CS321: Compilers  
Java ByteCode Generation

7/6/2020

# TEAM

|  |  |
| --- | --- |
| Arsany Atef Abdo | 10 |
| Kirellos Malak Habib | 35 |
| Michael Said Beshara | 38 |
| Yomna Gamal El-Din Mahmoud | 63 |

## National IDs:

## 

## 

## Abstract and objective

This phase of the assignment aims to practice techniques of constructing semantics rules to generate intermediate code using YACC and LEX.

## Introduction and literature review

A Java programmer does not need to be aware of or understand Java bytecode at all. However, as suggested in the IBM developer Works journal, "Understanding bytecode and what bytecode is likely to be generated by a Java compiler helps the Java programmer in the same way that knowledge of assembly helps the C or C++ programmer.

The JVM is both a stack machine and a register machine. Each frame for a method call has an "operand stack" and an array of "local variables".

The operand stack is used for operands to computations and for receiving the return value of a called method, while local variables serve the same purpose as registers and are also used to pass method arguments. The maximum size of the operand stack and local variable array, computed by the compiler, is part of the attributes of each method.

Each can be independently sized from 0 to 65535 values, where each value is 32 bits. long and double types, which are 64 bits, take up two consecutive local variables(which need not be 64-bit aligned in the local variables array) or one value in the operand stack (but are counted as two units in the depth of the stack).

Each bytecode is composed of one byte that represents the opcode, along with zero or more bytes for operands.

Of the 256 possible byte-long opcodes, as of 2015, 202 are in use (~79%), 51 are reserved for future use (~20%), and 3 instructions (~1%) are permanently reserved for JVM implementations to use.

Two of these (impdep1 and impdep2) are to provide traps for implementation-specific software and hardware, respectively. The third is used for debuggers to implement breakpoints.

Instructions fall into several broad groups:

* Load and store (e.g. aload\_0, istore)
* Arithmetic and logic (e.g. ladd, fcmpl)
* Type conversion (e.g. i2b, d2i)
* Object creation and manipulation (new, putfield)
* Operand stack management (e.g. swap, dup2)
* Control transfer (e.g. ifeq, goto)
* Method invocation and return (e.g. invokespecial, areturn)

There are also a few instructions for several more specialized tasks such as exception throwing, synchronization, etc.

Many instructions have prefixes and/or suffixes referring to the types of operands they operate on.

For example, iadd will add two integers, while dadd will add two doubles. The const, load, and store instructions may also take a suffix of the form \_n, where n is a number from 0–3 for load and store. The maximum n for const differs by type.

The const instructions push a value of the specified type onto the stack. For example, iconst\_5 will push an integer (32-bit value) with the value 5 onto the stack, while dconst\_1 will push a double (64-bit floating-point value) with the value 1 onto the stack. There is also an aconst\_null, which pushes a null reference. The n for the load and store instructions specifies the index in the local variable array to load from or store to. The aload\_0 instruction pushes the object in local variable 0 onto the stack (this is usually this object). istore\_1 stores the integer on the top of the stack into local variable 1. For local variables beyond 3 the suffix is dropped, and operands must be used.

## Description of the problem

* Develop a suitable Syntax Directed Translation Scheme to convert Java code to Java bytecode, performing necessary lexical, syntax and static semantic analysis.
* Proposed grammars are required to cover the following features:

• Primitive types (int, float) with operations on them (+, - , \*, / )

• Boolean Expressions (Bonus marks)

• Arithmetic Expressions

• Assignment statements

• If-else statements

• for loops (Bonus marks)

• while loops

## Algorithms and techniques

* IF-ELSE
* WHILE
* FOR
* BOOLEAN EXPRESSION

## Explanation of functions

* Main()

It reads the code in the input file and parse it.

* Print\_code(vector<string \*> \* code)

Takes a list of strings containing the bytecode and prints it to the output file.

