Compiler Design

Lecture 1: Introduction to Compilers

Sahar Selim

Credits for Dr. Sally Saad, Prof. Mostafa Aref, Dr. Islam Hegazy, and Dr. Abd ElAziz for help in content preparation and aggregation

Prerequisites

- Computer Architecture
- Analysis and Design of Algorithms
- Concepts of Programming Languages
- Theory of Computation

A little refreshment for those courses would help you progress well in our course ☺

Course Information

- Lecture
- Office Hours
- Moodle
- Evaluation

Lectures

Sahar Selim (SSelim@nu.edu.eg)

- Lecture (2 hrs/week)
  - Theoretical and Scientific Background
- Lecture
  - Tuesday 8:30 – 10:30
  - Thursday 2:30 – 4:30
- Office Hours
  - Sunday & Tuesday 10:30 – 12:30

Course Grading

Total score 100 degrees

Category	Total
Final exam	25
Midterm	20
Lecture Contribution	5
Quizzes	15 (3)
Assignments	10 (2)
Lab Tasks	5
Project	20 (2)

Percentages are subject to changes depending on circumstances at the time

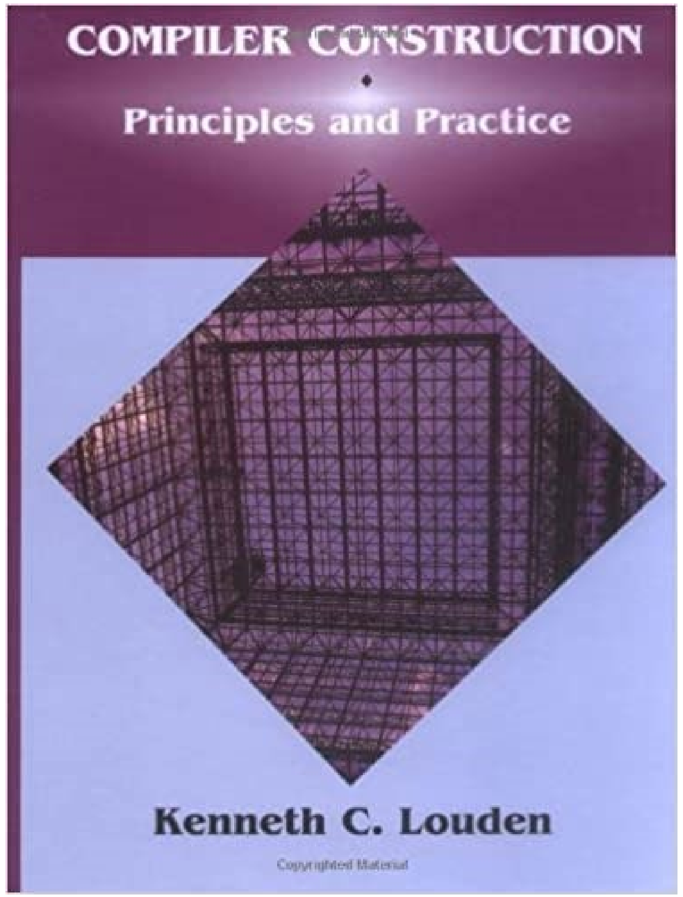
Course Objectives

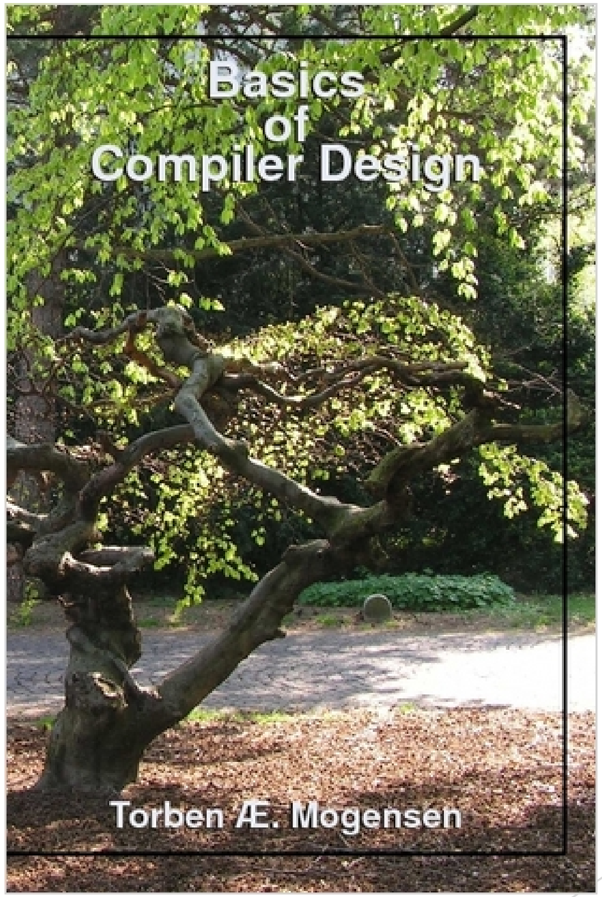
- Develop a fundamental understanding of **the issues that arise in program translation.**
  - including **syntax analysis**, **translation**, and **basics of program optimization**
  - Learn the **Science** behind building up a **Compiler**

Course Learning Objectives

- Study the basic concepts, theories and principles for writing compilers
- Build lexical analyzer, scanner, starting from regular expression
- Get students acquainted with programming language's definition (Syntax and Semantics)
- Identify and describe syntax of programming language by Context-Free Grammars
- Implement Techniques for Efficient Parsing
- Write syntax-directed translation schemes of Language constructs
- Learn optimization methods for better performance, maximum execution, efficiency and Minimum code size

Suggested Books



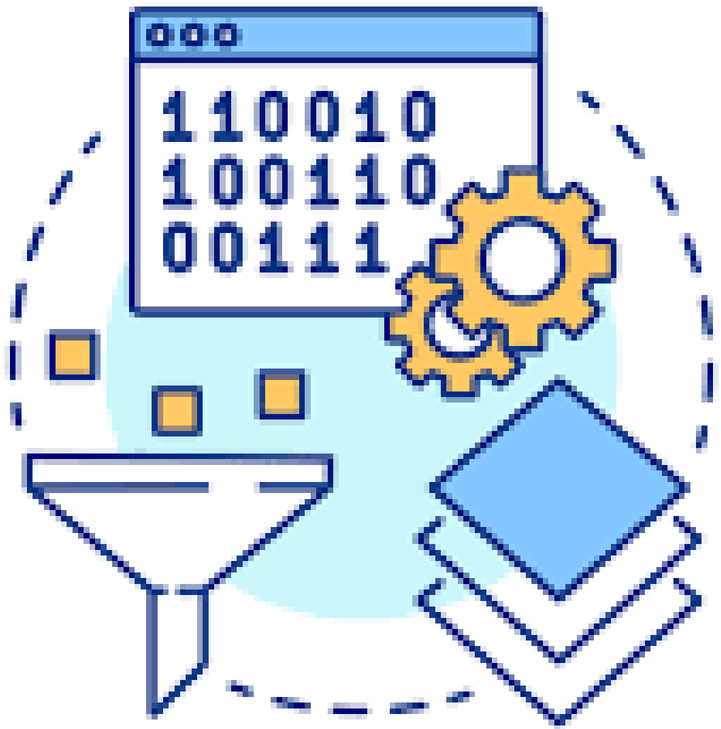


Course Outline

- Lexical analysis
- Syntax analysis
- Top-down parsing
- Bottom-up parsing
- Semantic analysis
- Runtime environment
- Code generation

