

THE BUCHAREST UNIVERSITY OF ECONOMIC STUDIES
THE FACULTY OF CYBERNETICS, STATISTICS AND ECONOMIC INFORMATICS
ECONOMIC INFORMATICS SPECIALISATION



Bachelor's degree thesis

Professor:
Cristian Valeriu **Toma** PhD.

Graduate:
Bogdan Andrei **Stoica**

Bucharest
2024

THE BUCHAREST UNIVERSITY OF ECONOMIC STUDIES
THE FACULTY OF CYBERNETICS, STATISTICS AND ECONOMIC INFORMATICS
ECONOMIC INFORMATICS SPECIALISATION



Compiler solution for JVM based language

Bachelor's degree thesis

Professor:
Cristian Valeriu **Toma** PhD.

Graduate:
Bogdan Andrei **Stoica**

Bucharest
2024

Authenticity declaration

The undersigned, Bogdan Andrei Stoica, graduate of the undergraduate studies, year 2024, of the Economic Informatics specialisation of the Bucharest University of Economic Studies, regarding the elaboration of the bachelor's degree thesis, I declare that:

1. The work was developed personally and belongs to me in its entirety.
2. No other sources than those mentioned in the Bibliography were used.
3. No texts, data or graphic elements have been taken from other works or from other sources without being quoted and without specifying the source of the processing, including if the source represents my other works.
4. The paper has not been used in other exam or competition contexts.

Table of contents:

Chapter 1 - Introduction	5
1.1 Topic Presentation	5
1.2 Purpose of the thesis	6
Chapter 2 - Technologies	8
2.1. Compilers history and design	8
2.2 Java	9
2.3 JVM and Java Bytecode	12
2.4 ObjectWeb ASM	16
2.5 Networking with UDP	16
Chapter 3 - Compiler Solution	20
3.1 Presentation of the application	20
3.2 General structure and design	21
3.3 Compiler implementation	24
3.4 Methods implementation	27
3.5 Generated bytecode	28
Chapter 4 - Graphical User Interface	33
4.1 General Design	33
4.2 File management and configurations	34
4.3 Compilation and execution	36
Chapter 5 - Conclusions	39
Bibliography	41

Chapter 1

Introduction

The main objective of this bachelor thesis and project is for me to learn more about compilers, how they work and what are some approaches to build one, what are the advantages and disadvantages in relation to other intermediary code generator applications, such as interpreters, and to learn more about Java, Bytecode and the Java Virtual Machine. The title “Compiler solution for JVM based language” suggests an attempt to create a better and more efficient, (or shorter) way to write code in order to perform a very specific task. The aim is to create a language with shorter syntax than what Java Standard Edition offers currently.

1.1 Topic Presentation

In Java SE programming, starting and implementing a server-client connection over a network protocol is a time consuming task, since it requires many lines of code and in-depth knowledge about each protocol in order to establish the communication. So in order to offer a solution to this problem, I propose a very short syntax that can start a server or establish a client connection to the server in a very compact form. The way the protocol works will be handled internally and the user can simply send messages easily in a short period of time.

The code to be compiled by my application into bytecode will be called ByteNet and will be written in text files with the extension “dot bynt” (.bynt). The main purpose of this language is connecting to a network or starting a UDP server and client, sending messages over the network, doing computations and running small loops using the Java Virtual Machine and .class files generated by my compiler application.

Learning about the implementation of a compiler offers many useful skills as a software developer in handling multiple tasks or better understanding how the programming language behaves behind the scenes. For a programming language to work properly, the user is required to write the specific syntax correctly for the compiler to understand and avoid

ambiguity. The parser is a crucial part of the compiler that splits the source code into tokens that are used as logical components in the program, text manipulation is undeniably a very important skill for any software engineer in any field of activity, quite often the data is offered in a String format and a parser is needed in order to extract the logic the program.

1.2 Purpose of the thesis

The reason I chose this topic is because learning and implementing a compiler offers a wide range of knowledge about various aspects of programming in JVM based languages and not only. This project will help me gain a better understanding of how network client-server relationships work at a low level and what are the procedures of communication between each machine based on the protocol used. Other valuable knowledge gained include simple arithmetic operations in bytecode instructions, memory allocation for variables and handling the information stored inside them, repetitive structures, loops and frames management at the JVM Bytecode level.

Compilers are at the base of the whole digital and informatic world and one of the most valuable tools for any programmer. Without a compiler, every piece of high level programming code is redundant, since, without a software to translate the source code into machine language, the code can not be executed or used in any practical way by the computer. The compiler has meant a huge step in productivity, because now software developers can use very specific high level languages for their domain of activity that help them perform tasks quickly and efficiently. Every language offers various advantages and disadvantages, since these languages are designed with the purpose of solving a particular problem. Some of them aim to shorten the syntax for some repetitive lines of code to improve time in the development process for a specific case (such as Kotlin for Android applications programming), other languages are designed to handle data in all sorts of different formats and manipulate or compute various indicators (such as R, SAS, Python), some other languages aim to make the written code very flexible and machine independent (such as Java, C#) and some are oriented towards performance at runtime (such as C, C++).

Knowing how the Java Virtual Machine behaves and how the language is translated in bytecode form unlocks new perspectives and offers the programmer the possibility to

manipulate the intermediate code in order to extract the best result in performing certain tasks or run sequences of operations that Java or other high level languages based on the JVM don't in their form and syntax.

Many applications nowadays are JVM based since it is still one of the fastest machine architecture independent frameworks and it is very flexible with multiple paradigms. There are various compilers that use the Java Virtual Machine as target of translation for its capabilities and flexibility, a proof of this is the diversity of paradigms some of these languages have: Java: Object-oriented, Scala: Functional and Object-oriented, Groovy: Scripting language etc.

Chapter 2

Technologies

2.1. Compilers history and design

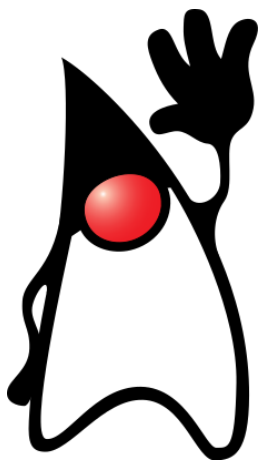
Compilers are software applications that aim to translate a high level language, that are especially designed for better productivity in development for the programmers, into machine or intermediate language code that can be understood by the system. This translation has the objective to process the code into an executable program that can be easily opened and used by the user for the purpose the application was designed for.

In present times, compilers are of many types, some aim to translate the source into low level code that is specific to a certain processor architecture or machine specifications, an example of such a compiler is the C compiler that directly generates assembly instructions for the machine the program is run on. Some other compilers make intermediary code that can be understood and run by specific software or environments, such as the Java Compiler, which transforms the source code into an intermediate language that can be executed only on a virtual architecture, that is called the Java Virtual Machine or the JVM. This intermediate language is called the Bytecode and it is composed of a set of sequential instructions that is in accordance with the specifications of the JVM. Each instruction has an opcode associated with it that is represented as a full byte or 8 bits as hexadecimal digits.[1]

The basic principles of modern digital computing were established by scientists, mathematicians, and engineers during World War II. Simple binary languages developed because of the fact that digital devices could only understand ones and zeros as well as the circuit designs within the machine's architecture. Assembly languages were developed in the late 1940s to provide a more practical representation of computer architectures. The early compilers faced significant technical challenges due to the limited memory capacity of early computers. As a result, the compilation process had to be split into multiple smaller programs. The analysis products created by the front end programs are utilised by the back end programs for the generation of target code.[2] With more resources from advances in computer technology, compiler designs could better match the compilation process.

Most of the compilers perform a series of phases on the source code in order to make the translation possible into the target code. These phases are the: preprocessing, in which the source code is striped of comments and other redundant pieces of code in order to ensure that only the useful information is processed, lexical analysis, in which the code is divided in smaller pieces with logical value for the program called tokens, parsing, in which the grammar of the language is verified and it is ensured that the sequence of tokens is corresponding to the language specifications and syntax, semantic analysis, which checks and gathers context information about each operation performed and verifies the types of the declared variables, intermediate representation, is the phase that structures the logical parts of the source code, code optimization, rewrites parts of the code in the most efficient way based on the compilers capabilities, and then machine specific code generation or intermediate code translation, based on the type of the compiler, in which each operation is translated one by one to the target language into an executable file that can be used by the target virtual or physical machine.[3]

2.2 Java



Java is an object-based and object-oriented programming language created by Sun Microsystems in 1995, which was later bought by Oracle in January 2010, and has shaped the world of programming ever since. It emerged as a language that aimed to change the world of software development with its unique, at that time, vision “write one, run anywhere”. The developer meant that they intended to make the language platform-independent as long as the machine was equipped with a Java Runtime Environment (JRE), a piece of software with a virtual machine that would translate the high level language, Java, into a intermediate language, named Java Bytecode, for the Java Virtual Machine (JVM) that could be executed from anywhere.

