**PROGRAMMING LANGUAGE:**

A programming language is a computer language programmers use to develop software programs, scripts, or other sets of instructions for computers to execute.

Although many languages share similarities, each has its own syntax. Once a programmer learns the languages rules, syntax, and structure, they write the source code in a text editor or IDE. Then, the translator translates the code into machine language that can be understood by the computer.

**5 major types of programming languages**

While you'll find dozens of ways to classify various programming languages, they generally fall into five major categories. Keep in mind that some languages may fall under more than one type:

**1. Procedural programming languages**

A procedural language follows a sequence of statements or commands in order to achieve a desired output. Each series of steps is called a procedure, and a program written in one of these languages will have one or more procedures within it. Common examples of procedural languages include:

* C and C++
* Java
* Pascal
* BASIC

**2. Functional programming languages**

Rather than focusing on the execution of statements, functional languages focus on the output of mathematical functions and evaluations. Each function performs a specific task and returns a result. The result will vary depending on what data you input into the function. Some popular functional programming languages include:

* Scala
* Erlang
* Haskell
* Elixir
* F#

**3. Object-oriented programming languages**

This type of language treats a program as a group of objects composed of data and program elements, known as attributes and methods. Objects can be reused within a program or in other programs. This makes it a popular language type for complex programs, as code is easier to reuse and scale. Some common object-oriented programming (OOP) languages include:

* Java
* Python
* PHP
* C++
* Ruby

**4. Scripting languages**

Programmers use scripting languages to automate repetitive tasks, manage dynamic web content, or support processes in larger applications. Some common scripting languages include:

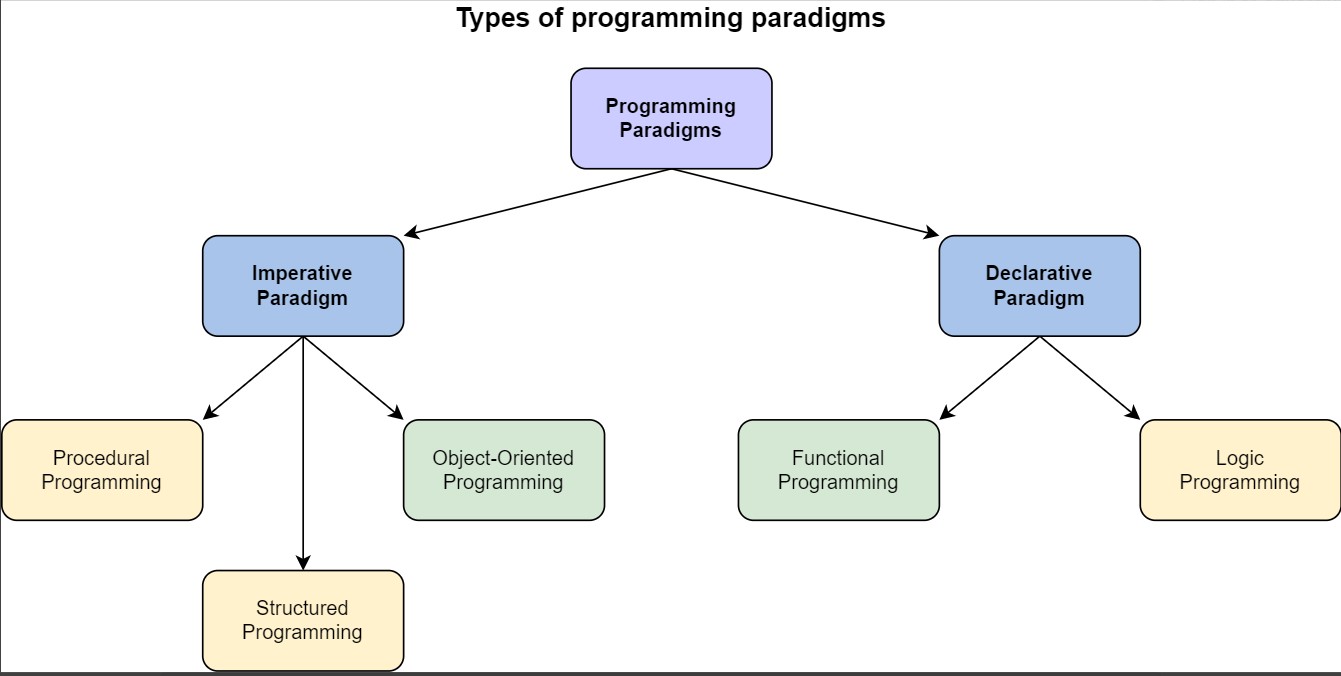
* PHP
* Ruby
* Python
* bash
* Perl
* Node.js

**5. Logic programming languages**

Instead of telling a computer what to do, a logic programming language expresses a series of facts and rules to instruct the computer on how to make decisions. Some examples of logic languages include:

* Prolog
* Absys
* Datalog
* Alma-0

**Programming paradigms** are approaches used to categorize or classify programming languages based on techniques and features they support.



**What is imperative programming?**

Imperative programming is the oldest and most basic programming approach. Within the imperative paradigm, code describes a step-by-step process for a program’s execution. Because of this, beginners often find it easier to reason with imperative code by following along with the steps in the process.

Examples of imperative programming languages include:

* Java
* C
* Pascal
* Python
* Ruby
* Fortran
* PHP

Say you want to build an app that returns the current weather and forecast for a given location. At a high level, you might design the app to work something like this when using an imperative approach:

* Begin  
  Accept location from user input of either location name or ZIP code.  
  Call OpenWeather's Geocoding API to convert location data into geographic coordinates.  
  Call OpenWeather's Current Weather Data API.  
  Send geographic coordinates to OpenWeather.   
  Call OpenWeather's Daily Forecast 16 Days API.  
  Resend geographic coordinates.  
  Parse JSON returned by the APIs to extract current weather and forecast data.   
  Return current weather and forecast.  
  Display current weather and forecast to user.  
  End

In this simple example, imperative instructions dictate what the app should do, when to do it, and how to do it. This pseudocode is comparable to imperative programming, with which you create the logic of the program by making looping statements, calling functions, etc., all in a particular order.

**What is declarative programming?**

In contrast with imperative programming, declarative programming describes **what you want the program to achieve** rather than how it should run.

In other words, within the declarative paradigm, you define the results you want a program to accomplish without describing its control flow. Ultimately, it’s up to the programming language’s implementation and the compiler to determine how to achieve the results. This places emphasis not on the execution process, but on the results and their ties to your overall goal.

Examples of declarative programming languages include:

* SQL
* Miranda
* Prolog
* Lisp
* Many markup languages (e.g., HTML)

Returning to our weather app example, the pseudocode might look something like this in the declarative paradigm:

* Begin  
  Location submitted by user is location name or ZIP code.  
  Location is converted into geographic coordinates.  
  Weather data is retrieved for geographic coordinates.  
  Weather data is displayed for user.  
  End

As shown, the pseudo code is descriptive but lacks in detail. Only the result, displaying the weather data, matters to you without regard for the process.

|  |  |
| --- | --- |
| **Imperative Programming** | **Declarative Programming** |
| **1. Computation** | |
| 1. You describe the **step-by-step instructions** **for how** an executed program achieves the desired results. | 1. You set the conditions that trigger the program execution to produce the desired results. |
| **2. Readability and complexity** | |
| 1. However, as you add more features and code to your program, it can become **longer and more complex**, making it increasingly confusing and time-consuming to read. | 1. You’ll discover that this paradigm is **less complex and requires less code**, making it easier to read. |
| **3. Customization** | |
| 1. A straightforward way to customize and edit code and structure is offered. You have **complete control** and can easily adapt the structure of your program to your needs. | 1. **Customizing the source code is more difficult** because of complicated syntax and the paradigm’s dependence on implementing a pre-configured algorithm. |
| **4. Optimization** | |
| 1. Adding extensions and making upgrades are supported, but doing so is **significantly more challenging** than with declarative programming, making it harder to optimize. | 1. You can **easily optimize code** because an algorithm controls the implementation. |
| **5. Structure** | |
| 1. The code structure can be **long and complex**. Due to the increased complexity, the code can sometimes be confusing because it may perform more than one task. | 1. The code structure is **concise and precise**, and it lacks detail. Not only does this paradigm vastly limit the complexity of your code, but the code is more efficient. |

**Computer Language Translator:**

A translator is a computer program that translates a program written in a given programming language into a functionally equivalent program in a different language.

Depending on the translator, this may mean changing or simplifying the flow of the program without changing its core. This makes a program that works the same as the original.

**Types of Language Translators:**

There are mainly three types of translators that are used to translate different programming languages into machine-equivalent code:

1. Assembler
2. Compiler
3. Interpreter
4. **Assembler**

An assembler translates assembly language into machine code.

Assembly language consists of mnemonics for machine op-codes, so assemblers perform a 1:1 translation from mnemonic to direct instruction. For example, LDA #4 converts to 0001001000100100.

**Benefits:**

Here is a list of the advantages of using assembler:

* As a 1 to 1 relationship, assembly language to machine code translation is very **fast**.
* Assembly code is often very **efficient** (and therefore **fast**) because it is a low-level language.
* Assembly code is fairly **easy to understand** due to the use of English, like in mnemonics.

**Drawbacks:**

* Assembly language is written for a **certain** instruction set and/or processor.
* Assembly tends to be optimized for the hardware it is designed for, meaning it is often **incompatible** with different hardware.
* **Lots** of assembly code is needed to do a relatively simple task, and complex programs require lots of programming **time**.

1. **Interpreter**

An interpreter program executes other programs directly, running through the program code and executing it line-by-line. As it analyses every line, an interpreter is **slower** than running compiled code, but it can take less time to interpret program code than to compile and then run it.

Interpreters are written for multiple platforms; this means code written once can be immediately run on different systems without having to recompile for each.

Examples of this include **flash-based** web programs that will run on your PC, Mac, gaming console, and mobile phone.

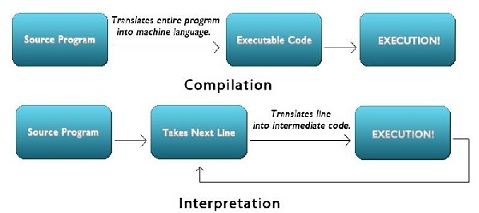
**Advantages:**

Here is a list of some of the main advantages of using an interpreter:

* **Easier to debug** (check errors) than a compiler.
* It is **easier to create multi-platform code,** as each different platform would have an interpreter to run the same code.
* Useful for **prototyping** software and **testing** basic program logic.

**Disadvantages:**

* Source code is required for the program to be executed, and this source code can be read, making it **insecure**.
* Due to the on-line translation method, interpreters are generally **slower** than compiled programs.



1. **Compiler**

A compiler is a computer program that translates code written in a high-level language into a low-level language, machine code.

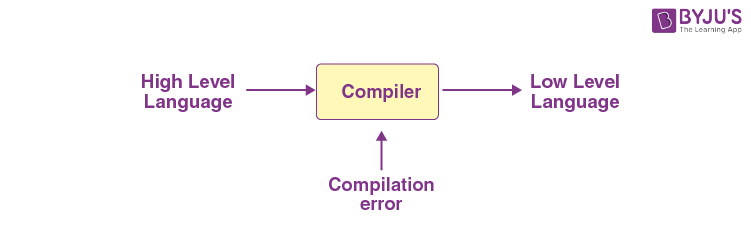
The most common reason for translating source code is to create an executable program (converting from high-level language into machine language).

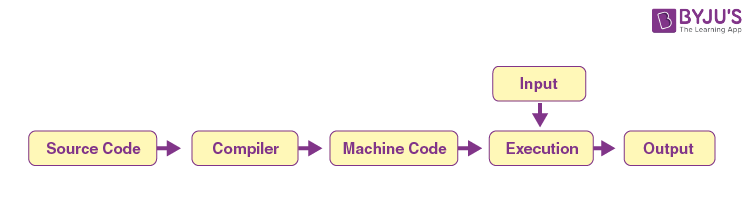
**Advantages:**

* Source code is **not included**; therefore, compiled code is more **secure** than interpreted code.
* Tends to produce **faster** code and is **better** at interpreting source code.
* Because the program generates an executable file, it can be **run without** the need for the **source code**.

**Disadvantages:**

* Before a final executable file can be created, object code must be generated; this can be a **time-consuming** process.
* The source code must be **100% correct** for the executable file to be produced.

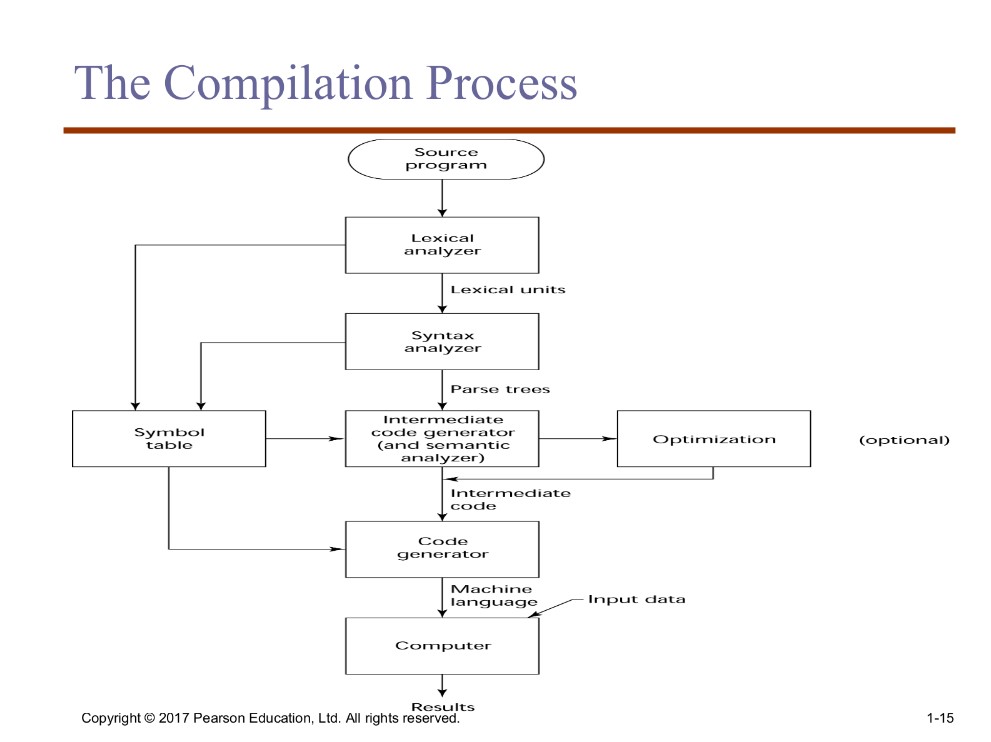




**Phases of Compiler:**

The 6 phases of a compiler are:

1. Lexical Analysis
2. Syntactic Analysis or Parsing
3. Semantic Analysis
4. Intermediate Code Generation
5. Code Optimization
6. Code Generation



**Lexical Analysis:** Lexical Analysis is the first phase of the compiler also known as a scanner. It converts the High level input program into a sequence of **Tokens**.