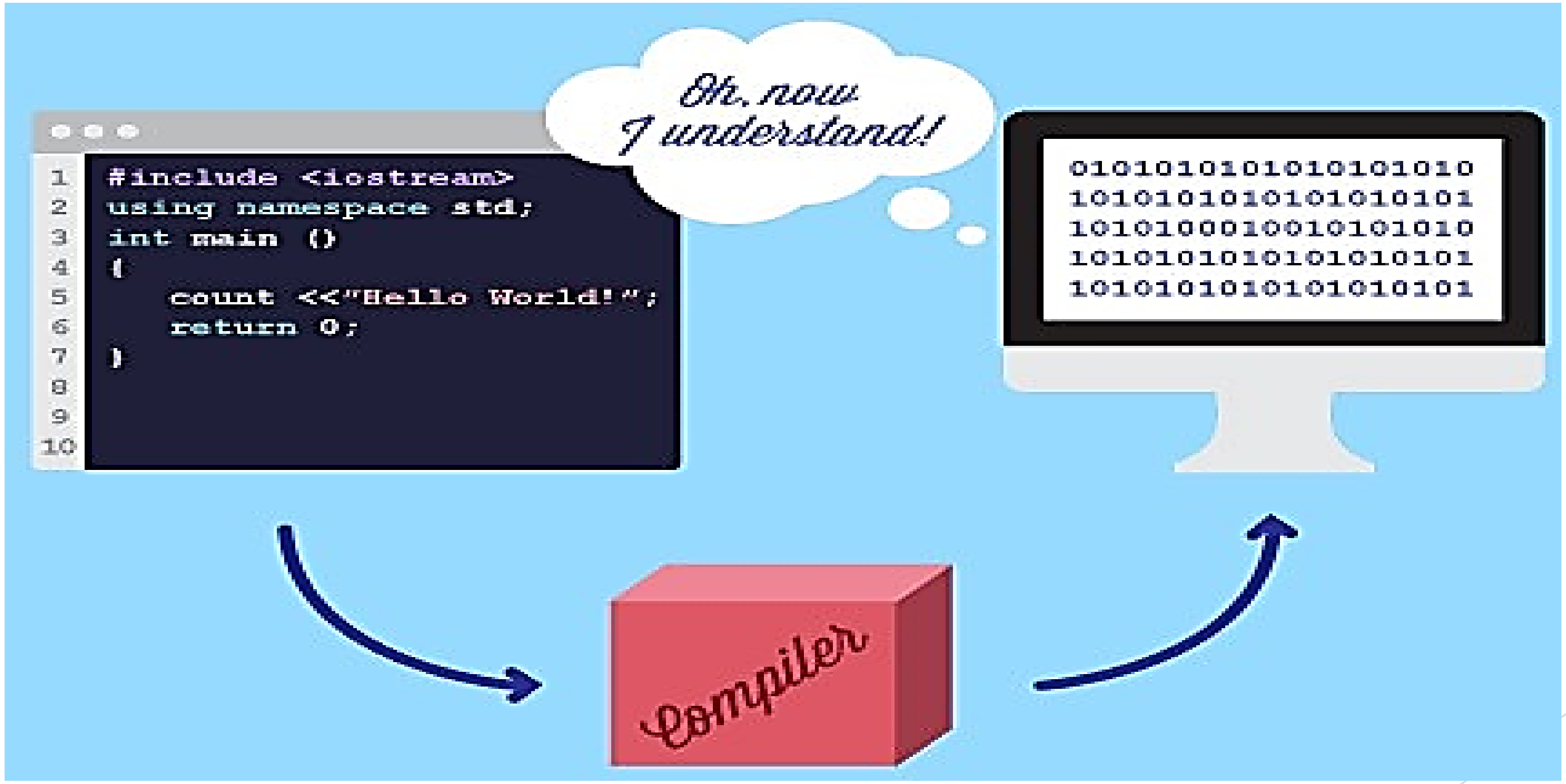
Lecture Agenda

- Introduction to Compilers Theory
- Compiler: A brief History
- Language Processing System
- Programs related to A Compiler
- Compiler versus Interpreter



Introduction to Compilers Theory

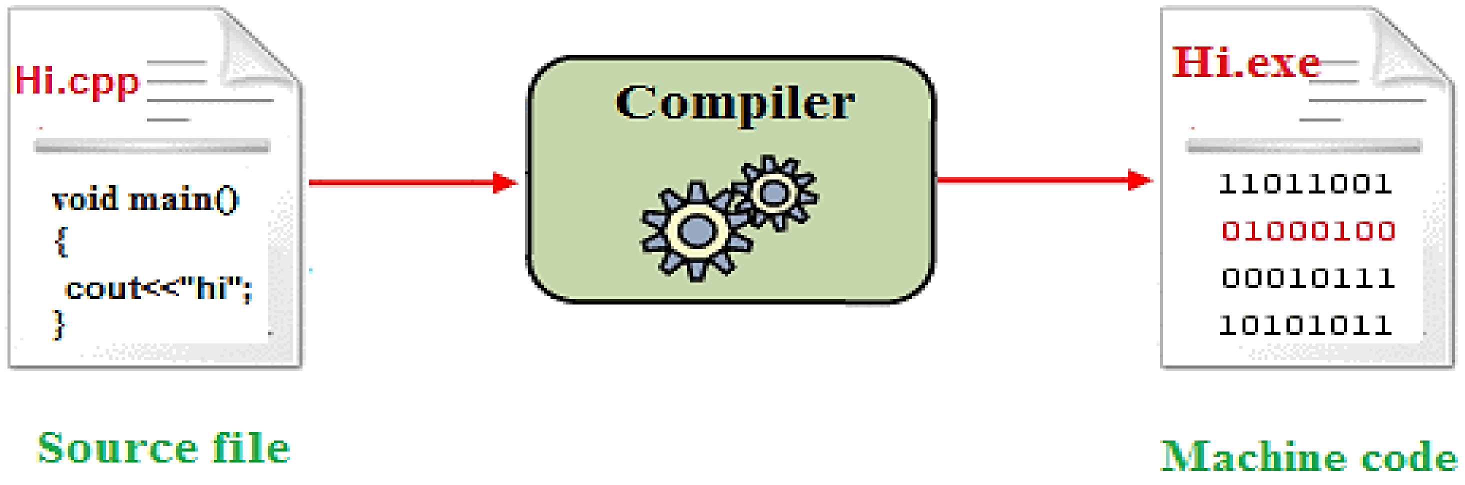
High-level versus low-level languages



```
#include <iostream>
using namespace std;
int main ()
{
    count <<"Hello World!";
    return 0;
}
```

What's a compiler?

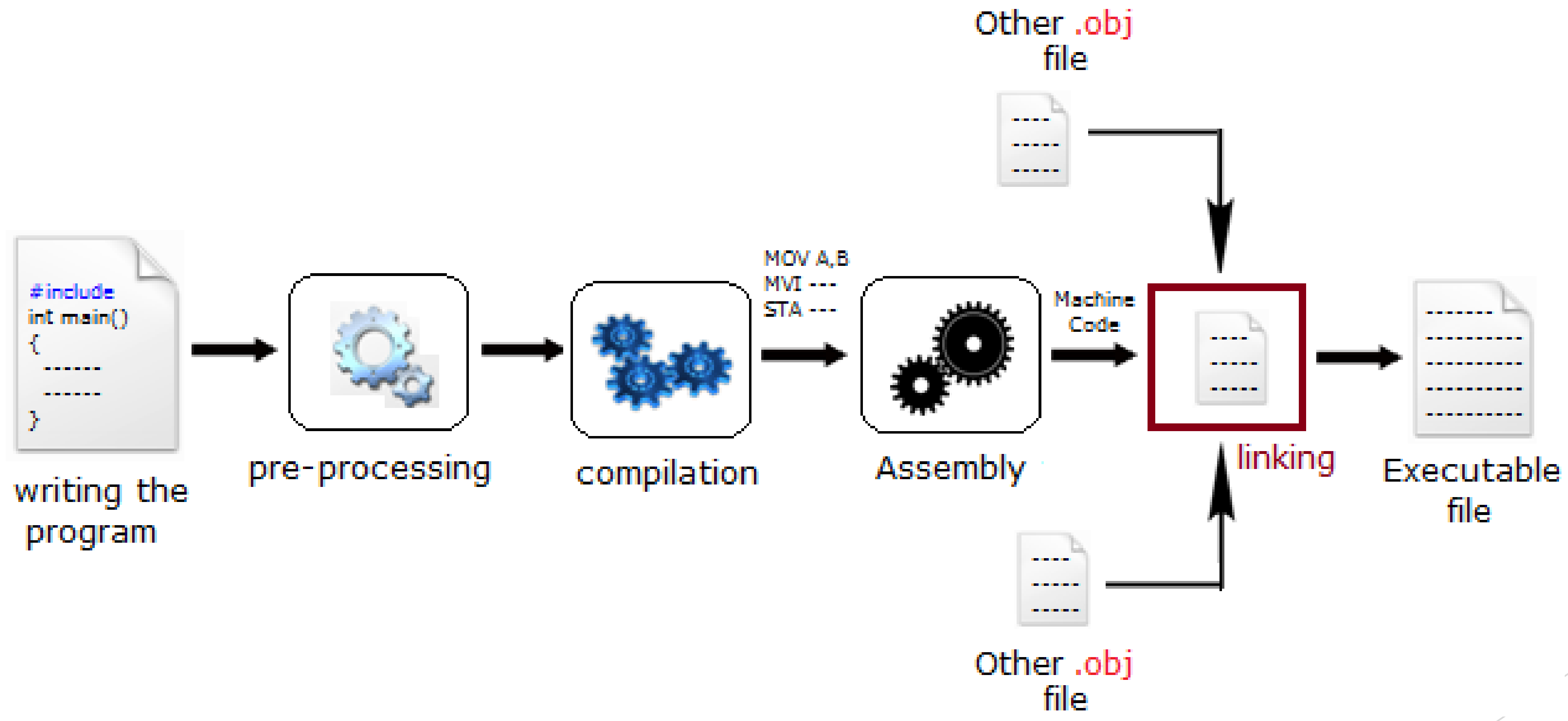
$$\begin{tikzcd}\text{Source Program} \arrow[r] & \boxed{\text{Compiler}} \arrow[r] & \text{Target Program}\end{tikzcd}$$

