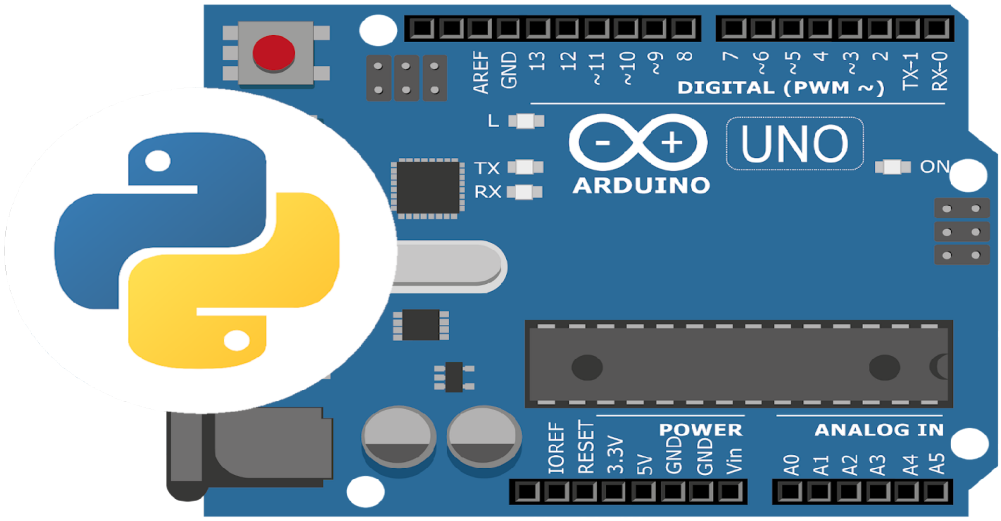
CHALMERS
UNIVERSITY OF TECHNOLOGY



NanoPython

Researching the Feasibility of Running Python Scripts on Arduino Using a Virtual Machine

BACHELOR'S THESIS IN COMPUTER SCIENCE AND ENGINEERING

TOBIAS CAMPBELL
KARL STRÅLMAN

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
GOTHENBURG, SWEDEN 2020

Degree Project Report

NanoPython

Researching the Feasibility of Running Python Scripts on Arduino Using a Virtual Machine

Tobias Campbell
Karl Strålman

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2020

NanoPython
Researching the Feasibility of Running Python Scripts on Arduino Using a Virtual Machine
Tobias Campbell, Karl Strålman

Examiner: Peter Lundin, Department of Computer Science and Engineering
Supervisors: Andreas Abel, Department of Computer Science and Engineering,
Björn Bergholm, Broccoli

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
SE-412 96 Göteborg Sweden
Telephone: +46 (0)31-772 1000

Cover: Made by Karl Strålman with a composition of images, Arduino Uno by Seven_au from Pixabay and Python logo from Python's official site.

Department of Computer Science and Engineering
Gothenburg, Sweden 2020

Sammanfattning

Virtualisering är idag en prominent metod både för att distribuera och underhålla applikationer till en låg kostnad. Det finns flera typer av virtuella maskiner inom olika datorvetenskapliga områden. Det svenska konsultföretaget Broccoli vill undersöka genomförbarheten av att exekvera Python-skripts på inbäddade system. Denna rapport fokuserar främst på virtuella maskiner som behandlar programmeringsspråk, kompilatorer och möjligheten att bygga en virtuell maskin för behandling av Python på utvecklingskortet Arduino Uno. Med en delmängd av Pythons grammatik som utgångspunkt utvecklades en kompilator skriven i Java för att generera bytekod med hjälp av verktyget ANTLR. Samtidigt utvecklades en virtuell maskin i C++ för att exekvera kompilatorgenererad bytekod. Genom att utbygga grammatiken med små grammatikstycken, och därefter skriva kod för att hantera den nya grammatiken, kunde mer funktionalitet bli tillagt till både kompilatorn och den virtuella maskinen. Användandet av tester garanterade projektets funktionalitet. Hela språkbearbetningen, där lexikalanalys, syntaktisk analys och kodgenerering är inkluderat, utförs av kompilatorn. Kod genereras därefter i ett specifikt binärt filformat. Detta möjliggör exekvering av enkla Python-skript, på både PC och det valda systemet, utan att ta hänsyn till optimering för varken kompilering eller exekvering. Vissa problem som kvarstår inkluderar det begränsade minnesutrymmet på Arduino:n samt brist på stöd för vissa språkdefinierande konstruktioner i Python. Trots dessa motgångar kan i framtiden den nuvarande implementationen, med ytterligare utveckling, användas som ett snabbt sätt att distribuera tester, vilket är önskevärt av Broccoli.

Nyckelord: Python, Java, C++, ANTLR, Language virtualization, Virtual Machine, Compilation chain, Embedded Systems, Arduino, Programming Language Technology

Abstract

Virtualization today is a prominent method of both deploying and maintaining applications at a low cost. Various types of virtual machines exist and are used in different fields of computer science. Searching for new innovative ways to deploy tests, Swedish consultancy firm Broccoli wants to research the feasibility of running Python on an embedded system. This thesis primarily focuses on language virtual machines, the compilation involved and researching the feasibility of developing a Python virtual machine for a particular embedded system development board, an Arduino Uno. Starting with a subset of the Python grammar, a bytecode compiler written in Java was developed with the ANTLR tool. Meanwhile, a virtual machine running the compiler-generated bytecode was developed using C++. By adding small fragments to the grammar and accordingly writing additional code to account for this, more functionality was added to both the compiler and the virtual machine. By utilizing tests, functionality of the project could be preserved. The compiler runs the whole process of language processing, including lexing, parsing and code generation. It then generates code in a specific binary file format. A binary file can then be interpreted by the virtual machine. This enables the execution of simple Python scripts on both PC and the chosen system, without regard to optimization for neither compilation nor execution. Although not perfect, the current implementation does run on both systems. Some problems include the limited memory space of the Arduino and the lack of support for important language-defining constructs. Despite the drawbacks, with further improvements the current implementation may be used in the future as a means to rapidly deploy tests, which is of interest to Broccoli.

Keywords: Python, Java, C++, ANTLR, Language virtualization, Virtual Machine, Compilation chain, Embedded Systems, Arduino, Programming Language Technology

Preface

Due to the current situation with COVID-19, we have been unable to spend as much time at Broccoli's office as we would have liked. We regardless would like to thank Broccoli for the hospitality, the tasty coffee and of course the Friday fika. We would also like to thank our supervisors Andreas and Björn for their help in the project, and Tommie Campbell for help with the language and proofreading. Last but not least, we would like to thank Robert for spiritually being with us the whole project.

Tobias Campbell and Karl Strålman, Gothenburg, June 2020

Technical terms and abbreviations

ISA - Instruction set architecture

CLI - Command-line interface

IDE - Integrated development environment

JVM - Java virtual machine

PVM - Python virtual machine

SVM - Snok virtual machine

OOP - Object-oriented programming

TOS - Top of stack

AST - Abstract syntax tree

ANTLR - ANother Tool for Language Recognition

Python - Python and CPython are used interchangeably in this thesis

Contents

List of Figures

List of Tables

1 Introduction

1.1 Background

Broccoli is a consultancy firm specialized in hardware and software development, with focus in three areas: engineers, education and self-developed embedded systems. They are a small company in Gothenburg with ten members in their administrative team and about 50 employed consultants. They want help to develop a new, innovative way to deploy scripts in order to unit test cars, using a microcontroller such as an Arduino Uno and connecting it to sensors of the vehicle. Additionally, they want to know about the advantages and disadvantages of using Python as the chosen script language. As such, one of the main problems will be to interpret and run a portion of the Python script language on the microcontroller.

1.2 Motivation

This thesis is inspired by a similar project, MicroPython, which is a Python compiler and virtual machine for ARM-processors [1]. Python is designed to be run on a desktop environment and can not be directly compiled, programmed and run on an embedded system without being specifically designed for that system. A Python implementation can be performed in various ways, for example having the compiler and virtual machine bundled together on the machine, or as two separate applications (compiler on PC, virtual machine on target system); the latter is the choice of this project. Arduino Uno was chosen partly because of how easy it is to program it, requiring no external programmer, and partly because of the challenge of fitting the VM on its limited memory size.

1.3 Aim

The aim of this project is to run a Python script on an Arduino Uno, while staying true to Python's format and constructs. Additionally, these questions will be answered throughout the report.

- *What are the pros and cons of using Python?*
- *Is it feasible to use Python as the chosen language on the Arduino?*
- *How much of the Python language can we implement on the Arduino (subset of Python)?*

1.3.1 Limitations

Since Python has been in development for over 20 years, everything that Python supports can not be reimplemented in this project due to time constraints. Therefore, only a subset of chosen constructs will be supported, and the constructs that are planned to be supported are the following:

Construct	Type	Example
SExp	Statement	1 + 4
SAss	Statement	x = 4
SPrint	Statement	print("hi")
SBlock	Statement	[stms]
EId	Expression	x
EString	Expression	"hi"
EInt	Expression	42
EAdd	Expression	x + 4 + 2
ESub	Expression	x - 4 - 2
EDiv	Expression	x / 4 / 2
EMul	Expression	x * 4 * 2
EUnary	Expression	-4, 19, -+-4

Table 1.1: Planned constructs to support.

Since the application will run on an embedded system, with limited memory, the program size of the VM limits what can be included in the subset of Python or functionality in itself. It is also important to emphasize that the available memory during runtime on the Arduino is limited.

1.4 Objective

The first objective is to create a Python compiler that compiles Python scripts to Python assembly code, which will then be assembled to bytecode. Then a virtual machine will have to be constructed to interpret and execute bytecode.

