



# A compiler for MicroC

Domenico Ferraro  
d.ferraro7@studenti.unipi.it  
559813

March 6, 2023

## Abstract

This report presents the design and implementation of microcc, a compiler for the simplified C programming language MicroC. The compiler supports separate compilation and enables the compilation of source code and LLVM bitcode together. The task of generating the executable is delegated to clang, which invokes the linker correctly. The report focuses on the key implementation choices and provides a comprehensive overview of the compiler's feature set.

## Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Scanner</b>	<b>2</b>
<b>3</b>	<b>Abstract Syntax Tree</b>	<b>3</b>
<b>4</b>	<b>Parser</b>	<b>3</b>
<b>5</b>	<b>Symbol Table</b>	<b>4</b>
<b>6</b>	<b>Semantic Analysis</b>	<b>6</b>
<b>7</b>	<b>Dead code detection</b>	<b>7</b>
<b>8</b>	<b>Code generation</b>	<b>8</b>
<b>9</b>	<b>Linking and executable generation</b>	<b>8</b>
<b>10</b>	<b>Unit testing</b>	<b>9</b>
<b>11</b>	<b>Conclusions</b>	<b>9</b>

# 1 Introduction

This report provides a detailed account of the design and implementation of a compiler for MicroC, a simplified variant of the C programming language. The compiler is built on top of the LLVM infrastructure and supports the compilation of both MicroC source code and LLVM bitcode. Moreover, it can generate object files and build final executables that can be run on a target machine using the clang compiler. The report also covers the implementation of five out of seven planned language extensions, which are listed below:

- pre/post increment/decrement operators, i.e., `++` and `--`, and abbreviation for assignment operators, i.e., `+=`, `-=`, `*=`, `/=` and `%=`.
- do-while loop, variable declaration with initialization and multiple declarations, e.g., `int i = 0, *j = &z; .`
- pointers, arrays and multi-dimensional arrays as in C.
- a new semantic analysis pass to detect dead code.
- separate compilation.

As mentioned earlier, the compiler can produce LLVM bitcode files and executable files, but also object files. In addition, several new features have been implemented, including:

- a richer run-time support library that provides `printchar` and `printbool` functions for printing characters and booleans, respectively.
- the functions can also return pointers, as in C.
- error messages are displayed in color, which improves the user experience and provides information about the location of errors in the source code.
- compared to clang, the semantic analysis pass provides more accurate results, particularly when dealing with symbols declared in separate source files or bitcode.
- the symbol table has been optimized for efficiency using some great properties of the OCaml HashTable module.

# 2 Scanner

The scanner for the MicroC compiler is implemented using the `ocamllex` tool and consists of the scanner interface (`scanner.mli`) and the implementation (`scanner.ml`) located in the `lib` directory. The scanner uses regular expressions to recognize tokens and a hash table to efficiently detect keywords, improving performance. When an alphanumeric string is recognized, the scanner looks it up in the hash table to determine if it is a keyword (such as `"if"`, `"while"`, `"return"`, etc.). If it is, the scanner emits the corresponding token; otherwise, it is recognized as an identifier for a variable or function. The scanner can recognize integers in both decimal and hexadecimal notation and checks whether the value exceeds the range of integers representable by the `int` type to prevent underflow or overflow errors. The scanner also skips white space and recognizes new lines to keep track of the current line in the source file, providing a precise location in case of errors. The boolean constants `"true"` and `"false"` are not stored as keywords in the hash table but rather as special tokens with associated boolean values. Additionally, the scanner recognizes pre/post increment/decrement and abbreviation for assignment operators (e.g., `+=`, `-=`), which are assigned their own specific token for ease of use in later phases of compilation.

The scanner is able to recognize both multiline and single-line comments. To recognize a character, the scanner uses a different rule that accepts only one character followed by a right single-quote. However, if the left single-quote is followed by a backslash, the scanner accepts a character and tries to recognize the special character `\c`, where `c` can be one of the following: `b`, `t`, `\`, `r`, `n`, `'`. If it is not a valid special character, a lexical error is raised.

To ensure the scanner's correctness and robustness, it was tested extensively with various input sources, including the provided sample sources, and a unit test suite was developed.

### 3 Abstract Syntax Tree

The abstract syntax tree provided has been modified slightly as follows:

- added `PreIncr`, `PostIncr`, `PreDecr` and `PostDecr` to represent pre/post decrement/increment.
- variable declaration has a new optional element of type `expr`, to accommodate the optional initialization value. If the variable is declared with initialization, the optional element is present, it isn't otherwise.
- a new type representing the type of `NULL`.
- function declaration has an optional and not always required body. If a source-code is parsed then the semantical analysis will ensure that the body is present. If a LLVM bitcode is parsed, functions declaration are parsed but not their body.