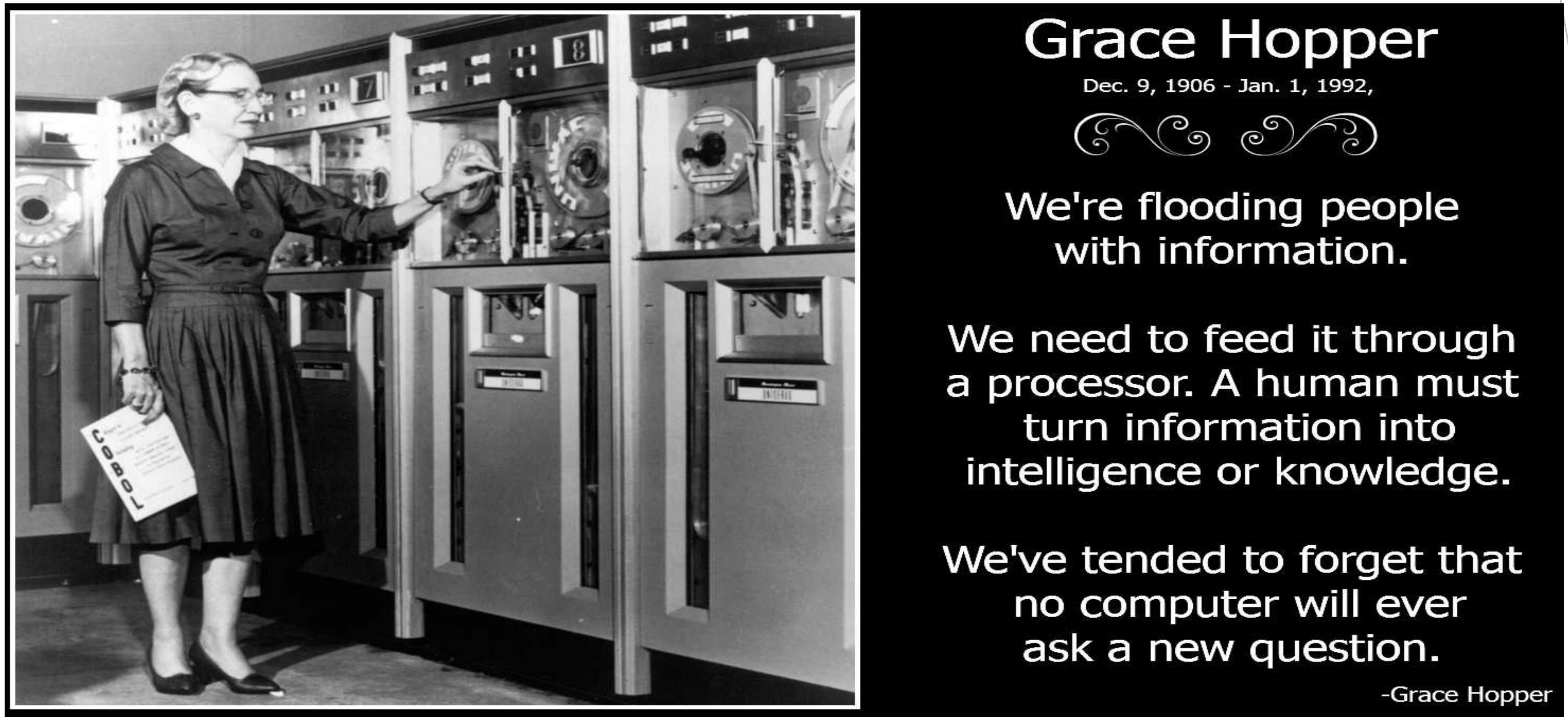


```
void main()
{
 cout<<"hi";
}
```

```
11011001
01000100
00010111
10101011
```

Execution process of C Program





Grace Hopper

Dec. 9, 1906 - Jan. 1, 1992,

We're flooding people with information.

We need to feed it through a processor. A human must turn information into intelligence or knowledge.

We've tended to forget that no computer will ever ask a new question.

-Grace Hopper

A brief History

A brief history

- The first compiler was written by **Grace Hopper**, in 1952, for the A-0 programming language.
- The first Complete compiler was developed between 1954 and 1957.
  - The FORTRAN language and its compiler by a team at IBM led by John Backus.
  - The structure of natural language was studied at about the same time by Noam Chomsky.

Continue . . .

The related theories and algorithms in the 1960s and 1970s

- The classification of language:
  - Chomsky hierarchy
- The parsing problem was pursued:
  - Context-free language, parsing algorithms
- The symbolic methods for expressing the structure of the words of a programming language:
  - Finite automata, Regular expressions

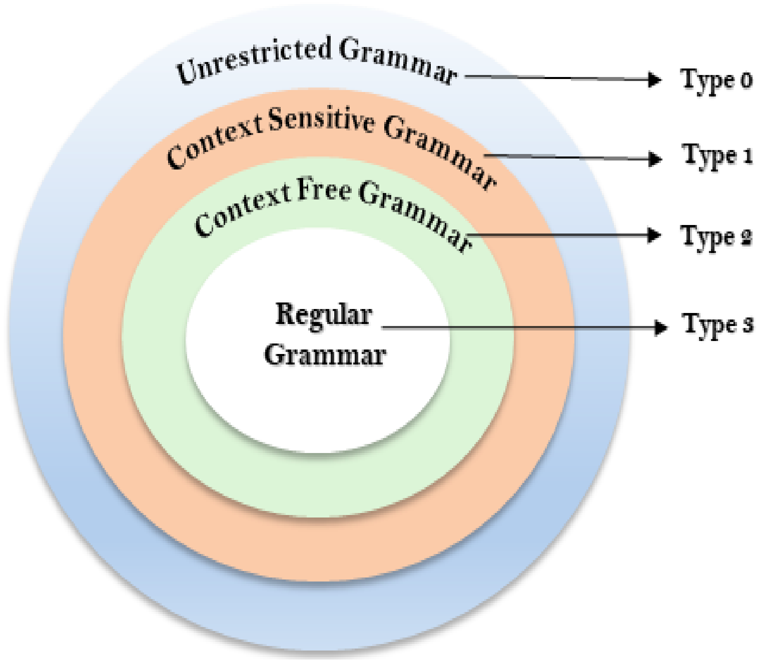


Fig: Chomsky Hierarchy

Continue . . .

Programs were developed to automate the complier development for parsing

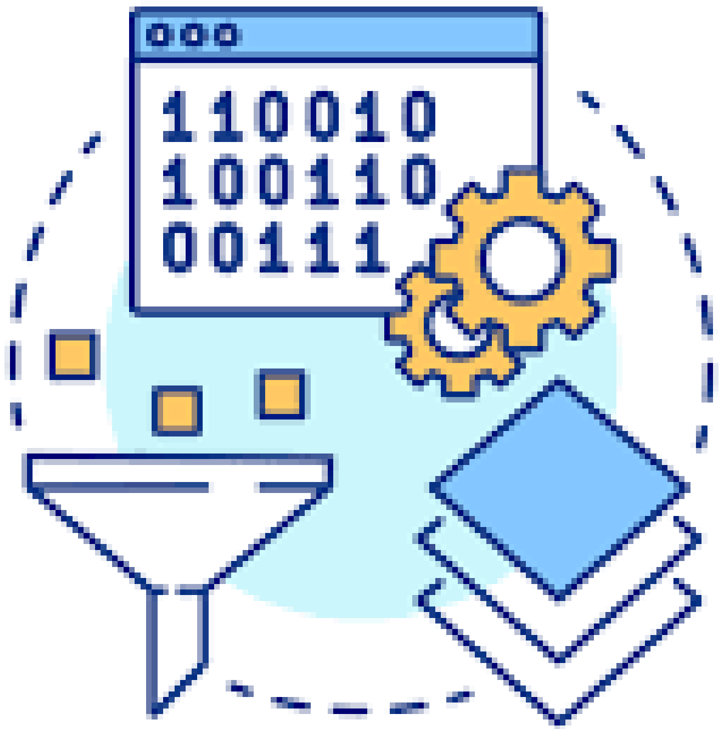
- *Parser generators* such as Yacc by Steve Johnson in 1975 for the Unix system
- *Scanner generators* such as Lex by Mike Lesk for Unix system about same time

- Projects focused on automating the generation of other parts of a compiler.
  - Code generation was undertaken during the late 1970s and early 1980s

Continue . . .

