COMPILER

Ioannis Michalainas

March-June 2025

Repository:
github.com/ioannisam/COMPILER

Abstract

This project presents the design and implementation of a compiler for a simple C-style programming language, developed as part of the Programming Languages and Compilers course under Professor *Panagiotis Katsaros*. This project serves as an educational tool.

Contents

1 Project Structure

1.1 Directory Layout

This project is organized for modularity and clarity.

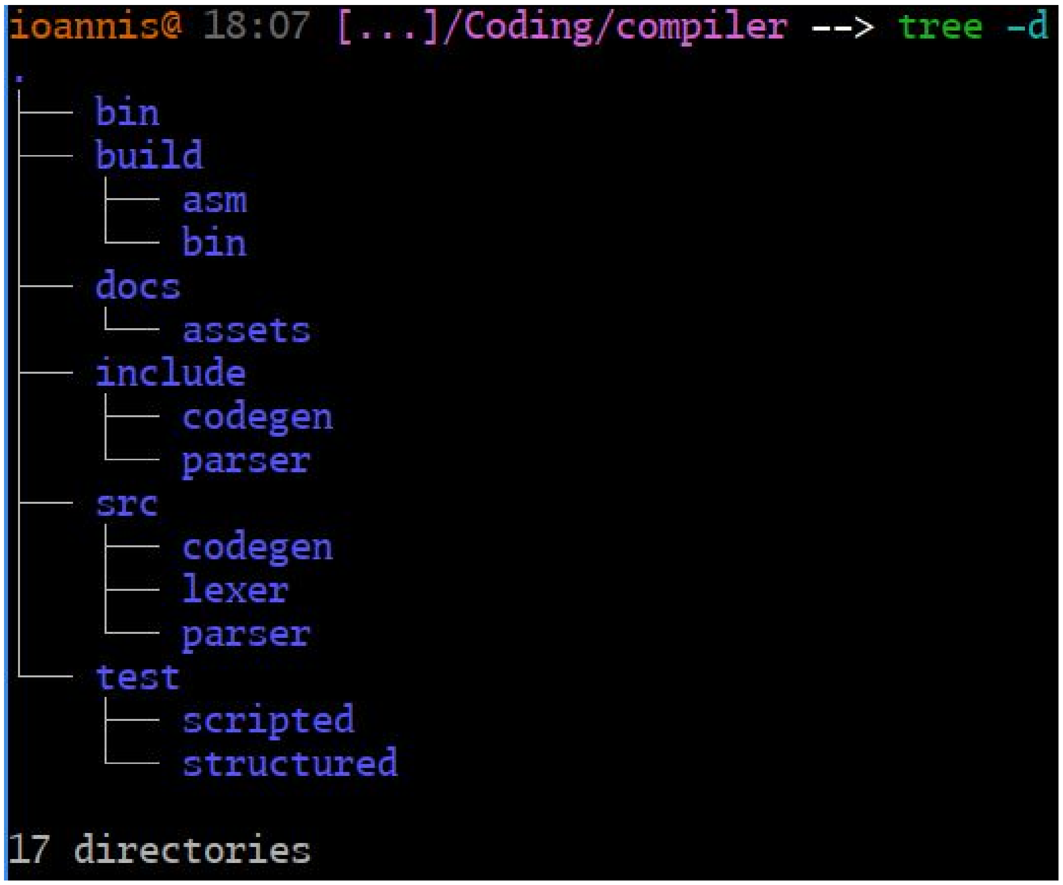
main branch

The root directory (./) contains general files such as `README.md`, `LICENSE`, `TODO` and the build script `utils.sh`. Source code is placed under `src/`, with further separation into `lexer/` for lexical analysis, `parser/` for parsing and AST logic and `codegen/` for code generation, symbol table management and helpers. Header files are located in the `include/` directory, mirroring the source structure.

During the build process, the `bin/` directory is created to store the compiler binary, while `build/` holds generated assembly and compiled binaries. The `test/` directory contains test cases, divided into `scripted/` (programs without a `main` function) and `structured/` (programs with a `main` function and additional functions). Documentation is maintained in the `docs/` directory, which also includes assets such as diagrams.

