Compiling Pascal to Java Bytecode

Mark P. de Boer

Dept. of Mathematics and Computer Science
Vrije Universiteit
Amsterdam, The Netherlands

August 19, 1997

Preface

This document describes an attempt to compile programs written in a programming language other than Java[1] to Java class files so that they can be run on the Java Virtual Machine (JVM). If this can be done it provides a whole new way of testing the security of the JVM and making programs portable. All programs ever written in that other language could then be compiled to Java class files and be run on the JVM to test if it still does what it is supposed to do and does not rely on the source language to be Java.

As the source language other than Java I took the original ISO definition of Pascal. Pascal is not a very complex programming language, but a lot can be done with it. Among other things it contains pointers, records and call by reference parameters.

[1] Java is a trademark of Sun Microsystems, Inc.

Contents

I The Java Virtual Machine 1

1 Introduction 3

2 Internals of the JVM 5
2.1 Types . . . 5
2.1.1 Primitive Types . . . 5
2.1.2 Reference Types . . . 6
2.2 Runtime Data Areas . . . 6
2.3 Frames . . . 7

3 The Java Class File Format 9
3.1 The Class File Structure . . . 9
3.2 Fields in a Class File . . . 10

4 The JVM Instruction Set 13
4.1 Typed Instructions . . . 13
4.2 The Instruction Set . . . 14
4.2.1 Load Instructions . . . 14
4.2.2 Store Instructions . . . 15
4.2.3 Arithmetic Instructions . . . 15
4.2.4 Conversion Instructions . . . 15
4.2.5 Class Instances and Array Instructions . . . 15
4.2.6 Stack Manipulation Instructions . . . 16
4.2.7 Branch and Comparison Instructions . . . 17
4.2.8 Method Instructions . . . 17
4.2.9 Miscellaneous Instructions . . . 18

5 Descriptors 19
5.1 Field Descriptors . . . 19
5.2 Method Descriptors . . . 20

II Compiling Pascal 21

6 The Project 23
6.1 Translating EM to Java Bytecode . . . 23
6.2 C as Source Language . . . 23
6.3 Pascal as Source Language . . . 24
6.4 Problems and Solutions . . . 24

7 Starting Out **27**
7.1 How to Store a Class File 27
7.2 Datastructures to Hold the Class File 27

8 The Empty Program **29**
8.1 Initial `constant_pool[]` Entries 30
8.2 `<init>` 30
8.3 `main` 31
8.4 Fit it in the Compiler 31

9 Declarations **33**
9.1 Variable and Constant Declarations 33
9.1.1 Variables and Constants as "Normal" Java Fields 33
9.1.2 Variables and Constants as Objects or Arrays 35
9.1.3 The Problem with Local Variables 37
9.2 Procedure and Function Declarations 38
9.2.1 How to Compile a Procedure or Function 39
9.2.2 Parameters 40

10 Statements **43**
10.1 Maintaining the Size of the Operand Stack 43
10.2 Expressions 43
10.2.1 Constant Values 44
10.2.2 Variable Values 44
10.2.3 Unary Operators 45
10.2.4 Binary Operators 45
10.3 Assignments 47
10.4 Procedure and Function Calls 47
10.5 `write` and `writeln` Statements 48
10.6 Control Statements 49
10.6.1 If Then Else 50
10.6.2 Other Conditional Statements 51
10.7 Function Returns 55

11 Things Left To Do **59**
11.1 Records 59
11.2 Pointers 60
11.3 `goto` Statements 60

12 Final Remarks **63**
12.1 Conclusion 63
12.2 Personal Experience 64
12.3 Acknowledgments 67

A An Example Program **69**
A.1 Pascal Source Code 69
A.2 Java Class File 69
A.2.1 *jdisco* Output 69
A.2.2 D-Java Output 76

Bibliography **79**

Part I

The Java Virtual Machine

This part describes the backgrounds of the Java virtual machine needed to understand how to compile to a Java class file. It describes how the JVM is structured internally, how a class file is structured, what the instruction set of the JVM is and how field and method descriptors are structured.

Chapter 1

Introduction

Java programs are compiled to Java class files. These class files are system independent files that can be run on a Java Virtual Machine (JVM). If it would be possible to compile other programming languages to Java class files it would provide us with a way to test the security of the JVM and a way to make programs written in this language system independent.

This document describes an attempt to compile programs written in the Pascal programming language to Java class files. It describes how various elements of Pascal can be compiled to Java bytecode. After reading this you should have an idea about how to compile and how not to compile Pascal programs to Java class files.

This document is divided into two parts: Part I describes the JVM and part II describes the actual project and contains chapters describing how I compiled Pascal programs to Java class files.

Part I contains chapters 2 to 5. The Java Virtual Machine's types and runtime data areas are described in chapter 2. Chapter 3 describes the Java class file format. Chapter 4 gives an overview of the JVM instruction set. Chapter 5 describes how field and method descriptors are structured.

Part II contains chapters 6 to 12. Chapter 6 describes the goal of the project and gives an overview of the problems faced during the project and their solutions. Chapter 7 describes how I decided to store the class file in the compiler. How to compile an empty Pascal program is described in chapter 8. In chapter 9 is described how to compile variable and constant declarations and procedure declarations. How statements are compiled is described in chapter 10. Chapter 11 describes how to compile some of the main elements of Pascal I did not implement in the compiler. Chapter 12 contains some final remarks.

Finally, appendix A shows an example Pascal program that is compiled by the compiler and the class file it is compiled into.

Most of the specifics of the Java Virtual Machine described in this document can be found in [JVM].

Chapter 2

Internals of the JVM

The JVM is a stack oriented machine. Method frames are pushed on the stack on method invocation and popped from the stack on method return. A frame contains an operand stack from which JVM instructions take values to operate on and return results to. For example, to add the integers 5 and 6, the instructions could be:

bipush 5	Push integer 5;
bipush 6	Push integer 6;
iadd	Pop two integers, add them, and push result.

The JVM instructions are called *bytecodes* and are stored in *class files*.

In this chapter the types the JVM operates on and the runtime data areas of the JVM are described.

2.1 Types

The JVM operates on two kinds of types: *primitive types* and *reference types*.

2.1.1 Primitive Types

There are two kinds of primitive data types supported by the JVM; *numeric types* and the `returnAddress` type.

The numeric types consist of the *integral types* :

- `byte`, 8-bit signed two's-complement integers;
- `short`, 16-bit signed two's-complement integers;
- `int`, 32-bit signed two's-complement integers;
- `long`, 64-bit signed two's-complement integers;
- `char`, 16-bit unsigned integers representing Unicode version 1.1.5 characters.

and the *floating-point types* :

- `float`, 32-bit IEEE 754 floating-point numbers;
- `double`, 64-bit IEEE 754 floating-point numbers.

Operations on values of a numeric type include arithmetic operations, comparison operations and conversion operations.

The values of the `returnAddress` type are pointers to the opcodes of JVM instructions, and it is not a Java language type like the numeric primitive types. There are no operations on values of the `returnAddress` type. This type is used by the JVM instructions *jsr*, *jsr_w* and *ret*, which jump to and return from a subroutine using a value of the `returnAddress` type to save the address of the instruction to return to after the subroutine.

There is no boolean type; operations on booleans are compiled to use the `int` data type to represent boolean values.

2.1.2 Reference Types

There are three kinds of reference types: class types, interface types, and array types, whose values are references to dynamically created class instances, arrays, or class instances or arrays that implement interfaces. A reference value may also be the `null` reference, a reference to no object.

Operations on values of a reference type include comparison to the `null` reference and equality comparison; there are no arithmetic operations on values of a reference type.

2.2 Runtime Data Areas

This section describes the various runtime data areas of the JVM. These data areas are:

1. **pc** register; each thread has its own program counter register which contains the the address of the JVM instruction currently being executed and is one word wide.
2. **stack**; each thread has its own stack which stores JVM frames and is equivalent to the stack of a conventional programming language such as C. The frames are described in section 2.3.
3. **heap**; shared among all threads from which memory for all class instances and arrays is allocated and garbage collected.
4. **method area**; shared among all threads and is analogous to the storage area for compiled code of a conventional language. It stores per class structures such as the constant pool and the code for methods and constructors.
5. **constant pool**; a per-class or per-interface runtime representation of the `constant_pool[]` table in a Java class file which has a similar function as a symbol table of a conventional programming language.
6. **native method stacks**; which may be used to support methods written in languages other than Java.

2.3 Frames

The JVM stack is divided into frames. A new frame is created each time a method is invoked. A frame is destroyed when its method completes. Each frame contains space allocated for local variables and an operand stack.

The local variables are stored into an array of words. Each local variable is one word wide, except for variables of type `long` or `double` which need two words. Local variables are addressed as word offsets from the base of the array.

The operand stack of a frame is the place where JVM instructions take their values from and return their results to. It is also used to pass arguments to methods and receive method results. Each entry on the operand stack is one word wide. Values of type `long` or `double` are pushed onto the operand stack as two words. Values on the operand stack must be operated on according to their types. Most of the JVM instructions that operate on the operand stack are only allowed to operate on one type; it is incorrect to let them operate on any type other than the allowed type. If this is done, the JVM should give an error complaining about it. This also means for example that it is not allowed to push two values of type `int` onto the stack and treat them as a `double`.

At any point in a thread of control there is only one frame that is active, the *current frame*. This is because there can only be one active method at any point, the *current method*. When a method returns, it passes its method result, if any, back to the previous frame, the one of the calling method. The frame of the returning method is then destroyed.

Chapter 3

The Java Class File Format

To give a clear idea about what a class file is, this chapter describes the various fields contained in a class file.

A Java class file consists of a stream of 8-bit bytes. All bigger quantities are constructed by reading in the right amount of consecutive bytes. The Java class files have a very clear format. This chapter describes the structure of the Java class file and its main fields. A full description can be found in [JVM].

3.1 The Class File Structure

A Java class file contains the following structure:

```
ClassFile
{
    u4              magic;
    u2              minor_version;
    u2              major_version;
    u2              constant_pool_count;
    cp_info         constant_pool[constant_pool_count-1];
    u2              access_flags;
    u2              this_class;
    u2              super_class;
    u2              interfaces_count;
    u2              interfaces[interface_count];
    u2              fields_count;
    field_info      fields[fields_count];
    u2              methods_count;
    method_info     methods[methods_count];
    u2              attributes_count;
    attribute_info  attributes[attributes_count];
}
```

The types u2 and u4 represent an unsigned two-, or four byte quantity, respectively. The fields of this structure are described in the next section.

3.2 Fields in a Class File

`magic`

This field holds the magic number `0xcafebabe` that identifies a Java class file.

`minor_version, major_version`

These fields hold the minor and major version numbers of the compiler that produced this class file. A JVM supports class files having a given major version number and minor version numbers 0 through some particular `minor_version`.

`constant_pool[]`

This is a table of `constant_pool_count` - 1 structures representing various constant values, names, descriptors and other things that are referred to within the class file. There is no entry 0; this entry is reserved for internal use by a JVM. Each of the entries have a tag to indicate its type.

The following table contains a list of the types and their description:

Constant type	Description
`CONSTANT_Class`	Represents a class
`CONSTANT_Fieldref`	Represents a field of a class
`CONSTANT_Methodref`	Represents a method of a class
`CONSTANT_InterfaceMethodref`	Represents a method of an interface
`CONSTANT_String`	Represents a `java.lang.String`
`CONSTANT_Integer`	Represents an `int`
`CONSTANT_Float`	Represents a `float`
`CONSTANT_Long`	Represents a `long`
`CONSTANT_Double`	Represents a `double`
`CONSTANT_NameAndType`	Represents a field or a method
`CONSTANT_Utf8`	Contains an UTF-8 string

Structures of type `CONSTANT_Long` and `CONSTANT_Double` take up two entries in the `constant_pool` for historic reasons.

`access_flags`

This is a mask of modifiers used with class and interface declarations. The following flags are defined:

Value	Name	Meaning	Used by
0x0001	`ACC_PUBLIC`	public	Class, interface
0x0010	`ACC_FINAL`	final	Class
0x0020	`ACC_SUPER`	use new *invokespecial*	Class, Interface
0x0200	`ACC_INTERFACE`	interface	Interface
0x0400	`ACC_ABSTRACT`	abstract	Class, Interface

`this_class`

This is an index into the `constant_pool[]`. The entry at that index must be a `CONSTANT_Class_info` representing the class or interface defined by this file.

`super_class`
For a class the value of this field is either zero, in which case this class file represents the class `java.lang.Object`, or an index into the `constant_pool[]`. The entry at that index must be a `CONSTANT_Class_info` representing the superclass of the class defined by this class file.

For an interface the value of this field must always be an index into the `constant_pool[]`. The entry at that index must be a `CONSTANT_Class_info` representing the class `java.lang.Object`.

`interfaces[]`
Each value in this table with `interfaces_count` entries must be an index into the `constant_pool[]`, the entry at that index must be a `CONSTANT_Class_info` representing an interface which is a direct superinterface of this class or interface.

`fields[]`
Each entry in this table with `fields_count` entries is a complete description of a field.

`methods[]`
Each entry in this table with `methods_count` entries gives a complete description of and JVM code for a method in the class or interface.

`attributes[]`
Each entry in this table with `attributes_count` entries must be a variable-length attribute structure. For the class file the only attribute defined is the `SourceFile` attribute containing the name of the source file.

The fields described here are the main fields of the class file structure. The other fields (i.e. the fields in the `constant_pool[]`, `fields[]`, `methods[]` and `attributes[]` tables) are described in [JVM].

Chapter 4

The JVM Instruction Set

4.1 Typed Instructions

Most of the JVM instructions operate on one specific type. Almost all those instructions have the type they operate on encoded in their name:

Letter	Type
i	`int`
s	`short`
l	`long`
d	`double`
f	`float`
c	`char`
b	`byte`
a	`reference`

For example, the instructions used to add two values of type `int`, `long`, `double` or `float`, are respectively *iadd*, *ladd*, *dadd* and *fadd*.

There is no JVM instruction that operates on type `boolean`. All variables of type `boolean` use the `int` data type, and therefore also the `int` instructions. The *newarray* instruction that is used to create arrays of some kind, can create arrays of type `boolean`, but there are no instructions to access or modify elements of `boolean` arrays. Instead instructions that operate on type `byte` are used.