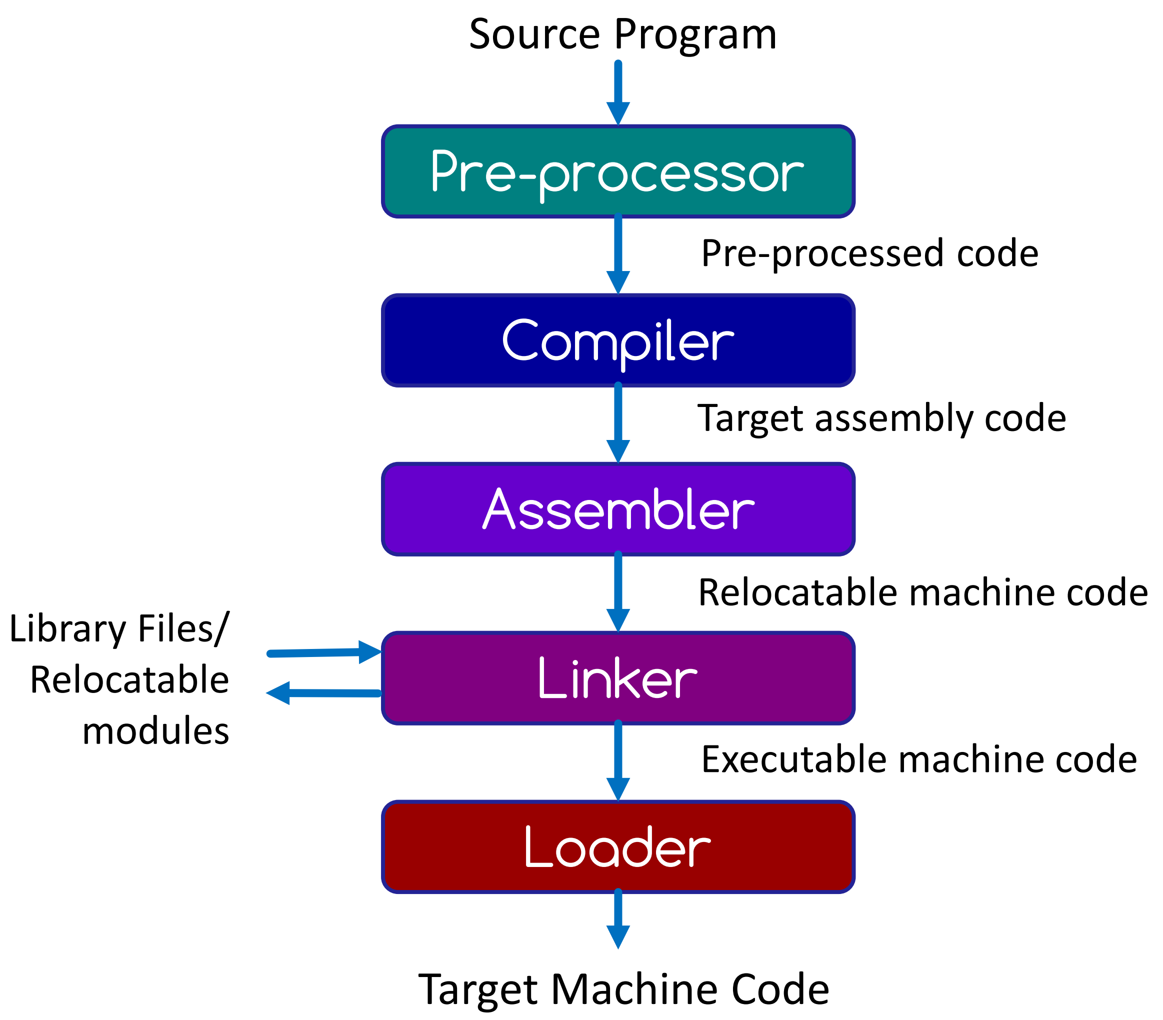
- Recent advances in compiler design
  - More sophisticated algorithms for inferring and/or simplifying the information contained in program.
    - such as the unification algorithm of Hindley-Milner type checking
- Window-based Interactive Development Environment
  - IDE, that includes editors, linkers, debuggers, and project managers.

<<However, the basic of compiler design have not changed much in the last 20 years>>

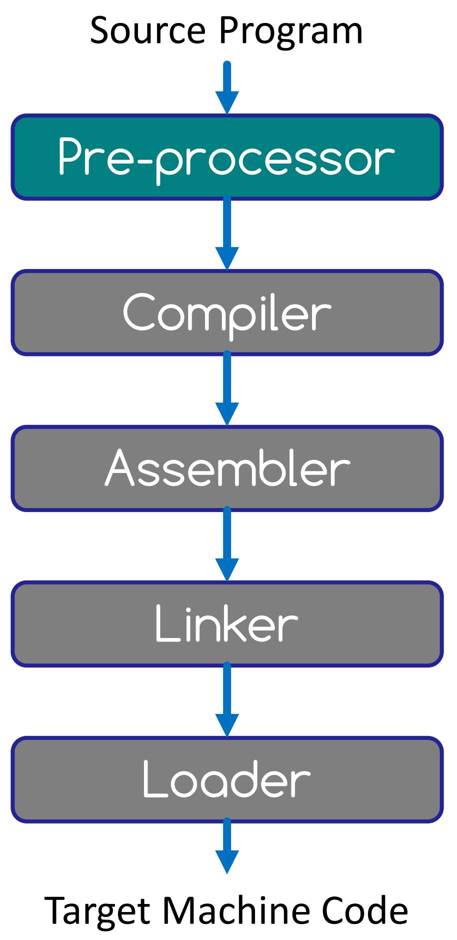


Language Processing System

Language Processing System

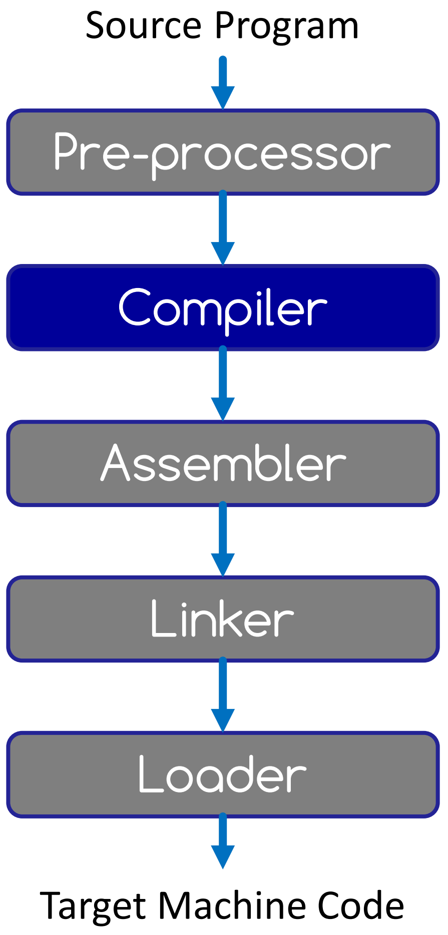


1. Preprocessors



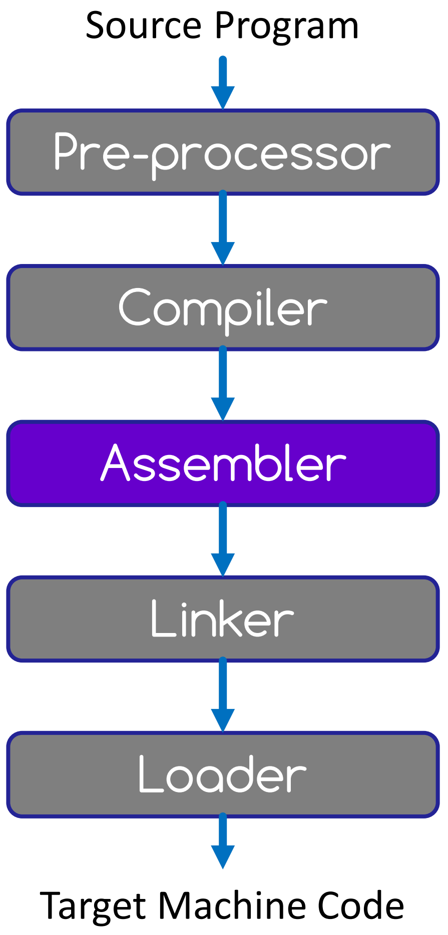
- Delete comments, include other files, and perform macro substitutions.
- Required by a language (as in C) or can be later add-ons that provide additional facilities

2. Compiler



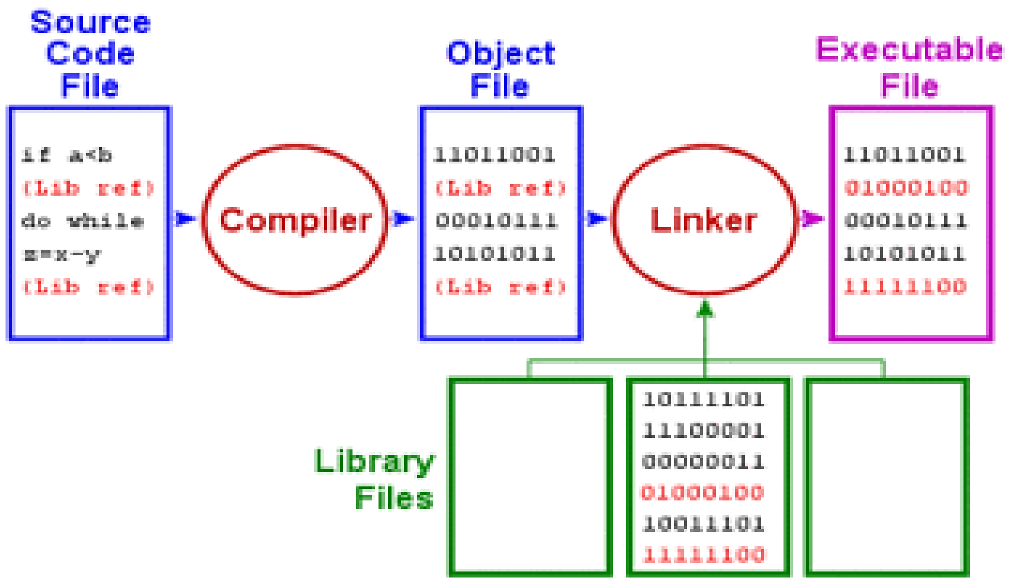
- A **compiler** translates the input pre-processed code and generate **assembly** language as its target language.
- It reveals any bugs or errors.
- It outputs target assembly code.

