U. PORTO

FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

PE23 - JVM Assembler

Projeto Integrador

Relatório

Grupo

Diogo Gomes	201805367	Contribution: 25%
João Silva	201906478	Contribution: 25%
Maria Dantas	201709467	Contribution: 25%
Rogério Rocha	201805123	Contribution: 25%

Index

Summary

This report was elaborated according to the project developed in scope of the course "Projeto Integrador", and its objective is the development of a new Java Virtual Machine Assembler that provides a number of useful features, such as a more advanced error identification, the possibility to insert instrumentation instructions, the possibility to use symbols instead of numbers for local variables, and the possibility to assign symbols to local variables.

Introduction

This project's main goal was to develop a new JVM Assembler with more user-friendly features. In this report we will discuss our approach to the project, and the different phases of the development aswell what we can do in the future and how to use our program.
The report has the following structure:

- **Java Virtual Machine** - what is the jvm, a small introduction.
- **Jasmin** - an introduction to jasmin.
- **JavaCC** - what is javacc, its main features and what can it be used for.
- **New Assembler Analysis** - Analysis of the current JVM Assembler status and our approach to this project.
- **Grammar** - Grammar structure and content.
- **Jasmin Generation** - Code generation with some code snippets.
- **Application** - access to our open source project page, how to run the developed program.
- **Retrospective** - a look on the project development, our struggles and how we were able to get over them.
- **Future Work** - what can we improve in future versions.
- **Conclusion** - project conclusions.

Java Virtual Machine

The Java Virtual Machine is the cornerstone of the Java platform. It is the component of the technology responsible for its hardware and operating system independence, the small size of its compiled code, and its ability to protect users from malicious programs. It is an abstract computing machine. Like a real computing machine, it has an instruction set and manipulates various memory areas at run time.

The Java Virtual Machine knows nothing of the Java programming language, only of a particular binary format, the class file format. A class file contains Java Virtual Machine instructions (or bytecodes) and a symbol table, as well as other ancillary information.

For the sake of security, the Java Virtual Machine imposes strong syntactic and structural constraints on the code in a class file. However, any language with functionality that can be expressed in terms of a valid class file can be hosted by the Java Virtual Machine. Attracted by a generally available, machine-independent platform, implementers of other languages can turn to the Java Virtual Machine as a delivery vehicle for their languages.

The Java Virtual Machine specified here is compatible with the Java SE 14 platform, and supports the Java programming language specified in The Java Language Specification, Java SE 14 Edition.

Jasmin

Jasmin is a Java Assembler Interface. It takes ASCII descriptions for Java classes, written in a simple assembler-like syntax using the Java Virtual Machine instructions set. It converts them into binary Java class files suitable for loading into a Java interpreter.

Jasmin was originally created as a companion to the book "Java Virtual Machine", written by Jon Meyer and Troy Downing and published by O'Reilly Associates. Since then, it has become the de-facto standard assembly format for Java. It is used in dozens of compiler classes throughout the world, and has been ported and cloned multiple times. Jasmin remains the oldest and the original Java assembler, for better or worse.

Jasmin was created to be a simple assembler with an almost 1-to-1 with the components of a class file. Does not make a series of checks such as: verification of the existence of classes being referenced, verification of the correct formation of the descriptors, inlining of mathematical expressions, substitution of variables, and macro support. Furthermore, it was capable of testing almost all virtual machine features.

JavaCC

JavaCC is a top-down parser generator, this allows the use of more general grammars. Top-down parsers have a number of advantages such as being easier to debug, having the ability to parse to any non-terminal in the grammar, and also having the ability to pass values (attributes) both up and down the parse tree during parsing.

By default, JavaCC generates an LL(1) parser. However, there may be portions of grammar that are not LL(1). JavaCC offers the capabilities of syntactic and semantic lookahead to resolve shift-shift ambiguities locally at these points.
JavaCC generates parsers that are 100% pure Java, so there is no runtime dependency on JavaCC and no special porting effort required to run on different machine platforms.

The lexical specifications and the grammar specifications are both written together in the same file. It makes grammars easier to read since it is possible to use regular expressions inline in the grammar specification, and also easier to maintain.
The lexical analyzer of JavaCC can handle full Unicode input, and lexical specifications may also include any Unicode character. This facilitates descriptions of language elements such as Java identifiers that allow certain Unicode characters (that are not ASCII), but not others.