MIXAL branch

The MIXAL branch has identical file structure, but instead of assembly x86, MIXAL code is generated. To compile said code mdk is used through a docker image for compatibility.[1]



```
ioannis@ 18:07 [...]/Coding/compiler --> tree -d
.
├── bin
├── build
│   ├── asm
│   └── bin
├── docs
│   └── assets
├── include
│   ├── codegen
│   └── parser
├── src
│   ├── codegen
│   ├── lexer
│   └── parser
└── test
    ├── scripted
    └── structured

17 directories
```

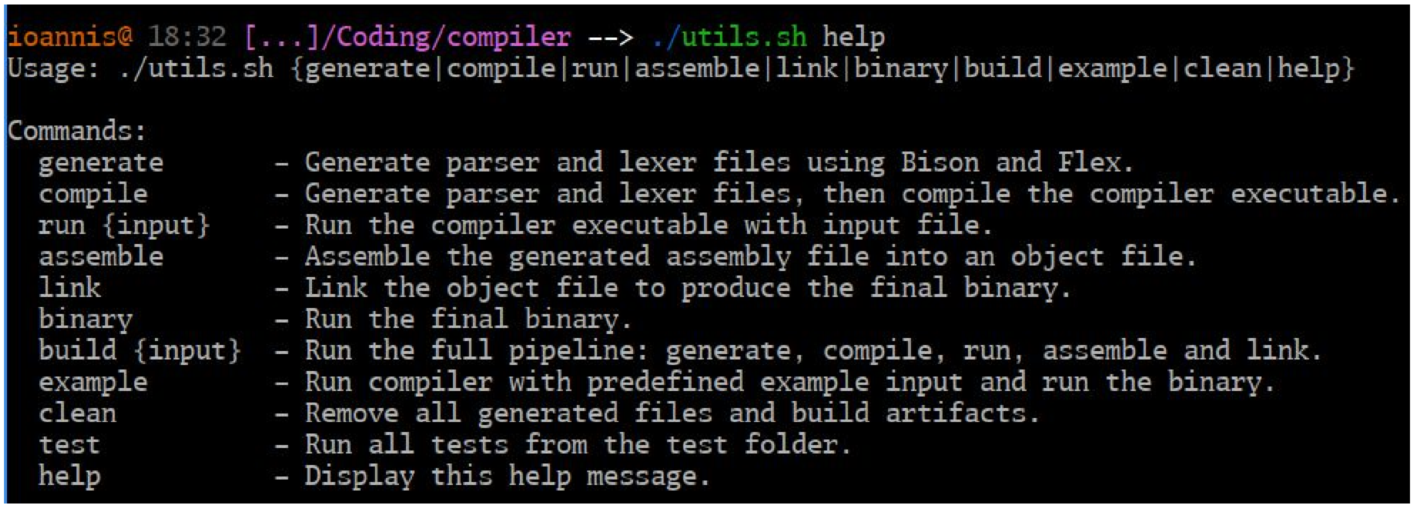
Figure 1: File Structure

1.2 Build and Utilities

The build process is managed by the `utils.sh` bash script, which provides a unified interface for generating, compiling, assembling, linking and testing the compiler. The script supports several commands:

- `generate` invokes Bison and Flex to produce the parser and lexer source files.
- `compile` builds the compiler executable, placing it in the `bin/` directory.
- `run` executes the compiler on a given input file, generating assembly code in `build/asm/`.
- `assemble` uses NASM to assemble the generated assembly into an object file.
- `link` produces the final binary using the system linker.
- `binary` runs the compiled program.
- `build` runs the full pipeline: generation, compilation, code generation, assembly, and linking.
- `example` demonstrates the pipeline using a predefined test case.
- `test` automatically runs all test cases in the `test/` directory, reporting results for each.
- `clean` removes all generated files and build artifacts.
- `help` displays usage information.

