# JTAC - A JVM Alternative for Simple Java Programs

https://github.com/licarijd/java-arm-compiler

CompSci 4TB3 - McMaster University, Department of Computing and Software

*Justin Licari - 001313781*
*Justin Staples - 001052815*

April 20, 2018

# Contents

# 1 Introduction

Compilers translate higher level code to lower level target code. Ultimately, for an executable program to run on a computer, high level code needs to be translated to a machine (assembly) language that can be understood by the computer hardware. Currently, one of the most common types of processors used for mobile devices is the ARM (Advanced RISC Machine) processor, which uses a reduced ISA (Patterson and Hennessy, n.d.). As well, one of the most popular types of processors used for personal computers is the Intel processor (x86 architecture), which uses a complex ISA (Wirth, 1996).

This paper outlines JTAC (Java to ARM Compiler) - a tool that will be able to generate assembly code for a particular hardware platform (ARM, Intel, etc.), given source code in a high level language. We discuss our prototype, which is designed to transform Java code into ARM assembly code.

Java code is not typically compiled into an assembly language like ARM. The Java compiler actually translates the Java code into Java Bytecode, which is the instruction set understood by the Java Virtual Machine (Sebesta, n.d). Therefore, to run a Java program, you must have the JVM installed on your computer. The motivation behind our project is to offer a different way of compiling and running a Java program (using ARM assembly instead of Bytecode). This will save developers the time and memory overhead of installing and running JVM.

# 2 Technical Overview

## 2.1 Initialization File

JTAC is intended to be run on a UNIX based terminal, such as a Mac/Linux terminal, or even Cygwin on Windows. The initialization file is written in bash, and is a file input mechanism. `JTAC.ini` starts the compilation process by:

a) taking a Java file as command line input: `java_file = $1`

b) passing it to the parser as a command line argument: `python scanner.py $1`

## 2.2 Scanner

The scanner will perform the first stage of lexical analysis. Source text (characters in the Java program file) will be classified as symbols, and converted to a sequence of symbols).

A high-level description of this programs methods and their functionality:

`main():`

a) Removes comments from input.

b) Group source text into words based on whitespace, and construct a list of words.

c) Remove whitespace.

d) Call getSym() on the first word in the list.

`getSym():`

a) First check if all of the words in the list have been checked. If words still need to be checked, check for reserved words by calling `checkForReservedWords()`, then check for reserved symbols by calling `checkForReservedSymbols()`.

b) If all words have been checked, add the 'eof' symbol and terminate the program successfully.

c) Recursively call getSym() until all words have been checked.

`checkForReservedWords():`

a) First, check if the word is a number.

b) If not, check if the word can be classified as a reserved reserved symbol, such as `public`, `class`, `main`, etc.

`checkForReservedSymbols():`

a) Check if the word is a 2-digit relational operator, such as '==' by calling `checkRelationalOp()`.

b) Check for reserved symbols, such as '=', '+', '-', etc.

c) If none can be found, assume the word is an identifier.

`checkForNum():`

a) Checks if the current word is a number using `type(sym) == int`

Now, a low-level explanation of this programs functionality is given in the following sections.

**Imports:**

The `sys` library is imported to allow the program to take command-line arguments.

**Global Variables:**

A list of words are generated according to whitespace. A variable, `currentPosition`, represents the position of a word in this list. When each character in a word is being checked individually, `currentBlockPosition` represents the position of a character in a word.

### 2.2.1 `main()`

The input file is read and split into a list.

```
with open(sys.argv[1], "r") as word_list:
    words = word_list.read().split()
```

Comments and whitespace are removed.

```
for i in range(len(words)-1):

    #Check for and disregard comments
    if (i<len(words)-1):
        if words[i] == '/' and words [i+1] == '/' or words[i] == '/' and words [i+1] == '*':
            commentBlock = True
        if words[i] == '*' and words [i+1] == '/' or words[i] == '\n':
            commentBlock = False
    if commentBlock:
        words[i] = 'comment'

    #Remove whitespace
    if words[i] == ' ' or words[i] == '\t' or words[i] == '\n':
        del words[i]

words = [i for i in words if i != 'comment']
```

