**COMPILE CONSTRUCTION LAB FINAL**

**SUBMITTED BY: DUA NADEEM**

**SUBMITTED TO: SIR. BILAL HAIDER**

**REGISTRATION N0: FA21-BCS-016**

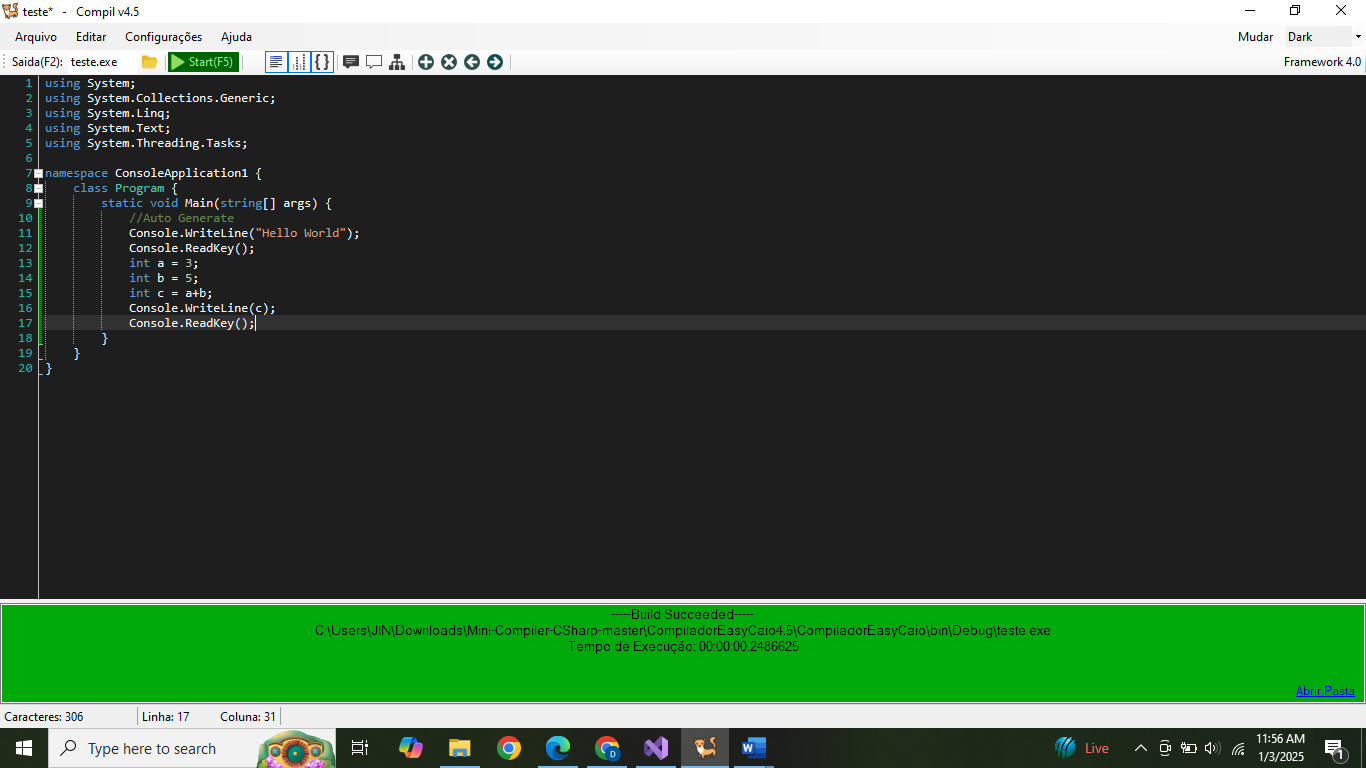
**Question#2**

**Explain the input output of your mini compiler.**

**Input of the Mini Compiler:**

* **Source Language Code:** The input to this compiler is source code written in the mini-language defined for this specific project. This is not a general-purpose programming language like C#, Python, or Java. The source code is meant to adhere to the grammar and features of this mini-language.
* **Text Format:** The source code is expected to be in plain text format. It will usually exist in files with specific extensions, but that doesn't apply here because of the interpreter pattern, it may read or received source code from the virtual file system provided to compiler object
* **Basic Elements:** This mini-language likely includes (based on my analysis and as you requested before):
  + Integer data type.
  + Arithmetic operations (+, -, \*, /).
  + Variable declarations and assignments (int x = 10;, x = 20;).
  + Conditional statements (if/else).
  + Looping constructs (while).
  + Basic input and output functionality (potentially with read and write equivalents, even print functionality using console output instead).
  + Function definitions and calls.
* **Input Mechanism:** The Program.cs is where it takes that raw text as an string which could be coming from hard-drive storage from file as a program to compile (when it reads file content from external source). So as it reads each line for this program input will store it a "memory" and it parses using internal data-structure

**Example Input**



**Output of the Mini Compiler:**

* **Intermediate Code:** The output is not executable machine code directly suitable for running on a physical CPU. It's a form of intermediate code representation for further running through virtual machine implemented with language specifications in virtual machines pattern. That kind of execution environment makes to compile easier rather creating compiled files and more complex file management for running every single time compilation processes by compiler developer. So instead of targeting an operating system format (which a common behavior) the main result for this type of projects for educational scope will usually follow "intermediate representation" way.
* **Structure:** The specific format is unique and custom made for this specific compiler project. As indicated before (as well in common compilers with the intermediary forms), they may resemble instructions at a high level.
  + They tend to consist of *opcodes* or a equivalent (an instruction or operator) and *operands*. It does look similar as "assembler".
  + Usually using a simple structured line notation way and a specific instruction per-line, but not as OS code output execution where can exists "memory zones" with different structure and meanings.
* **Symbolic-Text output**: It uses textual/symbolic way to produce those intermediate forms of the application program in compilation, since it isn't compiling as an actual operative-system execution process.
* **Not Executable Directly:** The output of the compiler needs further steps to make it actually "run".

**Assembler approach** the instructions also could be transformed with assembly-language and that through an assembler could end on a executable binary file for running, even a direct interpretation using code pattern instead an intermediate transformation stage before going to virtual execution phase, but is rarely found that process due code simplification goals and project focusing (which as told are "instruction execution by interpreter engine in code")

* **Example Output**

