**Compilers**

Lecture 1

Lecturer Dr. Zainab Hashim

Third stage – First semester

Computers Department

**2022-2023**

**Lecture1: Introduction to Compiler Structure**

**Programming Languages**

Hierarchy of Programming Languages based on increasing machine independence includes the following:

1- **Machine – level languages:** is the lowest form of computer. Each instruction in program is represented by numeric code, and numerical addresses are used throughout the program to refer to memory location in the computer’s memory.

2- **Assembly languages:** is essentially symbolic version of machine level language, each operation code is given a symbolic code such ADD for addition and MULT for multiplication.

3- **High – level or user oriented languages**: such as Pascal, C.

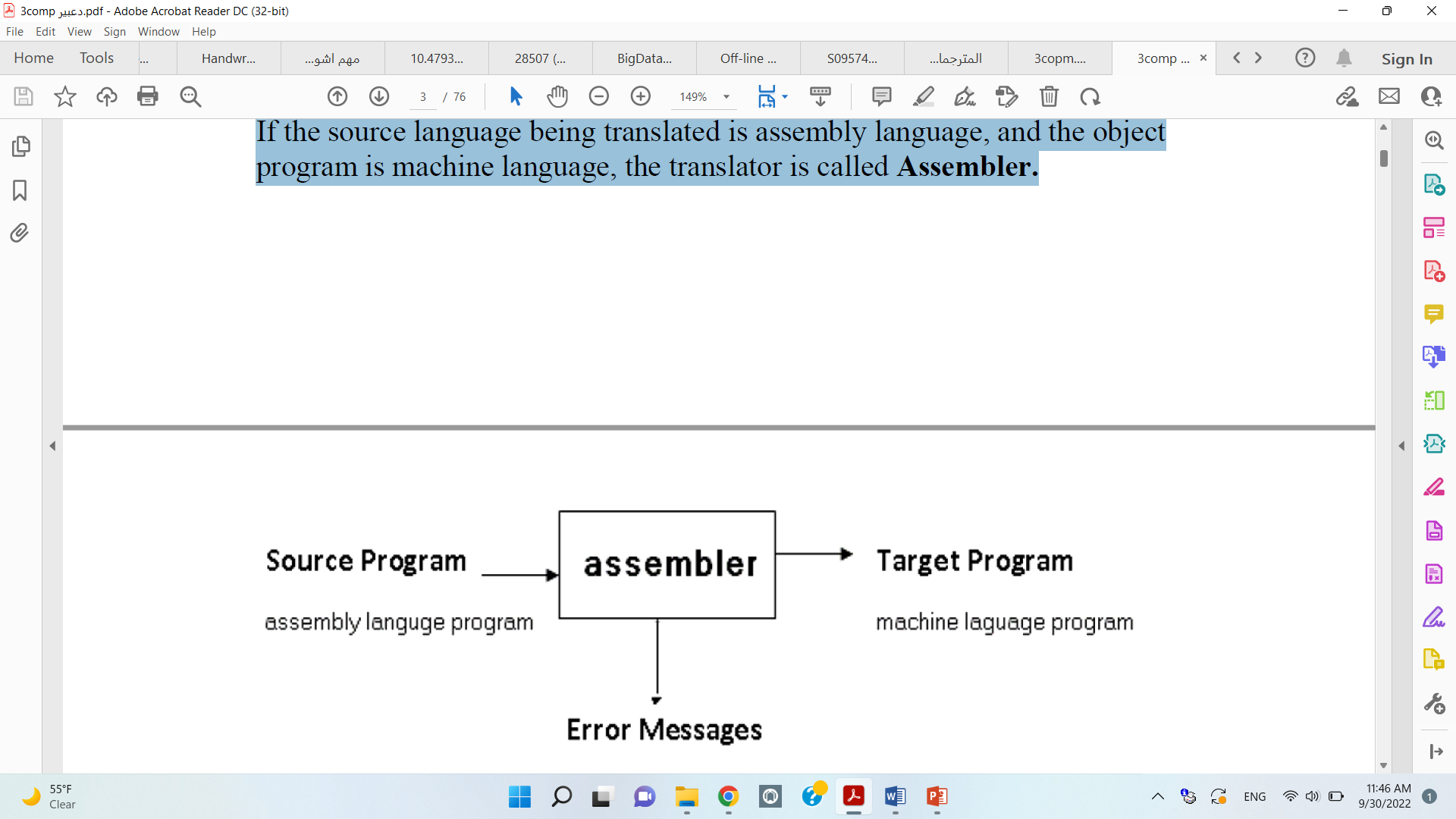
4- **Problem - oriented language**: provides for the expression of problems in specific application or problem area .examples of such as languages are SQL for database retrieval application problem oriented language.

