Compiler Design

Unit 1: Overview of Compilation

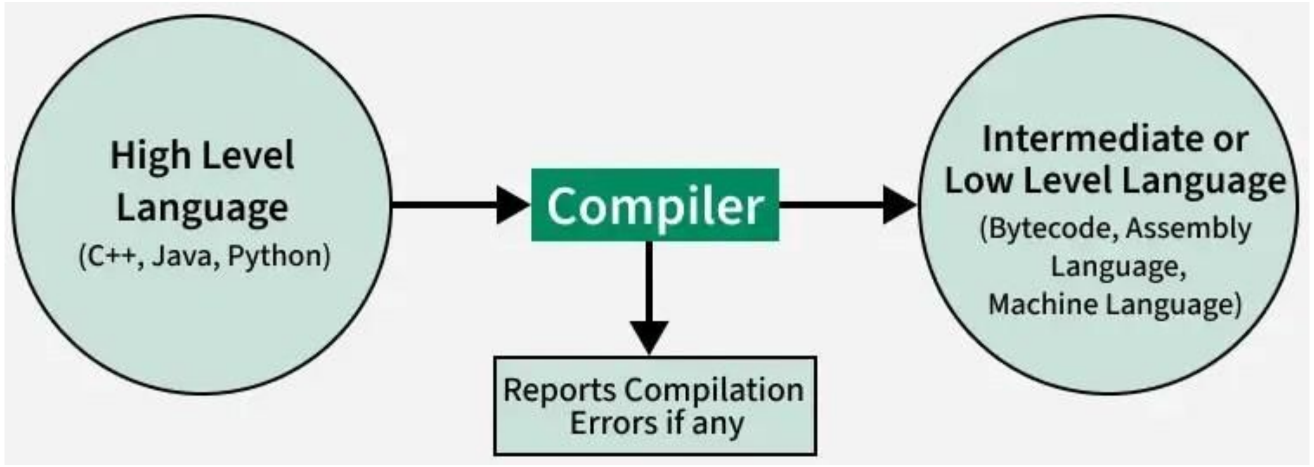
4 Hrs

1.1 Compiler and Interpreter
1.2 Compiler Structure
1.3 Overview of Translation
    1.3.1 The Front End
    1.3.2 The Optimizer
    1.3.3 The Backend