For the MIXAL branch, the build script provides the same functionality but uses different tools, such as `docker` for system compatibility, `mixasm` instead of NASM for assembly and `mixvm` to run the generated programs.



```
ioannis@ 18:32 [...]/Coding/compiler --> ./utils.sh help
Usage: ./utils.sh {generate|compile|run|assemble|link|binary|build|example|clean|help}

Commands:
  generate       - Generate parser and lexer files using Bison and Flex.
  compile        - Generate parser and lexer files, then compile the compiler executable.
  run {input}    - Run the compiler executable with input file.
  assemble       - Assemble the generated assembly file into an object file.
  link           - Link the object file to produce the final binary.
  binary         - Run the final binary.
  build {input}  - Run the full pipeline: generate, compile, run, assemble and link.
  example        - Run compiler with predefined example input and run the binary.
  clean          - Remove all generated files and build artifacts.
  test           - Run all tests from the test folder.
  help           - Display this help message.
```

Figure 2: Utilities

2 Lexical Analysis

2.1 Lexer Implementation (`lang.l`)

The lexer is implemented using Flex. It recognizes keywords, identifiers, literals, operators and delimiters. Whitespace and comments are ignored. Each token is returned to the parser with its semantic value, such as integer values for numbers or string pointers for identifiers and string literals.

2.2 Token Definitions

Tokens include keywords (`int`, `main`, `if`, `else`, etc.), operators (arithmetic, logical, bitwise), delimiters (parentheses, braces, semicolons, commas), literals (numbers, strings) and identifiers. Each token is associated with a unique value for the parser to distinguish during syntax analysis.

3 Syntax Analysis

3.1 Parser Implementation (`parser.y`)

The parser is written in Bison and defines the grammar for the language. It supports both structured (function-based) and scripted (block-based) modes. The parser constructs an abstract syntax tree (AST) representing the program structure, including functions, statements, expressions and control flow constructs.

3.2 AST Representation (`ast.c`)

The AST is composed of nodes for each language construct, such as declarations, assignments, function calls, control flow and expressions. Each node type is represented by a C struct, with fields for child nodes and relevant data. Helper functions are provided for constructing and manipulating AST nodes.

3.3 Error Handling

Syntax errors are reported with line numbers and context using the `yyerror` function. The parser attempts to recover from errors where possible, allowing multiple errors to be reported in a single run. Accurate newline tracking is essential for providing precise line number information in error messages..

4 Semantic Analysis

4.1 Symbol Table Management (`symbol.c`)

A symbol table is maintained to track declared variables and function parameters. Each symbol stores its name, type and a unique label for code generation. The symbol table is populated during AST traversal before code generation, ensuring all variables are declared and accessible.

4.2 Semantic Error Reporting

Semantic errors, such as undeclared variables or redefinitions, are detected during symbol table operations. Errors are reported with descriptive messages and halt compilation if encountered.

5 Code Generation

5.1 Assembly x86 (`codegen/`)

The code generator traverses the AST and emits x86-64 assembly code. The output is divided into `.data`, `.bss` and `.text` sections. String literals for print statements are stored in the data section, while variables are allocated in the BSS section. The text section contains the program logic, including function prologues, arithmetic operations, control flow and system calls for I/O and program exit. An `itoa` routine is included for integer-to-string conversion during print operations. The entry point is either the `main` function or a top-level block, depending on the program mode.

5.2 MIXAL (`codegen/`)

The MIXAL backend emits assembly code for the MIX architecture, mirroring the structure of the x86 backend but with MIX-specific conventions.[3] String literals for print statements are defined using the `ALF` directive and split into 5-character blocks. Variables and temporaries are declared at the end of the program with the `CON` directive, rather than in a separate BSS section.[2]

Program logic is organized under a `START` label, which either jumps to the `main` function or executes a top-level block, depending on the program mode. Arithmetic, logic, and control flow are implemented using MIXAL instructions (`ADD`, `SUB`, `MUL`, `DIV`, etc.), with conditional jumps and unique labels for flow control. Function calls and returns use explicit jump instructions and temporary storage for return addresses.

It is important to note that MIX is an imaginary computer and therefore the capabilities and support of its corresponding assembly language, MIXAL, are limited. As a result, the MIXAL version supports far fewer features than the x86 assembly version of this compiler.