There are also not many instructions for types `byte`, `char` and `short`. For writing to local variables and operand stacks, values of types `byte` and `short` are sign-extended internally to type `int` and values of type `char` are zero-extended to type `int`. This means most of the operations on values of these types are performed by instructions operating on type `int`.

All JVM instructions have an opcode size of one byte and so do all the operands. Remember that values of type `long` or `double` take up two positions on the operand stack (i.e. *value.word1, value.word2*).

If an entry in the `constant_pool[]` has an index that can not be contained in one byte, it has a *wide index*. Because the size of an operand is only one byte, the index is divided into two operands when an instruction needs this index as its operand. From these operands (*indexbyte1* and *indexbyte2*) the index can be reconstructed by the formula $(indexbyte1 \ll 8) | indexbyte2$.

For branching instructions, the same formula is used to construct the signed 16-bit offset out of the operands *branchbyte1* and *branchbyte2*. If the offset is a *wide* offset, a signed 32-bit offset, it is constructed out of the operands *branchbyte1*, *branchbyte2*, *branchbyte3* and *branchbyte4* by the formula $(\mathit{branchbyte4} \ll 24)|(\mathit{branchbyte3} \ll 16)|(\mathit{branchbyte2} \ll 8)|\mathit{branchbyte1}$.

When the index of local variables in the local variable table is a wide index, the *wide* instruction is used to let an instruction that normally has a one-byte index as its operand have a wide index as its operands. The *wide* instruction is discussed in subsection 4.2.9.

4.2 The Instruction Set

This section gives an overview of the JVM instruction set. Only non-trivial instructions that are important for this document are mentioned. For a more complete description of all the JVM instructions see [JVM].

4.2.1 Load Instructions

The load instructions load a value on the operand stack.

The following instructions load a value from a local variable on the operand stack:

- *iload*, *fload*, *lload*, *dload*, *aload*. Load respectively a value of type `int`, `float`, `long`, `double` and `reference` from a local variable at position *index* in the local variable table onto the operand stack. Operand: *index*. Stack: *... ⇒ ..., value*.

- *iload_<n>*, *fload_<n>*, *lload_<n>*, *dload_<n>*, *aload_<n>*. Load respectively a value of type `int`, `float`, `long`, `double` and `reference` from a local variable at index <*n*> onto the operand stack where <*n*> = (0..3). These are specialized instructions, introduced because it is often needed to access local variables at postitions 0 through 3. No operands. Stack: *... ⇒ ..., value*.

The following instructions load a constant on the operand stack:

- *bipush*. Push a `byte`. Operand: *byte*. Stack: *... ⇒ ..., byte*.

- *sipush*. Push a `short`. Operands: *byte1, byte2*. Stack: *... ⇒ ..., short*.

- *ldc*. Push an item from the `constant_pool[]`. Operand: *index*. Stack: *... ⇒ ..., item*.

- *ldc_w*. Push an item with a wide index from the `constant_pool[]`. Operands: *indexbyte1, indexbyte2*. Stack: *... ⇒ ..., item*.

- *ldc2_w*. Push a `long` or a `double` with a wide index from the `constant_pool[]`. Operands: *indexbyte1, indexbyte2*. Stack: *... ⇒ ..., item*.

- *iconst_<i>*, *fconst_<f>*, *lconst_<l>*, *dconst_<d>*. Push respectively `int` <*i*>, `float` <*f*>, `long` <*l*> and `double` <*d*> on the operand stack, where <*i*> is (0..5), <*f*> is (0..2), <*l*> is (0..1) and <*d*> is (0..1). No operands. Stack: *...* ⇒ *..., value*.

- *iconst_m1*. Push `int` -1. No Operands. Stack: *...* ⇒ *..., -1*.

- *aconst_null*. Push *null* `reference`. No operands. Stack: *...* ⇒ *..., null*.

4.2.2 Store Instructions

The store instructions store a value from the operand stack in a variable.

The following instructions store a value in a local variable:

- *istore*, *fstore*, *lstore*, *dstore*, *astore*. Store a value of respectively type `int`, `float`, `long`, `double` and `reference` on the operand stack into a local variable. Operand: *index*. Stack: *..., value* ⇒ *...*.

- *istore_<n>*, *fstore_<n>*, *lstore_<n>*, *dstore_<n>*, *astore_<n>*. Store a value of respectively type `int`, `float`, `long`, `double` and `reference` on the operand stack into a local variable at index <*n*> where <*n*> = (0..3). No operands. Stack: *..., value* ⇒ *...*.

4.2.3 Arithmetic Instructions

The arithmetic instructions take one or two values from the operand stack, compute a result and push that result on the operand stack.

The available operations include addition, subtraction, multiplication, division, remainder, shifting and logical operations like bitwise or. All of these instructions only operate on values of a specific type.

4.2.4 Conversion Instructions

The conversion instructions convert the numeric type of a value to another. Values of type `int`, `float`, `long` and `double` can be converted to each other, and values of type `int` can be truncated to values of type `byte`, `char` or `short`.

4.2.5 Class Instances and Array Instructions

The instructions presented here create and manipulate class instances and arrays.

The following instructions create an object or array:

- *new*. Create new class instance. Operands: *indexbyte1, indexbyte2*. Stack: *...* ⇒ *..., objectref*.

- *newarray*. Create new array. Operands: *atype*. Stack: *..., count* ⇒ *..., arrayref*.

- *anewarray*. Create new array of `reference`. Operands: *indexbyte1, indexbyte2*. Stack: *..., count* ⇒ *..., arrayref*.

- *multianewarray*. Create new multidimensional array. Operands: *indexbyte1, indexbyte2, dimensions*. Stack: *..., count1, [count2, ...]* ⇒ *..., arrayref*.

The following instructions access a field of a class or class instance:

- *getfield*. Load the value of a field of an object. Operands: *indexbyte1, indexbyte2*. Stack: *..., objectref* ⇒ *..., value*.
- *putfield*. Store a value in a field of an object. Operands: *indexbyte1, indexbyte2*. Stack: *..., objectref, value* ⇒ *...*.
- *getstatic*. Load the value of a field of a class. Operands: *indexbyte1, indexbyte2*. Stack: *...* ⇒ *..., value*.
- *putstatic*. Store a value in a field of a class. Operands: *indexbyte1, indexbyte2*. Stack: *..., value* ⇒ *...*.

The following instructions load or store values from or to an array:

- *baload*, *caload*, *saload*, *iaload*, *faload*, *laload*, *daload*, *aaload*. Load a value of respectively type `byte` or `boolean`, `char`, `short`, `int`, `float`, `long` and `double` from an array. No operands. Stack: *..., arrayref, index* ⇒ *..., value*.
- *bastore*, *castore*, *sastore*, *iastore*, *fastore*, *lastore*, *dastore*, *aastore*. Store a value of respectively type `byte` or `boolean`, `char`, `short`, `int`, `float`, `long` and `double` in an array. No operands. Stack: *..., arrayref, index, value* ⇒ *...*.

There are also instructions to give the length of an array and do type checking on objects.

4.2.6 Stack Manipulation Instructions

Stack manipulation instructions directly manipulate the operand stack.

- *pop*. Pop one word from the operand stack. No operands. Stack: *..., word* ⇒ *...*.
- *pop2*. Pop two words from the operand stack. No operands. Stack: *..., word2, word1* ⇒ *...*.
- *dup*. Duplicate one word on top of the operand stack. No operands. Stack: *..., word* ⇒ *..., word, word*.
- *dup2*. Duplicate two words on top of the operand stack. No operands. Stack: *..., word2, word1* ⇒ *..., word2, word1, word2, word1*.

Other stack manipulation instructions are available to duplicate one or two words on the operand stack and put them a number of words down and to swap two words on the operand stack.

4.2.7 Branch and Comparison Instructions

The branch instructions transfer control between instructions.

There are instructions to compare a value of type `int` on the opernad stack to zero in all possible ways and branch on succes. Other conditional branch instructions include comparing a `reference` on the operand stack to `null` and comparing values of type `int` or type `reference`. All branching instructions have the offset to the instruction to branch to as their operands.

Non-branching comparison instructions are available to compare values of type `long`, `float` or `double`. These instruction push the result of the comparison on the operand stack.

The following instructions are conditional branch instructions used to compile the Java `switch` statement (the `case` statement in Pascal). The *tableswitch* instruction is used when the case values of the `switch` can be efficiently used as indices into the table of target offsets. When the case values of the `switch` are sparse, it becomes more efficient to use the *lookupswitch* instruction. Both instructions only operate on `int` values; values types `float`, `long` and `double` have to be converted to `int` first.

- *tableswitch*. Access jump table by index and jump. Operands: *<0-3 byte pad>, defaultbyte1, defaultbyte2, defaultbyte3, defaultbyte4, lowbyte1, lowbyte2, lowbyte3, lowbyte4, highbyte1, highbyte2, highbyte3, highbyte4, jump offsets....* Stack: *..., index* ⇒ *....*

- *lookupswitch*. Access jump table by key match and jump. Operands: *<0-3 byte pad>, defaultbyte1, defaultbyte2, defaultbyte3, defaultbyte4, npairs1, npairs2, npairs3, npairs4, match-offset pairs....* Stack: *..., key* ⇒ *....*

Besides instructions to jump to and return from a subroutine, there also are the following unconditional branch instructions:

- *goto*, *goto_w*. Branch always (normal offset and wide offset). Operands *goto*: *branchbyte1, branchbyte2*. Operands *goto_w*: *branchbyte1, branchbyte2, branchbyte3, branchbyte4*. No change on the operand stack.

4.2.8 Method Instructions

The instructions presented here are used for method invocation or return.

- *invokevirtual*. Invoke instance method. Operands: *indexbyte1, indexbyte2*. Stack: *..., objectref, [arg1, [arg2, ...]]* ⇒ *....*

- *invokespecial*. Invoke instance method, special handling for superclass, private and instance initialization method invocations. Operands: *indexbyte1, indexbyte2*. Stack: *..., objectref, [arg1, [arg2, ...]]* ⇒ *....*

- *invokeinterface*. Invoke interface method. Operands: *indexbyte1, indexbyte2, nargs, 0*. Stack: *..., objectref, [arg1, [arg2, ...]]* ⇒ *....*

- *invokestatic*. Invoke class method. Operands: *indexbyte1, indexbyte2*. Stack: *..., [arg1, [arg2, ...]]* ⇒ *....*

- *return*. Return `void` from method. No operands. Stack: *...* ⇒ *[empty]*.

- *ireturn*, *freturn*, *lreturn*, *dreturn*, *areturn*. Return respectively `int`, `float`, `long`, `double` and `reference` from method, where *ireturn* is also used to return from methods of types `byte`, `char` and `short`. No operands. Stack: *..., value* ⇒ *[empty]*.

4.2.9 Miscellaneous Instructions

Besides instructions for synchronization, there is also an instruction that can be used to modify the behaviour of another instruction so that it can operate on a local variable with a wide index. This instruction has two formats: The first format takes as its operands the opcode of one of the instructions *iload*, *fload*, *lload*, *dload*, *aload*, *istore*, *fstore*, *lstore*, *dstore*, *astore* or *ret* and the bytes *indexbyte1* and *indexbyte2* to construct the wide index from.

The second format takes as its operands the *iinc* instruction, the bytes *indexbyte1* and *indexbyte2* to construct the wide index from of the local variable that has to be incremented, and the bytes *constbyte1* and *constbyte2* to construct a signed 16-bit constant to increment the local variable with.

The effect of the *wide* instruction is that these instructions can now access a local variable with a wide index and, in case of the second format, the local variable can be incremented with a larger constant.

Another instruction is the *athrow* instruction, used to throw exceptions. It requires the `reference` to the object which is an exception to be on top of the operand stack. This *objectref* is popped from the operand stack and is then thrown by searching if the current frame can catch the exception. If it can, the operand stack is cleared, *objectref* is pushed back on the operand stack and execution continues with the first instruction of the code that handles the exception. If the current frame does not contain a `catch` clause, that frame is popped, the frame of the invoker is reinstated, and the *objectref* is rethrown.

Because exceptions are objects, they can have parameters, which differs from exceptions in a programming language such as Ada where this is not possible. This means exceptions in Java can contain information just as objects can contain information.

Chapter 5

Descriptors

Descriptors in a class file are strings representing the type of a field or method. They are built according to the grammar described below. Nonterminal symbols in the grammar are shown in *italics* and terminal symbols are shown in **bold** and are all ASCII characters.

5.1 Field Descriptors

Field Descriptors are used to represent the type of a class or instance variable. They are constructed using the following grammar:

FieldDescriptor: FieldType

FieldType: BaseType | *ObjectType* | *ArrayType*

BaseType: **B** | **C** | **D** | **F** | **I** | **J** | **S** | **Z**

ObjectType: **L<classname>;**

ArrayType: [*ComponentType*

ComponentType: FieldType

The **<classname>** represents a fully qualified class name in modified internal form. This means that all periods ('.') are replaced by forward slashes ('/'). For example the fully qualified name `java.lang.Object` is replaced by `java/lang/Object`.

The following table shows the field types and their meaning:

FieldType	Primitive Type	Meaning
B	`byte`	byte
C	`char`	character
D	`double`	double
F	`float`	float
I	`int`	integer
J	`long`	long
S	`short`	short
Z	`boolean`	boolean
L<classname>;	...	instance of class
[	...	one array dimension

For example, the descriptor of a field of type `boolean` is **Z**. The descriptor of a field of type `String` is **Ljava/lang/String;**. The descriptor of a field that is a four dimensional `int` array is [[[[**I**.

5.2 Method Descriptors

Method descriptors are used to represent methods. They describe the types of the parameters and the return type and are constructed according to the following grammar:

MethodDescriptor: (*ParameterDescriptor**)*ReturnDescriptor*

ParameterDescriptor: *FieldType*

ReturnDescriptor: *FieldType* | **V**

A return type of **V** indicates that the method is of type `void`.
For example the method

```
int Sum(int a, int b)
```

is described by the descriptor

(II)I

The method

```
void Main(String[] args)
```

is described by the descriptor

([Ljava/lang/String;)V

Part II

Compiling Pascal

This part describes the actual project. After a chapter explaining why Pascal was chosen as the source language, there will be several chapters about how I compiled Pascal source programs to Java class files and the problems I had. I will conclude this document with a chapter about some of the things I did not implement and how they should be done and a chapter containing some final words about the project including what can be concluded from all this.

