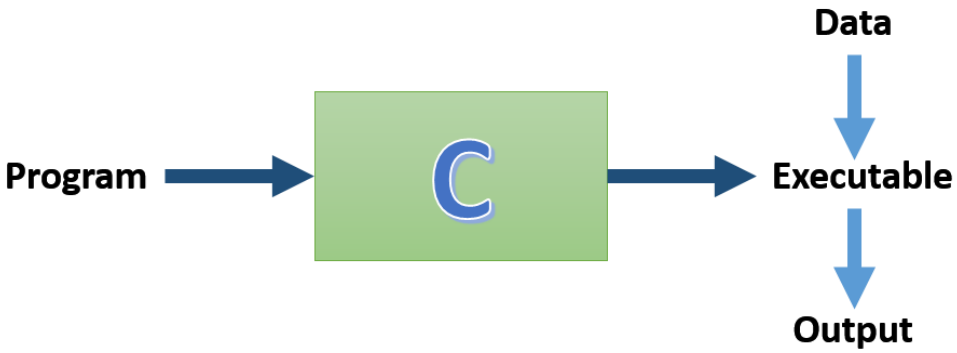
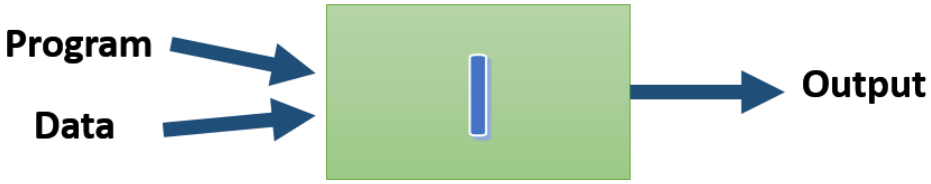
# Introduction to Compilers

There are two major approaches to implementing programming languages, compilers, and interpreters. Now, this Lecture is mostly about compilers. But let us discuss a few word about interpreter.



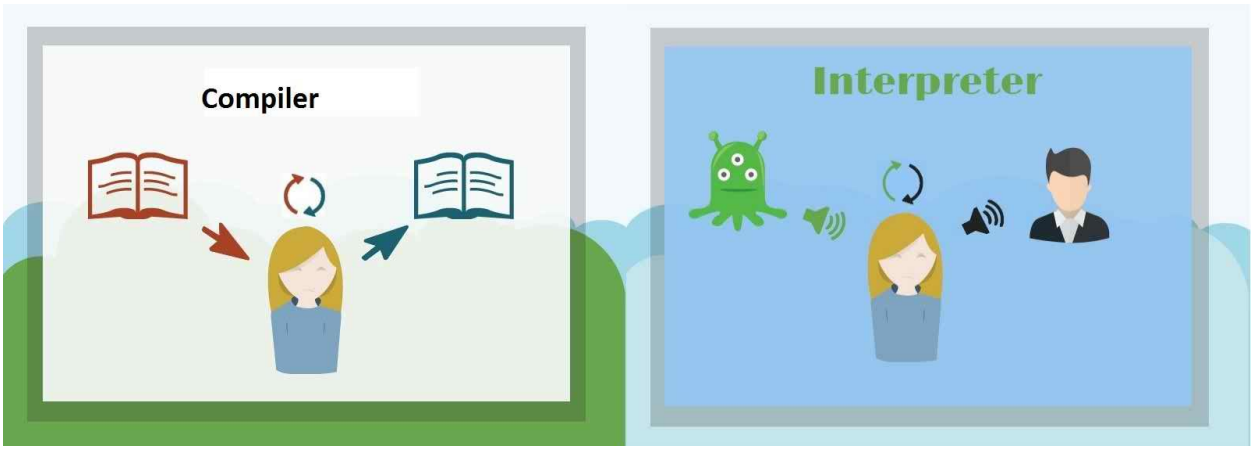

**So, what does an interpreter do?**




An interpreter is also a program that translates a high-level language into a low-level one, but it does it at the moment the program is run. You write the program using a text editor or something similar, and then instruct the interpreter to run the program. It takes the program, one line at a time, and translates each line before running it: It translates the first line and runs it, then translates the second line and runs it etc.