Tokens that are defined as special tokens in the lexical specification are ignored during parsing, but these tokens are available for processing by the tools. A useful application of this is in the processing of comments.
JavaCC offers many options to customize its behavior and the behavior of the generated parsers. Examples of such options are the kinds of Unicode processing to perform on the input stream, the number of tokens of ambiguity checking to perform etc.

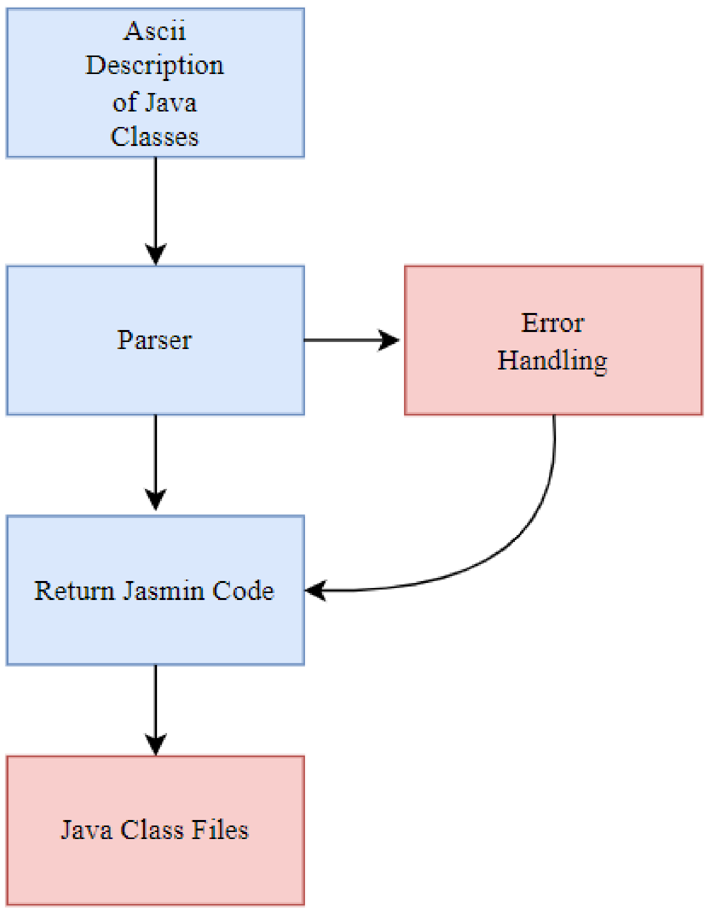
JavaCC generated parsers are able to clearly point out the location of parse errors with complete diagnostic information.

New Assembler Analysis

The current assembler used is Jasmin, which is very limited in terms of helpful features and in terms of error identification.

The new assembler developed in this project is able to provide a number of useful features, such as more advanced error identification, the possibility to insert instrumentation instructions, the possibility to use symbols instead of numbers for local variables, and the possibility to assign symbols to local variables.

The main focus was the improvement of the error handling, as the current assembler gives little to no useful information to the user regarding the error being generated, as well as sometimes missguide the user to where this error can be found.



1. Block Diagram

Grammar

Our first goal was to produce a grammar that parsed successfully Jasmin Code that was given as input. In later stages of this project this parsing is a way to retrieve the various elements and information present in the Jasmin code, so that we return the same code we just parsed back to the user and later to extend the functional abilities of Jasmin and create new ones in the form of JVM instructions.

The grammar was created in the form of a file with a jj extension that is associated with the JavaCC (Java Compiler Compiler) a parser generator for use related to Java applications. However for this project we used the most advanced of JavaCC which is the JavaCC 21. It has many feature improvements and generates modern and more readable Java code.

The grammar file starts with a list of options, each option binding specifies the setting of one option.
On our Jasmin Parser we set the PARSER_CLASS option to the name we wanted our parser java class to have (JasminParser) and as we did not need to generate a parser that automatically build an AST (Abstract Syntax Tree) we set the TREE_BUILDING_ENABLED option to false.

```
PARSER_CLASS=JasminParser;
TREE_BUILDING_ENABLED=false;
```

JavaCC 21 introduces a new functionality in form of the INJECT statement that allows us to “inject” code into to the files that the tool generates, so to fulfill goals of later stages of the project like generation of code we injected the code below in JasminParser class the tool produced.

```
INJECT JasminParser :
   import Jasmin.*;
           import java.util.ArrayList;
{
           ArrayList listInstr = new ArrayList();

           ArrayList getInstr() {return listInstr;}

   Jasmin jasmin = new Jasmin();

   public void jasminCode(){
      System.out.println(jasmin.getCode()); }}
```