`getSym()` is called.

```
getSym(words)
```

### 2.2.2 `getSym()`

First, check for reserved words.

```
checkForReservedWords(words[currentPosition], words)
symList = list(words[currentPosition])
```

If the entire word isn't reserved, check individual characters.

```
checkForReservedSymbols(symList,words)
currentBlockPosition = 0;
```

Call getSym() on the next word.

```
getSym(words)
```

Once all words have been checked, end the program.

```
sys.stdout.write("EofSym\n");
sys.exit()
```

### 2.2.3 `checkForReservedWords()`

`sym`, which represents the current word, and `words`, which represents the list of words are passed as arguments. `currentBlockPosition` is set to 0 to indicate that a new block of characters are being checked, and `checkForNum()` is called.

```
currentBlockPosition = 0
checkForNum(sym)
sym = words[currentPosition]
```

Then, the current symbol is checked against reserved words in Java.

```
if sym == "not":
    #Append the symbol to the output file
    f = open('symbols.txt','a')
    f.write('\n' + 'NotSym')
    f.close()

    sys.stdout.write("NotSym\n");
    currentPosition+=1;
    getSym(words)
elif sym == "or":
    #Append the symbol to the output file
    f = open('symbols.txt','a')
    f.write('\n' + 'OrSym')
    f.close()

    sys.stdout.write("OrSym\n");
    currentPosition+=1;
    getSym(words)
elif sym == "and":
    #Append the symbol to the output file
    f = open('symbols.txt','a')
    f.write('\n' + 'AndSym')
```

If the word isnt a reserved word in Java, check if it is comprised of reserved symbols.

```
elif "*" in sym or "/" in sym or "%" in sym or "&" in sym or "+" in sym or "-" in
    symList = list(words[currentPosition])
    checkForReservedSymbols(symList,words)
```

If the word isnt a reserved word in Java and is not comprised of reserved symbols, assume it is an identifier.

```
else:
    sys.stdout.write("IdentSym\n");
    currentPosition+=1;
    getSym(words)
```

### 2.2.4 `checkForReservedSymbols()`

`sym`, which represents the current word, and `words`, which represents the list of words are passed as arguments.

```
sym = symList[currentBlockPosition]
```

If the symbol is a reserved relational operator, the word may be a 2-digit relational operator. `checkRelationalOp()` is run to check this.

```
if sym == "<" or sym == "=" or sym == ">" or sym == "!":
    checkRelationalOp(words)
```

Next, single digit reserved characters are checked.

```
elif sym == None:
    sys.stdout.write("null\n");
    currentBlockPosition+=1;
    checkForReservedSymbols(symList,words)
elif sym == "*":
    #Append the symbol to the output file
    f = open('symbols.txt','a')
    f.write('\n' + 'TimesSym')
    f.close()

    sys.stdout.write("TimesSym\n");
    currentBlockPosition+=1;
    checkForReservedSymbols(symList,words)
elif sym == "/":
    #Append the symbol to the output file
    f = open('symbols.txt','a')
    f.write('\n' + 'DivSym')
    f.close()

    sys.stdout.write("DivSym\n");
    currentBlockPosition+=1;
    checkForReservedSymbols(symList,words)
elif sym == "%":
    #Append the symbol to the output file
    f = open('symbols.txt','a')
```

### 2.2.5 `checkForNum()`

`sym`, which represents the current word, is passed as an argument. The symbol is categorized as a number if it is an integer.

```
if type(sym) == int:
    sys.stdout.write("NumberSym\n");
    currentPosition+=1;
```

### 2.2.6 `checkRelationalOp()`

`Sym`, which represents the current word, is passed as an argument. A reference is also made to the next symbol, and both the current symbol and next symbol are checked to determine if a 2-digit relational operator is present.

```
sym = list(words[currentPosition])
next = list(words[currentPosition])
```

```
if sym == "!" and next == "=":
    #Append the symbol to the output file
    f = open('symbols.txt','a')
    f.write('\n' + 'NeqSym')
    f.close()

    sys.stdout.write("NeqSym\n");
    currentBlockPosition+=2;
    checkForReservedSymbols(symList,words)
elif sym == "<" and next == "=":
    #Append the symbol to the output file
    f = open('symbols.txt','a')
    f.write('\n' + 'LeqSym')
    f.close()

    sys.stdout.write("LeqSym\n");
    currentBlockPosition+=2;
    checkForReservedSymbols(symList,words)
elif sym == ">" and next == "=":
    #Append the symbol to the output file
    f = open('symbols.txt','a')
```

