CSE505 – Spring 2018

Assignment 1 – Object-Oriented Parsing

(may be done by a team of two students)

Due Date for **Part 1**: **Fri, Feb. 23**, (11:59 pm, online)
Due Date for **Part 2: Weds, Feb. 28** (11:59 pm, online)

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

```
program  ->  { function }+ end
function -> int id pars '{' body '}'
pars -> '(' { int id } ')'

 body    ->  decls stmts
 decls   ->  int idlist ';'
idlist   ->  id [',' idlist ]

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

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

relexp ->  expr ('<' | '>' | '<=' | '>=' | '==' | '!= ') expr

expr     ->  term   [ ('+' | '-') expr ]
term     ->  factor [ ('*' | '/') term ]
factor   ->  int_lit | id | '(' expr ')' | funcall

funcall ->  id '(' [ exprlist ] ')'
exprlist -> expr [ ',' exprlist ]
```

Following the method outlined in class, write an object-oriented top-down parser in Java that translates every **TinyPL** program into a sequence of **byte-codes** for the Java Virtual Machine.

Part 1: Assume:

1. that the rules for `program`, `function`, `pars`, `funcall`, and `exprlist` are deleted from the grammar;
2. that the rule for `stmt` does not include the case for `return` on its RHS and, similarly, the rule for `factor` does not include `funcall` on its RHS; and
3. `body` is the start symbol of the grammar and the rule for `body` has the keyword `end` on the RHS at the end of the rule.

Part 2: Consider the full grammar as shown above.

Expected Output

1. Sample test cases and their outputs for Parts 1 and 2 are posted on Piazza at: `Resources` → `Homeworks` → `A1.zip`.
2. For each test case, output the byte-codes on the console. Byte-code naming convention should follow that of Java byte-codes. Note that **TinyPL** only has the `int` type.
    a. Generate `iconst`, `bipush`, or `sipush` according to the numeric value of the literal.
    b. Generate `iadd`, `isub`, `imult`, and `idiv` for the arithmetic operators.
    c. For `iload` and `istore`, take the index of the variable into account when generating the bytecode.
    d. Relational expressions (`relexp`) may be translated using the `if_icmp*` bytecodes. Transfer of control is effected by the `goto` bytecode.
    e. Function calls may be translated using `aload_0` and `invokevirtual`, and function return by `ireturn`.
3. Save the object diagram produced by JIVE as a `.png` file at the end of execution. In saving the object diagram, choose the "Stacked with Tables" option.

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` as well as a class called `Code`, for code generation, and a class `SymTab`, for the symbol table.
3. The code for the lexical analyzer (classes `Lexer`, `Token`, and `Buffer`) will be given to you in their entirely – do not modify them.

Assumptions

1. All input test cases will be syntactically correct; syntax error-checking is not necessary.
2. Optimizations are *not* required: For programs in the **TinyPL** fragment, the Java compiler would perform two types of optimizations, both of which are *not* required for this assignment:

    a. Expressions such as 3 + (15 - 2 * 3) will be simplified by the Java compiler 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.

What to Submit for Part 1

Prepare a top-level directory named `A1_Part1_UBITId1_UBITId2` if the assignment is done by two students (list `UBITId`'s in alphabetic order); otherwise, name it as `A1_Part1_UBITId` if the assignment is done solo. In this directory place the code for `Parser.java`, `Lexer.java`, `Buffer.java`, and `Token.java` as well as the object diagrams, named `obj1.png`, `obj2.png`, `obj3.png`, `obj4.png` and `obj5.png` for the five test cases respectively. Compress the directory and submit the compressed file using the online submission procedure – instructions posted under `Resources` → `Homeworks`. Only one submission per team is required.

Part 2 submission details are similar and will be posted in due course.

End of Assignment 1