The JavaCC lexical specification is organized into a set of lexical states, each one containing an ordered list of regular expressions. One of the regular expression types we can define is the SKIP type that specifies the action it has to happen when the regular expression defined below is matched. The action is in fact to simply throw away the matched string, so we want the parsing to skip white spaces.

```
SKIP : <WHITESPACE : (" " | "\t" | "\n" | "\r" | "\f")+> ;
```

Another regular expression type we defined was the TOKEN that creates a token using the matched string and sends it to the parser. Below is a very small example version of our actual list of tokens that our parser can receive.

```
TOKEN :
   < INTEGER : (["0"-"9"])+ > |
   < LONG : (["0"-"9"])+ > |
   < CLASS : ".class" > |
   < PUBLIC : "public" > |
   < SUPER : ".super" > |
```

After we declare the SKIP and TOKEN regular expressions we start to declare other productions related to the parsing of the Jasmin code.
We basically aimed to follow the general structure of a Jasmin file, so our productions were based around concepts like: class declaration, super class, methods and instructions in methods.

We defined the Start production as:

```
Start:
                         (<LINE_COMMENT>)*      <CLASS>      <PUBLIC>      <IDENTIFIER>
{jasmin.setClassName(getToken(0).getImage());}
   (<LINE_COMMENT>)* (SuperDeclaration)?
   (<LINE_COMMENT>)* (MethodDeclaration)* (<LINE_COMMENT>)*;
```

It's worth mentioning that we found issues on integrating Jasmin comments on our grammar because of their special syntax, so we opted to integrate them in every production we have, which is the naive way to do it.

In almost all our productions we integrated Java code in a later phase of a project to make viable the return generation of the input code, that's why line like {jasmin.setClassName(getToken(0).getImage());} appears on Start production.

Our start production focus as said before on the general structure of a Jasmin file and leads the way to other productions:

```
ClassDeclaration:
        <CLASS> <PUBLIC> <IDENTIFIER>;

SuperDeclaration:
        <SUPER> <IDENTIFIER> {jasmin.setSuperClassName(getToken(0).getImage());}
  (<SLASH> {jasmin.setSuperClassName(getToken(0).getImage());}
  <IDENTIFIER> {jasmin.setSuperClassName(getToken(0).getImage());} )* ;
```

One important production is the Instruction that is filled with all of the JVM instructions existing and deals with the arguments of each one of them in the case that they exist.

A cut of our Instruction production:

```
Instruction:
                                                                     <NEW>
{jasmin.addInstruction(getToken(0).getImage(),getToken(-1).getImage(),getToken(-2).getI
mage());} (NewInst)  | ...
```

Methods are an important part of the Jasmin code structure, so here is how we built the productions to parse Jasmin methods using a split strategy:

```
StartMethod:
   <DOT_METHOD> <PUBLIC>
{jasmin.setMethodAccessModifier(getToken(0).getImage());}
    (<STATIC>)? {if(getToken(0).getImage().equals("static")) jasmin.setMethodStatic();}
;

MethodName:
   (<LESS> <IDENTIFIER> <GREAT>|
   <IDENTIFIER>);

MethodArguments:
   <OPEN_PAREN> (MethodArgumentNormalType)* (MethodArgumentArrayType)
   (<CLOSE_PAREN>  (ReturnTypeNormalType | ReturnArray) | <LINE_COMMENT>
{jasmin.setParameter(getToken(0).getImage());});

MethodArgumentNormalType:
   <INT> {jasmin.setParameter(getToken(0).getImage());};
```

```
MethodArgumentArrayType:
   (<ARRAY_IDENTIFIER>{jasmin.setParameter(getToken(0).getImage());})?
    (<IDENTIFIER> {jasmin.setParameter(getToken(0).getImage());}
     | <SLASH>{jasmin.setParameter(getToken(0).getImage());})*;

ReturnTypeNormalType:
   <INT> | <VOID>;

ReturnArray:
   (<ARRAY_IDENTIFIER> {jasmin.addOperandInstPath(getToken(0).getImage());})?
   (<IDENTIFIER> {jasmin.addOperandInstPath(getToken(0).getImage());} |
   <SLASH> {jasmin.addOperandInstPath(getToken(0).getImage());})*
   (<LINE_COMMENT> {jasmin.addOperandInstPath(getToken(0).getImage());})?;

MethodBody:
   (<LINE_COMMENT>)* (Instruction |LimitInstructions| Label) (<LINE_COMMENT>)* ;
```

We finalized our grammar file with multiple productions related to specific JVM instructions and their arguments like limit, field and arrays instructions and auxiliary productions.