**Interpreter characteristics:**

- Relatively little time is spent analyzing and processing the program
- The resulting code is some sort of intermediate code
- The resulting code is interpreted by another program
- Program execution is relatively slow




**Example of Interpreter:**

FORTAN, Python, JAVA (Java first compilers to an intermediate binary form called JVM byte code. After that the byte code is compiled and/or interpreted to run the program.)

**I think we should know some story about the compiler as well as interpreter.**

So the story begins in the 1950s and in particular with a machine called the 704 built by IBM. This was their first commercially successful machine, although there had been some earlier machines that they had tried out. But anyway the interesting thing about the 704 is, once customers started buying it and using it they found that the software costs exceeded the hardware costs. This cost is not a little bit. It was several multiple of hardware cost. Another hilarious matter is, the hardware cost itself was too high. So you can assume the software cost. It's much higher!

So people start thinking how they can solve this and how can bring something cost effective. After that there comes, **Speed Coding**, developed in 1953 by John Backus. The concept of Speed Coding is now known as Interpreter. Like all interpreters, it had some advantages and disadvantages. The primary advantage was that it was much faster, to develop the programs.

But it was 10-20 times slower than hand written programs and that's also true of interpreted programs today. So if you have an implementation that uses an interpreter, they're going to be much slower than either a compiler or writing code by hand.

Another thing is it was taking around 300 Bytes of memory. You can think this is tiny. For today you are right. But for that time it wasn't. **300 bytes was actually 30% of the whole memory** of 704 that time!

So that Speed Coding wasn't gain that much popularity. But it gave the idea of another project. Thus was the Formula Translation Project or FORTRAN Project had born.

It was run from 1954 to 1957. They thought it would take 1 years. But they were proven wrong. But FORTAN was highly successful. By 1958, 50% of all the codes was written in FORTAN.

FORTAN was raised the level of abstraction, improved programmer productivity, and allowed everyone to make much better use of these machines. So the impact of FORTAN in computer science is huge.

The impact of FORTRAN was not just on computer science research, of course, but also on the development of, practical compilers. And, in fact, its influence was so profound, that today, auto compilers still preserve the outlines of FORTRAN one.

**So what was the structure of FORTRAN one?**

Well, It consist five phases.

1. Lexical Analysis
2. Parsing
3. Semantic Analysis (Types, Scope Rules)
4. Optimization (Faster and less Memory)
5. Code Generation (Into Machine code or another Languages.)

**First step: recognize words. (Lexical Analysis)**

– Smallest unit above letters

**This is a sentence.**

**ist his ase nte nce**

## • Lexical analysis divides program text into “words” or tokens”

**if x == y then z = 1; else z = 2;**

Here if, then, else are keywords.
x, y and z are variables.
1, 2 are number and constant.
== and = are operators
There are semicolon, space.

## • Second Step: Parsing = Diagramming Sentences

- Once words are understood, the next step is to understand sentence structure.
- Diagram are called tree.

**The dog is a faithful animal.**

Let’s see the following one:

**if x == y then z = 1; else z = 2;**

Here, if → Predicate, then and else are statement.
X == y denotes a relation, z = 1 and z = 2 are assigns.

## • Third Step: Semantic Analysis

Once sentence structure is understood, we can try to understand “meaning”
– This is hard!

Compilers perform limited semantic analysis to catch inconsistencies.

Jack said Jerry left his assignment at home.

Jack said Jack left his assignment at home?

- Programming languages define strict rules to avoid such ambiguities.

```
{
    int Jack = 3;
    {
        int Jack = 4;
        cout << Jack;
    }
}
```

- Compilers perform many semantic checks besides variable bindings.

  **Example in English:** Jack left her homework at home.

  • A "type mismatch" between her and Jack; we know they are different people.

  - This is, analogous to type checking in programming or compilers.

## • Fourth Step: Code Optimization

- It's a little bit like editing a line.
  **It's akin to editing a line.**

• Compilers automatically modify programs so that they
  – Run faster
  – Use less memory

X = Y * 0  is the same as X = 0

- NaNs are float and not same as integer.

## • Fifth Step: Code Generation

- Produces assembly code (usually)
- A translation into another language (Analogous to human translation)

**The overall structure of almost every compiler are same as above. We are going to discuss and study more about those five phases.**