PROJECT REPORT

Simple Lang Compiler

By Parth Chavda

Abstract

This project focuses on building a simple compiler called SimpleLang, created to understand and demonstrate how real compilers work internally. Everything is developed from scratch in C, including the key parts of a compiler: a lexer that breaks the input into tokens, a recursive-descent parser that checks if the code follows the grammar, a parse tree builder, and a semantic analyzer that handles scopes and a symbol table.

SimpleLang supports a small but meaningful set of features such as variable declarations, assignments, expressions, and basic if statements. The compiler reads the source code, converts it into tokens, builds a parse tree using an LL(1) grammar, and performs semantic checks like verifying identifiers and scope rules. It also provides clear, helpful error messages whenever something goes wrong.

The aim of this project is to create an easy-to-understand yet complete compiler pipeline that highlights the essential concepts behind compiler design. SimpleLang can be used as a learning tool, and it also provides a solid foundation for adding more advanced features later, such as loops, functions, type checking, or even code generation.

Table of Contents

Abstract 2
Table of Contents 3
Table of Figures 5
INTRODUCTION 6
PROJECT OVERVIEW 7
1. SYSTEM ARCHITECTURE 8
    1. Compiler Pipeline 8
    2. Module Overview 10
2. LANGUAGE SPECIFICATION(SIMPLELANG) 11
    1. Lexical Structure: 11
    2. Grammer Rules: 11
3. LEXICAL DESIGN AND IMPLEMENTATION 14
    1. Token Type 14
    2. Tokenization Process: 14
4. PARSER DESIGN AND IMPLEMENTATION 17
    1. Grammer Classification and LL(1) justification: 17
    2. Recursive Decent Parser: 17
    3. Parse Tree construction: 18
    4. Syntax Error Handling: 19
5. SEMANTIC ANALYSIS DESIGN 21
    1. Scope Management: 21
    2. Symbol Table Design: 21
    3. Type Checking Rules: 22
    4. Symetic Error Handling: 23
6. Code Generation implementation 24

1. Instruction Set: 25
2. Code Emission Strategy: 26
3. Register and Memory Handling: 26
7. TESTING AND DEBUGING 27
1. Sample Programs 28
2. Test Results Summary 31
8. CONCLUSION 32
9. REFERENCES 33

Table of Figures

Figure 1.1Compiler Pipeline 9
Figure 1.2 Lexer Flowchart 16
Figure 1.3 Terminal output of Lexical Analysis 29
Figure 1.4 Terminal output of syntax analysis 30
Figure 1.5 Final output of 8 bit cpu 30

INTRODUCTION

Building a compiler from scratch sounds intimidating at first, but once you get into it, you realize it's basically a big puzzle made of smaller, logical pieces. The goal of this project was to design and implement a simple working compiler for a custom 8-bit CPU. Instead of relying on pre-built tools, the focus was on understanding what really happens behind the scenes—how raw text is turned into tokens, how those tokens become meaningful structures, and finally how those structures translate into executable instructions.

To achieve this, the project was divided into stages that mirror the workflow of a real compiler. The lexer handles breaking the input into recognizable tokens. The parser checks whether the structure of the code makes sense. Then semantic analysis verifies whether the program is meaningful and valid. At the end, the code generator converts everything into machine-readable instructions that match the CPU's architecture.

While each component works independently, the real challenge was making sure they all cooperate smoothly. What made this project interesting was not just writing code, but figuring out the logic and relationships between instructions, addresses, symbols, and memory. Overall, the project gave a practical understanding of how compilers function internally, rather than just learning the theory.

PROJECT OVERVIEW

This project centers around building a small but fully functional compiler for a custom language called SimpleLang. The goal wasn't to create something huge or production-ready, but to understand how the main stages of a real compiler actually come together. Instead of just reading theory, this project puts every concept—tokenizing, parsing, validating, and representing code—into practice through working modules.

SimpleLang itself is intentionally minimal. It supports variables, basic expressions, conditional statements, and a clean, predictable syntax. Because the language is small, each part of the compiler can be seen very clearly without getting buried under unnecessary features.