### 4 Parser

The parser for the MicroC compiler is implemented using Menhir. The parser's monolithic API is used with the `--exn-carries-state` flag, allowing the parser to report the state number when a syntax error is detected. This state number is used to select an appropriate syntax error message from the `parserMessages.messages` file, which is generated using Menhir's API. The grammar for the language is defined in the `parser.mly` file, and the `parsing.ml` and `parsing.mli` files define the parse function that is used to run the parser on a given input file.

In the MicroC language, a `program` is a list of top-level declarations, which can include global variables or functions. When the parser encounters a list of global variables declared on the same line, it appends them to the list of top-level declarations in the order they are parsed, rather than creating an inner list. This flattening strategy helps to avoid the creation of unnecessary inner lists. For multiple local variables declared on the same line within a block, the parser applies the same flattening strategy. In this case, this approach is much powerful as a block can have an inner block. In this case, without the flattening strategy, a potentially large number of inner lists nested inside other inner lists could be created, resulting in inefficient memory usage, and in an more difficult to parse abstract syntax tree.

In order to support the `do-while` and `for` loops, the parser converts them into a block with some initial expressions followed by a `while` loop. Specifically, a `do-while` loop is converted into a block of two statement: the first statement is the `do-while` body followed by a `while` statement with the same guard and body of the original `do-while` loop. The following example illustrates how the `do-while` loop is converted:

```

do {                                {
    content;                        content;
} while(guard);                     while(guard) {
                                    content;
                                    }
                                    }

```

Code 1: Conversion of a do-while to a while loop

A **for** loop is transformed into a block of two statements: the initial expression of the **for**, if any, followed by a **while** statement. The body of this **while** loop consists of a block of two statements, namely, the **for**'s body followed by **for**'s increment expression, if any. If the **for** loop has a guard, the resulting **while** loop has the same guard, otherwise, the guard is the constant **true**. The following examples illustrates how the **for** loop is converted:

```

for (init; guard; incr) {          {
    content;                        init;
}                                  while(guard) {
                                    content;
                                    incr;
                                    }
}                                  }

```

Code 2: Conversion of a for to a while loop

To address the dangling-else problem, the **THEN** token is assigned a lower precedence level than the **ELSE** token.

Parsing of LLVM bitcode files, on the other hand, is accomplished using the LLVM API rather than Menhir. The user's LLVM bitcode files are read and converted to their respective LLVM modules using the LLVM API, after which the parser can parse each module to generate an abstract syntax tree containing only top declarations. This AST is not linked with the ASTs generated by parsing the source code files, but the ASTs of all the LLVM bitcode files are linked together to create a single abstract syntax tree. Multiple LLVM modules are linked together to create a single module.

## 5 Symbol Table

The implementation of the symbol table is available in the files `symbol_table.ml` and `symbol_table.mli`. Some modifications have been introduced to the original symbol table interface:

- added `get_current_block` function to retrieve the content of the current block. It is needed by the dead code detection algorithm.
- added `show` function to print the symbol table to the standard output (used for debugging purposes).
- function `add_entry` does not return the added entry.
- function `lookup` returns an optional value.

Multiple approaches exist to implement the symbol table. A considerable effort was put into its design and development because it is used multiple times and in multiple steps of the compilation pipeline. Thus, having a fast and robust implementation of the symbol table is crucial. To ensure its robustness, a unit testing test suite

was implemented. Finding an element in the symbol table and adding a new element are the most used and important features, and they must be implemented efficiently. The design of the developed symbol table allows high performance.

- $O(1)$  time to create a new block.
- $O(1)$  time to lookup.
- $O(1)$  time to add a new entry.
- $O(|block|)$  to delete a block, where  $|block|$  is the number of elements in the block.

The symbol table is designed to be efficient and robust, given its crucial role in multiple steps of the compilation pipeline. It has been implemented using a single Ocaml's HashTable, which has a nice property: it associates each key to an ordered list of values, allowing the same key to have multiple associated values, and making possible to retrieve the previous associated values. The symbol table stores in the hash table the actual value together to its block level. The symbol table also keep tracks of the current block level and the list of identifiers in the current block.

To add a new entry, the key and its associated block level are inserted into the hash table. If the hash table already contains an element with the given key, the new entry may either shadow the old one or raise a `DuplicateEntry` exception if they are in the same block. In this way, the previous value is kept in memory for future use. All of these operations are done in constant time, ensuring the efficiency of the symbol table.

