CSE505 – Spring 2021

Assignment 1 – Object-Oriented Parsing

(may be done by a team of two students)

Due Date: **Monday, March 1** (11:59 pm, online code submission)

Consider the following grammar (with EBNF notation) for a simple programming language, called **TinyPL,** whose programs are a sequence of zero or more functions defined as follows:

```
program  ->  { function } end
function -> int id '(' [ pars ] ')' '{' body '}'
pars ->  int id { ',' int id }

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

stmts   ->  stmt [ stmts ]
stmt    ->  assign ';' | loop | cond | cmpd |
            return ‘;’ | print ‘;’

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

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

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

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

Write in Java an object-oriented parser that translates TinyPL programs into byte-codes as discussed in the lectures.

Starter code for this assignment is given in `A1_Starter.zip`. The only file in which you need to supply missing code is `Parser.java`. This file contains one Java class definition for each nonterminal of the above grammar. The places where missing code is to be supplied are indicated by Java comments.

In generating byte-codes, follow the naming convention 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. Note that TinyPL, unlike Java, has right-associative semantics for these 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.

Assumptions

1. All input test cases will be syntactically correct; syntax error-checking is not necessary.

2. TinyPL does ***not*** permit syntactically nested calls such as `x = f(g(10),h(20))`. When there is a need to write nested calls, the TinyPL programmer has to "flatten" them and write a sequence of assignments, like this: `t1 = g(10); t2 = h(20); x = f(t1,t2).`

Testing your Program

Test cases and their output are given in the `A1_Starter` directory as files `test_cases.txt` and `test_results.txt`.

Run your Java program and, when prompted for input, paste the contents of the file `test_cases.txt` into the `Console` panel. Save the output printed on the `Console` by your program in a file `A1_test_results.txt`.

Test that your byte-codes are correct by executing function calls using the byte-code interpreter supplied with the Starter code.

Perform another test of your program by running it under JIVE and obtain the object diagram for a test case consisting of just the `gcd` and `friends` functions. Choose the "Stacked" option and save the object diagram as `A1.png`.

A demo of parsing and executing a TinyPL program will be posted.

What to Submit

Prepare a top-level directory named `A1_UBITId1_UBITId2` if the assignment is done by two students (list `UBITId`'s in alphabetic order); otherwise, name it as `A1_UBITId` if the assignment is done solo.

In this directory place all the Java code as well as `A1_test_results.txt` and `A1.png`.

Compress the directory and submit the compressed file using the online submission procedure – instructions to be posted at `Resources` → `Assignments` → `Online_Submission.pdf`. Only one submission per team is required.

End of Assignment 1