6 Testing and Validation

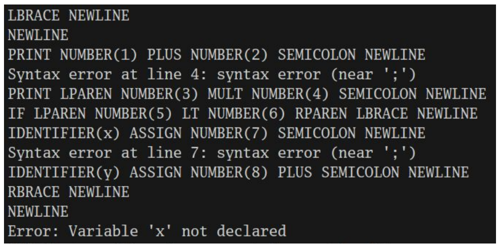
6.1 Scripted Tests

Scripted tests consist of code blocks without explicit function definitions. These are parsed and executed as the main program body. Test cases validate language features such as variable declarations, assignments, arithmetic and print statements.

Test Cases

syntax-erros.txt

```
{
    // deliberate errors
    print 1 + 2;
    print (3 * 4;
    if (5 < 6) {
        x = 7;
        y = 8 +;
    }
}
```



```
LBRACE NEWLINE
NEWLINE
PRINT NUMBER(1) PLUS NUMBER(2) SEMICOLON NEWLINE
Syntax error at line 4: syntax error (near ';')
PRINT LPAREN NUMBER(3) MULT NUMBER(4) SEMICOLON NEWLINE
IF LPAREN NUMBER(5) LT NUMBER(6) RPAREN LBRACE NEWLINE
IDENTIFIER(x) ASSIGN NUMBER(7) SEMICOLON NEWLINE
Syntax error at line 7: syntax error (near ';')
IDENTIFIER(y) ASSIGN NUMBER(8) PLUS SEMICOLON NEWLINE
RBRACE NEWLINE
NEWLINE
Error: Variable 'x' not declared
```

Figure 3: Error Output

precedence.txt

```
{
    // output should be -2
    print "-5 + 3*(2-1) - 5%2 + 3/2 = ";
    print -5 + 3*(2-1) - 5%2 + 3/2;
}
```

OUTPUT: -2

break-nested.txt

```
{
    int x = 0;
    while (x<2) {

        while (1) {
            print "In the nest...";
            break;
            print "...after the break";
        }
        x = x+1;
    } // should print the first message twice
}
```

OUTPUT: In the nest...
In the nest...

6.2 Structured Tests

Structured tests use explicit function definitions, including a `main` function as the entry point. These tests cover function calls, parameter passing, return values, and control flow constructs. Both types of tests ensure correctness of parsing, semantic analysis, and code generation.

Test Cases

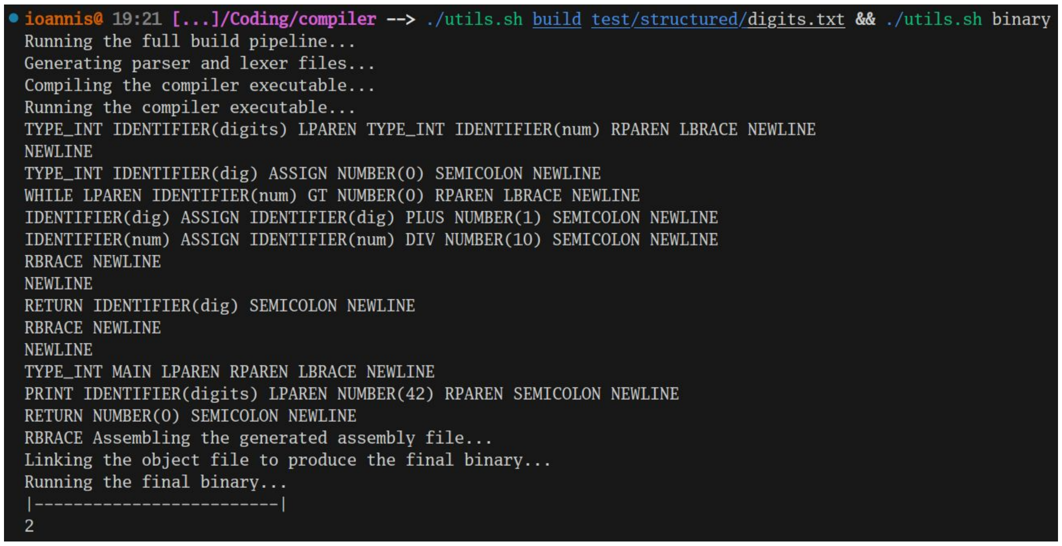
main.txt

```
int main() {
    int a = 4;
    print a;

    int b;
    b = a+10;
    return b;
}
```