Jasmin Generation

One of the goals we setted for this project was to return the Jasmin code we received as input to the user.

For this purpose we needed to create some Java classes that can store a lot of information we retrieve the moment we are parsing the input code and consequently integrate Java code inside the productions in our grammar file to do so.

We initialize the main class that generates the code (Jasmin Class) in the parser class that the tool generates (JasminParser) using the functionalities of the INJECT statement.

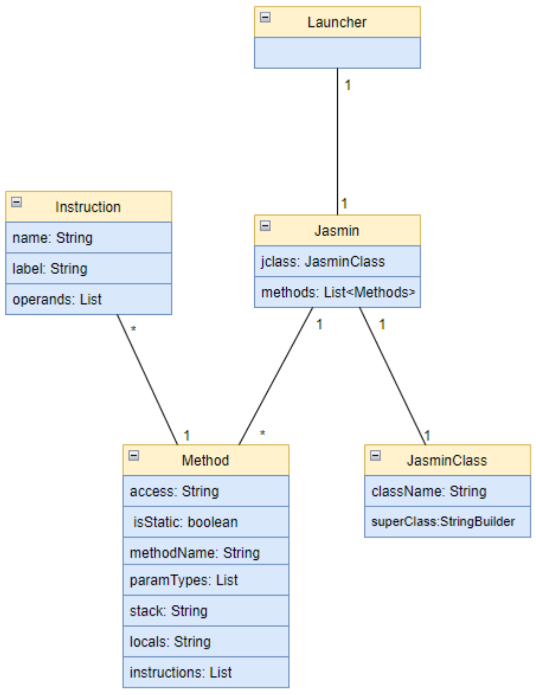
We created a Launcher Java class that is responsible for initializing the parsing tool and print to the user the input Jasmin code.

Recurring calls to methods that belong to this class is basically the way we generate the code from the parsing.

But this class alone can't do the job, we created more classes that stored specific elements present in the Jasmin code structure. These classes are: JasminClass

(class and superclass concepts), Method (information about the methods parsed) and Instruction.

The Structure behind this process is described in this UML class Diagram:



2. UML Class Diagram

Application

We keep our project open source here, for anyone that might be interested in using or improving our project further.
To run our program do the following terminal commands:

1. java -jar javacc-full.jar JasminParser.jj
2. javac *.java -d ../build
3. javac Jasmin/*.java -d ../build

Or to test on our sample exercises, give permissions to our run.sh script (chmod u+x run.sh) and then execute the script.

Retrospective

Looking back on the developed project, we weren't able to do all the work we were assigned for, but on the other hand, our project could be the foundation of a very useful tool for java developers, and with some more work we could maybe improve the current jvm assembler tools.

Working on this project gave us a better understanding of tools like Jasmin and JavaCC that was crucial for our performance on one of our main courses, the Compilers course. Since this project is different from our usual projects, we had the opportunity to get some insight on working and organizing freely as a team, this forced us to develop better team work, team organization and as leaders.

We had various problems throughout the development, being the major one time management, trying to sort our schedules in a way that we all could have meetings was very hard, this was a big setback cause it made things harder to develop and slowed our progress down. In terms of development there were also some struggles but we were capable of solving them, most of the time by researching and studying some of the manuals and code snippets provided by our teacher advisor.

Future work

Our project has a lot of room for improvement, from optimization to features, our project is very versatile and we feel like we can do even more with it. Some features that we would like to develop in the future are:

Better Error Handling

The current Jasmin error handling is very vague and lacks usability, to solve this problem we want to create a better error handling system that gives better warnings and error messages that are more direct and helpful for the user.

Debug Options

Something that we thought that could be interesting to develop, and very helpful for both students and developers, is improved debug options on our assembler, for example, giving the user different options like print the current stack content, or print the number of arithmetic operations in the program.

Conclusion

The main goal of this project was to build a new JVM Assembler, to make this possible we had to go through different phases of the project. We started by the development of a parser and ended with code generation. That said, we are still missing key functionalities to make this tool more useful. In conclusion, even with the missing part, we think that our project was successful, not only because we were able to show a good understanding of the JVM Assembler, Jasmin and JavaCC without prior experience, and also because we developed soft skills like team leading and organizing as a team.

