1 - What is a Compiler?

A good question. Let's start from the beginning - humans are lazy. High-level languages are easy for humans to understand; they are akin to a spoken language, such as **English**. But a computer cannot understand a high-level language. You could even say a computer is the opposite of lazy – they require everything to be perfect. They require programs to be written in a low-level language called **machine code**, but the difference in structure is so large, that there needs to be some way of bridging the gap between them. That's where the **compiler** comes in.

Compilers *compile* (i.e., *translate*) a program written in a **high-level** language that is easy for *humans* to understand into a **low-level** language that is easy for *computers* to understand. That way everyone is happy. There are many reasons for coding in a high-level language, but the main reasons are:

- Compared to machine code, high-level languages portray the way humans think about problems *way* better.
- Programs written in a high-level language tend to be *shorter* than the equivalent in machine code.
- The compiler can spot some obvious *mistakes* in the code.

Another significant advantage is that a high-level language is architecture independent. This allows it to be compiled to run on many different machines, contrasting with **machine code** which is architecture dependent.

1.1 - But what about Interpreters?

This article is about compilers, but people often get these two confused, and so I'm going to include this section anyway. They are related, but an **interpreter** executes a program *on-the-fly* without necessarily emitting a translation. Instead, they usually convert the program line-by-line to **bytecode**, which is code that is specifically written to be executed by a **virtual machine**. This approach is quite popular and can be seen in languages such as **Python**.

2 - The stages of Compilation, de… bugged

I hope you appreciate the pun in the title, but anyway, compilers are *complex creatures*. As a result, they are often broken down into distinct chunks. The process of creating an executable from source-code can involve several stages apart from compilation, such as **preprocessing**, and **linking**.

The simplified passes of a typical compiler include:

1. **Lexing**. Here, a lexer groups individual characters from a string of instructions into complete tokens. This is like grouping characters into words in a human language.
2. **Parsing**. Here, individual tokens are grouped into complete expressions. The parser arranges these expressions into a tree-like structure called an **AST (Abstract Syntax Tree)**. This is like grouping words into sentences in a human language.
3. **Semantic Analysis and Type Checking**. Here, the compiler makes sure the code adheres to the language's syntax and makes sense *semantically*.
4. **Code Generation and Optimization**. Here, a code generator converts the **AST** into a concrete program that can be executed by the target device.

Of course, this is an *abstraction*, and so the real deal would have many more additional sub-steps. However, as an introduction these are the main steps that you should know.

2.1 - Lexing

Suppose we wished to compile the following fragment of code:

```
newWidth = width + factor(oldWidth)
```

The **lexer** would read the code character-by-character, identify the *boundaries* that separate the symbols, and emit a series of **tokens**. Each token is a small data structure that describes the *nature and contents* of each symbol, and the original string that comprises the token is called a **lexeme**.

However, there is no *one-to-one* relationship between lexemes and tokens. For example, a **`for`** token consists of one lexeme, but a **number** token may consist of many lexemes. This is further complicated by tokens that overlap, e.g., **`>`** and **`>>`**. Should **`>>`** be interpreted as two greater than tokens, or a right shift token?

`id:newWidth`	`=`	`id:width`	`+`
`id:factor`	`(`	`id:oldWidth`	`)`

Above is what the lexer would split the code fragment into. However, at this stage the purpose of each token is not yet clear. For example, both **`width`** and **`factor`** are seen as **identifiers** by the lexer, even though we know **`factor`** to be the name of a

function. Likewise, we do not know the type of `width`, and so the `+` could potentially represent **addition**, or **concatenation**. This is a context problem for the parser to solve later.

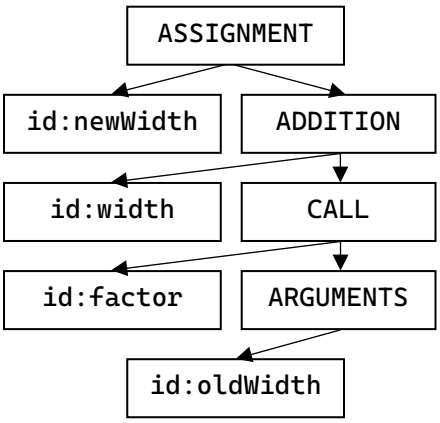
2.2 - Parsing

The next step is to determine whether this sequence of tokens forms a valid program. To do so, the parser looks for patterns that match the grammar of the language. For example, suppose our compiler understands a language with the following grammar:

```
1. expr -> expr = expr
2. expr -> expr + expr
3. expr -> id(expr)
```

Each line is called a rule and explains how the syntax of the language is constructed. For example, **Rules 1 – 2** indicate that an **expression** can be formed by joining two other **expressions** with an *operator*, and **Rule 3** represents a **function call**.