All throughout the project I used the Java compiler `javac` and the Java Virtual Machine `java` from the *Java Developer's Kit*. In the beginning I used *JDK1.0.2* and later I used *JDK1.1.2*.

Chapter 6

The Project

The original goal of this project was to write a back-end for the ACK C compiler. This could not be done because of some practical problems, described in section 6.1. An alternative was to change the ACK C front-end so that it generates Java bytecode. In section 6.2 is described why this can not be done. Another alternative was to change the ACK front-end of another programming language, this is described in section 6.3.

The last section of this chapter gives a short overview of the major problems and their solutions.

6.1 Translating EM to Java Bytecode

The ACK C compiler compiles programs written in C to EM, an intermediate code described in [ACK]. The back-end had to translate the EM code to Java bytecode. After studying the Java bytecode and the EM code it became clear that this was not possible. The problem was that the Java bytecode has typed instructions. This means that these instructions encode type information about the values they operate on. For example, the instruction to load the contents of an `int` variable onto the operand stack is the *iload* instruction. There also is a *lload* , a *fload* , a *dload* and an *aload* instruction for types `long`, `float`, `double` and `reference` respectively.

In the EM code there is no type information whatsoever, so mapping the EM instruction set to the JVM instruction set was doomed to fail. There is a code optimizer for EM code which tries to retrieve the types so it can do type specific code optimization, but it does not always succeed in the retrieval of those types. In that case it just gives up and just does not optimize that specific part of the code, so that is of no use to us.

So when there is no type information in the EM code, there is no back-end from EM to Java bytecode. The only other option was to change a specific front-end so that instead of generating EM code it generates Java bytecode.

6.2 C as Source Language

The goal of the project was now to change the ACK C front-end so that instead of generating EM code it generates Java bytecode. This was not possible because of yet another problem: There is no way of doing pointer arithmetic in the Java bytecode.

Because there are no pointers in Java or in the bytecode, only references to objects, pointers have to be simulated by those references. The problem with this is that there is no way to know the address of an object in the Java bytecode. The only thing you know is that the space needed for the object is allocated from the heap, and that there is a reference pointing to it. There is no way to get the address of the object in the heap, and so there is no way to manipulate the address of the object. The references to objects are of type reference, which means that they can not be loaded with, for example, an instruction to load an integer, and so there is no way to do any arithmetic with the reference. In C you can just about do anything with addresses of variables and constants, so compiling C directly to Java bytecode was going to be a problem.

A way to solve this problem was to simulate your own memory in the Java bytecode in which you do all the allocating and freeing yourself, but that would mean that you would create your own instructions for allocating and freeing memory and for loading and storing values. This way there would not be a lot of real JVM instructions left to use, and you take away the whole essence of compiling something to Java bytecode that is not meant for it and using the instructions for things they are not made for.

6.3 Pascal as Source Language

In the end we chose to take a different source language: Pascal. Pascal has no pointer arithmetic and is still fairly widely used, so a lot of people can compile their programs to Java class files when the compiler is done and this way test the JVM and make their programs portable.

The goal of the project was now to change the ACK Pascal front-end so that instead of generating EM code it generates Java bytecode.

6.4 Problems and Solutions

This section summarizes the major problems encountered when compiling Pascal to Java bytecode and their solutions. Full treatment is given in the subsequent chapters.

No call-by-reference in Java
: To create references to variables, they are compiled into arrays of one element. The reference to the array can then be passed in case the variable has to be passed by reference. See section 9.1.

No nesting in Java
: Because there is no nesting in Java or in the bytecode, nested procedures and functions have to be made global. This also means that local variables and constants have to be compiled in such a way that nested procedures and functions can still access the local variables and constants of the procedure or function they are declared in. This has to be done by passing the local variables as call-by-reference parameters to every nested procedure and function. Local constants are made global, which saves passing them around. See section 9.2.

No symbolic labeling in the bytecode

All jumps in the bytecode jump to an offset. An algorithm is designed that sets internal labels from which the offsets for the jump instructions are resolved once they are known. See section 10.6.

No non-local goto

In the bytecode the *goto* instruction can only jump inside a frame. This causes a problem with a `goto` statement in Pascal that causes a non-local jump. This problem is solved by using an exception to pass the target label and letting the exception handler complete the jump. See section 11.3.

Chapter 7

Starting Out

This chapter describes the design decisions I made for the storage of the class file.

7.1 How to Store a Class File

Before you can compile a program you have to decide where to put the generated code while the compiler runs. The original ACK Pascal front-end stores the generated code in a temporary file on disk and every time it generates code, it writes it to that file. In case of an error, the temporary file can be removed and when no error has occurred, the file can be renamed if necessary. Another way is to write all generated code to memory. In case of an error you just free the memory or else you write the generated code to disk.

The disadvantage of storing the generated code directly on disk is that it is not easy to search in the code or insert something in the middle of it. This is a big disadvantage in case of a Java class file. The `constant_pool[]` in a class file is a field that is almost at the beginning of the file, but all throughout the compiling process entries can be added to it. So when you store the class file on disk, you have to go through a some trouble to insert the `constant_pool[]` entries. A similar problem is that certain fields like `constant_pool_count` are only known when you are done compiling, which means they have to be changed in the file at the end of the compiling process.

This can all be done a lot easier if you store the generated code in one or more datastructures which you hold in memory until you are done and finally write the datastructures to a file on disk. When storing the generated code in datastructures it is simpler to insert fields in the code because you can make use of pointers. It is also very simple to change the fields of the class file. For these reasons I chose this option, which means that you have to come up with datastructures to hold the generated code.

7.2 Datastructures to Hold the Class File

When you want to hold a class file in memory, the only fields you actually have to store are the `constant_pool[]`, the `fields[]` and the `methods[]`. The rest of the fields are either always the same, for example the `magic` field, or can be derived from the three stored fields, for example the `constant_pool_count` field.

The `fields[]` and the `methods[]` have basically the same structure, so they can be stored using the same datastructure. This leaves only two datastructures to be designed: One for the `constant_pool[]` and one for the `fields[]` and `methods[]`.

For the `constant_pool[]` I designed a linked list of entries. Each entry, called `cp_entry`, contains a pointer to the next. In the compiler there is a global variable `cp` which is a pointer to the first `cp_entry` in the list.

The datastructures to hold the `fields[]` and `methods[]` are also linked lists of entries. Each entry, called `m_f` can hold a complete description of a field or a method. Each `m_f` structure contains a pointer to the next. For the `fields[]` there is a global variable `f` in the compiler that is a pointer to the first `m_f` entry of the list of fields. For the `methods[]` there is a global variable `m` that is a pointer to the first `m_f` entry of the list of methods.

The instructions of a method are stored in a list of `instruct` structures. Each `instruct` structure contains a pointer to the first entry in the list of operands of that instruction. It also contains the fields `idx`, that contains the index of the instruction in the `code[]` of the method, and `jmp_label`, that is used by the labeling algorithm discussed in subsection 10.6.

When all the code is generated, no errors have been found and the structures in memory are filled, it is time to write the class file to disk. This is done by writing all the necessary fields to a file with the name of the program as the basename followed by the extension `.class`. So in case of a program called `empty`, the name of the class file will be `empty.class`.

Chapter 8

The Empty Program

I started out by trying to compile an empty Pascal program. By this I mean a Pascal program with no statements or declarations. It looks like this:

```
program empty;

begin
end.
```

Before I could do this, I had to know exactly how a class file of an empty Java program looks. This is not described in [JVM], because it only describes what kind of fields a class file contains, not what these fields should contain, this can only be found out experimentally by looking at existing class files. This means that I had to find a way to represent the binary class files in such a way that I could read them. Because I could not find a program that showed me all the fields in the file, I decided to write one myself. The program I wrote, I called it *jdisco*[1], parses a class file and dumps all the fields in a readable layout on the screen. It does not resolve internal references in the class file, like references to `constant_pool[]` entries for example, it just writes the class file the way it is.

To find out what the contents of an empty Java class file is, I wrote an empty Java program and compiled it with the Java compiler. This is what an empty Java program looks like:

```
class Empty {
    public static void main(String[] args){}
}
```

With the help of *jdisco* I found out that a class file of an empty program contains the following items:

- Some initial `constant_pool[]` entries;
- The instance initialization method `<init>`;
- The `main` method.

Each of these items will be discussed in the following sections. After that I will describe what I did with this information.

[1] *jdisco* could mean something like *Java DISassembling COmpanion*

8.1 Initial constant_pool[] Entries

There are several entries in the constant_pool[] that have to be present in every class file. These entries are:

- a CONSTANT_Class_info describing the superclass, in our case the class java.lang.Object;
- a CONSTANT_Class_info describing this class, in case of a Pascal program that is the name of the program. This way a Pascal program becomes one Java class;
- a CONSTANT_Methodref_info entry describing the instance initialization method <init> of the class java.lang.Object, the superclass of this method.

Other entries needed by the above entries, like the CONSTANT_Utf8_info entries containing the names and CONSTANT_NameAndType_info entry needed by the CONSTANT_Methodref_info entry describing <init>, and entries containing the name and descriptor of the main method are of course also present.

Something I did not expect was that javac also put the CONSTANT_Utf8_info entries in the constant_pool[] containing the names of the attributes (Code, Exceptions, LineNumberTable, LocalVariableTable, SourceFile and ConstantValue), even if they are not used. This of course does not mean they *have* to be put in there.

8.2 <init>

The instance initialization method <init> also has to be present in every class file. This method has the descriptor ()V and is fully described in the methods[] table of the class file.

The entry in the methods[] table contains the following information:

- No access flags;
- The index of the CONSTANT_Utf8_info entry in the constant_pool[] containing the string "<init>", the name of the method;
- The index of the CONSTANT_Utf8_info entry in the constant_pool[] containing the string "()V", the descriptor of the method;
- A Code attribute containing the following information:
  - Maximum number of words on the operand stack is 1;
  - Maximum number of locals is also 1, the reference to this class;
  - The following lines of code:

aload_0	load the reference to this class on the stack;
invokespecial <index>	invoke java.lang.Object.<init>()V;
return.	

What the <init> method of this class does is invoke the <init> method of its superclass, in this case java.lang.Object. The class java.lang.Object will be the superclass of all classes containing a compiled Pascal program.

8.3 main

The last item that has to be present in a class file that can be run as an application is the main method. The main method in Java must be declared public, static and void and must have only one parameter of type String[] containing the *program arguments*.

The main method is also fully described in an entry in the methods[] table of the class file which contains the following information:

- Access flags ACC_PUBLIC and ACC_STATIC;
- The index of the CONSTANT_Utf8_info entry in the constant_pool[] containing the string "main";
- The index of the CONSTANT_Utf8_info entry in the constant_pool[] containing the string "([Ljava/lang/String;)V", the descriptor of the main method;
- A Code attribute containing the following information:
  - Maximum number of words on the operand stack is 0;
  - Maximum number of locals is 1, the parameter;
  - The following line of code:

    return.

8.4 Fit it in the Compiler

To put the initial entries in the constant_pool[] I made an initialization function that creates the list of cp_entry structures and returns a pointer to the first entry so it can be assigned to the global variable cp.

The function has one parameter, the name of the Pascal program. It creates the following entries, plus the entries they need, like CONSTANT_Utf8_info entries containing the names of the classes, methods, descriptors, etc.:

- A CONSTANT_Class_info entry for the class java.lang.Object;
- A CONSTANT_Class_info entry for this class, the program;
- A CONSTANT_Methodref_info entry for the <init> method of the class java.lang.Object, and the CONSTANT_NameAndType_info entry it needs;
- The CONSTANT_Utf8_info entries containing the names of the attributes that will be used by the compiler.

These entries will always be on fixed positions, which is easy when it is needed to refer to them by their index. The three attributes ConstantValue, Code and Exceptions are the only attributes that will be used by the compiler, the attributes LineNumberTable. LocalVariableTable and SourceFile are optional attributes and therefore will be forgotten for the moment.

To put the <init> method in the methods[] table, I also made a initialization function that creates the list of m_f structures and returns a pointer to the first entry so it can be assigned to the global variable `m`. Only the `<init>` method is put in the `methods[]` table by this function, the entry for the `main` method will follow later while the parser runs.

Because there are no variables or constants in an empty class file, the `fields[]` table stays empty for now. Just to initialize the global variable `f`, the pointer to the first entry of the `fields[]` list, I made an initialization function that creates an empty `m_f` structure as the first entry of the list and returns its address.

These three functions are called right after the program identifier has been parsed. The `main` method is inserted in the `methods[]` table after the semicolon following the program heading has been read. This is done by a function that returns a pointer to the `m_f` structure containing the `main` method. This way, when there is code that has to be added to the method, you do not have to search for the right entry in the `methods[]` table `m`. This means that you do have to pass the pointer around in the parser to every procedure that might need it.

The *return* instruction in the `main` method is inserted in the `code[]` list of the `m_f` structure of the `main` method when the final keyword `end` is read, and it is certain there can not be any code left in the Pascal source program.

When the parser returns and no errors have been found, the class file is written and the memory holding the structures is released.

Now we have compiled an empty Pascal program to a Java class file. This provides us with a solid basis to which we can add declarations and statements. The next step will be to compile a Pascal program containing variable and constant declarations.

Chapter 9

Declarations

In this chapter I will describe how I compiled variable and constant declarations and the problems I had doing that. Compiling variables and constants sounds easier than it is...

I will also describe how to compile procedure and function declarations and how to handle forward declarations of procedures and functions.

9.1 Variable and Constant Declarations

In ISO Pascal there are five kinds of constants. There are *integer* constants, *real* constants, *character* constants, *boolean* constants and *string* constants. Variables can be declared to be of one of the primitive types `integer`, `real`, `char` or `boolean`, or they can be of a type declared in a *type declaration*, like an array or a record. For the variables we will only focus on the primitive types for now.

The primitive Pascal types `integer`, `real`, `char` and `boolean`, correspond to respectively types `int`, `float`, `char` and `boolean` in Java, as you might have expected. A *string* constant in Pascal should be of type `java.lang.String` in Java.

At first I thought that compiling the Pascal variables and constants the same way as variables and constants in Java are compiled was the way to do it. A little later it turned out that that was not completely right. The next subsections will tell you why.

