version it represents a single valid line of Java code. Each statement con-

ArithmeticExpression a binary operation (addition, division, etc.)
ArrayAccessExpression loading of an array element
ArrayAssignmentExpression storing of value to a position in array
ArrayCreationExpression creation of all possible types of array
ArrayLengthExpression the access to the `length` field of an array
AssignmentExpression an assignment of an expression to a variable
BreakExpression the `break` keyword used in loops and switches
ConstantPrimaryExpression load of a constant
ConstructorInvocationExpression call of the constructor method
ContinueExpression the `continue` keyword used in loops
InstanceOfExpression the `instanceof` keyword
LambdaExpression the dynamic method invocation
LogicGateExpression the short-circuit operators `||(OR)` and `&&(AND)`
MethodInvocationExpression a call to a method
MonitorExpression the `synchronized` block
MultiConditionalExpression the comparison of two values
NewExpression the `new` keyword
PrimaryExpression the simplest expression (e.g. number, String, null)
ReturnExpression the `return` keyword
SingleConditional a comparison to zero
SwitchExpression a `switch-case` block
TernaryExpression a ternary operator `?:`
ThrowExpression the `throw` keyword
TryCatchExpression a `try-catch` block
UnaryExpression a unary increment operator
UnconditionalJump a general representation of `goto`
VariableDeclarationExpression a declaration of a variable
VariablePrimaryExpression an access to a variable

Figure 4.1: List of all `Expression` subtypes

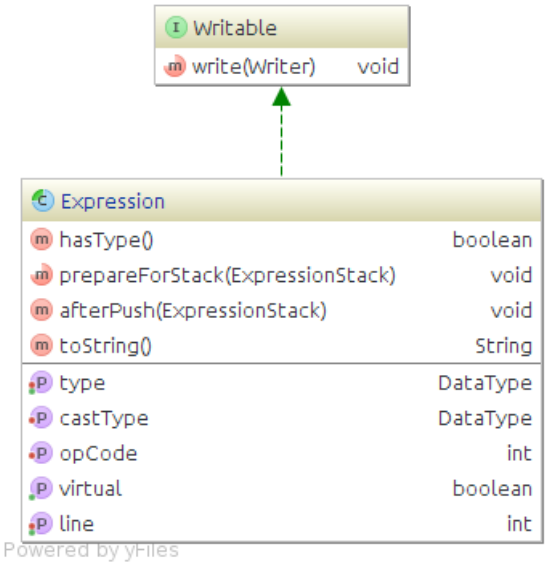

Figure 4.2: Class diagram of `Expression`

tains either an expression (e.g. `Statement`), or a list of statements (e.g. `BlockStatement`), or both (e.g. `SwitchStatement`).

Expressions that have been wrapped into statements are finished and can be printed out. The list in figure 4.3 summarizes all created subclasses of `Statement`.

Same as `Expression`, `Statement` also implements the `Writable` interface that enables it to be written out and propagate the writer further down to the expressions it contains. In fact, it also extends an abstract class called `CodeElement` which is a common superclass for both `Statement` and `Block`, as can be seen in a class diagram in figure 4.4. The common superclass provides subclasses with the ability to count parents which is used to count the depth, i.e. number of tabs needed for indentation.

4.2.3 Block

`Block` represents a member of a Java class on the class-level. However, the blocks are also responsible for converting class members from bytecode representation to Java source code. This entails con-

DoWhileLoopStatement a `do-while` loop

IfThenElseStatement a `if-then-else` control flow statement

IfThenStatement a `if-then` control flow statement

Statement a general statement for simple expressions

SwitchStatement a `switch-case` block

SynchronizedStatement a `synchronized` block

TryCatchStatement a `try-catch` block

WhileLoopStatement a `while` loop

Figure 4.3: List of all `Statement` subtypes

version of names and types, creation of fields, methods and inner classes. Specific `Block` classes were necessary only for representation of classes, methods and fields. All subtypes of `Block` are listed below together with their description.

4.2.3.1 ClassBlock

ClassBlock drives the translation of the `ClassNode`. Decompilation of other class components is driven from this block and results are stored in a general array of "children" that extend the `CodeElement` class. Thanks to this, the write-out of the decompiled code is done in the same way throughout the whole codebase.

`ClassNode` represents a single class file that is being decompiled and contains all the information about the class. This information is transformed from the bytecode representation back to Java code. It includes class annotations, access flags, class name, superclasses and interfaces and the inside contents of the class such as fields, methods and inner classes. Fields and methods are decompiled in separate blocks.

Inner classes, however, do not require a separate subclass of `Block` because they are treated as a standard `ClassNode`. The only difference is that they have a parent and are included in the children list of their parent. This unified treatment of all classes is possible thanks to the fact that they are stored in separate class files. And they are in fact treated as separate classes by the JVM.

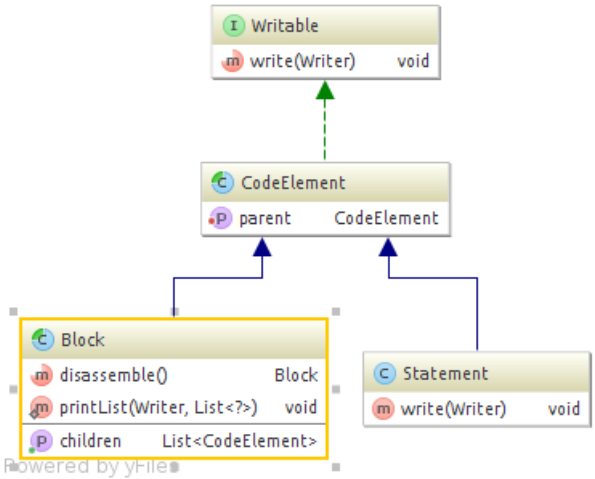

Figure 4.4: Class diagram of `CodeElement` and its subclasses

Name of the class file for an inner class consists of the owning class name followed by the `$` (dollar sign) and ended with the name of the inner class (e.g. `Outer$Inner.class`). A special case of inner classes are anonymous classes. These are used in cases when a local class is needed to be used only once and they do not have a name. Compiler assigns names to these classes automatically using numbers (e.g. `Outer$1.class`). The only information an outer class contains about its inner classes are their names. Thanks to that it is possible to decompile inner classes and add them as children to their outer class.

