Course Unit Compilers

Masters in Informatics and Computing Engineering (MIEIC)

Department of Informatics Engineering

University of Porto/FEUP

2º Semester – 2017/2018

Compiler of the yal0.4 language to Java Bytecodes

João M. P. Cardoso

With revisions by Luís Reis, Ricardo Nobre, and Tiago Carvalho

Table of Contents

Objectives: To apply the knowledge acquired in the course unit Compilers by building a compiler for programs in the *yal* language. The compiler shall generate valid JVM (*Java Virtual Machine*) instructions to *jasmin*, a tool to generate Java bytecodes given *assembly* programs with JVM instructions.

---

1. Introduction and Assessment

The intention of this assignment is to acquire and apply knowledge within the Compilers' course unit (3rd year MIEIC). In order to do so, you will learn how to develop a mini-compiler. The project is split into stages that essentially correspond to steps in the flow of a real compiler. It is natural and expected that in this project students face some of the problems that may be present when developing a commercial compiler. The stages of the project allow the students to manage the efforts and dedication to the project according to the level they like to achieve.

The assignment score is distributed by the following parameters:

i. 10% given to the organization and clarity of the code developed, as well as the documentation;
ii. 20% given to the solutions implemented to solve potential problems;
iii. 20% given to the optimization level performed by the compiler;
iv. 50% according to the results obtained by the functional tests in the two test suites provided:
    a. 30% obtained given the results when using the test suite given to the students to test the compiler during development;
    b. 20% obtained given the results obtained when using the test suite which is not known *a priori*.

Grades of the members within a group are typically equal. Different grades indicate potential issues, such as distribution of the work load, member's commitment, and the capability of explaining the work developed.

2. The yal2jvm Compiler

The compiler, dubbed yal2jvm, must translate programs in **yal**[1]**,** version 0.2 (we shall introduce the language in the next section), into java *bytecodes* [1]. Figure 1 the compilation flow of the

[1] There is no relation whatsoever with the programming language YAL (*Yet Another Language*).

compiler. The compiler must generate files of the classes with JVM instructions accepted by *jasmin* [2], a tool that translates those classes in Java *bytecodes* (classfiles).

The generated classes given **yal** code can be integrated in a Java application. Those classes can invoke Java methods previously compiled to *bytecodes* (the utility of this concept will be detailed later in this document).

The **yal2jvm** compiler is executed using the following notation:

```
java yal2jvm [-r=<num>] [-o] <input_file.yal>
```

or

```
java –jar yal2jvm.jar [-r=<num>] [-o] <input_file.yal>
```

Where <*input_file.yal*> is the **yal** module we would like to compile.

The "–r" option tells the compiler to use only the first <*num*>[2] local variables of the JVM when assigning the local variables used in each **yal** function to the local JVM variables. Without the "–r" option (similar to –r=0), the compiler will use the available JVM local variables to store the local variables used in each **yal** function.

With the "–o" option, the compiler should perform a set of code optimizations.

Section 4 (on page 6) details the "-r" and "-o" options.

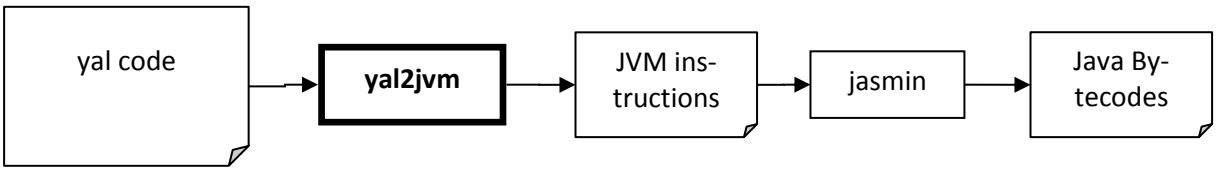

Figure 1. Compilation Flow.

The compiler shall generate a class with the name <*input_file*>.j. The .j classes will then be translated into Java *bytecode* classes (*classfiles)* using the tool *jasmin* [2].

The compiler shall include the following stages/phases: lexical analysis, syntactic analysis, semantic analysis, and code generation. The optimization stage (invoked by the "-o" option) is optional.

