Faiçal Tchirou Follow
Software Developer @Heetch. Lifelong Learner. INTJ.
Apr 10, 2017 · 7 min read

# Compilers and Interpreters

Sliding doors, Kanō Sanraku

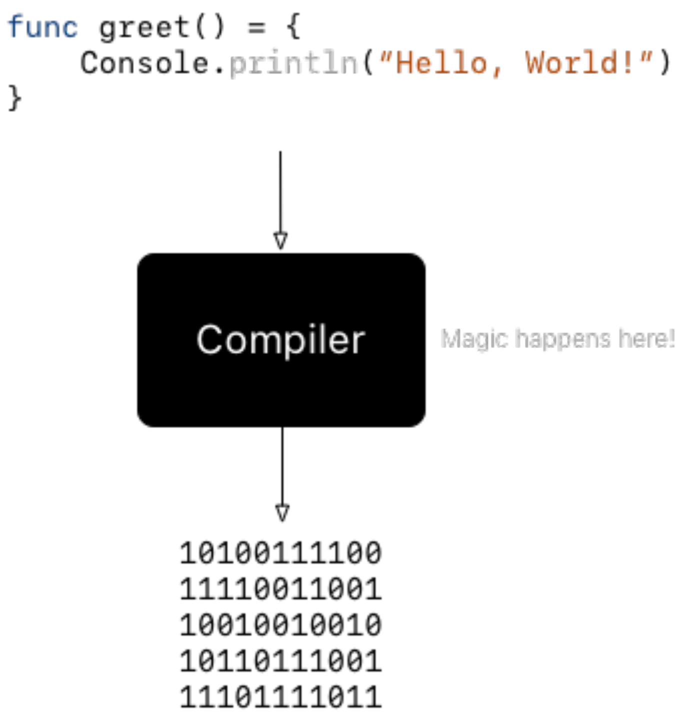
- Welcome to the second article in the *Let's Build a Programming Language (LBPL)* series. If you're not familiar with the series, the purpose of LBPL is to take you from 0 to 1 in implementing a programming language.

This article gives a very high level overview of the structure of compilers and interpreters.

## What is a compiler?

The simplest definition of a compiler is a program that translates code written in a high-level programming language (like JavaScript or Java) into low-level code (like Assembly) directly executable by the computer or another program such as a virtual machine.

For example, the Java compiler converts Java code to Java Bytecode executable by the JVM (Java Virtual Machine). Other examples are V8, the JavaScript engine from Google which converts JavaScript code to machine code or GCC which can convert code written in programming languages like C, C++, Objective-C, Go among others to native machine code.



```
func greet() = {
    Console.println("Hello, World!")
}
```

```
10100111100
11110011001
10010010010
10110111001
11101111011
```

## What's in the black box?

So far we've looked at a compiler as a magic black box which contains some spell to convert high-level code to low-level code. Let's open that box and see what's inside.