The compiler is organized into well-defined stages. The lexer breaks the input into tokens, the parser checks whether the structure of the program makes sense, and the semantic analyzer handles rules that go beyond syntax, like scope or type consistency. These modules work together in a straightforward pipeline, making it easier to understand how data flows in a compiler.

Overall, the project serves as a hands-on way to explore compiler construction from the ground up. By implementing each step manually, The understanding of compiler deepens.

1. SYSTEM ARCHITECTURE

The overall architecture of the compiler is organized as a simple, step-by-step pipeline. Each stage focuses on a specific part of understanding the input program, starting from reading plain text and ending with a validated structural representation. To keep the design manageable, the compiler is split into separate modules—one for tokenizing input, one for parsing, and one for semantic checks. These components work in sequence, passing their results forward, which keeps the system easy to follow, debug, and extend. The following sections break down how the pipeline flows and how each module contributes to the final output.

1. Compiler Pipeline

- **Source Code Input**
  The compiler receives the raw SimpleLang program exactly as written by the user.
- **Lexical Analysis (Lexer)**
  The lexer scans the source code and breaks it into tokens like identifiers, numbers, operators, and keywords.
- **Syntax Analysis (Parser)**
  The parser checks whether the tokens follow the grammar rules and builds a tree structure that represents the program's layout.
- **Semantic Analysis**
  This stage verifies meaning-related rules, such as valid declarations, scope handling, and detecting undeclared or duplicate variables.
- **Intermediate Representation (Parse Tree / AST)**
  After all checks, the compiler produces a structured form of the program that can be used for testing, interpretation, or later extensions.
- **Code Generation**

The final stage translates the validated structure into low-level instructions for your 8-bit CPU architecture. This is where the compiler produces the actual machine-like code that can run on your designed system.

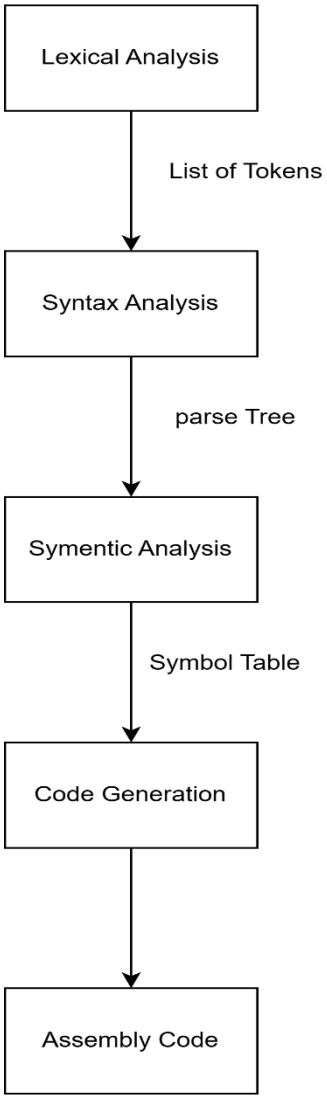

Figure 1.1.1Compiler Pipeline

2. Module Overview

The compiler is organized into a set of modules, each responsible for a specific stage of the compiler . Keeping the system modular makes the codebase easier to maintain, extend, and debug. Below is a quick overview of the major modules and what each one contributes to the workflow:

- **Lexer Module**

  This part scans the raw source code and breaks it into meaningful tokens such as identifiers, numbers, keywords, and operators. It also filters out whitespace so that later stages receive only the elements they need.
- **Parser Module**

  The parser reads the token stream and checks whether it follows the grammar rules of SimpleLang. Using recursive-descent parsing, it builds a structured parse tree that captures the program's shape and relationships between constructs.
- **Semantic Analyzer**

  This module adds meaning to the parsed structure. It manages scopes, tracks declared variables, and validates that identifiers are used correctly. It catches issues like undeclared variables, duplicate declarations, or type mismatches before any code is generated.
- **Code Generator**

  The final module generates assembly-like instructions from parse tree for the custom 8-bit CPU. It translates high-level constructs into low-level operations while keeping the generated code compact and consistent with your CPU design.