The symbol table keeps a list of lists, where each inner list represents the list of keys of a block successfully added into the hash table. When a new entry is added into the symbol table, its key is also added on top of the first list. When a new block is created, a new empty list is built on top of the other lists, and the current block level is increased by one. This approach ensures that the building of a new block takes constant time.

Compared to other symbol table designs that use one hash table for each block, the use of a single hash table allows for constant time lookups, irrespective of the number of blocks. This approach saves space and reduces lookup time because only one hash table needs to be searched. Overall, the symbol table design and implementation ensure high performance and robustness, as evidenced by the unit testing test suite.

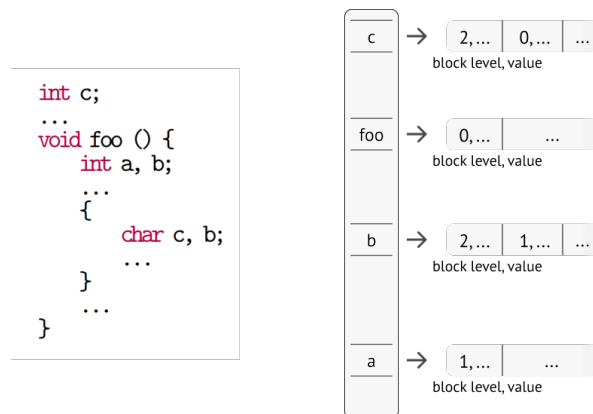


Figure 1: Symbol table example. The outermost scope has level 0. The scope of function `foo` has level 1 and its inner block has level 2. For simplicity, the figure does not show which value is assigned to each key, but it shows the block level

To remove a block from the symbol table, the keys associated with that block must be removed from the hash table. With this design, the list of keys to remove can be obtained in constant time, as it corresponds to the first list stored by the symbol table. Then, every key is removed from the hash table one after the other.

## 6 Semantic Analysis

The semantic analysis is implemented using a symbol table that keeps track of the functions and variables declared in the scope. The corresponding source code can be found in the files `semantic_analysis.ml`, `semantic_analysis.mli`, `sem_error.ml` and `sem_error.mli`.

The analysis has to be performed on the given source codes, but not on the given LLVM bitcodes. However, it has to take into account the external top declarations parsed from the provided LLVM bitcodes, if any. To accomplish this, the algorithm uses two abstract syntax trees: the first is the one on which the semantic rules are checked, while the second contains the external top declarations. The algorithm is straightforward:

1. add the external top declarations in the symbol table.
2. add the run-time support top declarations in the symbol table, without failing if it was already declared in a LLVM bytecode.
3. finally add the top declarations of the given source codes.
4. recursively traverse the source code's abstract syntax tree and perform rules checking.
5. if `main` is required, ensure it is declared properly.

To ensure proper type checking during recursion, an associated type is assigned to each expression. Arrays and pointers are considered equivalent and interchangeable. It is possible to assign an array to another array, and an array is treated as a pointer to its first element. Multidimensional arrays are allowed and, when used as function argument, must follow the same declaration rules as in C: for example `int a[][N][M]` is allowed, but `int a[N][][M]` or `int a[][][]` are not.

Pointer arithmetic is allowed and the mathematical operations are allowed between integers and pointers. Comparison operations are allowed between identical types, between `NULL` and pointers and between `NULL` and arrays. Global variables can only be initialized with compile-time constant values.

Dealing with separate compilation adds complexity to the semantic analysis algorithm. Identifiers may refer to functions or variables declared in other files, making it essential to prevent duplicate declarations. However, when `clang` compiles multiple source code files, its semantic analysis pass does not check if a symbol is defined in another file or report any errors if the symbol is used incorrectly. To illustrate this, consider the example of two files, `main.c` and `foo.c`, where `main.c` calls the function `foo` without any argument and assigns its result to an integer. However, `foo.c` declares `foo` as a void function that takes one integer argument.

The compilation of `main.c` and `foo.c` must fail because the function `foo` is called improperly in the main function: the result of calling `foo` is void but is assigned to an integer and despite the function `foo` requires one integer as an argument, it is called without passing any argument. Unfortunately, when compiled with `clang`, the compilation of `main.c` and `foo.c` succeeds, despite the improper usage of function `foo`.