## 2.3 A Simple Grammar for Java Programs

Shown below is a Java grammar for simple Java programs consisting of only a main method and print statements which prints a String. Terminals are shown in lower case and non-terminals are shown in upper-case:

$$PROG \rightarrow (IMPORT\_DECLARATION)^* \; CLASS\_DECLARATION$$

$$IMPORT\_DECLARATION \rightarrow "import" \; ["static"] \; identifier \; ";"$$

$$CLASS\_DECLARATION \rightarrow "public\, class" \; identifier \; "\{" MAIN\_METHOD\_DECLARATION "\}"$$

$$MAIN\_METHOD\_DECLARATION \rightarrow "public\, static\, void\, main\, (String\, []\, args)\{" \; METHOD\_BODY \; "\}"$$

$$METHOD\_BODY \rightarrow PRINT\_STATEMENT \; | \; EXIT\_STATEMENT$$

$$PRINT\_STATEMENT \rightarrow "System.out.println(" \; identifier \; ");"$$

$$EXIT\_STATEMENT \rightarrow "System.exit(0);"$$

$$identifier \rightarrow [a - zA - Z]+$$

## 2.4 Parser

The goal of the recursive descent parser is to construct a syntax tree for the sentence generated from `scanner.py`, get the context info for the syntax tree, and accept/reject the sentence.

The start symbol, PROG, is implemented with the prog() method, which checks for imports and class declaration (these are found at the top of Java programs).

```
def prog(sentence):
    if(currentPosition == 0):
        if (sentence[0] == 'PublicSym'):
            class_declaration(sentence)
        else:
            terminate(False, sentence[currentPosition])
    else:
        terminate(False, sentence[currentPosition])
```

`main()` takes a sentence generated from `scanner.py` as a command-line argument, and starts `prog()` to begin the recursive descent process.

```
sentence = sys.argv[1]
prog(sentence);
```

For each production, methods are called in the order that their corresponding non-terminal symbol appears. Sentences are accepted or rejected based on an expected set of terminal symbols, or an unexpected terminal symbol. After a terminal symbol is checked, the program iterates to the next symbol. This code illustrates the process for the CLASS_DECLARATION production.

```
def class_declaration(sentence):
    global currentPosition

    if (sentence[currentPosition] == 'PublicSym'):
        currentPosition+=1
        if (sentence[currentPosition] == 'ClassSym'):
            currentPosition+=1
            if (sentence[currentPosition] == 'IdentSym'):
                currentPosition+=1
                #identifier(sentence)
                if (sentence[currentPosition] == 'LCurlSym'):
                    currentPosition+=1
                    main_method_declaration(sentence)
                    if (sentence[currentPosition] == 'RCurlSym'):
                        currentPosition+=1
                        return True
                    else:
                        terminate(False, sentence[currentPosition])
                else:
                    terminate(False, sentence[currentPosition])
            else:
                terminate(False, sentence[currentPosition])
        else:
            terminate(False, sentence[currentPosition])
    else:
        terminate(False, sentence[currentPosition])
```

