# Base Paper References

- **Compiler Design Theory, Base Reference 1 Aho, Lam, Ullman, and Sethi (2006) Compilers: Foundations, Methods, and Resources (2nd Edition)** This textbook, which is widely recognized as the "Dragon Book," serves as the theoretical basis for practically all contemporary compiler implementations. The same traditional compiler pipeline outlined in this book is used in our mini-project: Lexical Analysis of the Dragon Book Concept in Our Mini Project Flex Syntax Analysis was used to implement Bison Abstract Syntax Tree (AST) is used for implementation. created with a custom AST.c Analyzing Semantics Semantic.c implementation Codegen.c is used to implement code generation. As a result, our transpiler's general architecture adheres to the Aho et al. standard multi-phase compiler paradigm.

- **Parsing Framework GNU Bison (GNU Project) Base Reference 2 Bison:** The YACC-Friendly Parser Generator The primary syntax analysis engine for our project is Bison. Parser.y defines the custom language's grammar, which Bison uses to produce: The parsing engine: parser.tab.c, parser.tab.h → definitions of tokens and data structures Bison's LALR(1) parsing technique, as explained in this reference, directly drives our grammar rules and AST construction.

- **Lexical Analysis Framework, Base Reference 3 GNU Project's GNU Flex Flex: The Quick Lexical Examiner** The compiler's lexical analysis phase is implemented using Flex. Token patterns based on regular expressions are defined in the file lexer.l, including: Identifiers of Keywords Operators in Literals The tokenizer that feeds tokens to the Bison parser is lex.yy.c, which is generated by Flex. This adheres to compiler theory's conventional lexer–parser separation. This approach embodies the classic lexer–parser separation principle in compiler design, where the lexer focuses on the structure of words and symbols, while the parser builds meaning from them.

- **Base Reference 4: Specification for the Target Language The C Language Standard, ISO/IEC 9899:2018 Standard C code is produced by our transpiler.** Consequently, ISO-compliant C syntax is mapped to the semantics of expressions, operators, control structures, and function calls in the custom language. This guarantees Adaptability Compliance with the GCC Appropriate runtime actions. As a result, the custom language inherits the reliability, efficiency, and standardized behavior of modern C environments.