Java was also one of the first and most important languages that contributed to the popularity of the object-oriented programming paradigm. Even though other languages, such as C++, already existed at that time in the same paradigm and had their fair share of popularity, they

were very tightly tethered to the specific operating system and hardware architecture the code was compiled on. By introducing bytecode, created from human-readable Java source code, Java overcame this restriction.[4] The JRE, a virtual machine available on different platforms, was then able to interpret this bytecode. This breakthrough eliminated the limitations of traditional languages, enabling the creation of cross-platform applications that could run seamlessly on Windows, macOS, and Linux systems. Java's liberation from the confines of specific operating systems opened doors for a new phase of platform-independent software development.

Java's strong points are rooted in its dedication to object-oriented programming (OOP) principles. In Java, every element is treated as an object, containing both data and the operations that act on it. This approach to object orientation encourages the reuse of code, modularization, and ease of maintenance. As software projects become more elaborate, Java's object-oriented framework enables developers to break down entire solutions into manageable components, resulting in well-organised and noticeably easier-to-manage code sequences. The structure of Java views its classes as a hierarchy of classes on different levels, similarly to a tree, and at the root level there is the Object class, thus every other class instance can be upcasted to an instance of Object.[5]

In addition to the features mentioned above, Java offers various advantages for developers when it comes to the language characteristics or framework to use for their project, some of these are:

Robust Type System: Declared data type of variables ensures type safety and reduces runtime errors. Type safety emphasis safeguards against unexpected behaviour and contributes to the overall stability of Java applications.

Automatic Memory Management: Java's virtual machine uses a garbage collector to automatically liberate unused memory, making the memory management for developers much easier. This automatic handling eliminates the need for manual memory allocation and deallocation, a common source of errors in other languages, such as C and C++.

Comprehensive Standard Library: Java gives a comprehensive set of pre-written classes and functionalities, saving programmer time and effort. This broad standard library offers a base for common programming assignments, file management I/O, organising, information structures, and collections.

Exception Handling: Java offers a strong system for handling exceptions in order to effectively handle errors and avoid program crashes. Exceptions enable developers to foresee and manage potential errors in their code, guaranteeing the program's consistent operation even when unexpected events occur.

Security: Security is a top priority in Java, utilising features such as sandboxing to limit code execution and block unauthorised access. This emphasis on security is especially important for apps that manage sensitive information or work in unsecure surroundings.

Although Java has its strengths, there are certain factors to take into account when choosing a programming language for particular projects. Java code can occasionally be quite verbose than other languages, needing more lines to accomplish the same task. Programmers used to shorter languages may need some practice to adapt to Java's syntax. In terms of performance, typically efficient, Java may be slightly slower than languages, such as C++, for extremely performance-sensitive applications. Java's emphasis on cross-platform compatibility and automatic memory handling may result in additional processing compared to languages with more direct hardware control. Yet, in the majority of development cases, Java's speed is sufficient, and its additional benefits surpass this downside. Just-In-Time (JIT) compilation further improves Java's performance by translating bytecode into machine code at runtime.[6] Another disadvantage would be that applications require the JRE for operation, potentially leading to an extra installation step for end-users. While this can be a minor problem, the widespread adoption of Java has mitigated this somewhat, as a large number of users already have the JRE installed on their computers. Moreover, the increasing popularity of cloud computing and browser-based apps is decreasing the need for local JRE installations.

JavaFX is a powerful, versatile framework for building rich, visually appealing graphical user interfaces (GUIs) in Java. JavaFX library offers a modern and flexible approach to UI development. It provides a wide range of built-in UI controls, such as buttons, tables, and charts, along with advanced features like CSS styling, hardware-accelerated graphics, and media playback support. JavaFX's Scene Graph architecture allows developers to create complex, interactive user interfaces with ease, leveraging a hierarchical tree of nodes. Additionally, JavaFX supports FXML, an XML-based language for defining UI components, which enhances the separation of design and logic, facilitating collaboration between designers and developers. With its robust capabilities, JavaFX is well-suited for creating cross-platform desktop applications that demand rich and dynamic user experiences.

2.3 JVM and Java Bytecode

Java Virtual Machine

The Java Virtual Machine or the JVM is a virtual machine developed by Sun Microsystems in the year 1994 with the main purpose at release of creating a virtual architecture on which all the compiled Java programs could run regardless of the architecture of the physical layer. The JVM in order to run correctly needed a compiled file of the source language code stored in a .class file that corresponds to the specifications of the virtual machine. The .class files are files in which the compiler generates an intermediate code named Bytecode or the Java Bytecode in a specific format. This code is not human readable and other software or programs are needed in order to visualise it, one of the most used tools are the JetBrains IntelliJ integrated decompiler which translates the Bytecode back to Java or the JavaP tool that comes by default with the Java Development Kit installation and help the user visualise the bytecode instructions sequentially. The virtual machine uses an interpreter that reads the bytecode instructions line by line and translates them for the physical machine's architecture in order for the program to be executed.

After the source code has been compiled into the .class files, the generated bytecode goes through various steps until the execution is completed. The first step is the class loader, which handles the loading, linking and the initialization of variables. During this phase the .class files are read and generate the binary data corresponding to the instructions of the files and are stored in the method area of the JVM. For every class the name of the class and the name of each method and attribute is stored in the memory, the linking checks that the files are correctly structured and allocates memory for all the static variables if any exists. The initialisation phase assigns the values to each static variable that has been defined in a top to bottom fashion inside a class and from a parent to child hierarchy between classes.[7] (as well as every link between files or classes, in order to keep track of what has been defined by the user in the source code.)

JVM Memory is the second big component of the virtual machine which is also divided into smaller parts. There are five of these parts and the first one is the Method Area

where all the information about the methods names, class names, parent classes and static variables is stored. This memory area is shared with all the execution paths in JVM and there is only one. The second part is the Heap Area which is responsible for storing all the object data into the memory and similar to the Method Area, it is also a shared resource in the Java Virtual Machine. Stack Area is not a shared memory area and it is responsible for storing the information about each method call and all the variables corresponding to it in each frame. This part of the memory is independent and for every execution thread of the JVM, a new run-time stack is allocated for each one of them, and when the thread terminates the virtual machine takes care to destroy the corresponding run-time stack. PC Registers manage the addresses of each current execution instruction for every single execution thread and The Native Method Stack manages the native method calls also independently for every thread. These are not shared parts of the memory.[8]

The last component of the Java Virtual Machine is the Execution Engine which is responsible for running the program and has three main components: Interpreter, Just-in-Time Compiler and the Garbage Collector. The interpreter takes each bytecode instruction line by line from the class file and executes it. The Just-in-Time Compiler or the JIT, compiles the entire file and provides the native code for it, it has the purpose of increasing the efficiency of the interpreter since it is relatively slow. When the interpreter identifies a piece of code that has been already interpreted, the JIT instantly provides the native code corresponding to that specific method call in order to avoid the re-interpretation of the same method, thus saving time. And lastly, the Garbage Collector is a complex part of the Java Virtual Machine that has the purpose of identifying un-referenced objects and then destroys them from the memory to efficiently manage the resources of the heap area.[9]

Java Bytecode

The set of instructions that the Java Virtual Machine uses in the interpretation process is called Java bytecode and it is the result of compilation of every JVM based high level programming language. The bytecode instructions' opcodes are represented on a single byte and are often represented as a combination of two hexadecimal digits. This machine language is specific for the virtual machine specification and can be executed consistently with the same results on any computer that has the JVM installed. The instruction set is interpreted

and compiled dynamically by the Just-In-Time compiler, for increased performance, in order to generate the machine code for the device the code is run on.

The Java bytecode is stored in a specific format in order for the JVM to be able to execute them, the same code is cross-platform compatible with various architectures and has a high level of security. Inside the virtual machine, the bytecode is used simultaneously by a stack machine and a register machine. The stack has the role of storing the operands information and the local variables are used for executing the operations. The local variables have a similar purpose as registers inside a processor, and are also used for passing the method arguments. All instructions are in conformity with Java's object-oriented model.