4.2.3.2 FieldBlock

FieldBlock handles the translation of the `FieldNode`. If the field contains annotations, they are also translated back to Java code. After the `FieldNode` is processed, it is added as a child to the list of children in the `ClassBlock`.

Field assignments are always done in the constructor (`<init>` for instance members or `<clinit>` for class members). So regardless of whether the fields in the original Java source file were assigned with the declaration or in the constructor, in the decompiled class they

will always be assigned in the constructor. The only exception is the case when the field is declared `static` and `final` and it is of one of the primitive types (`int`, `long`, `float`, `double`) or `String`. In this case it is a constant which is stored in the *constant pool* and it is assigned directly with the declaration.

4.2.3.3 MethodBlock

MethodBlock handles the translation of the `MethodNode`. This block contains all the information about a single method, which includes annotations, access flags, return type, name, arguments, checked exceptions and code.

The translation of method information from bytecode to Java is done in this block and it also initiates the decompilation of bytecode instructions. But this process is handled by a separate class and is described in the section 4.2.4.

A very important thing that `MethodBlock` and `FieldBlock` have in common is translation of *descriptor* and also *signature* (translation of the signature is done in `ClassBlock` too). Descriptor is a string that represents the type of a field or method and it is used by the JVM. Signature encodes Java declarations that use types which are outside the JVM typesystem, therefore it contains information about generic types [23]. Generic types can be either parametrized types [6] or type variables [7] [3]. Type parameters of generic methods are erased during compilation so bytecode contains only ordinary types, but the compiler must provide this information in the signature.

If the processed node contains a signature, then the signature is parsed and the whole types (possibly with generics) are used. Otherwise, the type information from the descriptor is used. A modified version of `TraceSignatureVisitor` class from `util` package of ASM is used to translate the signature.

---

6. `Collection<Integer>`
7. `abstract <T> Collection<T> method(T arg);` where T can be any type

4.2.4 Bytecode translation process

The process of bytecode decompilation is driven by the class called `InstructionTranslator` and it is done by iterating through the list of `AbstractInsnNode` objects called `InsnList`. Each node in this list represents either a bytecode instruction or a group of similar instructions. For each type of these nodes, there is an instance of `NodeHandler` that handles the processing of a specific node. Each specific handler extends `AbstractNodeHandler`, the class diagram of which can be seen in figure 4.5. This modular approach allows for simple creation and addition of new handlers in case there are new nodes added in the future releases of ASM.

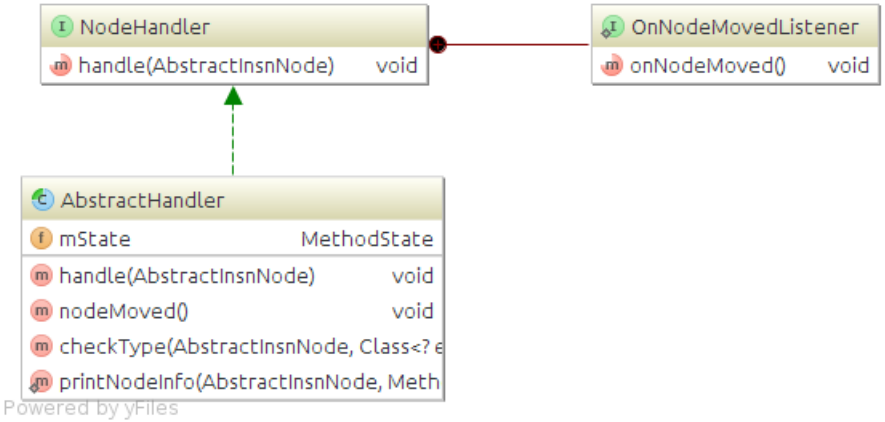

Figure 4.5: Class diagram of `NodeHandler`

The `AbstractNodeHandler` provides common functionality, such as access to the `MethodState`. This class contains all the state information about the currently processed method that any node might need. It is created by the `InstructionTranslator` and passed on to each node handler constructor so that every node has the reference to the same state. The diagram of this relationship can be seen in figure 4.6.

The `MethodState` contains information about the decompilation such as currently processed node, line number, map of local variables and their positions and others. It also contains a reference to the currently used `ExpressionStack`.

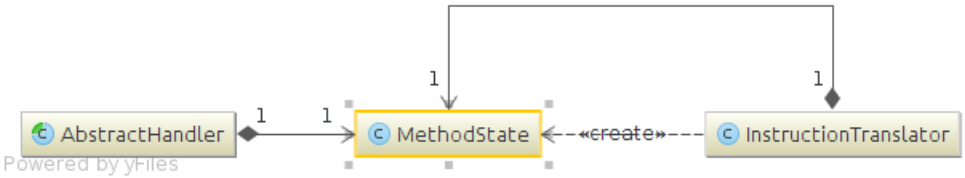

Figure 4.6: `MethodState` representing the translation context shared between the handlers

`ExpressionStack` provides a way to store resolved expressions and at the same time a way to resolve expressions on top of the stack. This is achieved by allowing expressions to modify the stack before and after being pushed onto it. Thanks to this approach, Java expressions are built from the bytecode instructions incrementally, in one traversal of the instruction list.

Figure 4.7 contains an example of this incremental process. It shows the state of `ExpressionStack` after processing of each bytecode instruction. Expressions that are created and pushed onto the stack in each step are the following:

1. VariablePrimaryExpression
2. ConstanPrimaryExpression
3. ArithmeticExpression
4. AssignmentExpression

The `ArithmeticExpression` pops its two operands from the stack and stores them inside itself, adding an arithmetic operation (in this case it is addition). The resulting expression that is pushed on the stack is later popped by the `AssignmentExpression` stores it inside itself as the right-hand side of the assignment and adds the left-hand side. Code generated from these instructions can look something like this `var1 = var1 + 1`.