This code illustrates the same process for the MAIN_METHOD_DELCARATION production.

```
def main_method_declaration(sentence):
    global currentPosition
    if (sentence[currentPosition] == 'PublicSym'):
        currentPosition+=1
        if (sentence[currentPosition] == 'StaticSym'):
            currentPosition+=1
            if (sentence[currentPosition] == 'VoidSym'):
                currentPosition+=1
                if (sentence[currentPosition] == 'MainSym'):
                    currentPosition+=1
                    if (sentence[currentPosition] == 'LparenSym'):
                        currentPosition+=1
                        if (sentence[currentPosition] == 'StringSym'):
                            currentPosition+=1
                            if (sentence[currentPosition] == 'ArraySym'):
                                currentPosition+=1
                                if (sentence[currentPosition] == 'ArgsSym'):
                                    currentPosition+=1
                                    if (sentence[currentPosition] == 'RparenSym'):
                                        currentPosition+=1
                                        if (sentence[currentPosition] == 'LCurlSym'):
                                            currentPosition+=1
                                            method_body(sentence)
                                            if (sentence[currentPosition] == 'RCurlSym'):
                                                currentPosition+=1
                                                return True
                                            else:
                                                terminate(False, sentence[currentPosition])
                                        else:
                                            terminate(False, sentence[currentPosition])
                                    else:
                                        terminate(False, sentence[currentPosition])
                                else:
                                    terminate(False, sentence[currentPosition])
                            else:
                                terminate(False, sentence[currentPosition])
                        else:
                            terminate(False, sentence[currentPosition])
                    else:
                        terminate(False, sentence[currentPosition])
                else:
                    terminate(False, sentence[currentPosition])
            else:
                terminate(False, sentence[currentPosition])
        else:
            terminate(False, sentence[currentPosition])
    else:
        terminate(False, sentence[currentPosition])
```

## 2.5 Code Generator

As mentioned before, Java programs are typically compiled into byte code which is then interpreted on the JVM (Java Virtual Machine). JTAC bypasses this requirement by generating ARM code that can be run on any ARM processor or ARM virtual machine.

It was necessary to understand the subset of the ARM instruction set that was necessary for the scope of the project. Meaning, it was necessary to identify which instructions needed to have code generated. Because of the need to load constants and variables, the `MOV` (move) and `LDR` (load register) instructions are required. As well, to actually execute print statements and to exit the program, the `SWI` (software interrupt) instruction is needed. This instruction hands control over to the operating system and performs a certain function depending on which flags are set.

The code generator module for JTAC follows the design pattern of the one studied in the class. The module serves as a small library of routines that are responsible for generating code for individual parts of the program. The module keeps tracks of the 16 available all purpose registers in the ARM architecture by use Python dictionaries. Similar to the pattern used by the compiler for P0, the methods in the code generator module are called at the appropriate time while the parsing procedures are taking place. The following sections outline the details of some of the these methods.

### 2.5.1 `genProgEntry()`

This method is called at the very beginning of the parsing procedures. It sets up necessary labels for the program entry point, which is the `main()` method. As well, it creates a label for the data section, which is where string variables will be stored.

This procedure will also automatically write two instructions to the output. These two `MOV` instructions fill registers R0 and R7 with appropriate values so that the operating system will issue a print command the next time that a software interrupt is called. Due to the simple structure of the Java grammar proposed, it can be certain that the first type of statement encountered will be a print statement. So, these flags are set as part of the set up routine.

### 2.5.2 `genPrintStatement()`

This routine is responsible for generating ARM code for Java print statements. Each print statement requires a few different instructions. First, the size of the string is moved into a register with the `MOV` command. Next, because the string is stored in memory, its address is moved into register using the `LDR` instruction. With these two values loaded in, the system call to print the string can be performed. This is done using `SWI`. It will print the number of bytes specified in register R2 starting from the address stored in R1.

Once the instructions are generated, the values of the registers are updated. Lastly, for each new print statement, the data section of the ARM code is updated to allocate space for each new string.

#### 2.5.3 `genProgExit()`

When the parser detects that there is a system exit statement, it knows that there must be no more print statements and that the program is terminating. In preparation of the program exit, the R7 register is set to a different value than before. This will serve as a flag that tells the operating system that the next time there is a software interrupt to exit, instead of printing or doing something else. The routine is simple, it just sets the flag and then writes the `SWI` instruction to exit.

#### 2.5.4 `genProgram()`

If the parsing procedure completes succesfully, then this method will take the code generated for the main method and the code generated for the data section and append them together. This creates one output, which is then written to a file.

## 3 Discussion

The JTAC prototype is able to compile simple Java programs consisting of the following elements:

a) main method

b) print statements

c) exit statements

However, JTAC can easily be extended to include:

a) variable declaration and assignment

b) arithmetic

c) defined methods

Here, we discuss code modifications to the various components of JTAC to make these extensions.