OUTPUT: 4 [ble: exit 14]

digits.txt

```
int digits(int num) {

    int dig = 0;
    while (num > 0) {
        dig = dig + 1;
        num = num/10;
    }

    return dig;
}

int main() {
    print digits(42);
    return 0;
}
```

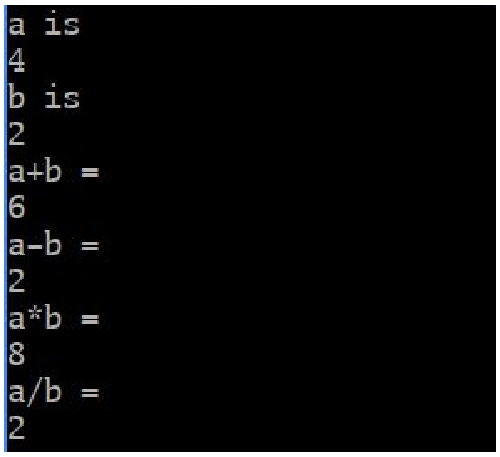


```
● ioannis@ 19:21 [...]/Coding/compiler --> ./utils.sh build test/structured/digits.txt && ./utils.sh binary
  Running the full build pipeline...
  Generating parser and lexer files...
  Compiling the compiler executable...
  Running the compiler executable...
  TYPE_INT IDENTIFIER(digits) LPAREN TYPE_INT IDENTIFIER(num) RPAREN LBRACE NEWLINE
  NEWLINE
  TYPE_INT IDENTIFIER(dig) ASSIGN NUMBER(0) SEMICOLON NEWLINE
  WHILE LPAREN IDENTIFIER(num) GT NUMBER(0) RPAREN LBRACE NEWLINE
  IDENTIFIER(dig) ASSIGN IDENTIFIER(dig) PLUS NUMBER(1) SEMICOLON NEWLINE
  IDENTIFIER(num) ASSIGN IDENTIFIER(num) DIV NUMBER(10) SEMICOLON NEWLINE
  RBRACE NEWLINE
  NEWLINE
  RETURN IDENTIFIER(dig) SEMICOLON NEWLINE
  RBRACE NEWLINE
  NEWLINE
  TYPE_INT MAIN LPAREN RPAREN LBRACE NEWLINE
  PRINT IDENTIFIER(digits) LPAREN NUMBER(42) RPAREN SEMICOLON NEWLINE
  RETURN NUMBER(0) SEMICOLON NEWLINE
  RBRACE Assembling the generated assembly file...
  Linking the object file to produce the final binary...
  Running the final binary...
  |-------------------------|
  2
```

Figure 4: Digits Output

calculator.txt

```
int add(int a, int b) {
    return a+b;
}

int sub(int a, int b) {
    return a-b;
}

int mult(int a, int b) {
    return a*b;
}

int div(int a, int b) {
    if (b == 0) {
        // handle undefined
        return 0;
    } else {
        return a/b;
    }
}

int main() {

    int a = 4;
    print "a is ";
    print a;
    int b = 2;
    print "b is ";
    print b;

    print "a+b = ";
    print add(a,b);
    print "a-b =";
    print sub(a,b);
    print "a*b =";
    print mult(a,b);
    print "a/b =";
    print div(a,b);

    return 0;
}
```



```
a is
4
b is
2
a+b =
6
a-b =
2
a*b =
8
a/b =
2
```

Figure 5: Calculator Output

A Future Work and TODO

Future work includes extending the language with additional data types, keywords, instructions and implementing further optimizations. Increased testing and thorough code review will also be necessary to ensure robustness and to address any remaining corner cases or errors.

B Tools and Frameworks

The following tools and frameworks were used:

- Bison
- Flex
- GCC
- NASM
- LD
- MDK

References

[1] GNU MDK

[2] Computer Organization and Assembly Language Programming

[3] Donald E. Knuth, *The Art of Computer Programming*, Addison-Wesley.

Teacher's Lab Notes: Materials and guidance provided by *Panagiotis Katsaros* during the course Programming Languages and Compilers. Lab documentation was used as a reference for this project.