9.1.1 Variables and Constants as "Normal" Java Fields

Each compiled global Java variable and constant has an entry in the `fields[]` table of the class file. This entry contains a field representing the access flags of the variable or constant, where one of the flags of a constant should be the `ACC_FINAL` flag. There is also a field containing the index in the `constant_pool[]` of the `CONSTANT_Utf8_entry` representing the name of the variable or constant and a field containing the index of the `CONSTANT_Utf8_entry` representing the descriptor of the variable or constant. Constants also have a `ConstantValue` attribute, containing the index in the `constant_pool[]` of an entry representing the value of the constant. Variables have no attribute.

Each global variable or constant also has a `CONSTANT_Fieldref_info` entry in the `constant_pool[]`. Besides the index of the `CONSTANT_Class_info` entry of the class it is declared in, this entry also contains the index of the `CONSTANT_NameAndType_info` entry that holds the indexes of the `CONSTANT_Utf8_info` entries containing the name and descriptor of the field.

Compiled local variables and constants only have an index into the local variable table. The only way to access them is by this index.

At first I thought that compiling all Pascal variables and constants into normal Java fields was the way to go. I compiled the variables as Java fields of the appropriate type and with access flags `ACC_PUBLIC` and `ACC_STATIC` as if they were declared as Java fields being `public` and `static`. The constants were also compiled as normal Java fields of the appropriate type with the same access flags as a variable plus the flag `ACC_FINAL` and with a `ConstantValue` attribute representing the value of the constant.

The variables and constants have to be declared `static` because the `main` method also has to be declared `static`, and you have to be able to access the fields and methods from the `main` method. The `main` method probably has to be declared `static` because there can be only one `main`, so it is a class method and the fields and methods it can access should also be class fields and methods.

I compiled local variables and constants the same way as the global ones, which also meant making them global. I did this because Pascal has nested procedures and functions and Java does not. This means that if you have a nested procedure or function you have to make this procedure or function global, it is simply the only way. This nested procedure or function still has to be able to access the local variables and constants of the procedure or function it is declared in. One way to do this is to make these local variables and constants also global. The same goes for parameters of procedures and functions, which in effect are local variables.

When you make a local variable or constant global, you have the problem that a global variable or constant with the same name can already exist. This means you have to make the name of the local variable or constant unique. I wanted to do this by prefixing the identifier with a unique number, simply a counter that is incremented each time an identifier is prefixed. This should create a unique identifier because identifiers in Pascal are not allowed to begin with a digit. Identifiers in Java are not allowed to start with a digit either. Surprisingly enough, this is also the case in a class file, where there is a very clear distinction between what is an identifier and what is not. To solve this, I prefixed the digit with the string "`$_`".

Now that I had solved this problem and it all seemed to work perfectly, I thought about procedure and function declarations. This was when I thought of a problem in my variable and constant declarations theory. The problem was the following: Procedures and functions in Pascal can have `var` parameters (call-by-reference parameters) and in Java there are only call-by-value parameters. The problem lies in the fact that when you declare the variables as regular Java fields, there is no reference to pass as actual parameter for a procedure call when the corresponding formal parameter of the procedure is call-by-reference, as Java fields do not have a reference.

The only way to solve this problem is to compile the variable declarations in such a way that you do have a reference. In Java the only references are *objectreferences* and *arrayreference* (besides the references to interfaces that are clearly not relevant here). This leaves us with two options: Compile the variables so that they are objects or so that they are arrays. These two options will be discussed in the next subsection.

9.1.2 Variables and Constants as Objects or Arrays

So the choice is between compiling variables of primitive types into objects or into arrays. Java features so called wrapper classes for primitive types. These classes allow you to store a value of a primitive type in an object of a class of the corresponding type. So if you have a variable of one of the Pascal types `integer`, `real`, `char` or `boolean`, you could compile these into an object of respectively class `Integer`, `Float`, `Character` or `Boolean`. The problem with these wrapper classes is that it is not possible to change the value in the object. This means that you would not be able to change the value of a variable, which is of course not what we want. The only other option left when you want to compile variables into objects is to implement the classes to store the values in yourself, in such a way that it is possible to change the values. These classes could be quite simple, like:

```
Class PascalType {
    public CorrespondingJavaType value;
}
```

You have to read `PascalType` as one of the primitive Pascal types and read `CorrespondingJavaType` as the corresponding primitive Java type. Compiling variables into instances of these classes would provide you with a reference to the variables so you can pass that reference when you have to pass that variable as a call-by-reference parameter. It would also mean that you have to make the necessary `constant_pool[]` entries needed to refer to the field of the object, like a `CONSTANT_Class_info` entry for each of the classes, a `CONSTANT_Fieldref_info` entry plus the entries it needs for each of the classes and a `CONSTANT_Methodref_info` entry for the `<init>` method for each of the classes. Making these `constant_pool[]` entries and implementing classes for each of the primitive types can be avoided by compiling the variables into arrays.

When you compile the variables into arrays of one element of the type of the variable, you also have a reference to the variable and it is quite simple to access or change the value. All you have to do when you want to access it is to load the *arrayref* and index on the operand stack and either load the value from the array or store a value in the array.

This is the option I chose. To handle variables and constants the same way I also compiled the constants into arrays of one element. This way you do not have make a difference when you want to load the value of a variable or a constant. It is of course not necessary as constants do not need references because it is not possible to pass them as call-by-reference parameters. The only constants that are not compiled into arrays of one element are string constants. Strings have to be stored in an object of the class `java.lang.String` which means they already have a reference. It would be unnecessary work to load the value if you store the reference in an array. String constants in Pascal are kind of special anyway because they can never be passed as actual parameters. String constants will just get an entry in the `fields[]` table as the other variables and constants but unlike the other variables and constants this entry will also have a `ConstantValue` attribute.

By making a testprogram in Java and viewing it with *jdisco*, I found out that global variables that are arrays are created in the class initialization method `<clinit>`. This method contains the following sequence of instructions for each array that has to be created:

iconst_1	load the size of the array;
newarray atype	create an array of *atype* with the size on top of the stack;
putstatic index	store the *arrayref* in the global field at *index* in the `constant_pool[]`.

The operand *atype* of the *newarray* instruction is a value indicating the type of the elements in the array.

The operand *index* of the *putstatic* instruction is the index of the `CONSTANT_Fieldref_info` entry in the `constant_pool[]` describing the field.

Because we also want constants to be arrays of one element, and there is no way to initialize an array when it is declared, we have to store the value of the constant in the array in a method. The `<clinit>` method is the right place to do that. For constants the sequence of instructions is therefore a bit different:

iconst_1	load the size of the array;
newarray atype	creates an array of *type* with the size on top of the stack;
dup	duplicate the *arrayref*;
putstatic index	store one *arrayref* in the variable;
iconst_0	load the index of the element to hold the value;
ldc valindex	load the value of the constant on the operand stack;
<t>astore	store the value in the element.

The *dup* instruction is used because we need the *arrayref* twice, the first time to store the reference itself and the second time to store the value.

The *ldc* instruction loads the value of the constant on the operand stack. The value is stored at index *valindex* in the `constant_pool[]`. Every time the compiler comes across a *real* value, an *integer* value or a *string* value in the Pascal program, it stores the value in respectively a `CONSTANT_Float_info` entry, a `CONSTANT_Integer_info` entry or a `CONSTANT_String_info` entry in the `constant_pool[]`. The index of such an entry is the *valindex*. The `ldc` instruction naturally has to be a `ldc_w` instruction when *valindex* is a wide index.

The *<t>astore* instruction stores the value in the element, where *<t>* is the letter for the type of the array, where *i* stands for `int`, *f* for `float`, *c* for `char` and *b* for `boolean`. The instructions to store and load elements of arrays of type `boolean` are the *bastore* and *baload* instructions, the same instruction used for arrays of type `byte`.

To summarize, **variables** are compiled as follows:

- They get an entry in the `fields[]` table;
- They are arrays of one element;

- The arrays are created in the <clinit> method.

Constants, except string constants, are compiled as follows:

- They get an entry in the `fields[]` table;
- They are arrays of one element;
- The arrays are created in the <clinit> method;
- They are initialized to their constantvalue in the <clinit> method.

String constants are compiled as follows:

- They get an entry in the `fields[]` table;
- They are instances of class `java.lang.String`;
- They have a `ConstantValue` attribute.

All variables and constants are also made global, because nested procedures still have to be able to access the local variables of the procedure they are declared in, as I described in the previous subsection. Local variables are made global by prefixing their name with the string "`$_`" followed by a unique number, also described in the previous subsection.

Making the local variables global solves some problems, but is inadequate in the presence of recursive procedures or functions, as will be explained in the next subsection.

9.1.3 The Problem with Local Variables

Compiling local variables and constants to global variables so that they can still be accessed by nested procedures that are also made global seemed like a good idea. For local constants it is, but for local variables it might cause troubles. The problem lies in the fact that when you have a recursive function in which variables are declared, all 'instances' of the function change the same global variables. This will cause problems, because values stored in the variables by one instance can be overwritten by another instance. If that first instance of the procedure is again active and uses these values, incorrect values are used, producing incorrect results.

An example of such a function is:

```
procedure ack(a :integer; var b :integer);

var     i, j    :integer;

begin
    if (a = 1) or (a = 2) then
        b := 1
    else
        ack(a-1, i);
        ack(a-2, j);
        b := i + j;
end;
```

This problem can be solved by not making the local variables global, but keeping them local instead. This causes the problem of nested procedures or functions not being able to access the local variables of the procedure or function they are declared in. This can be solved by passing all local variables as call-by-reference variables to every nested procedure or function. This way these procedures or functions can access the local variables and change them. Because they are passed call-by-reference the real local variables will be changed.

As the local variables stay local, other instances of the procedure or function in case of a recursive call have their own local variables. As it is impossible for one procedure or function to access the local variables of another, it also is not possible for one instance of a recursive procedure or function to change the local variables of another.

Local constants can still be declared as global constants, as they can never be changed. This way you do not have to pass them around as parameters.

Local variables do not have an entry in the `fields[]` table and do not have entries in the `constant_pool[]`. Instead they have an entry in the local variable table of the frame of the method. This also means the arrays are not created the same way as the global arrays. The arrays of the local variables are simply created in the code of the method they are declared in, instead of in the `<clinit>` method.

The first positions in the local variable table of the method are for the parameters of the method. These are automatically allocated by the JVM. In case of a local variable declaration, you simply take the next unused position in the local variable table. At that position you can store any value, provided they are all of the same type. So when you have a declaration of a local variable, you add the following sequence of instructions to the `code[]` array of the method:

iconst_1	load size of array;
newarray atype	create array and push *arrayref*;
astore next_unused_pos	store the *arrayref* in local variable.

The *astore* instruction stores the *arrayref* in the local variable table at position *next_unused_pos*, the next unused position. The next time the *arrayref* is needed, it can be found at that position.

9.2 Procedure and Function Declarations

Now that we know how to compile variable and constant declarations, it is time to move on to procedure and function declarations. As I mentioned before, all procedures and functions have to be compiled into global ones because Java has no nesting. Making the names of local procedures unique goes the same way as I described before with the variable and constant declarations in subsection 9.1.1, by prefixing the name with the string "`$_`", followed by a unique number.

First I will describe how to compile a procedure or function and how to handle forward declarations in the process. After that I will describe how to handle the parameters.

9.2.1 How to Compile a Procedure or Function

A procedure or function is compiled as if it was a method of the program class. For a method we need to put the following items in the class file:

- A `CONSTANT_Methodref_info` entry in the `constant_pool[]`;
- A `CONSTANT_NameAndType_info` entry in the `constant_pool[]`, needed by the `CONSTANT_Methodref_info` entry;
- A `CONSTANT_Utf8_info` entry containing the name of the method, needed by the `CONSTANT_NameAndType_info` entry and the entry in the `methods[]` table;
- A `CONSTANT_Utf8_info` entry containing the descriptor of the method, needed by the `CONSTANT_NameAndType_info` entry and the entry in the `methods[]` table;
- An entry in the `methods[]` table of the class file, containing the complete definition of the method.

In a Pascal program there are *normal* declarations of procedures and functions, and there are *forward* declarations, followed by a *definition* later in the program.

Normal declarations and forward declarations both cause the entries in the `constant_pool[]` and the entry in the `methods[]` table to be created. The function in the compiler that does this returns a pointer to the `m_f` structure in the `m` list, containing the entry in the `methods[]` table. By returning this pointer, later on when instructions have to be added to the `code[]` array of the current method, you do not have to search through the list of `m_f` structures to find the right one. Instead you pass the pointer as a parameter to every procedure or function in the compiler that needs it.

In case of a forward declaration, there is no need to keep the pointer, as you know that the definition of the function or procedure will not follow until later. When the definition of a forward declared procedure or function is encountered, the `m` list is searched for the `m_f` structure of the method to which a pointer is returned so it can be passed as a parameter in the compiler the same way as for a normal declaration.

The descriptor of the method has to be constructed when the list of parameters has been read by the parser. I do this by adding the descriptor of every parameter to a list that in the end contains the complete descriptor of the method. The elements of the list are the following structure:

```
struct descr {
    char            *elt;
    struct descr    *next;
}
```

The list is initialized by creating a first entry containing the string "`(`". After all the entries containing the descriptors of the variables are added, the list is ended by an entry containing the string "`)`" and an entry containing the string "`V`" in case of a procedure, or an entry containing the descriptor of the return type in case of a function. How the descriptors of the parameters and the descriptor of the return type look, will be discussed in the next subsection.

The function in the compiler that adds the method to the class file constructs a single string containing the complete descriptor of the method out of the list, and stores it in a `CONSTANT_Utf8_info` entry in the `constant_pool[]`.

The code of the method has to be ended by an instruction to return from the method. For a procedure this is the *return* instruction. How functions return will be discussed in section 10.7.

9.2.2 Parameters

When a procedure has a call-by-reference parameter, all that has to be passed as an actual parameter is the reference to the variable, in our case the *arrayref*. For a call-by-value parameter the value of the variable or constant has to be passed as the actual parameter. This implies that the descriptor of a call-by-reference parameter should be one of the strings "`[I`", "`[F`", "`[C`" or "`[Z`" for respectively Pascal types `integer`, `real`, `char` and `boolean`. The descriptor of a call-by-value parameter should be one of the strings "`I`", "`F`", "`C`" or "`Z`" for the respective types.

The descriptor of the return type is the same as a descriptor of a call-by-value parameter, as a function needs to return a value, not a reference to a value.