## 3.1 `scanner.py`

### 3.1.1 Arithmetic

`checkForReservedSymbols(symList,words)` already contains conditionals which recognize arithmetic operators.

```
elif sym == "*":
    #Append the symbol to the output file
    f = open('symbols.txt','a')
    f.write('\n' + 'TimesSym')
    f.close()

    sys.stdout.write("TimesSym\n");
    currentBlockPosition+=1;
    checkForReservedSymbols(symList,words)
elif sym == "/":
    #Append the symbol to the output file
    f = open('symbols.txt','a')
    f.write('\n' + 'DivSym')
    f.close()

    sys.stdout.write("DivSym\n");
    currentBlockPosition+=1;
    checkForReservedSymbols(symList,words)
```

### 3.1.2 Variable declaration and assignment

`checkForReservedSymbols(symList,words)` would need to be extended to recognize data types, such as `int` and `String`.

```
elif sym == "int":
        #Append the symbol to the output file
        f = open('symbols.txt','a')
        f.write('\n' + 'IntSym')
        f.close()
        currentPosition+=1;
        getSym(words)


elif sym == "String":
        #Append the symbol to the output file
        f = open('symbols.txt','a')
        f.write('\n' + 'StringSym')
        f.close()
        currentPosition+=1;
        getSym(words)
```

### 3.1.3 Methods

Most words needed to declare methods are already recognized in `checkForReservedSymbols(symList,words)`. Certain symbols, such as `void`, would need to be added.

```
elif sym == "void":
        #Append the symbol to the output file
        f = open('symbols.txt','a')
        f.write('\n' + 'VoidSym')
        f.close()
        currentPosition+=1;
        getSym(words)
```

## 3.2 Java Grammar

The following productions, highlighted in blue, would need to be added to define variable declaration and assignment (Strings and integers are shown as an example), arithmetic expressions, and method definition.

$$PROG \rightarrow (IMPORT\_DECLARATION)^* \ CLASS\_DECLARATION$$

$$IMPORT\_DECLARATION \rightarrow "import" \ ["static"] \ identifier \ ";"$$

$$CLASS\_DECLARATION \rightarrow "public\ class" \ identifier \ "\{" METHOD\_DECLARATION^* \ MAIN\_METHOD\_DECLARATION \ "\}"$$

$$METHOD\_DECLARATION \rightarrow TYPE \ identifier \ "(" \ (TYPE \ identifier)^* \ ")" \ METHOD\_BODY \ "\}"$$

$$MAIN\_METHOD\_DECLARATION \rightarrow "public\ static\ void\ main\ (String\ []\ args)\{" \ METHOD\_BODY \ "\}"$$

$$METHOD\_BODY \rightarrow PRINT\_STATEMENT \mid EXIT\_STATEMENT \mid ARITHMETIC\_STATEMENT \mid ASSIGNMENT$$

$$PRINT\_STATEMENT \rightarrow "System.out.println(" \ identifier \ ");"$$

$$EXIT\_STATEMENT \rightarrow "System.exit(0);"$$

$$ARITHMETIC\_STATEMENT \rightarrow (identifier \ ARITHMETIC\_OP)^+ identifier$$

$$ARITHMETIC\_OP \rightarrow "+" \mid "-" \mid "/" \mid "*"$$

$$TYPE \rightarrow "boolean" \mid "int" \mid "String"$$

$$STRING \rightarrow "."$$

$$INT \rightarrow [0-9]+$$

$$identifier \rightarrow [a-zA-Z]+$$

### 3.3 rdparser.py

New methods would need to be created to handle variable declaration and assignment (Strings and integers are shown as an example), arithmetic expressions, and method definition, as defined in the grammar above. For example, the following code could be used to accept or reject a method body.

```
def method_body(sentence):
    global currentPosition

    if (currentPosition < len(sentence)):
        if (sentence[currentPosition] == 'SystemSym') and (sentence[currentPosition+2] == 'OutSym'):
            print_statement(sentence)
            return True

        elif (sentence[currentPosition] == 'SystemSym') and (sentence[currentPosition+2] == 'ExitSym'):
            end_prog(sentence)
            return True

elif (sentence[currentPosition+1].type == 'int'):
            arithmetic_statement(sentence)
            return True

    else:
        terminate(True, sentence)
```