2. LANGUAGE SPECIFICATION(SIMPLELANG)

SimpleLang is a tiny language built purely for learning and experimentation. Instead of trying to mimic a full-featured programming language, it sticks to the basics so that the focus stays on how each compiler phase works. The idea is simple: keep the language small enough that anyone can follow what's going on, but expressive enough to feel like real code.

1. Lexical Structure:

SimpleLang uses a straightforward set of tokens, most of which are familiar from typical programming languages:

- **Keywords** — int, if
- **Identifiers** — names for variables
- **Numbers** — integer constants
- **Operators** — +, -, =, ==, !=, <, <=, >, >=
- **Brackets** — ( ) { }
- **Punctuation** — ;

The lexer reads the input program character by character and groups them into these tokens. Anything that doesn't match the expected patterns is treated as an invalid token.

2. Grammer Rules:

The grammar for SimpleLang is intentionally kept small so that each part of the language is easy to follow. It covers four basic building blocks—variable declarations, assignments, arithmetic expressions,

and simple conditional blocks. Below is the complete grammar that the parser uses:

- **Core Features and Patterns:**

I. Variable Declaration:
A variable is introduced using the int keyword.
Eg. int a;

II. Assignment:
A variable can store a value or the result of an expression.
Eg. a=b+c;
a=5;

III. Arithmetic Expressions:
Only + and - are supported, keeping evaluations simple.
Eg. a = b+c-5

IV. Condition Statement:
The language uses a very direct if structure with a condition and a block.
Eg. if(a==5) {
a=a+5;
}

- **Grammer Rules:**
  - program -> statement | epsilon
  - statement -> IfStatement
    | Declaration
    | Assignment
  - IfStatement -> 'if' '(' condition ')' '{' statement '}'
  - Declaration -> 'int' Identifier ';'
  - Assignment -> Identifier '=' expression ';'
  - condition -> expression comparator expression
  - comparator -> '=='

| '>='
| '<='
| '>'
| '<'
| '!='

- expression -> term { ('+' | '-') term }
- term -> Identifier
| Number
| '(' expression ')'

3. LEXICAL DESIGN AND IMPLEMENTATION

1. Token Type

The project defines a fixed set of token types (as seen in TokenType), covering all language elements:

- **Keywords**: int, if
- **Identifiers**: user-defined variable names(Only alphabets)
- **Numbers**: sequences of digits
- **Operators**: +, -, =, ==, !=, <, <=, >, >=
- **Delimiters**: ;, (,), {, }
- **EOF marker**
- **Unknown tokens** (for unexpected or unsupported characters)

Each token is stored in a simple struct:

```
typedef struct {
   TokenType type;
   char value[MAX_TOKEN_LEN];
} Token;
```

The value field preserves the exact characters scanned from the input, which makes debugging, displaying token streams, or building parse error messages much easier.

2. Tokenization Process:

The lexer's main routine, getNextToken(), reads input one character at a time and decides what kind of token to produce. The behavior is intentionally straightforward:

- **Whitespace→**ignored.
  The lexer skips spaces and tabs using isSpace(). Newlines aren't

used for tokenization in SimpleLang, so ignoring them keeps scanning simple.

- **Identifiers & Keywords**
  If a character begins with an alphabetic character, the lexer keeps reading alphanumeric characters until the identifier is fully captured.
  After capturing the lexeme, the code compares it against reserved words:
  - "int" → TOKEN_INT
  - "if" → TOKEN_IF
  - otherwise → TOKEN_IDENTIFIER

  The implementation uses a local buffer (token->value) to collect the characters and then assigns the correct token type.
- **Numbers**
  If the first character is a digit, the lexer continues consuming digits to form a whole number literal. This becomes a TOKEN_NUMBER.
- **Operators & Delimiters**
  Single-character operators like +, -, and delimiters such as parentheses or braces are returned immediately.
  Multi-character operators are handled carefully:
  - == → TOKEN_EQ
  - != → TOKEN_NEQ
  - <= → TOKEN_LTE
  - >= → TOKEN_GTE

  The lexer reads the first character, peeks at the next using fgetc, and applies ungetc when necessary so no input is lost.
- **Unknown characters**
  Anything that doesn't fit the above categories falls back to TOKEN_UNKNOWN.
  This includes the standalone '!' operator since SimpleLang does not define unary '!'.