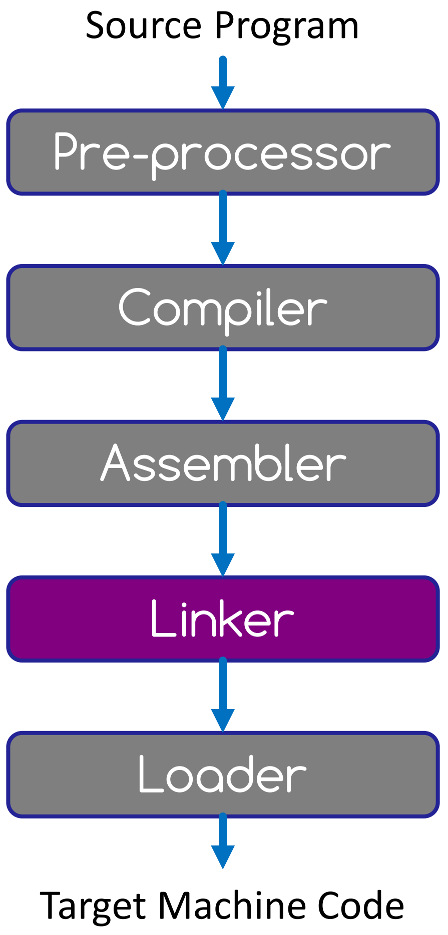
3. Assemblers



- An assembler translates assembly language programs into machine code.
- The output of an assembler is called an object file, which contains a combination of *machine instructions* as well as the *data* required to place these instructions in memory.

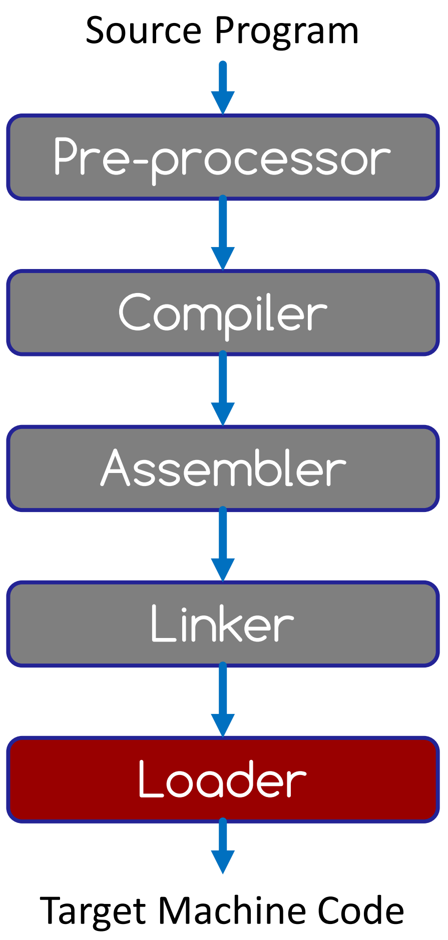
4. Linkers



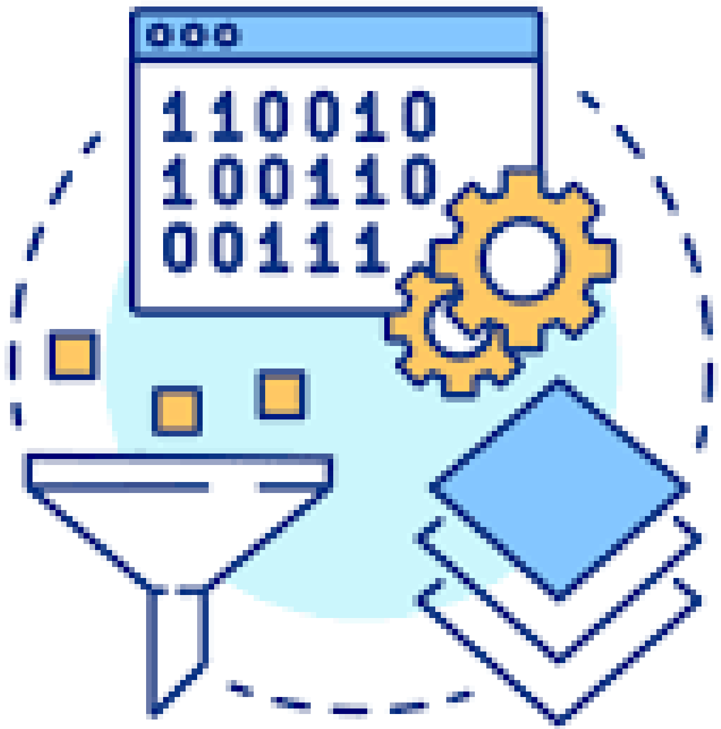


- Linker is a computer program that links and merges various object files together in order to make an executable file.
- It searches and locates referenced module/routines in a program and determines the memory location where these codes will be loaded, making the program instruction to have absolute references.

5. Loaders



- A part of an operating system that is responsible for loading programs and libraries.
- It calculates the size of a program (instructions and data) and creates memory space for it.
- It initializes various registers to initiate execution.



Programs related to a Compiler

Other Programs: Editors

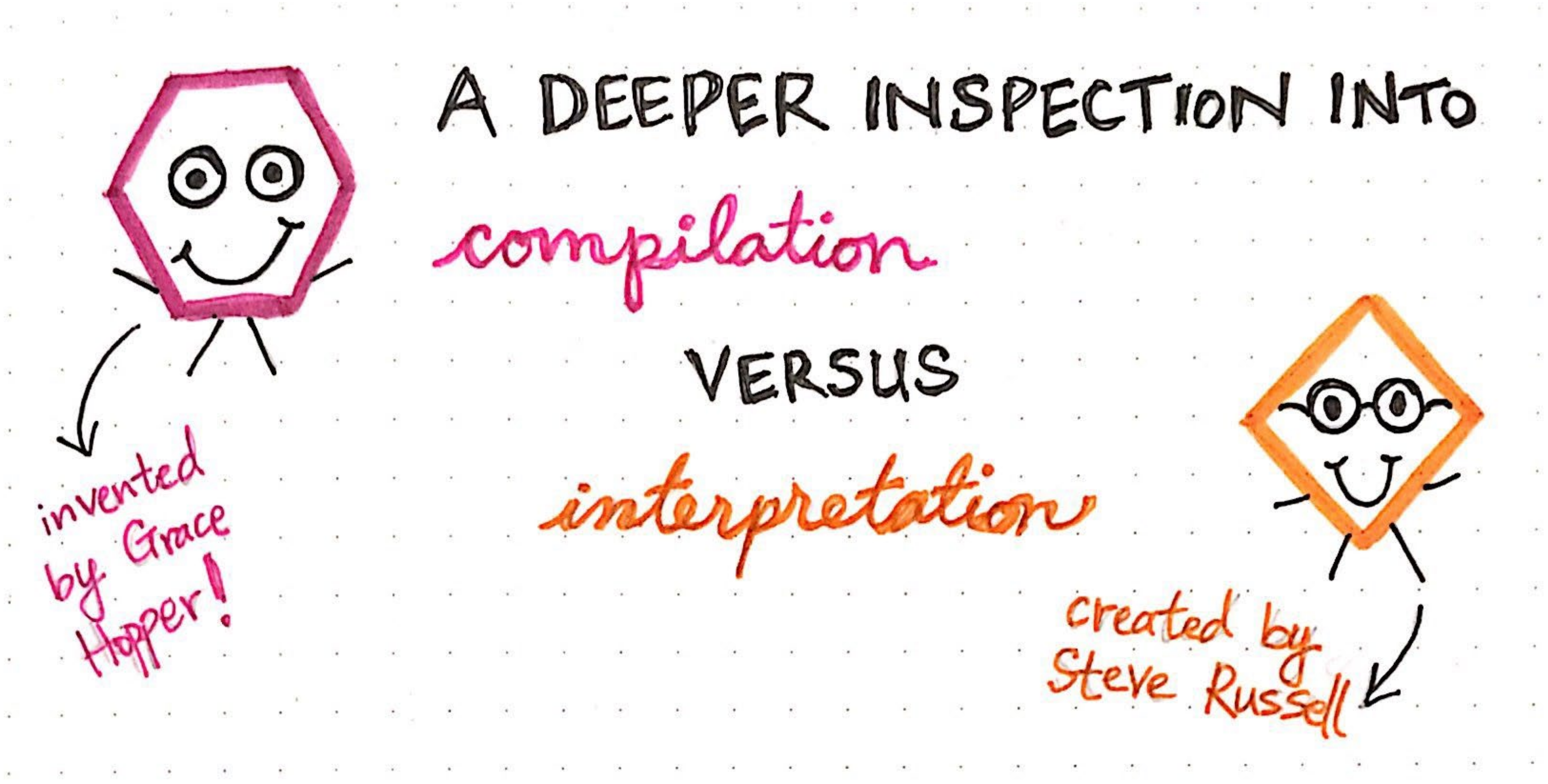
- Compiler have been bundled together with editor and other programs into an integrated development environment (IDE)
- Oriented towards the format or structure of the programming language, called structure-based
- May include some operations of a compiler like reporting about some errors.