In a more concrete manner, the software that will be developed:

- a compiler (lexer, parser, typechecker, compiler) that compiles Python scripts to Python assembly.
- an assembler to produce Python bytecode in a binary format.
- a virtual machine - a program which interprets and executes Python bytecode.

2 Theory and technical background

2.1 The Python Language

Python is an interpreted, object-oriented, high-level programming language. It is commonly used as a script or "glue language", which is used to connect different components together [2]. The language is a dynamically typed language, meaning it does not enforce or check type-safety at compile time, deferring such checks until runtime, as opposed to a statically typed language. Although Python is primarily an interpreted language, internally it has distinct compilation and execution phases [3].

Section 2.2 will cover the language processing for a generic language, from the initial phase of lexing to the final phase of code generation. In Python's case, code generation refers to creating *.pyc* files which are used to accelerate the process of subsequent runs of the imported script. Following is section 2.3, which will cover the execution phase of an interpreted language in terms of a virtual machine.

2.2 Language Processing

A compiler is a program that translates code written in one programming language into another language, commonly translating a high-level language to a lower level language. It does not run programs, whereas an interpreter does, despite containing the same language processing phases. When a compiler is analyzing and processing code, the process is divided into the following compilation phases: lexing, parsing, type-checking and code generation. Each phase is part of a pipeline that produces code from one format to another [4].

- The lexer analyzes and converts text into **tokens**.
- The parser matches tokens with predefined rules and creates a **parse tree**, which can be used to create an **abstract syntax tree** (AST).
- The typechecker analyzes the AST and adds extra information, creating an **annotated syntax tree**.
- The code generator converts the annotated syntax tree into target code instructions.

2.2.1 Lexing and Parsing

The objective of a lexer is to analyze a stream of text, e.g. a source file, and convert it to a stream of tokens, in order to categorize the text. This stream is later used by the parser to perform a syntactic analysis of the inputted text[4]. A token can be constructed in many ways: through a specific character, e.g. the '(' character; through the use of multiple tokens, like FLOAT which contains multiple NUMBER tokens; or through token fragments, such as

the NAME token, which starts with the the ID_START token fragment and is followed up by an arbitrary number of ID_CONTINUE token fragments. An example of a lexical analysis, including an example with fragments, can be seen below in Table 2.1.

Input code	**Tokens found**
a = 1 + 2	a = NAME (ID_START), 1 = INTEGER, 2 = INTEGER
var = "test"	var = NAME (ID_START, ID_CONTINUE), "test" = STRING

Table 2.1: Inputted code and their tokens.

Utilizing the token stream generated from the lexer, the parser matches the tokens in the token stream with parser rules defined in a grammar file, generating a parse tree containing the structure of the inputted program. An example of parser rules for a while loop is seen below.

```
sWhile: 'while' eCond ':' sBlock;
```

Figure 2.1: Parser rules for a Python while loop e.g. "while True : methodCall()"

These rules are used to identify what the program is supposed to do syntactically, i.e. what kinds of constructs of a programming language is used. The parse tree for the first example in Table 2.1 is shown in figure 2.2.

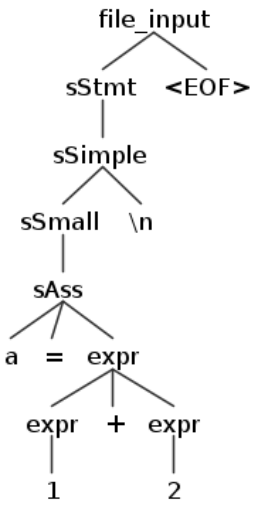


Figure 2.2: Parse tree for 'a=1+2'

Constructs in a language are commonly separated between statements and expressions, where statements can be seen as instructions that cause side effects that affect the environment they are run in, while expressions can be seen as values, variables, operators or function calls [4][5]. For both examples in Table 2.1, they follow the parser rules for an assignment, which is grammatically defined as an assignment statement, or sAss as it is defined in NanoPython's grammar. The full grammar used for NanoPython is included in appendix A.1.

By traversing the parse tree, an important structure called abstract syntax tree (AST) can be constructed. An AST can be seen as a more abstract version of the parse tree, which contains the statements and expressions, i.e. the only parts of the program that is necessary for the

compilation chain to continue [6]. The AST for the same example as for the parse tree can be seen in figure 2.3.

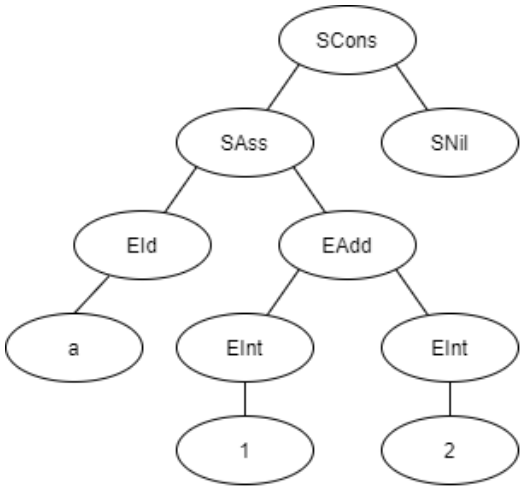


Figure 2.3: The abstract syntax tree for 'a=1+2'. Statements start with an 'S' whereas expressions start with an 'E'.

When the parser is finished, the generated AST is passed on to either the typechecker or the code generator.

2.2.2 Typechecking

The main objective of the typechecking phase is to find preventable errors in the code, that violate the chosen language's specification, either before a program is compiled or when it is run. A typechecker is aware of the context, i.e. what variables are declared and what types the variables have. Typecheckers that run during compile time are considered **static**, while the other typechecker is considered **dynamic** and is run during runtime. What kind of typechecker is used depends on whether or not the language is statically or dynamically typed. For instance, Python is a dynamically typed language, and thus uses dynamic typechecking[7].

By traversing the AST, the static typechecker compares nodes that contains statements or expressions with other nodes in the tree. In the AST in figure 2.3, after entering the node containing the statement*SAss* the typechecker needs to ensure that *EId(a)'s* type is coherent to what the expression *EAdd(EInt, EInt)* would infer, which is integer in this example. If both types are valid according to the specification, no error is thrown and the type checker continues. By storing type annotations while traversing the AST, a more informative tree can be constructed, i.e. an annotated syntax tree. The usage of the annotated syntax tree in the next phase of the compiler can enable more efficient binary code e.g. by emitting bytecode instructions that are designed for a specific type [4].

In contrast to a statically-typed language, a dynamically-typed language is primarily typechecked during runtime. Instead of traversing the AST to find preventable errors, the dynamic typechecking occurs as the bytecode instructions are interpreted. For instance, if an addition instruction is found, then it will check if the two operands are valid or not; if it is, then proceed as normal, and if not, throw an error. As with the static typechecker, if a variable is referenced where it isn't in the context, an error will also be thrown. A programming language can utilize both a static and dynamic typechecker, e.g. C++ [8].

The similarities and differences between typecheckers can be seen in the following examples,

which feature code snippets from Java (figure 2.4), utilizing static typechecking, and Python (figure 2.5), utilizing dynamic typechecking. Both typecheckers detect that the variables referenced in the following code snippets are not in the context and will throw an error, while the timing (before or during runtime) of the second error differs between the two.

```
// Java example
System.out.println(x); // error during compilation, can't find the variable x.
char x = 'a';
boolean y = true;
System.out.println(x); // 'a' is printed
System.out.println(x + y); // error during compilation, bad operand types for '+'
```

Figure 2.4: Example of static typechecking featuring Java

```
# Python example
print(x) # compiles, but gives an error (UnboundLocalError) during runtime;
         # won't print
x = 'a'
y = True
print(x) # 'a' is printed
print(x + y) # compiles, but gives an error (TypeError) during runtime; won't print
```

Figure 2.5: Example of dynamic typechecking featuring Python

Note that in figure 2.4, the error says that the variable can not be found, while in figure 2.5, the error means that the variable is used before it is referenced. This implies that the dynamic typechecker knows that x is a variable that will be assigned something, while the static typechecker is oblivious to the fact.

How strict the typechecking is depends wholly on the language used. For instance, in the previous example in figure 2.5, the Python interpreter throws a TypeError because a str object and a bool object can't be concatenated. In figure 2.6, featuring JavaScript, the code snippet works perfectly fine, because it is designed to be interpreted as such.

```
// JavaScript example
var x = 'a';
var y = true;
console.log(x); // 'a' is printed
console.log(x + y); // 'atrue' is printed
```

Figure 2.6: Example of dynamic typechecking featuring JavaScript

2.2.3 Code Generation

In the code generation phase of the compilation chain, the goal is to generate instructions that the target interpreter or runtime environment can understand and execute. In some programming languages, e.g. C, the annotated syntax tree is converted to machine code or assembly code, which is primarily designed to be run by a CPU or another piece of hardware. In other cases, such as Java, the annotated syntax tree is instead converted to bytecode or bytecode instructions, which is primarily designed to be run by a VM or a bytecode interpreter.

Assembly code and bytecode instructions are similar in the sense that they can be seen as a human readable format of their lower-level counterparts, i.e. machine code or bytecode, respectively. Since Python is a dynamically typed language, an AST will be used instead of an annotated syntax tree. Also, because Python bytecode, like JVM bytecode, is interpreted using a VM, the compiler will generate bytecode instead of machine code.

During the code generation of a program, the code generator traverses through a syntax tree in a similar fashion to how the typechecker traverses an AST. For every node in the syntax tree that is a statement, any expression or statement in the statement will be *compiled*, in order to generate the code needed to perform the function of that statement.