It should be noted that the above code includes a conditional branch to handle arithmetic (only for integers in this example).

```
elif (sentence[currentPosition+1].type == 'int'):
            arithmetic_statement(sentence)
            return True

    else:
        terminate(True, sentence)
```

### 3.4 generator.py

The code generator could be extended in a similar way to handle these new features. New methods could be added to this module that could be responsible for generating code for arithmetic operations or assignment statements. Consider the following statement, `x = y + z`, which adds two integers. A method in the code generator might generate the following instructions.

```
def genAssign(x, y, z):
    putInstruction(LDR, "R5", adr(y))
    putInstruction(LDR, "R6", ard(z))
    putInstruction(ADD, "R3", "R5", "R6")
    putInstruction(STR, "R3", adr(x))
```

Here, the values of y and z have been loaded in from memory. Their sum is stored in the register R3, and then that result is written to the memory address of x.

As well, extending this code generator to handle method calls can be done in a very similar way to how it is done in the P0 compiler. Namely, when a procedure is entered or exited, there are instructions generated for keeping track of the stack pointer. This can be done the same way for Java methods.

As we might have expected, the JTAC solution for compiling Java code has shown significantly faster compile times. One experiment that was done was to compile a simple HelloWorld Java program using two different approaches. The first was to compile it using a common IDE (Intelli-J) and the other was to compile it using JTAC. The following figures summarize the compile times for each method.

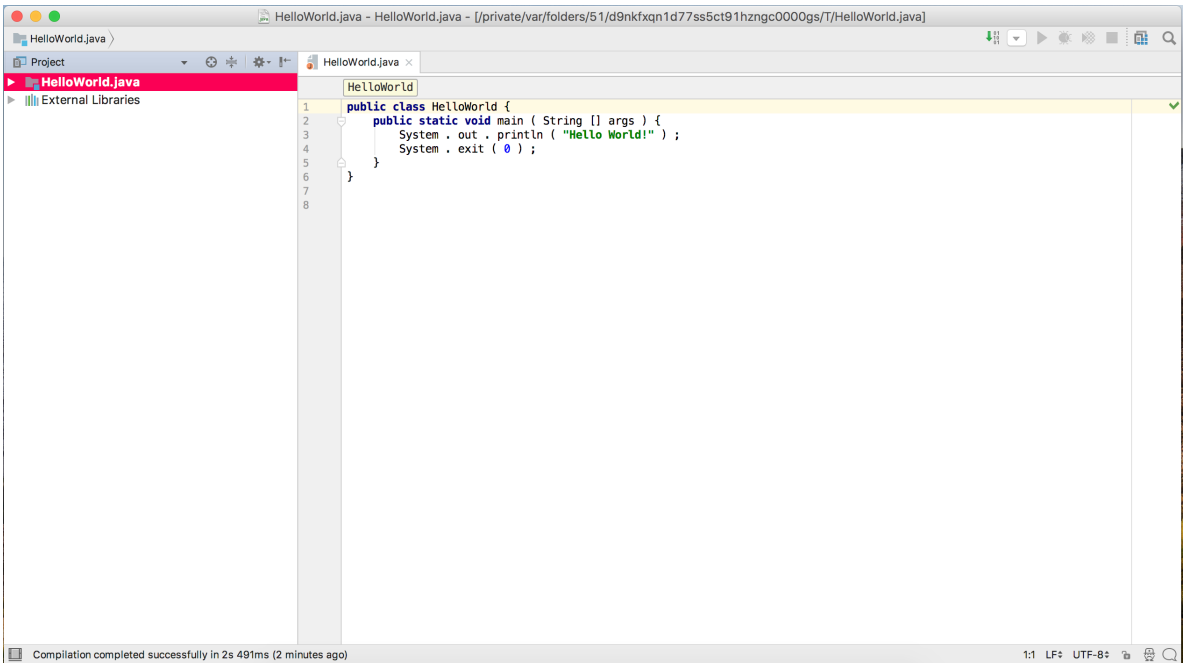



Figure 1: Compile time in Intelli-J was 2.491 seconds.

```
real    0m0.207s
user    0m0.087s
sys     0m0.096s
```

Figure 2: Compile time in JTAC was 0.207 seconds.