References

1. Meyer, Jonathan. “About Jasmin.” Source Forge, July 1996, http://jasmin.sourceforge.net/about.html. Accessed 11 06 2022.
2. Roberts, David A. “Jasmin.” GitHub, 2015, https://github.com/davidar/jasmin. Accessed 11 06 2022.
3. Viswanadha, Sreeni, e Sriram Sankar. JavaCC. https://javacc.github.io/javacc/. Accessed 12 06 2022.
4. Key Differences between JavaCC 21 and Legacy JavaCC. 2021, https://doku.javacc.com/doku.php?id=key_differences. Accessed 12 06 2022.
5. Lindholm, Tim, et al. The Java Virtual Machine Specification. Java SE 14 Edition, 2020, https://docs.oracle.com/javase/specs/jvms/se14/jvms14.pdf. Accessed 15 06 2022.

Annex

Tokens

All the tokens used in our .jj file

```
TOKEN :
    < INTEGER : (["0"-"9"])+ > |
    < LONG : (["0"-"9"])+ > |
    < CLASS : ".class" > |
    < PUBLIC : "public" > |
    < SUPER : ".super" > |
    < SEMICOLON : ";" > |
    < LINE_COMMENT: ";" (~["\n", "\r"])* ("\n" | "\r" | "\r\n")? > |
    < DOUBLE_DOT : ":" > |
    < DOT : "." > |
    < METHOD : "method" > |
    < DOT_METHOD : ".method" > |
    < END : ".end" > |
    < STATIC : "static" > |
    < INT : "I" > |
    < VOID : "V" > |
    < ARRAY_IDENTIFIER : "[L"> |
    < LIMIT : ".limit" > |
    < SLASH: "/" > |
    < OPEN_PAREN : "(" > |
    < CLOSE_PAREN : ")" > |
    < LESS : "<" > |
    < GREAT : ">" > |
    < LOCALS : "locals" > |
    < STACK : "stack" > |
    < GO_TO : "goto" > |
    < GO_TO_W : "goto_w" > |
    < A_NEW_ARRAY: "anewarray" > |
    < ARRAY_LENGTH: "arraylength"> |
    < ARETURN: "areturn" > |
    < AALOAD: "aaload" > |
    < AASTORE: "aastore" > |
    < ACONST_NULL: "aconst_null" > |
    < ALOAD : "aload" > |
    < ALOAD_0: "aload_0" > |
    < ALOAD_1: "aload_1" > |
    < ALOAD_2: "aload_2" > |
    < ALOAD_3: "aload_3" > |
    < BALOAD: "baload" > |
< CALOAD: "caload" > |
< CASTORE: "castore" > |
< CHECKCAST : "checkcast" > |
< D2F : "d2f" > |
< D2I : "d2i" > |
< D2L : "d2l" > |
< DADD: "dadd" > |
< DALOAD: "daload" > |
< DASTORE: "dastore" > |
< DCMPG : "dcmpg" > |
< DCMPL : "dcmpl" > |
< DCONST_0 : "dconst_0" > |
< DCONST_1 : "dconst_1" > |
< DDIV : "ddiv" > |
< DLOAD: "dload" > |
< DLOAD_0: "dload_0" > |
< DLOAD_1: "dload_1" > |
< DLOAD_2: "dload_2" > |
< DLOAD_3: "dload_3" > |
< DMUL: "dmul" > |
< DNEG: "dneg" > |
< DREM : "drem" > |
< DRETURN: "dreturn" > |
< DSTORE : "dstore" > |
< DSTORE_0: "dstore_0" > |
< DSTORE_1: "dstore_1" > |
< DSTORE_2: "dstore_2" > |
< DSTORE_3: "dstore_3" > |
< DSUB: "dsub" > |
< DUP: "dup" > |
< DUP_X1: "dup_x1" > |
< DUP_X2: "dup_x2" > |
< DUP2: "dup2" > |
< DUP2_X1: "dup2_x1" > |
< DUP2_X2: "dup2_x2" > |
< F2D: "f2d" > |
< ATHROW: "athrow" > |
< ASTORE : "astore" > |
< ASTORE_1: "astore_1" > |
< ASTORE_2: "astore_2" > |
< ASTORE_3: "astore_3" > |
< BASTORE: "bastore" > |
< ICONST : "iconst" > |
< ICONST_0 : "iconst_0" > |
< ICONST_1 : "iconst_1" > |
< ICONST_2 : "iconst_2" > |
< ICONST_3 : "iconst_3" > |
< ICONST_4 : "iconst_4" > |
< ICONST_5 : "iconst_5" > |
< ISTORE : "istore"> |
< ISTORE_1: "istore_1" > |
< ISTORE_2 : "istore_2" > |
< ISTORE_3 : "istore_3" > |
< ILOAD : "iload" > |
< ILOAD_0 : "iload_0" > |
< ILOAD_1 : "iload_1" > |
< ILOAD_2 : "iload_2" > |
< ILOAD_3 : "iload_3" > |
< IF_ACMPEQ : "if_acmpeq" > |
< IF_ACMPNE : "if_acmpne" > |
< IF_ICMPEQ : "if_icmpeq" > |
