

# FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

MESTRADO INTEGRADO EM ENGENHARIA  
INFORMÁTICA E COMPUTAÇÃO

COMPILADORES

JUNHO 2020

---

## Compiler of the Java— language to Java Bytecodes

---



Universidade do Porto

---

Faculdade de Engenharia

**FEUP**

Gustavo Macedo Torres - up201706473

Inês Rodrigues Roque de Lacerda Marques - up201605542

Joaquim Manuel Silva Cardoso Rodrigues - up201704844

Vítor Emanuel Moreira Ventuzelos - up201706403

# Contents

<b>1</b>	<b>Project Overview</b>	<b>2</b>
<b>2</b>	<b>Implemented features</b>	<b>2</b>
2.1	Parser . . . . .	2
2.2	AST . . . . .	2
2.3	Syntactic Analysis . . . . .	2
2.4	Symbol Tables . . . . .	2
2.5	Semantic Analysis . . . . .	3
2.6	Intermediate Representation . . . . .	3
2.7	JVM Code Generation . . . . .	3
2.8	Optimizations . . . . .	3
2.8.1	-r=ij optimization . . . . .	3
2.8.2	-o optimization . . . . .	4
2.8.3	-cf optimization . . . . .	5
2.8.4	-s optimization . . . . .	6
2.8.5	-u optimization . . . . .	7
2.8.6	-b optimization . . . . .	8
2.8.7	-l=ij optimization . . . . .	9
2.8.8	Final example . . . . .	10
<b>3</b>	<b>Extra features</b>	<b>10</b>
3.0.1	Strings . . . . .	10
3.0.2	. . . . .	10
3.0.3	> . . . . .	10
3.0.4	For loop . . . . .	11
3.0.5	Constructor with parameters . . . . .	11
3.0.6	Floats . . . . .	11
3.0.7	Casting integer to float and vice-versa . . . . .	11
<b>4</b>	<b>Possible improvement</b>	<b>11</b>
4.0.1	Conditional expression evaluation optimization . . . . .	11
4.0.2	Expression propagation and condensation optimization . . . . .	11
4.0.3	Expression storing optimization . . . . .	11
4.0.4	Loop unswitching optimization . . . . .	12
4.0.5	Object arrays . . . . .	12

## 1 Project Overview

For the Compilers class, the students were asked to develop a tool, named *jmm*, that compiles Java- (a subset of the Java language) to Java bytecodes. This compiler follows a well defined compilation flow, capable of performing syntactic and semantic analysis as well as code generation.

The tool starts by constructing an AST with the structures and expressions present in the code analyzing a code file in the Java- language and does the respective parsing, checking for the existence of incorrect grammar. After this process a symbol table is created and a semantic analysis of the code is done. To aid in the development of our tool, the compiler goes through a stage of intermediate representation. If no errors were found, the tool outputs the generated code to a file with the “.j” extension.

## 2 Implemented features

### 2.1 Parser

On the first stage, a parser was implemented in JavaCC following the given grammar, which can be found in */javacc/jmm.jtt*. A general lookahead of 1 was used, while a local lookahead of 2 was used in the parsing of a method’s body(line 163), statements(line 188) and expressions(lines 240-286). This parser was also altered to accept grammar relative to the extra features implemented.

### 2.2 AST

The previously described parser was modified to be used in JJTree, in order to generate an Abstract Syntax Tree representing the parsed program. Its implementation can be found in the *javacc* folder.

### 2.3 Syntactic Analysis

Syntactic analysis is incorporated in the parser, where any incorrect grammar not accepted by the parser is registered and reported to the user, while compilation is aborted. Within ”while” loops, up to 10 grammar errors will be reported before aborting compilation.

### 2.4 Symbol Tables

Once the AST is successfully generated, the tree’s root’s *createTable* method is called, which generates the current node’s Symbol Tables and calls the method for its child nodes. The Symbol Table’s implementation can be found in */src/SymbolTable.java*, while Descriptor Tables (*/src/DescriptorTable.java*) and Function Tables (*/src/FunctionTable.java*) were also developed to facilitate the calculation of scopes and the semantic analysis. The multiple variations of the *createTable* method can be found in the .java files within the *javacc* folder.

## 2.5 Semantic Analysis

If no exceptions occur during the generation of the Symbol Tables, the AST's root's `doSemanticAnalysis` method is called, which calls the same method for its child nodes and checks for semantic errors related to the content of the current node. A maximum number of 10 semantic errors will be registered before compilation is forcibly aborted. The implementation of the analysis can be found in the `.java` files within the *javacc* folder.

