**Software Requirements Specification**

**(SRS) for**

***SYNTAXISA***

(assembly code Compiler)

**Version 2.0**

**Contributors**:

Kashish Joshi

Yateen Dogra

Krishna Saglani

Yash Singholia

Bhatt Nidhish Hiteshbhai

Sunny Kumar

**3- March-2025**

# Table of Contents

1. **Introduction**

* 1.1 Purpose
* 1.2 Document Conventions
* 1.3 Intended Audience and Reading Suggestions
* 1.4 Product Scope
* 1.5 References

2. **Overall Description**

* 2.1 Product Perspective
* 2.2 Product Functions
* 2.3 User Classes and Characteristics
* 2.4 Operating Environment
* 2.5 Design and Implementation Constraints
* 2.6 User Documentation
* 2.7 Assumptions and Dependencies

1. **External Interface Requirements** 
   * 3.1 User Interfaces
   * 3.2 Hardware Interfaces
   * 3.3 Software Interfaces
   * 3.4 Communications Interfaces
2. **System Features**

* 4.1 Tokenization (Lexer)
* 4.2 Parsing (Syntax Analysis)
* 4.3 Semantic Checking
* 4.4 Intermediate Representation
* 4.5 Code Optimization
* 4.6 Machine Code Generation
* 4.7 Error Handling

1. **Other Nonfunctional Requirements** -

- 5.1 Performance Requirements

- 5.2 Safety Requirements

- 5.3 Security Requirements

-5.4 Software Quality Attributes

- 5.5 Business Rules

6 **Other Requirements**

# Introduction

1.1 **Purpose**

This document specifies the requirements for SYNTAXISA , a compiler designed to convert high-level programming code into an optimized, intermediate assembly code representation. The goal is to create an efficient, portable, and scalable compiler that aids both learners and developers in understanding compiler design and execution.

The primary objective is to create a robust compilation pipeline capable of error detection, optimization, and seamless execution on a hardware or emulator. The project is designed for use in academia and research while maintaining practical usability.

1.2 **Document Conventions**

* Keywords such as "MUST," "SHOULD," "MAY" are used to indicate priority levels.
* Technical terms, programming keywords, and key components (e.g., Lexer, Parser, AST) are bolded.
* Code snippets are written in monospaced font for better readability.
* Illustrations and diagrams are used to explain system components and workflows.

1.3 **Intended Audience and Reading Suggestions**

* Students & Researchers – This document provides a foundational overview of compiler development.
* Software Developers – The document outlines implementation details useful for extending the compiler.
* Educators & Instructors – Can use this material to introduce students to assembly code compilation.
* Test Engineers – Will reference this document to develop test cases for different compiler stages.

1.4 **Product Scope**

SYNTAXISA aims to provide a reliable compilation framework that supports:

* High-level Language Support – Accepts a structured, C-like language as input.
* assembly code Generation – Converts code into a structured instruction set for execution on a custom VM.
* Modular Design – Ensures easy integration of additional features such as Just-In-Time (JIT) compilation.
* Error Handling – Provides detailed lexical, syntax, and semantic analysis.
* Extensibility – Future releases will include support for optimization passes and debugging tools.

1.5 **References**

* "Modern Compiler Implementation in C" by Andrew W. Appel
* "Compilers: Principles, Techniques, and Tools" by Aho, Lam, Sethi, and Ullman
* LLVM and GCC Documentation
* Research Papers on Compiler Optimization Techniques

1. Overall Description

2.1 **Product Perspective**

The compiler is designed as a modular, multi-pass system with separate stages for lexical analysis, parsing, semantic analysis, optimization, and Assembly Code Generation. It is developed for educational and research applications, supporting cross-platform compatibility and integration with third-party development environments.

The compiled assembly code can be executed on a dedicated hardware or emulator that ensures efficient memory management and runtime execution.

2.2 **Product Functions**

* Source Code Input – Accepts source code in a predefined high-level language.
* Lexical Analysis – Tokenizes the input and validates syntax structures.
* Parsing & AST Construction – Generates an abstract syntax tree (AST) from tokens.
* Semantic Analysis – Ensures type correctness and scoping rules.
* assembly code Generation – Converts AST into an optimized instruction set.
* Virtual Machine Execution – Executes compiled assembly code on supported hardware.
* Error Reporting – Provides structured feedback to developers.

2.3 **User Classes and Characteristics**

* Beginners – Require intuitive error messages and comprehensive documentation.
* Advanced Programmers – May utilize debugging features and optimization capabilities.
* Educators – Will use the system for teaching compiler fundamentals.
* Software Engineers – May extend or integrate the compiler with existing tools.

2.4 **Operating Environment**

* Supported OS: Windows, Linux, macOS
* Programming Languages: Implemented in C++ with optional Python bindings.
* Memory Requirements: Minimum 2GB RAM recommended.
* Execution Environment: Command-line interface with planned GUI support in future versions.

2.5 **Design and Implementation Constraints**

* Modular architecture to facilitate easy upgrades and feature extensions.
* Limited external dependencies to ensure portability and performance.
* Must handle recursion and function calls efficiently within a controlled stack environment.

2.6 **User Documentation**

* Installation Guide – Step-by-step setup instructions.
* User Manual – Covers usage, syntax, and compiler options.
* Troubleshooting Guide – Common errors and solutions.
* Developer Documentation – Covers compiler architecture, assembly code format, and VM execution model.