[2] <num> is an integer between 0 and 255.

Error Handling

An important part of any compiler is the indication of associated errors in each analysis phase. The error messages must be readable and useful. For example, in the syntactic analysis, the compiler should give more error information than the default ones reported by JavaCC. We recommend reading the document about handling and recovering from errors in JavaCC [3]. A possibility is to force the analysis to ignore the terminal symbols (tokens). For instance, if there is an error in a *while* expression, the analysis could ignore every token until "{" is found.

Recovering from errors allows the compiler to avoid that specific errors originate unnecessary ones. Note that the compiler should not abort execution immediately after the first error, and instead should report a set of errors so that the developer proceeds with their correction (e.g., the compiler could report the first 10 errors found, before aborting the execution).

Semantic Analysis

In order to reduce the workload of the semantic analysis, the compiler must only perform the semantic analysis with respect to the invocation of functions with code included in the **yal** module we intend to compile. The compiler should assume that there are no semantic errors when invoking functions that belong to other modules or classfiles.

The detected semantic errors shall be reported and the compiler shall abort execution.

Intermediate Representation

Code generation may take as input the AST[3] and the symbol table(s) of the compiled **yal** module. However, a more advanced compiler could use the three address code representation in the optimization and code generation phases.

3. The yal Language

A program in the **yal** language contains one or more modules. Each program file contains one **yal** module. Each **yal** module has a name, possibly a set of attributes, and functions. Figure 2 depicts an example of a simple **yal** program. The function *main()* is where the program starts execution[4]. The BNF (*Backus–Naur Form*) grammar of the **yal** language can be found in *Appendix A: The yal language grammar defined in* BNF (in page 11).

[3] *Abstract Syntax Tree*.

[4] This function should be implemented by the Java's main method: "public static void main(String [] args)"

```
module first_program {
       function main() {
           io.print ("Hello World");
       }
}
```

Figure 2. First program in yal.

The **yal** language does not have specific support for read/write primitives (e.g., reading from a file or keyboard). This operations can, however, be performed by using *classfiles* that provide functions such as *read(), print(...), and println(...)*. In the context of this assignment, these functions are defined in the **io** class provided (see *Appendix B: The io*, in page 11).

Main characteristics of the **yal** language:

- In **yal**, there are only integer scalar variables and one-dimensional arrays with integer elements. By default, all data are integers (in this version of the language there is no support for other data types);
- A scalar variable can only be used after having a value assigned to it (just like Java);
- Arrays must be declared before being accessed (e.g., the instruction A=[20]; declares an array with 20 elements of the integer type);
- In each **yal** function, an identifier can only be used to identify one scalar variable or one array variable (for instance, in the same function, the same name cannot be used to first identify a scalar and later to identify an array);
- It is possible to use the same identifier for a function and a variable (i.e., in an **yal** module, a function “max” may co-exist with a variable “max”);
- In function argument passing and return values, the scalar variables are passed by value and the array variables by reference;
- There is only one control flow statement that allows code to be executed repeatedly: *while*;
- Each **yal** module can be seen as a static class in Java, with static attributes and methods;
- The language does not distinguish between lower and upper case (i.e., it is *case insensitive*).

Figure 3 show another **yal** program which uses two modules: a main module and another module where two functions used in the main module are. Note that this program assumes that there is an ***io*** class providing the function to the function invoked in the **yal** code (i.e., *io.println(...)*).

file program1.yal

```
module program1 {

   data=[100]; // vector of 100 integers
   mx; // attribute mx
   mn; // attribute mn

   function det(d[]) {
      i=0;
      M=d.size-1; //d.size equivalent to d.length (Java)
      while(i<M) { // version not optimized!
         a=d[i]; i=i+1; b=d[i];
         mx= library1.max(a,b);
         mn= library1.min(a,b);
      }
   }

   function main() {
      det(data);
      io.println("max: ",mx);
      io.println("min: ",mn);
   }
}
```

(a) main module;

file library1.yal

```
module library1 {

  function m=max(a,b) {
      if(a > b) {
          m = a;
      } else {
          m = b;
      }
  }

  function m=min(a,b) {
      if(a > b) {
          m = b;
      } else {
          m = a;
      }
  }

}
```

(b) module with auxiliary functions;

Figure 3. Second yal program.