Other Programs: Debuggers

- Used to determine execution error in a compiled program
- Keeps track of most or all the source code information
- Halt execution at pre-specified locations called breakpoints
- Must be supplied with appropriate symbolic information by the compiler

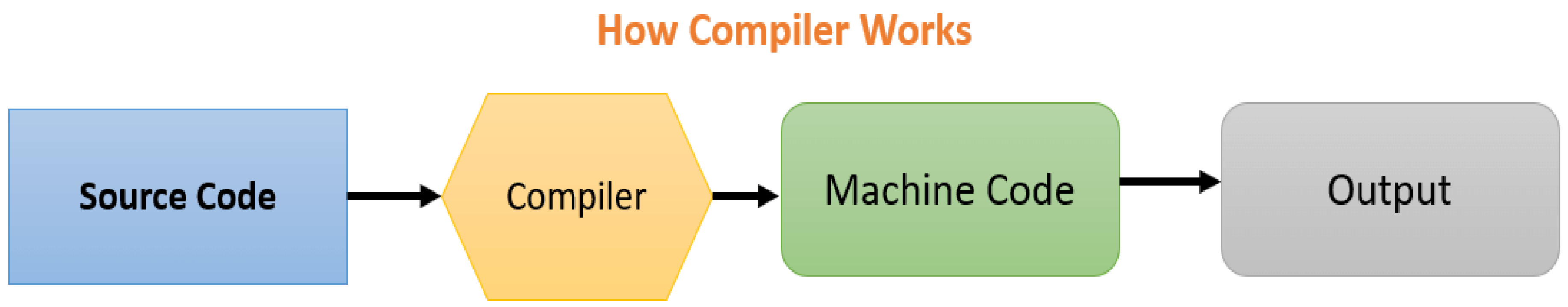
Other Programs: Profiles

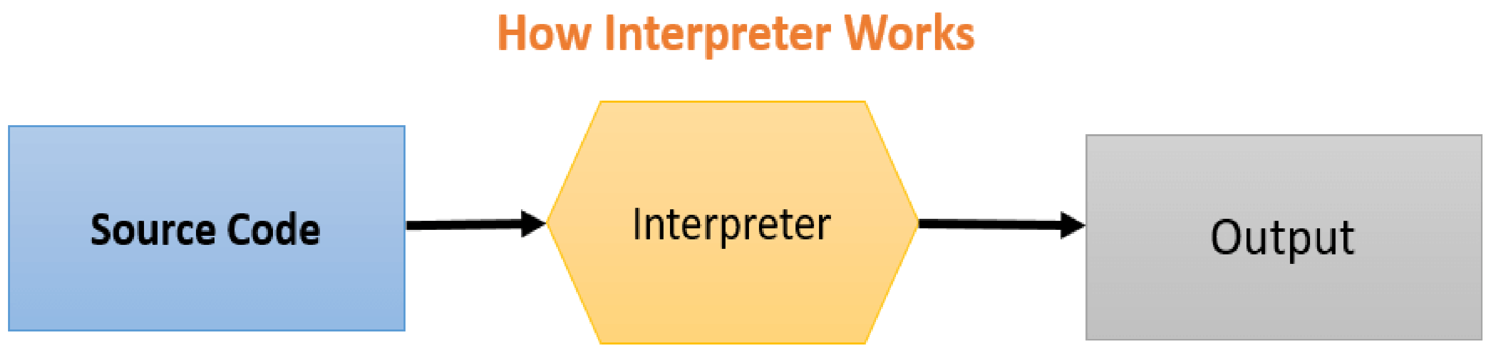
- Collect statistics on the behavior of an object program during execution
  - Called Times for each procedures
  - Percentage of execution time
- Used to improve the execution speed of the program