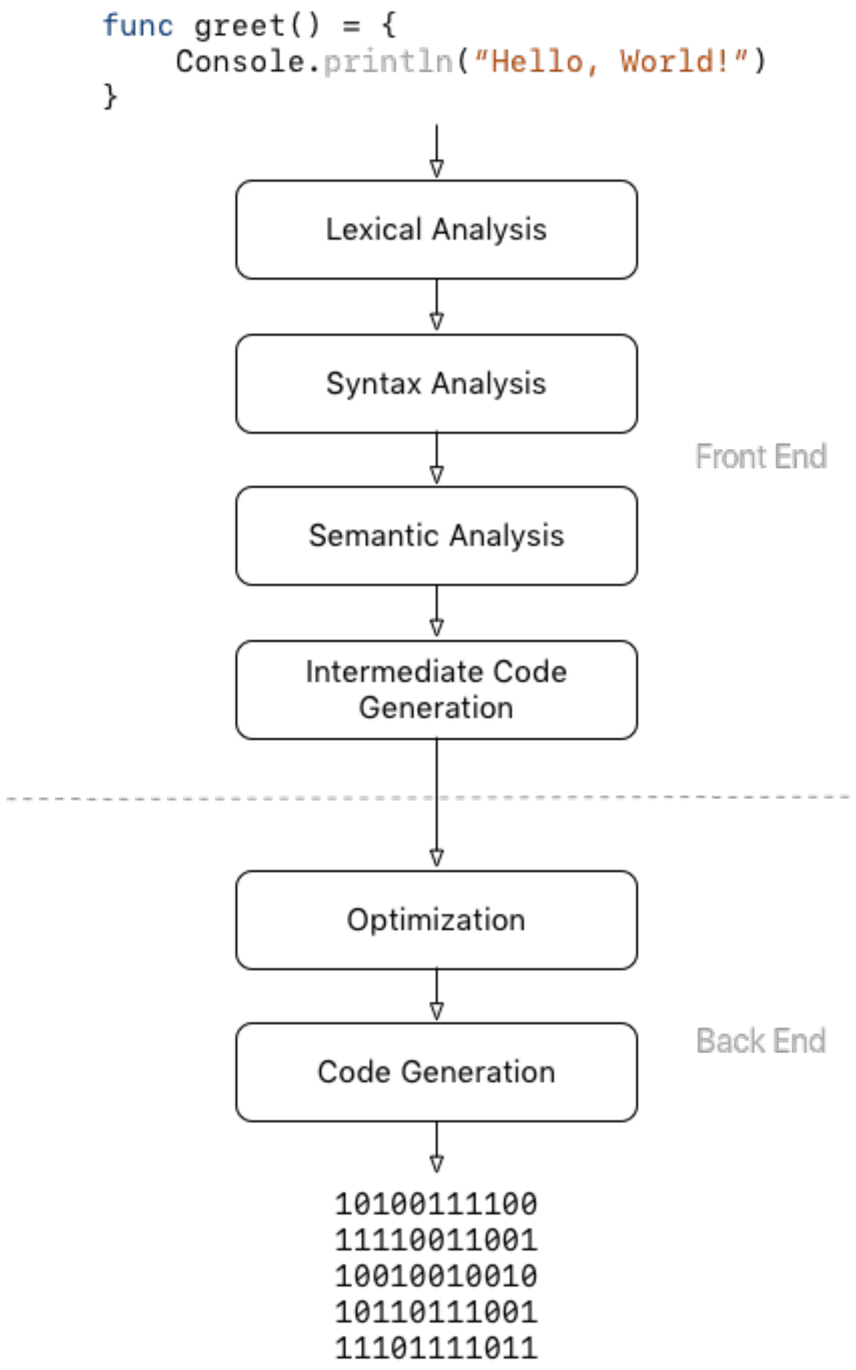
A compiler can be divided into 2 parts.

> The first one generally called **the front end** scans the submitted source code for syntax errors, checks (and infers if necessary) the type of each declared variable and ensures that each variable is declared before use. If there is any error, it provides informative error messages to the user. It also maintains a data structure called **symbol table** which contains information about all the *symbols* found in the source code. Finally, if no error is detected, another data structure, an *intermediate representation* of the code, is built from the source code and passed as input to the second part.

> The second part, the **back end** uses the *intermediate representation* and the *symbol table* built by the *front end* to generate low-level code.

Both the front end and the back end perform their operations in a sequence of phases. Each phase generates a particular data structure from another data structure emitted by the phase before it.