## 2.6 Intermediate Representation

We decided to generate an intermediate representation, a data structure used internally by the compiler, to aid in the analysis of input code. This representation is made after both the semantic and syntax are complete. Also, the IR helps us structure the Java- code in something more simpler and manageable. It will also help us in the optimizations of the code generation part of the project. We also did the Control Flow Graph for liveness analysis allowing us to do the register allocation optimization. Their implementations can be found in `/src/CFGNode.java`, `/src/IRNode.java` and `/src/IRBuilder.java`.

## 2.7 JVM Code Generation

If there aren't any errors the code for the specified file is generated. This process is done by the *Jasmin* class that receives the generated IR and a *Printstream* with the specified file for which the final code will be printed. It can be found in `/src/Jasmin.java`. The generated code can be found in the *jasmin.j* file placed in the root folder of the project.

## 2.8 Optimizations

### 2.8.1 `-r={n}` optimization

For this optimization we used the Control Flow Graph, where firstly we built the Use and Def sets and then did a backward liveness analysis in order to be able to compute each variable live range. In this computation we included each local variable, each parameter and the *this* in case of a non static method. From here we were able to build an interference graph and then do the coloring of the graph as described in the slides. Then we parse each statement in the method so we could assign each variable to its new and optimized register. We can also detect variables when a variable is not used from the moment it is assigned, thus, not even assigning it to a register and doing a `pop` instead. The *n* variable stands for the maximum number of registers we want to use, and in case *n* is too low, the compiler returns the minimum *n* for the current program. For our implementation, *n* represents the maximum register available to use, so with a *n* of 1 we can use register 0 and 1, because in the majority of times register 0 is reserved for the current object.

```

public int optimization(int a){
    int b;
    int c;

    b = a;
    c = b + 4;
    a = c * c;
    c = 4;
    return a;
}

38 .method public optimization(I)I
39   .limit stack 2
40   .limit locals 2
41
42   iload_1
43   istore_1
44
45   iinc 1 4
46
47   iload_1
48   iload_1
49   imul
50   istore_1
51
52   iconst_4
53   pop
54
55   iload_1
56   ireturn
57
58 .end method

```

Figure 1: Use of optimization -r=1

In this example the compiler did the liveness analysis and found that none of the live ranges of each variable intersected, so they could all share the same variable. It also found that the last assignment to *c* was not used, so it does not interfere with *a*, assuming there is no need to store the variable, and a pop is more efficient. This code, in this situation, does not affect the program and can be removed in an optimization described further, optimization -u.

### 2.8.2 -o optimization

In this optimization we did a constant propagation, in order to avoid unnecessary register loads. For this we kept a list of each variable as it is assigned in the code. In the case of a loop, the variables assigned inside it were removed from the list before doing the propagation, and the algorithm is then applied as before. In case of an if, the propagation is applied in each body of the if and else statements, and then each variable assigned inside the if and else body is removed from the list. This optimization also does an optimization known as loop inversion, where an if is done in the beginning of the loop to check if it will execute or not, and the condition is moved to the end, transforming the while into an if with a do-while inside. However, if it is possible to know if the loop will execute at least one or not even one time, the if will be removed from in the first case or the whole while will be removed in the second one.

```

public int optimization(){
    int a;
    int b;

    a = 1;
    if(a < 6){
        a = 2;
    }else{
        a = 5 + a;
    }

    return a;
}

```

```

38 .method public optimization()I
39     .limit stack 2
40     .limit locals 3
41
42 iconst_1
43 istore_1
44
45 iconst_1
46 bipush 6
47 if_icmpge else_if2
48
49 iconst_2
50 istore_1
51
52 goto end_if2
53
54 else_if2:
55 iconst_5
56 iconst_1
57 iadd
58 istore_1
59
60 end_if2:
61
62 iload_1
63 ireturn
64
65 .end method

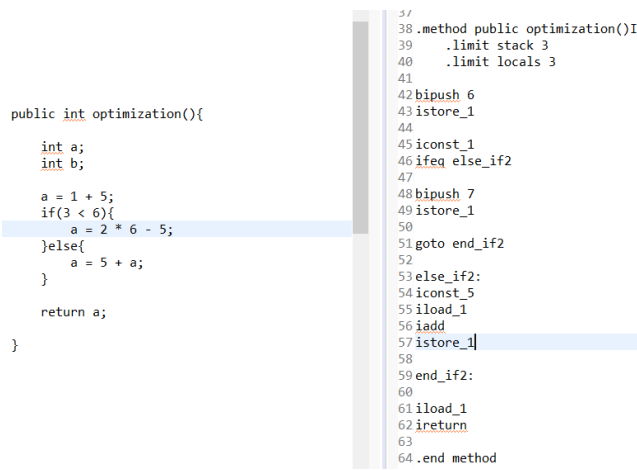
```

Figure 2: Use of optimization -o

In this image we can see that the value of variable *a* is propagated to the places where the variable is used. However after the if it is no longer propagated because the variable is defined in at least one of the if branches, so we can no longer be sure of its values.