< IF_ICMPNE : "if_icmpne" > |
< IF_ICMPLT : "if_icmplt" > |
< IF_ICMPGE : "if_icmpge" > |
< IF_ICMPGT : "if_icmpgt" > |
< IF_ICMPLE : "if_icmple" > |
< IFEQ : "ifeq" > |
< IFNE : "ifne" > |
< IFLT : "iflt" > |
< IFGE : "ifge" > |
< IFGT : "ifgt" > |
< IFLE : "ifle" > |
< IFNONNULL : "ifnonnull" > |
< IFNULL : "ifnull" > |
< IADD : "iadd" > |
< IINC : "iinc" > |
< NEW : "new" > |
< BIPUSH : "bipush" > |
< F2I : "f2i" > |
< F2L : "f2l" > |
< FADD : "fadd" > |
< FALOAD : "faload" > |
< FASTORE : "fastore" > |
< FCMPG : "fcmpg" > |
< FCMPL : "fcmpl" > |
< FCONST : "fconst" > |
< FCONST_0 : "fconst_0" > |
< FCONST_1 : "fconst_1" > |
< FCONST_2 : "fconst_2" > |
< FCONST_3 : "fconst_3" > |
< FDIV : "fdiv" > |
< FLOAD : "fload" > |
< FLOAD_0 : "fload_0" > |
< FLOAD_1 : "fload_1" > |
< FLOAD_2 : "fload_2" > |
< FLOAD_3 : "fload_3" > |
< FMUL : "fmul" > |
< FNEG : "fneg" > |
< FREM : "frem" > |
< FRETURN : "freturn" > |
< FSTORE : "fstore" > |
< FSTORE_0 : "fstore_0" > |
< FSTORE_1 : "fstore_1" > |
< FSTORE_2 : "fstore_2" > |
< FSTORE_3 : "fstore_3" > |
< FSUB : "fsub" > |
< GETFIELD : "getfield" > |
< GETSTATIC : "getstatic" > |
< I2B : "i2b" > |
< I2C : "i2c" > |
< I2D : "i2d" > |
< I2F : "i2f" > |
< I2L : "i2l" > |
< I2S : "i2s" > |
< IALOAD : "iaload" > |
< IAND : "iand" > |
< IASTORE : "iastore" > |
< IDIV : "idiv" > |
< INEG : "ineg"> |
< IMUL : "imul" > |
< INSTANCEOF: "instanceof"> |
< INVOKEDYNAMIC : "invokedynamic"> |
< INVOKEINTERFACE: "invokeinterface"> |
< INVOKESPECIAL : "invokespecial"> |
< INVOKESTATIC: "invokestatic"> |
< INVOKEVIRTUAL : "invokevirtual"> |
< INVOKENONVIRTUAL : "invokenonvirtual"> |
< IOR : "ior"> |
< IREM : "irem"> |
< ISHL : "ishl"> |
< ISHR : "ishr"> |
< ISUB : "isub"> |
< IUSHR: "iushr"> |
< IXOR : "ixor"> |
< JSR : "jsr"> |
< JSR_W : "jsr_w"> |
< L2D : "l2d"> |
< L2F : "l2f"> |
< L2I : "l2i"> |
< LADD : "ladd"> |
< LALOAD : "laload"> |
< LAND : "land"> |
< LASTORE : "lastore"> |
< LCMP : "lcmp"> |
< LCONST_0 : "lconst_0"> |
< LCONST_1 : "lconst_1"> |
< LDC : "ldc"> |
< LDC_W : "ldc_w"> |
< LDC2_W : "ldc2_w"> |
< LDIV : "ldiv"> |
< LLOAD : "lload"> |
< LLOAD_0 : "lload_0"> |
< LLOAD_1 : "lload_1"> |
< LLOAD_2 : "lload_2"> |
< LLOAD_3 : "lload_3"> |
< LMUL : "lmul"> |
< LNEG : "lneg"> |
< LOOKUPSWITCH : "lookupswitch"> |
< LREM: "lrem"> |
< LRETURN: "lreturn"> |
< LSHL: "lshl"> |
< LSHR: "lshr"> |
< LOR: "lor"> |
< LSTORE: "lstore"> |
< LSTORE_0: "lstore_0"> |
< LSTORE_1: "lstore_1"> |
< LSTORE_2: "lstore_2"> |
< LSTORE_3: "lstore_3"> |
< LSUB: "lsub"> |
< LUSHR: "lushr"> |
< LXOR: "lxor"> |
< MONITORENTER: "monitorenter"> |
< MONITOREXIT: "monitorexit"> |
< MULTIANEWARRAY: "multianewarray"> |
< NOP: "nop"> |
< POP: "pop"> |
< POP_2: "pop_2"> |
< PUTFIELD: "putfield"> |
< PUTSTATIC: "putstaic"> |
< RET: "ret"> |
< SALOAD: "saload"> |
< SASTORE: "sastore"> |
< SIPUSH: "sipush"> |
< SWAP: "swap"> |
    < TABLESWITCH: "tabelswitch"> |
    < WIDE: "wide"> |
    < IRETURN : "ireturn" > |
    < RETURN : "return" >
;

TOKEN:
<QUOTED:
    "\""
    (
         "\\" ~[]      //any escaped character
    |                  //or
        ~["\"","\\"]  //any character except quote or backslash
    )*
    "\"" >
;

TOKEN:
      < IDENTIFIER : (["A"-"Z","a"-"z"])(["0"-"9","A"-"Z","a"-"z","_"])*
>
;
```