When a procedure has call-by-value parameters, it has to be able to pass them as call-by-reference parameters to another procedure. This is why the values passed as call-by-value parameters have to be converted into arrays. This is done by adding the following sequence of instructions to the `code[]` array of the method that receives the values as call-by-value parameters:

iconst_1	load the size of the array;
newarray atype	create the array;
dup	duplicate the *arrayref*;
astore next_unused_pos	store the *arrayref* in local variable;
iconst_0	load index of the element to hold the value;
<t>load valindex	load the value on the stack;
<t>astore	store the value in the element.

In effect it is the same way as the constants are initialized to their constant-value, except this time we use local variables instead of global constants.

The *<t>load* instruction loads the value from the local variable at *valindex* in the local variable table, where *<t>* is the letter for the type of the array, where *i* stands for `int`, `f` for `float`, `c` for `char` and `b` for `boolean`. The *valindex* is simply the number of the parameter in the parameterlist of the method.

Call-by-reference parameters can just stay the way they are, because they are already references to the variable, and therefore are already arrays of one element.

I used to make the parameters global variables, as I did with the local variables of a procedure or function. I did this because parameters are in effect local variables of a function or procedure. But when you do that, you have the same problem as described in subsection 9.1.3, which should be solved exactly the same way, by passing all parameters as call-by-reference parameters to each of the nested procedures or functions declared in this procedure or function, just as you should do with the local variables.

Chapter 10

Statements

Before being able to compile assignments or other statements, you have to be able to handle expressions. After a section on how to keep track of the number of words on the operand stack, this chapter describes how to generate code for expressions and after that how to compile statements like assignment statements, procedure calls, `write` and `writeln` statements, control statements and how to handle function returns, which I also regard as being statements in Pascal.

10.1 Maintaining the Size of the Operand Stack

Every instruction that is added to the code of a method may have changed the size of the operand stack, which could mean that the variable containing the maximum number of words on the operand stack may have to be updated. To check if it has to be updated, I created a variable `st_size` in the `Code` attribute that keeps track of the current number of words on the operand stack. I also created an array indexed by the opcodes of the instructions that contains all the changes in stack for every instruction. Every time an instruction is added, the change in stack is looked up in the array and added to or subtracted from the variable `st_size`. After that is checked if the current number of words on the operand stack is larger than the value of the variable containing the maximum number of words on the operand stack. If this is so, the maximum number of words on the stack is updated.

There are some instructions that have no constant change in stack, for example the *invokestatic* instruction to invoke a static method where the parameters are popped from the operand stack. For these instructions the minimum change in stack is in the array, and if there is more change in stack, you have to count that yourself, so this can be added or subtracted in addition to the minimum change in stack.

10.2 Expressions

The ACK Pascal compiler is structured in such a way that all expressions are calculated at compile-time as far as possible. For expressions of type `integer` and type `real` the result of these parts of the expression are put in the `constant_pool[]` so that they can be loaded with the `ldc` or `ldc_w` instructions. For results of `boolean` expressions it is not necessary to put it in the `constant_pool[]` as these results can be loaded on the operand stack by the `bipush` instruction because the value can never exceed 255.

As an example of how the compiler calculates expressions in advance, take a look at the following expression:

```
Int1 := 1 + 2 + Three + Int2 + Four + Five;
```

The identifiers `Int1` and `Int2` are variables and the identifiers `Three`, `Four` and `Five` are constants with the respective values 3, 4 and 5. The compiler adds the first 3 elements of the expression as it already knows their values at compile-time. It stores the result of the addition of these three values, the value 6, in the `constant_pool[]`. It also adds the last two elements of the expression, the constants `Four` and `Five`, and stores the result in the `constant_pool[]`. This way the compiler only has to generate code for the addition of three values.

For every expression an expression tree is built, and the procedure in the compiler that generates the code for expressions starts at the root of the tree and generates the code in a *depth-first* manner. The expression tree has the operator with the highest precedence as root. When the code is generated this way, the instruction(s) performing the operation at the root of the tree will be generated after the code for all the subtrees is generated.

In the next subsections I will discuss how the code of expressions can be generated.

10.2.1 Constant Values

Constant values are either values of named constants, values of unnamed constants, like the values 1 and 2 in the expression showed earlier, or values of precalculated results of parts of the expression. The generated code has to load the value on the operand stack.

If the value is of one of the types `integer`, `real` or `string`, the value is stored in the `constant_pool[]` and is loaded on the operand stack by the following code:

ldc index

If the index of the value in the `constant_pool[]` is a wide index, the *ldc_w* instruction is used instead.

If the value is of type `char` or `boolean`, the value is not stored in the `constant_pool[]` because it fits in one byte. Therefore the code to load the value on the operand stack is:

bipush value

10.2.2 Variable Values

To load the value of a variable of a primitive type on the operand stack, which is stored in an array of one element, you first have to push the reference to the array and the position of the element in the array on the stack. After that you can load the value from the array at the right position.

If the variable is a **local** variable, the code that does the above is:

aload index	load the *arrayref* on the stack;
iconst_0	load the index in the array on the stack;
<t>aload	load the value from the array on the stack.

The *index* of the *aload* instruction is the index of the local variable in the local variable table of the method.

The *<t>* of the *<t>aload* instruction is the letter for the appropriate type, where `i` is for type `int`, `f` for type `float`, `c` for `char` and `b` for type `boolean`, so you get one of the instructions *iaload*, *faload*, *caload* or *baload*.

If the variable is **global**, the *getstatic* instruction is used instead of the *aload* instruction. The *getstatic* instruction takes the index into the `constant_pool[]` of the `CONSTANT_Fieldref_info` entry describing the variable as its operands.

10.2.3 Unary Operators

Unary operators operate on one value. In Pascal there are two of these operators, the negation operator '`-`' and the `not` operator. For each of these operators the value to be operated on has to be on top of the operand stack.

To negate '`-`' a value of Pascal type `integer` or `real` you use respectively the instructions *ineg* and *fneg*.

To do the Pascal `not` operation, there is no JVM instruction, so it has to be done by hand. To do a `not` you check if the value on the operand stack is equal to 0, if so you push a 1 in its place and a 0 if it is not.

This is done by the following sequence of instructions:

ifeq 7	if equal to 0, jump 7 bytes (skipping next 2 instructions);
iconst_0	not equal to 0, load a 0 on the stack;
goto 4	jump 4 bytes further (after the next instruction);
iconst_1	load a 1 on the stack.

Every branch instruction, like the *ifeq* and *goto* instructions here, take byte offsets as operands. All the instructions and all their operands are one byte each. See section 4.2 for the number of operands for each instruction.

10.2.4 Binary Operators

Binary operators operate on two values. In Pascal, these are the arithmetic operators, the comparison operators and the boolean operators. All the operators require the values they operate on to be on top of the operand stack, where the value on the right of the operator is on top of the stack and the value on the left is under it.

10.2.4.1 Arithmetic operators

The arithmetic operators pose no problem since they correspond directly to JVM instructions of the appropriate type.

10.2.4.2 Comparison Operators

To compare two values of type `integer`, `boolean` or `char`, the following code is used:

if_icmp<cond> 7	If comparison succeeds jump 7 bytes (skip 2 instructions);
iconst_0	not succeeded, so push a 0;
goto 4	jump 4 bytes (skip next instruction);
iconst_1	succeeded, so push a 1.

The *if_icmp<cond>* instruction is one of the instructions *if_icmplt*, *if_icmple*, *if_icmpgt*, *if_icmpge*, *if_icmpeq* and *if_icmpne* for respectively the Pascal operations ‘<’, ‘<=’, ‘>’, ‘>=’, ‘=’ and ‘<>’.

If the two values of the comparison operation are of type `real` they have to be compared by the *fcmpg* instruction first and the resulting value is compared to zero. The *fcmpg* instruction pushes the value -1 on the stack when the first value is less than the second value, it pushes the value 0 on the stack when they are equal and it pushes the value 1 on the stack when the first value is greater than the second value. The code is:

fcmpg	Compare the two float values;
if<cond> 7	If comparison to zero succeeds jump 7 bytes (skip 2 instructions);
iconst_0	not succeeded, so push a 0;
goto 4	jump 4 bytes (skip next instruction);
iconst_1	succeeded, so push a 1.

The *<cond>* of the *if<cond>* instruction is the same as the *if_icmp<cond>* instruction.

10.2.4.3 Boolean Operators

To do a boolean `and` between two values of type `boolean` you have to design the code yourself. The general idea is that you check to see if the first value is equal to 0, if it is, the result of the `and` is also 0. If the value is 1, you move on to the next value to see if it is equal to 0. If it is, the result of the `and` is also 0. If it is equal to 1, you found that both values are equal to 1 and the result of the `and` is also 1.

The code to do this is:

ifeq 10	if first value is equal to 0, jump 10 bytes (skip 3 instructions);
ifeq 8	if second value is equal to 0, jump 8 bytes (skip 3 instructions);
iconst_1	both values are equal to 1 so push a 1 as the result;
goto 5	jump 5 bytes to skip the next 2 instructions;
pop	pop the second value if the first value was equal to 0;
iconst_0	one or both of the values were equal to 0, so push a 0.

As the code shows, when you know that the first value is 0, you also know that the result of the expression is 0, so you pop the second value from the operand stack and push a 0 as the result of the `and`.

Similarly, the code for a boolean `or` is:

ifne 10	if first value is equal to 1, jump 10 bytes (skip 3 instructions);
ifne 8	if second value is equal to 1, jump 8 bytes (skip 3 instructions);
iconst_0	both values are equal to 0 so push a 0 as the result;
goto 5	jump 5 bytes to skip the next 2 instructions;
pop	pop the second value if the first value was equal to 1;
iconst_1	one or both of the values were equal to 1, so push a 1.

10.3 Assignments

Now the expressions are done, assignment statements are pretty easy. As we have all the variables stored in arrays, we have to generate code that loads the reference to the array and the index on the operand stack. After the code for the expression is generated that loads the value of the expression on the stack, the instruction to store the value in the array has to be generated.

If the variable to assign to is a **local** variable, the code for an assignment is:

aload loc_index	load the *arrayref* in local variable *loc_index* on the stack;
iconst_0	load the index in the array on the stack;
(code to load the value of the expression on the stack)	
<t>astore	store the value of the expression in the array.

The *<t>astore* instruction stores the value in the array, where *<t>* is the letter for the type of the array, where *i* stands for `int`, *f* for `float`, *c* for `char` and *b* for `boolean`.

If the variable to assign to is **global**, the *getstatic* instruction is used instead of the *aload* instruction.

10.4 Procedure and Function Calls

For a procedure or function call you first have to load the parameters on the operand stack, in the order they are declared. For **call-by-value** parameters this means you have to load the value of the actual parameter on the stack, which is an expression. This means that the code to load this value on the stack is as described in section 10.2 about expressions.

The actual parameter of a **call-by-reference** parameter is the reference to the variable. For a local variable the instruction that does this is the *aload* instruction, which takes the index of the local variable in the local variable table of the method as its operand. For a global variable the *getstatic* instruction loads the reference to the variable on the operand stack. It takes the index of the `CONSTANT_Fieldref_info` in the `constant_pool[]`, that describes the variable, as its operands.

Every actual parameter loaded on the operand stack has to be counted, so when all the parameters are on the stack you know how many words were pushed on the stack. These words will be popped from the stack by the instruction that invokes the method, so the variable of the current method in the compiler that contains the current stack size has to be updated with this value. If the method is a function, the variable has to be updated with the number of words of the parameters it pops from the operand stack, and the word containing the returnvalue of the function that is on the stack when the function returns. This means subtracting the value `counter - 1` from the variable in case of a function and value `counter` when it is a procedure. Function returns will be described in section 10.7.

The JVM instruction that invokes a method is *invokestatic*. It takes as operands the index of the `CONSTANT_Methodref_info` entry in the `constant_pool[]` that describes the method. The *invokestatic* instruction is only used to invoke methods that are declared to be `static`, which are all the methods in our case.

10.5 `write` and `writeln` Statements

The ACK Pascal compiler is structured in such a way that a `write`, `writeln`, `read` and `readln` statement containing more than one argument is compiled into several `write` or `read` statements with only one argument. For `readln` and `writeln` these statements are followed by a statement that only reads or writes the end-of-line. So for example the `writeln` statement:

```
writeln('This writeln is compiled into ', 4, ' parts');
```

is compiled as if it were the following four statements:

```
write('This writeln is compiled into ');
write(4);
write(' parts');
writeln;
```

I only implemented `write` and `writeln` statements writing to standard output, so I will only discuss these here.

I wrote Java methods for all the kinds of `write` statements. These methods are all part of class `pcio`, which acts as a library. The following table contains all the kinds of `write` statements in Pascal with one argument, and the corresponding Java method and its arguments. The letters `i`, `r`, `c`, `b` and `f` stand respectively for the types `integer` or `int`, `real`, `char`, `boolean` and `float`.

Pascal Statement	Java Method
`write(r: i: i)`	`pcwrite(f, i, i)`
`write(r: i)`	`pcwrite(f, i)`
`write(r)`	`pcwrite(f)`
`write(b: i)`	`pcwrite(b, i)`
`write(b)`	`pcwrite(b)`
`write(c: i)`	`pcwrite(c, i)`
`write(c)`	`pcwrite(c)`
`write(i: i)`	`pcwrite(i, i)`
`write(i)`	`pcwrite(i)`
`write(s: i)`	`pcwrite(s, i)`
`write(s)`	`pcwrite(s)`
`writeln`	`pcwriteln()`

In the Java source, all the Java methods writing an argument of a numeric type, except for the methods writing an argument of type `float`, call the method writing a string, with their argument converted to type `String`.

After the parameters are loaded on the operand stack, the method is invoked by the *invokestatic* instruction that is also used for normal procedure and function calls. As this instruction (and every other instruction to invoke a method) needs the index in the `constant_pool[]` of a `CONSTANT_Methodref_info` entry describing the method, these have to be created.

A `CONSTANT_Methodref_info` entry in the `constant_pool[]` has to be created for each of the Java methods in the `pcio` library. This means that for the complete library the following entries in the `constant_pool[]` have to be created:

- A `CONSTANT_Class_info` entry describing the `pcio` class;
- A `CONSTANT_Utf8_info` entry containing the string "`pcio`", needed by the `CONSTANT_Class_info` entry;
- A `CONSTANT_Utf8_info` entry containing the string "`pcwrite`";
- A `CONSTANT_Utf8_info` entry containing the string "`pcwriteln`";
- For each of the Java methods a `CONSTANT_Utf8_info` entry containing the descriptor of the method;
- For each of the Java methods a `CONSTANT_Methodref_info` entry containing the index of the `CONSTANT_Class_info` entry of the `pcio` class and the index of the `CONSTANT_NameAndType_info` entry described below;
- For each of the Java methods a `CONSTANT_NameAndType_info` entry describing the method. It contains the index of the `CONSTANT_Utf8_info` entry of the name of the method and the index of the `CONSTANT_Utf8_info` entry of the descriptor of the method.

