apl. Prof. Dr. Thomas Noll
Christoph Matheja, Matthias Volk

Compiler Construction 2016

— Series 8 —

Hand in until July 12th before the exercise class.

General Remarks

- Follow the naming convention for the zip file: `ex8_MATRNO1_MATRNO2_MATRNO3` and include the complete framework provided to you via our webpage, i.e. the top level directory is the directory `While2JasminCompiler` and especially the `bin` directory is present.
- It is allowed to hand in your solutions for the theoretical part via email **as a separately attached PDF file**. If you hand in solutions via email please make sure that the size is reasonable (< 1 MB). Solutions that our printer rejects to print within 30min due to large scanned images will not be corrected.
- Please hand in your solutions in groups of 3 or 4.

Exercise 1 (1 Points)

Which of the following procedure stacks could be result of the execution of an EPL-programm? Why?

(a) $p_1 = 13:3:9:1:4:3:2:2:4:5:5:15:1:3:2:12:0:0:0:17:3$

(b) $p_2 = 13:3:9:1:4:3:2:2:5:4:5:15:1:3:2:12:0:0:0:17:3$

(c) $p_3 = 13:3:9:1:4:3:2:2:8:4:5:15:1:3:2:12:0:0:0:17:3$

Exercise 2 (2 Points)

Consider the following intermediate code:

```
      ⋮
 7:   LOAD(1, 2);       (dif, off)
 8:   ADD;
 9:   RET;
10:   LOAD(2, 2);       (dif, off)
      ⋮
25:   CALL(7, 1, 2);    (ca, dif, loc)
26:   CALL(38, 1, 3);   (ca, dif, loc)
27:   ADD;
```

Give the next four states of the abstract machine starting in:

$$(ca, d, p) := (7, -3, 9:4:26:3:7:4:3:36:5:10:4:40:1:2:5:4:\ldots)$$

Recall that the procedure stack has the form:

sl	dl	ra	v_1	$\ldots$	v_n	$\ldots$

and the *base*-function is defined as:

$$\begin{aligned} base(p, 0) &:= 1 \\ base(p, dif + 1) &:= base(p, dif) + p.base(p, dif) \end{aligned}$$

Exercise 3 (3 Points)

Note: The translation of programs will be explained in the lecture on Thursday.

In addition to `while`-loops we want to have `for`-loops with implicit declaration of the counter variable in our example programming language:

$$\texttt{for (var } X := A \texttt{ ; } B \texttt{ ; } C_1 \texttt{) } C_2$$

(a) Extend the translation function ct accordingly. You may assume that the variable X is already declared, i.e., it is $update(var\ X, st, l)$ with st the symbol table and l the current level.

(b) Generate intermediate code for

```
for (var x := 0; x < 10; x := x + 1) P();
```

without parameters for the `CALL` instruction generated for `P()`.

Exercise 4 (6 Points)

We now finish the implementation of our `While2JasminCompiler` by generating the corresponding `Jasmin` code for our parsed program.

As stated on the Jasmin Webpage (http://jasmin.sourceforge.net):
"*Jasmin is an assembler for the Java Virtual Machine. It takes ASCII descriptions of Java classes, written in a simple assembler-like syntax using the Java Virtual Machine instruction set. It converts them into binary Java class files, suitable for loading by a Java runtime system.*"

Our compiler generates code for the Jasmin language from the parsed `While` language. As a recap the grammar for the `While` language is given as follows:

start	→	program EOF
program	→	statement program | statement
statement	→	declaration SEM | assignment SEM | branch | loop | out SEM
declaration	→	INT ID
assignment	→	ID ASSIGN expr | ID ASSIGN READ LBRAC RBRAC
out	→	WRITE LBRAC expr RBRAC | WRITE LBRAC STRING RBRAC
branch	→	IF LBRAC guard RBRAC LCBRAC program RCBRAC | IF LBRAC guard RBRAC LCBRAC program RCBRAC ELSE LCBRAC program RCBRAC
loop	→	WHILE LBRAC guard RBRAC LCBRAC program RCBRAC
expr	→	NUM | ID | subexpr | LBRAC subexpr RBRAC
subexpr	→	expr PLUS expr | expr MINUS expr | expr TIMES expr | expr DIV expr
guard	→	relation | subguard | LBRAC subguard RBRAC | NOT LBRAC guard RBRAC
subguard	→	guard AND guard | guard OR guard
relation	→	expr LT expr | expr LEQ expr | expr EQ expr | expr NEQ expr | expr GEQ expr | expr GT expr

Implement `generator.Generator.translateProg(ASTNode)` which given the *program* node in an abstract syntax tree returns the generated Jasmin Code in a string.

Hint: It is a good approach to write methods for every language construct and call them recursively (similar to the recursive descent parser). Once you get the idea, it is actually less effort than you might think!

Methods for generating `Jasmin` code for the main class, for writing to and reading from the console are already provided. You should also implement the method `translateExpr(ASTNode)` which translates an *expression* into `Jasmin` code and used in the method for writing to the console.

Here are some helpful resources for the `Jasmin` language:

- Jasmin main page: http://jasmin.sourceforge.net/
- Jasmin user guide: http://jasmin.sourceforge.net/guide.html
- List of machine instructions for Jasmin: http://jasmin.sourceforge.net/instructions.html
- Explanation of the machine instructions: http://homepages.inf.ed.ac.uk/kwxm/JVM/codeByNm.html

To test your implementation you can write code in the *WHILE* language and run it through our compiler with:

```
$java -cp bin Main tests/gcd.txt gcd.j
```

The generated code is written to the given filename (in this example `gcd.j`). Next you can use `Jasmin` to build an executable Java class file:

```
$java -jar lib/jasmin.jar gcd.j
```

You can execute the generated class file as a java program and observe its behavior:

```
$java gcd
42
27
GCD:
3
```

After finishing this exercise you now have your own simple compiler for Java code!