

- [Spreadsheet Formula Compiler](#)
 - [Features](#)
 - [Phases of Compilation](#)
 - [Project Structure](#)
 - [How to Build](#)
 - [How to Test](#)
 - [Usage \(Command-Line Flags\)](#)
 - [Example 1: Simple Execution](#)
 - [Example 2: Verbose Output \(The "Screenshot" Mode\)](#)
 - [All Options](#)

Spreadsheet Formula Compiler

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.

Features

- **Full Parsing Pipeline:** Implements all major phases of a modern compiler.
- **Rich Grammar:** Supports arithmetic (+, -, *, /, ^), logic (AND, OR, NOT), comparisons (>, <, ==), and nested parentheses.
- **Built-in Functions:** IF, SUM, AVERAGE, MIN, MAX.
- **Robust Semantic Analysis:** Detects undefined cells, type mismatches, circular dependencies, and invalid function arguments.
- **Bytecode Generation:** Compiles formulas into a custom stack-based bytecode.
- **Optimization:** Includes a constant-folding optimizer (`--optimize`) to pre-calculate parts of the formula at compile time.
- **Dual Execution Back-Ends:**
 1. **AST Interpreter (`--mode=ast`):** Evaluates the formula by directly walking the Abstract Syntax Tree.

- 2. **Virtual Machine (- -mode=vm)**: Executes the generated bytecode on a stack-based VM.
- **Verbose Debugging**: Provides flags to visualize every step, including the AST, bytecode, and a full execution trace.

Phases of Compilation

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

1. Phase 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.

2. Phase 3: Abstract Syntax Tree (AST)

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

3. Phase 4: Semantic Analysis

- `semantic.c` traverses the AST to find logical errors.
- `syntab.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'`.

4. Phase 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.

5. Phase 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.

6. Phase 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, etc.)

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

Example 2: Verbose Output (The "Screenshot" Mode)

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 theAST Interpreter .
--mode=vm	Execute using theVirtual Machine(Default).
--ast-tree	Show AST as a tree (box-drawing).
--ast-dot	Show AST in Graphviz .dot format.

Flag	Description
<code>--ast-lisp</code>	Show AST in Lisp S-expression format.
<code>--no-ast</code>	Do not print the AST.
<code>--bytecode</code>	Show the generated stack-based bytecode.
<code>--trace</code>	Show VM/Interpreter execution trace.
<code>--optimize</code>	Enable bytecode constant-folding optimization.
<code>--verbose</code>	Show all compilation phase headers.
<code>--help</code>	Show this help message.