Compiler *versus* Interpreter

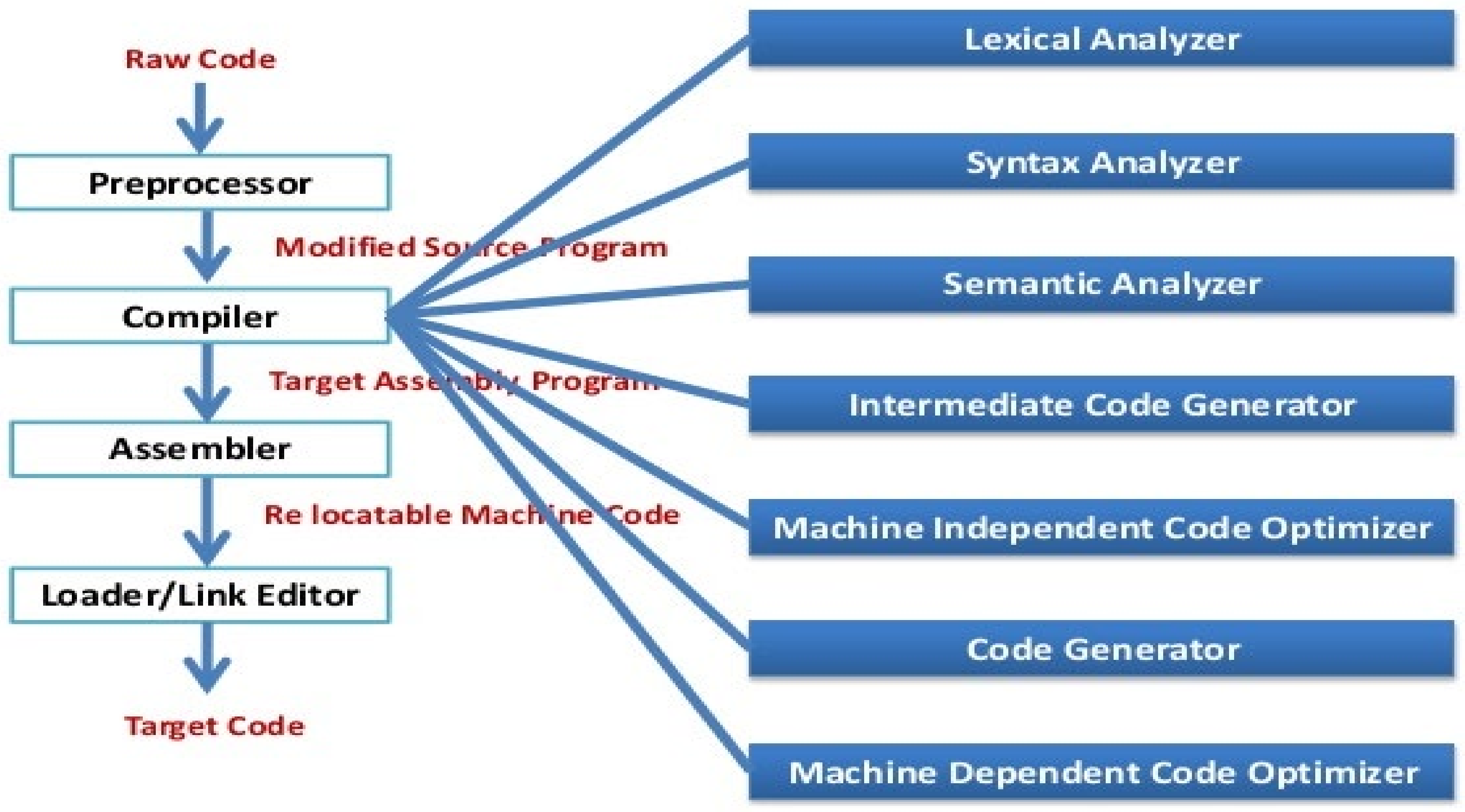
Compilers Vs Interpreters



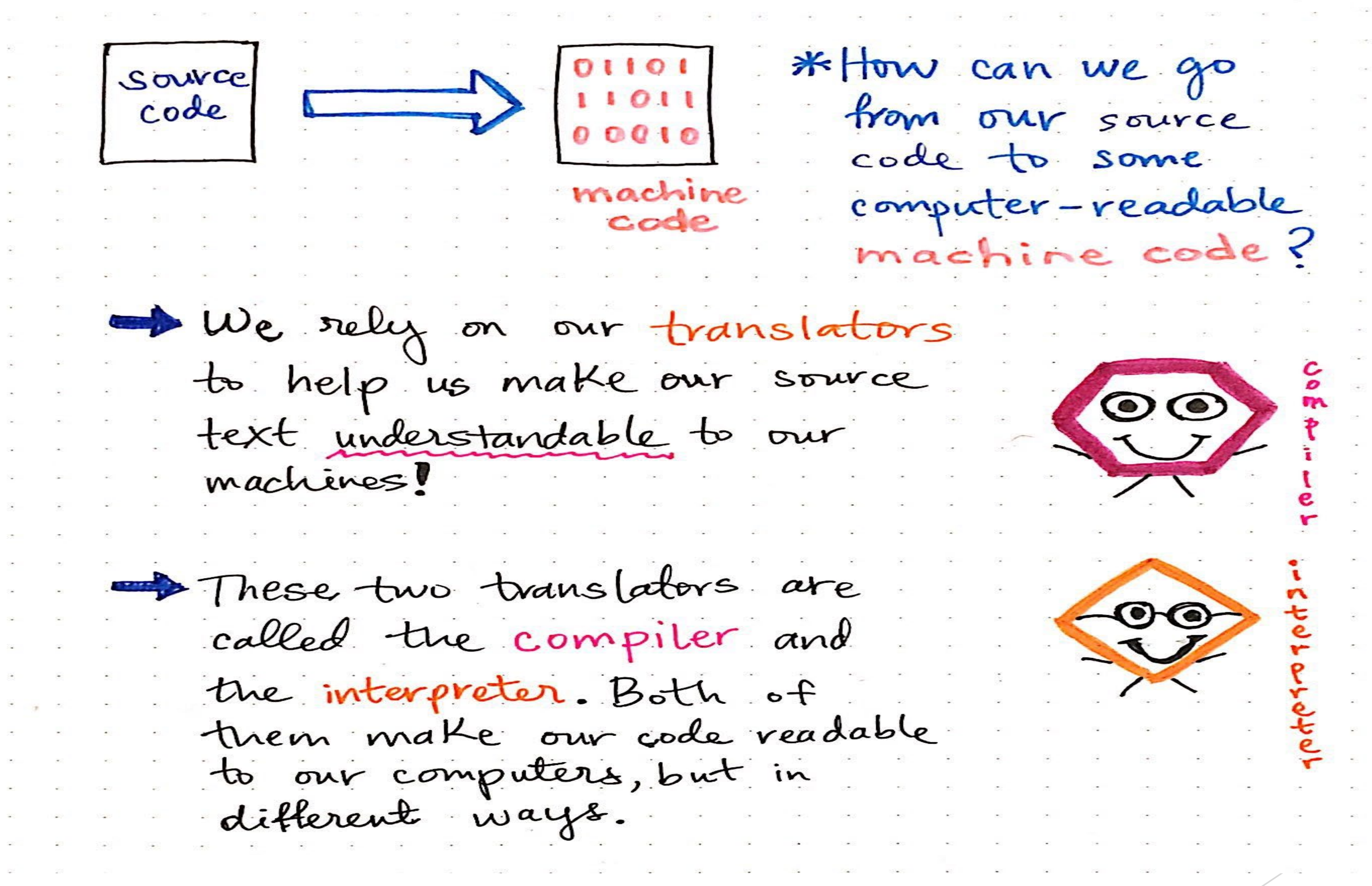


Interpreter	Compiler
Translates a program one statement at a time	Scans the entire program and translates it into machine code
Interpreters usually take less amount of time to analyze the source code. However, the overall execution time is comparatively slower than compilers	Compilers usually take a large amount of time to analyze the source code. However, the overall execution time is comparatively faster than interpreters
No intermediate object code is generated, hence are memory efficient	Generates intermediate object code which further requires linking, hence requires more memory
Programming languages like JavaScript, Python, Ruby use interpreters	Programming languages like C, C++, Java use compilers

Next Lecture



See you next lecture



source code

01101
11011
00010

machine code

*How can we go from our source code to some computer-readable machine code?

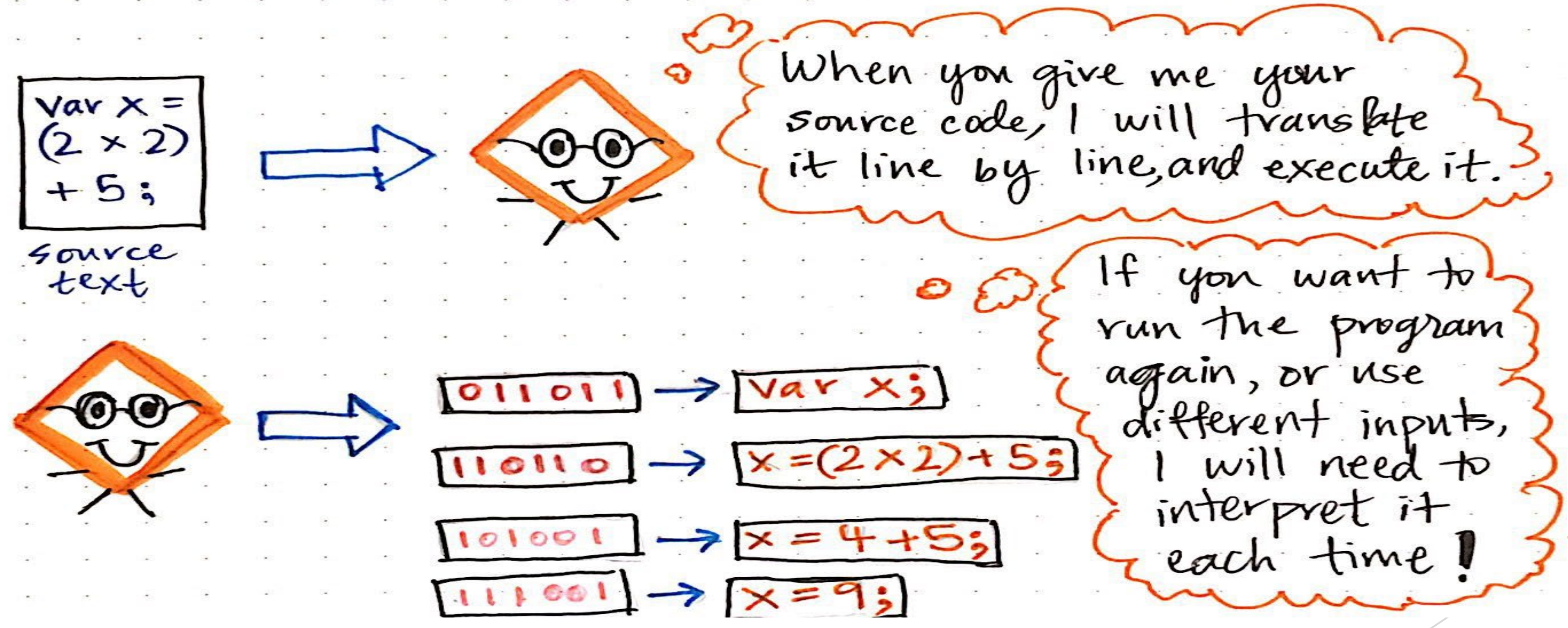
→ We rely on our translators to help us make our source text understandable to our machines!

compiler

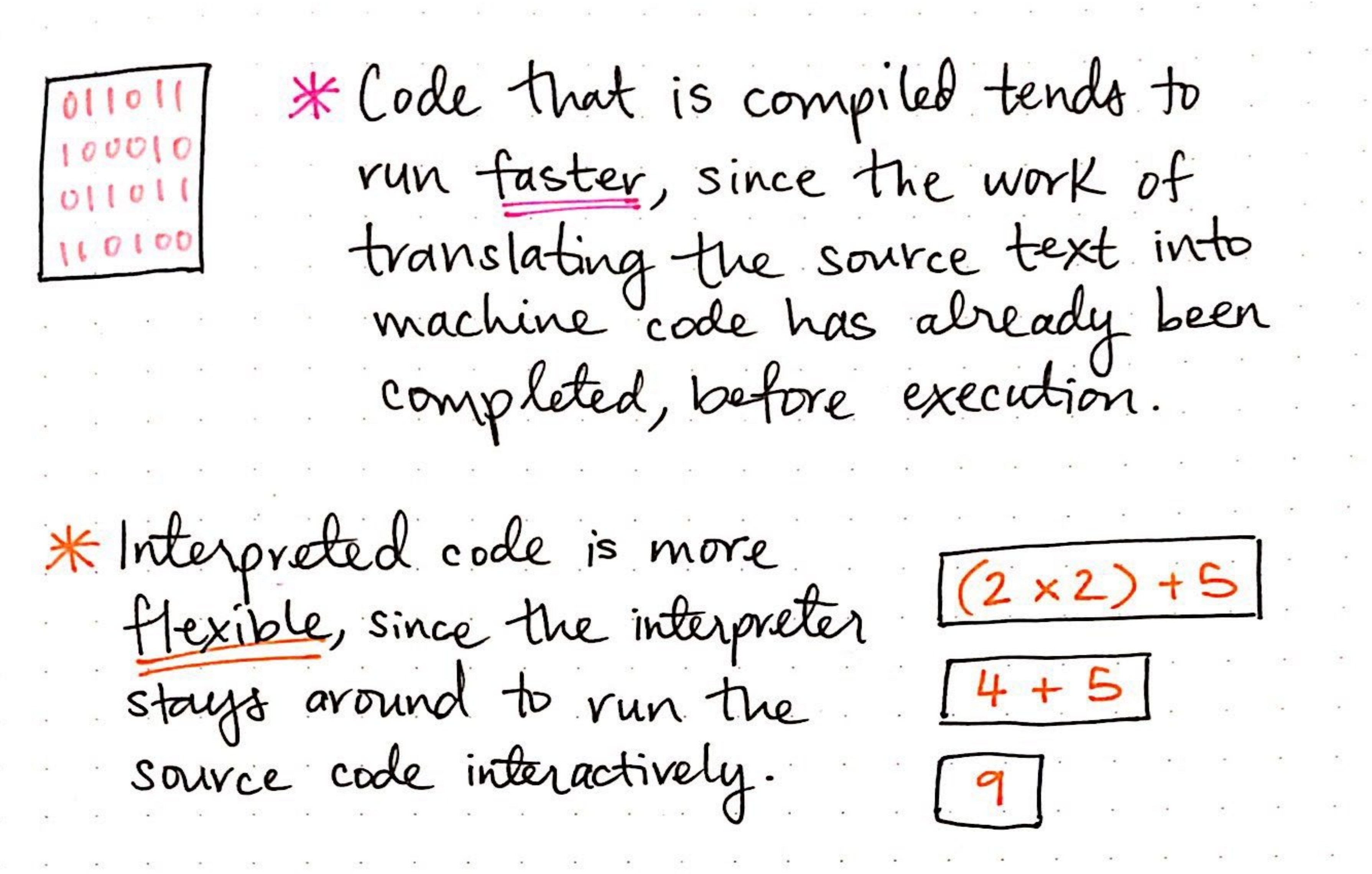
→ These two translators are called the compiler and the interpreter. Both of them make our code readable to our computers, but in different ways.