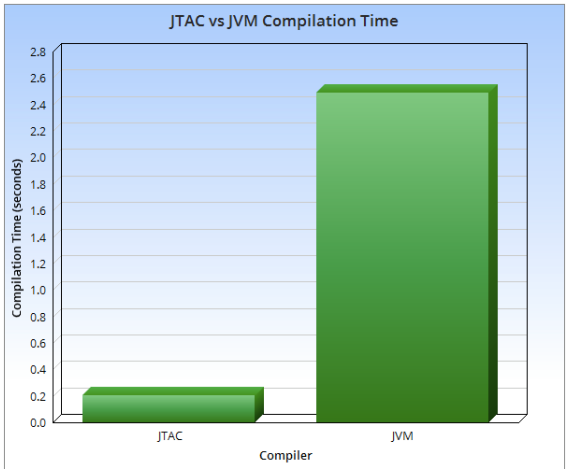
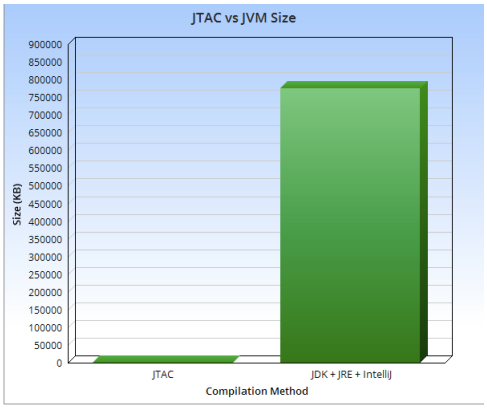



Figure 3: A side by side comparison of the compiler times.

Another important consideration for developers is the memory requirement to set up and run their programs. JTAC offers a solution that is significantly more lightweight than an IDE solution. The following figure summarizes the memory expense for each option. Clearly, JTAC offers significant improvements.



# 4 Conclusion

JTAC is a program that accepts a potential Java program as input. JTAC performs lexical and syntactic analysis, and generates the relevant ARM code once a valid Java program is detected. In theory, this ARM code could then be run on a computer that has an ARM processor or that has access to an ARM virtual environment.

As expected, JTAC is significantly smaller and faster than JVM for simple Java programs. JTAC is 26 kilobytes as opposed to 775 megabytes to run program using JRE, JDK, and IntelliJ, and can compile a simple HelloWorld program in 0.207s as opposed to 2.491s through Intelli-J.

We discussed how to extend our code to handle arithmetic, variable declaration and assignment, and methods. To extend JTAC further (for example, to handle interfaces), developers would first have to extend the scanner to recognize new reserved words, such as `static` in `checkForReservedWord()s`, and new symbols, such as the mod symbol, %, in `checkForReservedSymbols()`. New productions would need to be added to JTACs Java grammar, (e.g INTERFACE_DECLARATION $\rightarrow$ ...) and the appropriate productions would need to be added as new methods (e.g def interface_declaration(sentence)) in rdparser.py. Existing productions, such as PRINT_STATEMENT, would need to be extended to handle more complex arguments. Finally, additional code generation methods would need to be added in
tt generator.py.

## References

[1] Anon, (2018). [ebook] Available at: http://vision.gel.ulaval.ca/ jflalonde/cours/1001/h17/docs/arm-instructionset.pdf [Accessed 21 Apr. 2018].

[2] Ltd., A. (2018). Architecture — Instruction Sets  Arm Developer. [online] ARM Developer. Available at: https://developer.arm.com/products/architecture/instruction-sets [Accessed 21 Apr. 2018].

[3] Patterson, D. and Hennessy, J. (n.d.). Computer Organization and Design.

[4] Sebesta, R. (n.d.). Concepts of Programming Languages.

[5] Wirth, N. (1996). Compiler Construction. Amsterdam: Addison-Wesley.