2.7 **Assumptions and Dependencies**

* Users have a basic understanding of programming and CLI usage.
* Development tools like GCC/Clang or Visual Studio are installed.
* The system assumes that input files are correctly formatted before compilation.
* The project may depend on external parsing libraries such as ANTLR in future versions.

# External Interface Requirements

## User Interfaces

* **Command-line interface**: Users will input code via standard text files or directly on the command line. The program will display errors and results directly on the console.
* **Error Output**: The compiler will generate meaningful error messages that indicate the line number, type of error, and possible causes.
* **Help Command**: The CLI will have a built-in help command to provide basic information on how to use the compiler.

## Hardware Interfaces

* The **minimum hardware** required will be a computer with 2 GB of RAM and at least a

1.0 GHz CPU.

* The compiler will **optimize resource usage** to ensure smooth execution on lower-end systems.

## Software Interfaces

* File I/O: Source code will be input through plain text files, and output will be in the form of assembly code stored in another file.
* **Standard Libraries**: The system will use libraries like stdlib in C or Python’s os and sys for file I/O and other basic operations.

## Communications Interfaces

* **No network communication** will be required for the basic operations of the compiler.
* Future versions may include **network-based debugging** or **distributed compilation** features, but these will not be part of the initial release.

# System Features

4.1 **Tokenization (Lexer)**

* Recognizes keywords, identifiers, literals, and symbols from the source code to convert them into tokens.
* Handles whitespace and comments by ignoring them unless they affect the syntax.
* Identifies and classifies different token types, such as operators, delimiters, and special characters.
* Generates an initial symbol table to store variable and function names for later stages.

4.2 **Parsing (Grammar Analysis)**

* Uses LL(1) parsing techniques, ensuring efficient top-down syntax analysis.
* Constructs an Abstract Syntax Tree (AST) to represent program structure hierarchically.
* Detects syntax errors, such as missing parentheses or misplaced operators, and attempts recovery strategies.
* Ensures proper nesting of loops, conditions, and function calls for structured execution.

4.3 **Semantic Checking**

* Verifies type compatibility to ensure operations occur between appropriate data types.
* Checks scope rules to prevent conflicts between variable and function names in different contexts.
* Ensures functions are defined before use and their parameters match expected argument types.
* Detects potential runtime errors, such as division by zero or invalid type casting.

4.4 **Intermediate Representation**

* Converts AST into an Intermediate Representation (IR) to act as a bridge between parsing and machine code generation.
* Uses a three-address code (TAC) or Static Single Assignment (SSA) form for efficient optimization.
* Simplifies control flow by flattening nested conditions into sequential operations.
* Applies basic optimizations, such as constant propagation (precomputing constant expressions) and dead code elimination (removing unnecessary instructions).

4.5 **Code Optimization**

* Eliminates redundant operations to improve runtime efficiency.
* Implements loop unrolling to reduce loop overhead and increase execution speed.
* Uses strength reduction techniques to replace expensive operations (e.g., multiplication) with cheaper ones (e.g., bitwise shifts).
* Performs inlining of frequently used small functions to reduce function call overhead.
* Optimizes memory allocation by minimizing redundant memory accesses and unnecessary variable assignments.

4.6 **Machine Code Generation**

* Converts optimized IR into low-level machine code tailored for specific architectures (x86, ARM, etc.).
* Translates high-level control structures like loops and conditionals into efficient jump instructions.
* Allocates CPU registers efficiently to minimize memory usage and improve execution speed.
* Produces an executable binary that can run independently without requiring the source code.

4.7 **Debugging and Error Reporting**

* Provides meaningful error messages that specify error type, location, and suggested fixes.
* Supports step-through debugging, allowing users to execute code line-by-line and inspect variables.
* Implements a logging mechanism to track internal compiler operations for advanced debugging.
* Includes runtime checks to detect memory leaks, uninitialized variables, and out-of-bounds access.

# 5. Other Nonfunctional Requirements

5.1 **Performance Requirements**

* Must be capable of compiling a 1000-line program in under 3 seconds on a modern processor.
* Optimized for multi-core execution, enabling parallel processing for large programs.
* Memory consumption should not exceed 500MB during compilation to ensure efficiency.

5.2 **Safety Requirements**

* Detects infinite loops and provides warnings or automatic halting mechanisms to prevent system crashes.
* Monitors recursion depth to prevent stack overflow errors during function calls.
* Ensures compiled code does not inadvertently access unauthorized memory regions.

5.3 **Security Requirements**

* Restricts access to system resources like files and network interfaces unless explicitly permitted.
* Executes compiled code in an isolated environment (sandboxing) to prevent malicious execution.
* Implements security checks against buffer overflows and unintended memory leaks.

5.4 **Software Quality Attributes**

* Scalability: The compiler should handle large codebases with millions of lines efficiently.
* Extensibility: The modular design allows for easy integration of new language features or optimization techniques.
* Maintainability: The codebase should follow industry best practices to facilitate updates and debugging.

5.5 **Business Rules**

* Released as an open-source project under the MIT License, encouraging community contributions.
* Maintains version control with a well-documented change history to track improvements and bug fixes.
* Ensures user-friendly documentation to lower the barrier for new contributors and users.

6. Other Requirements

* Future plans include adding Just-in-Time (JIT) compilation to improve execution speed.
* A cloud-based compilation option is under consideration to enable remote access and distributed computing.
* A graphical user interface (GUI) version may be developed to provide an intuitive visual representation of compilation stages.
* A built-in testing framework will be integrated to automate validation and debugging of compiled programs.