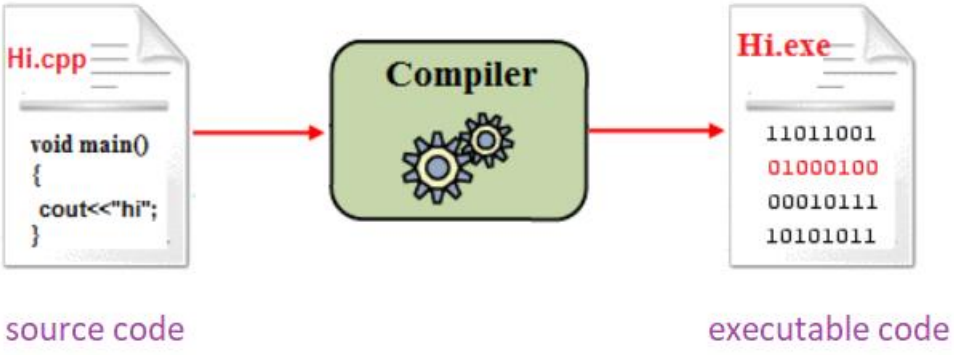
Language Translators

Source code and executable code

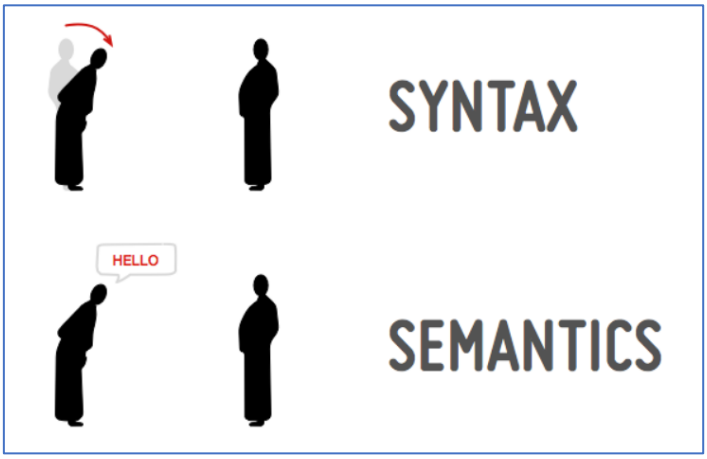
When we write a program in our preferred programming language this program is called **source code**. We then need a **translator** (which itself is another program) to transform our source code in **executable code** i.e. the binary language that the computer understands.



Note how far-removed high-level languages are from the machine code instructions understood by the CPU. This shows how richer the high-level constructs are. There is a **one-to-many** relation between a high-level construct and the equivalent low-level instructions (i.e., one high-level statement is translated in many executable code instructions).

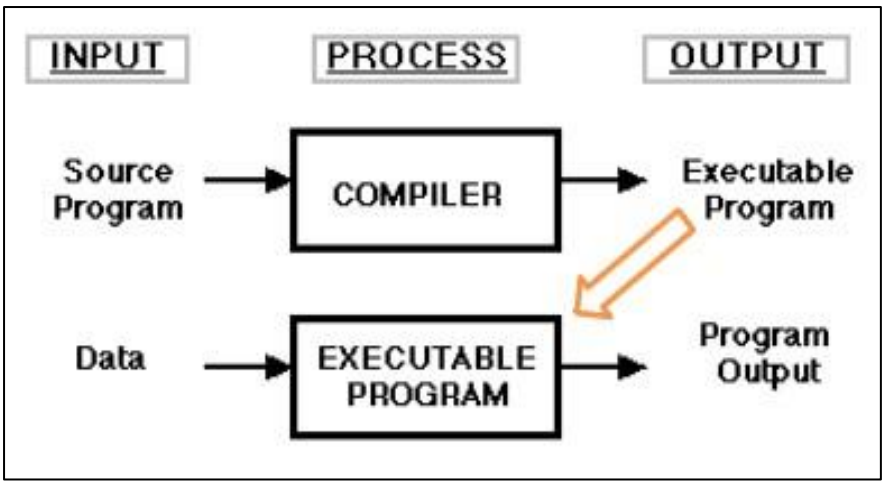
Syntax and semantics

Language translation obviously preserves the **semantics** (semantics refers to the meaning of what is written) of a program – hence a high-level statement may result in many low-level instructions. Syntax and semantics are two terms used in the context of the study of languages. **Syntax** refers to the way we write programs. There are syntax rules which the programmer needs to follow.

