Team Name Dalton’s Atom

Spreadsheet Formula Compiler

Course Project - Compiler Design
Faculty - Dr. Maithilee Laxmanrao Patawar (Assistant Professor)
Prepared By - Priyank Jain

Contents

This is a complete, end-to-end compiler for a spreadsheet formula language, built from scratch in C using Flex (Lex) and Bison (Yacc).

*It takes a string formula (like =A1 + B2 * 3) as input and processes it through a 7- phase pipeline, including lexical analysis, parsing, semantic analysis, code generation, optimization, and finally, execution by two different back-ends: a tree-walk interpreter and a stack-based virtual machine.*

1

Meet the Compiler

Key Features

1

Full Parsing Pipeline

Implements all major phases of a modern compiler.

2

Rich Grammar

Supports arithmetic (+, -, *, /, ^), logic (AND, OR, NOT), comparisons (>, <, ==), and nested parentheses.

3

Built-in Functions

IF, SUM, AVERAGE, MIN, MAX.

4

Robust Semantic Analysis

Detects undefined cells, type mismatches, circular dependencies, and invalid function arguments.

5

And Many More…

Bytecode Generation, Optimization, Dual Execution Back-Ends, AST Interpreter, Virtual Machine, Verbose Debugging

Phases Of Compiler

The compiler processes input through the following stages, all of which can be visualized with the --verbose flag:

1&2

Parsing (Lexer & Parser)

- src/lexer.l (Flex) turns the input string into a stream of tokens (e.g.,NUMBER, CELL_REF, PLUS).
- src/parser.y (Bison) organizes these tokens into a valid grammatical structure.

3

Abstract Syntax Tree (AST)

- The parser builds an in-memory tree (ASTNode) that represents the formula logic.
- ast_printer.c can print this tree in three formats: tree (default), dot, or lisp.

4

Semantic Analysis

- semantic.c traverses the AST to find logical errors.
- symtab.c (Symbol Table) is used to look up cell values and track dependencies.
- error.c reports any issues, such as Error: Undefined cell reference: 'B99'

Phases Of Compiler

The compiler processes input through the following stages, all of which can be visualized with the --verbose flag:

5

Code Generation & Optimization

- codegen.c traverses the AST and generates an intermediate representation (stack-based bytecode).
- ir.c defines the bytecode instructions (e.g., OP_PUSH, OP_ADD, OP_HALT).
- optimizer.c can (optionally) clean up this bytecode.

6

Execution

- The compiler can execute the formula using one of two methods:
- interpreter.c (Method 1) walks the AST directly.
- vm.c (Method 2) executes the generated bytecode on a stack-based virtual machine.runtime.c provides the core logic for built-in functions (e.g., rt_sum) used by both methods.

7

Testing

- run_tests.sh provides a complete test suite to validate all compiler functionality.

Project Structure

```
spreadsheet-compiler/
├── bin/
│   ├── compiler      (The final executable)
│   └── test_lexer    (Standalone lexer tester)
├── obj/
│   └── (Object files .o and generated .c/.h files)
├── src/
│   ├── ast.h
│   ├── ast_printer.c
│   ├── ast_printer.h
│   ├── codegen.c
│   ├── codegen.h
│   ├── error.c
│   ├── error.h
│   ├── interpreter.c
│   ├── interpreter.h
│   ├── ir.c
│   ├── ir.h
│   ├── lexer.l
│   ├── Makefile
│   ├── optimizer.c
│   ├── optimizer.h
│   ├── parser.y
│   ├── runtime.c
│   ├── runtime.h
│   ├── semantic.c
│   ├── semantic.h
│   ├── symtab.c
│   ├── symtab.h
│   ├── value.h
│   ├── vm.c
│   └── vm.h
├── tests/
│   ├── cells/
│   │   └── base_cells.txt
│   ├── execution/
│   ├── semantic/
│   └── syntax/
├── Makefile
├── README.md
└── run_tests.sh
```

How to Build

A comprehensive Makefile handles all compilation and dependencies.

```
# Clean up all old object files and binaries
make clean

# Build the final 'bin/compiler' executable
make
```

How to Test

A BASH-based test suite is provided. This script will automatically build the compiler and run all tests.

```
# Make the script executable (only need to do this once)
chmod +x run_tests.sh

# Run the full test suite
./run_tests.sh
```

Usage (Command-Line Flags)

The compiler reads a formula from stdin or an input file and accepts several flags to control its behavior.

```
Usage: ./bin/compiler [options]
```

Example 1: Simple Execution

This pipes a formula to the compiler and gets the result. Uses default cell values:
A1=10, A2=20

```
$ echo "=A1 + 5" | ./bin/compiler
Setting up test environment...
Parsing formula...
...
VM Result: 15.000000
```

Example 2: Verbose Output

Use input files and all verbose flags to see the full compilation pipeline.

formula.txt: =A1 + B2 * 3

cells.txt: A1=10 B2=7

```
$ ./bin/compiler --input formula.txt --cells cells.txt --verbose --ast-tree
--bytecode --trace
======================================
 SPREADSHEET FORMULA COMPILER v1.0
======================================
Input Formula: =A1 + B2 * 3
✓ Loading cell data from: cells.txt
✓ Reading formula from: formula.txt

=== PHASE 1 & 2: PARSING ===
✓ Parse tree constructed
✓ No syntax errors detected

=== ABSTRACT SYNTAX TREE ===
AST VISUALIZATION
└── BINARY_OP (+)
    ├── CELL_REF (A1)
    └── BINARY_OP (*)
        ├── CELL_REF (B2)
        └── NUMBER (3.000000)

=== PHASE 4: SEMANTIC ANALYSIS ===
SYMBOL TABLE
Cell | Value    | Status
-----|----------|----------
B2   | 7.00     | DEFINED
A1   | 10.00    | DEFINED
✓ Semantic analysis passed!

=== PHASE 5: CODE GENERATION ===
STACK-BASED BYTECODE
--- Bytecode ---
0000: PUSH_CELL A1
0001: PUSH_CELL B2
0002: PUSH 3.000000
0003: MUL
0004: ADD
0005: HALT
---------------

=== PHASE 6: EXECUTION ===
EVALUATION RESULTS

Method 1: Direct AST Interpretation
Stack Trace:
Evaluating NODE_BINARY_OP
  Evaluating NODE_CELL(A1) = 10.00
  Evaluating NODE_BINARY_OP
    Evaluating NODE_CELL(B2) = 7.00
    Evaluating NODE_NUMBER = 3.00
Result: 31.000000
RESULT: 31.000000

Method 2: Virtual Machine Execution
--- VM TRACE ---
0000: 0000: PUSH_CELL A1
    STACK: [ ]
...
0005: 0005: HALT
    STACK: [ 31 ]
--- END TRACE ---
RESULT: 31.000000

======================================
 COMPILATION SUMMARY
======================================
Status:       SUCCESS
Tokens:       6
AST Nodes:    5
Instructions: 6
```

All Options

Flag	Description
--input <file>	Read formula from <file>.
--cells <file>	Load cell values from <file>.
--mode=ast	Execute using the**AST Interpreter** .
--mode=vm	Execute using the**Virtual Machine**(Default).
--ast-tree	Show AST as a tree (box-drawing).
--ast-dot	Show AST in Graphviz .dot format.
--ast-lisp	Show AST in Lisp S-expression format.
--no-ast	Do not print the AST.
--bytecode	Show the generated stack-based bytecode.
--trace	Show VM/Interpreter execution trace.
--optimize	Enable bytecode constant-folding optimization.
--verbose	Show all compilation phase headers.
--help	Show this help message.