* Lexical Analysis can be implemented with the [Deterministic finite Automata](https://www.geeksforgeeks.org/introduction-of-finite-automata/).
* The output is a sequence of tokens that is sent to the parser for syntax analysis

**Regular expressions** play a crucial role in defining lexemes by specifying the patterns they should follow. Lexical analysis, performed by a tool called a lexer or scanner, uses regular expressions to tokenize the source code into meaningful lexemes.

**What is a token?**

A lexical token is a sequence of characters that can be treated as a unit in the grammar of the programming languages.

Example of tokens:

* **Keywords Examples:** for, while, if etc.
* **Identifier Examples:** Variable name, function name, etc.
* **Operators Examples:** '+', '++', '-' etc.
* **Separators Examples:** ',' ';' etc

**Example of Non-Tokens:**

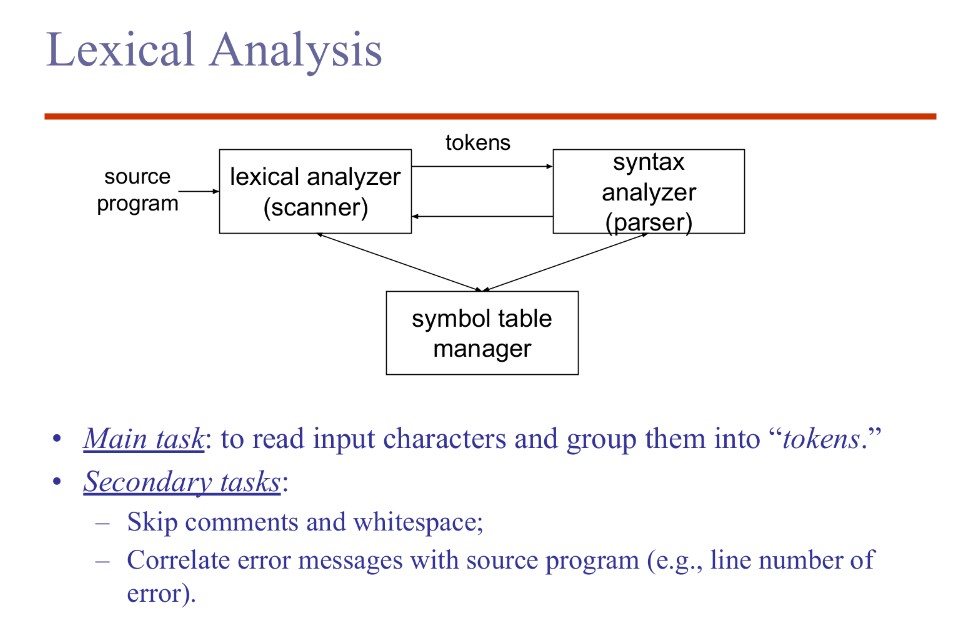
* Comments, preprocessor directive, macros, blanks, tabs, newline, etc.

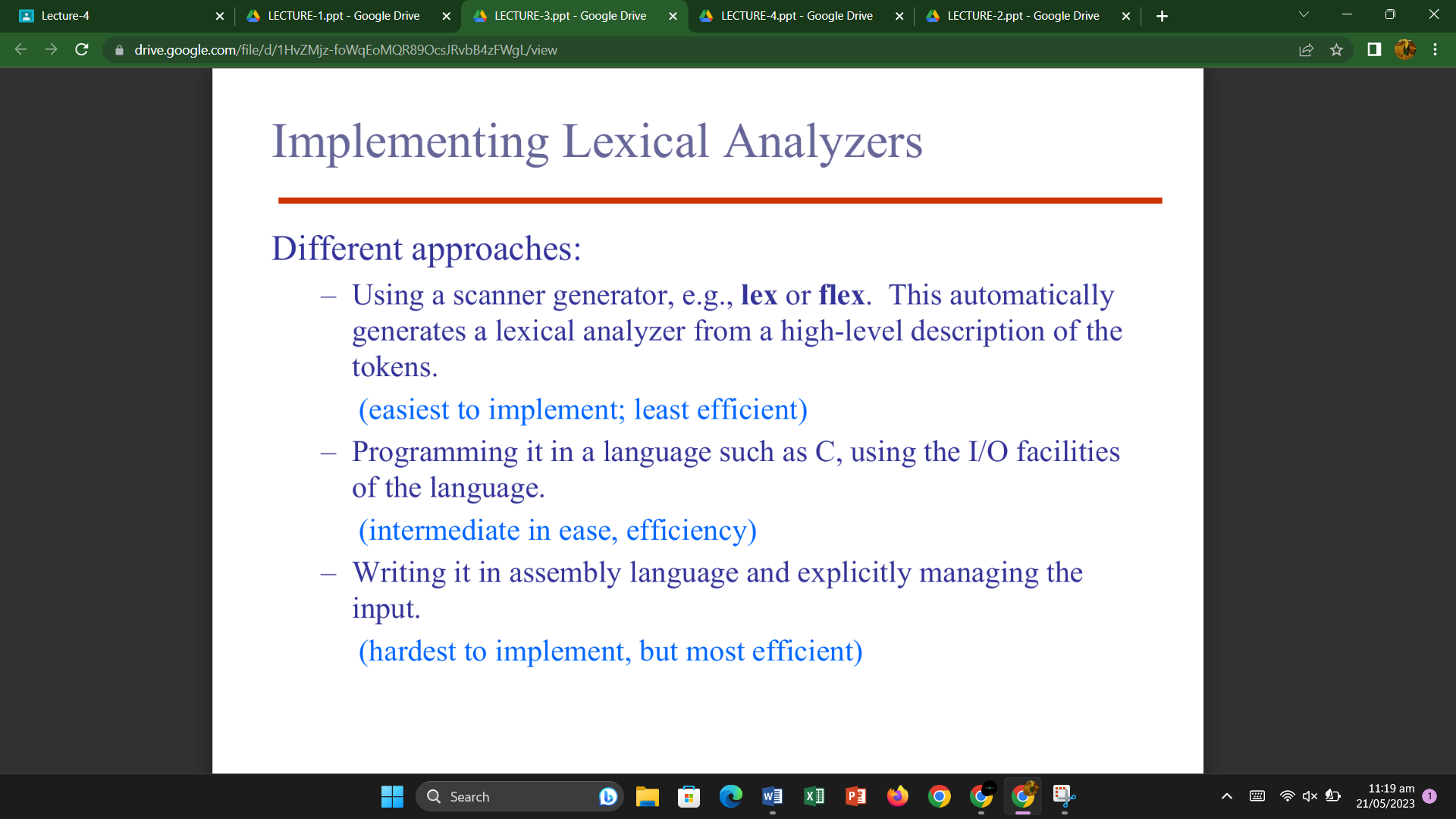
 In computer science, a program that executes the process of lexical analysis is called a **scanner**, tokenizer, or lexer.

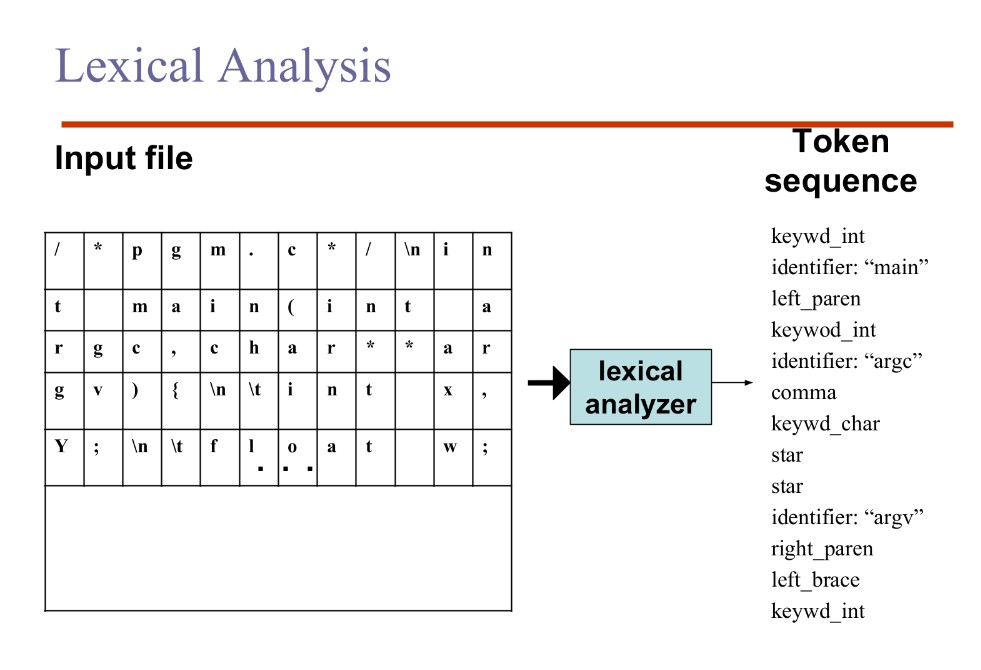
**pattern**: a rule describing the set of strings associated with a token.

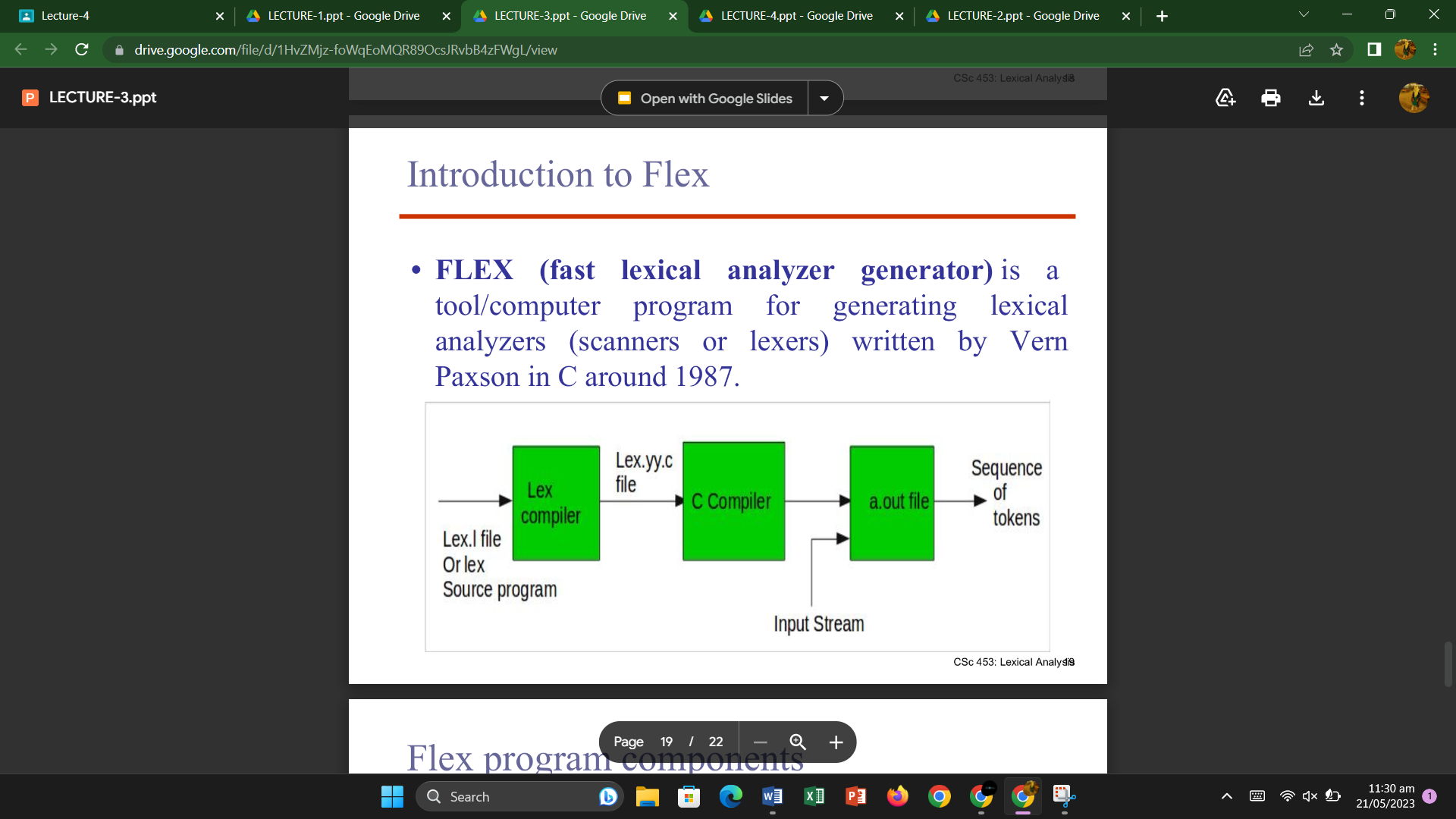
Example: “a letter followed by zero or more letters, digits, or underscores.”

