CSE505 – Spring 2017

Assignment 1 – Object-Oriented Parsing

(may be done by a team of two students)

Assigned: **Fri, Feb. 10**

Due Date for **Part 1**: **Sun, Feb 19**, (11:59 pm, online)

Due Date for **Part 2: Sun, Feb. 26** (11:59 pm, online)

Consider the following grammar for a simple programming language, **TinyPL**:

```
program -> decls stmts end
decls   -> int idlist ';'
idlist  -> id [',' idlist ]

stmts   -> stmt [ stmts ]
stmt    -> assign ';' | cmpd | cond | loop

assign  -> id '=' expr
cmpd    -> '{' stmts '}'
cond    -> if '(' rel_exp ')' stmt [ else stmt ]
loop    -> for '(' [assign] ';' [rel_exp] ';' [assign] ')' stmt

rel_exp -> expr ('<' | '>' | '==' | '!= ') expr
expr    -> term   [ ('+' | '-') expr ]
term    -> factor [ ('*' | '/') term ]
factor  -> int_lit | id | '(' expr ')'
```

Write an object-oriented top-down parser in Java that translates every **TinyPL** program into an equivalent sequence of **byte-codes** for a Java Virtual Machine.

Part 1 (due Feb 17): Assume that `stmt` is of the form: `stmt -> assign ; | cmpd`

Part 2 (due Feb 24): Assume that `stmt` is of the form: `stmt -> assign ; | cmpd | cond | loop`

Assumptions

1. All input test cases will be syntactically correct; syntax error-checking is not necessary.
2. Byte-code naming convention for all opcodes will follow Java conventions.

Program Structure

1. There should be one Java class definition for each nonterminal of the grammar. Place the code for the top-down procedure in the class constructor.
2. There should be a top-level driver class called `Parser` and another class, called `Code`, which has methods for code generation.
3. The code for the lexical analyzer will be given to you – do not modify it.

Expected Output

1. For each test case, output the byte codes generated on the console and also save the object diagram produced by JIVE as a .png file at the end of execution: In generating the object diagram, choose the "Stacked" (i.e., without tables) option while saving the object diagram.
2. Sample test cases and their outputs for Parts 1 and 2 will be posted on Piazza.
3. File naming convention will also be posted on Piazza. Please follow them carefully.

Clarifications

1. Generate `iconst`, `bipush`, or `sipush` depending upon the numeric value of the literal:

   - For small constants, in the range 0..5, the constant is implicit in the name of the instruction: `iconst_0 ... iconst_5`
   - In generating code for integers in the range 6..127, the actual value comes immediately after the opcode `bipush` We are not dealing with negative literal constants in TinyPL, but Java encodes numbers from -128 to +127 using 8 bits (one byte). Therefore, Java leaves one byte after the instruction for `bipush`.
   - For short integers greater than 127, the generated opcode is `sipush`. Now we need two bytes to encode the value and hence Java leaves two bytes after the instruction for `sipush`.

   Unlike opcodes such as `iadd, imul, isub`, and `idiv`, for which the operands come before the opcode, in the case of `bipush` and `sipush` the operand comes after the opcode because that is how the JVM will know how many bytes to push on the stack.

2. The `iload` and `istore` instructions have two variations each:

   - For the first three variables declared, the load and store instructions are, respectively, `iload_1, iload_2, iload_3` and `istore_1, istore_2,` and `istore_3.`
   - For the fourth and subsequent variables, the load and store instructions are, respectively, `iload n` and `istore n` respectively, where `n` > 3. The number `n` is encoded in one byte and placed after the `iload` and `istore` instructions.

3. Note that the initialization, test, and increment components of a `for`-loop are all optional, and the simplest loop is of the form `for ( ; ; )` … Your byte-code generation should work correctly whether or not a particular component of the for-loop is present.

4. Optimizations are *not* required: For programs in the TinyPL fragment, the Java compiler would perform two types of optimizations:

   a. Expressions such as 3 + (15 - 2 * 3) will be simplified to an integer value, namely, 12. This is part of a more general process called "constant folding" and this is typically done in the (machine-independent) optimization phase.
   b. When there is a chain of goto's in the generated byte codes, each one transferring control to the next, the Java compiler will optimize them by generating "goto x", where x is the location of the final destination.

End of Assignment 1