## Main files

* LEX.l: It is the lexical analyzer which returns a token type.
* SYN.y: It contains grammar rules and semantic actions to produce bytecode
* Makefile: It is the main file which runs the project.
* Label.cpp/Label.h: Helper class for generation of labels.

## Sample run

## Conclusion

## References

* https://docs.oracle.com/
* https://www.ibm.com/developerworks/ibm/library/it-haggar\_bytecode/
* http://jasmin.sourceforge.net/
* https://www.javaworld.com/article/2072355/learn-to-speak-jamaican.html
* https://wiki.freepascal.org/FPC\_New\_Features\_3.0.0#Support\_for\_the\_Java\_Virtual\_Machine\_and\_Dalvik\_targets
* https://wiki.freepascal.org/FPC\_JVM
* http://headius.blogspot.com/2007/01/invokedynamic-actually-useful.html
* http://dinosaur.compilertools.net/bison/bison\_6.html
* http://alumni.cs.ucr.edu/~lgao/teaching/bison.html
* https://docplayer.net/25186893-6-7-backpatching-one-pass-code-generation-using-backpatching-410-chapter-6-intermediate-code-generation.html
* http://www.brainkart.com/article/Implementing-L-Attributed-SDD-s\_8158/
* https://blog.jamesdbloom.com/JavaCodeToByteCode\_PartOne.html
* http://www.talkplayfun.com/Bison-and-Flex/

## Data structures

-There are three structs used for types:

1. Exp:

* It contains (int type, vector<string \*> \*code, vector<string \*> \*next) that this expression has.
* It is used for EXPRESSION and SIMPLE\_EXPRESSION.

1. Factor:

* It contains (int type, vector<string \*> \*code)
* It is used for TERM and FACTOR.

1. Block:

* It contains (int l\_id, vector<string \*> \*code, vector<string \*> \*next) that this block has.
* It is used for METHOD\_BODY, STATEMENT\_LIST, STATEMENT, IF, WHILE, DECLARATION and ASSIGNMENT.

1. Enum {INT\_T, FLOAT\_T, BOOL\_T, ERROR\_T} contains primitive types.
2. unordered\_map<string, pair<unsigned, int>> symbol\_table / unordered\_map<unsigned, string> memory\_table:

* the two maps used to manage declared variables and handle the life scope variables.