interpreter

* An interpreter does its job piece by piece. It translates section(s) of the source code at a time, and runs that section right after interpreting it. Once one piece of code has been translated & executed, it will move on to translate/run the next piece.



var x = (2 × 2) + 5;
source text
When you give me your source code, I will translate it line by line, and execute it.
If you want to run the program again, or use different inputs, I will need to interpret it each time!
011011 → var x;
110110 → x = (2 × 2) + 5;
101001 → x = 4 + 5;
111001 → x = 9;



011011
100010
011011
110100

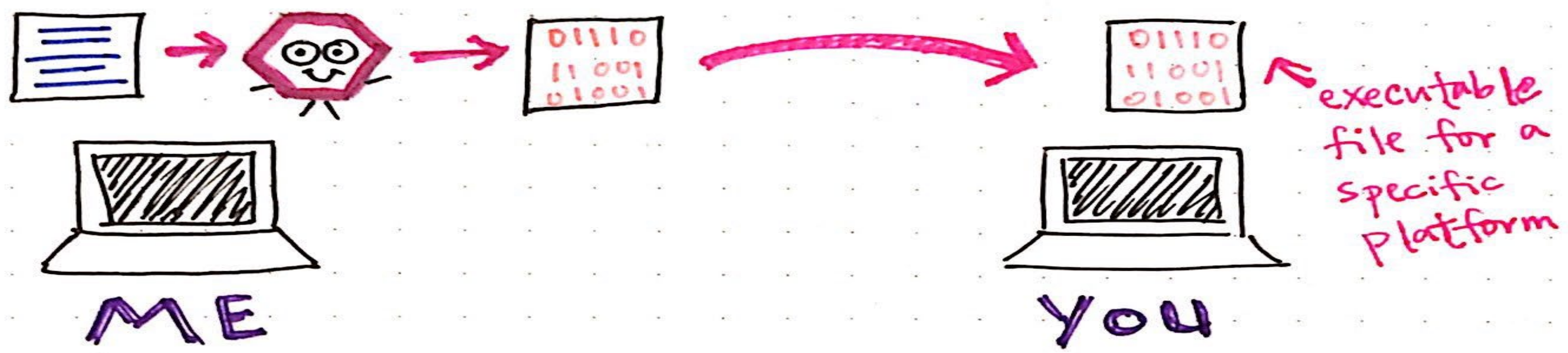
* Code that is compiled tends to run faster, since the work of translating the source text into machine code has already been completed, before execution.

* Interpreted code is more flexible, since the interpreter stays around to run the source code interactively.

$\boxed{(2 \times 2) + 5}$

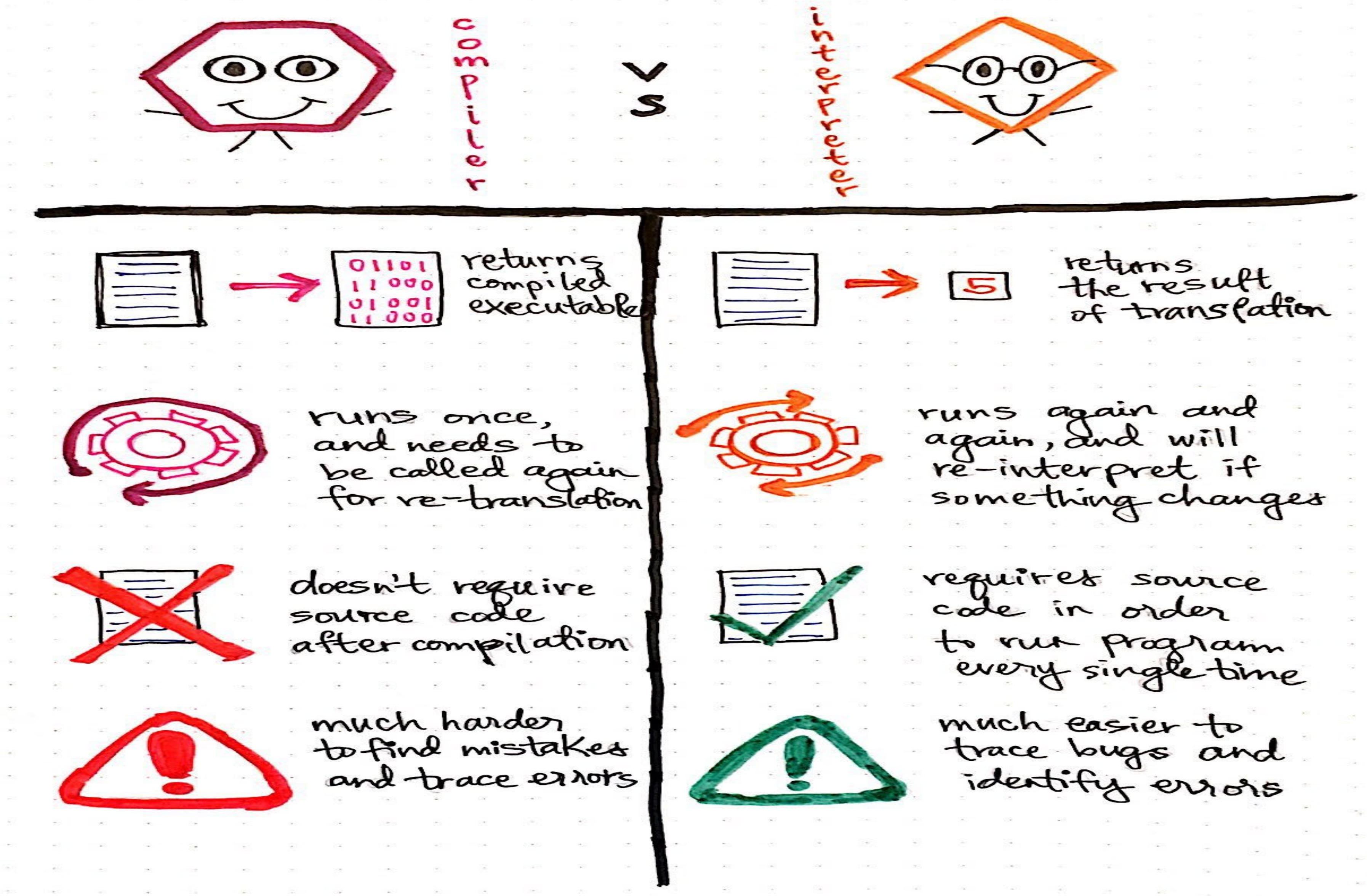
$\boxed{4 + 5}$

$\boxed{9}$



* Using compilation, I can easily + efficiently distribute a program in an executable file, which allows someone else to run the same application without ever seeing or needing the source code.

→ However, if the system that will run the code is on a different platform, the code will need to be recompiled!


compiler
vs
interpreter
01101
11000
01001
11000
returns compiled executable
5
returns the result of translation
runs once, and needs to be called again for re-translation
runs again and again, and will re-interpret if something changes
doesn't require source code after compilation
requires source code in order to run program every single time
much harder to find mistakes and trace errors
much easier to trace bugs and identify errors

Copyright. Vaidehi Joshi, A Deeper Inspection Into Compilation And Interpretation.