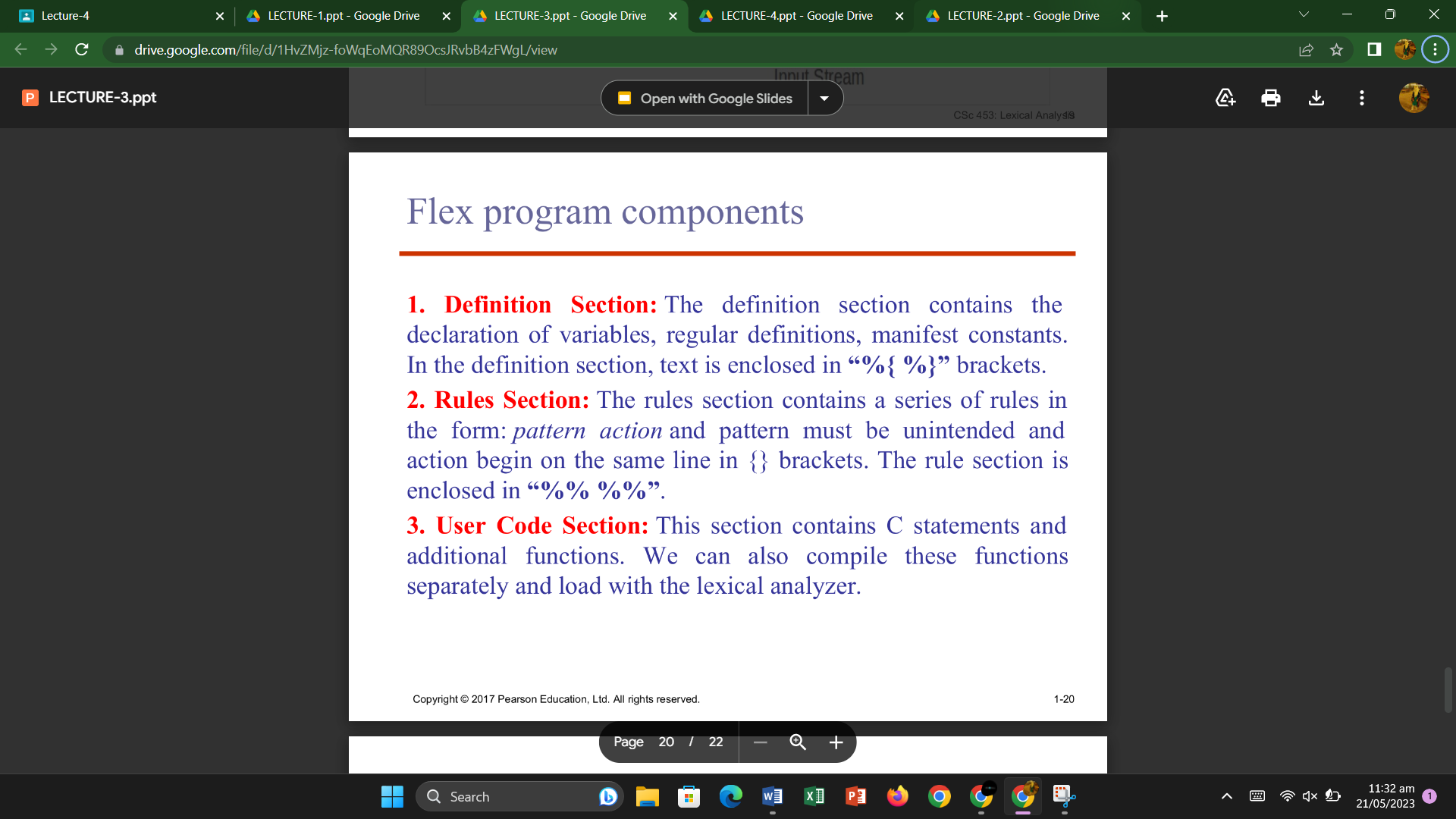
**Lexeme**: The sequence of characters matched by a **pattern** to form the corresponding **token** is called a **lexeme**. eg- “float”, “abs\_zero\_Kelvin”, “=”, “-”, “273”, “;” .

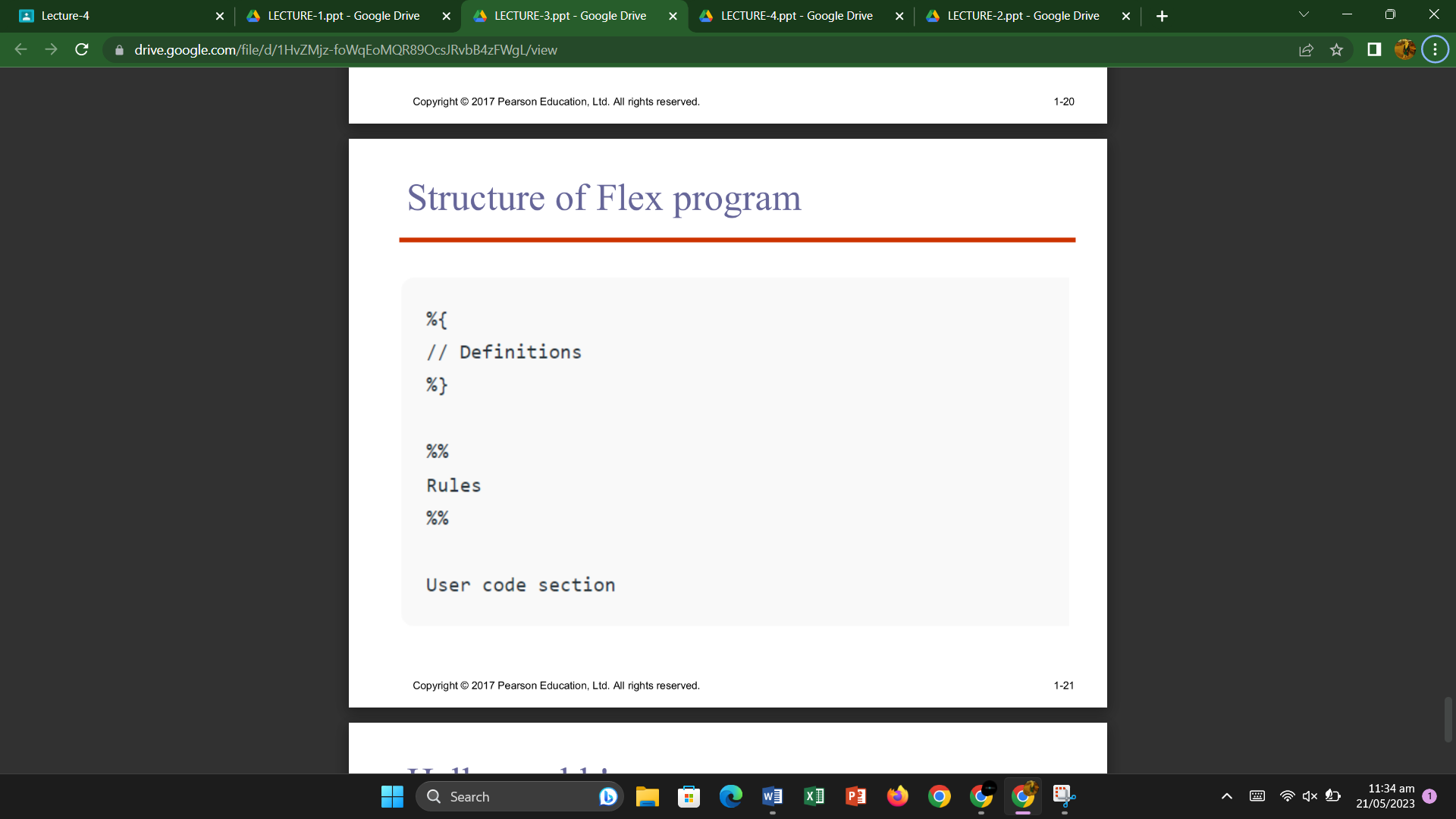


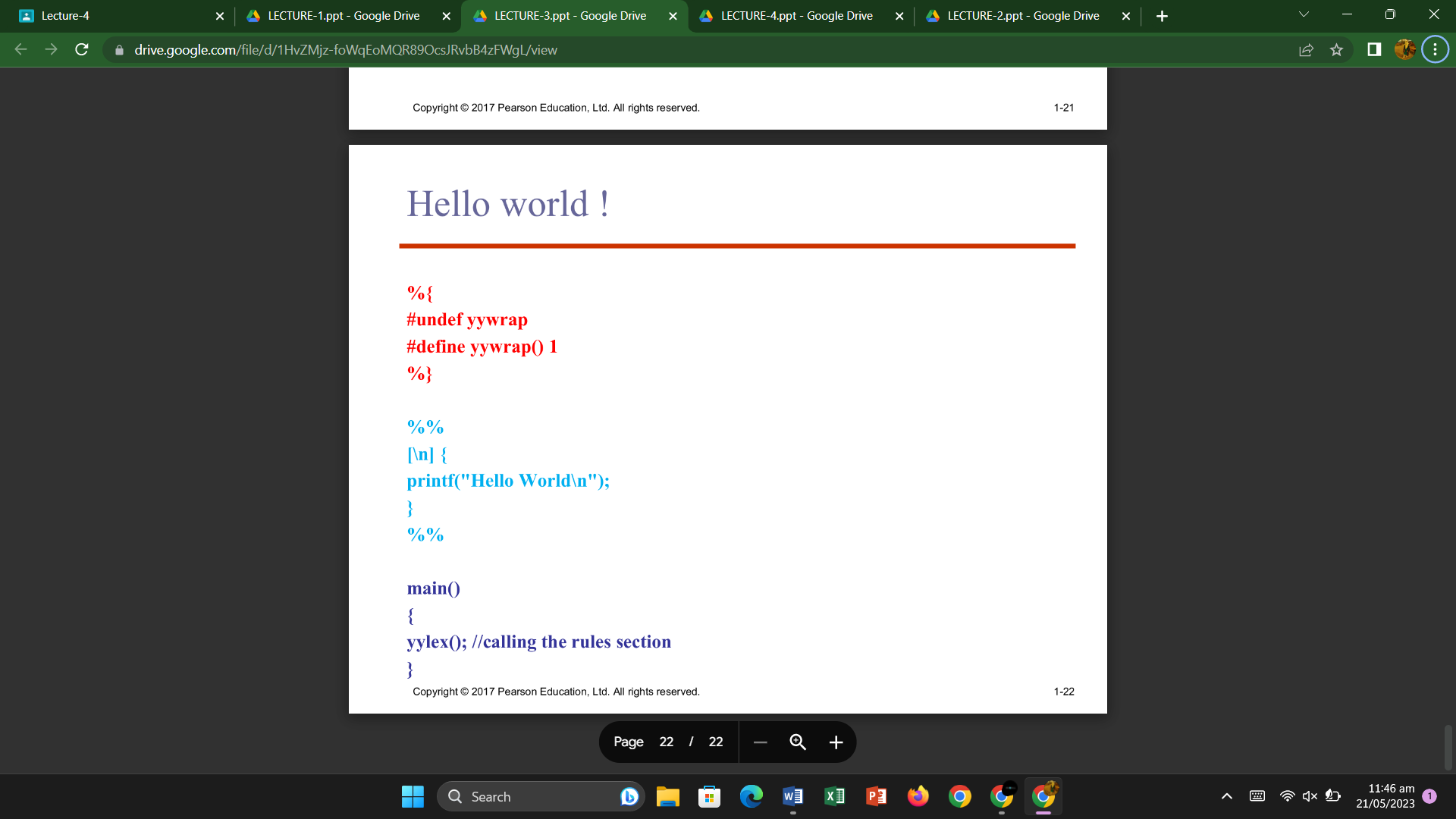








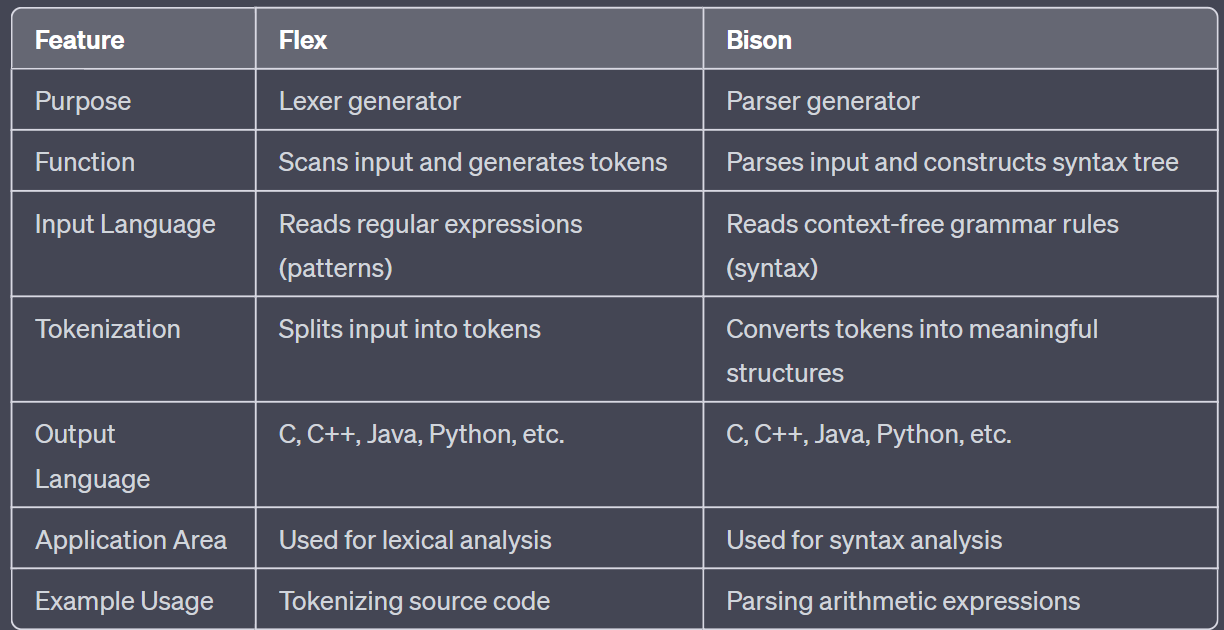




**ADVANTAGES:**

* 1. **Efficiency**: Lexical analysis improves the efficiency of the parsing process because it breaks down the input into smaller, more manageable chunks. This allows the parser to focus on the structure of the code, rather than the individual characters.
  2. **Flexibility**: Lexical analysis allows for the use of keywords and reserved words in programming languages. This makes it easier to create new programming languages and to modify existing ones.
  3. **Error Detection:** The lexical analyzer can detect errors such as misspelled words, missing semicolons, and undefined variables. This can save a lot of time in the debugging process.
  4. **Code Optimization:** Lexical analysis can help optimize code by identifying common patterns and replacing them with more efficient code. This can improve the performance of the program.