4.2.4.1 Detecting control flow

More complex expressions such as `JumpExpression`, `TryCatchExpression` or `SwitchExpression` contain one or multiple instances of `Ex-`

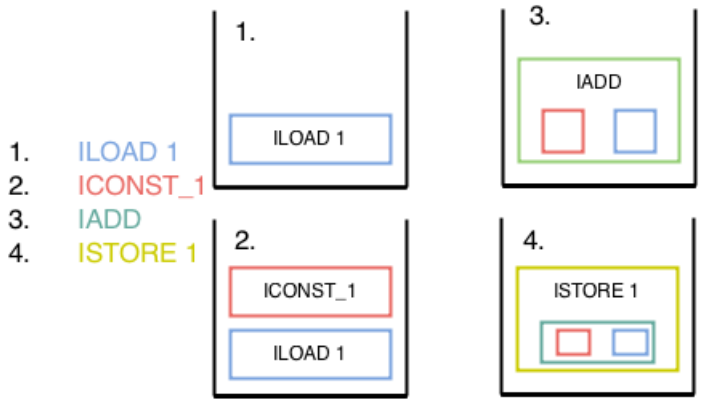

Figure 4.7: Contents of the `ExpressionStack` after pushing each `Expression`

`pressionStack` inside them. These contain expressions that are inside the block defined by the expression (e.g. a `JumpExpression` contains two stacks, one is for *"then branch"* the other for *"else branch"*). Including the stack inside the expression clearly defines the scope of the block in the original Java code. Therefore, it simplifies the translation back to Java.

Logic that handles when a new `ExpressionStack` should be started and finished is contained within node handlers that create these expressions. This is also the reason why `NodeHandler` defines `OnNodeMovedListener` (figure 4.5). It provides the `InstructionTranslator` a way to always process the moved node in the same way regardless of which handler moved the node. The possibility to change nodes inside a handler and continue translation on the moved node effectively means that the translation is done recursively (a form of indirect recursion).

To support both the recursive behavior of the translation process and the ability to change the active `ExpressionStack` on the fly, the `MehtodState` contains a `Stack` of currently used `ExpressionStacks`. The active `ExpressionStack` is always on top and it is popped from the top when it is finished (e.g. then block of an `if-then-else` statement has ended). The stack that was below is then on top. It represents the enclosing block and the finished expression is pushed on it. This idea is presented in the figure 4.8. It shows a simple method

with two `ExpressionStacks` (marked by the blue and red boxes) with the `Stack` that keeps the active `ExpressionStack` on top.

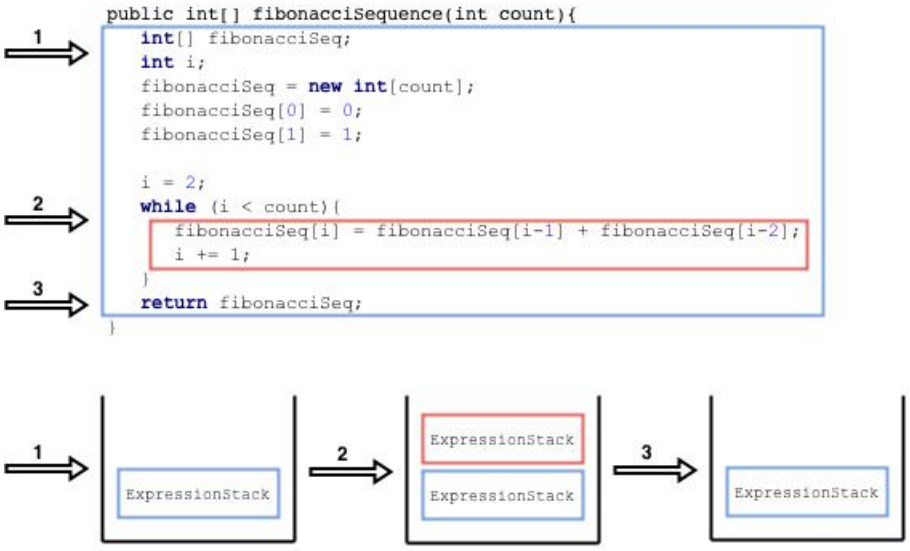

Figure 4.8: Switching of instances of `ExpressionStack` according to what code block is being processed. Below is the stack that keeps currently used `ExpressionStacks`.

Handlers that create a new `ExpressionStack` must be able to determine when to start and finish the stack. To help achieve that, the `MethodState` contains fields like current label[8] (the last encountered label), set of visited labels (all labels that have been encountered), and frame label (label of the stack map frame which is present at every jump destination). Having this information about the current location in the bytecode together with the location information present in the bytecode of jump instructions (or blocks such as try-catch-finally, or switch-case) is enough to decide when to start and finish a stack.

Not all Java code features can be correctly recreated from bytecode in one traversal of the node list. For this reason, the `ExpressionStack` provides a way to enhance the expressions in it. This is done through an interface called `StackEnhancer`. The stack can hold a num-

8. position in the bytecode used for jump instructions, switch instructions and `try-catch` blocks

ber of these enhancers and each can modify the stack in any way it wants. Current implementation contains only one enhancer called `LoopEnhancer`. It is used to detect statements such as `do-while` loop or `continue` and `break` statements. Other enhancers were not necessary but they can be created and added in a very simple way.

4.3 User interface

In order to be easy to use, the GUI of the application is as simple as possible. It provides a standard open dialog to select a class file. The interface contains two code panes. Left side contains bytecode in ASM format, right side holds Java code generated by the decompiler. Screenshot of the window is in the figure 4.9. The interface also allows to save the generated Java code into a file.

