Assignment Description

In this assignment, you will implement the infrastructure for the intermediate representation in the compiler, including the High IR, the Low IR, and the control flow graph representation. We expect you to build upon the code that you wrote for the previous assignments. You are required to implement the following:

- **High IR**. This is the high-level intermediate representation of the computation in the program. It is similar to the AST, but doesn't represent the information already stored in the symbol table, such as class, field, method, and variable declarations. It only represents the computation in the program, that is, the method bodies. For each method, you will have a high-level tree representation of body of that method.

  The nodes in the High IR consist of statement and expression nodes. Statement nodes include if nodes, while nodes, blocks, assignments, break, continue, return, and method call nodes. Expression nodes include unary and binary expressions, array accesses, array and field accesses, variables, method calls, new constructor expressions, length-of expressions, or constants. Your are allowed to use additional High IR nodes if you consider so, but you must clearly describe them in the write-up.

  You should represent variables using references to their entries in the symbol table. Also, each block should contain a list of children statements representing the body of the block; and each method call node should contain a list of all the actual arguments. Although your High IR doesn't represent variable declarations, it must explicitly represent initializations in such declarations. Each variable initialization is represented using an assignment at the beginning of the scope where the variable is declared.

  Because High IR is similar to the AST representation, you don't have to create a new class hierarchy; you can re-use the AST hierarchy instead. You can change your AST construction to directly generate High IR, or have an additional pass over the AST to perform these changes. Finally, given a class and a method, your compiler should be able to provide the High IR of that method.

- **Low IR**. This is the low-level intermediate representation of the computation. The Low IR uses three-address code and models the instruction set of an abstract machine. For each method, you will have a low-level sequence of instructions representing the body of that method.

  The low-level instructions may include: unary and binary instructions, copy instruction, array access instructions, length-of instruction, field access instructions, method calls, return instructions, labels, unconditional jumps, and conditional jumps branching on true or false conditions. You will also use a special instruction for library function calls. Your are allowed to extend this instruction set with additional Low IR instructions if you consider so, but you must clearly describe them in the write-up.

  The operands of the Low IR instructions can be either constants, program variables, or temporary variables created by the compiler during the translation to low-level code. Whenever it creates temporary variables, the compiler must insert them in the symbol table.

- **Lowering the IR**. You must implement a translation of the High IR code to Low IR instructions. Your lowering phase should translate high-level constructs such as if and while statements, short-circuit conditional expressions, break and continue statements into low-level code using jump instructions. There should be no consecutive labels in the generated Low IR code.

  When the compiler lowers an array access `a[i]`, it must insert a library call `__checkNullRef(a)` to check that `a` is not null, and a library call `__checkArrayAccess(a,i)` to check that the access is within bounds. The compiler inserts these checks right before the array access instruction. To lower the length-of expression `a.length`, you will generate a null check `__checkNullRef(a)`, followed by the length-of instruction. To lower a dynamic array allocation `new T[n]`, you must generate a check `__checkSize(n)` to check if the array size is positive, then invoke a library function `__allocateArray(n,s)`, which will return a reference to the newly allocated array. The allocation function takes two arguments, the number of elements `n` and the element size `s`. To determine the size required for different types, you will assume 4 bytes for integers or references, and 1 byte for booleans.

For each field access `o.f` or method call `o.m()`, the compiler generates a null check `__checkNullRef(o)`, followed by the field access or call instruction. The call instruction generated for a method call `o.m(...)` must have the receiver object `o` as its first argument, followed by the actual parameters. To lower a dynamic object allocation `new C()`, you will invoke a library function `__allocateObject(s)`, which will return a reference to the newly allocated object. The size `s` of the allocated object must accommodate all of the fields of the allocated object, plus 4 bytes (to store a reference to the dynamic dispatch table).

Finally, to lower the concatenation of strings `s + t`, you will generate a library call `__stringCat(s,t)`.

- **Control flow graphs**. You have to design the class hierarchy for the control flow graph (CFG) representation and build the CFG for each method in the program. Each control flow graph is a graph whose nodes represent basic blocks. Each basic block consists of a sequence of low-level IR instructions. There should be no label instructions, except at the beginning of basic blocks. Similarly, there should be no jump instructions except at the end of basic blocks. Basic blocks don't necessarily begin with label instructions or end with jump instructions. Also, each CFG has a special start basic block which represents the starting point of the computation in the graph.

  You have to build the CFG of the program starting from the low-level intermediate code. The resulting CFG should minimize the number of empty basic blocks (i.e. blocks with no instructions other than labels and jumps). Also, it must ensure that there are no blocks B and B' such that B' is the only successor of B, and B is the only predecessor of B'.

Command line invocation. Your compiler must be invoked with a single file name as argument:

```
java IC.Compiler <file.ic>
```

With this command, the compiler will parse the input file, build the AST, and perform semantics check using the code from the previous assignments. Next, it will convert the AST into High IR and the it will translate it to Low IR. Finally, it will build the CFG representation. In addition to all of the options from the previous assignments, your compiler must support two additional command-line options:

1. The "`-dump-lir`" option will print at `System.out` a textual description of the Low IR for each method in the program. The code for each method must begin with a line showing the class name and the method name. The code for different methods must be separated by blank lines.
2. The "`-dump-cfg`" option will print a textual description of the CFG for each method in the program. You must indicate the start basic block and, for each basic clock, what are its children blocks.

Package Structure: Create a sub-package for the Low IR and a sub-package for the CFG representation.

What to turn in

Turn in on paper:

- A brief, clear, and concise document describing the your code structure and testing strategy. Include a description of your High IR structure, of your Low IR nodes, and of your CFG structure.
- Feedback. Please continue to provide one paragraph to tell us your overall thoughts about the assignment, especially how much time you spent on it and how you think it could be improved.

Electronic submission:

Your electronic submission is expected at the same time as your written submission: at the beginning of class on the due date. Electronic submissions after 10AM will be considered a day late. Place your files in `\\crane\courses\cs412-sp03\grpX\pa3`, where `grpX` is your group identifier. Please organize your top-level directory structure as follows :

- `src` : all of your source code. Do not submit the class files – we should be able to build your compiler from its source code using `make compiler`.
- `doc` : javadoc output that we can browse on-line. We should be able to browse through all your classes and interfaces, including all their methods and fields. Also include in this directory your write-up (`writeup.ps` or `writeup.txt`) and a `readme.txt` file containing any known bugs and other information that we might find useful when grading your assignment.
- `test` : any test cases you used in testing your project.

Note: Failure to submit your assignment in the proper format may result in deductions from your grade.