**Translator**

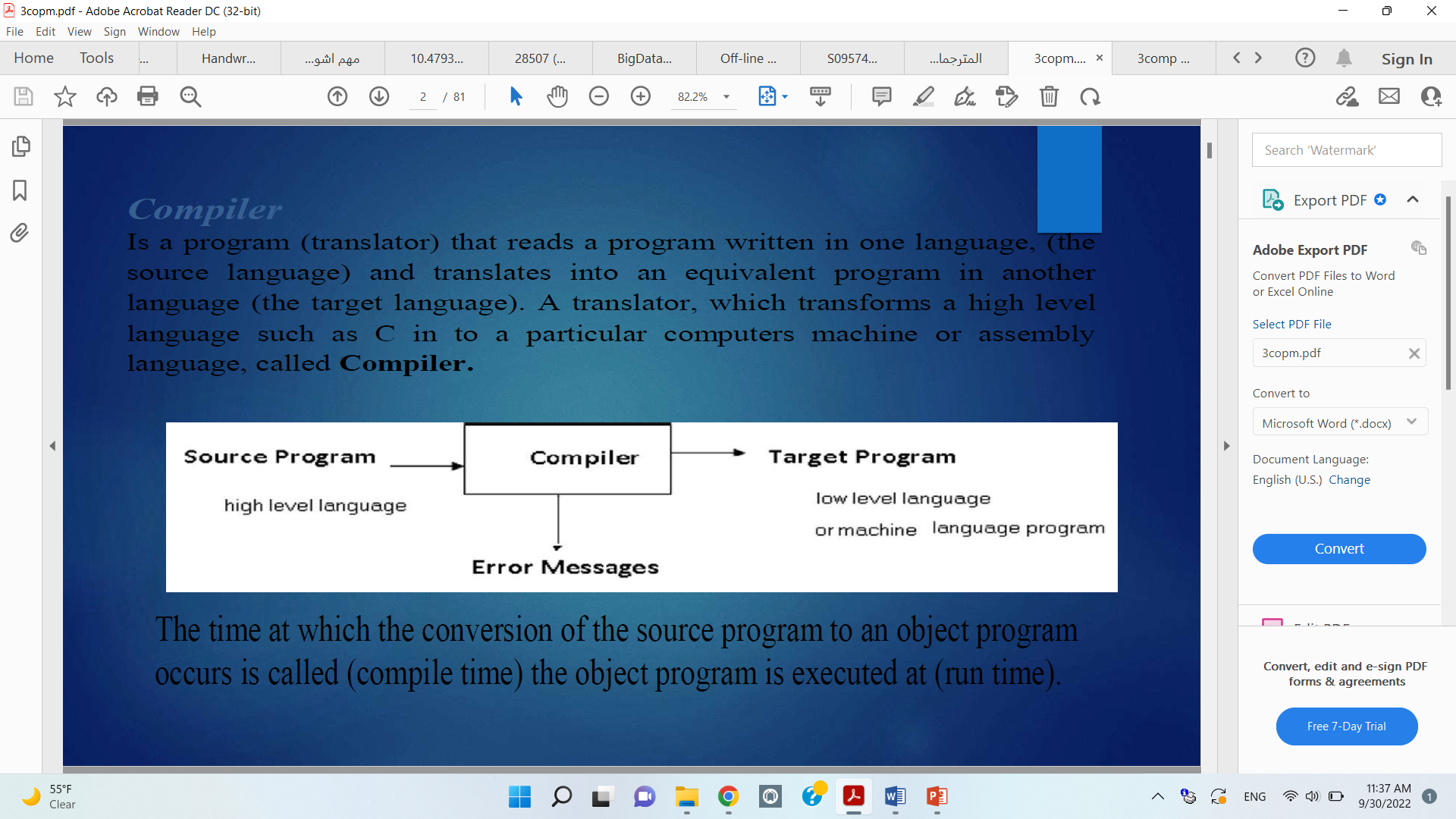
**A translator** is program that takes as input a program written in a given programming language (the source program) and produce as output program in another language (the object or target program).

* If the source language being translated is assembly language, and the object

program is machine language, the translator is called **Assembler.**



* If a translator transforms a high level language such as C in to a particular computers machine or assembly language, called **Compiler.**



**Compiler:** Is a program that reads a program written in one language -the Source Language- and translates it into an equivalent program in another language – the Target Language.