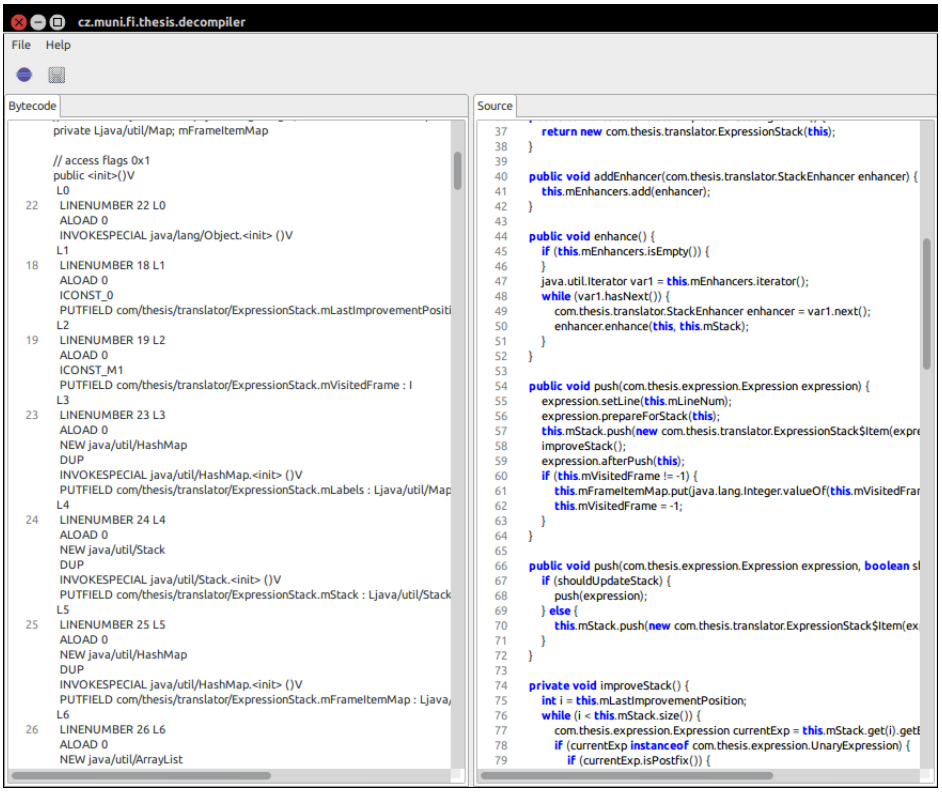

Figure 4.9: Simple decompiler GUI

From the implementation point of view, the GUI logic is implemented in simple Java classes that are registered on the framework in `Application.e4xml`. To give these classes access to framework objects, such as *service* or *application*, the Eclipse RCP framework uses annotations. They are also used to mark methods that are to be called at certain times of application run (e.g. `@PostConstruct` is called after a class is initialized and all fields have been injected).

5 Achieved results

The following chapter contains a few examples of compiled Java code together with their decompiled counterparts. The aim is to showcase the abilities of the created tool. These examples also contain descriptions that provide some more in-depth information. The intention to improve understanding of the bytecode and effectively of the Java language. In addition to this chapter, 6 of the tests used in the decompiler surveys of 2003 and 2009 [16] [22] are provided in the appendix A together with their results using this decompiler.

5.1 Switches

There are two possible instructions for creating Java `switch` statement in bytecode, `tableswitch` and `lookupswitch`. Both have variable lengths and contain offset of the `default` branch of the statement. Also, they both take the expression from the top of the stack to use as a key. The difference between these instructions is in the way they indicate the case branches.

The `tableswitch` instruction contains minimum and maximum case values. These values indicate the endpoints of the range included in this instruction. Following the maximum value are `case` offsets for each value [37]. The `switch` statement that gets compiled to `tableswitch` by *javac* can be seen in the listing 5.1 and the instruction itself in listing 5.2. Here it is clearly visible that even though the statement in Java code only contains three cases, the instruction contains all `case` values within the range but they just point to the same location as the `default` case.

Listing 5.1: A `switch` statement with cases that are close to each other

```
switch (number) {
      case 1:
        System.out.println("one");
        break;
      case 2:
        System.out.println("two");
        break;
      case 5:
```

```
        System.out.println("five");
        break;
      default:
        System.out.println("default");
    }
```

Listing 5.2: `tableswitch` and `lookupswitch` instructions with cases followed by the labels of the code locations in the ASM format

```
TABLESWITCH                    LOOKUPSWITCH
  1: L1                        1: L1
  2: L2                        8: L2
  3: L3                        16: L3
  4: L3                        default: L4
  5: L4
  default: L3
```

Decompiling this statement with the created decompiler, we get exactly the same result as the source Java code in listing 5.1. The reason for this is the fact even though the instruction contains the whole table with keys and their locations, the decompiler maps the `case` keys to their locations. Therefore, the two keys that point to the same location as the `default` case are not printed.

The `lookupswitch` is a more general instruction but the lookup is less efficient than in the case of `tableswitch`. The `case` values are stored in pairs with their `case` offsets, sorted in increasing order. The lookup must check the `case` value of each pair until it finds a match. Default `case` is selected if the lookup runs out of values or encounters a value greater than the one it is searching for [37]. An example of a `switch` statement that is compiled to bytecode using `lookupswitch` is in the listing 5.3.

Listing 5.3: A `switch` statement with cases that are far away from each other

```
switch (number) {
      case 1:
        System.out.println("one");
        break;
      case 16:
        System.out.println("sixteen");
        break;
      default:
        System.out.println("default");
```

```
}
```

Until the version 7 of Java language, only `switch` statements with `char`, `int`, `byte`, `short` (and their respective classes) or enum were supported. Version 7 adds a new feature that allows using `String` as a key, listing 5.4. Interestingly, however, this new feature did not change the bytecode instructions and they do not support switching on a string [12]. The implementation of this feature lies solely on the compiler, which means that there are many possible translation schemes. The decompiled example is in the listing 5.5.

From the decompiled code it is clear that *javac* creates two `switch` statements. The first one is compiled using `lookupswitch` over the hash code of the string. The second one is compiled with `tableswitch` since values of keys are increased incrementally.

Listing 5.4: A `switch` statement with `String`

```
switch (text) {
  case "first":
    System.out.println("one");
    break;
  case "second":
    System.out.println("eight");
    break;
  case "third":
    System.out.println("sixteen");
    break;
  default:
    System.out.println("default");
    }
```

Listing 5.5: Decompiled `switch` statement with `String`

```
java.lang.String var2 = text;
int var3 = -1;
switch (var2.hashCode()) {
  case 97440432:
    if (var2.equals("first")) {
      var3 = 0;
    }
  case -906279820:
    if (var2.equals("second")) {
      var3 = 1;
    }
  case 110331239:
```

```
    if (var2.equals("third")) {
      var3 = 2;
    }
  default:
    switch (var3) {
      case 0:
        java.lang.System.out.println("one");
        break;
      case 1:
        java.lang.System.out.println("eight");
        break;
      case 2:
        java.lang.System.out.println("sixteen");
        break;
      default:
        java.lang.System.out.println("default");
  }
}
```

5.2 Lambda expressions

Arguably, one of the most significant additions to Java language in version 8 was so called *Project Lambda*[1] which added lambda expressions. Their goal is to simplify work with anonymous classes and make the code more concise by allowing passing a functionality to a method as an argument without the need to create an object. The listing 5.6 contains a simple example where these two approaches are compared. Other features that were added thanks to lambda expressions include method references or default interface methods.

