# Inside a Compiler

Matthew Roever
Feb 10 · 9 min read ★

An overview of the systems to go from source code to an executable program.

Photo by Scott Webb on Unsplash

At some point every program we use was compiled by a compiler. From desktop apps to embedded software in a microwave. All programming languages, including assembly, are compiled.

A compiler is a program that translates text or other programs into a new program.

The output of a compiler can vary. Compilers can change the source code language (transpilers), produce bytecode for interpretation, or machine code for native execution. At their core, compilers are translators.

## Compiler Structure

The complex systems of a compiler can be grouped into three stages. The front end is responsible for understanding the source code. It will read, validate, and transform its input into a common intermediate representation (IR) that will be used in later steps. The second step is the optimizer. The optimizer uses a series of passes to modify the intermediate representation. With each modification the final behavior of the program will remain the same, but its execution will be improved. The last step is the back end. The back end takes the optimized intermediate representation and converts it to the target output. The output can be a bytecode for use with an interpreter or native executable code for a specific machine architecture.

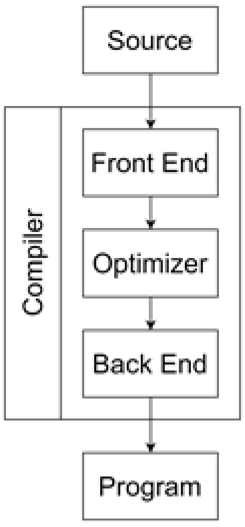


Figure 1: Compiler overview — by author

The distinction between steps enables the creation of versatile compilers. The division of responsibilities allows the GCC and LLVM compilers to support a wide variety of languages. For each language, a unique front end is used to understand and validate the rules of the input language. All front ends process their input to produce a common intermediate representation. This enables the creation of a single optimizer — which is the most complicated part of a compiler. To support many hardware architectures (x86, x64, ARM, MIPS, …) a unique back end is written. Each back end accepts the same intermediate representation, then emits a program for its designated target.

## Front End

The compiler front end is responsible for understanding the language. It verifies the correctness of input code and produces all information necessary for later steps. The front end will produce the intermediate representation and symbol tables.

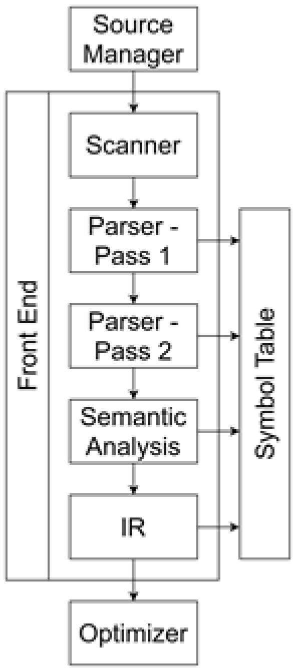


Figure 2: Compiler front end — by author

The front end of a compiler can take multiple forms. The primary components are the scanner, parser, analyzer, and intermediate representation conversion. Throughout these stages, a series of symbol tables are used to store and retrieve data about the code that has been processed. Symbol tables store details about all classes, functions, and global data.

### The Scanner

The scanner — which can also be referred to as a lexer or lexical analyzer — is tasked with dividing the source code into numbers, operators, and words. For example, given the following sentence `Scanners identify words.` , a scanner would produce the following lexemes:

```
Scanners

_        // A single space

identify

_        // A single space

words

.
```

To produce the lexemes, the scanner relies on patterns. A pattern is an acceptable sequence of characters that can be combined to form a lexeme. A pattern can most often be represented using a regular expression or a fixed sequence of characters.

Each lexeme represents a specific pattern in the code. Based on the observed pattern, some lexemes will be combined with a tag. A lexeme-tag pair is called a token. Unlike the lexemes which account for all text in the source file, a token will only be produced for symbols that are relevant to the parser. In the example above, whitespace is a separator. Once the scanner has finished, the whitespace is no longer relevant. The tokens the scanner would produce make no reference to the whitespace:

```
< Scanners : WORD >

< identify : WORD >

< words : WORD >

< . : OPERATOR >
```

## The Parser

The parser is responsible for knowing the fundamental rules of the language. It knows that `2 + 2` is valid, but `2 + × 2` is invalid. The parser also detects classes and functions, which it records in the symbol table. What the parser won't know is whether `x + y` is valid. Determining valid type matching will be the responsibility of the semantic analyzer.

The parser will process the tokens given to it by the scanner to form an abstract syntax tree (AST) for each expression. An abstract syntax tree stores expressions identified by the parser. In an abstract syntax tree, the highest node is called the root. When processing the tree, the nodes furthest from the root must be computed first.