4. Optimizations and Register Allocation in the yal2jvm Compiler

The stages described in this section are necessary to be eligible for final grades between 18 and 20 (out of 20). We expect that by default (i.e., without the option "–o"), the compiler generates JVM [1] code with lower cost instructions in the cases of: *iload*, *istore*, *astore*, *aload*, loading constants to the stack, use of *iinc*, etc.

All the optimizations included in your compiler shall be identified in the **README.txt** file that shall be part of the files of the project to be submitted once completed.

Option –r=<n>

With the option "–r", and for $n \geq 1$, the compiler tries to assign the local variables used in each function of each **yal** module to the first <*n*> local variables of the JVM (or to the maximum local variables of the JVM when *n* is greater than the maximum):

- It shall report the local variables used in each function of the **yal** module that are assigned to each local variable of the JVM.

- If the value of *n* is not enough to have the required local variables of the JVM sufficient to store the variables of the yal function, the compiler shall abort execution and shall report an error and indicate the minimum number of JVM local variables required.
- This option needs the determination of the lifetime of variables using *dataflow analysis* (regarding lifetime analysis, please consult the slides of the course and/or one of the books in the bibliography). An efficient implementation of dataflow analysis uses, for *def* and *use* sets, objects of the *BitSet* Java class.
- The register allocation, sometimes also referred as "register assignment", (in this particular case, it corresponds to the assignment of the variables of a function to the first *n* local variables of the JVM) can be performed with the use of the graph coloring algorithm described in the course. However, we accept that your compiler includes a different register allocation algorithm.

Option "–o":

With the option "-o" the compiler shall include the following two optimizations:

- Identify uses of the local function variables that can be substituted by constants. It can diminish the number of local variables of the JVM used as variables with statically known constant values can be promoted to constants. Note that in this optimization known as "constant propagation" your compiler does not need to include the evaluation of expressions with constants (an optimization known as "constant folding").
- Use templates for compiling "while" loops that eliminate the use of unnecessary "goto" instructions just after the conditional branch that controls if the loop shall execute another iteration or shall terminate (the compilers that have included this optimization by default do not need to make modifications and do not need that this optimization be controlled by option "-o")

Include an additional code optimization selected by your group. Surprise us!!!!

5. Suggested Stages for the Compiler

The suggested development stages for the **yal2jvm** compiler are the following:

1. Develop a *parser* for **yal** using JavaCC and taking as starting point the yal grammar furnished (note that the original grammar may originate conflicts when implemented with

parsers of LL(1)[5] type and in that case you need to modify the grammar in order to eliminate those conflicts);
2. Include error treatment and recovery mechanisms;
3. Proceed with the specification of the file **jjt**[6] to generate, using JJTree, a new version of the *parser* including in this case the generation of the syntax tree (the generated tree should be an AST[7]), annotating the nodes and leafs of the tree with the information (including tokens) necessary to perform the subsequent compiler steps; **[*checkpoint*[8] 1]**
4. Include the necessary symbol tables;
5. Semantic Analysis[9];
6. Generate JVM code accepted by *jasmin* corresponding to the invocation of functions in **yal**;
7. Generate JVM code accepted by *jasmin* for arithmetic expressions; **[*checkpoint* 2]**
8. Generate JVM code accepted by *jasmin* for conditional instructions (*if* and *if-else*);
9. Generate JVM code accepted by *jasmin* for loops;
10. Generate JVM code accepted by *jasmin* to deal with arrays.
11. Complete the compiler and test it using a set of yal modules; **[*checkpoint* 3]**
12. Proceed with the optimizations related to the code generation, related to the register allocation ("-r" option) and the optimizations related to the "-o" option.
    [this task is necessary for students intending to be eligible for final project grades greater or equal than 18 (out of 20)]

As soon as you have the generation of the AST concluded, it is a good idea to proceed to the tasks needed to generate the JVM instructions for the example in Figure 2. Then, you can consider the more generic case related to the invocation of functions.