Listing 5.6: Lambda expression compared to anonymous class

```
public void lambdaRunnable() {
  Runnable r = () -> System.out.println("Hello world!");
  r.run();
}

public void anonymousClass() {
  Runnable r = new Runnable() {
    @Override
    public void run() {
```

1. http://openjdk.java.net/projects/lambda/

```
        System.out.println("Hello world!");
      }
    };
    r.run();
  }
```

These lambda features take advantage of the new instruction added in version 7, called `invokedynamic`. This instruction, also referred to as *dynamic call site*, originally does not contain any target method that could be invoked. Before calling the call site, a method must first be *linked*. A *bootstrap* method is called to accomplish linking. This method is specified for each `invokedynamic` instruction statically in the constant pool. Name and type of the call site are also specified in constant pool, just like for any `invoke` instruction.

Decompiled version of the `lambdaRunnable` method in listing 5.6 is shown in listing 5.7. Note that, the `import` statement is not actually in the decompiled code and invocations of all methods from `java.lang.invoke` package are with fully qualified names, only here they are shortened for the sake of readability.

This decompiled lambda expression seems very complex, and indeed it would probably suffice to simply display some symbolic information in pseudocode that the `invokedynamic` instruction is invoked. However, this representation, despite it being rather complex, provides a lot of information about the implementation of lambdas.

Listing 5.7: Decompiled lambda expression

```
import java.lang.invoke.*;

public void lambdaRunnable() {
  java.lang.Runnable r;
  r = LambdaMetafactory.metafactory(
      /*stacked automatically by the VM*/
      MethodHandles.lookup(),
      "run",
      MethodType.methodType(java.lang.Runnable.class),
      MethodType.methodType(void.class),
      MethodHandles.lookup().findStatic(
          InvokedynamicInsnNode_lambda.class,
          "lambda$lambdaRunnable$0",
          MethodType.methodType(void.class)),
      MethodType.methodType(void.class));
  r.run();
```

```
}

private static /*synthetic*/ void lambda$lambdaRunnable$0()
{
  java.lang.System.out.println("Hello world two!");
}
```

5.3 `try-catch` blocks

The `try-catch` statements of Java language are not represented by any special instructions in bytecode. Instead, their locations and scopes are stored in a table called *exception table* shown in listing 5.8.

The table is present only in methods that contain `try-catch` blocks. Columns in the table represent the starting and ending location of the `try` block and the location of exception handler (`catch` block), followed by an exception type handled by that block. From this exception table we can see that the method contains two `try` blocks, one between labels `L0` and `L1` handled by code at label `L2`. The other is between labels `L0` and `L3` and has four `catch` blocks. Additionally, we can also see that the `try-catch` block in the first line is enclosed in the block described in the remaining lines. The four lines with `null` point to a handler that handles any exception type, which effectively means that it is a `finally` block.

Listing 5.8: Exception table in ASM representation

```
TRYCATCHBLOCK L0 L1 L2 java/lang/NegativeArraySizeException
TRYCATCHBLOCK L0 L3 L4 java/lang/IndexOutOfBoundsException
TRYCATCHBLOCK L0 L3 L5 java/lang/NullPointerException
TRYCATCHBLOCK L0 L3 L5 java/lang/ArithmeticException
TRYCATCHBLOCK L0 L3 L6 null
TRYCATCHBLOCK L4 L7 L6 null
TRYCATCHBLOCK L5 L8 L6 null
TRYCATCHBLOCK L6 L9 L6 null
```

An important change in compilation of `finally` blocks came in Java version 6. Until then, instructions `jsr` and `ret` were used to compile the `finally` blocks (more information in section 3.13 of JVMS [23]). Class files of version 50.0 and higher do not use these instructions. Instead, the code contained in the `finally` block is repeated

after each `catch` block. This can be seen in the listing 5.9 which contains an excerpt from the same method as the table in listing 5.8.

Listing 5.9: Repeated `finally` blocks

```
L4
  LINENUMBER 32 L4
  FRAME FULL [TryCatchBlockNode java/lang/String] \
    [java/lang/IndexOutOfBoundsException]
  ASTORE 3
L16
  LINENUMBER 33 L16
  BIPUSH -3
  ISTORE 2
L17
  LINENUMBER 34 L17
  GETSTATIC java/lang/System.out : Ljava/io/PrintStream;
  LDC "index out of bounds exception caught"
  INVOKEVIRTUAL java/io/PrintStream.println \
    (Ljava/lang/String;)V
L7
  LINENUMBER 39 L7
  GETSTATIC java/lang/System.out : Ljava/io/PrintStream;
  LDC "called finally"
  INVOKEVIRTUAL java/io/PrintStream.println \
    (Ljava/lang/String;)V
L18
  LINENUMBER 40 L18
  GOTO L15
L5
```

Listing 5.9 shows the `catch` block for `IndexOutOfBoundsException`. Code for this block is between labels `L4` and `L7` and it simply stores value -3 to a local variable at position 2 and then prints a message invoking the `println` method. The exception table in listing 5.8 contains a handler for this exact range which specifies that the exception handler for this part starts at label `L6` (`finally` block). The label `L5` marks the starting position of another handler. Interestingly, the code between labels `L7` and `L18` contains the copy of `finally` block which is executed when the code executes without any exceptions. This copy of `finally` block is followed by an unconditional jump (`goto`) to the position after the whole `try-catch` block.