The parser looks for sequences of tokens that can replace the right side of any of the rules in the grammar. Each time a rule is applied, a node is created in the **AST**, and the sub-expressions are connected. The **AST** shows the *relationship* between tokens, for example we now know that `width + factor` is addition, while `factor` is a function call with an argument of `oldWidth`. Here is the AST for the example we saw earlier:



2.3 – Semantic Analysis

Now that the compiler has an **AST**, it's time to make sure that the code *can be compiled*, because as they say there's no point trying to fix what's broken. This step involves several key aspects:

2.3.1 – Type Checking

Type checking makes sure the operations and expressions in the code are *valid* for their type, as defined by the programming language. Let's have a look at the following examples:

Example A: 1 + 1	Example B: 1 + "1"

In **Example A**, the compiler here sees an *addition* with two **integers**, and that's perfectly valid code in any language. However, in **Example B**, the compiler would see an **integer** being added to a **string**. In most languages (except for languages like **JavaScript** which have their own rules), this would result in an *error*. Type checking detects and reports these inconsistencies.

However, it is slightly more complicated than just that, as there are *two types* of type checking that a language may use.

- **Static Typing**. In *statically* typed languages, type checking occurs at **compile-time**. This allows for *early error detection* before the code is executed.
- **Dynamic Typing**. In *dynamically* typed languages, type checking occurs at **run-time**. This leads to potential *runtime* errors if types are found to be incompatible.

2.3.2 – Variable and Function Verification

In this step, the **compiler** ensures that *variables* (and *functions* which are also technically variables) are used correctly in the program. It does this with a few different checks, some of which are:

- **Variable Scope**. The compiler checks whether variables are *declared* before they are used, and whether they *exist* within the correct scope (**local** or **global**).
- **Function Calls**. The compiler checks that functions are being called with the *correct* number of arguments, and appropriate *types*.

2.4 – Code Generation and Optimization

Code generation usually involves translating the validated AST into one of two forms:

- **Machine Code.** We've already come across machine code at the start of this article, but to abstract, machine code is code written in the language of computers. This means it can be understood by computers without requiring any *translation*.
- **Intermediate Representation (IR).** IR is a middle-level representation of the code that is closer to machine code, but independent of the target platform's architecture. IR facilitates *optimization* and allows for easier

translation to specific *machine instructions* than the actual source code.

2.4.1 – Optimization Techniques

Optimization aims to enhance the *performance* of the generated code while preserving its functionality. There are many optimization techniques, but I will only go over *three* for simplicity.

2.4.1 – Local Optimization

This is the **simplest** form of optimization, and in it the compiler optimizes *one basic block* at a time. Let's look at this example:

```
widthSquared = width ** 2
```

Typically, a CPU will not have an **exponentiation** instruction and so this operator would usually be implemented by the software. This would make it slower, but for special cases such as the above code, we can simplify replace it with **multiplication** like so:

```
widthSquared = width * width
```

2.4.2 – Global Optimization

In this stage, the compiler analyses the *relationship* between different parts of the code and performs *optimizations* accordingly. For example, in the following code:

```
for var in range(11):
    print(var * sin(x) / sin(y))
```

The compiler would see that sin(x) / sin(y) is a constant, and so replace it like so:

```
mult = sin(x) / sin(y)
for var in range(11):
    print(var * mult)
```

2.4.2 – Function Inlining

In this stage, function calls are replaced by the *body of the function* itself. This saves a lot of time as a new **stack** does not have to be created for each function. For example, in the below code:

```
def add(a, b): return a + b
x = 1
y = 1
print(add(x, y))
```

The call to the add function can be replaced with the actual content of the add function, like so:

```
def add(a, b): return a + b
x = 1
y = 1
print(x + y)
```

2.4.3 – Code Generation

The final stage involves generation **machine code** specific to the *target platform*. This code may also be further optimized to the **architecture**, **instruction set**, and **memory layout** of the intended platform, ensuring maximum *efficiency* and *compatibility*.

3 – Tips for creating a compiler

Now this article would be quite long if I went over how to exactly to make your own compiler, especially in a language such as C. Due to this I will finish this article of with a few tips, but if you want a look at some full exemplar code then why not have a look at my own compiler (link is at the bottom)?

- **Language design is key**. Make sure your language design is *clear*, *consistent*, and *user-friendly*. Well-defined **syntax** and **semantics** make the compiler's job *easier*.
- **Use modularity and separation of components**. Divide the compiler into *distinct modules* (lexer, parser, etc), as this aids in **debugging** and **maintenance**, and makes it easier to add future modifications.
- **Don't use parser / lexer generators**. While it may seem tempting to use *pre-built* generators for components such as lexers and parsers, it is much better to not to. I cannot stress how much this is, for many reasons. It is *very hard* to integrate blocks of generated code together, you do not know the code as well as if you wrote it which makes it *harder to debug* and doing it yourself allows for a *deeper understanding*.

4 – Thank You!

Whew, I hope that was not too overwhelming for you. Thank you for reading the article, as it really did take me quite a lot of time to write. If you have any more *questions*, then feel free to contact me on **Teams**.

My Compiler (Called The Exeme Language): https://github.com/exeme-project/exeme-lang