On the other hand, the MicroC compiler detects the error and fails the compilation, reporting a bad usage of the function `foo`, providing the following error:

```
$ microcc.exe main.c foo.c
Syntax error, File main.c, line 2: Function 'foo' expects 1 arguments
but here 0 arguments are passed to it
  int a = foo();
           ^^^^^
```

Figure 2: Compared to Clang, Microcc, instead, complains about bad usage of function `foo` and stops the compilation process

To ensure that the semantic analysis algorithm can detect any instance of an identifier being declared in multiple files and obtain information about any symbol declared at any time, it is necessary for the algorithm to have access to the top declarations of both source files and LLVM bitcode files.

## 7 Dead code detection

The source code for dead code detection can be found in the files `deadcode.ml`, `deadcode.mli`, `warning.ml`, and `warning.mli`. A piece of code is detected as dead code if and only if it is never used in any source code or LLVM bitcode file, or if it is unreachable.

```
void foo() { // never used
    ...
}

void main() {
    int a; // never used
    return;
    // from here everything is unreachable
    if (a == 0) {
        ...
    }
}
```

For example, in the previous code snippet, there are multiple instances of dead code: the function `foo` is declared but never used, variable `a` is defined but never used, and the `if` statement following the return statement in the main function is unreachable. Compiling this code with microcc generates three warnings, highlighted in yellow, each with its location and explanation. Although warnings do not halt the compilation process, they serve as indicators of potential issues in the code. The warnings are displayed in the figure below.

```
$ microcc.exe main.c
Warning at file main.c, line 8: Unreachable line
    if (a == 0) {
    ^
Warning at file main.c, line 6: Variable 'a' declared but never used
    int a;
    ^
Warning at file main.c, line 1: Function 'foo' declared but never used
void foo() {
^
```

Figure 3: Example of warning reporting when dead code is detected

## 8 Code generation

The source code for code generation can be found in the files `codegen.ml` and `codegen.mli`. This step involves converting the semantically checked abstract syntax tree into LLVM IR code. It requires both the abstract syntax tree and an LLVM module in which the LLVM IR will be generated. Only the code specified in the source code files is generated, so the abstract syntax tree does not contain anything related to the provided LLVM bitcode files, if any. If the user has provided any LLVM bitcode files, they are parsed by the parser, which generates the abstract syntax tree and the LLVM module. All the parsed LLVM modules are then linked together into a single module, which is provided to the code generation algorithm. If the user has only provided source code files, an empty LLVM module is passed to the code generation algorithm.

The code generation algorithm uses a symbol table to keep track of declared symbols and their corresponding LLVM values. The symbol table is first populated with the functions and global variables of the run-time support library and the LLVM module. Note that the run-time support library may already be declared in the LLVM module, so any errors of this kind are ignored at this stage. Next, the algorithm declares all the functions and global variables and then emits LLVM IR for the functions' bodies. The algorithm recursively traverses the abstract syntax tree, but anything after a return statement is not emitted, as required by LLVM to ensure a valid LLVM module.

NULL is a special case since it can be used interchangeably with integer pointers, character pointers, or boolean pointers. When the semantic analysis algorithm has enough information about the types involved, the generic NULL pointer is converted to the appropriate pointer type. The first time it is encountered, it is treated as an undefined LLVM value of a pointer to a void. Later, as needed, it is converted to the proper LLVM value and type:

- in case of a `return NULL;` it is converted to constant null (zero) pointer of the same type of the function return type (which is a pointer to an integer, boolean or character).
- in case of an assignment, for example `int *a = NULL;`, it is converted to the constant null (zero) pointer of an integer, because `a` is an integer.
- in case of a binary operation, for example `a == NULL`, it is converted to the constant null (zero) pointer of the same type of `a`.

Global variables can only be initialized with compile-time constant values. If a global variable does not have any initialization value, a default value is assigned. For example, an array of size `S` and type `T` is assigned an array of `S` default values of type `T`.

If a void function does not have a return statement, it is still considered a valid function. However, LLVM always requires a block terminator at the end of a function. Therefore, after emitting the LLVM IR code of a function body, the code generation algorithm ensures that the function has a block terminator, which may involve adding a void return statement if necessary.