Java bytecode is divided in the following instructions categories:

- Load and store
- Arithmetic and logic operations
- Type conversion instructions
- Object creation and manipulation
- Operand stack management
- Control transfer instructions
- Method invocation and return

Part of each method, the maximum dimension of the operand stack and local variables is computed by the compiler. The values these can take are sized independently from 0 up to 65535 values, where each value has a size of 32 bits, except the long and double types that occupy two consecutive local variables for a total of 64 bits. Most of the operations have a prefix or a suffix that indicates the type of the operands used in the execution. The operand types defined by the JVM are the following:

Prefix/suffix	Operand type
i	integer
l	long
s	short
b	byte
c	character
f	float
d	double
a	reference

Fig. 1: Operand Prefix/Suffix table

https://en.wikipedia.org/wiki/Java_bytecode

The Java class file, (extension of the file: .class), is the file format in which the Java bytecode instructions are stored. This file is generated by the compiler from a source file (.java or any other JVM specification compiled languages) and it is used by the Java virtual machine for execution of the program. The format of the class file is composed of ten basic sections:

- Magic Number: 0xCAFEBADE
- Version of Class File Format: the minor and major versions of the class file
- Constant Pool: Pool of constants for the class
- Access Flags: for example whether the class is abstract, static, etc.
- This Class: The name of the current class
- Super Class: The name of the superclass
- Interfaces: Any interfaces in the class
- Fields: Any fields in the class
- Methods: Any methods in the class
- Attributes: Any attributes of the class (for example the name of the sourcefile, etc.)

The class file is parsed sequentially starting from the first byte towards the end due to its variable-sized items and lack of embedded file offsets or pointers. Certain basic types are later re-interpreted as higher-level values (like strings or floating-point numbers) based on the situation. Word alignment enforcement is nonexistent, therefore padding bytes are never utilised. General layout of the bytecode file looks in the following manner: 4 bytes - magic number, 2 bytes - minor version of the file, 2 bytes - major version of the file, java version, 2 bytes - constant pool, the constant pool table and then the file continues with access flags, interfaces, field count and table, methods count and table and attribute count and table.[10] Usually all size-indicating sections take up two bytes of memory in the file.

2.4 ObjectWeb ASM

ASM or ObjectWeb ASM library is a OW2 consortium project that aims to help the user decompose, modify or generate Java Bytecode in binary format from bytecode instructions defined as methods. This library was created and initially released in 2002 by Eric Bruneton[11].

The components have been designed to facilitate the user in keeping the class file format properly structured through several visitors depending on the context of execution or the type of instructions used. The main two classes used in making this project were ClassVisitor that helped structuring the class and MethodVisitor that offered functions for writing instructions inside the main method of the class.

2.5 Networking with UDP

The User Datagram Protocol (UDP) is a network protocol that offers efficient and fast communication over the network. UDP offers a distinct approach to network communication within Java applications. UDP, in general, stands out for its simplicity, speed, and efficiency. It is a core protocol of the Internet Protocol (IP) suite, offering a connectionless datagram service that is often compared to its more reliable counterpart, the Transmission Control Protocol (TCP).

UDP is a transport layer protocol that functions at the same level as TCP, but it uses a distinct approach to transmitting data. Unlike TCP, which sets up a connection before sending data and guarantees packet delivery, UDP does not establish a connection and does not ensure delivery, order, or error verification. The absence of extra processing makes UDP well-suited for situations where quickness and effectiveness are crucial, even if occasional data loss is acceptable.

User Datagram Protocol has quite some distinct characteristics.

UDP is a connectionless protocol that does not require a connection to be established before data is sent, therefore avoiding the necessity of handshaking procedures. Every individual datagram is sent with all the required routing information to reach the destination.

No mechanisms exist for retransmitting lost packets, correcting errors, or acknowledging receipt in unreliable transmission. This lack of reliability is exchanged for quicker transmission of data.

Low Overhead: The UDP header is simply 8 bytes in length, much smaller compared to the 20-byte TCP header. This small amount of extra processing enables quicker data packet processing and transmission.

UDP provides support for broadcasting and multicasting, which allows for sending packets to all devices on a network or a specific group of devices, respectively, benefiting certain applications.

These characteristics give UDP a handful of advantages compared to other protocols for data transmission. With a very small overhead and direct transmission without checks, make this protocol very fast, a great choice in speed-sensitive applications. Also it is very efficient in terms of memory since only the data and minimal information regarding the destination are sent over the network, making it very lightweight, and its scalability is great, since it supports and can handle a great number of clients simultaneously. With these advantages come some drawbacks to it in some certain scenarios, such as the fact that it is not a reliable transport protocol, there is no mechanism for error detection or for successful delivery. For some very sensitive applications in terms of the quality of data transmission, UDP is not a great choice, in those cases a tradeoff between speed and reliability should be taken into consideration. Also, because the absence of connection setup makes UDP more vulnerable to specific network attacks, like IP spoofing and amplification attacks, making it not the best choice if security is a big concern.

Java provides robust capabilities for handling UDP connections through its `java.net` package, specifically with the `DatagramSocket` and `DatagramPacket` classes. Using these classes, Java developers can create UDP sockets for sending and receiving datagram packets across the network. `DatagramSocket` facilitates the creation of both client and server applications, while `DatagramPacket` encapsulates the data to be transmitted or received. Java's inherent platform independence and extensive standard libraries make it a suitable choice for developing network applications that require the speed and efficiency of UDP, such as real-time gaming, multimedia streaming, and network discovery services. With its straightforward API, Java simplifies the process of implementing UDP communication, ensuring quick development and deployment of high-performance network applications[12].

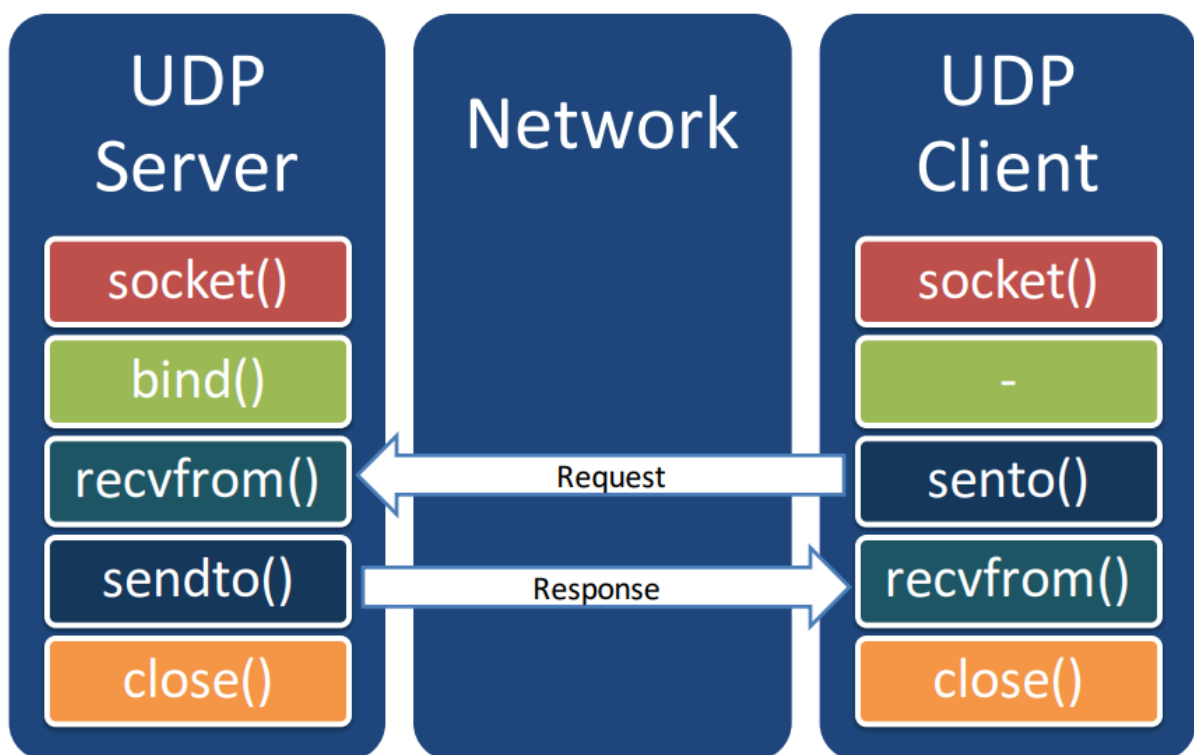


Fig. 2: UDP communication schema[12]

https://github.com/critoma/javase/blob/master/presentations/c10_JSE_TCP_UDP_Dev.pdf

The figure from above illustrates how the communication is made between two instances using the User Datagram Protocol. Firstly, after both socket instances have been

created, the server has to be binded to a port at its address, this way the UDP Server instance receives an identity over the network where it can receive the packets of data. UDP Client can send data encapsulated in a datagram packet with the destination address and port of the server, while the server is already waiting and expecting to receive the information. The server should have the means to transform the input stream of the packet back into the information sent by the client using the protocol specific operations of de-encapsulation. This communication can be made both ways. When each side of the transmission does not want to continue receiving or sending data, the instances can be closed.

