**Compilation – PA4**

**Package Hierarchy**

* **IC**: Main package,contains the project's main() function.
  + **IC.Parser:** Contains the lexer and parser classes (handles PA1 and PA2).
  + **IC.AST:** Contains the ASTnodes classes (like ASTnode, Return, While etc') for the AST*.*
  + **IC.SemanticChecks:** Handles semantic checks (PA3) and symbol table creation.
    - **IC.SemanticChecks.SymTables:** Contains the specific table classes for the symbol table (eg. ProgramTable, ClassTable, MethodTable).
      * **IC.SemanticChecks.SymTables.Symbols:** Contains the symbols ("values") for the symbol table (eg. ClassSymbol, MethodSym).
  + **IC.TypeTable:** Handles type table creation.
    - **IC.TypeTable.Types:** Contains types for the type table (eg. ClassType, MethodType).
  + **IC.lir:** Handles LIR lowering (PA4).
    - **IC.lir.Instructins:** Contains all LIR Instruction classes (eg. Move instruction, jump instruction).

**IR Lowering**

For the IR Lowering we use a visitor (LirVisitor) which visits the program's AST, translates each node to the LIR code necessary to execute it, and adds these LIR instructions to a list containing the LIR code.

For modularity and easy parsing in PA5, we add each instruction as an Instruction node (new class for LIR instructions), and not just as a simple string.

We've decided to base our instruction nodes on MicroLIR. So we have an abstract class Instruction, which represents a LIR instruction (specific instructions like "Jump", "Move" etc' extend it) and an abstract class Operand (specific operands like regs, memory, etc' extend it).

The IR Lowering starts by building the class layout table (based on the symbol tables), and then proceeds to visit the program's AST.

We keep a global counter named "regCount" which starts from 1 and represents what's the lowest available register. Additionally, our convention is that when LirVisitor visits an Expression ASTnode, the expression's calculated result will be saved in register number regCount – and the node's "visit" return value will be the register the result is saved into (for conveniance).

Additional misc info:

* We build the string literals table on the fly (each time we encouter a string literal, we assign it a label and add it to the string literal list).
* We build the variable shadowing on the fly.

**Notes:**

1. We assume the filename given to the program contains only one "." .
2. Runtime checks aren't implemented yet (Or said in the forum they don't have to be implemented for PA4).
3. We print a newline before each LIR label (including function and jump labels).
4. It wasn't clear on the instructions, so we never print to the screen the LIR translation.  
   If the "-print-lir" flag was provided – we write the translation to the file (as you specified).  
   If the flag wasn't provided – we make the LIR translation, but don’t write or print it.

**Notes from PA3:**

1. In PrettyPrinter: for some nodes, there was no reference in the given examples, so we weren’t sure whether to print the type or not, so we did what we thought you expected.
2. We do not allow assignment of to primitive types, except for string, since it wasn’t described in the IC specifications, not the examples.
3. Since we did not implement the bonus task in PA2 to prevent from declaring a variable in an statement without a statement block, our compiler does not handle these issues.
4. According to the IC specifications, classes cannot extend themselves.
5. When overriding a method, we demand that the exact same types will be used – no extending types allowed.
6. We don’t allow overriding a virtual method with a static one, and vice versa, since it wasn’t clear in the specifications.