In the present compiler all these entries are always added to the `constant_pool[]`. I did this for convenience reasons. It would be nicer to put the entries for a method in the `constant_pool[]` only if the method is used.

10.6 Control Statements

Compiling the various control statements requires doing labeling, so you can jump to these labels. The problem with the bytecode is that you do not have any labels. All the jump instructions of the JVM jump to some offset instead of to a label. This means you have to come up with your own labeling in the compiler.

To do labeling I added a `jmp_label` field to the `instruct` structure that holds an instruction and its parameterlist. If you set a label, you add an `instruct` structure to the list with the label set. The label contains the index of the instruction that jumps to it. In all the `instruct` structures with no label, the value of the `jmp_label` field is set to the value -1. Every instruction that is added to the `code[]` list of a method is now inserted before the first `instruct` structure in the list with its label set instead of being added to the end of the list. When a label is set, the `instruct` structure containing the label is also inserted before the first label in the list. This way the first label in the list is allways the label for the last jump instruction added that has an unknown offset.

The list of instructions can now be seen as consisting of two parts: The first part is the list of instructions of the method and the second part is the list of the labels. Every instruction (or label) that is added to the list is added to the end of the list of instructions and before the list of labels, as can be seen in the following figure:

instruction$_1$
$\vdots$
instruction$_n$
instruction$_{n+1}$
label$_m$
$\vdots$
label$_1$

If the index of the instruction to jump to is known, i.e. all the instructions are inserted before the label, the offset is computed by subtracting the index contained in the label (the index of the instruction to jump from) from the index of the `instruct` structure that is labeled (the index of the instruction to jump to). This offset is added as parameter to the instruction to jump from. After this is done, the label can be set to -1 or the whole `instruct` structure can be removed from the list if it did not contain an instruction.

I only implemented `if` statements in the compiler, just to see how the labeling works. These will be discussed in the next subsection. After that I will present ideas on how to compile other conditional statements.

10.6.1 If Then Else

With the labeling algorithm described above it is not very difficult to compile `if` statements. After the code for the expression representing the condition of the `if` has been generated, the *ifeq* instruction is added, with no parameters. The parameters of this instruction should contain the offset for the jump, but as the index of the instruction to jump to (the first instruction following the code of the then-part) is not known, a label has to be set. For this we add an empty `instruct` structure containing the index of the *ifeq* instruction as its label. After the code of the then-part has been generated (remember, code is inserted before any label), there are two possibilities: either the `if` statement is finished or there is an else-part.

If there is no else-part and the `if` statement is finished, all that has to be done is to resolve the label. The index of the instruction to jump to if the condition of the `if` statement evaluates to `false` is the index of the first instruction following the then-part. The index of this instruction is the same as the index of the first `instruct` structure in the list where the `label` field is not -1. This label contains the index of the *ifeq* instruction, the instruction to jump from. Now the index contained in the label is subtracted from the index of the structure holding the label. A parameterlist is constructed from the resulting offset and is added as the parameterlist of the instruction at the index contained in the label, the *ifeq* instruction. After this is done, the `instruct` structure holding the label is removed from the list, and the compiler can continue.

If there is an else-part in the `if` statement however, there is a slightly different scenario. If the condition of the `if` statement evaluates to `true`, the else-part has to be skipped after the code of the then-part has been executed. This means that there has to be a *goto* instruction at the end of the then-part and before the else-part, which causes a jump to the first instruction following the else-part. For this we add a *goto* instruction to the code with no parameters. This instruction is of course inserted before any label. The label it is inserted before has to be the label containing the index of the *ifeq* instruction. As the first instruction of the else-part is the instruction following the *goto* instruction, the label of the *ifeq* instruction can now be resolved and the `instruct` structure holding it removed, just as I described before. After this label is deleted, the label for the *goto* instruction can be added. If it would have been added before the label of the *ifeq* instruction has been deleted, things would have gone wrong. Every time a label is resolved and deleted, the first label in the list is taken; this would have been the label of the *goto*. So the label of the *goto* instruction is added after the label of the *ifeq* instruction has been resolved and deleted, and the code for the else-part can be generated. After all the code for the else-part has been generated, the label for the *goto* can be resolved and deleted the same way as with the *ifeq* instruction before, and the `if` statement is finished.

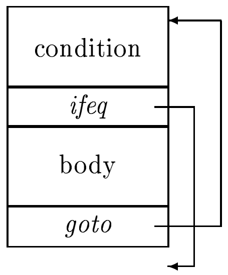
By adding the labels in the correct order and adding all the instructions before the labels, the first label to be deleted is *always* the right one. This is also the case when `if` statements are nested. If the labeling is done the way I described above the labels are always added and deleted in the right order and at the right time.

10.6.2 Other Conditional Statements

Next I will describe how to use the labeling algorithm for the other conditional statements. The labeling algorithm may have to be altered a bit, but the general idea stays the same. Using this labeling algorithm, the conditional statements can be nested any way possible.

10.6.2.1 While

The bytecode for the while statement can be structured as follows:



This can be compiled in the following way:

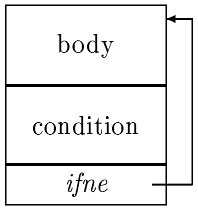
1. Save the index of the next instruction that will be added, we need this later for the *goto* instruction;
2. Add the code for the condition, of which the first instruction is added at the saved index;
3. Add an *ifeq* instruction with no parameters;
4. Add an empty `instruct` structure containing the index of the *ifeq* instruction as its label;
5. Add the code for the body of the `while` statement, which is inserted before the label of the *ifeq* instruction;
6. Add a *goto* instruction with the offset to the saved index as its parameterlist. This offset is computed by subtracting the current index of the *goto* instruction + 1 from the saved index. This instruction is also added before the label of the *ifeq* instruction;
7. Resolve the label of the *ifeq* instruction by subtracting the index contained in the label from the index of the structure holding the label. The resulting offset is constructed into a parameterlist and added to the *ifeq* instruction (which is at the index in the label). Remove the structure holding the label from the list.

By saving the index in the beginning, we saved the index of the first instruction of the code for the condition of the `while` statement. This is the index of the instruction to jump to when the code for the body of the `while` statement has been executed, and the condition has to be checked again.

The *ifeq* instruction causes a jump to the first instruction of the code following the `while` statement in case the condition evaluates to `false`.

10.6.2.2 Repeat

The bytecode for the `repeat` statements can be structured in the following way:



This can be compiled using the same techniques as for the `while` statement.

10.6.2.3 For

Although the Pascal `for` statement looks a bit different than the Java `for` statement, they are quite the same. The Pascal `for` statement can be in one of the two following forms:

`for` *control* `:=` *initial* `to` *final* `do` *(body)*

or:

`for` *control* `:=` *initial* `downto` *final* `do` *(body)*

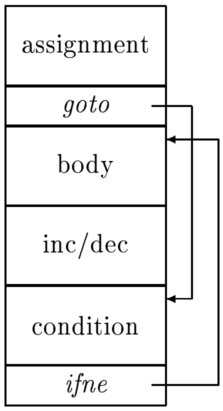
The corresponding Java versions are:

`for(`*control* `=` *initial*`;` *control* `<` *final*`;` *control*`++)` *(body)*

and:

`for(`*control* `=` *initial*`;` *control* `>` *final*`;` *control*`--)` *(body)*

The bytecode for the `for` statement is structured as follows:



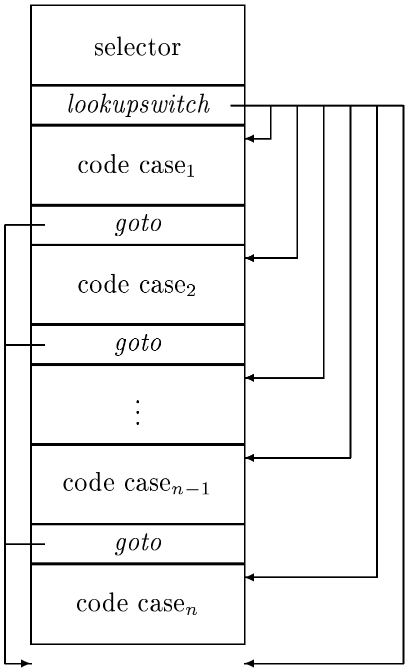
This can also be compiled using the same techniques as for the `while` statement.

10.6.2.4 Case

The JVM has two instructions to handle `case` statements, the *tableswitch* and the *lookupswitch* instruction. The *tableswitch* instruction is used when the case values can be efficiently used as indices into the table of target offsets. When the values are more sparse, it is better to use the *lookupswitch* instruction. This instruction looks up the value of the case in its table of case values and jumps to the offset where the code of that value is. If the value is not in the table, it takes a default offset.

Because the *lookupswitch* instruction will always work, I will use that. The operands of the instruction contain an offset for every case label of the `case` statement. How the list of operands is exactly structured can be found in [JVM]; a short description of the instruction is given in subsection 4.2.7. I will only give an idea on how to compute and fill in the correct offsets when they are known.

The bytecode for a `case` statement may be structured the following way:



- First the code to load the selector on the operand stack;
- After that the *lookupswitch* instruction with the table in its parameterlist;
- After the *lookupswitch* instruction there is a piece of code for every case label. Every piece of code, except the last one, is ended with a *goto* statement to jump to the first instruction of the code following the `case` statement.

As you do not know in advance at what offset the code for a specific case label begins, you can not fill them in in the table. What you can do is allocate the entries in the table, i.e. allocate the `params` structures in the parameterlist of the `instruct` structure containing the *lookupswitch* instruction, and leave the entries for the offsets empty. This way you do know the indexes of the instructions following the *lookupswitch* instruction, so you can fill the offsets in when they are known.

To use the labeling algorithm used with the other conditional statements, we have to change the `jmp_label` field. Instead of containing an `int` value, it must now contain a list of `int` values. When an instruction is added to the list, it is now inserted before the first `instruct` structure of which the `jmp_label` field has a value other than the `null` pointer, instead of a value other than -1.

A `case` statement may now be compiled as follows:

1. Add the code to load the value of the selector of the `case` statement on the operand stack;

2. Add the *lookupswitch* instruction, leaving all the entries for the offsets empty;

3. Save the index of the *lookupswitch* instruction, so you know where to find it when you want to fill in the offsets;

4. Add an empty `instruct` structure to the list, which will serve as the label for the *goto* instructions of all the pieces of code for the case labels. The index of the *lookupswitch* instruction can be added as entry in the label list;

5. For each of the case labels do the following:

   (a) Compute the offset for the case label by subtracting the index of the *lookupswitch* from the index of the next instruction to add, because that is the first instruction of the code for the case label. Fill in the resulting offset at the appropriate entry in the table in the parameterlist of the *lookupswitch* instruction;

   (b) Add the code for the case label, which is inserted before the label of the *goto* instructions;

   (c) Add the *goto* instruction with an empty parameterlist to jump to the first instruction of the code following the `case` statement when the code of this case label has been executed (no *goto* for the last case label). This *goto* instruction is also inserted before the label of the *goto* instructions;

   (d) Add the index of the *goto* instruction to the label list of the first labeled `instruct` structure in the list, which is the structure containing the label list for the *goto* instructions;

6. Resolve the labels of the *goto* instructions by going through the label list and for every entry subtract the index from the current index of the `instruct` structure containing the label list. The resulting index is constructed into a parameterlist which is added to the *goto* instruction at the index contained in the entry in the label list. When the index of the *lookupswitch* instruction is encountered in the label list, compute the offset the same way as usual and insert it into the table in the parameterlist of the *lookupswitch* instruction for the default case label. After all offsets are computed, remove the `instruct` structure from the list.

10.7 Function Returns

In Pascal the returnvalue of a function has to be assigned to the function identifier. This assignment can take place anywhere in the function, even in nested procedures or functions. A function does not return its value until all the code in the function has been executed.

This differs from the function returns in Java where a function returns its value by a `return` statement, at which point the function returns immediately.

In the bytecode there are specific instructions to return a value from a function. To return a value of type `int`, `float`, `long`, `double` or `reference` there are respectively the instructions *ireturn*, *freturn*, *lreturn*, *dreturn* and *areturn*. As usual, the instruction to return a value of type `int` also returns values of types `byte`, `boolean` and `char`. There is also an instruction that returns no value, the *return* instruction we used in subsection 9.2.1 to return from a procedure.

As in Pascal the function result has to be assigned to the function identifier, we need to have a variable to assign to. I made this variable global, but this causes the same problems as described in subsection 9.1.3, which are to be solved the same way, by making the variable local and passing it as a call-by-reference parameter to every nested procedure or function.

Before the function returns, the value of the variable has to be loaded on the operand stack for the JVM instruction that returns it. This causes a problem if the function identifier was not assigned to. It is practically not possible to check at **compile-time** if the function identifier has always been assigned to. Instead the ACK Pascal compiler checks if the function identifier was assigned at least once in the code. This means you have to think of a way to check at **runtime** if the variable contains a value to return.

For this I created a local `boolean` variable that acts as a flag and is passed together with the variable that contains the function result. When the function identifier is assigned to, the variable is set to `true`. Before the function returns it is checked to be set and if so, the return value is loaded on the operand stack and the instruction that returns it is invoked.

If the `boolean` variable contains the value `false`, there is no value for the function to return. This means a runtime error has to be generated. For this I implemented the following Java class:

```
public class FunctionResultNeverAssignedException extends Error
{
    FunctionResultNeverAssignedException()
    {
        super("Function result was not assigned to function " +
              "identifier");
    }
}
```

This class defines a `FunctionResultNeverAssignedException()`. Because it extends class `Error` this exception can never be caught once it is thrown. Because of this, the JVM will stop the execution of the program and will write the name of the exception, the string "`Function result was not assigned to function identifier`" and some additional information about the function the error occurred in.