**Difference Between Flex and Bison**



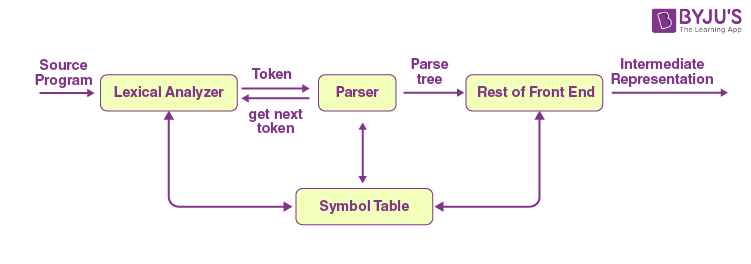
**2. Syntax Analysis:**

When an input string (source code or a program in some language) is given to a compiler, the compiler processes it in several phases, starting from lexical analysis (scans the input and divides it into tokens) to target code generation.

Syntax Analysis or Parsing is the second phase, i.e. after lexical analysis. It checks the syntactical structure of the given input, i.e. whether the given input is in the correct syntax (of the language in which the input has been written) or not. It does so by building a data structure, called a **Parse tree** or **Syntax tree**. The parse tree is constructed by using the pre-defined **Grammar** of the language and the input string. If the given input string can be produced with the help of the syntax tree (in the derivation process), the input string is found to be in the correct syntax. if not, the error is reported by the syntax analyzer.

Syntax analysis, also known as **parsing**, is a process in compiler design where the compiler checks if the source code follows the grammatical rules of the programming language.

The main goal of syntax analysis is to create **a parse tree** or **abstract syntax tree (AST**) of the source code, which is a hierarchical representation of the source code that reflects the grammatical structure of the program.



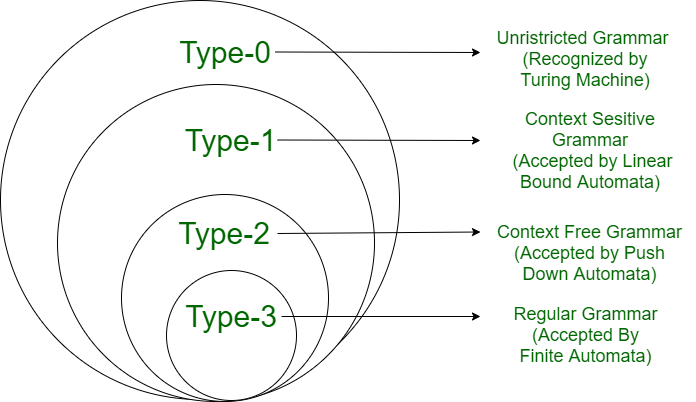
**ADVANTAGES:**

* **Structural validation:** Syntax analysis allows the compiler to check if the source code follows the grammatical rules of the programming language, which helps to detect and report errors in the source code.
* **Improved code generation:** Syntax analysis can generate a parse tree or abstract syntax tree (AST) of the source code, which can be used in the code generation phase of the compiler design to generate more efficient and optimized code.
* **Easier semantic analysis:** Once the parse tree or AST is constructed, the compiler can perform semantic analysis more easily, as it can rely on the structural information provided by the parse tree or AST.

**Chomsky Hierarchy in Theory of Computation**

According to Chomsky hierarchy, grammar is divided into 4 types as follows:

1. Type 0 is known as unrestricted grammar.
2. Type 1 is known as context-sensitive grammar.
3. Type 2 is known as a context-free grammar.
4. Type 3 Regular Grammar.



**Type 0: Unrestricted Grammar:**

* + Type 0 grammar languages are recognized by **turing** **machine**.
  + These languages are also known as the **Recursively Enumerable languages**.

**Type 1: Context-Sensitive Grammar:**

* Type-1 grammars generate **context-sensitive languages**.
* The language generated by the grammar is recognized by the **Linear Bound Automata.**

**Type 2: Context-Free Grammar:**

* Type-2 grammars generate **context-free languages**.
* The language generated by the grammar is recognized by a **Pushdown automata.**
* The left-hand side of production can have only one variable and there is no restriction on β.

S --> AB

A --> a

B --> b

**Type 3: Regular Grammar:**

* Type-3 grammars generate **regular languages**.
* The language generated by the grammar is recognized by a **Finite automata.**
* It is the most **restricted** form of grammar.

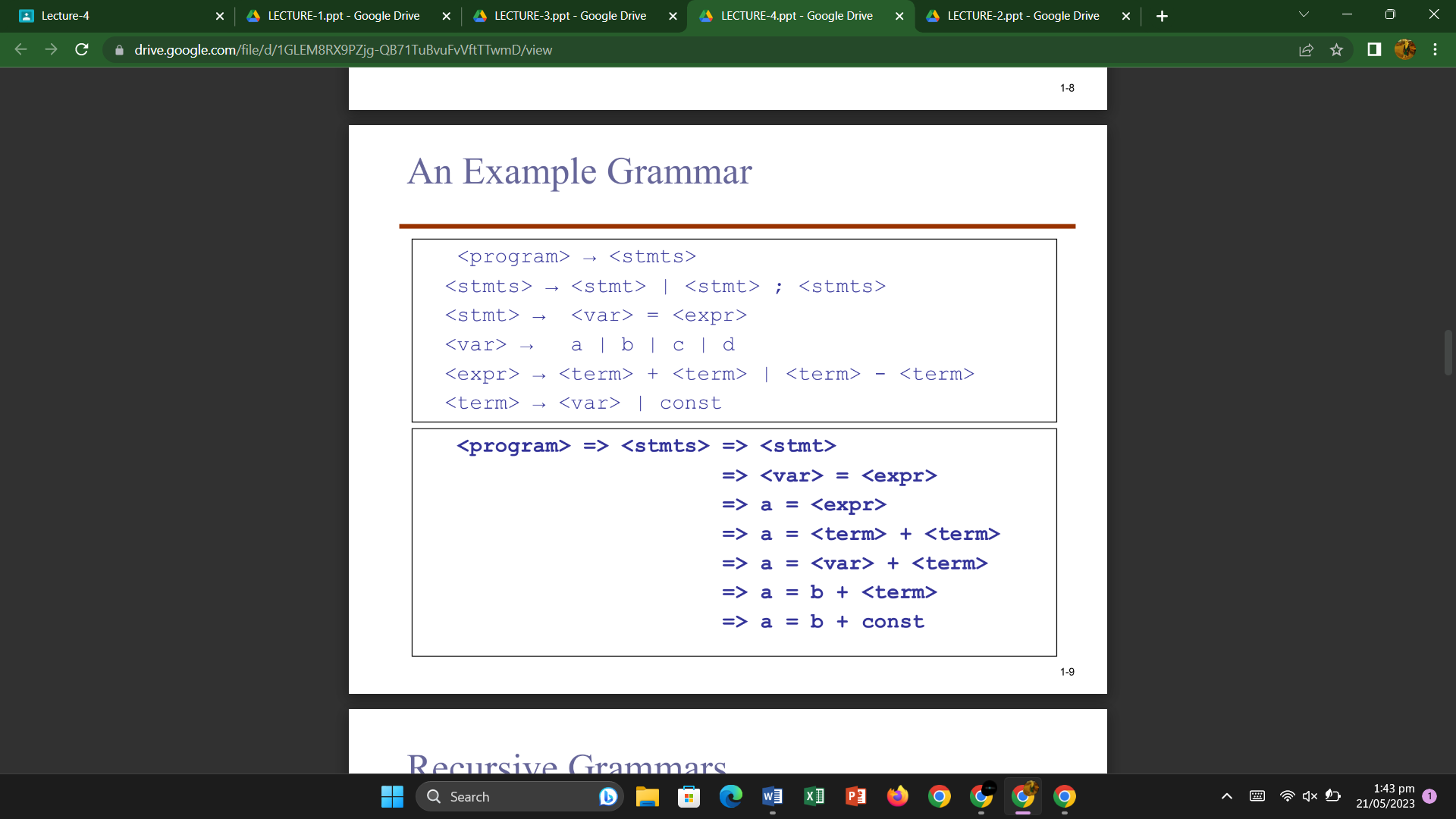
**Backus Naur Form** :

BNF stands for **Backus Naur Form** notation. It is a formal method for describing the **syntax** of programming language which is understood as Backus Naur Formas introduced by John Bakus and Peter Naur in 1960. BNF and CFG (Context Free Grammar) were nearly **identical**.

* Every name in Backus-Naur form is surrounded by angle brackets, < >, whether it appears on the left- or right-hand side of the rule.
* An expansion is an expression containing terminal symbols and non-terminal symbols, joined together by sequencing and selection.
* A vertical bar | indicates choice.
* Examples of BNF rules:

<ident\_list> → identifier | identifier, <ident\_list>

<if\_stmt> → if <logic\_expr> then <stmt>



**Extended BNF:**

Optional parts are placed in brackets [ ]

**<proc\_call> -> ident [(<expr\_list>)]**

Repetitions (0 or more) are placed inside braces {}

**<ident> → letter {letter|digit}**

1. **Recursive Grammar:**

A recursive grammar is a type of grammar where one or more production rules of a non-terminal symbol can reference that same non-terminal symbol on the right-hand side of the rule. In other words, the grammar allows a non-terminal symbol to appear on both sides of the production rules, creating the possibility of generating an infinite number of strings.

**Example**:

Consider a simple grammar to generate expressions using addition:

**Expression -> Number | Expression + Expression**

In this grammar, the non-terminal symbol "Expression" appears on both sides of the production rule for addition. This recursive definition allows the grammar to generate expressions of increasing complexity, such as "5 + 3," "(2 + 1) + (4 + 7)," "(8 + 9) + (6 + 3) + 10," and so on.

1. **Non-Recursive Grammar:**

A non-recursive grammar is a type of grammar where no production rule of a non-terminal symbol references that same non-terminal symbol on the right-hand side of the rule. In other words, there are no rules in the grammar that allow direct recursion back to the same non-terminal symbol.

**Example**:

Consider a simple grammar to generate simple arithmetic expressions:

**Expression -> Number | Expression + Number**

In this grammar, the non-terminal symbol "Expression" does not appear on the right-hand side of any production rule. As a result, the grammar is non-recursive and can generate expressions such as "5," "7 + 3," "2 + 6," but it cannot generate expressions with nested addition like "(4 + 8) + 3."

**Types of Recursive Grammars:**   
Based on the nature of the recursion in a recursive grammar, a recursive CFG can be again divided into the following:

* Left Recursive Grammar (having left Recursion)
* Right Recursive Grammar (having right Recursion)
* General Recursive Grammar(having general Recursion)

**Programming domains**

Programming domains deals with the areas & computer applications and their associated languages. Some of them are:

**1.** **Scientific applications:** The first digital computers, which appeared in the 1940’s, are invented for scientific applications. In the II world war, the US defense department wanted a computer that could easily calculate the values in their applications like trajectory of missiles.

**Characteristics**:

* Consists of simple data structures
* Can do large number of floating point arithmetic.

The main **languages** that were developed were FORTRAN, ALGOL 60.2.

**2. Business applications:** The use of computers for business applications began in 1950’s

**Characteristics**: The business people wanted a programming language that can:

* Produce elaborate reports.
* Have precise way of describing and sorting decimal numbers and character data.
* The ability to specify decimal arithmetic operations.

COBOL (Common business oriented language) was the first high level **language** for business.

**3. Artificial intelligence**: In the years of 1960, artificial intelligence played a very important role.

**Characteristics**: AI is a broad area of computer applications characterized by

* use of symbolic rather than numeric computation i.e. it must contain symbols, names
* more flexible language for symbolic computation
* The ability to create and execute some code segments during execution.

The main **languages** that were developed during this time were the LISP and PROLOG. These were mainly used for logic programming like designing moves of chess.

**4. Systems Programming:** In the 1960’s and 1970’s system software was used. The OS & all of the programming support tools of a computer system are collectively known as its system software. System software is almost used continuously.

**Characteristics:** The system software must have the following:

* As it is continuously used it must be efficient.
* It must have low level features that allow the s/w interface to external devices to be written
* It must provide security

**Developed Languages**

I) PL/S, a dialect of PL/I, initially called BSL was developed by IBM mainframes.

II) For digital systems, BLISS was developed

III) For Burroughs extended ALGOL

IV) Unix OS was written almost in C But it is too dangerous to use C on large important Systems.

**5. Scripting Languages:** These languages have evolved over the last 25 years.

Characteristics:

* A scripting language is used by putting a list of commands, called a SCRIPT in a title to be executed.
* For simple applications like file management and file filtering

**Languages Developed**

I) SH is the first scripting language.