Note that the compiler shall report the possible errors that may occurred in the syntactic and semantic analysis.

[5] Although it is recommended, it is not strictly necessary to use *lookahead* with value 1.
[6] Basically, you can copy the file **.jj* and make the modifications so that JJTree generates the code to generate the sintax trees.
[7] *Abstract Syntax Tree*.
[8] It corresponds to the presentation of the stages already developed.
[9] The compiler shall not verify if invokations to functions with code not included in the yal module under compilation are according to the prototypes of that functions.

6. JVM Instructions and the generation of Java Bytecodes

As a first basis to learn about the JVM instructions and the Java bytecodes generated we include herein a set of examples. Figure 4 shows a simple Java class to print "Hello World".

```
public class Hello {
   public static void main(String args[]) {
        System.out.println("Hello World!");
   }
}
```

Figure 4. Java "Hello" class (file "Hello.java").

After compiling the class above with the *javac* (*javac Hello.java*) we obtain the file *Hello.class* (file with the Java bytecodes for the given input class). To see the JVM instructions and the class attributes (see Figure 5) we can execute[10]:

```
javap –c Hello
```

```
public class Hello extends java.lang.Object{
public Hello();
  Code:
   0: aload_0
   1: invokespecial #1; //Method java/lang/Object."<init>":()V
   4: return
public static void main(java.lang.String[]);
  Code:
   0: getstatic     #2; //Field java/lang/System.out:Ljava/io/PrintStream;
   3: ldc           #3; //String Hello World!
   5: invokevirtual #4; //Method java/io/PrintStream.println:(Ljava/lang/String;)V
   8:   return
}
```

Figure 5. JVM instructions in the *bytecodes* obtained after compilation of the "Hello" class with *javac*.

We can program with JVM instructions a class equivalent to the previous class. In this case, we will obtain the *bytecodes* Java from the *assembly* with JVM instructions using *jasmin*. The class equivalent to the *Hello* class (in the file *Hello.java*) is the class described in the file *Hello.j* using an *assembly* language with JVM instructions and syntax supported by *jasmin*. Figure 6 presents the code of the *Hello* class in file *Hello.j*.

[10] To obtain information about the number of local variables and the number of levels of the stack necessary for each method one can use: ***javap –verbose Hello***.

```
; class with sintax accepted by jasmin 2.3

.class public Hello
.super java/lang/Object

;
; standard initializer
.method public <init>()V
   aload_0

   invokenonvirtual java/lang/Object/<init>()V
   return
.end method

.method public static main([Ljava/lang/String;)V
   .limit stack 2
   ;.limit locals 2 ; this example does not need local variables

   getstatic java/lang/System.out Ljava/io/PrintStream;
   ldc "Hello World!"
   invokevirtual java/io/PrintStream.println(Ljava/lang/String;)V
   return
.end method
```

Figure 6. Programming of the "Hello" class (file "Hello.j") using JVM instructions in a syntax accepted by *jasmin*.

We can now generate the Java bytecodes for this class using *jasmin*:

```
java –jar jasmin.jar Hello.j
```

(this way we obtain the *Hello.class* classfile which has the same functionality as the class generated previously from the Java code)

7. References

[1] The Java Virtual Machine Specification, http://java.sun.com/docs/books/jvms/
[2] Jasmin Home Page, http://jasmin.sourceforge.net/
[3] JavaCC [tm]: Error Reporting and Recovery, https://javacc.org/tutorials/errorrecovery
[4] JavaCC, https://javacc.org/

