**Homework 3 – MiniJava Generating intermediate code**

In this part of the project you have to write visitors that convert MiniJava code into an intermediate language which we are going to call Piglet. You can see most of the elements of this language by reading its grammar ([BNF form](http://cgi.di.uoa.gr/%7Ethp06/project_files/piglet.html), [JavaCC form](http://cgi.di.uoa.gr/%7Ethp06/project_files/piglet.jj)). Below is some explanation and elaboration:

1. Methods are defined as "*function\_name [arguments]*" and they take as arguments the temporary variables "*TEMP 0*", *"TEMP 1*" etc. Identifiers "*TEMP x*" are used thereafter as temporary variables, temporary values, for expression calculations, etc. .
2. The expression "*CALL expression (expression\_1 expression\_2 ... expression\_n)*" calls the function located at the memory location equal to the value of expression and is passed as arguments the values of *expression\_1*, *expression\_2*, etc. Each function call is dynamic so the location of the memory function is known only at runtime. The result of the CALL expression is the value returned by the function.
3. The expression *"HALLOCATE expression*" takes as argument an integer value and allocates dynamically as many memory slots as this value. Typically the value will be a multiple of 4, since both addresses and integers are 4 bytes. Each time you allocate an object, you should also allocate space for its virtual table.
4. The command "*MOVE temp expression*" stores the value of the expression “expression” to a temporary variable temp (a "*TEMP x*").
5. The command "*HSTORE expression\_1 integer\_literal expression\_2*" stores the value of expression\_2 in a memory location with base address expression\_1 and offset integer\_literal. This can be used both for array element assignments and object field assignments.
6. The command "*HLOAD temp expression integer\_literal*" reads the value of the memory location with base address expression and offset integer\_literal to the temporary variable temp. This will be used not only to read data from arrays and class member fields but also to call methods by reading their address from the virtual table.
7. The command "*JUMP label*" makes a jump to label. Labels are practically memory locations and we consider them to be integer expressions with their value being the memory location to which they refer.
8. The command "*CJUMP expression label*" moves forward to the next command (branch not taken) only if the value of expression is exactly equal to 1, otherwise it makes a jump to label (branch taken).
9. The expression "*LT expression\_1 expression\_2*" checks whether expression\_1 is less than expression\_2, and similarly for PLUS, MINUS, TIMES, which execute addition, subtraction, and multiplication. To check whether an object is null you can use the expression "*LT expression 1*".
10. The language allows “*statement expressions*”, which are sets of commands that end with “*RETURN expression*”. The value of the statement expression is the value of this returned expression. A statement expression starts with "*BEGIN*" and ends with "*END*".
11. The "*ERROR*" command terminates the program with an error. One possible reason for this to happen is if the program tries to access an array out-of-bounds.
12. The "*NOOP*" command does nothing.

If you do not remember or haven't seen how a virtual table (v-table) is constructed, essentially it forms a table structure in the first 4 bytes of an object and defines an offset for each function. Consider a function foo in position 0 and bar in position 1 of the table (with actual offset 4). If a method is overridden, the overriding version is inserted in the same location of the virtual table as the overridden version. Virtual calls are implemented by finding the address of the function to call through the virtual table. If we wanted to depict this in C, imagine that object obj is located at location x and we are calling foo that is in the 3rd position of the v-table. The function that is going to be called is in memory location x[0][12].

Many things that this description may lack can be found by reading the grammar of Piglet. It is small enough and understandable. You can also understand the form of Piglet seeing the examples given [here](http://cgi.di.uoa.gr/%7Ethp06/project_files/piglet-examples/) (corresponding to the earlier [MiniJava examples](http://cgi.di.uoa.gr/%7Ethp06/project_files/minijava-examples-new/) from HW2). You can use [this](http://cgi.di.uoa.gr/%7Ethp06/project_files/pretty-printer.jar) program in order to give "pretty-print" to the Piglet code you produce, for readability(\*). Additionally you can use [this](http://cgi.di.uoa.gr/%7Ethp06/project_files/pgi.jar) program to "run" your Piglet code and see if it produces correct results. The correct results can be seen by comparing the output that javac and java produce with the output you get from executing this Piglet interpreter on your output code.

Your program should run as follows:  
  
java [MainClassName] [file1.java] [file2.java] ... [fileN.java]  
  
That is, your program must compile to Piglet all **.java** files given as arguments. Moreover, the outputs must be stored in files named file1.pg, file2.pg, ... fileN.pg respectively.  
  
(\*) run: java -jar pretty-printer.jar [file1.pg] [file2.pg] ... [fileN.pg]   
This command stores the outputs at file1-pretty.pg, file2-pretty.pg, etc.