## 9 Linking and executable generation

The linking phase in microcc is handled by the code in `linker.ml` and `linker.mli`. If multiple source files are provided, they are parsed one at a time, producing a number of abstract syntax trees that are then merged together. Similarly, when multiple LLVM bitcode files are provided, each file is parsed individually, resulting in an abstract

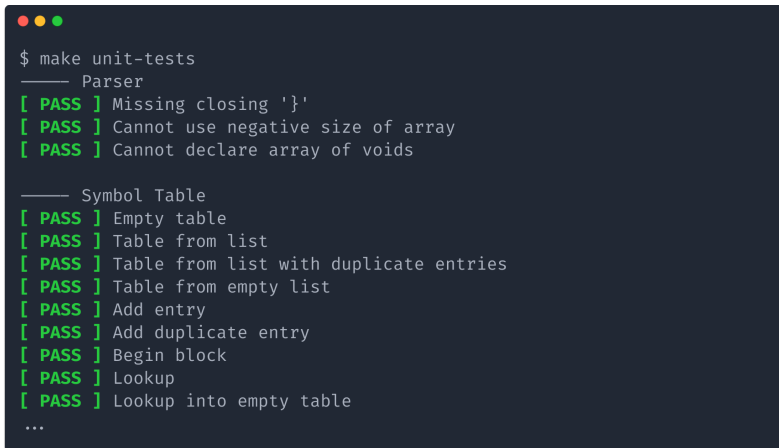


syntax tree and a LLVM module for each file. These LLVM modules are then linked together and given to the code generation algorithm.

Although the code to call the linker LD is still present in microcc, it is commented out because the use of `clang` simplifies the process of linking executables. However, it may be interesting in the future to explore and gain more knowledge about the world of linking executables, with the goal of making microcc independent from `clang`. It should be noted that `clang` was chosen for simplicity and any other compiler, such as `gcc`, could be used in its place. However, in the current version of microcc, it is not possible to specify which compiler to use for the linking phase.

## 10 Unit testing

To ensure the highest level of robustness, a comprehensive unit testing suite consisting of a total of 93 tests was developed. Additionally, various test samples were added to the `test` folder. The compiler was built following the Test Driven Development approach wherever possible. The entire unit testing suite can be executed by running the command `make unit-tests`. Below is a sample output from running the unit testing suite:

A terminal window with a dark background and light-colored text. The output shows the command '\$ make unit-tests' followed by a section header 'Parser' and three test results, all marked '[ PASS ]'. The tests are: 'Missing closing ''', 'Cannot use negative size of array', and 'Cannot declare array of voids'. This is followed by another section header 'Symbol Table' and eight more test results, all marked '[ PASS ]'. The tests are: 'Empty table', 'Table from list', 'Table from list with duplicate entries', 'Table from empty list', 'Add entry', 'Add duplicate entry', 'Begin block', and 'Lookup'. The last line visible is 'Lookup into empty table', followed by an ellipsis '...'.

```
$ make unit-tests
----- Parser
[ PASS ] Missing closing ''
[ PASS ] Cannot use negative size of array
[ PASS ] Cannot declare array of voids

----- Symbol Table
[ PASS ] Empty table
[ PASS ] Table from list
[ PASS ] Table from list with duplicate entries
[ PASS ] Table from empty list
[ PASS ] Add entry
[ PASS ] Add duplicate entry
[ PASS ] Begin block
[ PASS ] Lookup
[ PASS ] Lookup into empty table
...
```

Figure 4: Example run of the unit testing test suite

It is also possible to run parsing, semantic analysis and code generation tests on the whole set of sample source files. It can be done by running the command `make test-parser`, `make test-semantic` and `make test-codegen`, respectively.

## 11 Conclusions

Based on the previous paragraphs, it is clear that a lot of effort has been put into ensuring the quality of the software development process. The use of best practices such as Test Driven Development and Unit Testing, has resulted in a more reliable and robust compiler.

Furthermore, the compilation pipeline has been well-defined and optimized, allowing for efficient generation of executable code. The use of LLVM as the underlying technology has proven to be an excellent choice, providing a powerful and flexible toolchain for code generation and optimization.

Let's examine an example of the compilation pipeline by detailing each step taken by the compiler. For this example, let's assume we're compiling a collection of source code files and LLVM bitcode files, and our aim is to generate the final executable. Here are the steps the compiler performs:

1. Parse each source code file and create an abstract syntax tree (AST). At the end, link all of the ASTs to generate one "Sources AST".
2. For each LLVM bitcode file, read it and create an LLVM module. Then, parse the LLVM module to create an AST. At the end, link all the ASTs to produce one "Externals AST". Finally, link all the LLVM modules to create a single LLVM module.
3. Perform semantic analysis on "Sources AST", but consider the run-time support library and the content of "Externals AST".
4. Run dead code detection and print all warnings, again considering the run-time support library and the content of "Externals AST".
5. Execute code generation to translate "Sources AST" into LLVM IR code in the already built LLVM module.
6. Verify the LLVM module with the LLVM API, and perform optimizations if specified by the user.
7. Set the target machine's information in the LLVM module and build a temporary object file.
8. Use the `fork` system call to delegate the task of calling the linker to clang, which will eventually build the executable. Remove the temporary object file.