Appendix A: The yal language grammar defined in BNF

TOKENS (TERMINALS)

```
<DEFAULT> SKIP : {
" "
| "\t"
| "\n"
| "\r"
| <"//" (~["\n","\r"])* ("\n" | "\r" | "\r\n")>
| <"/*" (~["*"])* "*" ("*" | ~["*","/"] (~["*"])* "*")* "/">
}


/* reserved words */
<DEFAULT> TOKEN : {
<RELA_OP: ">" | "<" | "<=" | ">=" | "==" | "!=">
| <ADDSUB_OP: "+" | "-">
| <ARITH_OP: "*" | "/" | "<<" | ">>" | ">>>">
| <BITWISE_OP: "&" | "|" | "^">
| <NOT_OP: "!">
| <WHILE: "while">
| <IF: "if">
| <ELSE: "else">
| <ASSIGN: "=">
| <ASPA: "\"">
| <LPAR: "(">
| <RPAR: ")">
| <VIRG: ",">
| <PVIRG: ";">
| <LCHAVETA: "{">
| <RCHAVETA: "}">
| <FUNCTION: "function">
| <MODULE: "module">
| <SIZE: "size">
}


<DEFAULT> TOKEN : {
<INTEGER: (<DIGIT>)+>
| <ID: <LETTER> (<LETTER> | <DIGIT>)*>
| <#LETTER: ["$","A"-"Z","_","a"-"z"]>
| <#DIGIT: ["0"-"9"]>
| <STRING: "\"" (["a"-"z","A"-"Z","0"-"9",":"," ","="])+ "\"">
}
```

NON-TERMINALS

```
Module       ::= <MODULE> <ID> <LCHAVETA> ( Declaration )* ( Function )* <RCHA-
                 VETA>
Declaration  ::= ( ArrayElement | ScalarElement ) ( <ASSIGN> ( ( "[" ArraySize "]"
                 ) | ( <ADDSUB_OP> )? <INTEGER> ) )? <PVIRG>
Function     ::= ( ( <FUNCTION> ( ArrayElement | ScalarElement ) <ASSIGN> <ID>
                 <LPAR> ( Varlist )? <RPAR> ) | ( <FUNCTION> <ID> <LPAR> ( Varlist
                 )? <RPAR> ) ) <LCHAVETA> Stmtlst <RCHAVETA>
Varlist      ::= ( ArrayElement | ScalarElement ) ( <VIRG> ( ArrayElement | Sca-
                 larElement ) )*
ArrayElement ::= <ID> "[" "]"
Sca-         ::= <ID>
larElement
Stmtlst      ::= ( Stmt )*
Stmt         ::= While
             |   If
             |   Assign
             |   Call <PVIRG>
Assign       ::= Lhs <ASSIGN> Rhs <PVIRG>
Lhs          ::= ArrayAccess
             |   ScalarAccess
Rhs          ::= ( Term ( ( <ARITH_OP> | <BITWISE_OP> | <ADDSUB_OP> ) Term )? )
             |   "[" ArraySize "]"
ArraySize    ::= ScalarAccess
             |   <INTEGER>
Term         ::= ( <ADDSUB_OP> )? ( <INTEGER> | Call | ArrayAccess | ScalarAccess
                 )
Exprtest     ::= <LPAR> Lhs <RELA_OP> Rhs <RPAR>
While        ::= <WHILE> Exprtest <LCHAVETA> Stmtlst <RCHAVETA>
If           ::= <IF> Exprtest <LCHAVETA> Stmtlst <RCHAVETA> ( <ELSE> <LCHAVETA>
                 Stmtlst <RCHAVETA> )?
Call         ::= <ID> ( "." <ID> )? <LPAR> ( ArgumentList )? <RPAR>
ArgumentList ::= Argument ( <VIRG> Argument )*
Argument     ::= ( <ID> | <STRING> | <INTEGER> )
ArrayAccess  ::= <ID> "[" Index "]"
ScalarAccess ::= <ID> ( "." <SIZE> )?
Index        ::= <ID>
             |   <INTEGER>
```

Appendix B: The io module

To test the compiler, we will use the **io** module provided as Java *bytecodes*:

io

```
int read(); // reads an integer from the keyboard

void print(String, int); // prints on the screen a String followed by an int

void print(String); // prints on the screen a String

void println(String, int); // prints on the screen a String followed by an
int and carriage return

void println(String); // prints on the screen a String and a carriage re-
turns

void println(); // prints a carriage return on the screen
```

The equivalent Java class (*io.*class) to this library is provided to this assignment. The prototype methods are as follows.

```
public static int read()

public static void print(String, int)

public static void print(String)

public static void print(int)

public static void println(String, int)

public static void println(String)

public static void println(int)

public static void println()
```