Project Coin [12], added in version 7 of JDK, brought along a cou-

ple of useful improvements of `try-catch` blocks. One addition is so-called *multicatch* which allows handling of multiple exceptions in a single `catch` block. This was only a small change which simply allows using "`|`" ("OR") operator in `catch` clauses to list multiple exceptions. In the exception table of listing 5.8 is multicatch represented as multiple lines (lines 3 and 4) that have the same locations but different exception types.

Another improvement added by the Project Coin is so-called `try-with-resources` statement. This addition is simply a syntactic sugar that automatically closes declared resources. The declared object must simply implement the `java.lang.AutoCloseable` interface. An example of a method that uses this statement is in listing 5.10. Its decompiled counterpart is provided in listing 5.11.

Listing 5.10: Method that contains a `try`-with-resources statement

```
void tryWithResourcesPrintFile() throws IOException {
  try(FileInputStream input = new FileInputStream("f")) {
    int data = input.read();
    while(data != -1){
      System.out.print((char) data);
      data = input.read();
    }
  }
}
```

Listing 5.11: Decompiled `try`-with-resources statement

```
void tryWithResourcesPrintFile() throws java.io.IOException {
  java.io.FileInputStream input =
          new java.io.FileInputStream("f");
  java.lang.Object var2 = null;
  try {
    int data = input.read();
    while (data != -1) {
      java.lang.System.out.print((char) data);
      data = input.read();
    }
  } catch (java.lang.Throwable var3) {
    var2 = java.lang.Object;
    throw var3;
  } finally {
    if (input != null) {
      if (var2 != null) {
```

```
        try {
          input.close();
        } catch (java.lang.Throwable var5) {
          var2.addSuppressed(var5);
        }
      } else {
        input.close();
      }
    }
  }
}
```

6 Conclusion

The aim of this thesis was to create a working prototype of a Java decompiler. This goal was successfully implemented. The created application can indeed decompile class files generated by the javac compiler. It provides a simple GUI that can be used for viewing bytecode alongside the generated Java source code. Additionally the decompiler was developed as a library so it can even be used with different user interfaces.

Nevertheless, the created tool has some known limitations. The first limitation is the fact that the decompiler can be only used for decompilation of class files compiled using *javac*. Although it is surely a limiting factor in usability of the tool, it allowed to implement the pattern-matching decompilation strategy based on the bytecode patterns generated by this one compiler. There has been a number of decompilers over the years that have used a form of pattern-matching approach and therefore targeted only one compiler so this is a relatively popular approach. A great advantage and main reason for using this approach was the fact that the decompiled code can look very similar to the original source code. This allows to better see a connection between Java code and bytecode.

Second limitation is absence of any form of type inference. However, this does not hinder the decompilation process in any way, especially since this tool is mainly intended to be used for educational purposes. In that case the class files can be compiled with the debug information which would mean that they contain all information about types. The addition of a type inference algorithm could be an interesting problem to solve and it is one of the possible future improvements of this tool.

Lastly, there is a problem with testing of a tool like this and validating that it is able to correctly decompile any class file (generated by *javac*). Even using more than 40 distinct test classes, which test that all basic Java statements and features are correctly decompiled, cannot provide enough certainty that the decompiler can correctly decompile any class file. For this reason a program that would be able to validate that the decompiler works correctly, would be a valuable tool and definitely an interesting challenge to tackle.

In conclusion, the created decompiler can be used to help improve understanding of the bytecode, which was the main goal of this work. The aim to use a bytecode manipulation library was also successfully fulfilled and this work shows that the tools such as ASM can be used for this kind of application with very good results. On the other hand, the original intention to use the visitor pattern provided by the ASM and perform the decompilation without the need to create a tree representation was not successful. However, the resulting approach that works with the tree structure could probably be modified to fit the method visitor pattern. This could improve the speed and reduce the memory footprint of the decompilation process. Yet, using the tree API of the ASM takes full advantage of the framework's abilities of bytecode ananlysis.

A Evaluation tests

Here are 6 of the 9 tests that were used in the evaluation surveys in 2003 and 2009 [16] [22]. Only the tests that were compiled using *javac* in the original survey were tested. In each section there is always a short description of the test followed by the original source code and decompiled code. For testing here were tests compiled with the option `-g` which means that the class files contain debugging information.

A.1 Casting

A program to test if decompiler can correctly detect the need to cast `char` to `int`.

Listing A.1: Casting original

```
public class Casting {
  public static void main(String args[]) {
    for (char c = 0; c < 128; c++) {
      System.out.println(" ascii " + (int) c
                         + " character " + c);
    }
  }
}
```

Listing A.2: Casting decompiled

```
public class Casting {

  public Casting() {
    super();
  }

  public static void main(java.lang.String[] args) {
    char c = 0;
    while (c < 128) {
      java.lang.System.out.println(
            new java.lang.StringBuilder()
                .append(" ascii ").append((int) c)
                .append(" character ").append(c)
                .toString());
      c = (char) (c + 1);
```

```
    }
  }
}
}
```

A.2 ControlFlow

A program which tests detection of control flow.

Listing A.3: ControlFlow original

```
public class ControlFlow {
  public static int foo(int i, int j) {
    while (true) {
      try {
        while (i < j)
          i = j++ / i;
      } catch (RuntimeException re){
        i = 10;
        continue;
      }
      break;
    }
    return j;
  }

  public static void main(String[] args) {
    System.out.println(foo(1, 2));
  }
}
```

Listing A.4: ControlFlow decompiled

```
public class ControlFlow {

  public ControlFlow() {
    super();
  }

  public static int foo(int i, int j) {
    try {
      while (i < j) {
        i = (j++) / i;
      }
    } catch (java.lang.RuntimeException re) {
```

```
      i = 10;
    }
    return j;
  }

  public static void main(java.lang.String[] args) {
    java.lang.System.out.println(ControlFlow.foo(1, 2));
  }
}
```

A.3 Fibo

A program that contains many of the basic Java language constructs.