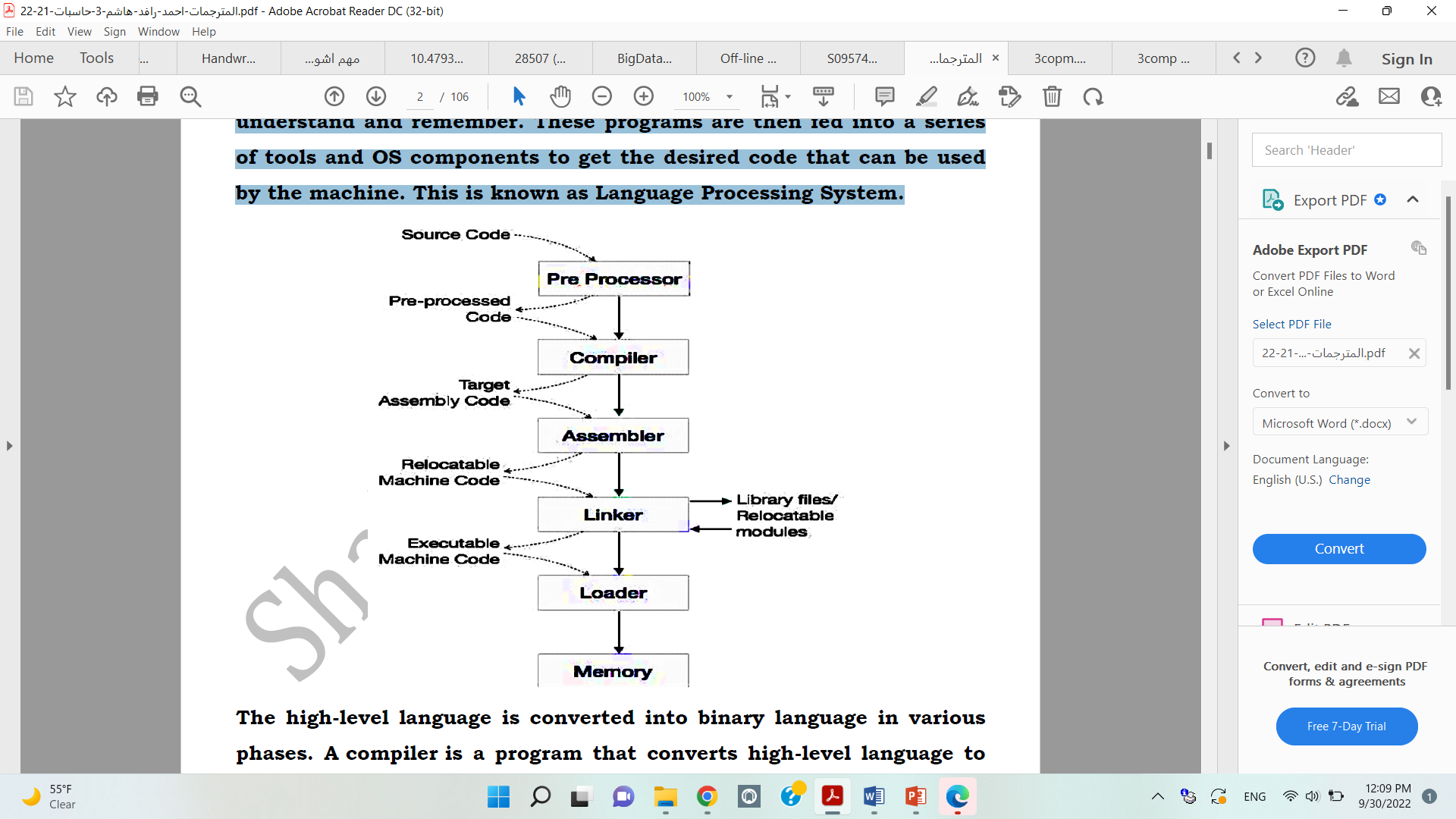
A compiler translates the code written in one language to some other language **without changing the meaning of the program.** It is also expected that a compiler should make the target code efficient and optimized in terms of time and space. As an important part of this translation process, the compiler reports to its user the presence of errors in the source program.

**Compiler Design**

Computers are a balanced mix of software and hardware. Hardware is just a piece of mechanical device and its functions are being controlled by compatible software. Hardware understands instructions in the form of electronic charge, which is the counterpart of binary language in software programming. Binary language has only two alphabets, 0 and 1. To instruct, the hardware codes must be written in binary format, which is simply a series of 1s and 0s. It would be a difficult task for computer programmers to write such codes, which is why we have compilers to write such codes.