Chapter 3

Compiler Solution Application

3.1 Presentation of the application

In this chapter I will focus on the design and implementation of the ByteNet compiler, a compiler for a programming language designed to offer simplified syntax for creating UDP clients and servers for message transmission in string format, simple operations, loops and printing means, and how each technology presented in the previous sections were used in the development of this solution. The application works as an intermediary translator between an overly simplified grammar that helps the user compile ByteNet code into Java class files in Java bytecode, corresponding in structure and grammar to the specifications of the Java Virtual Machine. The resulting class can be used by the JRE and can be simply called via the command line with a simple Java command in order to run the program like any other Java application.

The name of the language comes from the “bytecode” that stands as the intermediary language between the compiler and the JVM interpreter and the main solution it offers: simplified network communication of data in string format, thus Byte + Net. Source code for this programming language is stored in “.bynt” file format, used by the compiler for reading ByteNet source code as input for translation.

The main idea of the compiler was the facilitation of network connection of UDP server - client type relation, and providing simple methods to exchange messages in string format between instances for the JVM. The user only has to specify the port to which he desires to communicate and optionally the address, which is the “localhost” or the “127.0.0.1” address by default if no other address is specified in the declaration of the socket variable. Each instance of the socket is stored into a variable that can be named how the user desires, methods, such as send or receive, can be called on it later in the source code. Sending and receiving messages is done through DatagramPackets internally, hence the user only needs to interact with the DatagramSocket through an instance of ByteNet Socket. When the message reaches the destination it arrives as a byte array, for the receiver to be able to see it,

the “print” method can be called before receiving which takes care of all the steps in order to translate the message from a byte array to an easily readable string that has been formatted and striped of unnecessary buffer spaces before or after the information. The compiler aims to simplify not only the description of an UDP server or client, but also format the data inside the packets discreetly just by calling a method.

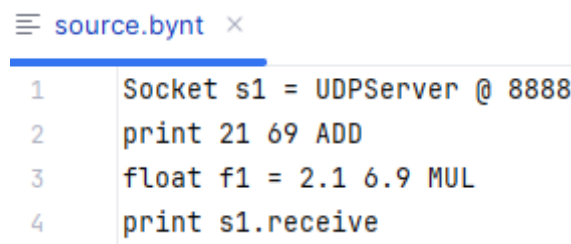
The compiler is designed to also save and handle simple operations between integer numbers, floating point numbers and strings. The results of these operations can be stored into user defined variables or printed to the console as the user pleases. A loop structure is present as well, that behaves similarly to an infinite `while(true)` from Java, to give the programmer more control over the UDP connection, thus maintaining the connection to the port open even after the first packet has been received. Also the client that is sending the messages can be kept open with the help of the loop to send multiple messages to the same server address. Conditional structure “when - exit” can be used inside the loops to dictate the compiler a simple comparison condition to break out of the repetitive structure. Even though it is Java based, this language doesn’t have the characteristics of an object-oriented language since all the class instances from Java, such as the `Socket`, are treated as primitive types in ByteNet.

3.2 General structure and design

ByteNet is a simple programming language that offers efficient solutions to opening and maintaining User Datagram Protocol communications between different instances over the network. The key feature of the language is the syntax, which is very simple and a lot shorter than what other Java Virtual Machine based languages offer for handling such operations. In only two lines of code a string message of any dimension can be sent to the desired destination over the network.

The syntax is somewhat closer to Java and C in terms of the meaning the symbols carry, for example: the equal sign (`=`) is an assigning operator, dot (`.`) is used for accessing the methods of a variable, double equals sign (`==`) is the logic equals comparison operator. Also, the declaration of variables is similar to Java and C following the grammar [Type of the variable + identifier name + equals operator + the value assigned to it]. All variables must be

assigned a value at declaration. Arithmetic operations follow a syntax that is much closer to some scripting languages: [First operand + second operand + operator]. The operator is written as a short 3 letter word that suggests the abbreviation name of the operation that is intended to be made between the operands. A crucial difference to previous programming languages used for comparison is that every statement is written on a separate line, no semicolon is used in identifying the end of the statement, the end of the line (“\n”) is used for that matter.



```
1 Socket s1 = UDPServer @ 8888
2 print 21 69 ADD
3 float f1 = 2.1 6.9 MUL
4 print s1.receive
```

Fig. 3: ByteNet code sample

Above code snippet interpretation: A socket variable is assigned to the UDP server at port 8888 at the default inet address “localhost”, internally a DatagramSocket is instantiated on that port number. On the following line a simple print of the result of the addition of the two numbers is displayed to the console, this is a print for integer values. Format of number operations has the shape [operand1 + operand2 + operator]. Then a variable of type float, named “f1”, is assigned to the result of the multiplication of the two floating point numbers. Finally, on the last line of code, the previously declared UDP server socket calls the receive method, which inside the compiler declares a Datagram Packet that is ready to receive the first message that is going to be sent to the localhost address on port 8888. The printing call made at the beginning of the line processes the received byte array from the packet and formats it into a string and trims all the unnecessary spaces sent with it, thus giving a very comfortable and readable way of visualising the received message in string format to the receiver.

When the ByteNet compiler is called by the user, the application starts from the “Main.class” file, which through a for each loop for every path argument provided to the

compiler, the Parser’s method “parse” is called and the processing of each file begins independently and sequentially. After the source file has been processed and the syntax has been verified and validated, the bytecode generator is called which translates the ByteNet language into Java Bytecode, with the help of the Object ASM framework for bytecode manipulation and analysis, into byte arrays. And finally, each byte array corresponding to each ByteNet file is written directly in binary format into “.class” files that share the name of the source file. This operation is done by the Writer Class and the resulting class files are generated in a special output directory within the compiler files named by default “CompilerOutput”.