Listing A.5: Fibo original

```
class Fibo {
  private static int fib(int x) {
    if (x > 1)
      return (fib(x - 1) + fib(x - 2));
    else return x;
  }

  public static void main(String args[]) throws Exception {
    int number = 0, value;
    try {
      number = Integer.parseInt(args[0]);
    } catch (Exception e) {
      System.out.println(" Input error ");
      System.exit(1);
    }
    value = fib(number);
    System.out.println(" fibonacci (" + number + ") = "
                       + value);
  }
}
```

Listing A.6: Fibo decompiled

```
class Fibo {

  Fibo() {
    super();
  }
```

```
  private static int fib(int x) {
    if (x > 1) {
      return (Fibo.fib(x - 1)) + (Fibo.fib(x - 2));
    }
    return x;
  }

  public static void main(java.lang.String[] args)
                          throws java.lang.Exception {
    int number = 0;
    try {
      number = java.lang.Integer.parseInt(args[0]);
    } catch (java.lang.Exception e) {
      java.lang.System.out.println(" Input error ");
      java.lang.System.exit(1);
    }
    int value = Fibo.fib(number);
    java.lang.System.out.println(
                  new java.lang.StringBuilder()
                      .append(" fibonacci (")
                      .append(number).append(") = ")
                      .append(value).toString());
  }
}
```

A.4 Sable

A program to test ability to perform type inference for local variables.

Listing A.7: Sable original

```
public class Sable {
  public static void f(short i) {
    Circle c;
    Rectangle r;
    Drawable d;
    boolean is_fat;
    if (i > 10) {
      r = new Rectangle(i, i);
      is_fat = r.isFat();
      d = r;
    } else {
      c = new Circle(i);
      is_fat = c.isFat();
```

```
      d = c;
    }
    if (!is_fat) d.draw();
  }

  public static void main(String args[]) {
    f((short) 11);
  }
}

public interface Drawable {
  public void draw();
}

public class Circle implements Drawable {
  public int radius;

  public Circle(int r) {
    radius = r;
  }

  public boolean isFat() {
    return false;
  }

  public void draw() {
  // Code to draw ...
  }
}

public class Rectangle implements Drawable {
  public short height, width;

  public Rectangle(short h, short w) {
    height = h;
    width = w;
  }

  public boolean isFat() {
    return (width > height);
  }

  public void draw() {
  // Code to draw ...
  }
```

```
}
```

Listing A.8: Sable decompiled (only the Sable.java class)

```
public class Sable {

  public Sable() {
    super();
  }

  public static void f(short i) {
    Drawable d;
    boolean is_fat;
    if (i > 10) {
      Rectangle r = new Rectangle(i, i);
      is_fat = r.isFat();
      d = (Drawable) r;
    } else {
      Circle c = new Circle(i);
      is_fat = c.isFat();
      d = (Drawable) c;
    }
    if (!is_fat) {
      d.draw();
    }
  }

  public static void main(java.lang.String[] args) {
    Sable.f(11);
  }
}
```

A.5 TryFinally

A program to test whether decompiler can decompile the implementation of try-finally blocks using in-line code.

Listing A.9: TryFinally original

```
public class TryFinally {
  public static void main(String[] args) {
    try {
      System.out.println(" try ");
    } finally {
      System.out.println(" finally ");
```

```
    }
  }
}
```

Listing A.10: TryFinally decompiled

```
public class TryFinally {

  public TryFinally() {
    super();
  }

  public static void main(java.lang.String[] args) {
    try {
      java.lang.System.out.println(" try ");
    } finally {
      java.lang.System.out.println(" finally ");
    }
  }
}
```

A.6 Usa

A program to test if decompiler can reconstruct inner classes.

Listing A.11: Usa original

```
public class Usa {
  public String name = " Detroit ";

  public class England {
    public String name = " London ";

    public class Ireland {
      public String name = " Dublin ";

      public void print_names() {
        System.out.println(name);
      }
    }
  }
}
```

Listing A.12: Usa decompiled

```
public class Usa {
```

```
  public java.lang.String name;

  public Usa() {
    super();
    this.name = " Detroit ";
  }

  public class England {
    public java.lang.String name;
    final  /* synthetic */ Usa this$0;

    public England(Usa this$0) {
      this.this$0 = this$0;
      super();
      this.name = " London ";
    }

    public class Ireland {
      public java.lang.String name;
      final  /* synthetic */ England this$1;

      public Ireland(England this$1) {
        this.this$1 = this$1;
        super();
        this.name = " Dublin ";
      }

      public void print_names() {
        java.lang.System.out.println(this.name);
      }
    }
  }
}
```

B Archive

The attached archive contains:

- Source code of the thesis in LaTeX format
- Thesis in the pdf format
- Executable file of the application
- README file with the installation instructions
- Folder with the decompiler source code and the testing files
- Folder with the GUI source code

Bibliography

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[2] Lee Benfield. CFR - another java decompiler. `http://www.benf.org/other/cfr/index.html`, 2011. Accessed: 2015-05-01.

[3] Gilad Bracha, Sun Microsystems, Norman Cohen Ibm, Christian Kemper Inprise, Martin Odersky Epfl, David Stoutamire, and Sun Microsystems. Adding generics to the java programming language: Public draft specification, version 2.0. Technical report, 2003.

[4] Eric Bruneton. ASM 4.0 - A Java bytecode engineering library. `http://download.forge.objectweb.org/asm/asm4-guide.pdf`, 2011. Accessed: 2015-04-25.

[5] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. ASM: a code manipulation tool to implement adaptable systems. `http://asm.ow2.org/current/asm-eng.pdf`, 2002. Accessed: 2015-04-25.

[6] C. Cifuentes and Queensland University of Technology. School of Computing Science. *Reverse Compilation Techniques*. Queensland University of Technology, Brisbane, 1994. URL: `https://books.google.cz/books?id=DWEFNQAACAAJ`.

[7] Oracle Corporation. javac - Java programming language compiler. `http://docs.oracle.com/javase/6/docs/technotes/tools/solaris/javac.html`, 1993. documentation, [Online].

[8] Oracle Corporation. javap - The Java Class File Disassembler. `http://docs.oracle.com/javase/6/docs/technotes/tools/solaris/javap.html`, 1993. documentation, Accessed: 2015-03-15.

[9] Oracle Corporation. javac.main.RecognizedOptions.java. http://hg.openjdk.java.net/jdk6/jdk6/langtools/file/a9008b46db24/src/share/classes/com/sun/tools/javac/main/RecognizedOptions.java#l553, 2007. source code, Accessed: 2015-03-15.