For instance, in the piece of code whose AST can be seen in figure 2.3,

```
a = 1 + 2
```

the statement will be compiled recursively. The first expressions to be compiled are the constants 1 and 2, and bytecode for the two are emitted because these are necessary for the VM or machine to run. Then the expression that relies on those, i.e. the addition, is compiled and bytecode is emitted. Finally, the variable a is compiled and the result is saved to that variable, and more bytecode is emitted for the statement. The details of the compilation process vary depending on the compiler, and an assignment may be handled differently by another compiler.

Aside from generating bytecode for instructions, the code generator must also generate bytecode for a variety of information: what constants are used and details about them, when the code was compiled, checksum, among other data, while outputting it in a format that is comprehensible for the machine interpreting the bytecode. The particular format used for this thesis is explained later in section 4.3.3.

As mentioned in section 2.1, the Python programming language may not only be interpreted, but also compiled to an intermediary format in the form of **.pyc* files. The contents of such a file is precisely the bytecode and all other information needed to run the code and is generated as part of the code generation phase. This implies that code generation is not necessarily the only final step, as it is possible to only interpret the code without generating bytecode for it. Compiling it into bytecode accelerates the process of interpreting though, since subsequent runs of the code can be done without the need to compile it again.

2.3 Virtual Machine

A virtual machine (VM) is a computing system where the goal is to execute programmed logic. This logic can be expressed at a very low level with all the details of an actual computer, essentially emulating hardware, or at a very high level with scripts. In this report, focus is given to the latter.

Virtual instruction set architecture (ISA) is a type of virtual language that defines the set and execution model of a virtual machine. A virtual language implying that it is generated by tools is only used for intermediate representation purposes, not to be readable by humans.

Using the definitions above, there are two common types of VMs for executing application code.

- Virtual ISA VM - a runtime engine that a virtual ISA can execute on e.g. JVM, Python.
- Language VM - executes from source code, no intermediate language e.g. Javascript.

VMs can be constructed in various ways, but common components of a virtual machine are considered to be: [9]

- Loader and Dynamic Linker - A loader loads the package of the application to be run into memory, parses the package into data structures and loads additional resources if needed. The dynamic linker may trigger the loader to load more data and code if something referenced is not loaded or if additional packages need to be loaded.
- Execution engine - After the application is loaded and linked, the execution engine performs the actual operations of application code. Since the purpose of application code is to execute, the execution engine is considered the core component of a VM. Two basic execution mechanisms are interpretation and compilation.
- Memory manager - The memory manager manages the internal memory and application data. Data that is considered *dead*, in the sense that it is no longer used and will not be used again, is deallocated and more memory space is available for other data to be loaded in the current runtime environment.

2.4 Python Virtual Machine (PVM)

In this thesis, the Python 3 bytecode interpreter will be referred to as Python virtual machine (PVM), and to avoid complex C implementation specifics, many parts are generalized. When Python interprets a file, it is compiled into bytecode subsequently interpreted by the PVM. To accelerate the initial parsing time, usually when importing other scripts, the bytecode is stored in a binary file (*.pyc). The fact that Python executes a virtual language implies that it is a virtual ISA machine [9].

Code that has been loaded into the PVM will be placed in code blocks, encapsulating the sequence of bytecode that was loaded. Code blocks, referred to as code objects in the VM, are central to the operation of PVM since it contains the program to be executed. The following are blocks: a module, a function body and a class definition. A code block is executed in an execution frame (PyFrame), which contains administrative and contextual information that is required to run a code block and determines how the execution continues after the code block's execution has completed. The frame contains the local and global symbol tables (scope), where constants, variables and globals are stored, and an evaluation stack [10], [11].

PVM is a virtual stack machine, and when it is running, all the computed values are evaluated on a stack. All values on the evaluation stack are regarded as PyObjects. A PyObject in the PVM is a construct that contains a type object and additional information used by the memory manager. All built-in data types are subtypes of PyObject, created as a type object, for example, an integer is placed into a PyObject called PyLong[1] [10]. An example of the stack can be seen in Table 2.2, two PyObjects, A and B, are pushed on the stack.

Index	Value	SP
1	PyObject A	x
0	PyObject B	

Table 2.2: The stack before any operation is perfomed. Stack pointer (SP) points to TOS

If the operation '+' is performed, without knowing the type of A and B and assuming that A's

[1] All Python's integers are implemented as "long" integer objects of arbitrary size.

```
def abc():
    c = 5
    print(c)
```

Figure 2.7: Example Python method before being compiled

and B's type do not differ, there is no way of knowing what value PyObject A + PyObject B will produce until after the operation has been evaluated.

Index	Value	SP
0	(PyObject A + PyObject B)	x

Table 2.3: After operation '+'

For example, assuming that both A and B are integers 5 and 6 respectively, they would produce an integer $5 + 6 = 11$ on the stack, similarly if both are strings "A" and "B", they would produce "AB" instead. The same operator is used, but different results are given depending on the objects' types. The VM provides an interface between operators and objects, allowing the underlying functionality of operators, for an object, to be implemented. These methods defines operators' behavior, where the object itself together with an argument, returns a resulting object. For adding objects together, the method *object.__add__(self, arg)* is called. This is referred to as emulating objects, and methods used when emulating are commonly called "magic methods". In the example above, "+" adds two integer objects together, creating an integer object where the sum is stored, whereas strings concatenate into a new string container object, *object.__add__(5, 6)* and *object.__add__("A", "B")*, respectively [12].

2.4.1 PVM Instructions

When a script is being executed, it is the PVM that executes Python bytecode instructions. An instruction is divided into two bytes: the first byte contains the opcode (operation), and the second byte contains an argument (if required by the operation).

By forcing Python to compile a script, and then disassembling it, the internal PVM instructions can be seen. An example of this can be seen in the figure and table below.

After the code has been compiled, these are the instructions that the PVM executes when running the script:

(1)	(2)	(3)	(4)	(5)	(6)	(7)
2			0	LOAD_CONST	1	(5)
			2	STORE_FAST	0	(c)
3			4	LOAD_FAST	0	(c)
			6	PRINT_ITEM		
			8	PRINT_NEWLINE		
			10	LOAD_CONST	0	(None)
			12	RETURN_VALUE		

Table 2.4: Generated PVM instructions

Where the following opcodes are as follows:

Opcode	Effect
LOAD_CONST 1	Pushes 5 onto the stack. Index 1 in local constant storage
LOAD_CONST 0	Pushes none onto the stack
STORE_FAST 0	Stores ToS (c) into local variable storage.
LOAD_FAST 0	Pushes a local variable (reference) onto the stack.
PRINT_ITEM	Prints ToS to sys.stdout. One instruction per item in *print* statement.
PRINT_NEWLINE	Prints a new line to sys.stdout. Generated as the last operation of a *print* statement
RETURN_VALUE	Returns with ToS to the caller of the function.

Table 2.5: Explanation of generated PVM instructions

An explanation of the different columns found in the previous example:

1. The corresponding **line number** in the source code.
2. (optional) **Current instruction** executed.
3. Label which denotes a possible JUMP **from an earlier instruction** to this one.
4. The **address in the bytecode** that corresponds to the byte index (Python has 2 bytes for each instruction)
5. **Opcode/opname**.
6. **Argument**, if any, that is used internally by Python. *An argument is two bytes, with the more significant byte last*
7. **Human-friendly interpretation**.

2.5 Arduino

Arduino is an open-source platform used for building electronics projects. It consists of both a programmable circuit board (the microcontroller) and an Integrated Development Environment (IDE), which is a text editor with tools commonly used for developing and a compiling toolchain that is used to program the device. Arduino code is written in C/C++ using the Arduino environment and library specifically targeted for Arduino devices [13]. Support for standard C-libraries comes from the AVR toolchain [14] whereas there is no official support for C++ standard libraries.

2.5.1 Arduino Uno

Arduino Uno is a device in the USB-based series, that can be programmed via USB without external tools. It also features built-in support for serial communications which can be directly connected to a computer via USB.

Specifiations:

- CPU: ATmega328P (16MHz)
- Memory:
  - 32kB flash (program space)
  - 2kB SRAM

– 1kB EEPROM

Flash memory and EEPROM is non-volatile, and data stored here will not disappear when turning off the power. SRAM on the other hand is volatile and any data stored here will be lost when the power is cycled. When a program (sketch) is transferred to the Arduino it is stored in flash memory. When running a program, the variables and other types of runtime data is stored in SRAM. When dealing with data that takes up more memory, like long string arrays or big data structures, there is a risk that the device may run out of SRAM [15]. If the data is not to be modified, it can be stored in flash by using the *PROGMEM* keyword.[16]

3 Method

3.1 Software

Tools that was used to create the compiler and virtual machine are covered in this section. Additionally, descriptions of how the tools were used and the essential commands for these tools are detailed.

3.1.1 ANTLR

ANTLR (ANother Tool for Language Recognition) is a parser generator for analyzing, executing, or translating files. It is used to build languages, tools, and frameworks. From a grammar, ANTLR generates a parser that can build and traverse parse trees.[17] The tool itself is divided into two parts: the generated lexer and parser, and the runtime which is needed to run them. Since the ANTLR tool is written in Java, it can be utilized on many different systems. Descriptions of languages are stored in grammar files called *.gX*, where *X* specifies the version of ANTLR which should be used when reading that file. Since ANTLR version 4 was used, a *.g4* grammar was used.

There are some alternatives that were considered when picking a lexer and parser, including the combination of the lexer tool Flex and the parser tool Bison[18]. ANTLR was picked because it already contains both the lexer and the parser, meaning there was no need to combine different tools to get the parsing done. Another reason was that multiple grammars for various Python versions were found for ANTLR while researching what possibilities regarding parsing were available.