II) Ksh, developed David Kon

III) AWK developed by AL ATO

IV) TCL developed by John Outerhost

V) PERL

1. **Web Software:**

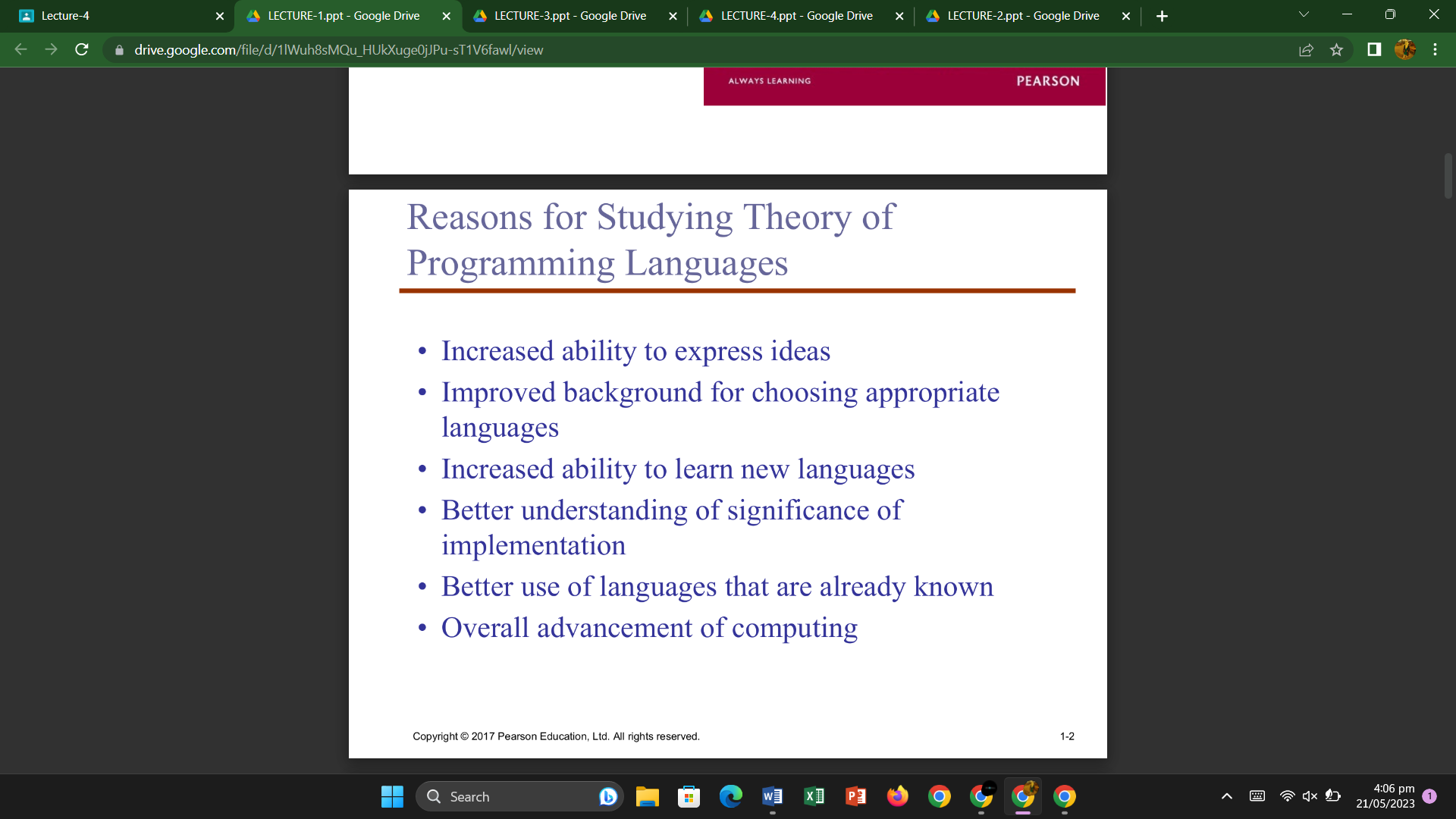
The emergence of the World Wide Web in the **1990s** revolutionized the way information is accessed and shared. Web software refers to programming languages and frameworks used for creating websites, web applications, and web services.

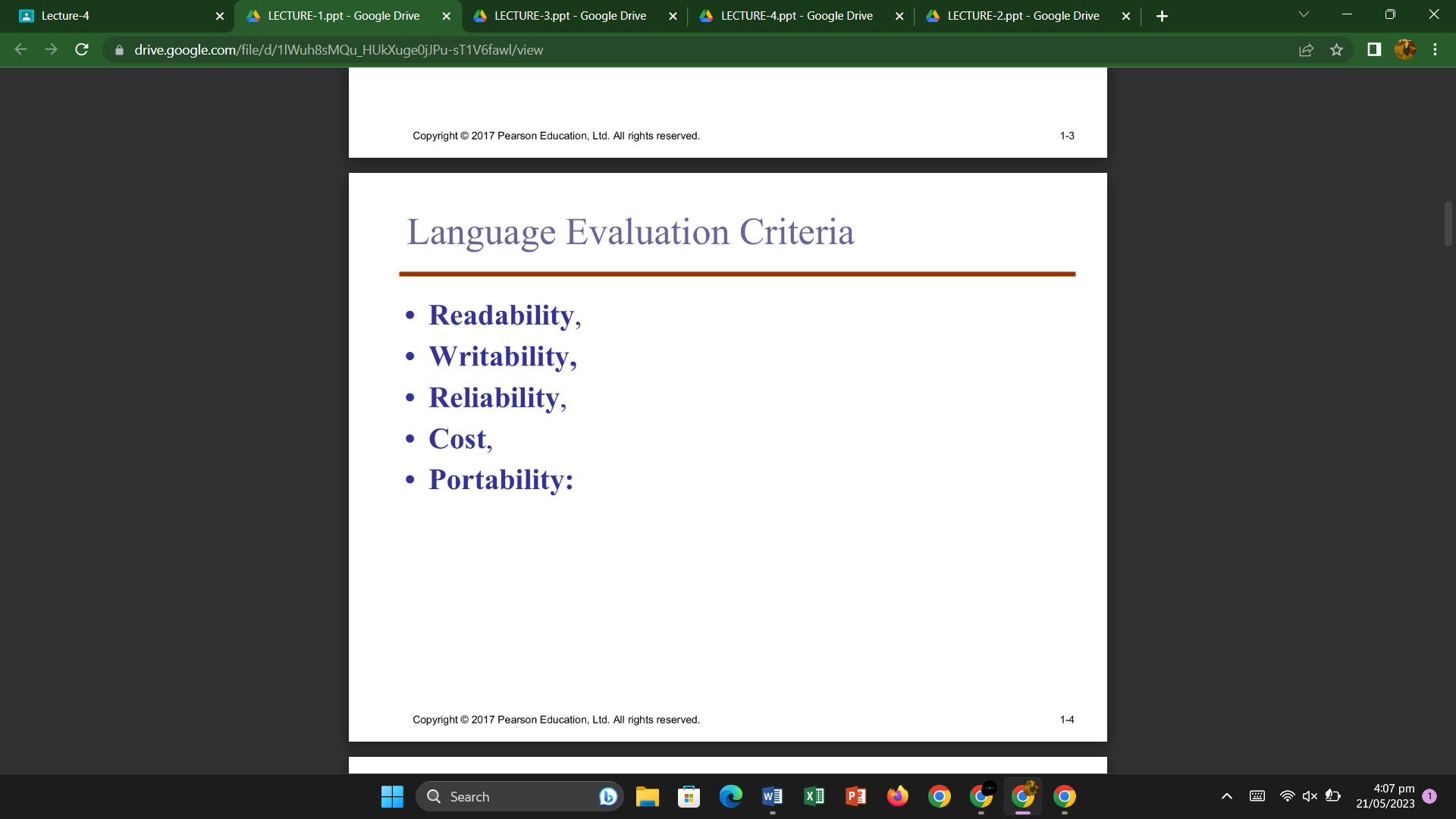
**Characteristics:**

* Client-Server Architecture
* Client-Side Scripting:
* Server-Side Scripting:
* Web Frameworks:
* Database Integration:
* Web Services and APIs:
* **Security Considerations:**

Languages and Technologies:

* HTML
* CSS
* JavaScript
* PHP
* Python (with frameworks like Django and Flask)
* Ruby (with frameworks like Ruby on Rails)
* Java (with frameworks like Spring)
* ASP.NET
* Express.js
* Laravel
* MySQL
* PostgreSQL
* MongoDB
* SQLite





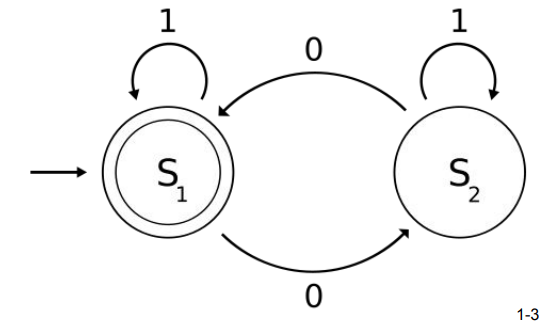
**Model of Computation**

A model of computation is a model which describes how an output of a mathematical function is computed given an input.

A model describes how units of computations, memories, and communications are organized

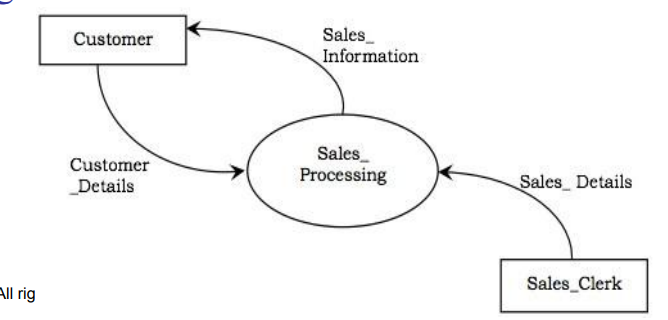
**Categories:**

1. **Sequential Model**: A sequential computational model is one in which instructions are executed one after another.

There may be branches in the program, but the general principle is that each instruction follows on from the previous one.

Sequential models are:

* FSM
* PDA
* TM

1. **Functional model:** A function model or functional model is a structured representation of the functions (activities, actions, processes, operations) within the modeled system or subject area. Functional models are:

* Lambda calculus
* Abstract rewriting systems

1. **Concurrent model:** Concurrent computing describes a process where multiple programs or threads are managed by one computing system through the process of alternating timeshared slices.

Concurrent models are:

* Actor model
* Cellular automaton

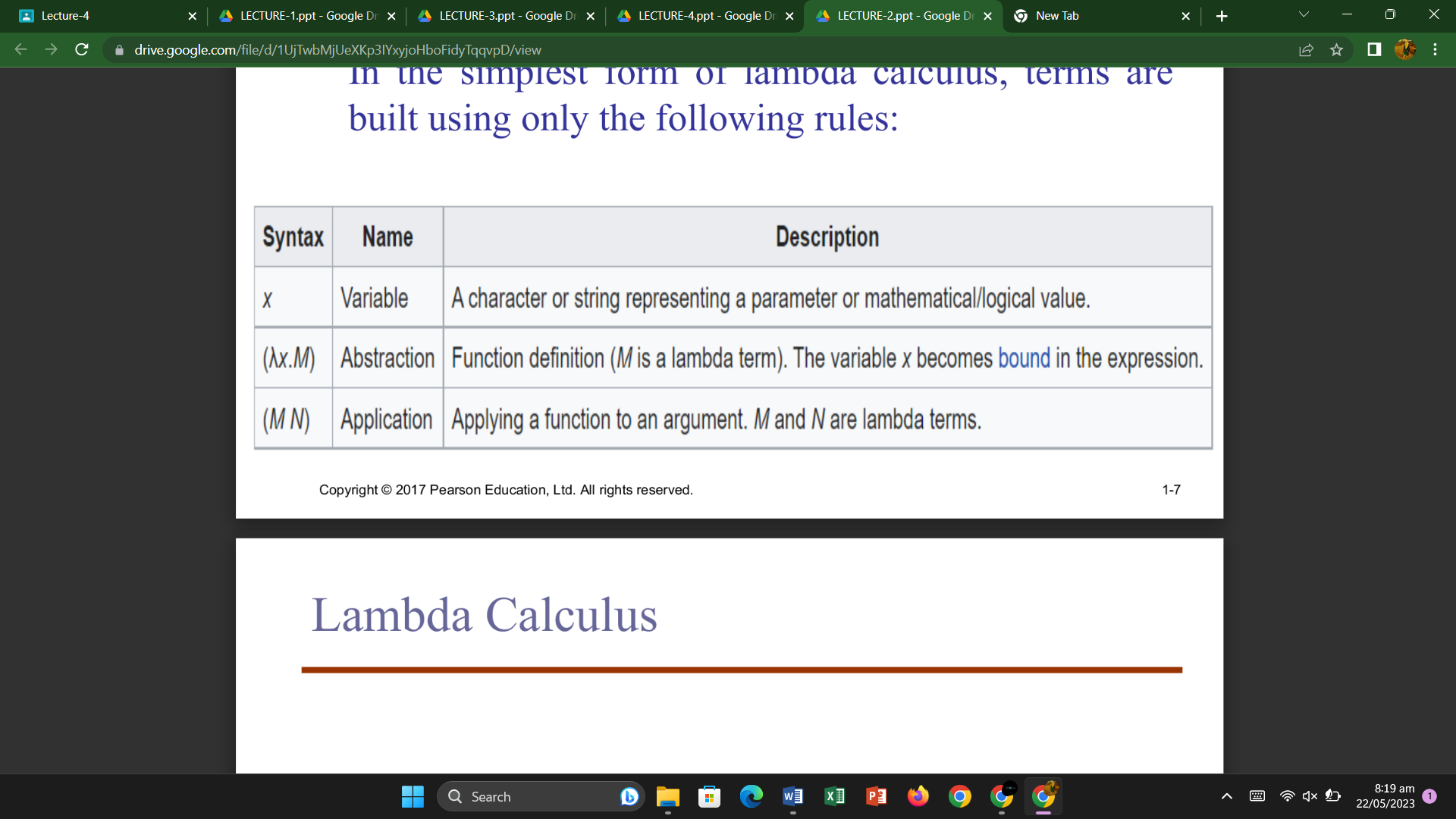
**Lambda Calculus:**

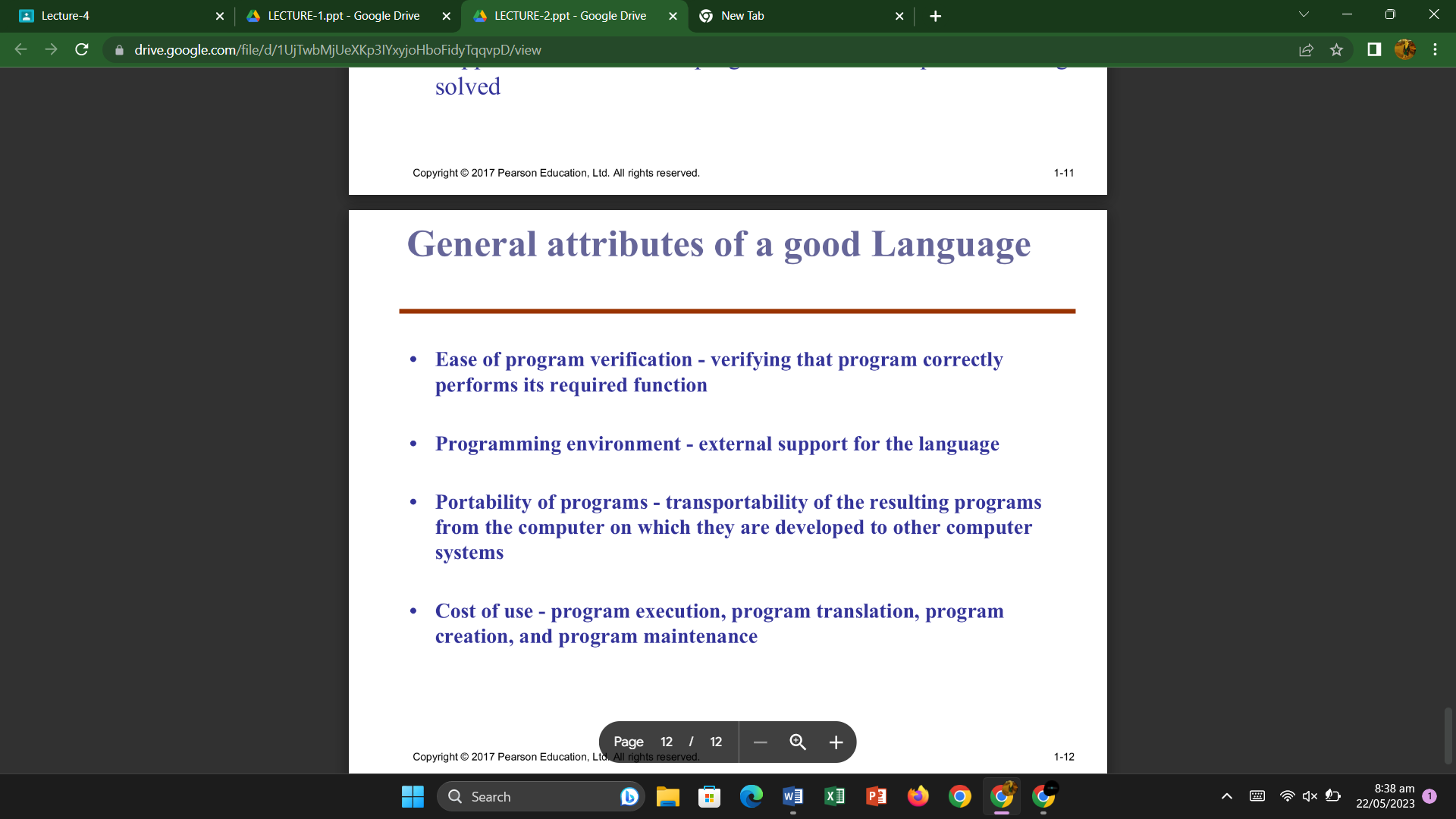
* Lambda calculus is a formal system in mathematical logic for expressing computation based on function abstraction and application using variable binding and substitution.
* It is a universal model of computation that can be used to simulate any Turing machine. It was introduced by the mathematician Alonzo Church in the 1930s as part of his research into the foundations of mathematics.
* Lambda calculus consists of constructing lambda terms and performing reduction operations on them.

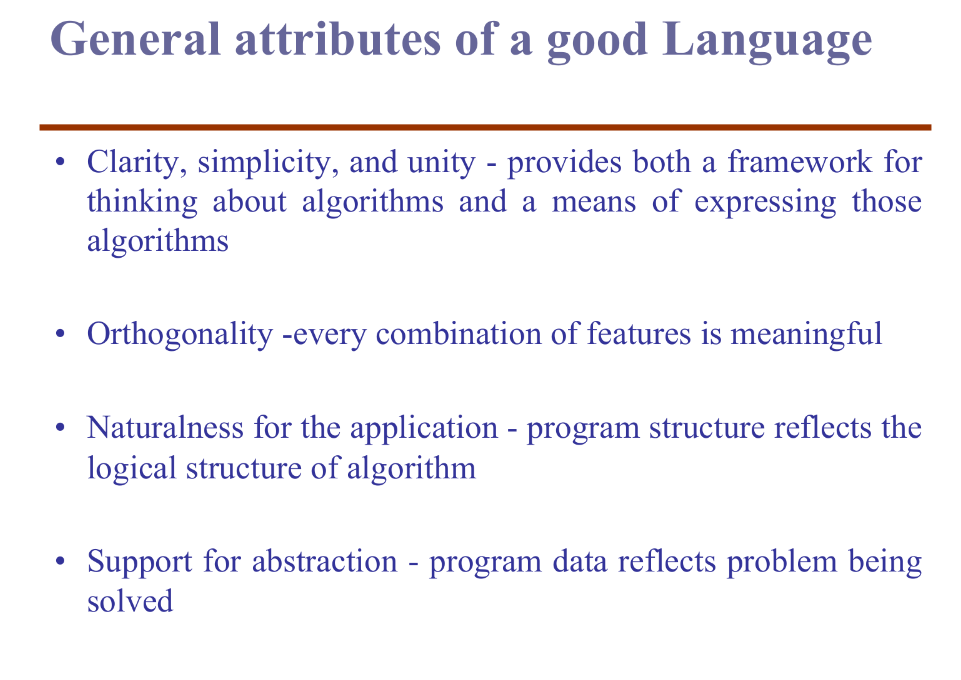
Lambda calculus provides two fundamental operations:

1. **Abstraction**: The process of creating a lambda expression to represent a function.
2. **Application**: The process of applying arguments to a lambda expression to evaluate the result.

* In the simplest form of lambda calculus, terms are built using only the following rules:







**Influences on Language Design**

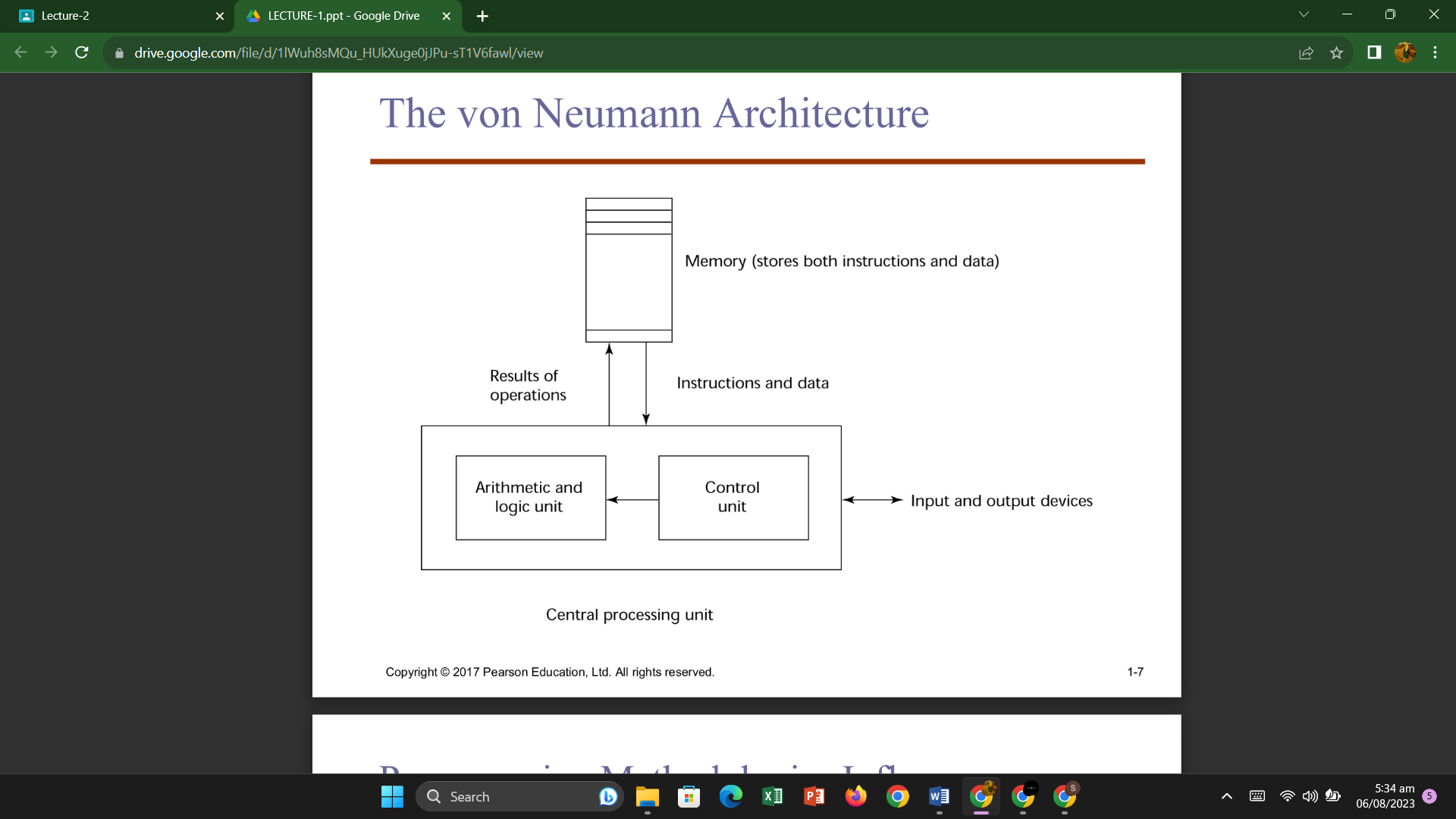
Computer architecture and program design methodologies have played crucial roles in shaping the development of programming languages. Let's explore each influence and provide examples to better understand their impact on language design.

1. **Influence of Computer Architecture:**

Computer architecture refers to how computers are designed and how they work internally. The most common architecture is the von Neumann architecture. It influenced the design of imperative programming languages.

**Example:**

In languages like C and Pascal, variables are used to represent memory cells, and assignment statements are used to move data between memory and the CPU. This closely aligns with the von Neumann architecture, where data is stored in memory and processed by the CPU.



1. **Influence of Program Design Methodologies:**

Different approaches to designing software, called program design methodologies, have led to the creation of new programming languages. Over time, these methodologies evolved, giving rise to different programming paradigms.

**Examples**:

**a.** **Structured Programming**: In the late 1960s, structured programming emphasized writing code with clear control structures to improve readability and maintainability. Languages like Pascal were created to support this approach.

**b.** **Object-Oriented Programming (OOP):** In the mid-1980s, OOP became popular, focusing on data abstraction, inheritance, and polymorphism. Languages like C++, Java, and Python were developed to support this paradigm.

In summary, computer architecture influenced the design of imperative languages, which dominate the programming landscape. Program design methodologies influenced the creation of languages that support different paradigms like structured programming and object-oriented programming. Understanding these influences helps programmers choose the right language for specific tasks and contributes to the evolution of programming languages over time.

**Implementation methods**

* 1. **Compilation**:

Compilation is the process of converting human-readable source code written in a high-level programming language into machine-readable code, typically in the form of binary instructions or bytecode. The result of compilation is an executable file that can be directly run by the computer's hardware.

During compilation, the C compiler (e.g., GCC) translates this source code into machine code that the computer's CPU can understand, producing an executable file (e.g., "hello.exe" on Windows or "hello" on Linux). When you run this executable, it directly executes the machine code, displaying "Hello, World!" on the screen.

1. **Interpretation**:

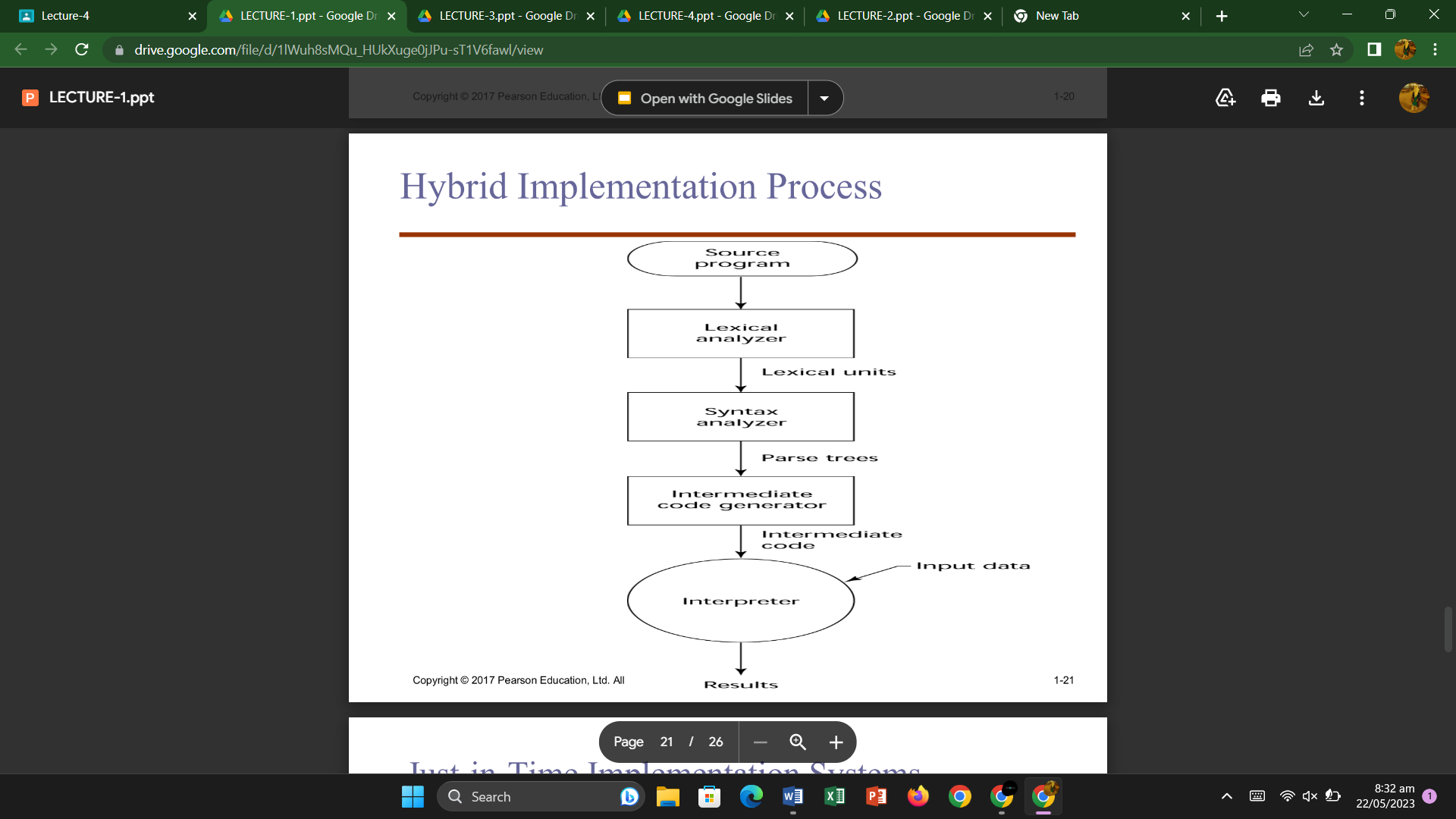
Interpretation is the process of executing the source code directly without prior conversion to machine code. The interpreter reads the source code line-by-line and executes it on-the-fly, translating each instruction into machine code as it encounters them.

