**Compiler Overview**

**Tommer Toledo 209706795 -** [**GitHub Project**](https://github.com/gwchar2/Compiler)

**Overview**

This project is a compiler that processes a custom programming language and generates quad intermediate code.   
The system consists of a lexer, a parser, an **abstract syntax tree (AST)**, a **visitor**-**based** semantic checker, and a quad code generator. The goal is to generate a clear and structured intermediate representation of the input program, even if optimizations are minimal.

**Code Structure**

**Lexer** (Flex): Recognizes tokens (NUM, ID, OPERATOR, etc.).

**Parser** (Bison): Constructs AST based on language grammar.

**AST Nodes**: Each node represents a part of the program (e.g., ASTAssignNode, ASTBinaryExprNode) .

**Visitor Classes**:

* SemanticChecker: Ensures types and rules are correct.
* QuadGenerator: Generates quad instructions for execution.

**Quad Code**: A list of instructions that represent operations in a structured form.

**Symbol Table & Scope Management**:

* The compiler maintains a symbol table that stores variable names, types, and other attributes.
* The symbol table is implemented as a **stack-based scope system**, where each new block (e.g., function or loop) pushes a new scope onto the stack.
* When a block exits, its scope is popped, ensuring that variable names do not persist beyond their intended lifetime.

**Visitor Pattern**

The **Visitor Pattern** is used for both semantic analysis and quad code generation. The visitor is implemented as a base class (**ASTVisitor**) with functions for different AST nodes.   
The **QuadGenerator** and **SemanticChecker** inherit from this and implement their own versions of visit() functions. The traversal is done by calling accept(\*this), which ensures the correct visit() function is executed for each node.

Example flow:

* ASTAssignNode.accept(QuadGenerator) → Calls QuadGenerator::visit(ASTAssignNode&).
* ASTBinaryExprNode.accept(QuadGenerator) → Calls QuadGenerator::visit(ASTBinaryExprNode&).

**Implementation Details**

* **Temporary Variables**: Every immediate value is assigned a temporary variable.
* **Zero Division Handling**: Division with 0 results in 0 (for int) or 0.0 (for float).
* **Type Preservation**:
  + Temporary variables remain int or float based on their first assignment.
  + Casting int → float happens automatically, when necessary, but float → float and   
    int → int casts do nothing.
* **Expression Evaluation**:
  + If int + float, the int is first converted to a float (using a temporary), then the operation proceeds.
  + For comparisons (>=, ==), the operands are converted appropriately, but the result is always an int (1 or 0).
* **Assignments**:
  + Assigning float → int automatically casts before assignment (and vice-versa).

**Conclusion**

The compiler is structured to be clear and easy to follow, even at the cost of optimizations.   
**Type conversions are handled automatically**, and **temporary variables maintain type consistency** throughout the program.   
The use of the visitor pattern makes the compiler modular, separating semantic checking and quad generation while maintaining a structured AST traversal.

**Key Files**

src/AST/Base/QuadGenerator.cpp → Generates the quad code.

src/AST/Base/SemanticChecker.cpp → Semantic Analysis.

src/global\_scope.cpp → Handles the global scope.

src/symbol\_table.cpp → Handles the symbol tables within the scope.

src/cpq.cpp → Holds the main logic of the program.