### 2.8.3 -cf optimization

This optimization is known as constant folding, or in other words, simplifying and evaluating expression or sub-expression made of constant values at compile time, thus, improving overall run-time of the program. It also reduces program size and need for run-time expression evaluation. We can see real improvements on code generation when using this optimization with -o optimization.



```

public int optimization(){
    int a;
    int b;

    a = 1 + 5;
    if(3 < 6){
        a = 2 * 6 - 5;
    }else{
        a = 5 + a;
    }

    return a;
}

```

```

37
38 .method public optimization()I
39   .limit stack 3
40   .limit locals 3
41
42 bipush 6
43 istore_1
44
45 iconst_1
46 ifeq else_if2
47
48 bipush 7
49 istore_1
50
51 goto end_if2
52
53 else_if2:
54 iconst_5
55 iload_1
56 iadd
57 istore_1
58
59 end_if2:
60
61 iload_1
62 ireturn
63
64 .end method

```

Figure 3: Use of optimization -cf

With this optimization we can see that code size has decreased, leading to smaller programs and more efficient expression computation, because constant expressions are evaluated at compile time.

#### 2.8.4 -s optimization

This optimization joins integer divide and integer multiply optimizations. Both of them consist in using arithmetic shifts to do the corresponding operation whenever one of the operands is a power of two in case of a multiplication or only the right in case of a division. These instructions are faster than division and multiplication improving the run-time of the program.

```

public int optimization(){
    int a;
    a = 0;
    if(3 * 32 < 6 * 31){
        a = 8 * a;
    }else{
        a = a / 2;
    }
    return a;
}

```

```

38.method public optimization()I
39    .limit stack 3
40    .limit locals 2
41
42.iconst_0
43.istore_1
44
45.iconst_3
46.iconst_5
47.ishl
48.bipush 6
49.bipush 31
50.imul
51.if_icmpge else_if2
52
53.iload_1
54.iconst_3
55.ishl
56.istore_1
57
58.goto end_if2
59
60 else_if2:
61 iload_1
62 iconst_1
63 ishr
64 istore_1
65
66 end_if2:
67
68 iload_1
69 ireturn
70
71 .end method

```

Figure 4: Use of optimization -s

As we can see, the use of multiplications and division involving a constant power of two, the compiler uses right and left shift as they are more efficient than an explicit multiplication or division operation, improving run-time.

### 2.8.5 -u optimization

This optimization removes unnecessary code and expressions, with focus on arithmetic expression. For example statements where there is only a load of one integer or one variable, it does not make any change in the program flow, so instead of a load and a pop, we can remove entirely this statement. This optimization works better with previous optimization, like -r, -o, -cf, we can remove entire statements of the type  $a = 4 + 5$ , if  $a$  is not used after the moment it is defined.



```

public int optimization(){
    int a;
    a = 0;
    if(3 * 32 < 6 * 31){
        10;
    }else{
        6 + 8;
    }
    a;
    return a;
}

```

```

38 .method public optimization()I
39     .limit stack 2
40     .limit locals 2
41
42     iconst_0
43     istore_1
44
45     iconst_1
46     ifeq else_if2
47
48     goto end_if2
49
50 else_if2:
51
52     end_if2:
53
54
55
56     iload_1
57     ireturn
58
59 .end method
60

```

Figure 5: Use of optimization -u and -cf

In this example we can see that any expression and variable that does not make any change in the program are removed, we used -cf to evaluate the expression in the else body, so we could remove it if possible.

### 2.8.6 -b optimization

This optimization removes unnecessary if branches when knowing the value of the if condition at compile time, by removing the condition evaluation and the unreachable code blocks and no longer used goto's. It also works with whiles, when a while will always execute we can remove the condition checking, or remove the while if it will never execute, improving code performance and size. This optimization works better with -cf and -o optimizations.

```

public int optimization(){
    int a;
    a = 0;
    if(false){
        a = 1;
    }else{
        if(true)
            a = 2;
        else
            a = 3;
    }
    return a;
}

```

```

39 .method public optimization()I
40     .limit stack 1
41     .limit locals 2
42
43
44     iconst_0
45     istore_1
46
47     iconst_2
48     istore_1
49
50     iload_1
51     ireturn
52
53 .end method
54
55
56

```

Figure 6: Use of optimization -b

As we can see it is possible to predict the exact program flow, as the compiler is able to know beforehand which branches will be chosen, so we can remove the condition evaluation and the unused code of the branches never visited.