To call this function in the bytecode I created a `pcrterror` class that contains a method that throws the `FunctionResultNeverAssignedException`. For this class similar `constant_pool[]` entries have to be created as for the `pcio` class described in section 10.5. This means a `CONSTANT_Class_info` entry to describe the class and `CONSTANT_Methodref_info` entries and the entries it needs to describe the methods in the class (only one at this time). The entries are put in the `constant_pool[]` in the same procedure of the compiler that puts the entries for the `pcio` class in.

The bytecode for function return is now as follows:

aload bool_index	load the reference to the `boolean` variable on the stack:
iconst_0	load index into array on the stack;
baload	load the value of the `boolean` variable on the stack;
ifne 6	if value is `true` jump 6 bytes (skip next instruction);
invokestatic exc_index	value is `false` so generate runtime exception;
aload funcres_index	load the reference to the variable containing the function result on the stack;
iconst_0	load index into array on the stack;
<*t*>*aload*	load the function result on the stack;
<*t*>*return*	return the function result.

The <*t*> of the <*t*>*aload* is the letter for the type of the array, the return type of the function, where *i* stands for `int`, *f* for `float`, *c* for `char` and *b* for `boolean`.

If the return type of the function is one of the types `integer`, `char` or `boolean`, the <*t*> of the <*t*>*return* instruction is the letter *i*. If the return type of the function is `real`, it is the letter *f*.

When the value of the `boolean` variable is `false` the instructions following the *invokestatic* instruction that invokes the method to throw the exception, will never be reached. The program will be stopped immediately.

Chapter 11

Things Left To Do

Because I only generate code for the basic elements of the Pascal programming language, there are some elements of Pascal I did not cover. This chapter presents ideas about how some of these elements that are not so obvious to compile to Java bytecode can be compiled.

11.1 Records

Records in Pascal can be compiled as if they are separate classes in Java. For each recordtype defined in a type declaration of a Pascal program, a class has to be created that contains the fields of the record as instance variables.

Making a class also includes writing it to a class file, so the same procedures and structures in the compiler that are used for the class file containing the program can be used for a class file containing a definition of a record, except that the fields in a class defining a record must not be `static`. The fields in the class containing the Pascal program are declared `static` because they can not be accessed by the `main` method if they are not. This is because the `main` method has to be declared `static` (see also subsection 9.1.1).

The fields in a class defining a record must not be declared `static`. This way, every time a variable is declared to be of the recordtype, an instance of the class can be created to represent the variable. This way variables of recordtypes become objects of the class defining the record.

Besides creating a class and writing the class file for every recordtype when the type is declared, entries for the class and its fields have to be created in the `constant_pool[]` of the class file that contains the program. The following entries have to be created for every record class:

- A `CONSTANT_Class_info` entry to describe the class;
- A `CONSTANT_Utf8_info` entry containing the name of the class (i.e. the name of the recordtype). This entry is used by the `CONSTANT_Class_info` entry;
- For each field of the class (record) the following entries have to be created:
  - A `CONSTANT_Fieldref_info` entry to describe the field;

- A `CONSTANT_NameAndType_info` entry, needed by the `CONSTANT_Fieldref_info` entry;
- A `CONSTANT_Utf8_info` entry containing the name of the field, this is needed by the `CONSTANT_NameAndType_info` entry;
- A `CONSTANT_Utf8_info` entry containing the descriptor of the field, this is also needed by the `CONSTANT_NameAndType_info` entry.

With these entries it is possible to access the fields in an instance of the class. For example, the bytecode to store the value `5` in an `integer` field of a local variable of a recordtype is now:

aload var_index	load the *objectref* on the stack;
bipush 5	load `int` value `5` on the stack;
putfield cp_index	store the value in the field of the record.

The *cp_index* is the index in the `constant_pool[]` that describes the field of the record.

To load a value from a field of a variable of a recordtype is:

aload var_index	load the *objectref* on the stack;
getfield cp_index	load the value in the field of the record on the stack.

As before with the variables of primitive types that are actually references to arrays of one element that hold the value, variables of recordtypes are now references to objects that hold the fields of the record.

11.2 Pointers

Because all variables of primitive types are references to arrays, all variables of recordtypes are references to objects and variables of arraytypes are references to arrays, the pointers should not be a problem anymore. Working with a pointer to a variable is now very similar to working with a normal variable. In fact as all variables are already references, the declaration of a pointer variable is the same as a declaration of a non-pointer variable.

Because the JVM has automatic garbage collection, the `dispose` statement, used to free the memory of a variable, can now be ignored.

11.3 `goto` Statements

A Pascal `goto` statement can cause one of two kinds of jumps: It can cause a **local** jump or a **non-local** jump. In case of a local jump, a label in the body of the same procedure or function is jumped to. In case of a non-local jump, a label in the body of another procedure or function is jumped to.

Local jumps can be handled by simply adding the JVM *goto* instruction to the code with as parameterlist the offset to the first instruction of the code that follows the label in the Pascal source. This offset can be computed by maintaining two lists while generating code. The first list contains all the labels parsed in the current procedure or function and the index of the instruction following the label. The other list contains labels instructions want to jump to, and the index of the instruction that wants to jump to it.

If the target label is in the first list, the offset can be calculated immediately, otherwise the label and the index of the *goto* instruction are put in the second list. Every time a label is parsed, it is put in the first list, and it is checked if it is in the second list. If this is so, these offsets can be calculated now, added as a parameterlist to the appropriate *goto* instruction, and that entry can be removed from the second list.

If at the end of the procedure or function there still are entries in the second list, this means these `goto` statements are non-local jumps. Non-local jumps are more difficult because there is no JVM instruction that lets you jump to an instruction in the code of another method. This problem can be solved by using exceptions.

For every label in the second list, instructions have to be added to the end of the code for the method, after the *return*. The offset to the first instruction of this piece of code has to be computed and added as a parameterlist to the *goto* instruction that wants to jump to the non-local label. This piece of code must cause an exception to be thrown that contains the target label.

The Java code for the exception could be:

```
public class GotoException extends Exception
{
    GotoException(int l)
    {
        label = l;
    }

    public int label;
}
```

If the flow of control reaches the *goto* instruction that causes a non-local jump, it causes a jump to the instructions that throw the exception containing the target label.

If a procedure or function has nested procedures or functions, and contains labels that can be jumped to, the code for that procedure or function has to contain code that can handle the exception. This code has to contain a *goto* instruction that can now jump to the label contained in the exception. This way the exception is caught by this procedure or function and the non-local jump has become a local jump.

If the target label is not in the calling procedure or function, and therefore there is no exception handler in the calling procedure or function, the exception mechanisme throws the exception automatically further up the method invocation chain until it reaches a method that can handle the exception, and complete the jump.

The *athrow* instruction causes all the frames of the methods that can not catch the exception to be popped from the stack. When the method is reached that can handle the exception, its operand stack is cleared and the *objectref* of the exception is pushed on it. This is the same as in Pascal where the working stack of the procedure or function is cleared before the execution proceeds at the label that was jumped to.

Chapter 12

Final Remarks

This chapter contains a section about what may be concluded from this project. It also contains a section about what I thought of the project when looking back. I conclude with a few acknowledgments.

12.1 Conclusion

A relatively simple programming language like Pascal should not be too big a problem when you want to compile it to Java bytecode. The variable declarations are probably the most difficult part because Pascal has nested procedures and functions where all variables of the procedure or functions containing these nested procedures or functions should be accessible.

Because nested procedures or functions have to be compiled as if they were global methods the only option seemed to be to declare the local variables global as well. This way they can be used in the nested functions. The problem with this is that in recursive calls of procedures or functions the same global variables are used in every 'instance' of the procedure or function. This may produce wrong results because values of one instance can be overwritten by another instance while the first instance still needs it.

This problem was solved by making local variables local and passing them as call-by-reference parameters to every nested procedure or function so they can be used there as well. For the same reason parameters of a procedure or function that were first compiled into global variables, were later compiled into local variables and passed as call-by-reference parameters to every nested procedure or function. The same goes for the internal variables of a function that contain the function result and the `boolean` to check if the function result was set.

Because procedures and functions in Pascal can have call-by-reference parameters, and neither Java nor the Java bytecode has a way to implement these, variables of primitive types had to be compiled as if they were arrays of one element. This way every variable had a reference that could be passed in case the variable was passed as an actual call-by-reference parameter to a procedure or function. In case of a call-by-value parameter, the value of the expression that is the actual parameter was passed. When this value is received by the method, it is a normal Java variable which has no reference. This meant we had to make a new variable that was an array and initialize it to the value. This variable was used instead of the normal variable.

For convenience reasons constants were also compiled into arrays of one element, even though they did not need a reference. This meant the value of the constant had to be put in the array somewhere. The `<clinit>` method proved to be the perfect place to do this. This is the same place where all the arrays are created and the references to these arrays stored in the appropriate variable.

Library procedures had to be used to handle I/O. The library procedure to handle `write` and `writeln` statements that write to standard output were given. For the `read` and `readln` statements and writing to and reading from files library procedures have to be made in the same manner.

Another problem was the fact that the bytecode has no labeling itself. For this an algorithm had to be designed. This algorithm sets internal labels from which the offsets for the jump instructions are resolved once they are known. Solutions were provided for every conditional statement.

The difference between function returns in Pascal, where the function result has to be assigned to the function identifier, and the Java bytecode that returns the value with a return statement, was solved by using two internal variables. The first variable was used as the variable to assign the function result to. The second variable was a `boolean` variable that was used to check at runtime if there was a function result stored in the first variable. If there was no function result to return from a function, an exception was thrown after which the program halted.

For each recordtype declaration a separate class with the name of the recordtype had to be made. This class, containing the fields of the record, was written to a class file. Every variable of the recordtype was now an instance of this class.

Because variables of primitive types, recordtypes and arraytypes already had references to them, pointers to variables were not a problem, and were almost solved implicitly.

Exceptions can be used to handle `goto` statements that cause a non-local jump. A procedure or function had to throw an exception containing the target label if this label is not in the code of the procedure or function. This exception was caught by the procedure or function that did have the label in its code, and the jump could be made by the exception handler.

This document described how to compile the non-trivial elements of the Pascal language. Compilation of every other element of the language that is not covered here can be solved by using the ideas and methods for compilation of the elements that were described.

12.2 Personal Experience

This section describes the way the project went from beginning to end.

When I started in February 1997, I knew absolutely nothing about the Java Virtual Machine and I had only heard a few things about the Java programming language. Therefore the first thing I did was read [JVM]. This book covers all the details of the JVM, the class files and the JVM instruction set. Although the class files are fully described in the book, I found it hard to visualize what the precise contents of the `constant_pool[]` and other fields of the class file were. To view the contents of a class file, I first dumped all the bytes in hexadecimal form to a file. This way it was possible to see the class file byte by byte. The problem with this was, that if I wanted to find a specific field in this hex-dump, I had to go through all the bytes until I found the field I was looking for. If you made a small mistake in counting the bytes, you were looking at the wrong field.

I started looking for a program that outputs the whole class file in a way that it is possible to quickly find the field you need. I searched the Web for such a program, but I could not find what I needed. I did find the D-Java program written by Shawn Silverman, that shows you the class file as if it was a program written in Java bytecode. It did not show the contents of the `constant_pool[]`, so it was not exactly what I was looking for. Later, when I generated code for Pascal programs, it proved to be a big help after all.

I also found the Jasmin program. Jasmin is an assembler that assembles programs written in Java bytecode to Java class files. D-Java also contains an option to output the class file in 'jasmin' code. When writing a program in jasmin, you do not have to give the entries of the `constant_pool[]` yourself, the assembler constructs them for you. During the project I was really sorry that I did not compile Pascal to jasmin, instead of generating the class file myself. It would have saved me a lot of trouble constructing the `constant_pool[]`. On the other hand, by generating the class file yourself, you can specifically define every field it contains.

Although I found jasmin and D-Java, they were not what I was looking for. I decided to write my own program, *jdisco*. I wrote it to just output all the fields in the class file, it did not have to resolve the indexes to any of the fields. Later I regretted that I did not do that, as I had to do it myself. This is not much work with a small class file, but the bigger the class file, the more work it gets. I could have changed the program, but that would have probably been more work, so I could not be bothered.

It took me about one and a half month to know everything about the JVM I needed to know. In this time I wrote a lot of little testprograms to see how the `javac` compiler compiled some things.

After this period, I started on the compiler. I started out by designing the structures that had to hold most of the fields of the class file in memory during compilation. After doing this, I wrote the procedures that had to fill the class file with its minimum fields, so I could compile an empty Pascal program. This took me another few weeks. Believe me if I say that finding an error in a class file is not really nice work. When checking if a class file is correct, I ran it on the JVM. Most of the time, when there is an error, all the JVM says is "can't find class whateveritsname". That is a big help.

After being able to compile an empty Pascal program, I started working on the variables. It took me quite some time to get these right. At one point it almost looked like trial and error. Just when I thought I had them compiled the right way, I found something that did not work. One of the last days I was working on the compiler and ran some more tests, I found the problem described in subsection 9.1.3. I did not have enough time to implement the solution, so that is why you see the local variables still as global in the example described in appendix A. I did implement the parameters as local variables, but I do not pass them to the nested functions, I just did not have any more time.

Generating code for the statements was relatively easy compared to the problems I had with the variables. In a few weeks I was able to compile a program that could do quite a lot.

At the end of the project, when I was running some tests, I found some little errors (besides the local/global variable problem). Most of them could be solved very simply, and were things I simply forgot or thought I had done. There was one error I did not solve. This was that it is possible in Pascal to compare a value of type `integer` with a value of type `real`, or assign a value of one of these types to a variable of the other type. I did not know that. When you run a program that does this on the JVM, it will work, but if you use the `-verify` flag, to verify the class file before being run, the JVM complains. I did solve the problem with the assignments, by simply casting the rightvalue of the assignment to the type of the leftvalue. I did not solve the comparison problem, because it was not so easy to do because of the way the ACK Pascal compiler was structured, and I did not have time to do the restructuring of the compiler necessary to know when to cast or not.

As floats are stored in the `constant_pool[]` in IEEE 754 floating-point "single format" bit layout, I used the properties of the SPARC processor that also stores floats in this format. This means the compiler can only be used on this platform. To solve this you have to convert the float to this format manually. There must be some C library functions that can do this.

After I was able to compile programs containing `write` and `writeln` statements, I thought it would be a nice idea to implement the `read` and `readln` statements as well. As I was using *JDK1.0.2*, and the only functions available there were functions to read values byte by bytes, I did not think it was possible. With these functions, you could only read an integer from standard input by reading in four bytes. This meant that the user of a program had to divide the integer in 4 bytes before he typed it. My guess was that this was not really the way it should go.