An already existing grammar for Python was found in one of ANTLR's GitHub repositories [19]. Although many different versions of Python were covered, a grammar which only covered a subset of Python was used due to limitations with the memory size. With the ANTLR tool, a lexer and parser for Python was generated with the command:

```
antlr4 Python3.g4
```

The lexer and parser were generated as *.java* files. In order to use the lexer and parser, the java files were compiled using the command:

```
javac Python3*.java
```

The compiled files include all the constructs and rules defined in the grammar file, and could be used to generate a parse tree and an AST.

In the ANTLR tool set, a program used to generate an image of a parse tree from input called *grun*, was included. An example of how to run this tool, with a chosen grammar from a specific file, can be seen here:

```
grun Python3 file_input -gui -tokens test.py
```

Explanation of the arguments used in the command:

- *grun* - the tool itself
- *Python3* - selected grammar file
- *file_ input* - parse rule inside grammar file for how to handle inputs from a file
- *-gui* - enable visual representation of a tree
- *-tokens* - selects what will be included in a leaf
- *test.py* - the Python file that is read

An example of an image of a parse tree provided by this tool can be seen in figure 2.2, in subsection 2.2.1.

3.2 The Visitor Design Pattern

The visitor design pattern is used to traverse the different syntax trees during the language processing phase, as well as to handle magic methods in the virtual machine. A visitor is an object, implementing an interface specifying what classes it may visit, where functionality for other classes is stored. It is convenient for a class to accept a visitor, because it separates cross-class functionality between different classes from itself, resulting in high cohesion and low coupling between classes, which are desirable features in structured design [20]. In this section, the visitor design pattern will be described using language procession in mind, but it works in a similar way when handling magic methods.

First, an interface is needed for the base visitor, which contains a visit method for all classes that requires it, i.e. all statements and expression. This visitor is produced from the parser of ANTLR, and all it does at this point is to visit each child in each node. To build the AST, a new visitor, *BuildASTVisitor*, is used that inherits the visitor from the parser, and overrides the visitors for each statement and expression. The new visitor methods are what ultimately creates the AST.

3.3 Development Environment

The work environment for both the compiler and virtual machine was on a UNIX system, running Ubuntu, utilizing Makefiles, shell commands and scripts to compile and test. Makefiles are used in conjunction with the *make* command, and together make a powerful tool for building projects by chaining commands, eliminating the need to manually input complex command lines one after another[21]. For Java development, openjdk-11 was used to compile Java source files into runnable programs. Example of usage:

```
javac file.java -o outputfolder/
```

For C++ development, gcc and avr-gcc were used for PC and Arduino, respectively. Command lines, similar to javac, were issued to compile for respective platform. All the compilation methods listed were not used directly, but instead were placed into Makefiles making the repeated process of compiling simpler.

Arduino's own IDE was not used since it complicated the development process. Instead, an already existing Makefile [22] was configured to build and program the device, using the avr

toolchain [23]. Makefiles for both platforms made the building of applications simpler and more uniform.

3.4 Testing

To test the functionality of SVM, that is, that it produced the same results as the PVM, a simple test bench was developed using bash script, in order to compare the outputs of the two VMs. First, the test bench compiles SVM and the bytecode compiler. Then, using multiple test scripts written in Python, the test bench produces a bytecode file using the compiler and copies the file to the SVM directory, where it is then run. The output of SVM is redirected to a file called **a**, ready to be compared.

The PVM is then run on the same Python script while also redirecting the output to another file, called **b**. The two files a and b are then compared using the *diff* command on UNIX systems, which will either output nothing if the two files are identical or point out where and what differences there are between them. If differences are found, a new file with the same name as the script with the *.diff* extension will be created. With files testing all implemented functionality, it is simple to test all aspects when implementing new features while ensuring that previous implementations work.

A similar test environment could be used for testing the VM on Arduino by developing a tool that reads data from a serial port, enabling the output to be compared. The serial tool can be found in appendix A.3. Additionally, to examine the size of the compiled VM, the tool *avr-size*, provided by the avr toolchain, was used.

3.5 Implementing New Functionality

When adding new constructs and functionality to NanoPython, the following steps were done to make sure that everything worked as intended:

1. Add the necessary grammar to the source grammar file (ANTLR).
2. Create classes and visitors for the constructs necessary for the lexing and parsing phases (ANTLR/Java).
3. Create classes and visitors for the constructs that are necessary for generating the correct bytecode (Java).
4. Add necessary code to VM in order to handle the bytecode of the new construct properly (C/C++).
5. Test the functionality of the new construct as described in section 3.4 by adding new Python source files using the same construct (Python and Bash).

4 System Design

4.1 Lexer and Parser

The lexer and parser is generated using ANTLR4. Running the commands mentioned in 3.1.1, ANTLR4 generates basic visitor classes for lexing and parsing that are used to traverse code based on a grammar file. Using the generated classes, the lexer converts the source file into a token stream, which is passed into a parser class. The parser in turn has a method, *file_input()*, which returns the parse tree of the inputted token stream. Given that there is no error during parsing, the AST is constructed by setting up a visitor for the parse tree.

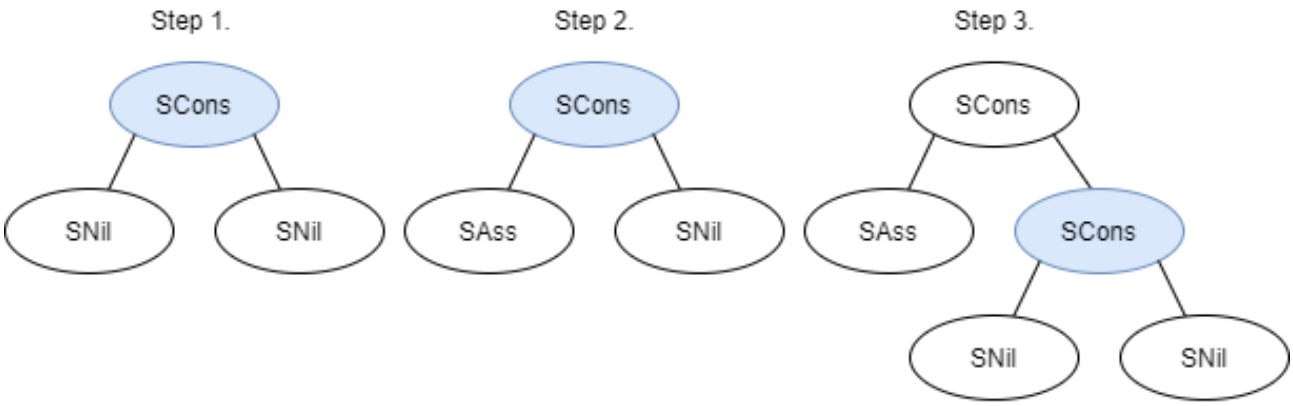


Figure 4.1: The three main cycling steps of creating the AST. The node reference as *current* is marked as blue. Note that the leaves of *SAss*, which should be two expressions, have been omitted from the figure.

The visitor creates the AST by first making the root of the tree an *SCons* node, which in turn has two additional nodes as leaves. These leaves are initiated as *SNil* nodes, meaning that they are just there as placeholder nodes. Additionally, a reference to the current position in the AST, simply called *current*, is kept in order to expand the tree. Then, the first statement in the source code is found in the parse tree, and a statement node is constructed depending on what statement it is. The statement node is added as the left leaf of the current node in the AST. After the statement has been processed and added to the AST, *current* is adjusted by making the right child a new *SCons* and setting *current* to that new *SCons*. From here, the process is repeated until the whole parse tree has been traversed. An example of how this works is illustrated in figure 4.1.

When visiting a statement node, such as if-else statements or while loop statements, blocks of code are encountered. For instance, a while loop may look like the following piece of code:

```
a = 0
while a < 3:
    print(a)
    a += 1
```

Figure 4.2: A simple while loop in Python.

Code blocks are special in that they are a special construct that keeps a list of multiple statements, in this case printing *a* and incrementing *a*. In order for the AST to not treat the two statements as standalone nodes, but as part of the while loop, they are added to the list of a node called *SBlock*. When the *SBlock* node has been visited, it is set as the statement of the while loop, and the construction of the AST proceeds.

4.2 Typechecker

As mentioned in section 2.1, Python is a dynamically typed language. This is important to keep in mind, because it means that the type of typechecking used is **dynamic**. In other words, the parser generates an AST and the compilation chain directly proceeds to the code generation phase. Any and all typechecking is instead handled by the virtual machine, which is described in section 4.4.

4.3 Compiler

The following section covers the process of using the AST generated from the parser to generate all bytecode, including information about the source file and its contents, needed for the interpreter to run the source file. The binary file format of the compiled filed bytecode is also described in detail.

4.3.1 The Process of Compilation

The compilation starts by setting up an object called a *PyCodeObject*, which is designed to mirror the object that the VM uses during interpretation. Information that is immediately stored in this object are as follows:

- A magic number, which can be seen as the version of the compiler.
- A stub of the checksum, which is used to verify that there is no corruption during transmission of the generated bytecode.
- The moddate, or modification date, which tells the current time of the compilation given in the UNIX timestamp format, which counts the number of seconds passed since January 1st in 1970[24].

After the initial three parameters are set, the AST given by the parser is traversed, which fills various data structures with data required to produce bytecode for the given AST. The data is used to fill the remaining properties of the *PyCodeObject*, which includes:

- All bytecode instructions, complete with parameters necessary for them to work correctly.
- Information about constants that are used and generated during the compilation.
- Variables used and the values connected to them.