The phases of the front end generally include **lexical analysis**, **syntax analysis**, **semantic analysis** and **intermediate code generation** while the back end includes **optimization** and **code generation**.



```
func greet() = {
    Console.println("Hello, World!")
}
```

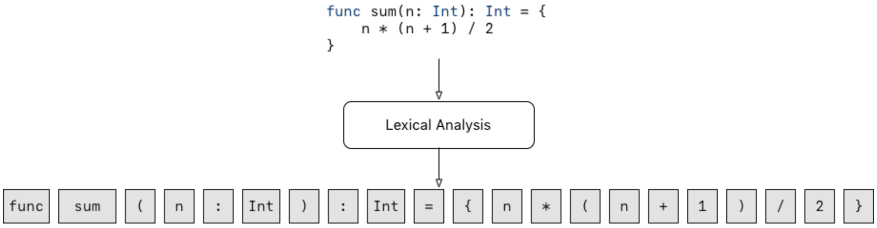
Structure of a compiler

## Lexical Analysis

The first phase of the compiler is the *lexical analysis*. In this phase, the compiler breaks the submitted source code into meaningful elements called **lexemes** and generates a sequence of **tokens** from the lexemes.

A *lexeme* can be thought of as a uniquely identifiable string of characters in the source programming language, for example, *keywords* such as `if` , `while` or `func` , *identifiers*, *strings*, *numbers*, *operators* or *single characters* like `(` , `)` , `.` or `:` .

A *token* is an object describing a *lexeme*. Along with the value of the *lexeme* (the actual string of characters of the lexeme), it contains information such as its type (*is it a keyword? an identifier? an operator?* …) and the position (line and/or column number) in the source code where it appears.



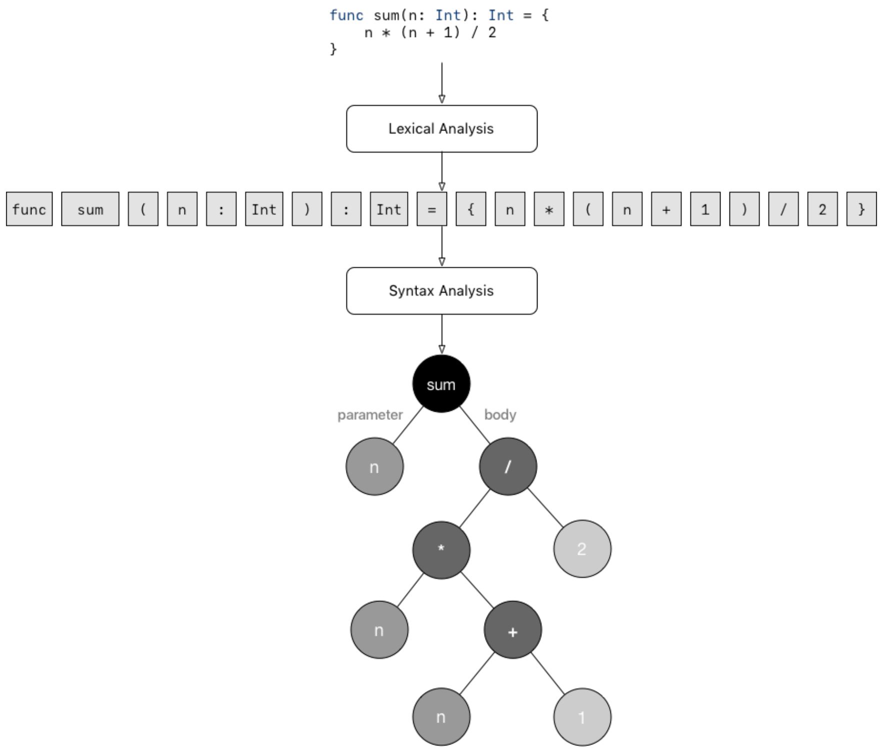
```
func sum(n: Int): Int = {
    n * (n + 1) / 2
}
```

Sequence of lexemes generated during lexical analysis

If the compiler encounters a string of characters for which it cannot create a *token*, it will stop its execution by throwing an error; for example, if it encounters a malformed string or number or an invalid character (such as a non-ASCII character in Java).

