Eta Compiler Project: Design and Implementation Overview

Together with a partner, I contributed to the full-stack development of a compiler for the Eta programming language, progressing through six major programming assignments that spanned lexical analysis, syntactic analysis, semantic analysis, intermediate representation (IR) generation, assembly code generation, and optimization.

Project Scope & Technologies:
Our compiler was implemented in Java, leveraging tools such as JFlex for lexical analysis, CUP for parsing, and Argparse4j for command-line argument parsing. The project required designing custom data structures, including token and AST node classes, and implementing the Visitor pattern to facilitate traversal and manipulation of the syntax tree and IR nodes.

Design Decisions & Challenges:
Key design choices included grouping keywords and symbols for efficient tokenization, implementing a stack of hashmaps for context management in semantic analysis, and employing the Visitor pattern for both type checking and IR generation to maximize code reuse. Notable challenges involved handling unicode and escape sequences during lexing, resolving grammar ambiguities for parsing, and managing error reporting across compiler stages. Throughout the project, we iteratively refined our parser and typechecker to align with the Eta Language Specification, balancing flexibility with semantic rigor.

Implementation & Testing:
Our workflow emphasized incremental development and testing, with team members specializing in different compiler components. We developed comprehensive unit and integration tests for each stage, supplemented by custom edge cases to ensure robust coverage. The compiler was validated using provided test harnesses and our own test suites, with continuous integration via git to streamline collaboration.

Results & Impact:
By the final stage, our compiler successfully processed Eta programs from source code to executable binaries, generating assembly code compliant with the Linux ABI and supporting major language features. We implemented optimizations such as constant folding and nontrivial instruction selection. While some advanced features (e.g., multidimensional arrays, global strings) remained partially implemented, our modular architecture and documentation facilitate future extension.

Reflection:
This project honed my skills in compiler construction, collaborative software engineering, and problem-solving under tight deadlines. I gained hands-on experience with language specification interpretation, modular design, and automated testing, and contributed to a codebase that demonstrates best practices in maintainability and extensibility.

One lesson that I took away is the importance of clear communication. There were a few instances where independent systems did not mesh because expectations were not set prior to development. Stitching these systems together proved to be a major source of frustration. When we became more collaborative, not only did we avoid these headaches, but the project became more efficient and enjoyable to work on.