At the end of the compilation, the *PyCodeObject* is converted to an array of characters, which is used to get the hexadecimal representation of the bytecode. The array is then written bytewise to a file using a *DataOutputStream* in conjunction with a *FileOutputStream*. When the final byte has been written, the compiler is done, and the compilation chain is complete. The format of the bytecode is described in subsection 4.3.3.

4.3.2 Traversing the AST

Similarly to how the parser traversed the parse tree, the compiler uses the visitor pattern to visit every node in the AST in succession. When nodes in the AST are reached, the compiler creates new objects that represent the bytecode instructions for their action. For instance, a statement expression (SExp) will compile an expression and put the value of the compiled expression on the stack. Since in the case of a statement expression the statement does not need the value on the stack, the value is popped from the stack after the compilation. Had the statement instead been the print statement, then the expression's value would be used to be printed. The code for compiling an expression statement node is seen in figure 4.3.

```
public Void visit(SExp v, Void arg) {
    compile(v.exp, true);
    stack.pop();
    return null;
}
```

Figure 4.3: The code for an expression statement.

When an expression is evaluated, not only will a value be pushed onto the stack; most of the time, some code will be emitted. That is, bytecode that represents the code in a script file is saved into a list that will be used to output all bytecode into a file later on. Using an integer expression as an example, the value of that integer is first saved into a list of constants. Then, the bytecode that represents pushing that number onto the stack is emitted, together with information about its location in the code. Continuing, the integer value is pushed onto the internal stack used by the compiler. This process and the code for it is illustrated in figure 4.4, where the bytecode instruction *LOAD_CONST* is emitted.

```
public Void visit(EInt v, Boolean arg) {
    // Add to consts from file
    addExConst(v.val);
    // Emit bytecode and increase nextLocal accordingly
    emit(new Load_const(nextLocal, consts_existing.indexOf(v.val), v.val.toString()));
    nextLocal += 2;

    stack.push(v.val);
    return null;
}
```

Figure 4.4: The code for an integer expression.

Because the generated bytecode references the constants by position in a data structure instead of directly using their values, all constants have to be saved in a specific manner to a list. There are essentially two parts to the list of constants: one part containing the constants found directly in the source file, and one part containing constants that are a direct result of operations, e.g. additions. Using the following code snippet:

```
a = 1 + 2
```

the first part of the constant list would be filled with 1 and 2, because they are two constants found directly in the source code. The second part of the constant list would be filled with 3, since $1 + 2 = 3$.

4.3.3 Compiled Binary Format

Apart from the magic number, moddate and checksum, there are other attributes required for the format to work. These are:

- The stack size.
- Flags that are set during traversal of the AST.
- The number of constants used, and their types.
- The number of variables used, and their lengths.
- The file name of the source script, and its length.

All this information is compiled and stored in a binary file format called *.snok*, inspired by the normal *.pyc* file format, which can be separated into four parts: header, code, variables and footer, which are all illustrated in figure 4.6 and figure 4.7.

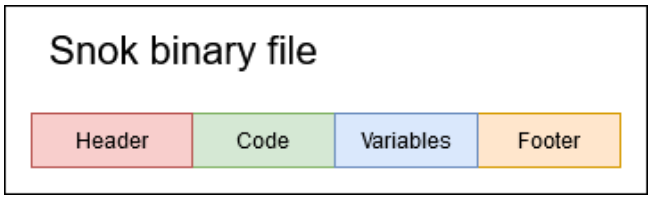


Figure 4.5: Snok binary file format

The magic number is a hexadecimal number and, as mentioned earlier, is used to signify which version of a compiler is used. In the case of NanoPython, the arbitrary number used is BAADC0FE. The stack size is determined by the largest size of the stack needed at any time during compilation. For instance, when adding two or more numbers, the stack size will be two as the two numbers are on the stack and are then added to make a new number.

Figure 4.6: Header and code

Information about constants and variables are set and kept in lists during compilation time, and the file name of the script is sent as a parameter to the compiler. The checksum is computed by using the sum of the first byte of the magic number, the bytecode bytecount, the constants count, the names count and filename character count. For instance, for a compiled file with the hexadecimal checksum C8, it could be derived by $BA + 04 + 02 + 01 + 07 = C8$.

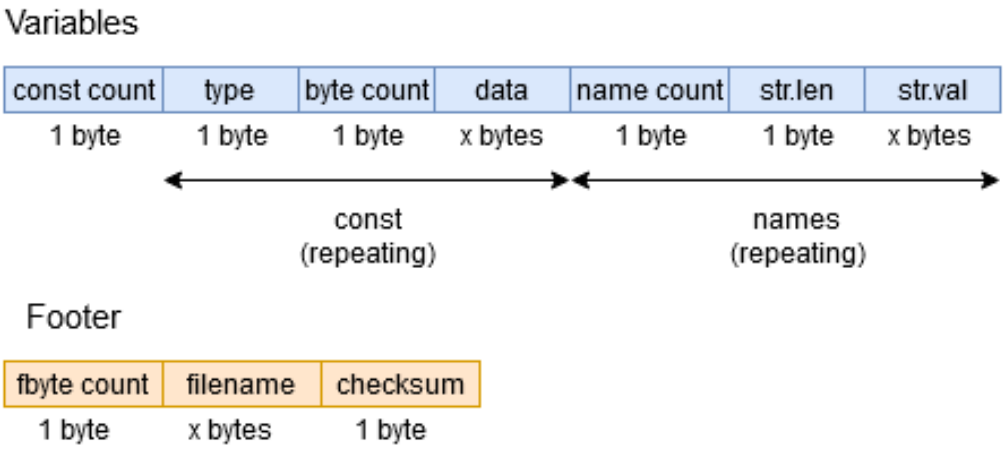


Figure 4.7: Variables and footer

4.4 Virtual machine

This section describes the design of the virtual machine interpreting the *.snok* files, *Snok Virtual Machine (SVM)*, in terms of a VM containing a loader, an execution engine and a memory manager. Since both PC and Arduino are targeted platforms, where both are very different in both architecture and memory size, the script loader differs in design and implementation.

4.4.1 Loader and Dynamic Linker

Both the PC and Arduino target uses the same core loader and only differs in the initial loading phase, when the binary data from the file is transferred to SVM. PC loads data directly from a file but for the Arduino, the VM can only load a script by using a tool developed for PC, that is able to transfer a compiled script serially.

When running the VM on PC, a script is loaded directly from a *snok* file, and since it only runs one script at a time, there is no need to add or remove previously loaded scripts. However, when programmed onto the Arduino, the VM is always running when the device is powered and is therefore designed to be able to load several scripts in succession. If the binary data from the file is valid, the VM creates a new frame in both cases and, if needed, replaces the old one. The logic of how an Arduino loads data is described using a flowchart in figure 4.8.

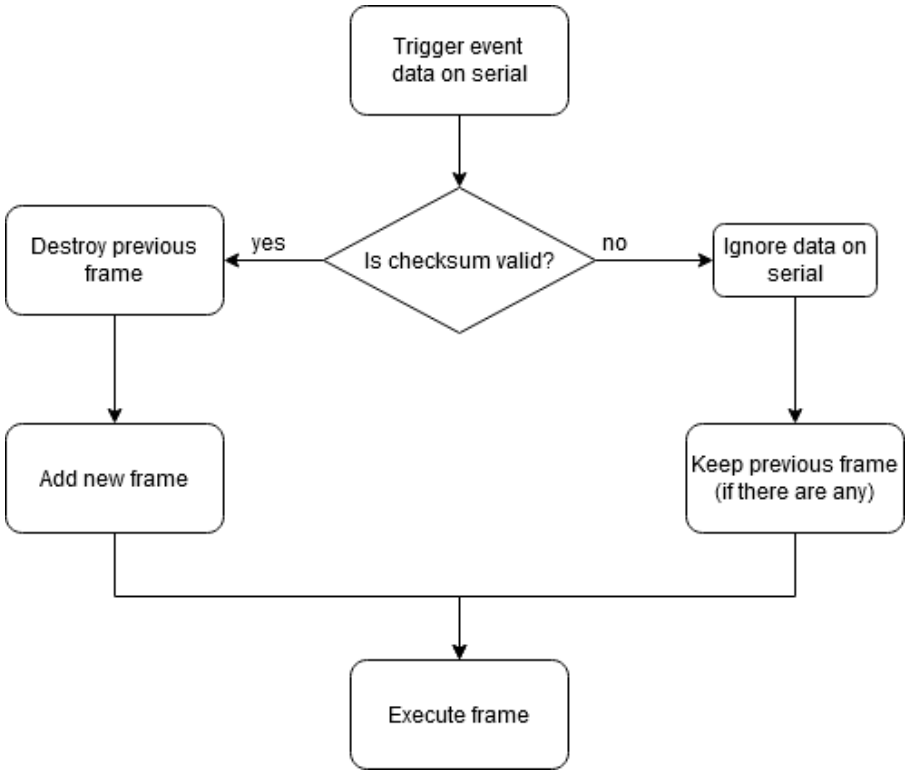


Figure 4.8: Arduino initial loading phase

The end goal of the SVM loader is to create an object (PyCodeObject), that stores code and variables derived from binary data, to be placed inside a frame (PyFrame), making the loaded code runnable. While creating objects to store the data in, error checking is performed to prevent the VM from crashing e.g. from loading uninitialized data. Initially, the first four bytes of data are compared to the magic hexadecimal number *BAADC0FE*, which is the most basic check to ensure that a file with correct format has been loaded. Following the header data, bytecode consisting of two bytes, operand and argument, is loaded and stored in an object (PyByteCode). PyObjects with valid types are then created in the following way:

- integer requires two bytes, type and value [1]
- boolean requires two bytes, type and value (true/false).
- string requires one + x bytes, type and x bytes of characters, respectively.