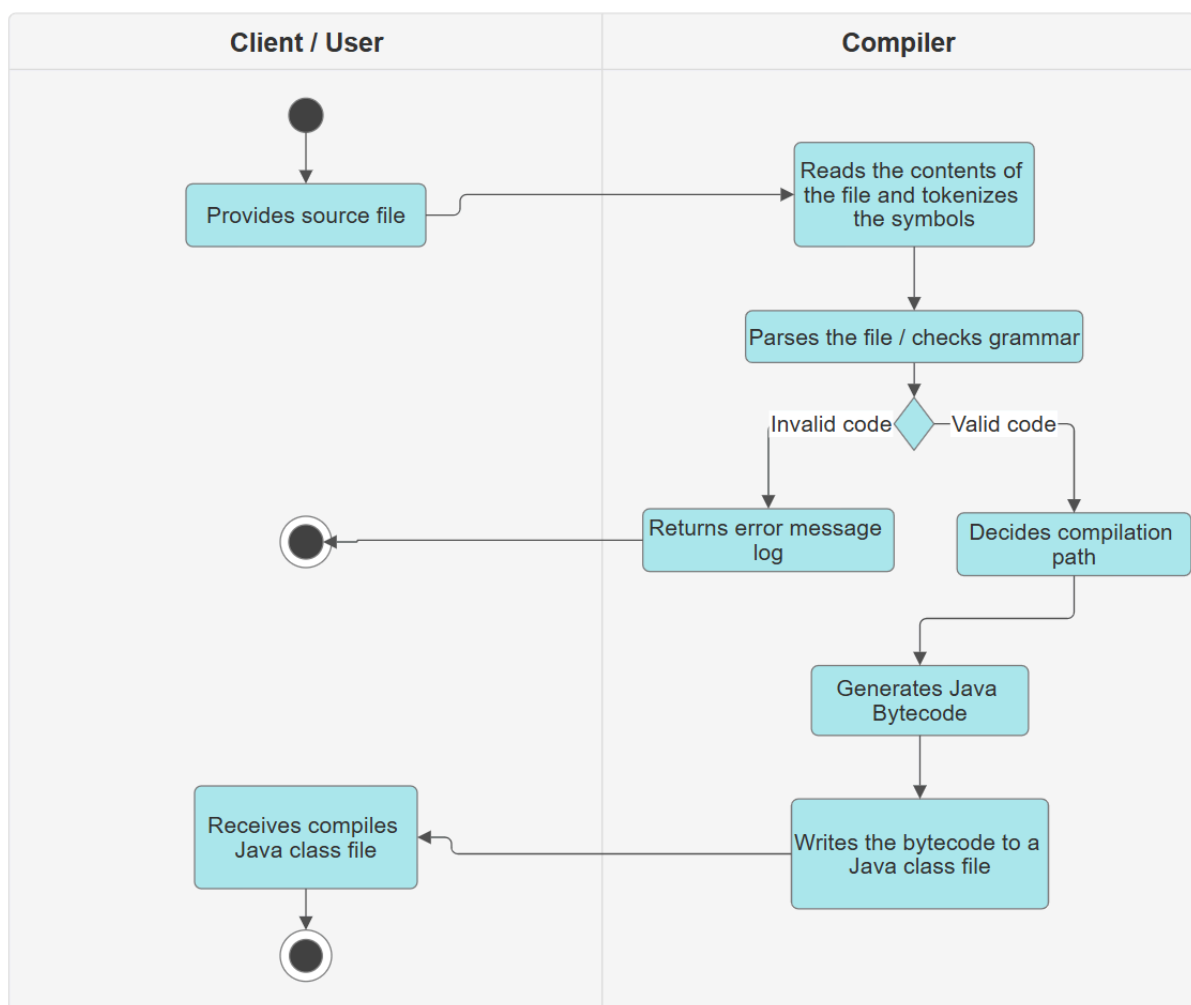


Fig. 4: UML diagram for data flow in the compiler

3.3 Compiler implementation

The Parser is a complex class that manages the entire flow of information within the compiler processes in a centralised manner. Inside it there are declared 7 static variables for managing its tasks:

- “index” - integer: keeps tracks of the current token that is being manipulated inside a line
- “line” - string array: stores all the tokens of a single line of source code at a time
- “identifierIndex” - integer: keeps track of the current internal index of a variable
- “identifiers” - hash map(string, string array): keeps track of all identifiers names, type and their internal index
- “sockets” - hash map(string, string array): keeps track of all socket names, their port and address
- “labelStack” - stack(asm label): helps in managing labels of the current loop structure
- “variablesTypes” - array list(strings): manages the type of the variables declared before initialising a frame

The execution flow of the parser is concentrated in the only public method of the class which calls the other methods in the intended order. Method `parse()` receives as a parameter the path of the file as a string and then calls the `SourceReader.readSource()` and stores the entire code inside a `Scanner` instance. The name of the class is extracted from the path and the class creation bytecode is generated for it, then inside a while loop that checks the lines of the scanner the index of the line is set to the default value zero, the next line is read from the scanner into the line array and the statements are checked sequentially. After all lines have been processed the `WriterClass.writeClass()` is invoked and receives the name of the class and the bytecode in a byte array format and writes all the intermediary code to a file of form `[class name + .class]` in the output directory.


```

public static void parse(String path) {
    final Scanner scanner = new Scanner(SourceReader.readSource(Path.of(path)));
    final String className = WriterClass.pathToClassName(path);
    BytecodeGenerator.createClass(className);
    while (scanner.hasNextLine()) {
        index = 0;
        line = Tokenizer.tokenize(scanner.nextLine());
        statement();
    }
    WriterClass.writeClass(className, BytecodeGenerator.closeClass());
}

```

Fig. 5: Parse method of Parser class

Function `statement()` has the role of going through all the tokens of a line of code and verifying the grammar. Takes the initial input token and based on the type of the symbol tries to find a possible parsing path through multiple conditional structures. If the code is a valid statement, the specific course of execution is determined based on the the expression inside the statement and is checked by the `expression()` function and calls the generator methods for that equivalent line of code.

The Expression function receives an integer value that indicates the type of statement the expression is intended for, a path decider mechanism for a strategy-like pattern. Each path has multiple branches and grammar checks for identifying whether it is a valid code sequence. For either case, it is the job of this function to take action in each case. In the situation in which an unexpected symbol is found, a context sensitive error notifies the programmer that some illegal token has been processed and offers a helping message back to the user. If the entire expression is deemed correct at the end of the line, it is the job of the expression function to call the specialised bytecode generator methods dynamically.

For checking the correctness of the token sequence, two methods have been defined: `check()` and `match()`. When the parser reaches a branching point, multiple conditional blocks are defined in order to decide the path the grammar is going with the help of the check method that simply returns true if the current token is the expected one, or false if it could not be validated. In case the parser is inside a non branching situation the match method is called. If the current token matches the expected token, it increments the index of the current position inside the line advancing to the following token to be processed, otherwise an

exception will be thrown since no other token should be found at that time. The two methods are pretty similar, the difference being that “check” looks at the current token and only notifies back the caller if it fits the expected symbol, on the other hand “match” is deterministic and stops the entire execution flow of the compiler if the expected token does not match the current one.

The type of the current token is determined by the `nextSymbol()` function which matches the value of the characters with a symbol defined by the `Symbol` interface with the help of regular expressions and string matching in case of reserved keywords. If the current token doesn't match any key word of the ByteNet language, then checks if the token is numeric, in that case it will be treated as an arithmetic operand or assigned value, either as a float or an integer value, based on its shape of a floating point number or not. If the token is not numeric and no language key words were matched, then the token will be considered to be a potential identifier name for a previously declared variable, in that situation is the job of the caller function to determine if the name was previously used by the programmer in the declaration of a variable, or in the alternative case, the token is declared unidentified and a context sensitive exception is thrown. In case the line has reached its end, a special end of the line (EOL) symbol will be returned. The following line is retrieved inside the while loop within `parse()` when the `statement()` execution is finished, since the ByteNet language supports only one statement per line.

The `Tokenizer` class has declared internally a `tokenize()` function with a `StringBuilder` for attaching the characters of a single token, an `ArrayList` of `Strings` in which the tokens are added and a `Scanner` that divides every character of a line into an individual string. A while loop of `scanner.hasNext()` goes through the entire line fed to the scanner and checks at every step for delimiters of a token. The delimiters for the end of a token are: blank space, dot and left and right parentheses. If a delimiter has been found, the current token's string builder is converted into a string, added to the array of tokens and the string builder is then reset to empty for processing the next token. Finally, the array of tokens is returned to the caller function of the parser at the end of every line of code.

Class `BytecodeGenerator` offers multiple static methods that call sequences of methods for writing Java Bytecode from the ASM framework for the specific scenario the parser has reached. In this class, there are declared two global variables for the entire class

that manage an instance of `ClassWriter`, which provides methods for defining the structure of a Java class, and `MethodWriter`, that gives access to functions for managing the bytecode of a single method, more specific: the main of the class. Almost all methods do not return anything, instead they write the bytes directly through the ASM's `ClassWriter` and `MethodWriter` instances, except the `visitLabel()` that returns the label of the next bytecode operation and the `closeClass()` which retrieves the byte array from the `ClassWriter` and yields it to the parser. The compiler is designed to compile the source code statements into a standard Java main method that can be executed by the virtual machine, thus all the generation calls are done on a singular `MethodVisitor` instance[13].

The `Symbol` enum class contains all the defined types of the ByteNet language. The Parser makes use of this class for identifying which type each token is and matches each of them to the expected token by the grammar definition. Serves as a way of identification for each sequence of characters fed to the compiler.

3.4 Methods implementation

ByteNet has implemented several methods to help the user manipulate and observe the data used for several operations. One of the most important is “print” which gives the user the possibility to read the data inside the program from the terminal. Print is a dynamic method and knows how to behave in several situations. For example, based on the data type it is provided, it decides how it will act and what bytecode instructions are necessary in order to display the information to the console.

Much like the `System.out.println()` from java, print needs to know what is the type of data it works with. ByteNet offers the possibility to print integers, floating point numbers, strings and even the results of other methods. For primitive data types it simply analyses the token by getting the type of `Symbol` returned by the parser and judges how the bytecode will be generated. If the symbol is an identifier, the parser will search for it in the `Identifiers` hashmap and will retrieve its data type. The specific bytecode instructions for getting the assigned value to that variable are invoked in the back not visible to the user, thus the printing is done on a single line.

When it comes to Socket methods, the parser first checks the type of the variable and if the type matches a socket, the operation can be validated. The bytecode generation methods are called for that method, send or receive. “Send” can receive as a parameter either a locally written string or a previously declared and initialised string instance. In case of receive, when the JVM executes the program, the byte stream received over the network won’t be displayed unless the programmer calls print before it, similarly to Java. Print gets the byte stream from the datagram packet and formats the input for a user-friendly console output. It internally calls multiple Java methods: the received data from the packet’s buffer is converted from a byte stream into string format, then the string is trimmed of any unnecessary buffer spaces and then it is displayed to the console.