In an interpreted environment, the Python interpreter reads and executes each line of the code directly, without creating a separate executable. When you run the Python script, the interpreter interprets the code and prints "Hello, World!" to the screen.

1. **Hybrid Implementation System:**

A hybrid implementation system combines elements of both compilation and interpretation. It first translates the source code into an intermediate form (bytecode) during the compilation phase. Then, an interpreter executes the bytecode on the target machine.

**Example**: Java uses a hybrid implementation system.



**Semantic Analysis**

Semantic Analysis is an essential phase in the compilation process, which comes after the Syntax Analysis (parsing) phase. Its main job is to ensure that the declarations and statements in the program make sense and are logically correct.

One of its important tasks is type checking. It makes sure that the operations and calculations in the program are done using the right types of data. For example, it will catch if you try to add a string to a number or use a variable that hasn't been declared before.

If everything is fine, the program is considered "semantically correct" and moves on to the next phase. If any issues are found, the compiler will report an error and ask the programmer to fix it before proceeding.Top of Form

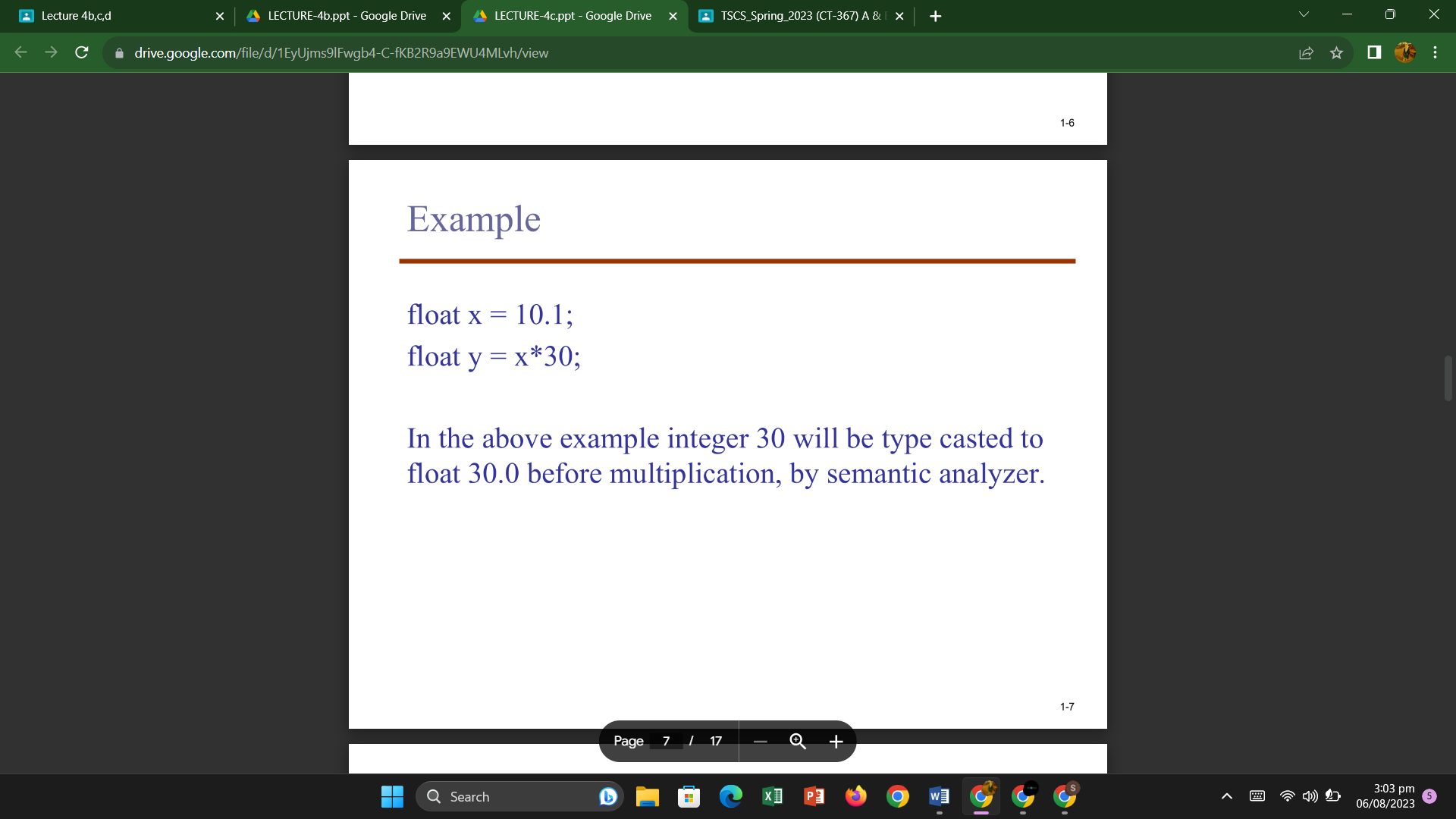
**Semantic Analyzer:**  
It uses syntax tree and symbol table to check whether the given program is semantically consistent with language definition. It gathers type information and stores it in either syntax tree or symbol table. This type information is subsequently used by compiler during intermediate-code generation.

**Semantic Errors:**  
Errors recognized by semantic analyzer are as follows:

* Type mismatch
* Undeclared variables
* Reserved identifier misuse

**Functions of Semantic Analysis:**

1. **Type Checking –**  
   Ensures that data types are used in a way consistent with their definition.
2. **Label Checking –**  
   A program should contain labels references.
3. **Flow Control Check –**  
   Keeps a check that control structures are used in a proper manner.(example: no break statement outside a loop)



**Attribute Grammar**

Attribute grammar is a special form of context-free grammar where some additional information (attributes) are appended to one or more of its non-terminals in order to provide context-sensitive information.

Attribute grammar is a medium to provide semantics to the context-free grammar and it can help specify the syntax and semantics of a programming language.

**Example:**

**E → E + T { E.value = E.value + T.value }**

**Static Semantic:**

Static semantics refers to the aspects of a programming language's semantics that can be checked at compile-time without the need for program execution.

Static semantic checks are performed during the compilation process, before the program is executed. These checks include:

1. Type Checking
2. Scope Checking
3. Declaration Checking

**Dynamic Semantic:**

Dynamic semantics refers to the aspects of a programming language's semantics that are related to the behavior and meaning of programs during program execution. It deals with the runtime behavior of the program, considering actual data and control flow.

Dynamic semantic checks are performed during the execution of the program and include:

1. Runtime Type Checking
2. Memory Management
3. Exception Handling

**Operational semantics:**

Operational semantics is a formal and mathematical way of describing the behavior and meaning of programming languages. It provides a step-by-step operational description of how the programs or expressions within the language are executed or evaluated.

To use operational semantics for a high-level language, a virtual machine is needed.

Two different levels of uses of operational semantics:

- Natural operational semantics

- Structural operational semantics

**Denotational Semantics:**

Denotational semantics is a formal approach used to describe the meaning of programming languages. Denotational semantics is a formal method to describe the meaning of programming languages using mathematical structures, making it easier to analyze and reason about the behavior of programs in a clear and consistent manner.

The benefits of denotational semantics include providing a rigorous and precise way to reason about the behavior of programs, enabling proofs of correctness and helping to understand the underlying principles of programming languages.

**Axiomatic Semantics:**

Instead of focusing on how programs are executed or what they mean, axiomatic semantics concentrates on specifying the properties and relationships between different parts of a program.

In simple terms, axiomatic semantics is like creating a set of rules or laws that describe how a program behaves under certain conditions. These rules are expressed using mathematical logic and help to prove the correctness of a program, i.e., whether the program fulfills its intended purpose and satisfies specific requirements

**Designing Problems**

* 1. **Names:**
* Length
* Case Sensitive
* Connotative
* Special Characters
* Reserved Word
  1. **Variables:**

A variable is an abstraction of a memory cell.

Six tuple of attributes:

* Name
* Address
* Value
* Type
* Lifetime
* Scope

The l-value of a variable is its address

The r-value of a variable is its value

* 1. **Binding:**

A binding is an association between an entity and an attribute, such as between a variable and its type or value.

**Binding time** is the time at which a binding takes place.

* **Language design time** -- bind operator symbols to operations
* **Language implementation time**-- bind floating point type to a representation
* **Compile time** -- bind a variable to a type in C or Java
* **Load time** -- bind a C or C++ static variable (to a memory cell)
* **Runtime** -- bind a nonstatic local variable to a memory cell

**Static Binding:**

A binding is static if it first occurs **before run time** and remains **unchanged** throughout program execution.

**Dynamic Binding:**

A binding is dynamic if it first occurs **during execution** or can **change** during execution of the program.

**Explicit declaration:**

An explicit declaration is a **program statement** used for declaring the types of variables

**Implicit declaration:**

An implicit declaration is **a default mechanism** for specifying types of variables through default conventions, rather than declaration statements.

Basic, Perl, Ruby, JavaScript, and PHP provide implicit declarations.

**Variables (Storage Binding and Lifetime)**

1. **Storage Bindings:**

* **Allocation**: It refers to obtaining a memory cell from a pool of available cells to store the value of a variable.
* **Deallocation**: It means putting a memory cell back into the pool when it is no longer needed.

1. **Categories of Variables by Lifetimes:**

**Lifetime of a variable**: It is the duration during which a variable is associated with a specific memory cell.

**a) Static Variables:**

* Bound to memory cells before program execution starts.
* Remains bound to the same memory cell throughout program execution.
* Found in languages like C and C++ as static variables in functions.
* **Advantages**: Efficient
* **Disadvantage**: Lack of flexibility

**b) Stack-Dynamic Variables:**

* Storage bindings are created when the declaration statements are executed.
* If scalar, all attributes except the address are statically bound.
* Examples include local variables in C subprograms and Java methods.
* **Advantages**: Allows recursion
* **Disadvantages**: Overhead of allocation and deallocation,

**c) Explicit Heap-Dynamic Variables:**

* Allocated and deallocated by explicit directives specified by the programmer during execution.
* Referenced only through pointers
* Examples include dynamic objects in C++ and all objects in Java.
* **Advantage**: Provides dynamic storage management.
* **Disadvantage**: Can be inefficient and unreliable.

**d) Implicit Heap-Dynamic Variables:**

* Allocation and deallocation occur due to assignment statements.
* Examples include all variables in a programming language, and all strings and arrays in Perl, JavaScript, and PHP.
* **Advantage**: Offers flexibility (allows generic code).
* **Disadvantages**: Inefficient and loss of error detection.

**Type Checking**

Type checking is a crucial process in programming languages that ensures the compatibility and correctness of data types used in expressions, assignments, and function calls. There are generally three types of type checking:

1. **Static Type Checking**: Static type checking is performed **at compile time,** before the program is executed. The compiler examines the code and verifies that the data types used in various expressions and operations are consistent and well-defined. If any type errors are found during static type checking, the compiler reports them as compilation errors, and the program won't be executed until these errors are fixed.

**Advantages:**

* Early detection of type-related errors, leading to safer and more reliable code.
* Improved performance as the type information is known during compilation.

**Disadvantages**:

* It can be more rigid and restrictive, requiring explicit type declarations and conversions.

Languages with static type checking include C, C++, Java, and Swift.

1. **Dynamic Type Checking**: Dynamic type checking is performed at **runtime**, while the **program is running.** The interpreter or runtime environment verifies the data types of variables and objects during execution. If there is a type mismatch or an operation is attempted on incompatible types, an error, such as a **runtime exception,** is generated.

**Advantages:**

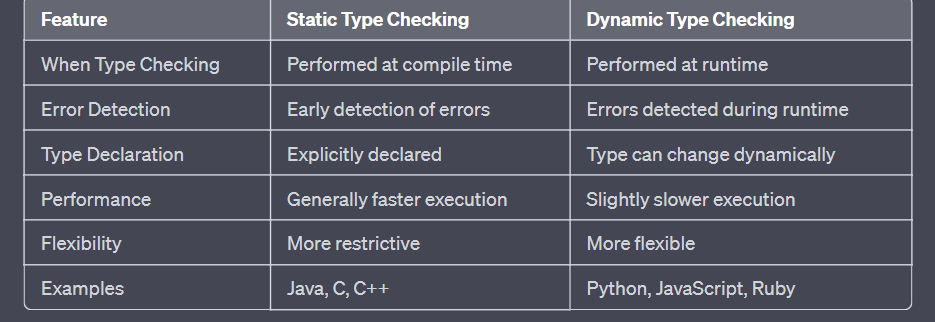
* Allows more **flexibility**, as types can be determined at runtime.
* **Less need for explicit type declarations**, making code writing quicker and easier.

**Disadvantages:**

* Type errors may occur during program execution, leading to potential runtime crashes or unexpected behavior.
* Slightly reduced performance due to the overhead of runtime type checks.

Languages with dynamic type checking include Python, JavaScript, and Ruby.