When everything has been loaded correctly, the checksum is calculated and compared to the checksum bytes received. If the checksum are equal, a PyCodeObject is created and subsequently stored in a PyFrame to be evaluated.

[1] an integer is only one byte in SVM due to memory limitations

4.4.2 Execution Engine

The primary function of the execution engine is the evaluation cycle, where a frame is evaluated and bytecode interpreted. Evaluation properties required to evaluate a frame are set when first entering the evaluation function, e.g. program counter (PC) is set to 0 and stack pointer (SP) is set to -1 (stack is empty). PC's value is set to point to the current bytecode instruction (opcode) to be executed and SP the top PyObject on the stack. For every new loop a new opcode and an optional argument are fetched from an array of PyByteCode objects, previously obtained from the loader, using the current PC value. This is referred to as the fetch stage in a CPU[25].

After the opcode has been fetched, the instruction is executed, and evaluation properties are adjusted accordingly. These properties can be observed when running the VM, for example a script is loaded, containing the statement: $a = 1 + 2$. The changes in PC and SP, as well as the result on the stack, can be seen in figure 4.9.

```
============= ITERATION 0  =================
PC: 0   SP: -1  OPCODE: 64      OPERAND: 1
============================================
Printing stack:
--+---------
 STACK EMPTY
--+---------
Decode/Execute instruction:
LOAD_CONST

============= ITERATION 1  =================
PC: 2   SP: 0   OPCODE: 64      OPERAND: 2
============================================
Printing stack:
--+---------
0 | 1
--+---------
Decode/Execute instruction:
LOAD_CONST
```

```
============= ITERATION 2  =================
PC: 4   SP: 1   OPCODE: 17
============================================
Printing stack:
--+---------
1 | 2
--+---------
0 | 1
--+---------
Decode/Execute instruction:
BINARY_ADD

============= ITERATION 3  =================
PC: 5   SP: 0   OPCODE: 7d      OPERAND: 0
============================================
Printing stack:
--+---------
0 | 3
--+---------
Decode/Execute instruction:
STORE_FAST
```

Figure 4.9: Output from evaluating 'a = 1 + 2'

PC increments by two for iteration 0-1 but only by one in iteration 2 because BINARY_ADD requires no argument. Note that in **ITERATION 2** the top value of the stack *PyInt(2)* is added to the bottom value *PyInt(1)* creating a new object *PyInt(3)*, that can be seen on the stack in **ITERATION 3**.

A PyObject can perfom operations on other PyObjects utilizing *magic methods*, where the magic method is selected when decoding the opcode. In the specific example above, *BINARY_ADD* uses the magic method *__iadd__* with the two top objects on the stack, subsequently decreasing the stack pointer.

Magic methods are implemented using a visitor pattern, where every built-in type (PyObject) has its own visitor function. Every operation is performed based on the visited object's type, argument object's type and the magic method in use e.g. *object.__iadd__(self, int) = int*. All arithmetic and logical operators are implemented like this.

4.4.3 Memory Manager

Where other VMs contain memory managers for example JVM's garbage collector or PVM's reference count, SVM do not have any explicit memory manager other than the allocate/deallocate functions that are used when a value is created or removed from the evaluation stack.

4.5 CLI

A CLI is designed, using Apache commons cli, to use the compiler and serial tool written in Java together with the virtual machine written in C++. A CLI is needed to make the process of compiling and running simpler. Since it is also written in Java, the compiler and serial tool sources can be included directly. To include the virtual machine, a new process is created in the CLI. Manuals on how to use the CLI are included in appendix A.2.

5 | Result

5.1 Compiler

The compiler can take a Python source file, and generate bytecode which the VM can in turn interpret, producing results that are very similar to the Python interpreter. The following constructs can be used to produce bytecode:

Construct	Type	Example
SExp	Statement	1 + 4
SAss	Statement	x = 4
SPrint	Statement	print("hi")
SWhile	Statement	while ECond: SBlock
SBlock	Statement	[stms]

SIfElse	Statement	if ECond:
		  SBlock
		elif ECond:
		  SBlock
		else:
		  SBlock

EId	Expression	x
EString	Expression	"hi"
EInt	Expression	42
EAdd	Expression	x + 4 + 2
ESub	Expression	x - 4 - 2
EDiv	Expression	x / 4 / 2
EMul	Expression	x * 4 * 2
ETrue	Expression	print(True)
EFalse	Expression	if False: ...
EUnary	Expression	-4, 19, -+-4
ECond	Expression	1 > 0, True == False

Table 5.1: Python constructs supported in compiler and VM.

The original plan was to have a separate assembler apart from the compiler, having the compiler produce assembly code which the assembler in turn would use to produce bytecode. Instead, the resulting compiler does both tasks. While the assembly code, or rather the bytecode instructions, can be seen using a verbose option during compilation, the only code that is written to a file is the bytecode. Because most of the focus has been given on producing bytecode instead of the bytecode instructions, details such as the line number does not work

properly in the bytecode instructions. Due to the structure of the compiler, it is an easy task to modify the program to write the bytecode instructions instead e.g print LOAD_CONST instead of 0x64.

In order to use the compiler, it can be done directly from the compiler by running the command:

```
java SnokC <x.py> <options>
```

or, by using the CLI:

```
java snok compile <x.py> <options>
```

both of which generate an *x.snok* binary file.

5.2 Virtual machine

The virtual machine can interpret *.snok* binary files both when run on PC and Arduino, supporting all the basic Python constructs mentioned in previous section. It can be executed in multiple ways using a command prompt. For PC it can be run directly from executable:

```
./snokvm <x.snok> <options>
```

or using the CLI:

```
java snok run <x.snok/x.py> <options>
```

5.2.1 VM on Arduino

To be able to run the VM on Arduino, it must first be programmed onto the device:

```
make program
```

In the Makefile, a serial port connected to the Arduino must be selected for this to work. If the VM is successfully programmed, a script can be sent over serially with the same port, using the CLI:

```
java snok send <x.snok/x.py> <options>
```

The CLI manual can be seen in appendix A.2

5.2.2 Usage Example

When executing a script (a) using the VM, with the commandline below, output (b) is printed in a terminal.

```
java snok run loop.py
```

```
x = 2
a = True
while a == True:
  if x <= 0:
    a = False
  print(x)
  x = x - 1
print("done")
```

```
2
1
0
done
```

Figure 5.1: (a) Python code

Figure 5.2: (b) Terminal output

5.2.3 SVM Compared to PVM

There are a multitude of constructs that are not implemented, but of those that are, the similarities and differences in implementation between SVM and PVM can be seen in Table 5.2.

Construct	PVM	SVM
PyObject	Abstract data structure	Abstract class
String	Unicode object (char array)	PyStr (char array)
Integer	PyLong, integer of arbitrary size	PyInt, 1 byte size
Boolean	Subclass to integers (Py_True/Py_False)	Subclass to integer PyBool (true/false)

Table 5.2: Differences between PVM and SVM

5.2.4 Memory Usage

One of this project's limitation was the memory size of our embedded device, that was limiting how much of Python could be implemented. It was not possible to determine this beforehand, thus certain Python constructs were selected that were considered important for the language. In figure 5.3 the memory size of SVM, after compilation, can be seen. *Program* refers to the memory usage in the flash memory, whereas *Data* refers to the static SRAM usage.

```
AVR Memory Usage
----------------
Device: atmega328p

Program:   22704 bytes (69.3% Full)
(.text + .data + .bootloader)

Data:        352 bytes (17.2% Full)
(.data + .bss + .noinit)
```

Figure 5.3: Size of VM

Variables and objects that are static are not dynamically allocated during runtime and are instead assigned prior to compilation. Dynamically allocated memory usage increases during runtime. Considering that 69% of the flash memory is utilized, additional Python constructs can be implemented.

5.3 Problems

In this section, potential problems in the system design are presented and later discussed in section 6.1.

5.3.1 Testing Bad Files

The current test bench used to verify that the result of running script between PVM and SVM are identical. In other words, only scripts that are programmed to work are tested, and not scripts that are intended to fail. There are as of now no way to use tests to check for bad usage

of the compiler, and faulty code may not be detected. Since the goal for this project was to run Python scripts on an Arduino, focus was given to test scripts that work on Python.

5.3.2 Dynamic Memory

When running the VM on Arduino for an extended period of time, the device may eventually run out of memory, on the heap (SRAM), as a consequence of allocating and freeing memory (memory fragmentation)[26]. At what given time this will occur is entirely based on a loaded file's extent in terms of runtime memory usage.

Additionally, when memory is not freed after data has been allocated, and the reference to that data stored in memory is lost, a memory leak occurs. Using the tool *Valgrind*, a minor memory leak was detected in the loader, which also may eventually lead to the device running out of memory after subsequent runs of loading files.

Due to the nature of our testing, where small scripts run only a short amount of time, none of the two cases has occurred to crash the VM.

6 Discussion

6.1 Problems

Problems and their potential solutions regarding our system design are discussed in this section.

6.1.1 Compiler

The objective was to make the bytecode instructions produced from the compiler to be as much like the instructions from Python as possible. Using the *dis* module from Python, used to disassemble a piece of code, it's possible to analyze exactly what instructions are derived from the given code. This was used together with Python's documentation where all bytecode instructions are described in detail [27]. There were some problems with how Python generates the bytecode instructions, which can be seen by using the following line of code:

```
a = 1 + 2 + 3
```

The bytecode instructions that were expected to be produced were

LOAD_CONST	1
LOAD_CONST	2
BINARY_ADD	
LOAD_CONST	3
BINARY_ADD	
STORE_FAST	0

Table 6.1: Unoptimized bytecode generation

which first adds $1+2$, then adds $(1+2)+3$ and then stores it to variable *a*. Instead, what was actually produced was

LOAD_CONST	6
STORE_FAST	0

Table 6.2: Optimized bytecode generation

which puts the result of the addition to the stack, and then stores it to variable *a*. This implies that there is some optimization going on behind the curtains. It was attempted to solve it in the same way as Python does, but a simpler solution which produces the same non-optimized code as in the first example was decided upon instead, to not stall the development unnecessarily.