- **End of File**
  When there is no more input to read, the lexer emits a TOKEN_EOF token.

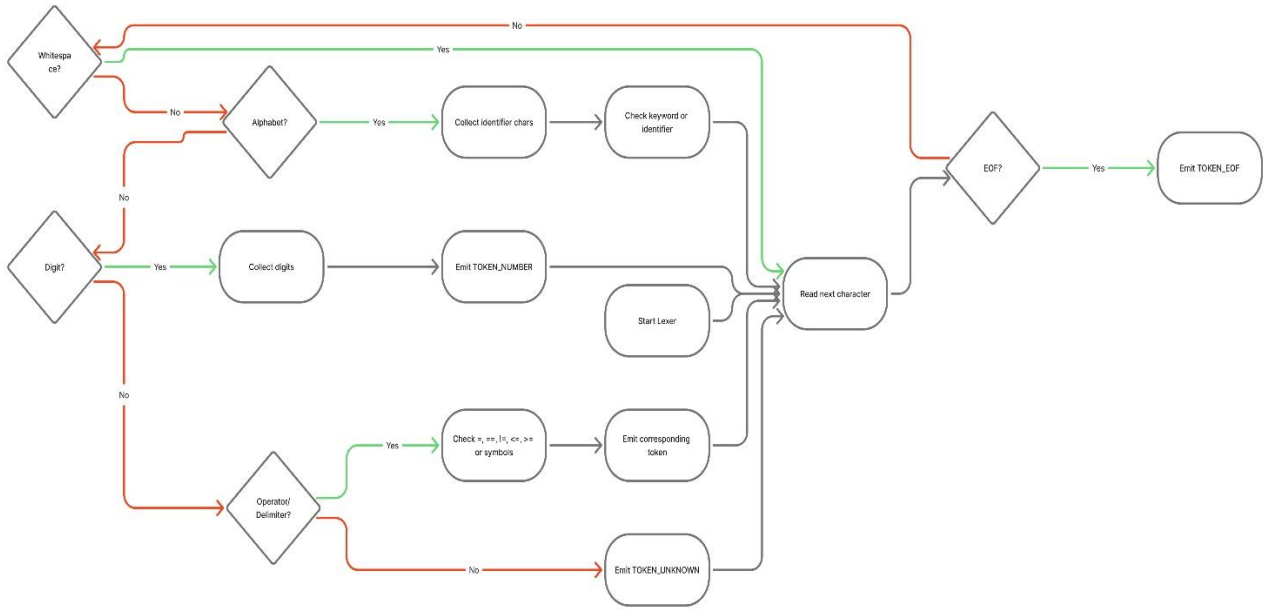

Figure 1.3.1 Lexer Flowchart

4. PARSER DESIGN AND IMPLEMENTATION

1. Grammer Classification and LL(1) justification:

The grammar used for SimpleLang falls under a typical context-free grammar, where every rule expands from a single non-terminal into a structured sequence of tokens. What makes it work especially well for this project is that it naturally fits the behavior of an LL(1) parser.

The rules are arranged so that each choice can be made by looking at just one upcoming token. For example, a statement beginning with if, int, or an identifier leads the parser toward the correct branch without any guessing.

There's no left recursion, no overlapping FIRST sets, and no need for backtracking. Because of that, the grammar aligns perfectly with the recursive-descent approach and keeps the parsing logic clean and predictable.

2. Recursive Decent Parser:

The parser follows a classic recursive-descent structure, where each rule in the grammar maps directly to a function in the code. This design keeps the control flow intuitive: when the grammar says a statement contains an expression or a condition, the parser just calls the corresponding function and continues parsing from there.

Since SimpleLang is small and its grammar is unambiguous, this method works smoothly without needing backtracking. The parser always looks at the current token and knows which rule applies next. For example:

- If the token is int, the parser knows it's dealing with a declaration.
- If the token is an identifier, it expects an assignment.
- If the next token is if, it enters the conditional branch.

