Code structure

IC – Contains the main Compiler class, which creates all the compiling phases: Lexing, parsing and semantic symbol table building & checking. Also contains all the basic enums used in the project – DataTypes, BinaryOps etc.
IC.ast – Contains all the AST node types used in the project, and the Visitor interface
IC.parser – Contains the Lexer & Parser files from previous PA1, PA2, after generation by the lex.CUP & IC.cup files.

IC.semanticChecks – The main package of PA3, contains the following files:

3 classes implementing the visitor interface:

- **SymbolTableBuilder** – Visits the program AST nodes from its root, and creates a FrameScope tree of all scopes in the program, during which it connects each AST node's scope to its parent to form the hierarchy.
- **TypeTabelBuilder** – Creates a data structure of all primitive, user and method types used in the program by going over the AST nodes from its root down, including array types.
- **SemanticChecker** - Checks the code for semantic errors - all checking rules defined by PA3, and a bonus check for "a method with a non-void return type returns a value on every control path".

Additional classes:

- **FrameScope** –
  The class acts as the scope of each node of the AST, and by that obeys the hierarchy rules. It possess all necessary scope details of the scope such as methods, fields, local variables etc. Each scope is defined differently according to his scope type as in Section 10 of the IC specification.
  SemanticException –
  A unique exception to be thrown when a semantic error is discovered during the checking.

IC.lir - The main package of PA4, contains the following files:

- **DispatchTableBuilder** – Responsible for printing out all classe's inner fields data in LIR format. For each class, it specifies all methods and their offsets, and all the fields and their offsets. In case of a subclass – The class inherits its parent's dispatch table, and updates its own method table for inheriting methods. Field shadowing is handled in the semantic checker, so existing fields aren't checked, and additional fields are added to the field table if exist.
- **StringsBuilder –** Responsible of forming a unique LIR label for each existing constant string in the file. Visits the entire AST node constructed in previous section, and if it finds a string literal – Attaches a unique label to it and prints it out in LIR format.
- **LirTranslator** – The main LIR translation file. Goes over the AST node constructed in previous section, and visits from the root down. For each instruction, it prints out the LIR translation, including all the checking requires. For example – In case of an array location access, attaches checking instructions in LIR format – Check the array is initialized, index in bounds etc.

IR implementation

- The LIR translation is divided into 3 phases: String literals, Dispatch table, and the instructions translation.
  When **building the string literals,** the string builder class visits the AST tree from the root down, and that way collects all the string literals and labels them.
  When **building the dispatch table**, the dispatch table class uses the symbol table to analyze the classes, starting from the root global scope. Then it goes over all the classes, and adds their methods & fields to a HashMap. It continues recursively over the scopes – SubClasses aren't children scopes of the global scope, instead they are attached as children of their parent's scope.
  When **translating into LIR instructions,** the LIR translator implements a visit method for each AST node and attaches its appropriate LIR string to the LIR file. That includes labeling jump calls for conditions, error checking for sensitive access to data – Array access, class' field access, division by 0 etc. & calling to library methods – such as print. Every time a use of a new register is required, it gets a unique label, the next available label – If R3 isn't required anymore, for instance, it will be the next register label used by the program.

Remarks

- when a string is not initialized we observe 2 different behaviors:
  - If it's concatenated with other string we get a null pointer exception.
  - If it's passed to a library method we get a null pointer exception.

Known bugs

- Passing arguments in Main isn't supported in the microLIR, it is a known bug and our program doesn't recognize it.
- microLIR simulator returns an error when a return statement appears in Main. Our program doesn't refer to it.
- Library methods in libic.sig don't behave as expected in microLIR, a known problem referred in the PA forum.
- microLIR doesn't support special characters passed in a string, so we removed '\n' from the strings to prevent microLIR failures.

Fixes from last PA

- TypeTableBuilder was fixed and now supports the ID's according to instructor's demands.
- PrettyPrinter was fixed, and when using –print-ast, symbol names and tables are printed.
- Now main found bug fixed – Now when a main doesn't appear in the program, an appropriate error is returned.
- Local Variable shadowing a parameter is now allowed in a method.
- Array Length access on a non-array type variable returns an error.
- Exception for method call on an array type expression fixed