There were also initial problems with how the binary format would be structured, because it was difficult to interpret the contents of *.pyc* files. Fortunately, a short script used to disassemble the contents of such files, making them human-readable, were found, along with explanations of how the tools used in the script work[28]. Using this information, the current structure of *.snok* files, which is heavily inspired by the structure of *.pyc* files, was constructed.

6.1.2 Arduino and PC Incompatibilities

It was thought to be more convenient to write code for the PC target and then translate it to the Arduino, using the same source files but with different SVM loaders. The reasoning behind this was that it was easier to test and verify implementations, for both the compiler and the SVM, on PC. Since Arduino does not support the C++ standard library, the code for SVM is an amalgamation of both C-library functions and OOP support from the C++ language, which is considered to be bad practice if the SVM would be designed only for PC usage.

One instance of such an amalgamation is when PyStr, the string type in Python, was implemented. The string library from Arduino could not be used, since it is not supported on PC, and the string type included in C++ standard library could not be used either, because it is not supported by Arduino. The problem was solved using character arrays, also known as "C-style" strings, which is supported on both platforms. Problems such as this one occur when trying to support multiple targets using the same core. Separating the code per target could be another possible solution to the issue, meaning that each target code uses the target's best way of implementation. Take the PyStr for example, it could be more convenient to use Arduino's string type *<WString>* for Arduino and use C++ *<string>* for PC contra using a char array.

6.1.3 Testing Bad Files

A possible solution that would be simple to implement is expand the test bench with a part that runs intentionally bad programs. Because the compiler stops the compilation process by throwing exceptions when needed, bad programs can be caught by printing lines that are recognized by the test bench. By comparing the number of passed tests with the number of bad programs tested with, the test bench can signal that not all tests passed. With this information, the functionality can be tweaked in the compilation chain where the exception was thrown.

6.1.4 Dynamic Memory

Since the fragmentation comes from allocating memory for different sized data an extended amount of time, one could fix this issue by allowing the memory manager to only create equally sized data to be allocated. This may possibly also solve the memory leaks, if all references are monitored by the memory manager.

Another possible way of solving these issues, that arises when using dynamic memory allocation, is to force the VM to only use static memory allocation. Instead of allocating new memory during runtime, static arrays, fixed in size, are used throughout the duration of the program. This was attempted, when the memory leaks were discovered, but could not be implemented due limited knowledge of scoping rules in C++.

6.1.5 Consequences of Using Dynamic Memory in Safety Critical Systems

The use of dynamic memory allocation and the common issues that come with it prevents a potential application use for the VM in critical real-time systems. Dynamic memory allocation in safety critical systems is unethical because there is no guarantee that there will be enough memory to be allocated during runtime. If used in a situation where people are at risk of injury, such as during a car ride, failing to allocate memory may cause the system to malfunction. A malfunction in a safety critical system may in turn cause injury or even death to not only the passengers, but to the surrounding. Since the VM is not designed to be used in a safety critical system, this is not much of a problem.

6.2 Trading Power Consumption for Portability

Assuming a language can both be compiled to bytecode and machine code, subsequently being run by a VM and CPU, respectively. When comparing the power consumption of the CPU running a VM to the CPU running native code, the CPU running the VM requires more instructions per task since it has to run the VM running the code compared to running the code directly. More instructions per task means an increase in power consumption. According to a master thesis on the subject, "A busy virtualized database server can consume 30% more energy than its nonvirtualized counterparts"[29]. Therefore, compiling directly to a target CPU would be more power efficient i.e. a better solution for the environment.

If Python would be compiled into machine code rather than bytecode, portability of the compiler would suffer. Additional compilers need to be developed to support more platforms. One could argue that if Python were translated to another high-level language like C, the portability would not suffer as great, since there are C compilers targeting many different computers, but instead, the process of deploying scripts would be more complex. Instead of transferring a script to be executed, the device itself must be reprogrammed each and every time. It is a trade-off between power efficiency and portability.

6.3 Discussing the Results

Much of the time was spent on experimenting and researching how each of the components of the project were supposed to work, while at the same time applying that knowledge to this thesis. This led to a somewhat slow and unstructured start to the project, but in time the process improved and more structure was established.

What are the pros and cons of using Python?
Some advantages of using Python is its relatively simple syntax and its built-in modules for disassembling code to bytecode instructions, which simplifies the process of generating custom bytecode for the language. Python also has detailed documentation which significantly helps in understand how the language works behind the scenes. The disadvantages of using Python is the amount of constructs that have been added over the many years it has been in development which makes for a heavy task if the complete grammar is to be supported.

Is it feasible to use Python as the chosen language on the Arduino?
Yes.

How much of the Python language can we implement on the Arduino (subset of Python)?

It is hard to approximate the number of Python constructs that may fit on the Arduino device. What units should we choose to represent a percentage of the language implemented? Although additional constructs can be implemented, it is not far-fetched to assume that adding support for class definitions is not possible due to the memory limitation.

6.4 Further Development and Future Applications

Currently, the number of supported constructs are not sufficient for the VM to be of any practical use. Useful constructs, like methods and classes, both being a necessity in an object-oriented language, are not implemented. Also, due to the small integer size (1 byte) and no support for float numbers, bigger and precise computations can not be performed. These implementation-specific details, in the compiler and VM, can both be implemented and fixed in the future. By adding hardware support, e.g. as a built-in Python library, allows the Arduino to use its digital and analog pins, supporting Arduino hats or other peripherals, which increases the application scope of the VM.

In conclusion, the VM is not suitable for unit tests in vehicles yet, but after further development and adding support for CAN communication, NanoPython might be a possible way of rapidly performing tests. Given the simple syntax of Python, and its built-in disassembly method, generating bytecode similar to Python's was made easier. However, due to the many constructs included in Python, making rules for parsing all of it is a daunting and time-consuming task. Due to the sheer size of the complete grammar, it is not possible to implement Python as a whole due to the memory constraint; all of the simple constructs already implemented utilizes 69% of the flash memory.

Bibliography

[1] *MicroPython - Python for microcontrollers.* [Online]. Available: https://micropython.org/ (visited on 2020-06-10).

[2] Python, *What is Python? Executive Summary | Python.org.* [Online]. Available: https://www.python.org/doc/essays/blurb/ (visited on 2020-05-20).

[3] L. Tratt, "Dynamically Typed Languages," *Advances in Computers*, vol. 77, M. V. Zelkowitz, Ed., pp. 149–184, 2009-07.

[4] A. Ranta, *Implementing Programming Languages - An Introduction to Compilers and Interpreters.* Göteborg, 2012, p. 207, ISBN: 978-1-84890-064-6. [Online]. Available: https://www.grammaticalframework.org/ipl-book/.

[5] *2.6. Statements and Expressions — How to Think like a Computer Scientist: Interactive Edition.* [Online]. Available: https://runestone.academy/runestone/books/published/thinkcspy/SimplePythonData/StatementsandExpressions.html (visited on 2020-05-11).

[6] J. Longley, *Types and Static Type Checking (Introducing Micro-Haskell)*, 2015-10. [Online]. Available: https://www.inf.ed.ac.uk/teaching/courses/inf2a/slides/2015%7B%5C_%7Dinf2a%7B%5C_%7DL12%7B%5C_%7Dslides.pdf (visited on 2020-05-20).

[7] *Python: A Strong, Dynamically Typed Language - OzNetNerd.com.* [Online]. Available: https://oznetnerd.com/2017/05/16/python-strong-dynamically-typed-language/ (visited on 2020-04-01).

[8] *Runtime Type Identification.* [Online]. Available: https://docs.oracle.com/cd/E19957-01/806-3571/RTTI.html (visited on 2020-06-10).

[9] X.-F. Li, *Advanced design and implementation of virtual machine*, ISBN: 9781466582606.

[10] O. Ike-Nwosu, *Read Inside The Python Virtual Machine | Leanpub.* [Online]. Available: https://leanpub.com/insidethepythonvirtualmachine/read (visited on 2020-02-13).

[11] Python, *4. Execution model — Python 3.8.2rc1 documentation.* [Online]. Available: https://docs.python.org/3/reference/executionmodel.html (visited on 2020-02-13).

[12] ——, *3. Data model — Python 3.8.3 documentation.* [Online]. Available: https://docs.python.org/3/reference/datamodel.html (visited on 2020-05-23).

[13] *Sketch build process - Arduino CLI.* [Online]. Available: https://arduino.github.io/arduino-cli/sketch-build-process/ (visited on 2020-04-13).

[14] Microchip, *AVR- and Arm- Toolchains (C Compilers) | Microchip Technology.* [Online]. Available: https://www.microchip.com/mplab/avr-support/avr-and-arm-toolchains-c-compilers (visited on 2020-04-13).

[15] Arduino, *Arduino - Memory*, 2019. [Online]. Available: https://www.arduino.cc/en/Tutorial/Memory%20https://www.arduino.cc/en/tutorial/memory (visited on 2020-03-30).

[16] ——, *Arduino - Reference - Progmem*, 2016. [Online]. Available: https://www.arduino.cc/reference/en/language/variables/utilities/progmem/%20https://www.arduino.cc/en/Reference/HomePage (visited on 2020-03-30).