The functions themselves follow a natural, readable pattern: consume the expected token, check for errors, and move forward. Any unexpected token is immediately flagged, keeping debugging straightforward.

3. Parse Tree construction:

Every grammar function not only parses the rule — it also builds the corresponding subtree.

- **Node creation mirrors the grammar**
  Each parser function creates a new node using createNode(). Examples:
  Declarations → NODE_DECLARATION
  Assignments → NODE_ASSIGNMENT
  Expressions → NODE_EXPRESSION
  If-statements → NODE_IF
- **Children added as the parser processes input**
  Arguments, identifiers, operators, and nested structures are attached using addChild().
  The sequence in the tree matches the actual parsing order.
- **Expression trees grow naturally**
  For expressions like a + b - c:
  Each operator gets its own node
  Left and right operands become its children
  Parentheses generate nested expression subtrees

No reordering or post-processing is needed.

- **If-statement structure**
  The parser creates a node for the condition and another for the body.
  Example structure:
  IF
  — CONDITION
    — left expression
    — comparator
    — right expression
  — BODY
    — statement1
  — BODY
    — statement2
- **Final output**
  By the end of parsing, the entire program is captured as a clean hierarchy, ready for semantic analysis or code generation.

4. Syntax Error Handling:

The parser enforces the grammar strictly. Any rule mismatch immediately triggers an error through parserError().

- **Immediate detection**
  When the parser expects a specific token and doesn't see it, an error is thrown.
  Examples:
  - Missing ; after declarations or assignments
  - Missing) in conditions
  - Missing { or } around an if-block
  - Comparator not being one of ==, !=, etc.

  Each error prints the specific message and current token.
- **No recovery attempts**
  The parser does **not** insert missing tokens or try to guess what the programmer meant.

It logs the error and continues scanning to find further issues, but the parse tree for the broken part will not be constructed.

- **Error count tracked**
  All errors are counted.If at least one syntax error occurs, the final parse tree is not printed and the parser explicitly reports that errors were found.
- **Protection against corruption**
  Because the parser refuses to match incorrect structures, invalid or partial constructs never enter the parse tree.Only valid parts of the program become part of the tree

5. SEMANTIC ANALYSIS DESIGN

Once the parser confirms that the program is structurally valid, the semantic analyzer steps in to check whether the code actually *makes sense*. This phase works with identifiers, scopes, and rules like "variables must be declared before use." The implementation for SimpleLang stays deliberately small, but still captures the essential behavior of a real compiler.

1. Scope Management:

In SimpleLang, the scope system is built around a simple parent–child structure:

- The program starts with a global scope.
- Any time an if block is entered, a new inner scope is created.
- When the block ends, control returns to the parent scope.

Each scope holds its own set of declared variables. If a variable isn't found in the current scope, the analyzer walks upward through parent scopes until it either finds a match or confirms the variable was never declared.

This approach is lightweight but effective: it ensures variables inside an if block don't leak outward, while still allowing access to outer variables.

2. Symbol Table Design:

The symbols are stored in a linked list fashion. A single symbol structure contains following fields:

- name – variable identifier
- type – data type (int, bool, unknown)
- next – pointer to the next symbol

The global symbol table (sym_table) acts as the final merged table after semantic analysis. There is a function addSymbolToTable(), which store the symbols in sym_table.
In every function call the function adds symbol in front and then updates it header to point to new symbol node.

3. Type Checking Rules:

The semantic analyzer checks this rules while parsing each node in the parser So that there is no mistake in declaration or type mismatch in the source code.

- **Expression Rules**

All arithmetic operations (+ and -) require **integer operands**.
Expressions with operators are validated by checking:

  - left term type
  - operator validity
  - right term type

If any mismatch occurs → semantic error.

- **Condition Rules**

A condition must be:

  - an expression with a valid comparator (==, !=, <, <=, >, >=)
  - both sides must have the **same type**

The resulting type of a condition is always **boolean**.