Grammar Rules

Rules we used in our .jj file.

```
Start:
    (<LINE_COMMENT>)* <CLASS> <PUBLIC> <IDENTIFIER>
{jasmin.setClassName(getToken(0).getImage());}
    (<LINE_COMMENT>)* (SuperDeclaration)?
    (<LINE_COMMENT>)* (MethodDeclaration)* (<LINE_COMMENT>)*
;

ClassDeclaration:
      <CLASS> <PUBLIC> <IDENTIFIER>
;

SuperDeclaration:
      <SUPER> <IDENTIFIER>
{jasmin.setSuperClassName(getToken(0).getImage());}
    (<SLASH> {jasmin.setSuperClassName(getToken(0).getImage());}
    <IDENTIFIER> {jasmin.setSuperClassName(getToken(0).getImage());} )*
;
```

```
Label:
    <IDENTIFIER> <DOUBLE_DOT>;

Path1:
    <IDENTIFIER> (<SLASH> | <DOT>) ((Path1) | (MethodName)
(MethodArguments));

MethodDeclaration:
    (StartMethod)  (MethodName) {if(getToken(0).getImage().equals(">"))
jasmin.setMethodName("<init>");
    else jasmin.setMethodName(getToken(0).getImage());}
     (MethodArguments) (<LINE_COMMENT>)* (MethodBody)* (<LINE_COMMENT>)*
EndMethod;

StartMethod:
    <DOT_METHOD> <PUBLIC>
{jasmin.setMethodAccessModifier(getToken(0).getImage());}
     (<STATIC>)? {if(getToken(0).getImage().equals("static"))
jasmin.setMethodStatic();}
;

MethodName:
    (<LESS> <IDENTIFIER> <GREAT>|
    <IDENTIFIER>);

MethodArguments:
    <OPEN_PAREN> (MethodArgumentNormalType)* (MethodArgumentArrayType)
    (<CLOSE_PAREN>  (ReturnTypeNormalType | ReturnArray) |
<LINE_COMMENT> {jasmin.setParameter(getToken(0).getImage());});

MethodArgumentNormalType:
    <INT> {jasmin.setParameter(getToken(0).getImage());};

MethodArgumentArrayType:
    (<ARRAY_IDENTIFIER>{jasmin.setParameter(getToken(0).getImage());})?
      (<IDENTIFIER> {jasmin.setParameter(getToken(0).getImage());}
       | <SLASH>{jasmin.setParameter(getToken(0).getImage());})*;

ReturnTypeNormalType:
    <INT> | <VOID>;

ReturnArray:
    (<ARRAY_IDENTIFIER>
{jasmin.addOperandInstPath(getToken(0).getImage());})?
    (<IDENTIFIER> {jasmin.addOperandInstPath(getToken(0).getImage());} |
    <SLASH> {jasmin.addOperandInstPath(getToken(0).getImage());})*
    (<LINE_COMMENT>
{jasmin.addOperandInstPath(getToken(0).getImage());})?;

MethodBody:
    (<LINE_COMMENT>)* (Instruction |LimitInstructions| Label)
(<LINE_COMMENT>)* ;

InvokeMethodName:
    (<LESS> {jasmin.addOperandInstPath(getToken(0).getImage());}
    <IDENTIFIER> {jasmin.addOperandInstPath(getToken(0).getImage());}
    <GREAT> {jasmin.addOperandInstPath(getToken(0).getImage());}|
    <IDENTIFIER> {jasmin.addOperandInstPath(getToken(0).getImage());});

InvokeArgument:
    (<IDENTIFIER> {jasmin.addOperandInstPath(getToken(0).getImage());} |
    <SLASH> {jasmin.addOperandInstPath(getToken(0).getImage());} )*
    (<DOT> {jasmin.addOperandInstPath(getToken(0).getImage());})?
    (InvokeMethodName) (InvokeMethodArguments);

InvokeMethodArguments:
    <OPEN_PAREN> {jasmin.addOperandInstPath(getToken(0).getImage());}
    (InvokeMethodArgumentNormalType)*
    (InvokeMethodArgumentArrayType)
    (<CLOSE_PAREN> {jasmin.addOperandInstPath(getToken(0).getImage());}
    (ReturnTypeNormalType
{jasmin.addOperandInstPath(getToken(0).getImage());}  | ReturnArray) |
    <LINE_COMMENT>
{jasmin.addOperandInstPath(getToken(0).getImage());});

InvokeMethodArgumentNormalType:
    <INT> {jasmin.addOperandInstPath(getToken(0).getImage());};

InvokeMethodArgumentArrayType:
    (<ARRAY_IDENTIFIER>
{jasmin.addOperandInstPath(getToken(0).getImage());})?
      (<IDENTIFIER> {jasmin.addOperandInstPath(getToken(0).getImage());}
       | <SLASH> {jasmin.addOperandInstPath(getToken(0).getImage());})*
;

NewInst:
    (<IDENTIFIER> {jasmin.addOperandInstPath(getToken(0).getImage());} |
    <SLASH> {jasmin.addOperandInstPath(getToken(0).getImage());})*;

LdcInst:
    (<QUOTED> {jasmin.addOperandInst(getToken(0).getImage());}
     | <INTEGER> {jasmin.addOperandInst(getToken(0).getImage());});
```

```
CheckCastInst:
    (<IDENTIFIER> {jasmin.addOperandInstPath(getToken(0).getImage());}
    (<SLASH> {jasmin.addOperandInstPath(getToken(0).getImage());}|
    <DOT> {jasmin.addOperandInstPath(getToken(0).getImage());})?)*;

NewArrayInst:
    (<ARRAY_IDENTIFIER>
{jasmin.addOperandInstPath(getToken(0).getImage());})?
    (<IDENTIFIER> {jasmin.addOperandInstPath(getToken(0).getImage());}
    (<SLASH> {jasmin.addOperandInstPath(getToken(0).getImage());}|<DOT>
{jasmin.addOperandInstPath(getToken(0).getImage());})?)*
    (<SEMICOLON> {jasmin.addOperandInstPath(getToken(0).getImage());})?;

FieldInst:
    (<IDENTIFIER> {if(getToken(0).getImage().equals("Ljava") ||
getToken(0).getImage().equals("java"))
        jasmin.addOperandInstPath(" " + getToken(0).getImage());
    else jasmin.addOperandInstPath(getToken(0).getImage());}
    (<DOT> {jasmin.addOperandInstPath(getToken(0).getImage());})? |
    <SLASH> {jasmin.addOperandInstPath(getToken(0).getImage());})*;

LimitInstructions:
    <LIMIT> LimitStatementFactorization;

LimitStatementFactorization:
    <STACK> <INTEGER> {jasmin.setStack(getToken(0).getImage());}
    |
    <LOCALS> <INTEGER> {jasmin.setLocals(getToken(0).getImage());};

EndMethod:
      <END> <METHOD>
;
```