## Syntax Analysis

During syntax analysis, the compiler uses the sequence of *tokens* generated during the lexical analysis to generate a tree-like data structure called **Abstract Syntax Tree**, **AST** for short. The *AST* reflects the syntactic and logical structure of the program.



```
func sum(n: Int): Int = {
    n * (n + 1) / 2
}
```

Abstract Syntax Tree generated after syntax analysis

Syntax analysis is also the phase where eventual syntax errors are detected and reported to the user in the form of informative messages. For instance, in the example above, if we forget the closing brace `}` after the definition of the `sum` function, the compiler should return an error stating that there is a missing `}` and the error should point to the line and column where the `}` is missing.

If no error is found during this phase, the compiler moves to the *semantic analysis* phase.

## Semantic Analysis

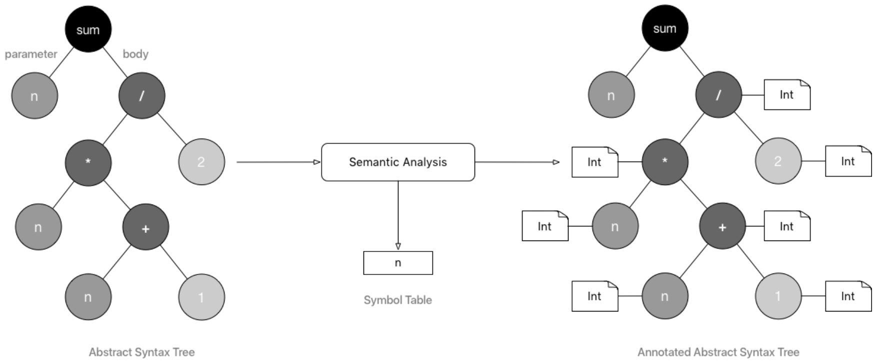
During semantic analysis, the compiler uses the *AST* generated during syntax analysis to check if the program is consistent with all the rules of the source programming language. Semantic analysis encompasses

> **Type inference**. If the programming language supports type inference, the compiler will try to infer the type of all untyped expressions in the program. If a type is successfully inferred, the compiler will **annotate** the corresponding node in the *AST* with the inferred type information.
>
> **Type checking**. Here, the compiler checks that all values being assigned to variables and all arguments involved in an operation

have the correct type. For example, the compiler makes sure that no variable of type `String` is being assigned a `Double` value or that a value of type `Bool` is not passed to a function accepting a parameter of type `Double` or again that we're not trying to divide a `String` by an `Int` , `"Hello" / 2` (unless the language definition allows it).

**Symbol management**. Along with performing type inference and type checking, the compiler maintains a data structure called **symbol table** which contains information about all the symbols (or names) encountered in the program. The compiler uses the *symbol table* to answer questions such as *Is this variable declared before use?*, *Are there 2 variables with the same name in the same scope? What is the type of this variable? Is this variable available in the current scope?* and many more.

The output of the semantic analysis phase is an **annotated AST** and the **symbol table**.

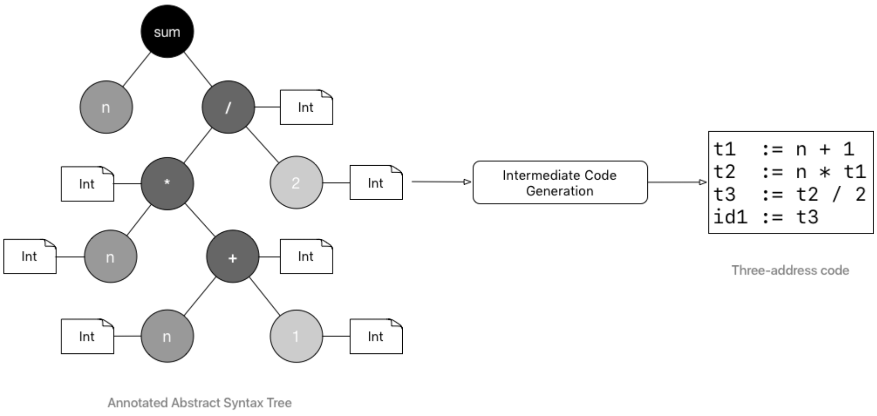


## Intermediate Code Generation

After the semantic analysis phase, the compiler uses the *annotated AST* to generate an intermediate and machine-independent low-level code. One such intermediate representation is the **three-address code**.