Thanks to some people on the #java IRC channel, I found out that in *JDK1.1.2* there were classes that could read characters from standard input. This only meant that you had to parse the characters typed yourself to construct values of type `integer`, `char` or `real`. I started writing the procedures for it, but soon I found it too much work for something that would definitely work, so I focused my attention to other more important things. This is why you can see the entries for the `read` and `readln` procedures of the library class in the `constant_pool[]` of the example program described in appendix A.

Because I used the Reader and Writer classes of *JDK1.1.2* I also had to use the JVM of this package. The JVM of *JDK1.0.2* could not find a class definition of these classes.

Looking back on the project, I must say I found it pretty interesting and most of the time very annoying when I could not think of a proper solution to a problem fast enough. It really helps to talk to someone else to hear his opinion. Often other people have a clear mind and can help you to view things from a different perspective while you are thinking about it so much that you keep thinking in circles.

12.3 Acknowledgments

I would like to thank David Doolin, who is working on a similar project compiling Fortran to Java source and partly to jasmin. It was always very helpful to exchange ideas.

I also want to thank all the other people on the #java IRC channel on Efnet for helping me out when I was programming in Java. Thanks also to the #koffie IRC channel on IRCnet for helping me and not kicking me off the channel when I was talking about my problems.

Also I would like to thank Ceriel Jacobs for helping me when I was lost in the ACK Pascal compiler, which happened quite a lot...

Finally I want to thank Menno van Zaanen and Ivo van der Wijk for giving me their view on some of the problems and Dick Grune for his supervision.

Appendix A

An Example Program

This appendix contains a simple Pascal program that is compiled into a Java class file. I chose the classic example that prints the string "`Hello, world`", just to show what a class file looks like.

A.1 Pascal Source Code

```
program example(output);

begin
    writeln('Hello, world');
end.
```

A.2 Java Class File

A.2.1 *jdisco* Output

Just to show how a class file looks, here is the output of the *jdisco* program. It shows all the fields in the class file. Each operand of an instruction is shown in hexadecimal form.

```
Reading from file example.class...
magic_number: cafebabe
minor_version: 3
major_version: 45
constant_pool_count: 76
constant_pool[1]: (CONSTANT_Class_info)
                tag: 7
                name_index: 2
constant_pool[2]: (CONSTANT_Utf8_info)
                tag: 1
                length: 16
                string: java/lang/Object
constant_pool[3]: (CONSTANT_Class_info)
                tag: 7
```

```
                name_index: 4
constant_pool[4]: (CONSTANT_Utf8_info)
                tag: 1
                length: 7
                string: example
constant_pool[5]: (CONSTANT_Methodref_info)
                tag: 10
                class_index: 1
                name_and_type_index: 6
constant_pool[6]: (CONSTANT_NameAndType_info)
                tag: 12
                name_index: 7
                descriptor_index: 8
constant_pool[7]: (CONSTANT_Utf8_info)
                tag: 1
                length: 6
                string: <init>
constant_pool[8]: (CONSTANT_Utf8_info)
                tag: 1
                length: 3
                string: ()V
constant_pool[9]: (CONSTANT_Utf8_info)
                tag: 1
                length: 8
                string: <clinit>
constant_pool[10]: (CONSTANT_Utf8_info)
                tag: 1
                length: 4
                string: Code
constant_pool[11]: (CONSTANT_Utf8_info)
                tag: 1
                length: 13
                string: ConstantValue
constant_pool[12]: (CONSTANT_Utf8_info)
                tag: 1
                length: 10
                string: Exceptions
constant_pool[13]: (CONSTANT_Class_info)
                tag: 7
                name_index: 14
constant_pool[14]: (CONSTANT_Utf8_info)
                tag: 1
                length: 9
                string: pcrterror
constant_pool[15]: (CONSTANT_Utf8_info)
                tag: 1
                length: 14
                string: ThrowFunResExc
constant_pool[16]: (CONSTANT_Methodref_info)
                tag: 10
```

```
                class_index: 13
                name_and_type_index: 17
constant_pool[17]: (CONSTANT_NameAndType_info)
                tag: 12
                name_index: 15
                descriptor_index: 8
constant_pool[18]: (CONSTANT_Class_info)
                tag: 7
                name_index: 19
constant_pool[19]: (CONSTANT_Utf8_info)
                tag: 1
                length: 4
                string: pcio
constant_pool[20]: (CONSTANT_Utf8_info)
                tag: 1
                length: 7
                string: pcwrite
constant_pool[21]: (CONSTANT_Utf8_info)
                tag: 1
                length: 9
                string: pcwriteln
constant_pool[22]: (CONSTANT_Utf8_info)
                tag: 1
                length: 4
                string: (F)V
constant_pool[23]: (CONSTANT_Utf8_info)
                tag: 1
                length: 5
                string: (FI)V
constant_pool[24]: (CONSTANT_Utf8_info)
                tag: 1
                length: 6
                string: (FII)V
constant_pool[25]: (CONSTANT_Utf8_info)
                tag: 1
                length: 4
                string: (Z)V
constant_pool[26]: (CONSTANT_Utf8_info)
                tag: 1
                length: 5
                string: (ZI)V
constant_pool[27]: (CONSTANT_Utf8_info)
                tag: 1
                length: 4
                string: (C)V
constant_pool[28]: (CONSTANT_Utf8_info)
                tag: 1
                length: 5
                string: (CI)V
constant_pool[29]: (CONSTANT_Utf8_info)
```

```
                tag: 1
                length: 4
                string: (I)V
constant_pool[30]: (CONSTANT_Utf8_info)
                tag: 1
                length: 5
                string: (II)V
constant_pool[31]: (CONSTANT_Utf8_info)
                tag: 1
                length: 21
                string: (Ljava/lang/String;)V
constant_pool[32]: (CONSTANT_Utf8_info)
                tag: 1
                length: 22
                string: (Ljava/lang/String;I)V
constant_pool[33]: (CONSTANT_Methodref_info)
                tag: 10
                class_index: 18
                name_and_type_index: 34
constant_pool[34]: (CONSTANT_NameAndType_info)
                tag: 12
                name_index: 20
                descriptor_index: 22
constant_pool[35]: (CONSTANT_Methodref_info)
                tag: 10
                class_index: 18
                name_and_type_index: 36
constant_pool[36]: (CONSTANT_NameAndType_info)
                tag: 12
                name_index: 20
                descriptor_index: 23
constant_pool[37]: (CONSTANT_Methodref_info)
                tag: 10
                class_index: 18
                name_and_type_index: 38
constant_pool[38]: (CONSTANT_NameAndType_info)
                tag: 12
                name_index: 20
                descriptor_index: 24
constant_pool[39]: (CONSTANT_Methodref_info)
                tag: 10
                class_index: 18
                name_and_type_index: 40
constant_pool[40]: (CONSTANT_NameAndType_info)
                tag: 12
                name_index: 20
                descriptor_index: 25
constant_pool[41]: (CONSTANT_Methodref_info)
                tag: 10
                class_index: 18
```

```
                name_and_type_index: 42
constant_pool[42]: (CONSTANT_NameAndType_info)
                tag: 12
                name_index: 20
                descriptor_index: 26
constant_pool[43]: (CONSTANT_Methodref_info)
                tag: 10
                class_index: 18
                name_and_type_index: 44
constant_pool[44]: (CONSTANT_NameAndType_info)
                tag: 12
                name_index: 20
                descriptor_index: 27
constant_pool[45]: (CONSTANT_Methodref_info)
                tag: 10
                class_index: 18
                name_and_type_index: 46
constant_pool[46]: (CONSTANT_NameAndType_info)
                tag: 12
                name_index: 20
                descriptor_index: 28
constant_pool[47]: (CONSTANT_Methodref_info)
                tag: 10
                class_index: 18
                name_and_type_index: 48
constant_pool[48]: (CONSTANT_NameAndType_info)
                tag: 12
                name_index: 20
                descriptor_index: 29
constant_pool[49]: (CONSTANT_Methodref_info)
                tag: 10
                class_index: 18
                name_and_type_index: 50
constant_pool[50]: (CONSTANT_NameAndType_info)
                tag: 12
                name_index: 20
                descriptor_index: 30
constant_pool[51]: (CONSTANT_Methodref_info)
                tag: 10
                class_index: 18
                name_and_type_index: 52
constant_pool[52]: (CONSTANT_NameAndType_info)
                tag: 12
                name_index: 20
                descriptor_index: 31
constant_pool[53]: (CONSTANT_Methodref_info)
                tag: 10
                class_index: 18
                name_and_type_index: 54
constant_pool[54]: (CONSTANT_NameAndType_info)
```

```
                tag: 12
                name_index: 20
                descriptor_index: 32
constant_pool[55]: (CONSTANT_Methodref_info)
                tag: 10
                class_index: 18
                name_and_type_index: 56
constant_pool[56]: (CONSTANT_NameAndType_info)
                tag: 12
                name_index: 21
                descriptor_index: 8
constant_pool[57]: (CONSTANT_Utf8_info)
                tag: 1
                length: 9
                string: pcread_fl
constant_pool[58]: (CONSTANT_Utf8_info)
                tag: 1
                length: 9
                string: pcread_ch
constant_pool[59]: (CONSTANT_Utf8_info)
                tag: 1
                length: 10
                string: pcread_int
constant_pool[60]: (CONSTANT_Utf8_info)
                tag: 1
                length: 8
                string: pcreadln
constant_pool[61]: (CONSTANT_Utf8_info)
                tag: 1
                length: 3
                string: ()F
constant_pool[62]: (CONSTANT_Utf8_info)
                tag: 1
                length: 3
                string: ()I
constant_pool[63]: (CONSTANT_Utf8_info)
                tag: 1
                length: 3
                string: ()I
constant_pool[64]: (CONSTANT_Methodref_info)
                tag: 10
                class_index: 18
                name_and_type_index: 65
constant_pool[65]: (CONSTANT_NameAndType_info)
                tag: 12
                name_index: 57
                descriptor_index: 61
constant_pool[66]: (CONSTANT_Methodref_info)
                tag: 10
                class_index: 18
```

```
                name_and_type_index: 67
constant_pool[67]: (CONSTANT_NameAndType_info)
                tag: 12
                name_index: 58
                descriptor_index: 62
constant_pool[68]: (CONSTANT_Methodref_info)
                tag: 10
                class_index: 18
                name_and_type_index: 69
constant_pool[69]: (CONSTANT_NameAndType_info)
                tag: 12
                name_index: 59
                descriptor_index: 63
constant_pool[70]: (CONSTANT_Methodref_info)
                tag: 10
                class_index: 18
                name_and_type_index: 71
constant_pool[71]: (CONSTANT_NameAndType_info)
                tag: 12
                name_index: 60
                descriptor_index: 8
constant_pool[72]: (CONSTANT_Utf8_info)
                tag: 1
                length: 4
                string: main
constant_pool[73]: (CONSTANT_Utf8_info)
                tag: 1
                length: 22
                string: ([Ljava/lang/String;)V
constant_pool[74]: (CONSTANT_String_info)
                tag: 8
                string_index: 75
constant_pool[75]: (CONSTANT_Utf8_info)
                tag: 1
                length: 12
                string: Hello, world
acces_flags: <no flags>
this_class: 3
super_class: 1
interfaces_count: 0
interfaces[]: <none>
fields_count: 0
fields[]: <none>
methods_count: 2
methods[0]:
        acces_flags: <no flags>
        name_index: 7
        descriptor_index: 8
        attributes_count: 1
        attributes[0]:
```

```
                attribute_name_index: 10 (Code)
                attribute_length: 17
                max_stack: 1
                max_locals: 1
                code_length: 5
                code[]:
                        0       aload_0
                        1       invokespecial  00 05
                        4       return
                exception_table_length: 0
                exception_table[]: <none>
                attributes_count: 0
                attributes[]: <none>
methods[1]:
        acces_flags: ACC_PUBLIC ACC_STATIC
        name_index: 72
        descriptor_index: 73
        attributes_count: 1
        attributes[0]:
                attribute_name_index: 10 (Code)
                attribute_length: 21
                max_stack: 2
                max_locals: 1
                code_length: 9
                code[]:
                        0       ldc  4a
                        2       invokestatic  00 33
                        5       invokestatic  00 37
                        8       return
                exception_table_length: 0
                exception_table[]: <none>
                attributes_count: 0
                attributes[]: <none>
attributes_count: 0
attributes[]: <none>
Done, closing example.class...
```

A.2.2 D-Java Output

The D-Java program, written by Shawn Silverman, gives a more readable output of the class file.

```
------- D-JAVA (Tue 12-11-1996) --------
  written by: Shawn Silverman
      e-mail: umsilve1@cc.umanitoba.ca
----------------------------------------


Major version: 45
Minor version: 3
```

```
class example extends Object
{

Method void <init>()
>> max_stack=1, max_locals=1 <<
    0 aload_0
    1 invokenonvirtual #5 <Method Object.<init>():void>
    4 return

Method public static void main(String[])
>> max_stack=2, max_locals=1 <<
    0 ldc #74 <String "Hello, world">
    2 invokestatic #51 <Method pcio.pcwrite(String):void>
    5 invokestatic #55 <Method pcio.pcwriteln():void>
    8 return
}
```

Bibliography

[JVM] Tim Lindholm and Frank Yellin, *The Java Virtual Machine Specification.* The Java Series, Addison-Wesley, 1997

[JPL] Ken Arnold and James Gosling, *The Java Programming Language.* The Java Series, Addison-Wesley, 1997

[JLS] James Gosling, Bill Joy and Guy Steele, *The Java Language Specification.* The Java Series, Addison-Wesley, 1997

[APC] Aad Guedeke and Frans Hofmeester, *The ACK Pascal Compiler.* Vrije Universiteit, Amsterdam

[PC] Katheleen Jensen and Niklaus Wirth, *Pascal User Manual and Report.* Springer-Verlag, 1978

[LL] Ceriel J. H. Jacobs, *LLgen, an extended LL(1) parser generator.* Vrije Universiteit, Amsterdam

[EM] A. S. Tanenbaum, H. van Staveren, E. G. Keizer and J.W. Stevenson, *Description of a Machine Architecture for use with Block Structured Languages.* Informatica Rapport IR-81, Vrije Universiteit, Amsterdam, 1983