- **Term Rules**

Identifiers → must be declared
Number → always int
Parenthesized expression → recursively validated
Anything else → invalid term

- **Assignment Rules**

Left side must be a declared identifier
Right side must be an integer expression

Type mismatch → error

- **If Statement Rules**

  Condition must evaluate to boolean

  A new scope is created for the if-block

  All statements inside are validated recursively

4. Symetic Error Handling:

Semantic error handling basically acts as the compiler's way of sanity-checking the program. Even if the syntax looks fine, the compiler still needs to confirm that the code actually makes sense. To do that, it uses a dedicated semanticError() function, which prints messages in a fixed format ("Semantic Error: <message>") whenever something goes wrong.

Every time this function runs, it also increments a counter, so instead of stopping at the first mistake, the compiler keeps going and reports as many issues as it can in a single pass. This makes it a lot easier for the programmer to spot all the problems at once.

Most of these errors show up when the meaning of the code doesn't line up with the rules of the language. For example, using a variable that was never declared, declaring the same variable twice, assigning a value of the wrong type, or even using operators that don't fit the expression.

Also Conditions are checked to ensure they behave like actual boolean expressions, and the compiler verifies that assignments match the type that the variable was originally declared with.

All of this ties directly into how the symbol table and scope system work. If a lookup fails or a symbol clashes with an existing one, the compiler raises an error right away.

By the time the semantic phase finishes, it prints the total number of errors found, giving a clear picture of whether the program is safe to move ahead or needs to be fixed before any further processing.

6. Code Generation implementation

The code generation stage is where the compiler turns the validated AST into executable assembly for the target 8-bit CPU. Up to this point, the program has only existed as a set of tokens, parsed structures, and semantic rules. Code generation gives it concrete form. The goal here is not to optimize or restructure the program, but to translate each construct into a direct sequence of instructions that the CPU can follow.

Every AST node corresponds to a predictable emission pattern, and each pattern maps cleanly onto the instruction set provided by the hardware. Expressions are evaluated using the accumulator model of the CPU, statements are expanded into linear instruction sequences, and variables are represented as simple memory locations in the data section. Control flow is managed with compiler-generated labels and conditional jumps, allowing if-statements to branch cleanly without additional machinery.

The resulting assembly stays close to the structure of the source program. By avoiding unnecessary transformations and relying on a small, consistent subset of instructions, the code generator remains easy to reason about and aligns well with the design of the underlying processor. The following sections describe the instruction set involved, the emission approach used for each construct, and the rules the backend follows for managing registers and memory.

1. Instruction Set:

This compiler targets the 8-bit CPU from the open-source project at *lightcode/8bit-computer* on github. Although the processor provides a broad instruction set, the backend only relies on a small, practical subset of it. This subset is enough to support assignments, arithmetic expressions, stack-based evaluation, and conditional jumps.

The data-movement instructions used are:

- **ldi r D** — loads an immediate value into a register, typically used for putting constants into A during expression evaluation.
- **lda %var** — loads a variable's content into A by referencing its label.
- **sta %var** — stores the accumulator value into a variable's memory slot.
- **push r / pop r** — used heavily during expression reduction, letting the code generator preserve intermediate values across multi-operator expressions.

The arithmetic instructions used include:

- **add** and **sub**, which operate on A and B, producing their result back into A.

For branching, the backend emits:

- **jz Lx**, **jnz Lx**, **jc Lx**, **jnc Lx** depending on the meaning of the comparison.
- A plain **jmp Lx** for unconditional transfers.

Finally:

- **cmp** is used to compare A and B before generating conditional jumps.
- **hlt** marks the end of execution.

2. Code Emission Strategy:

The code generator translates one Parser node at a time and let the CPU's flags and accumulator register do all the heavy lifting. The .text section starts with a single entry label (start), flows straight through all statements, and ends by halting the machine. The .data section is created afterwards by enumerating symbols and giving each a memory slot initialized to zero.

Assignments have following pattern, evaluate the expression into A, then store it. There's no temporary register management beyond push/pop, because the backend leans entirely on the stack when expressions get deeper than a single operator. Multi-operator expressions are flattened by evaluating the left side first, pushing A, evaluating the right side, popping into B, performing the operation, and repeating.