[10] Oracle Corporation. Compatibility Guide for JDK 8. http://www.oracle.com/technetwork/java/javase/8-compatibility-guide-2156366.html, 2015. Accessed: 2015-03-10.

[11] Oracle Corporation. Compilation Overview. http://openjdk.java.net/groups/compiler/doc/compilation-overview/, 2015. Accessed: 2015-03-10.

[12] Joe Darcy. Small Enhancements to the JavaTM Programming Language, JSR 334. https://jcp.org/en/jsr/detail?id=334, 2014. Accessed: 2015-03-10.

[13] Mikhail Dmitriev. *Safe Class and Data Evolution in Large and Long-Lived Java Applications*. PhD thesis, University of Glasgow, May 2001. Chapter 7.1.2 The Features of the HotSpot JVM.

[14] Emmanuel Dupuy. JD Project. http://jd.benow.ca/, 2014. Accessed: 2015-05-01.

[15] Eldad Eilam. *Reversing: The Hacker's Guide to Reverse Engineering*. John Wiley & Sons, 2005.

[16] M. V. Emmerik. Java Decompiler Tests. http://www.program-transformation.org/Transform/JavaDecompilerTests, 2003. Accessed: 2014-10-05.

[17] The Apache Software Foundation. Apache Commons BCEL™. http://commons.apache.org/proper/commons-bcel/, 2014. Accessed: 2015-04-25.

[18] EtienneM. Gagnon, LaurieJ. Hendren, and Guillaume Marceau. Efficient inference of static types for java bytecode. In Jens Palsberg, editor, *Static Analysis*, volume 1824 of *Lecture Notes in Computer Science*, pages 199–219. Springer Berlin Heidelberg,

2000. URL: http://dx.doi.org/10.1007/978-3-540-45099-3_11, doi:10.1007/978-3-540-45099-3_11.

[19] Brian Goetz. Lambda Expressions for the Java(TM) Programming Language, JSR 335. https://jcp.org/en/jsr/detail?id=335, 2014. Accessed: 2015-03-10.

[20] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. The Java® Language Specification, Java SE 8 Edition. http://docs.oracle.com/javase/specs/jls/se8/html/index.html, 2015. Accessed: 2015-02-05.

[21] Robert Grosse. Krakatau Bytecode Tools. https://github.com/Storyyeller/Krakatau, 2012. Accessed: 2015-05-01.

[22] James Hamilton and Sebastian Danicic. An evaluation of current java bytecode decompilers. In *Ninth IEEE International Workshop on Source Code Analysis and Manipulation*, volume 0, pages 129–136, Edmonton, Alberta, Canada, 2009. IEEE Computer Society. doi:http://doi.ieeecomputersociety.org/10.1109/SCAM.2009.24.

[23] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. The Java® Virtual Machine Specification, Java SE 8 Edition. http://docs.oracle.com/javase/specs/jvms/se8/html/index.html, 2015. Accessed: 2015-03-15.

[24] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. The Java® Virtual Machine Specification, Java SE 8 Edition. http://docs.oracle.com/javase/specs/jvms/se8/html/jvms-2.html#jvms-2.11.4, 2015. Section 2.11.4 Type Conversion Instructions, Accessed: 2015-03-15.

[25] Sun Microsystems. The Java HotSpot Performance Engine Architecture. Technical report. http://www.oracle.com/technetwork/java/whitepaper-135217.html#3.

[26] Sun Microsystems. The Java HotSpot Performance Engine Architecture. Technical report. http://www.oracle.com/technetwork/java/whitepaper-135217.html.

[27] Jerome Miecznikowski. New algorithms for a Java decompiler and their implementation in Soot. Master's thesis, McGill University, 2003.

[28] Jerome Miecznikowski and Laurie Hendren. Decompiling java bytecode: Problems, traps and pitfalls. In R.Nigel Horspool, editor, *Compiler Construction*, volume 2304 of *Lecture Notes in Computer Science*, pages 111–127. Springer Berlin Heidelberg, 2002. URL: `http://dx.doi.org/10.1007/3-540-45937-5_10`, `doi:10.1007/3-540-45937-5_10`.

[29] Vijay Sundaresan Patrick Lam Etienne Gagnon Raja Vallée-Rai, Laurie Hendren and Phong Co. Soot - a java optimization framework. In *Proceedings of CASCON 1999*, pages 125–135, 1999. URL: `www.sable.mcgill.ca/publications`.

[30] John Rose. Supporting Dynamically Typed Languages on the Java(TM) Platform , JSR 292. `https://jcp.org/en/jsr/detail?id=292`, 2011. Accessed: 2014-09-10.

[31] Yunhe Shi, Kevin Casey, M Anton Ertl, and David Gregg. Virtual machine showdown: Stack versus registers. *ACM Transactions on Architecture and Code Optimization (TACO)*, 4(4):2, 2008.

[32] Eric Smith. Mocha, the Java Decompiler. `http://www.brouhaha.com/~eric/software/mocha/`, 1996. Accessed: 2015-05-01.

[33] Mike Strobel. Procyon - Java Decompiler. `https://bitbucket.org/mstrobel/procyon/wiki/Java%20Decompiler`, 2014. Accessed: 2015-05-01.

[34] Egor Ushakov. Fernflower. `https://github.com/fesh0r/fernflower`, 2015. Accessed: 2015-05-01.

[35] Raja Vallée-Rai, Etienne Gagnon, Laurie Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. Optimizing java bytecode using the soot framework: Is it feasible? In DavidA. Watt, editor, *Compiler Construction*, volume 1781 of

Lecture Notes in Computer Science, pages 18–34. Springer Berlin Heidelberg, 2000. URL: http://dx.doi.org/10.1007/3-540-46423-9_2, doi:10.1007/3-540-46423-9_2.

[36] M.J. Van Emmerik. *Static Single Assignment for Decompilation*. University of Queensland, 2007. URL: http://books.google.cz/books?id=lC7ANQAACAAJ.

[37] B. Venners. *Inside the Java Virtual Machine*. McGraw-Hill, 1999.

[38] A. Abram White. Serp. http://serp.sourceforge.net/, 2007. Accessed: 2015-05-01.