[17] *ANTLR*. [Online]. Available: https://www.antlr.org/ (visited on 2020-01-30).
[18] *Flex and Bison*. [Online]. Available: https://aquamentus.com/flex%7B%5C_%7Dbison.html (visited on 2020-06-10).
[19] *Grammars-v4/python at master · antlr/grammars-v4*. [Online]. Available: https://github.com/antlr/grammars-v4/tree/master/python (visited on 2020-05-21).
[20] W. P. Stevens, G. J. Myers, and L. L. Constantine, "STRUCTURED DESIGN.," *IBM Systems Journal*, vol. 13, no. 2, pp. 115–139, 1974, ISSN: 00188670. DOI: 10.1147/sj.132.0115.
[21] *Makefile Tutorial - What is a Makefile?* [Online]. Available: http://www.sis.pitt.edu/mbsclass/tutorial/advanced/makefile/whatis.htm (visited on 2020-05-21).
[22] *AVR-makefile-base.mk*. [Online]. Available: https://gist.github.com/entrity/5424505 (visited on 2020-02-20).
[23] Microchip, *AVR- and Arm- Toolchains (C Compilers) | Microchip Technology*. [Online]. Available: https://www.microchip.com/mplab/avr-support/avr-and-arm-toolchains-c-compilers (visited on 2020-05-25).
[24] *Unix Time Stamp - Epoch Converter*. [Online]. Available: https://www.unixtimestamp.com/ (visited on 2020-05-21).
[25] R. Johansson, "8 Styrenheten," in *Grundläggande Datorteknik*, 1:2, Göteborg: Studentlitteratur, 2016, ch. 8, pp. 197–202, ISBN: 978-91-44-07650-8.
[26] Atmel, *1.2.2 Dynamic Memory Requirements*, 2014. [Online]. Available: http://ww1.microchip.com/downloads/en/AppNotes/Atmel-42370-Optimizing-ASF-Code-Size-to-Minimize-Flash-and-RAM-Usage%7B%5C_%7DApplicationNote%7B%5C_%7DAT08569.pdf (visited on 2020-05-20).
[27] Python, *dis — Disassembler for Python bytecode — Python 3.8.3 documentation*. [Online]. Available: https://docs.python.org/3/library/dis.html (visited on 2020-05-20).
[28] N. Batchelder, *Ned Batchelder: The structure of .pyc files*. [Online]. Available: https://nedbatchelder.com/blog/200804/the%7B%5C_%7Dstructure%7B%5C_%7Dof%7B%5C_%7Dpyc%7B%5C_%7Dfiles.html (visited on 2020-02-21).
[29] Y. Bai, "Power Consumption of Virtual Machines in Cloud Computing: Measurement and Enhancement," Master thesis, University of Minnesota, 2016, p. 44.

A | Appendix

A.1 NanoPython Grammar

```
/*
 * parser rules
 */

// startRules:
//single_input: NEWLINE | sSimple | sCompound NEWLINE;
file_input:  (NEWLINE | sStmt)* EOF;

dFunc:      'def' NAME parameters ':' func_body;

parameters: '(' (expr (',' expr)*)? ')';
func_body: sSimple | NEWLINE INDENT sStmt+ DEDENT;

sStmt:      sSimple | sCompound;

sSimple:    sSmall NEWLINE;
sSmall:     sAss | sFlow | sExp | sPrint ;
sAss:       NAME '=' expr;
sFlow:      sBreak | sContinue | sReturn;
sBreak:     'break';
sContinue:  'continue';
sReturn: 'return';
sExp:       expr ;
sPrint: 'print' '(' expr ')';

sCompound: sIfElse | sWhile;
sIfElse: 'if' eCond ':' sBlock ('elif' eCond ':' sBlock)* ('else' ':' sBlock)?;
sWhile: 'while' eCond ':' sBlock;
sBlock: sSimple | NEWLINE INDENT sStmt+ DEDENT;

eCond: (left=expr comp_op right=expr) | BOOLEAN_TRUE | BOOLEAN_FALSE ;

comp_op: op=(OP_LT | OP_GT | OP_EQ | OP_GEQ | OP_LEQ | OP_NEQ);

OP_LT:  '<';
OP_GT:  '>';
OP_EQ:  '==';
```

```
OP_GEQ: '>=';
OP_LEQ: '<=';
OP_NEQ: '!=';

expr:
    ( op=('+' | '-' ) e=expr)           # eUnary
  | left=expr op=('*'|'/') right=expr   # eInfix
  | left=expr op=('+'|'-') right=expr   # eInfix
  | id=NAME parameters                  # eCall
  | id=NAME                             # eId
  | value=NUMBER                        # eNum
  | value=FLOAT                         # eFloat
  | value=STRING                        # eString
  | '(' expr ')'                        # ePar
  | left=expr comp_op right=expr        # eComp
  | value=BOOLEAN_TRUE                  # eTrue
  | value=BOOLEAN_FALSE                 # eFalse
;

OP_ADD: '+';
OP_SUB: '-';
OP_MUL: '*';
OP_DIV: '/';

/*
 * lexer rules
 */

STRING
 : STRING_LITERAL
 ;

NUMBER
 : INTEGER
 ;

INTEGER
 : DECIMAL_INTEGER
 ;

FLOAT
 : NUMBER '.' NUMBER+
 ;

BOOLEAN_TRUE
 : 'True'
 ;

BOOLEAN_FALSE
 : 'False'
```

```
 ;

NEWLINE
 : ( '\r'? '\n' | '\r' | '\f' ) SPACES?
 ;

NAME
 : ID_START ID_CONTINUE*
 ;

STRING_LITERAL
 : '"' .*? '"'
 ;

DECIMAL_INTEGER
 : NON_ZERO_DIGIT DIGIT*
 | '0'+
 ;

OPEN_PAREN : '(';
CLOSE_PAREN : ')';
OPEN_BRACK : '[';
CLOSE_BRACK : ']';
OPEN_BRACE : '{';
CLOSE_BRACE : '}';

SKIP_
 : ( SPACES | COMMENT | LINE_JOINING ) -> skip
 ;

UNKNOWN_CHAR
 : .
 ;


/*
 * fragments
 */

fragment NON_ZERO_DIGIT
 : [1-9]
 ;

fragment DIGIT
 : [0-9]
 ;

fragment SPACES
 : [ \t]+
 ;
```

```
fragment COMMENT
 : '#' ~[\r\n\f]*
 ;

fragment LINE_JOINING
 : '\\' SPACES? ( '\r'? '\n' | '\r' | '\f' )
 ;

fragment ID_START
 : '_'
 | [A-Z]
 | [a-z]
 ;

fragment ID_CONTINUE
 : ID_START
 | [0-9]
 ;
```

A.2 Snok CLI

```
usage: snok <command> <file> [options]
Command:
compile    run compiler, file argument must be .py
run        run VM, file argument must be .snok
send       serially transmit file
version    prints version
snok <command> --help for more information about the different commands
 -h,--help             print this message
    --logfile <file>   use given file for log (verbose sets to enabled)
 -v,--verbose          extra verbose
```

snok compile

```
usage: snok compile <py file> [options]
Compiles .py file into .snok file that may be executed using snokVM
 -h,--help             print this message
    --logfile <file>   use given file for log (verbose sets to enabled)
 -v,--verbose          extra verbose
```

snok run

```
usage: snok run <py/snok file> [options]
If x.py is used as an argument, x.py will first be compiled to x.snok
before being executed by snokVM
 -c,--keep-snok        keep intermediate binary file .snok after
                       compilation
 -h,--help             print this message
    --logfile <file>   use given file for log (verbose sets to enabled)
 -v,--verbose          extra verbose
```

snok send

```
usage: snok send <py/snok file> [options]
If x.py is used as an argument, x.py will first be compiled to x.snok
before being sent.
 -B,--baud <baud rate>     change baudrate (default 115200)
 -h,--help                 print this message
    --logfile <file>       use given file for log (verbose sets to
                           enabled)
 -P,--port <serial port>   choose which port to send binary file to
 -v,--verbose              extra verbose
```

A.3 Serial Tool Used for Testing

```
import com.fazecast.jSerialComm.*;
import java.io.*;
import java.util.ArrayList;
import java.util.Arrays;
import java.nio.charset.StandardCharsets;

class SerialReader extends Thread {
  private SerialPort sp;
  private int cnt = 0;

  public SerialReader(SerialPort sp) {
    this.sp = sp;
  }

  public void run() {
    try {
      while (true) {
        byte[] readBuffer = new byte[sp.bytesAvailable()];
        int numRead = sp.readBytes(readBuffer, readBuffer.length);
        if(numRead > 0 && cnt > 0) {
          System.out.println(new String(readBuffer, "UTF-8"));
          break;
        }
        ++cnt;
      }
    } catch (Exception e) { e.printStackTrace(); }
  }
}

public class Test {
  public static void main(String[] args) throws Exception {
    String filepath = null;
    if(args.length > 0) {
      filepath = args[0];
    }

    final String serialPort = "/dev/ttyS0";
    final int baudrate = 115200;

    OutputStream out;
    InputStream in;
    SerialPort sp;

    sp = SerialPort.getCommPort(serialPort);
    if(!sp.openPort())
      System.out.println("Could not open serial port");

    Thread.sleep(4000);
```

```
    sp.setComPortParameters(baudrate, 8,
        SerialPort.ONE_STOP_BIT, SerialPort.NO_PARITY); //baud, databits, stopbits, par

    sp.setComPortTimeouts(SerialPort.TIMEOUT_READ_SEMI_BLOCKING, 1000, 0);
    out = sp.getOutputStream();

    (new Thread(new SerialReader(sp))).start();

    FileInputStream fs = new FileInputStream(filepath);
    fs.transferTo(out);
    out.flush();
    out.close();
  }
}
```