The *three-address code* (*3AC*), in its simplest form, is a language in which an instruction is an assignment and has at most 3 operands.

Most instructions in *3AC* are of the form `a := b` **`<operator>`** `c` or `a := b`.



```
t1  := n + 1
t2  := n * t1
t3  := t2 / 2
id1 := t3
```

The above drawing depicts a *3AC* code generated from an *annotated AST* created during the compilation of the function

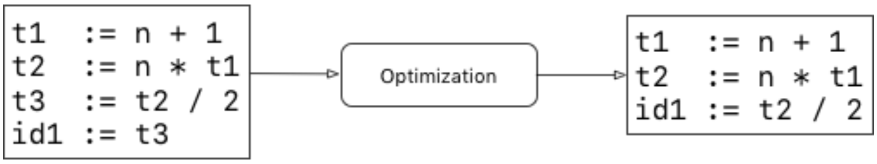
```
func sum(n: Int): Int = {
    n * (n + 1) / 2
}
```

The intermediate code generation concludes the *front end* phase of the compiler.

## Optimization

In the optimization phase, the first phase of the *back end*, the compiler uses different optimization techniques to improve on the intermediate code generated by making the code faster or shorter for example.

For example, a very simple optimization on the *3AC* code in the previous example would be to eliminate the temporary assignment `t3 := t2 / 2` and directly assign to `id1` the value `t2 / 2`.

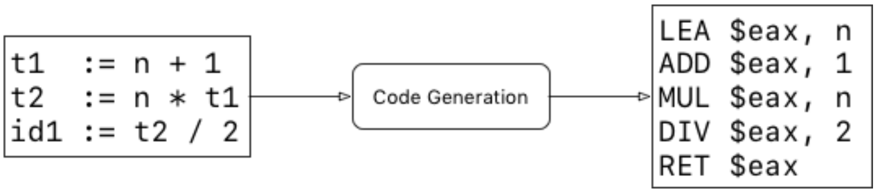


```
t1  := n + 1
t2  := n * t1
t3  := t2 / 2
id1 := t3
```

```
t1  := n + 1
t2  := n * t1
id1 := t2 / 2
```

## Code Generation

In this last phase, the compiler translates the optimized intermediate code into machine-dependent code, *Assembly* or any other target low-level language.



```
t1  := n + 1
t2  := n * t1
id1 := t2 / 2
```

```
LEA $eax, n
ADD $eax, 1
MUL $eax, n
DIV $eax, 2
RET $eax
```

## Compiler vs. Interpreter

Let’s conclude this article with a note about the difference between compilers and interpreters.

Interpreters and compilers are very similar in structure. The main difference is that an interpreter directly executes the instructions in the source programming language while a compiler translates those instructions into efficient machine code.

An interpreter will typically generate an efficient intermediate representation and immediately evaluate it. Depending on the interpreter, the intermediate representation can be an *AST*, an *annotated AST* or a machine-independent low-level representation such as the *three-address code*.

## What’s next?

Head to the next article in which we will look at lexical analysis in depth and review all the concepts necessary to build a lexical analyzer for Blink. You will also get your hands dirty by completing your first challenge.

You’ve reached the end. 🎉

> *Hacker Noon is how hackers start their afternoons. We’re a part of the @AMI family. We are now accepting submissions and happy to discuss advertising & sponsorship opportunities.*

> *If you enjoyed this story, we recommend reading our latest tech stories and trending tech stories. Until next time, don’t take the realities of the world for granted!*