Conditional statements are handled by generating a pair of labels and creating a jump sequence based on the comparison outcome. The condition is evaluated using a standard pattern: compute left expression, push it, compute right expression, pop into B, issue a cmp, and choose the appropriate jump instructions based on the operator.

3. Register and Memory Handling:

Everything revolves around just two general-purpose registers — A, B and the stack. The accumulator A is the "landing zone" for nearly every expression; whenever an expression is reduced, the final value must reside in A. B is only used as a scratch register, almost always in the pattern (left in B, right in A) before performing arithmetic or comparison.

Variables reside in the .data section as simple byte-sized memory cells. Accessing them is done by referencing their label names directly through lda %name or sta %name. There is no pointer arithmetic, offset

addressing, or indirect loads/stores; every variable maps to a flat memory slot.

The stack saves intermediate results. The sequence of push/pop avoids the need for a register allocation algorithm by the compiler. This, in turn, makes the code generator predictable: A holds the value of the current expression, B is loaded only from a popped value, and memory is touched only to read or write variables.

7. TESTING AND DEBUGING

Testing and debugging were performed using the compiler's own runtime diagnostics and by manual inspection of intermediate artifacts. The process is lightweight and iterative:

- **Stage-by-stage inspection.** The driver (main.c) intentionally prints the token stream after lexing and prints the parser tree (printParser) after parsing. These outputs are the primary verification points used to confirm lexer and parser behavior.
- **Semantic checks gating codegen.** The compiler runs semantic analysis only when parser_err_count == 0. Semantic errors are emitted via semanticError(...) and counted in semantic_err_count. Code generation is skipped if semantic errors exist, which prevents generating assembly from an invalid AST.
- **Assembly verification.** When code generation runs the compiler writes a .s file containing a .text section with instructions and a .data section listing declared symbols initialized to 0. Debugging consists of inspecting this file to make sure emitted instructions reflect the AST.
- **Resource and failure reporting.** The code prints explicit messages for allocation failures (e.g., Memory Reallocation Failed) and exits deterministically; these messages were used to detect and fix memory-handling bugs during development.

1. Sample Programs

The compiler was tested with the following sample program:

```
int b;
int a;
b = 2;
a = 5;
if (a > 5) {
   int c;
   c = a;
}
```

This program demonstrates variable declarations, basic assignments, and a conditional block. The compiler handles each declaration by allocating a memory slot in the symbol table. The assignments produce simple load–store sequences using LDI, MOV, and STA instructions. The if condition generates a comparison followed by a conditional jump based on the CPU's limited branch instruction support.

Since the compiler does not support complex expressions or nested blocks beyond the provided features, this sample program remains within the functional scope of the language and tests the system's ability to manage scopes, symbol creation, and code emission for a conditional branch.

The following are the images of terminal after running the code :

```
parth123@Parth:~/SimpleLang_Compiler$ make run INPUT=examples/input.txt OUTPUT=output.asm
./program examples/input.txt output.asm
Processing Lexical Analysis...
0) Token: 0, Value: int
1) Token: 1, Value: b
2) Token: 13, Value: ;
3) Token: 0, Value: int
4) Token: 1, Value: a
5) Token: 13, Value: ;
6) Token: 1, Value: b
7) Token: 6, Value: =
8) Token: 2, Value: 2
9) Token: 13, Value: ;
10) Token: 1, Value: a
11) Token: 6, Value: =
12) Token: 2, Value: 5
13) Token: 13, Value: ;
14) Token: 3, Value: if
15) Token: 14, Value: (
16) Token: 1, Value: a
17) Token: 10, Value: >
18) Token: 2, Value: 5
19) Token: 15, Value: )
20) Token: 16, Value: {
21) Token: 0, Value: int
22) Token: 1, Value: c
23) Token: 13, Value: ;
24) Token: 1, Value: c
25) Token: 6, Value: =
26) Token: 1, Value: a
27) Token: 13, Value: ;
28) Token: 17, Value: }
29) Token: 18, Value:
```