**Diff b/w Static and Dynamic Type Checking:**



**Operand Evaluation Order**

Operand evaluation is the process of finding out the values of the individual things (operands) that are part of a math or programming expression before actually doing the math or operation. For example, in the expression "3 + 5," operand evaluation means figuring out what "3" and "5" are before adding them to get the final result "8."

1. **Variables**: When an expression involves variables, their values are fetched from the computer's memory. Variables are like containers that hold data, and their values can change during the program's execution. To evaluate an expression, the program looks up the current value of each variable in memory and uses it in the calculation.

**For example**, consider the expression "x + y." If x holds the value 5 and y holds the value 3, the program fetches these values from memory and adds them together to get the result 8.

1. **Constants**: Constants are fixed values that do not change during the program's execution. They can be directly embedded in the machine language instruction or stored in memory and fetched when needed. In some cases, the value of a constant may be directly available in the instruction itself, so no memory access is required.

**For example,** in the expression "5 + 3," both 5 and 3 are constants. If these values are directly available in the instruction, the program can use them directly for the addition without the need to fetch them from memory.

1. **Parenthesized expressions**: Parentheses in an expression indicate that the contents inside the parentheses must be evaluated first. When encountering a parenthesized expression, the program evaluates all the operands and operators inside the parentheses before moving on to the rest of the expression.

**For example**, consider the expression "(2 + 3) \* 4." The program first evaluates "2 + 3" inside the parentheses, resulting in 5. Then, it multiplies the result 5 by 4 to get the final result 20.

1. **Function calls**: When an operand in an expression is a function call, the program needs to execute the function and obtain its return value before proceeding with the evaluation. Functions are blocks of code that perform specific tasks and may take parameters as inputs. The result of the function's execution becomes the value of the operand in the expression.

**For example**, consider the expression "sqrt(9) + 5." Here, "sqrt" is a function that calculates the square root of its input. The program calls the "sqrt" function with the value 9 as the parameter, and the function returns the result 3. The program then adds 3 to 5 to get the final result 8.

**Side Effects**

In programming, a side effect occurs when a function changes one of its parameters or a global variable. It can cause unexpected behavior in expressions because the order in which operands are evaluated matters.

Let's consider an example expression: "a + fun(a)". If the function "fun" doesn't change the value of "a," the order of evaluating "a" and "fun(a)" doesn't affect the final result. However, if "fun" changes "a," the order of evaluation matters.

For instance, if "fun" returns 10 and changes "a" to 20, and we have "a = 10; b = a + fun(a);", then depending on the order of evaluation, the value of "b" can be either 20 or 30.

To solve this, languages can either disallow functions from changing values or guarantee a specific evaluation order for operands. Disallowing changes in functions can be limiting and less flexible for programmers.

On the other hand, specifying a fixed evaluation order may prevent certain optimizations and can lead to other complexities.

Languages like Java choose to guarantee a left-to-right operand evaluation order, which helps avoid the problem caused by side effects in expressions.

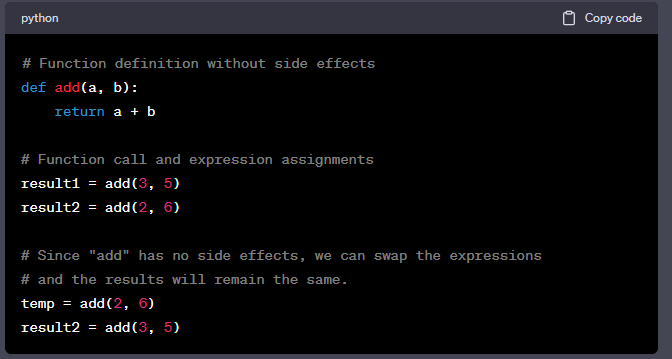
**Referential Transparency and Side Effects**  
Referential transparency is a property of programs where any two expressions that have the same value can be replaced with each other without changing how the program works. In simpler terms, if you have two expressions that give you the same result, you can swap one with the other without any problems in the program.

Referential transparency is closely related to functional side effects. Functional side effects occur when a function changes something outside of itself, like modifying a variable or a global value. When a function has side effects, it can affect the program's behavior and make it harder to reason about.

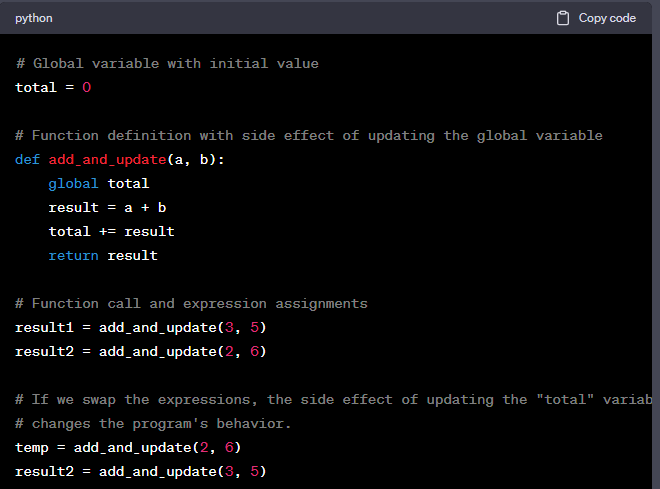
In the example, if a function "fun" has no side effects, then expressions "result1" and "result2" will be equal because they are equivalent. However, if "fun" has side effects, the two expressions may not be equal, and this violates the referential transparency of the program.

Having referential transparency in programs is beneficial because it makes the program's behavior easier to understand and reason about, similar to mathematical functions. In pure functional languages, where there are no variables or side effects, functions are automatically referentially transparent since their output only depends on their inputs (parameters).

Example without functional side effects (referentially transparent):



Example with functional side effects (not referentially transparent):



**Type Conversion**

Type conversions are operations in programming where a value of one data type is converted into another data type. There are two main types of type conversions:

1. **Narrowing Conversion**: This type of conversion occurs when a value is converted to a data type that cannot hold all the values of the original data type. For example, converting a floating-point number (like 3.14) to an integer (like 3) is a narrowing conversion because the fractional part is lost.
2. **Widening Conversion**: This type of conversion occurs when a value is converted to a data type that can hold at least approximations of all the values of the original data type. For example, converting an integer (like 5) to a floating-point number (like 5.0) is a widening conversion because all integer values can be represented as floating-point values.

In programming, sometimes expressions involve operands of different data types. This is known as a **mixed-mode expression**. To make such expressions work, some programming languages use implicit type conversions called ***coercions***. Coercions automatically convert one or more operands to a common data type so that the expression can be evaluated. However, coercions can be disadvantageous as they may hide potential type errors in the program.

Some programming languages, like ML and F#, do not have coercions in expressions, meaning they strictly enforce type compatibility, and **explicit type conversions** must be used when necessary. In C-based languages, explicit type conversions are done using casting. For example, in C, you can use **(int)angle** to explicitly convert the variable **angle** to an integer.

Explicit type conversions give more control to the programmer and help ensure that the correct data types are used in expressions, avoiding potential errors caused by automatic coercions.

**Short-circuit**

Short-circuit evaluation is a concept in programming where the result of an expression is determined without evaluating all of its parts. It's like taking a shortcut to save time and resources.

In arithmetic expressions, short-circuit evaluation doesn't really happen because all parts of the expression are always evaluated, even if some parts don't affect the final result.

But in Boolean expressions (expressions that produce either "true" or "false"), short-circuit evaluation can be useful. If the result of a Boolean expression can be determined early based on the first part of the expression, the rest of the expression doesn't need to be evaluated. This saves unnecessary computation and can help prevent errors.

For example, if we have the expression **(a >= 0) && (b < 10)**, and **a** is less than zero, we already know the whole expression will be "false" regardless of the value of **b**. So there's no need to evaluate the second part **(b < 10)**.

Languages like **Ruby**, **Perl**, **ML**, **F#**, and **Python** use short-circuit evaluation for **all their logical operators**, making expressions more efficient. However, C-based languages like **C, C++, and Java** use short-circuit evaluation for their **logical AND (&&) and OR (||) operators**, but not for their bitwise AND (**&**) and OR (**|**) operators.

Using short-circuit evaluation can be helpful, but it's essential to be careful with expressions that have side effects (like changing the value of a variable) because the side effect may or may not occur, depending on whether the whole expression is evaluated. This can lead to unexpected bugs in the program.

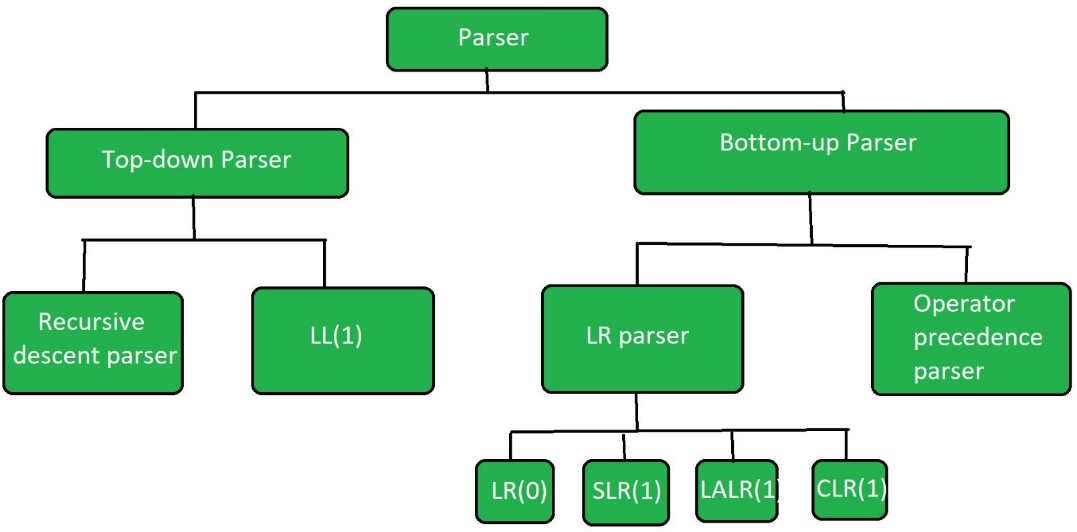
**Operator Evaluation Order**

The operator precedence and associativity rules of a language dictate the order of evaluation of its operators.

* 1. Precedence
  2. Associativity
  3. Parentheses

**Types of Parser:**

The parser is mainly classified into two categories, i.e. Top-down Parser, and Bottom-up Parser. These are explained below:



**Top-Down Parser:**

The top-down parser is the parser that **generates parse for the given input string**with the help of grammar productions by expanding the non-terminals i.e. it starts from the start symbol and ends on the terminals. It uses left most derivation.   
Further Top-down parser is classified into 2 types:

A recursive descent parser, and Non-recursive descent parser.

1. **Recursive descent parser**is also known as the **Brute force parser** or the **backtracking parser**. It basically generates the parse tree by using brute force and backtracking.
2. **Non-recursive descent parser** is also known as **LL(1) parser** or **predictive parser** or **without backtracking parser** or **dynamic parser.** It uses a parsing table to generate the parse tree instead of backtracking.

**Bottom-up Parser:**

Bottom-up Parser is the parser that generates the parse tree for the given input string with the help of grammar productions by **compressing the terminals** i.e. it starts from terminals and ends on the start symbol. It uses the reverse of the rightmost derivation.   
Further Bottom-up parser is classified into two types: LR parser, and Operator precedence parser.

* **LR parser**is the bottom-up parser that generates the parse tree for the given string by using unambiguous grammar. It follows the reverse of the rightmost derivation.   
  LR parser is of four types:

**(a)**LR(0)

**(b)**SLR(1)

**(c)**LALR(1)

**(d)**CLR(1)

* **Operator precedence parser** generates the parse tree from given grammar and string but the only condition is two consecutive non-terminals and epsilon never appears on the right-hand side of any production.
* The operator precedence parsing techniques can be applied to **Operator grammars.**

**Subprograms**

Subprograms, also known as functions or procedures, are reusable blocks of code that perform specific tasks or computations. They are essential in programming because they allow us to break down complex tasks into smaller, manageable pieces, making the code more organized, modular, and easier to maintain.

Let's look at an example of a simple subprogram in Python:

# Subprogram definition (function)

def greet(name):

# This subprogram takes a 'name' as input and prints a greeting message.

print("Hello, " + name + "! How are you today?")

# Main program

# Calling the 'greet' subprogram

greet("Alice")

greet("Bob")

**Design Issues faced by a subprogram:**

1. **Parameter Passing Method**: How parameters are passed to a subprogram when it is called. There are different approaches, like passing by value, passing by reference, etc.
2. **Type Checking of Parameters**: Whether the types of the actual parameters passed to a subprogram are checked against the types of the expected (formal) parameters.
3. **Allocation of Local Variables**: Whether the local variables inside a subprogram are statically allocated (known at compile time) or dynamically allocated (created and destroyed during runtime).
4. **Nesting of Subprogram Definitions**: Whether subprogram definitions can appear inside other subprogram definitions.
5. **Passing Subprogram Names**: Whether subprogram names can be passed as parameters to other subprograms.
6. **Referencing Environment**: If subprogram names can be passed as parameters and subprograms can be nested, what is the referencing environment of a passed subprogram.
7. **Functional Side Effects**: Whether subprograms (especially functions) are allowed to have side effects, which can cause issues and make code harder to understand.
8. **Return Values from Functions:** What types and how many values can be returned from functions.
9. **Overloading**: Whether subprograms can have the same name as another subprogram in the same context but with different parameter types.
10. **Generics**: Whether subprograms can operate on different data types in different calls.

**Function vs. Procedure**

**Function:** A function is a subprogram that returns a value after performing some computations. It takes inputs (arguments) and produces an output. The value returned by a function can be used in expressions or assignments.

**Procedure:** A procedure is a subprogram that performs a sequence of actions without returning a value. It can modify the program's state or provide some side effects. Procedures are often used for tasks that involve performing actions but don't need to return a result.

**# Example of a function in Python**

def add\_numbers(a, b):

    return a + b

```

**# Example of a procedure in Python**

def greet\_user(name):

    print("Hello, " + name + "! Welcome.")

```