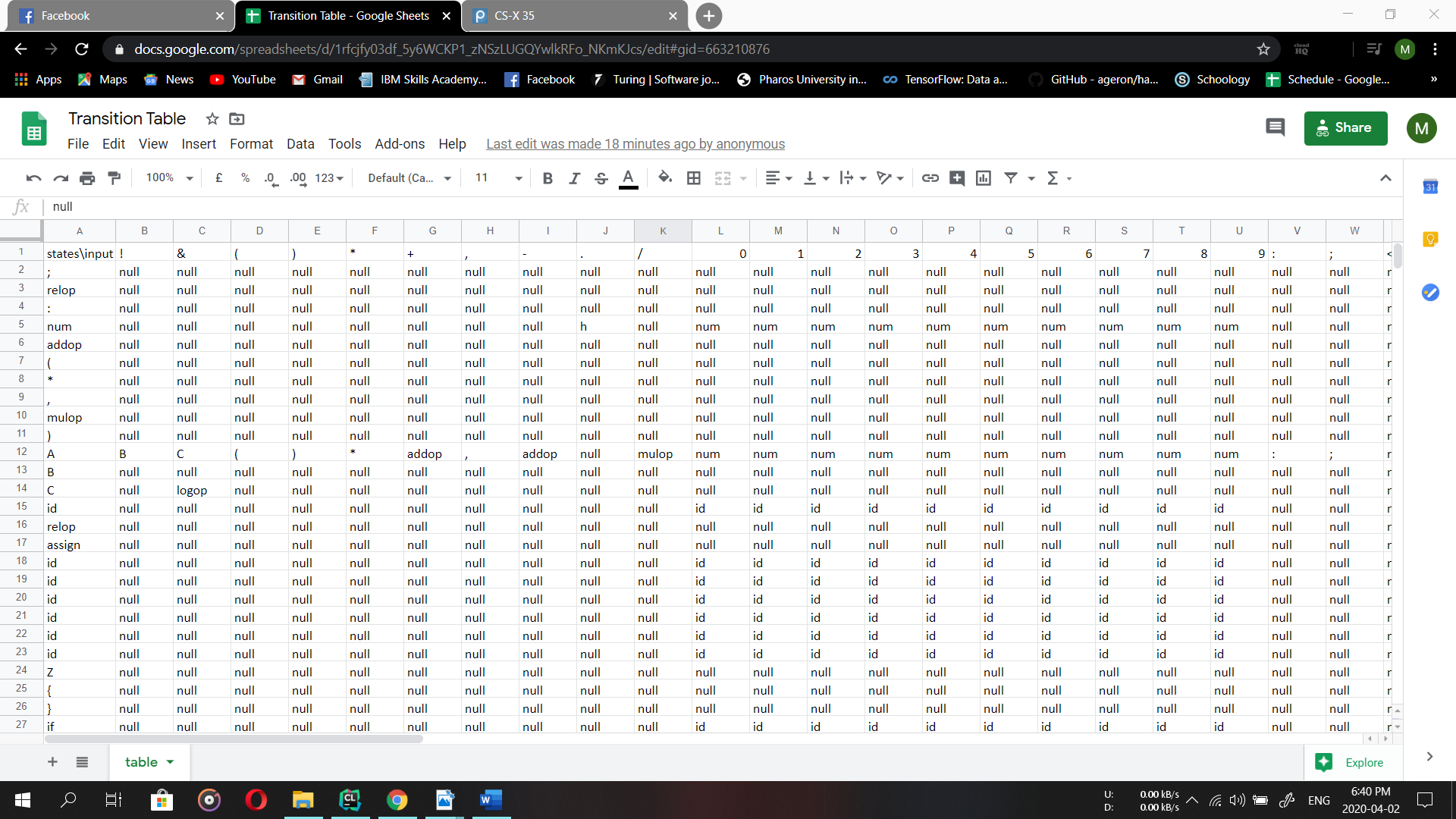
1. unordered\_map< char, string> op\_map / unordered\_map< string, string> real\_ops:

* the two maps used to map between operation and its byte code.

## The resultant transition table for the minimal DFA in part 1

Fully spreadsheet is found here for the resultant transition table:

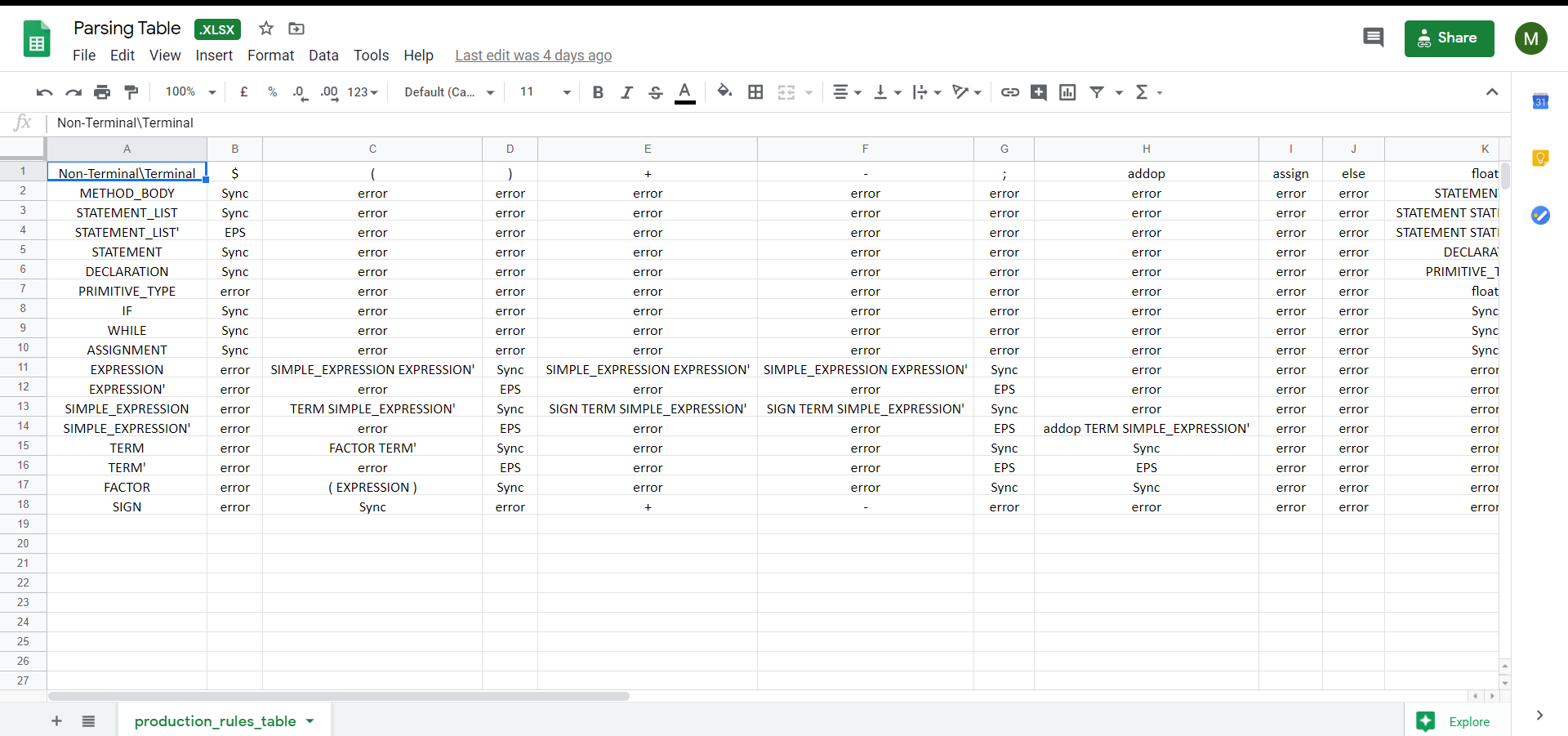
<https://docs.google.com/spreadsheets/d/1rfcjfy03df_5y6WCKP1_zNSzLUGQYwlkRFo_NKmKJcs/edit?usp=sharing>



## Parsing Tables if any in part 2

Fully spreadsheet is found here for the resultant grammar table:

<https://drive.google.com/file/d/1Cbf5-nn8W5NPp9cnPAML1PzFLI5SqDzj/view?usp=sharing>



## Comments on tools

* Flex: It is represented LEX .l file which contains the lexical rules. It takes the input code and returns tokens which are used later by bison.
* Bison: it is represented SYN.y file which contains context free grammar and semantic rules. It takes tokens from flex and generates java bytecode.
* Jasmin: it is used to generate from java bytecode output file out .class file that we can run it by java compiler to test compiler. After generating java bytecode file, we added multiple lines to put our code inside a class and a main to be able to be converted by Jasmin.

We use this command (java - jar jasmin. jar out. j) to convert java bytecode file into out.class, then we use this command (java out) to run the program and test it.

## Assumptions and Justification

* Our compiler supports the Primitive types (int, float, boolean).
* Our compiler supports if, if-else, while and for statements.
* Our compiler supports arithmetic operations (+, -), multiplication operations (\*, /, %).
* Our compiler supports Boolean Expressions with relational and logical operations.
* Our compiler supports if, if-else, while and for statements.
* We define new production (BLOCK) to handle life scope variables.
* Our compiler supports casting from int to float.
* Our compiler supports declaration with certain value.
* We assume order of operations (! , ||, && ) and ( \* , / , + , - ).
* Our compiler does not support (++, --) operations.

## The role of each student in the project