Figure 7.1 Terminal output of Lexical Analysis

```
Syntax Analysis complete
   ROOT
      BODY
         DECLARATION(int)
            IDENTIFIER(b)
      BODY
         DECLARATION(int)
            IDENTIFIER(a)
      BODY
         ASSIGNMENT(b)
            IDENTIFIER(b)
            EXPRESSION(2)
               NUMBER(2)
      BODY
         ASSIGNMENT(a)
            IDENTIFIER(a)
            EXPRESSION(5)
               NUMBER(5)
      BODY
         IF(if)
            CONDITION(a)
               EXPRESSION(a)
                  IDENTIFIER(a)
               COMPARATOR(>)
               EXPRESSION(5)
                  NUMBER(5)
            BODY
               DECLARATION(int)
                  IDENTIFIER(c)
            BODY
               ASSIGNMENT(c)
                  IDENTIFIER(c)
```

Figure 7.2 Terminal output of syntax analysis

```
parth123@Parth:~/SimpleLang_Compiler/8bit-computer$ make clean && make run
rm -rf computer
iverilog -o computer -Wall \
        rtl/alu.v rtl/cpu.v rtl/cpu_control.v rtl/cpu_registers.v rtl/machi
        rtl/library/clock.v rtl/library/counter.v rtl/library/ram.v rtl/lib
        rtl/tb/machine_tb.v
vvp -n computer
VCD info: dumpfile machine.vcd opened for output.
Memory: set [0x1a] => 0x02 (  2)
Memory: set [0x1b] => 0x05 (  5)
Memory: set [0xff] => 0x05 (  5)
Memory: set [0x1c] => 0x05 (  5)
==========================================
CPU halted normally.
REGISTERS: A: 05, B: 05, C: xx, D: xx, E: xx, F: xx, G: xx, Temp: xx
rtl/tb/machine_tb.v:54: $stop called at 2040 (1s)
```

Figure 7.3 Final output of 8 bit cpu

2. Test Results Summary

The following summarizes observed behavior from iterative, manual testing of small inputs during development:

- **Lexical stage**
  - Identifiers, numbers, single-character tokens and multi-character relational tokens (==, !=, >=, <=) were tokenized correctly and printed in order.
  - Token buffer truncation prevented overflow; excessively long lexemes are truncated to MAX_TOKEN_LEN.
- **Parsing stage**
  - The parser constructed readable ASTs for supported constructs (declarations, assignments, expressions, if blocks). printParser output matched expected node shapes.
  - Syntax errors were detected and reported with contextual messages and increased parser_err_count.
- **Semantic stage**
  - The semantic pass built a symbol list for declared variables and correctly detected semantic errors such as undeclared-variable use, redeclaration, and type mismatches. If semantic errors were present, the pipeline stopped before code generation.
- **Code generation**
  - When no semantic errors existed the generator produced assembly that directly reflected the AST:
    - Constants loaded with ldi A <value>, variables loaded with lda %name, and stored with sta %name.
    - Multi-term expressions were evaluated via push A / pop B with add / sub.
    - Comparisons used cmp followed by conditional jumps chosen according to the comparator.
    - A terminating hlt instruction was appended; .data listed declared variables initialized to 0.

8.CONCLUSION

Working on this compiler turned out to be a lot more practical than I initially expected. Even though the language I designed is tiny, getting all the pieces—lexing, parsing, semantic checks, and finally pushing out instructions for my 8-bit CPU—forced me to understand how each stage depends on the next.

The whole pipeline runs end-to-end now, and it handles the core things I wanted: integer variables, assignments, basic expressions, comparisons, and an if block with its own scope.The language itself is very limited on purpose, but the project still gave me a solid foundation.

Now that everything works in a basic form, it's much clearer what could be added next: loops, functions, smarter register handling, maybe even some lightweight optimizations. More than anything, the project made me appreciate how messy real compiler work can get. Scope handling, catching simple mistakes, and generating correct low-level instructions were the trickiest parts, but also the ones that taught me the most.

9.REFERENCES

- *www.Figma.com for flowchart*
- Target CPU design and instruction semantics used as reference: *lightcode/8bit-computer* — https://github.com/lightcode/8bit-computer
- Project source files used in this report:
  - main.c
  - lexer.h, lexer.c
  - parser.h, parser.c
  - semantic.h, semantic.c
  - codegen.h, codegen.c