All of these operations done behind the scenes help the user achieve more by writing less code. The methods were designed to give optimal solutions to most common use cases of the User Datagram Protocol communication methods over the network in Java.

3.5 Generated bytecode

The Java Bytecode from the class file resulting from the ByteNet Compiler’s output is designed to respect the structure specifications of the Java Virtual Machine. This code is dynamically generated sequentially at every step of compilation, more precisely, at the end of every parsing validation, the corresponding intermediary code equivalent is generated. The ASM library helps generate the desired bytecode in binary form, but it is still the job of the compiler to determine the intermediary code operations, their arguments and their order of execution.

Since the class loader passes through each instruction sequentially only once, it is pretty convenient to generate the code in the same manner, sequentially as each line is validated by the parser’s methods. But there are certain instructions, such as the jump instructions, that need supplementary information by the JVM in order for the bytecode to be correctly optimised and interpreted. For correctly identifying the type of the variables, their values and if they are inside the scope of the structure, Oracle introduced with Java 7 a way of offering the virtual machine metadata about the variables within that scope without demanding the interpreter to verify the code again and analyse the type of the variables,

called frames. In this way the interpretation is much faster done in a single pass. All of the frames are stored in order in an internal structure called the StackMapFrame, these are saved as differences of the previous frame to save space and prevent storing the information about the same variables twice. The compiler manages the frames creation at every jump instruction of the language, mainly the loop and the exit condition that jumps the execution outside the repetitive structure.[14] A visit is done at the begging of the loop to capture the state of the constant pool, when the conditional structure is processed a new jump is needed for the case in which the exit condition is met and a third frame is created at the end of the loop for the jump instruction that goes to the beginning of the loop.

When calling the ASM's visitFrame() method it takes as arguments, the type of the frame, the number of variables declared, a list of their types which is internally managed by the compiler and assigns each type to the position it is encountered in and the frame stack (if there is one already). The visitLabel() method visits a declared label instance and instantiates it with the current position of the operand stack, a way of letting the interpreter know what is the intended location of a jump instruction.[15]

```

7: iload_1
8: ldc          #11          // int 5
10: if_icmpne    16
13: goto         31

```

Fig. 6: Bytecode instructions sequence for a conditional structure

Here there is an example of two jump instructions that translate as a conditional expression in Java. In the bytecode above the instruction at offset 7 loads the first variable stored, at bytecode offset 8 the interpreter pushes a constant 5 integer value, next instruction: if_icmpne, gets the previous two values loaded and checks if they are not equal, if the value returned is true it proceeds to the instruction at offset 16, if the value returned is false then it goes to the next instruction which, in this case, is at offset 13 and this instruction is a direct GOTO jump instruction to bytecode offset 31.[16]

Offset (h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00000000	CA	FE	BA	BE	00	00	00	41	00	1B	01	00	04	74	65	73
00000010	74	07	00	01	01	00	10	6A	61	76	61	2F	6C	61	6E	67
00000020	2F	4F	62	6A	65	63	74	07	00	03	01	00	04	6D	61	69
00000030	6E	01	00	16	28	5B	4C	6A	61	76	61	2F	6C	61	6E	67
00000040	2F	53	74	72	69	6E	67	3B	29	56	01	00	07	6D	65	73
00000050	73	61	67	65	08	00	07	01	00	10	6A	61	76	61	2F	6C
00000060	61	6E	67	2F	53	74	72	69	6E	67	07	00	09	03	00	00
00000070	00	05	03	00	00	00	01	01	00	10	6A	61	76	61	2F	6C
00000080	61	6E	67	2F	53	79	73	74	65	6D	07	00	0D	01	00	03
00000090	6F	75	74	01	00	15	4C	6A	61	76	61	2F	69	6F	2F	50
000000A0	72	69	6E	74	53	74	72	65	61	6D	3B	0C	00	0F	00	10
000000B0	09	00	0E	00	11	01	00	13	6A	61	76	61	2F	69	6F	2F
000000C0	50	72	69	6E	74	53	74	72	65	61	6D	07	00	13	01	00
000000D0	07	70	72	69	6E	74	6C	6E	01	00	15	28	4C	6A	61	76
000000E0	61	2F	6C	61	6E	67	2F	53	74	72	69	6E	67	3B	29	56
000000F0	0C	00	15	00	16	0A	00	14	00	17	01	00	04	43	6F	64
00000100	65	01	00	0D	53	74	61	63	6B	4D	61	70	54	61	62	6C
00000110	65	00	21	00	02	00	04	00	00	00	00	00	01	00	09	00
00000120	05	00	06	00	01	00	19	00	00	00	3D	00	02	00	03	00
00000130	00	00	20	11	00	00	3C	12	08	4D	1B	12	0B	A0	00	06
00000140	A7	00	12	1B	12	0C	60	3C	B2	00	12	2C	B6	00	18	A7
00000150	FF	EB	B1	00	00	00	01	00	1A	00	00	00	0B	00	03	FD
00000160	00	07	01	07	00	0A	08	0E	00	00						

Fig. 7: Java class file in hexadecimal values

The above file is a class file written in Java Bytecode seen with a hexadecimal number representation. The highlighted sequence in red is the binary representation of the previous four bytecode instructions. The following code is the decompiled bytecode instructions in Java using the IntelliJ integrated decompiler. Conditional structure from above translates as a while condition, first the variable var1 is loaded as an integer, then the constant 5 is pushed on to the stack and then the different operator is applied between the two numbers.

```

public class test {
    public static void main(String[] var0) {
        int var1 = 0;
        String var2 = "message";

        while(var1 != 5) {
            ++var1;
            System.out.println(var2);
        }
    }
}

```

Fig. 8: Decompiled bytecode in Java from

Since the main focus of ByteNet is simplifying the network communications of UDP instances, the compiler internally handles port and address at the declaration of a Socket variable and stores their values until any methods are called upon it. When the send method is called upon a Socket instance, all the previously provided information in the declaration is loaded into memory and the port and address are set internally.

```

0: new          #7                // class java/net/DatagramSocket
3: dup
4: sipush       7777
7: invokespecial #9                // Method java/net/DatagramSocket."<init>":(I)V
10: astore_1

```

Fig. 9: Bytecode instructions sequence for instantiating a DatagramSocket object

Instantiating a DatagramSocket is no different than any other object in Java. First the new instruction is called for getting the DatagramSocket class from java.net files, dup instruction duplicates the object reference on the stack, so when the constructor pops the reference there is still one remaining, the port number or the constructor argument is pushed

as a short value and invokespecial instruction gets the class constructor and invokes it using the pushed values on the stack. Finally astore_1 saves the object reference as variable 1. After this point, the specific methods of the class can be called on this variable.

Chapter 4

Graphical User Interface

4.1 General Design

The second half of the applicative part is a graphical user interface that helps the client write and visualise the ByteNet source code, invoke compilation, run the program and see how the results are printed or returned. This part of the application was written in Java with the help of the JavaFX library for creating graphic desktop applications. The main purpose of this is to mimic the behaviour of an IDE which provides tools for source file management, gives the user the possibility to modify, edit, compile and run those files and a console that allows reading the results of a given program run by the JVM.

When started, the application presents at the top a menu with multiple options for creating, opening, closing or saving source files, a settings button that opens a new JavaFX Scene for various configurations and a help button with basic information about the software. After that, there is a title logo for the compiler and in the middle of the window there are two JavaFX TextAreas corresponding to two source files and right below them a compile and run button for each of them. At the bottom on black background colour and light green text colour (an imitation of the old computers command line colour palette), there are two terminals that display the results of all the commands given to the program.