**Language Processing System (نظام معالجة اللغات)**

Any computer system is made of hardware and software. The hardware understands a language, which humans cannot understand. So we write programs in high-level language, which is easier for us to understand and remember. **These programs are then fed into a series of tools and OS components to get the desired code that can be used by the machine. This is known as Language Processing System.**



**Let us first understand how a program, using C compiler, is executed on a host machine.**

1. User writes a program in C language (High-Level Language).

2. The C *compiler* compiles the program and translates it to assembly program (Low-Level Language).

3. *An Assembler* then translates the assembly program into machine code (object).

4. *A Linker* tool is used to link all the parts of the program together for execution (Executable Machine Code).

5. *A Loader* loads all of them into memory and then the program is executed.

**Compiler Architecture**

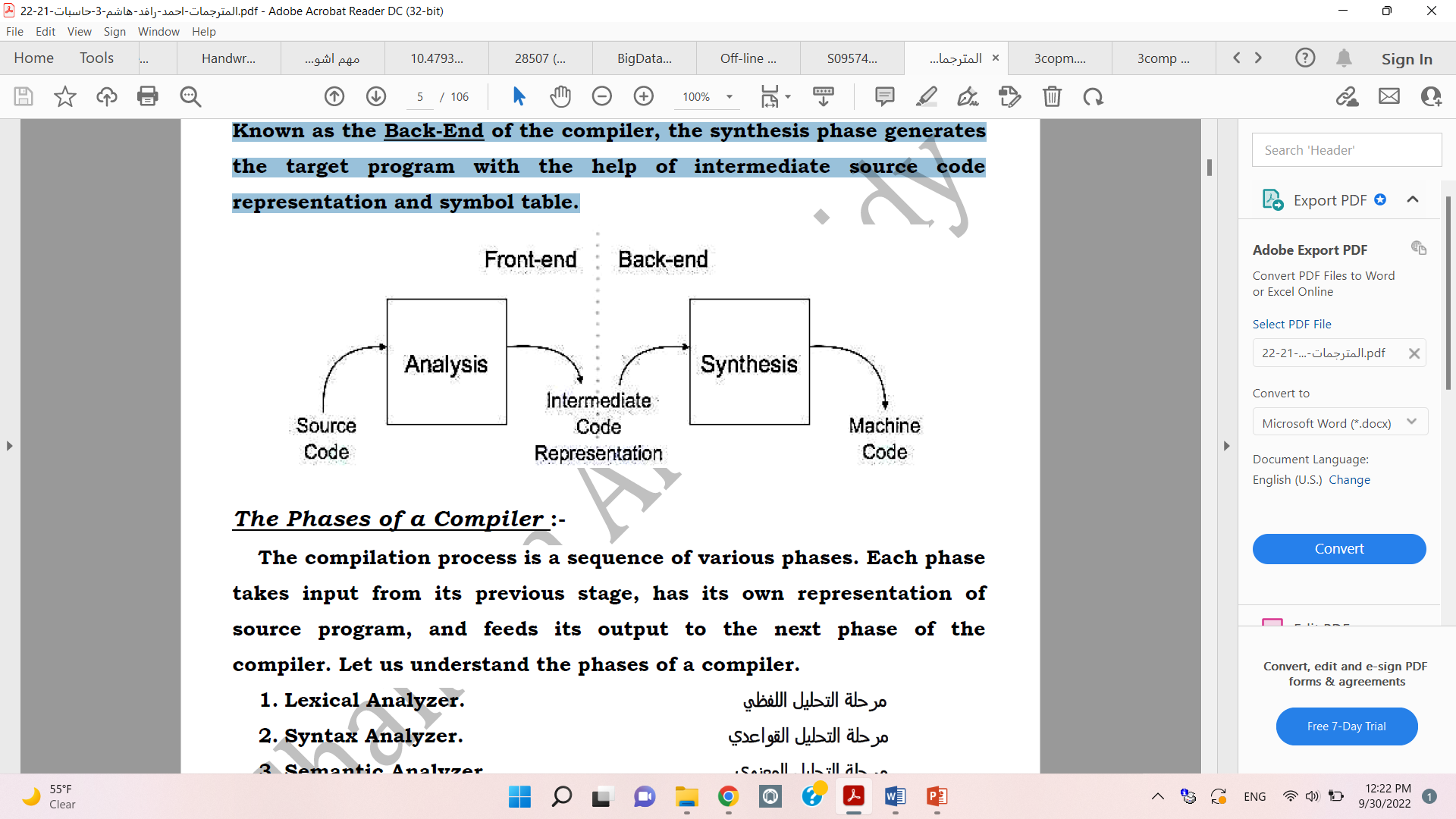
**A compiler can broadly be divided into two phases based on the way they compile.**