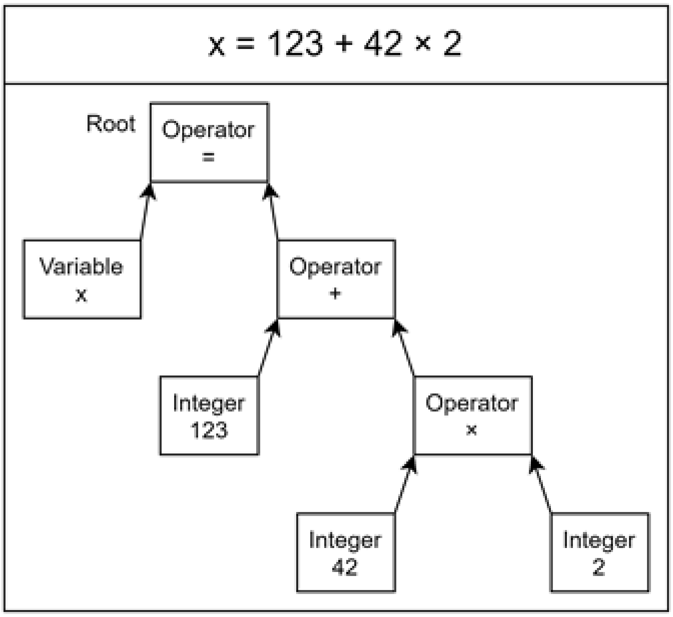


Figure 3: Abstract syntax tree — by author

To solve for x, the integers 42 and 2 will be multiplied first. Then 123 will be added to the result of the multiplication before finally being set equal to x.

When producing the abstract syntax tree, it is the responsibility of the parser to ensure operator precedence. This means all multiplication and division occurs before addition and subtraction, which must likewise occur before assignment. If the parser built nodes from left to right without enforcing operator precedence, x would be set to 123.

In a C compiler, the scanner and parser can be combined into a single step, and there is no need for two pass parsing. This is possible because C requires all symbols to exist before they are used. You can't call a function on line 2 that's declared on line 256. C bypasses this limitation with forward declaration.

By splitting the parser into two stages, there is no need for forward declaration. In the first pass, the parser will look for class and function declarations. Each declaration will be registered in the symbol table for later look up. The first pass will also keep a list of where it found declarations in the token stream for use by the second pass. The second pass will process the function bodies that were skipped by the first. It will produce the abstract syntax trees and verify that all function calls and types correspond to a legitimate symbol. However, it will not try to verify the usage of data types or select the correct overload of a function.

## Semantic Analysis

The semantic analyzer verifies the abstract syntax trees produced by the parser. It will perform any type inferencing, selection of function overloads, and template resolution.

## Intermediate Representation

An intermediate representation is a simplified way of representing a program. The intermediate representation must be a faithful representation. This means it must yield the same result as the source code. A commonly used form of intermediate representation is the static single assignment form (SSA). SSA is produced by converting each set of nodes in an abstract syntax tree into a single operation. This form best represents mathematical expressions such as Figure 3. In SSA form, `x = 123 + 42 × 2` can be written as:

```
temp1 := 42 × 2

x := 123 + temp1
```

This is just one form of intermediate representation. An intermediate representation is any form of abstraction used by the compiler to store your code as its being processed. This includes abstract syntax trees, to the bytecode in Java.

## Optimization

The optimizer improves the efficiency of the program. This can include precomputation of static content such as `2 + 2`. It can also include the elimination of branches in if-else chains when it can be statically proven that a specific case never occurs. There are potentially hundreds of optimizations. For a sense of the range of topics optimizations can cover, see the list of optimization options provided by GCC.

The key with all optimizations is that the output of the program must remain unchanged. Any optimization that improves the final program is valid so long as its faithful to the preoptimized version. Say there is an optimization that could greatly decrease the time it takes for the program to execute. However, there is a 1 in 1,000,000 chance the output will be incorrect. No matter how great the reduction in time, the optimization must not be performed.

Optimizations come in many flavors. Peephole optimizations try to substitute complex operations with simpler alternatives. Loop optimizations seek to improve the efficiency of, well, loops. Data flow analysis tries to reason about the lifetimes of data. And there are many, many more.

## Back End

The back end is responsible for converting the intermediate representation into the targeted output. Depending on the programming language, the back end may produce a bytecode representation of the program or native machine code.

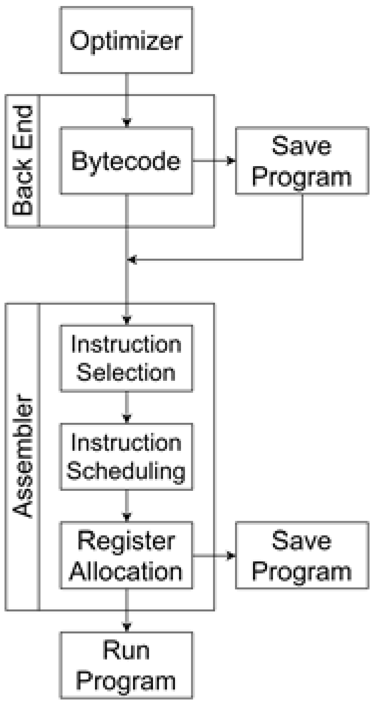


Figure 4: Compiler back end — by author

In a C compiler, the assembler would be the back end. Having the additional bytecode step enables write once, run anywhere. If no bytecode were produced, the program would have to be compiled to each target architecture directly, including assumptions about the capabilities of the machine that will run the code.