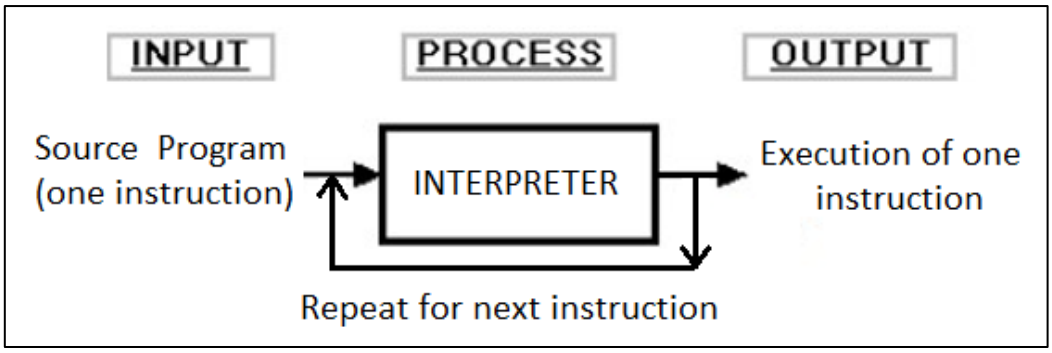


Compilers and interpreters

A **compiler** is a program that translates source code into executable code. The compiler derives its name from the way it works, looking at the entire piece of source code and translating it. Thus, a compiler differs from an **interpreter**, which analyses and executes each line of source code in succession, without looking at the entire program.



Function of a compiler



Function of an interpreter

The advantage of interpreters is that they can execute a program immediately. Compilers require some time before an executable program emerges. However, programs produced by compilers run much faster than the same programs executed by an interpreter.

Because compilers translate source code into object code, which is unique for each type of computer, many compilers are available for the same language. For example, there is a FORTRAN compiler for PCs and another for Apple Macintosh computers. In addition, the compiler industry is quite competitive, so there are many compilers for each language on each type of computer.

Advantages and disadvantages of compilers and interpreters

	Compilers	*Interpreters*
Advantages	• Compiled programs do not need the compiler to be executed • Run faster than interpreted programs • Compilers optimize code. • They give error messages to coder.	• Interpreted programs need the interpreter each time they are executed. • A program can still be partially run even if there are syntax errors (program will run until it encounters a syntax error
Disadvantages	• Does not compile if there is even one single syntax error. • Large programs take quite some time to compile.	• Programs run slower than compiled programs

Assembler

An **assembler** is a program that takes basic computer instructions (in assembly language) and converts them into executable code.

In the earliest computers, programmers wrote programs in machine code, but assembler languages or instruction sets were soon developed to speed up programming. Today, assembler programming is used only where very efficient control over processor operations is needed. It requires knowledge of a particular computer's instruction set. That is why it is called a low-level language.

```
global  _main
extern  _printf
section .text
_main:
push    message
call    _printf
add     esp, 4
ret
message:
db  'Hello, World!', 10, 0
```

Advantage of assembly languages:

- Programmer has more control on the code; therefore a programmer can make faster code.

Disadvantages of assembly languages

- It takes a lot of time and effort to write the code for the same.
- It is very complex and difficult to understand.
- The syntax is difficult to remember.
- It has a lack of portability of program between different computer architectures.

Note that

1) Symbolic addressing is important in assembly language. A **symbolic address** represents an address that during execution takes a physical value e.g., in LDA PAY, ADD BONUS and STO TOTAL the names PAY, BONUS, and TOTAL all represent addresses. During different executions, the physical addresses reserved for these symbolic addresses may (and very probably will) be different.
2) There is a one-to-one correspondence between an assembly language instruction and a machine code instruction (the instructions are arithmetic, logical, fetch and store, branch, and input/output.
3) **Software portability** means the characteristic of being able to use one program on different types of computers. This, in HLL and 4G languages is possible only if the translators for different computer systems exist.