1. **Analysis Phase**

Known as the Front-End of the compiler, the analysis phase of the compiler reads the source program, divides it into core parts and then checks for lexical, grammar and syntax errors. The analysis phase generates an intermediate representation of the source program and symbol table, which should be fed to the Synthesis phase as input.

1. **Synthesis Phase**

Known as the Back-End of the compiler, the synthesis phase generates the target program with the help of intermediate source code representation and symbol table.



**The Phases of a Compiler**

The compilation process is a sequence of various phases. Each phase takes input from its previous stage, has its own representation of source program, and feeds its output to the next phase of the compiler. Let us understand the phases of a compiler.

**1. Lexical Analyzer.** مرحلة التحليل اللفظي

**2. Syntax Analyzer.** مرحلة التحليل القواعدي

**3. Semantic Analyzer.** مرحلة التحليل المعنوي

**4. Intermediate Code Generator.** مرحلة توليد الشفرات الوسطية

**5. Code Optimizer.** مرحلة تحسين الشفرات

**6. Code Generator.** مولد الشفرات

**In each phase we need variables that can be obtained from a table called *Symbol Table manager*, and in each phase some errors may be generated so we must have a program used to handle these errors, this program called *Error Handler*.**

***Symbol Table: -*** It is a data-structure maintained throughout all the phases of a compiler. All the identifiers’ names along with their types are stored here. The symbol table makes it easier for the compiler to quickly search the identifier record and retrieve it.

Symbol table operations:

1-insert(s,t) : this function is to add a new name to the table

2-Lookup(s) : returns index of the entry for string s

3-Delete a name or group of names from tables

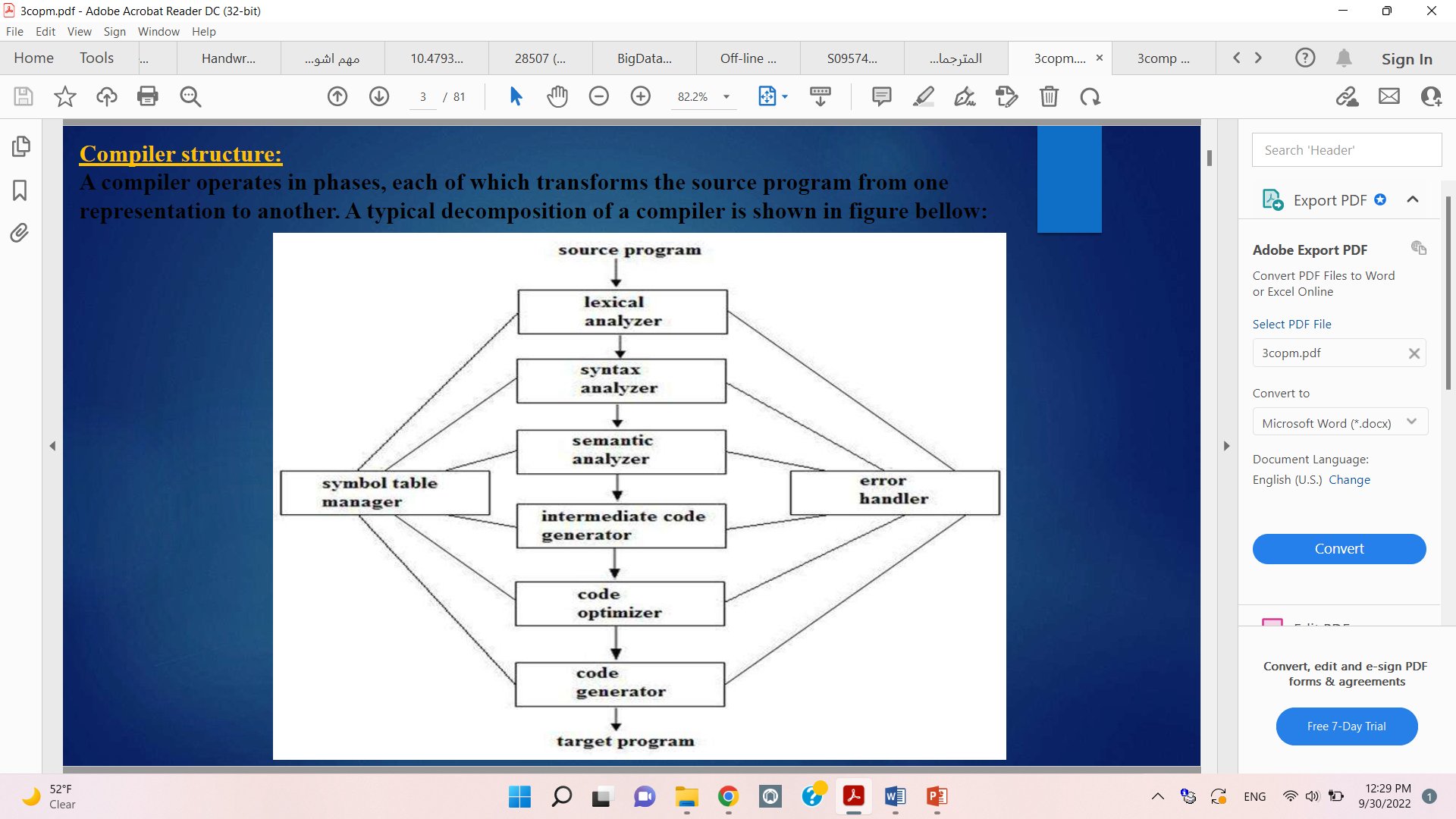
4- Update

5-Search

5-Sear

***Error Handler: -*** Each phase can produce errors. However, after detecting an error, a phase must deal with that error, so that the compilation can proceed. So dealing with that error is done by a program known as Error Handler which is software used to handle any error that may be produced from any phase and it is needed in all phases of the compliers.

Q) What is the use of a symbol table in compiler design?



**Lexical Analyzer***:* Is the initial part of reading and analyzing the program text (source program); The text is read (character by character) and divided into tokens, each of which corresponds to a symbol in the programming language, e.g., a variable name, keyword or number.

**Syntax analyzer**: The next phase is called the syntax analysis or parsing. It takes the token produced by lexical analysis as input and generates a parse tree (or syntax tree) that reflects the structure of the program. This phase is often called parsing.

**Semantic Analysis***:* Semantic analysis checks whether the parse tree constructed follows the rules of language. Also is known as Type checking which main function is to analyze the syntax tree to determine if the program violates certain consistency requirements, such as, if a variable is used but not declared, assignment of values is between compatible data types, and adding string to an integer.

***Intermediate Code Generator*:** After syntax and semantic analysis, It is in between the high-level language and the machine language. This intermediate code should be generated in such a way that it makes it easier to be translated into the target machine code. This phase bridges the analysis and synthesis phases of translation.

**Code Optimization phase**: The code optimization phase attempts to improve the intermediate code which results that the output runs faster and takes less space. Optimization can be assumed as something that removes unnecessary code lines, and arranges the sequence of statements in order to speed up the program execution without wasting resources (CPU, memory).

**Code Generator**: The final phase of complier is the generation of target code, which represents the output of the code generator in the machine language.

|  |  |
| --- | --- |
| المرحلة الثالثة المادة: مترجمات  الفصل الاول عدد الساعات: 3  مفردات المنهج | |
| 1 | . Introduction: The Need for Compilers and Assemblers |
| 2 | .Assembly Language Used as Compiler Object Code: Instruction Set and Machine Code Format |
| 3 | Lexical Analysis: Translation from Character Stream to Symbol Stream |
| 4 | An Assembler: Label Table, Backpatching, Assembly of Individual Instructions, A complete Assembler Program |
| 5 | Address Binding: Relocation, Multisegment Programs, Linking and Loading |
| 6 | Context-Free Grammars: Parse Trees, Leftmost and Rightmost Derivations, Ambiguous Grammars, Extended Backus-Naur Form, Bottom-Up and Top-Down Backtrack Parsing |
| 7 | Predictive Parsing: Parse Table, Non-Recursive Parsing Algorithm, Construction of Parse Table, Definition of LL(1) Grammars, Transformation to LL(1), Recursive Descent Parsing Algorithm, Recursive Descent Compilation of Simple Expressions |
| 8 | Semantic Analysis: Type Checking, Attribute Grammars, Optimization, Errors |
| 9 | . Paradigmatic Issues: Compiling Different Language Types (Imperative, Functional, Relational and Object-Oriented) |