My compiler will separate the bytecode generation from the assembler. The compiler will produce bytecode during compilation. The assembler will be included with the runtime library which will be installed on any machine wishing to use a program made with my bytecode. The assembler will finish the compilation process, emitting machine code specialized for the device the program will run on.

## Bytecode

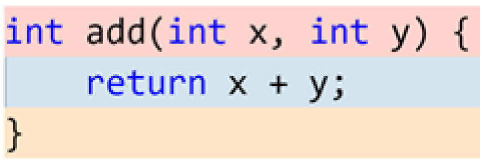
Bytecode — also known as portable code — is a near assembly form of intermediate representation. It is machine independent and can be used by an assembler or an interpreter during execution. The primary motivation to use bytecode is machine independence. The most common example is the Java bytecode.

Machine independence provides compatibility across a wider variety of machines. Intrinsic functions are special compiler supported functions to access CPU instructions that are not native to the programming language. Their use allows the creation of faster programs, but the instructions are not guaranteed to be present on all machines. By using bytecode, intrinsic functions can be used without having to test for their support or needing to produce fall backs. The assembler will detect whether an intrinsic function is supported. It will then use it when available or automatically switch to a fall back. No extra work for you, no overhead checking for support, and guaranteed compatibility on all systems.

## The Assembler

It is the job of the assembler to make your program come to life. The assembler coverts an intermediate representation into machine specific opcodes. Assembly is the lowest level programming language most developers have access to. Computers don't actually run on assembly though. They run on a series of numeric instructions called opcodes. Assembly gives a word to each opcode because thinking in rows of hex is not something many (anyone???) is good at.

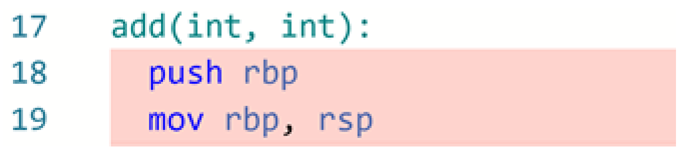
Let's take a look at how all of this comes about. Consider the following function:



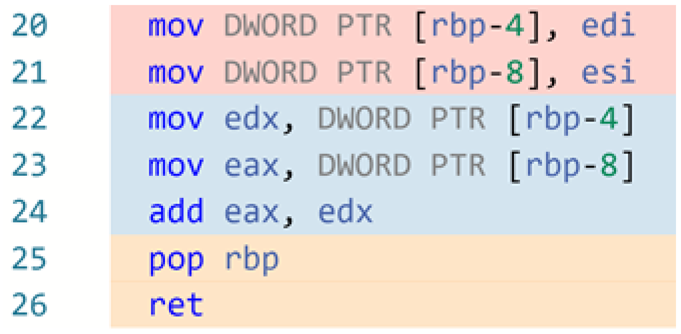
```c
int add(int x, int y) {
    return x + y;
}
```

Figure 5: Simple addition in C — by author using Compiler Explorer

Once converted to assembly, the function looks like:



```
add(int, int):
  push rbp
  mov rbp, rsp
```



```
20    mov DWORD PTR [rbp-4], edi
21    mov DWORD PTR [rbp-8], esi
22    mov edx, DWORD PTR [rbp-4]
23    mov eax, DWORD PTR [rbp-8]
24    add eax, edx
25    pop rbp
26    ret
```

Figure 6: Assembly instructions for add — by author using Compiler Explorer

To get to this point the assembler had to perform three main tasks. First it selected the instructions that it needed to perform. In this case, the only selection is `add` on line 24. While `mov` , `pop` , `push` , and `ret` get the data where it needs to be.

The next task is instruction scheduling. Though not relevant in this example, instruction scheduling can help or harm a program's performance. CPUs measure time in cycles. Each operation takes 1 or more cycles to run. `add` normally takes 1 cycle, `mul` takes 2. Not all are fast, `div` can take 20 cycles and memory loads can take hundreds of cycles. Fortunately, CPUs can issue subsequent operations before the previous operation has finished. If an operation is dependent on the result of a prior operation, it will not be able to execute until the first has finished. Instruction scheduling accounts for the data dependence between operations and the time it takes each operation to execute. It will try to separate dependent operations, maximizing CPU throughput.

The final step is register selection. A register is a specialized piece of memory that can be accessed by the CPU when performing an operation. The registers are represented by `rbp` , `edi` , `esi` , `edx` , and `eax` in the example above. CPUs have a limited number of registers. On an x64 machine, there are only sixteen 64-bit general purpose registers and eight 80-bit floating point registers. This forces the assembler to conserve registers when selecting operations. In reality, register selection occurs during both scheduling and instruction selection.

This article is part of an ongoing series about compilers. The goal is to produce a new programming language.

Why I Am Creating a Programming Language

The need for a fast, safe, and easily understood language.

levelup.gitconnected.com

Divide by Zero Prevention: Traps, Exceptions, and Portability

An examination of multiple solutions.

levelup.gitconnected.com

Series Overview: Creating a Safe Programming Language

Programming Computer Science Compilers Software Development Technology