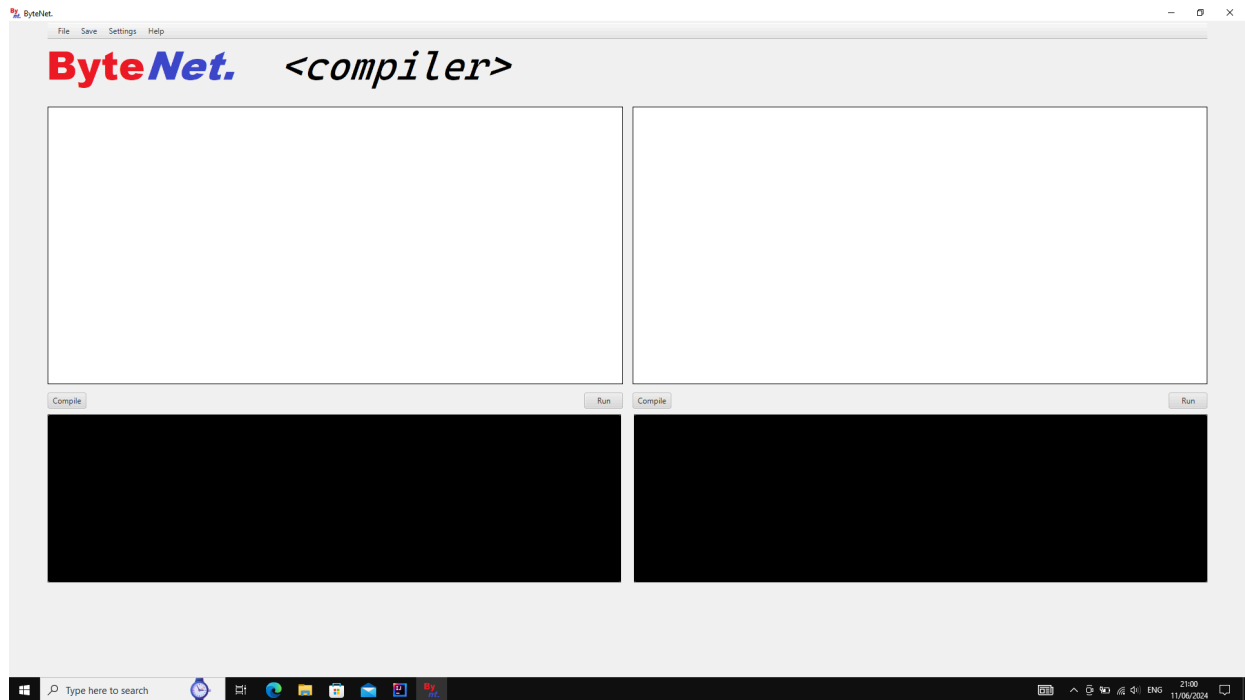


Fig. 10: Main window of the graphical interface

4.2 File management and configurations

File handling is done through several string instances declared inside the main controller class. When the application starts, the default values for these are set to null and all functionalities regarding the source files (such as run or compile) are blocked until a new file is created or opened. An existing source file on the local machine can be opened through the context menu's option, when that action is performed the text area designed for code visualisation and editing is filled with the contents of that file and the internal instance of the controller receives the absolute path of that file. The two source files are handled independently from one another and each time the application is closed, all the opened files are closed as well. Closing the files wipes clean the source code text area and set the value for the current file back to null again.

This graphical user interface for the ByteNet compiler needs several configurations before use. For this, a new separated scene has been designed for this purpose and it can be accessed through the menu option Settings > Configure. The necessary configurations are:

- Providing the path to the ByteNet compiler: absolute path to where it has been downloaded to the computer
- Providing the absolute path to where the ow2.asm library jar is stored in the computer, this is very much needed for calling multiple system commands behind the scenes for the compiler
- And lastly, offering the absolute path to the directory in which the resulting class files of the compiler are stored, the compiler output file

These configurations are mandatory in the usage of this software, without them the graphical interface would have no binding to the software it was designed to support. It is necessary to provide this information to the application only once, because the graphical interface application has internal configuration files which store the values fed to it and loads them into memory every time the application starts. After everything is set the user can use one of the following buttons: Apply, which saves the values to the configuration files and OK, which switches back to the main scene. It is important to keep in mind that the OK button does not save the configurations and returns without confirmation, the Apply is necessary for saving before going back.

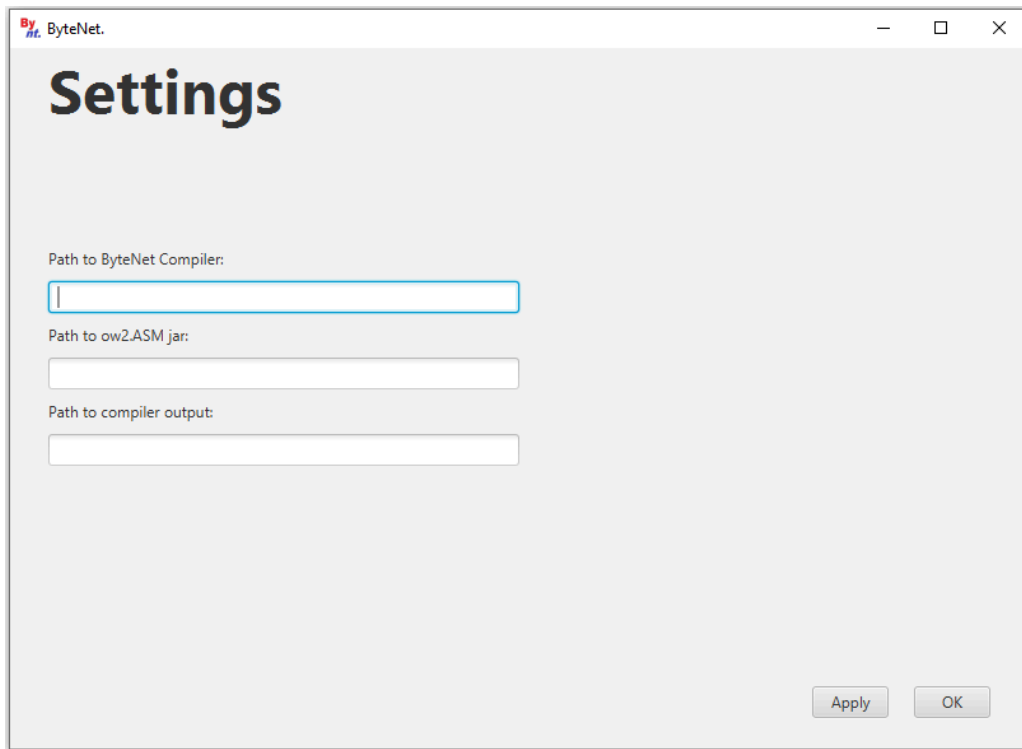


Fig. 11: Settings/configurations window

4.3 Compilation and execution

When the user presses the compile button the app will first check if there is an open file to be compiled, if no file is already open, the main controller will check if the text area is populated with any code, if not, nothing will happen, if there is any written code, the content will be first saved in a new source file with a default name “source1.bynt” or “source2.bynt” depending on the text area used for writing. After these verifications were made, the existing source file or the newly created file will be passed to the compiler using the “Runtime.getRuntime().exec()” code sequence, this will execute the command directly to the system command line and will call the Java Virtual Machine to start the compiler and give to it the file as an argument. The command contains the paths to the ASM library jar and the path to the compiler itself, information that is extracted from the previously mentioned configuration files. After compiling a file, several messages will be displayed to the terminal to confirm that the steps have been completed. Contrary, if any error has occurred or the

source code is not validated by the ByteNet parser, the information will be displayed in the terminal.

If the Run button is pressed the main controller will look for any opened files, if no opened file is found, an alert window will appear informing the user of this. If the user desires to run the code he wrote in the text area, they will need to first save that code to a file with the extension .bynt and compile it.

When executing the ByteNet code, the application will search for its equivalent Java class file in the compiler's output directory. Another system command will call the Java Virtual Machine again to execute the class file and its output will be displayed in the terminal after a message that signals the beginning of execution. If any errors are returned, they will be printed in the terminal and if the program ran successfully a confirmation message will be displayed at the end of the execution flow.

Since JavaFX is a single thread application, displaying messages or other output information to the user has to be done separately from any other processing that needs to be made. For displaying appended pieces of text in a text area, the program needs to finish its cycle in order to update the contents.[17] In order to overcome this limitation, the text appending is in a separate dedicated function, so after it is called and processed, the function cycle finishes and the contents are displayed on the screen. For long and demanding tasks, such as running a sequence of code that contains a loop, running the commands is done on a new separate thread using a local lambda expression and waits for receiving the results of execution. Displaying the messages returned by the background processes is done on the main application thread by calling `Platform.runLater()`, which handles the delivery of information between the two threads.