### 2.8.7 -l= $n$ optimization

This optimization is known as loop unrolling. Here we do a simple loop unrolling, where instead of performing the code inside the loop and then a jump, checking the condition before or after the jump, we replicate the code inside the loop  $n$  times, followed by the condition, if the condition fails we jump to the end of the loop. With these optimizations we are able to reduce the number of jumps inside the loop by a factor of  $(n-1)/n$ .

```

public int optimization(){
    int a;
    a = 0;
    while(a < 5){
        a = a + 1;
    }
    return a;
}

38.method public optimization()I
39    .limit stack 2
40    .limit locals 2
41
42    iconst_0
43    istore_1
44
45    loop1:
46    iload_1
47    iconst_5
48    if_icmpge end_loop1
49    begin_loop1:
50
51    iinc 1 1
52
53    iload_1
54    iconst_5
55    if_icmpge end_loop1
56    iinc 1 1
57
58    goto loop1
59    end_loop1:
60
61    iload_1
62    ireturn
63
64    .end method

```

Figure 7: Use of optimization -l=2

Here we can see that the inner loop instruction was replicated, alongside with the condition, avoiding extra jumps to be the first condition. In this example we used a  $n$  of 2, so the instruction appears two times.

### 2.8.8 Final example

```

public int optimization(int a){
    int b;
    int c;

    b = 4;
    c = 3 + b;

    if(c < 5){
        b = a + 5;
        a = 4;
    }else{
        b = a - 5;
        b = b + 1;
        a = 5;
    }
    c = c + 3;
    return b + c;
}

```

```

43
44 .method public optimization(I)I
45     .limit stack 2
46     .limit locals 2
47
48
49
50 iinc 1 -5
51
52 iinc 1 1
53
54
55
56 iload_1
57 bipush 10
58 iadd
59 ireturn
60
61 .end method
62
63

```

Figure 8: Use of optimization -r=1 -cf -o -u -b

In this example we can see almost all optimizations working together to achieve the following result. As we can see the value of *b* was propagated to *c* and then a constant folding was performed, allowing the propagation of *c*, leading to the evaluation of the if condition, returning a false. Then we knew that the else branch was going to be chosen, so we could remove the if branch and stick with the else one. From here, the compiler performed a liveness analysis and knew that *a* would not interfere with variable *b*, so they could be stored in the same register. The assign of the value 5 to *a* is never used, so we can simply remove it from the program. Finally, the value of *c* was not defined in the if statement, so it was propagated beyond that, only leaving the final return statement.

## 3 Extra features

### 3.0.1 Strings

We decided to implement strings as we consider it to be a fundamental data type to any programming language. The usage of the strings is based only on printing because the string operations are not implemented. Also, the characters allowed are mainly the alphanumerical ones.

### 3.0.2 ||

We decided to implement the logical OR, in order to simplify conditional and boolean expression generation by the programmer and make his task easier.

### 3.0.3 >

We decided to implement the greater than symbol, in order to be able to compare easier if a value is greater than another.

### **3.0.4 For loop**

We implemented for loops, allowing to make more compact and readable code for loop statements. The compiler, internally, then transforms the for loop into a while loop with an assign to the variable before and a variable update/increment to a variable in the end of the loop.

### **3.0.5 Constructor with parameters**

A class constructor with parameters was also implemented to make it easier to create objects with different arguments. With this feature, we are able to adapt the object to the user's needs instead of using set functions that can take more resources and increase the run-time of the program.

### **3.0.6 Floats**

The compiler can handle the float data type and can be used for arithmetic and logical operations.

### **3.0.7 Casting integer to float and vice-versa**

Casting from integer to float and from float to integer. The casting is only available in variable assign, not during operations.

## **4 Possible improvement**

### **4.0.1 Conditional expression evaluation optimization**

Sometimes it is not needed to evaluate the whole condition to evaluate if it is true or not, being useful for branch optimization/removal.

### **4.0.2 Expression propagation and condensation optimization**

Sometimes a variable is only used for computation and then used right after, however its live range make it impossible to share the register with the variable the assign is made, so propagating the expression would save a register. Also sometimes it is possible to join two expression into a more compact and efficient one, like when doing two increments in a row.

### **4.0.3 Expression storing optimization**

Sometimes an expression is used a lot, the same value is evaluated over and over again, so we would allocate one register to store this expression and just load the expression. This optimization must do a trade-off with the optimization above.

#### **4.0.4 Loop unswitching optimization**

Moving the ifs inside the loop with variables not assign inside the loop, to the outside, leading to more code, but less overhead because the if expression would be only evaluated one and not several times.

#### **4.0.5 Object arrays**

Arrays with objects other than integers would be a good implementation for the programmer.