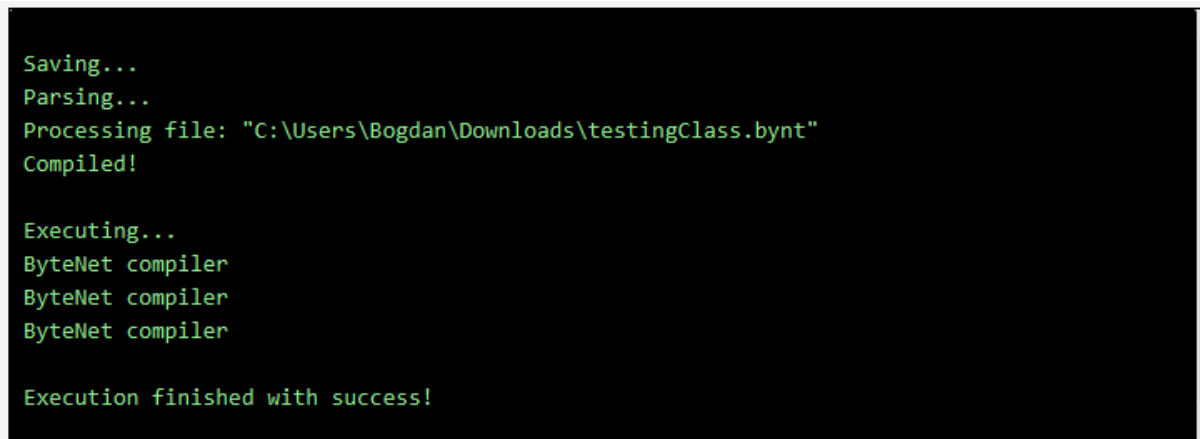
This way it is very easy to track how the program behaves and to observe when its execution finished running and what are the outputs of that code sequence. "Executing..." message marks the beginning of the flow and ends with "Execution finished with success" or "Error", anything in between is the output of the program.

```

string s = "ByteNet compiler"
int i = 0
loop {
    when i == 3 exit
    i 1 INC
    print s
}

```

Fig. 12: ByteNet sample code



```

Saving...
Parsing...
Processing file: "C:\Users\Bogdan\Downloads\testingClass.bynt"
Compiled!

Executing...
ByteNet compiler
ByteNet compiler
ByteNet compiler

Execution finished with success!

```

Fig. 13: Bytenet code execution results

In the first of images above, there is presented a piece of ByteNet code with a loop that prints the string “ByteNet compiler” for 3 times. In the second image there is a screenshot of the graphical interface’s terminal. In it there is the output message of the compilation phase, which informs the user of each step of processing and the file path of the source code and then the execution messages, the string data from the output and a final message that confirms the execution finished without any errors or exceptions.

Chapter 5

Conclusions

Designing this solution helped me learn a lot about compiler architecture, the components of a compiler, the data flow process and how a high-level language with many capabilities is translated into low level instructions for a machine, either virtual or physical. This learning journey gave me a better understanding of some limitations of a source code translator and how some code sequences can be more efficient than others in achieving the same objective. At the end of this thesis, looking at the world of compilers seems to be just an vast introduction to all branches of programming and software development, it offers many new perspectives about security, optimization, networking, software or hardware architectures and many others aspects of information processing done at the computer level, since every program is processed and executed on physical computer somewhere even if it is done remotely through web services or on the personal device.

Creating a new programming language involves thinking about an optimization to the source code one can make in order to offer the end-user, the programmer, a better, simpler and more efficient way of writing code for a specific purpose. For my solution I decided to create a shorter syntax for server-client data transmission using the User Datagram Packet protocol. As any other programming language, ByteNet code, my solution's name, offers basic instruments for doing simple calculations with integers and floating point numbers, storing values in variable instances and means for opening repetitive or conditional structures to control how the data is manipulated. This language has the ability of creating UDP instances for sending and receiving string messages and presents methods for printing the results to the console and formats the content.

Graphical interfaces in software development are a huge boost for the productivity of the programmer, making it very easy to run, test or save and visualise the written source code. For my solution, my graphical application created using the JavaFX library, has the purpose of facilitating the demonstration of how the ByteNet compiler processes the source code. The output of compilation can be observed in the console in the bottom of the main window, either as a success message or error log for invalid syntax or other wrong inputs, this way

notifying the user of a possible mistake. The interface also offers the possibility of calling the JVM for executing the resulting class file with a simple button click, its output will be displayed to the user in the console area.

Java was another important aspect of this project, I got to learn more about one of the most popular and more versatile programming languages in the world at the moment. With this work I got to study about almost every layer of the language and how it is processed from a high level language to an intermediary code in the form of a sequence of instructions for its virtual machine and then translated to physical machine code depending on the computer the Java Virtual Machine is installed on. Java provides numerous opportunities for creating software in nearly every field of development, but for some applications the syntax can be quite long and verbose for some simple operations. I got to reflect more about this sort of problem and familiarise myself more with Java.

For the future I am interested in learning more about computer level processing and how data is manipulated behind the scenes. Nowadays even the programmer gets to work in a very friendly environment in terms of interfaces and various processes are facilitated for them in order to increase productivity of work. This can sometimes become a drawback on performance or security matters of some applications, since there is less control over every single process inside the machine.

This solution is an example of an optimised syntax for simplifying writing code for one particular scenario. Offers quick access to the methods needed in order to do message transmission as quickly and as simple as possible over the network. Compared to languages like Java, besides the better writing time advantage, this can be a drawback because the user has less control over the operations made behind the scenes, which is a valid trade off for every language, it is the freedom of the developer to choose the syntax or the platform which is most suitable for his application that brings him the best productivity and results. I am determined to learn more about this subject and work in the future in creating helpful software solutions for bringing the world of computers a step forward.

Bibliography

- [1] Alfred Aho, “Compilers - Principles, Techniques and Tools” Second Edition, Publisher: Pearson Education Inc., year: 2006, ISBN: 0-201-10088-6
- [2] “Compiler”, n.d., para 2., <https://en.wikipedia.org/wiki/Compiler#History>, [Accessed December 2023]
- [3] Douglas Thain, “Introduction to Compilers and Language Design” first edition, publisher: Lulu, year: 2019, ISBN: 978-0-359-14283-5
- [4] Contributor: goelshubhangi3118, “Introduction to Java”, date: 03.April.2023, <https://www.geeksforgeeks.org/introduction-to-java/>, [Accessed December 2023]
- [5] Cristian Valeriu Toma, "Java Programming–Software App Development.", lecture 2, year: 2024, https://acs.ase.ro/Media/Default/documents/java/lectures/c02_JSE_Intro_OOP_RoboCode_Strings_RecapOOP.pdf, [Accessed February 2024]
- [6] Bill Venners, “Introduction to Java’s Architecture”, “Chapter 1 of Inside the Java Virtual Machine”, n.d., <https://www.artima.com/insidejvm/ed2/introarch.html>, [Accessed March 2024]
- [7] Scott Selikoff, Jeanne Boyarsky, “OCP Oracle Certified Professional Java SE 17 Developer Study Guide”, Publisher: Sybex, year: 2022, ISBN: 1-119-86458-5
- [8] Tim Lindholm, “The Java Virtual Machine Specification”, Publisher: Oracle, year: 2015, ISBN 978-0-133-90590-8
- [9] Tim Lindholm, Frank Yellin, Gilad Bracha, Alex Buckley, Daniel Smith, "The Java [15]Virtual Machine Specification Java SE 21 Edition", date: 23.August.2023 <https://docs.oracle.com/javase/specs/jvms/se21/html/index.html>, [Accessed February 2024]
- [10] Tim Lindholm, Frank Yellin, Gilad Bracha, Alex Buckley, Daniel Smith, “Chapter 4. The class File Format”, date: 23.August.2023, <https://docs.oracle.com/javase/specs/jvms/se21/html/jvms-4.html#jvms-4.1-200-B.2>
- [11] Eric Bruneton, Romain Lenglet, Thierry Coupaye, “ASM: a code manipulation tool to implement adaptable systems”, n.d., <https://citeseerx.ist.psu.edu/pdf/5f4da1df76b8878bf8358ec24d6592a8008d2b0d>, [Accessed February 2024]
- [12] Cristian Valeriu Toma, "Java Programming–Software App Development.", lecture 10, year: 2024,

- https://github.com/critoma/javase/blob/master/presentations/c10_JSE_TCP_UDP_Dev.pdf, [Accessed February 2024]
- [13] Kevin Gilmore, “A Guide to Java Bytecode Manipulation with ASM”, date: 08.January.2024, <https://www.baeldung.com/java-asm>, [Accessed March 2024]
- [14] User: Antimony, “What is a stack map frame”, date: 04.August.2014, <https://stackoverflow.com/a/25110513>, [Accessed March 2024]
- [15] Eric Bruneton, Eugene Kuleshov, Andrei Loskutov, Rémi Forax, “ObjectASM Javadoc”, date: 2023, <https://asm.ow2.io/javadoc/index.html>, [Accessed March 2024]
- [16] “List of Java bytecode instructions”, n.d., table, https://en.wikipedia.org/wiki/List_of_Java_bytecode_instructions, [Accessed February 2024]
- [17] Jonathan Giles, “JavaFX API Documentation”, n.d., <https://openjfx.io/javadoc/22>, [Accessed April 2024]