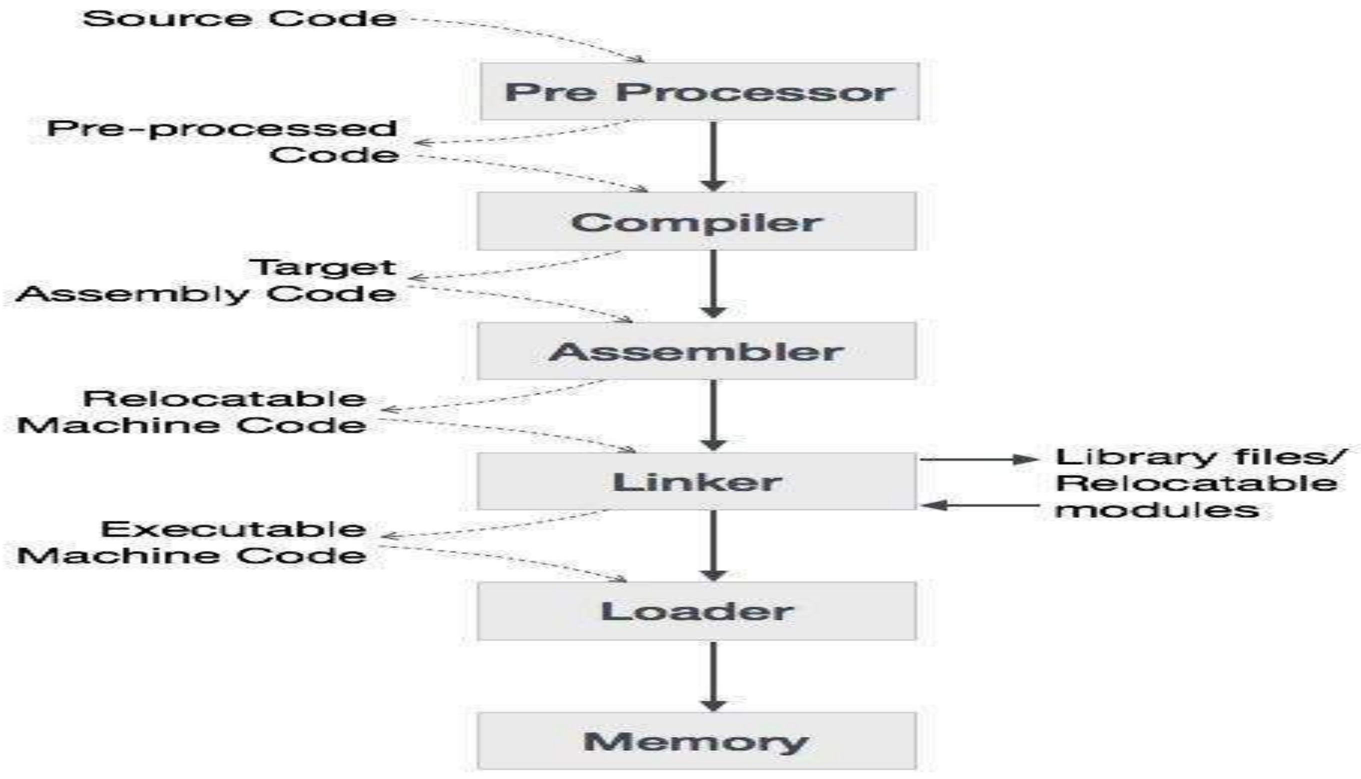
Introduction

What is a Compiler?

A compiler is a software. This is basically a special program that takes the code as input. The code is generally written in a high-level language and turns it into machine language that a computer can run. It acts as a translator, reading the source code, and producing optimized machine code.


High Level Language
(C++, Java, Python)
Compiler
Intermediate or Low Level Language
(Bytecode, Assembly Language, Machine Language)
Reports Compilation Errors if any

It goes through several stages, like **scanning, parsing, and checking** meanings.This is for checking whether everything is correct or not. And, it makes sure that the original logic is preserved. It is a fundamental tool for software development, and it makes computers to understand and execute our instructions written in higher level language.

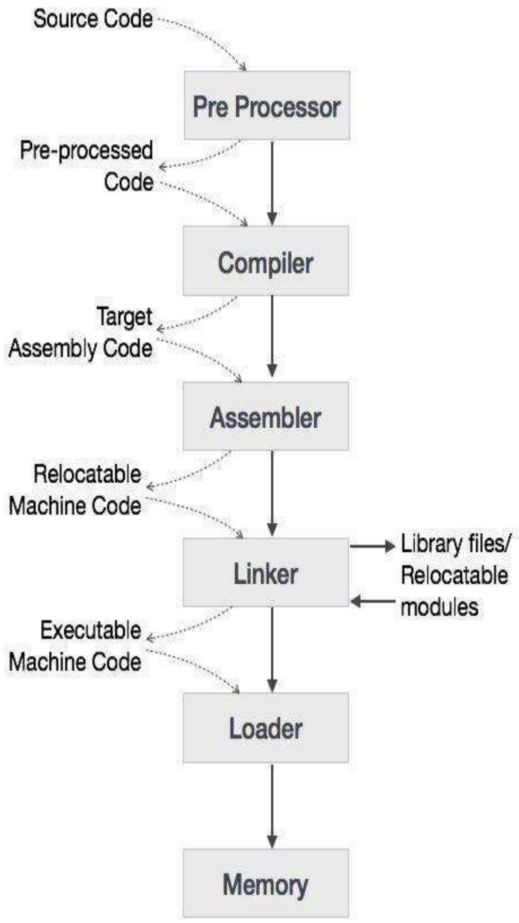


How Compiler Works?

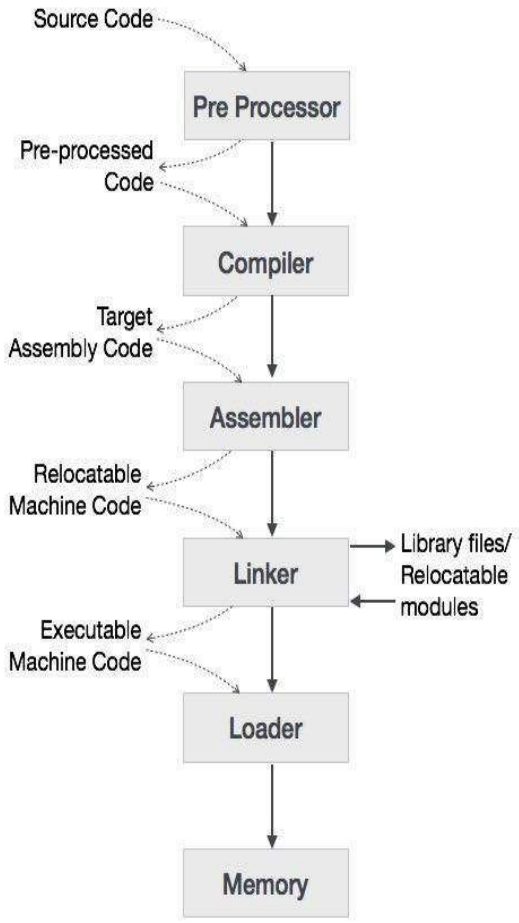
The high-level language is converted into binary language in various phases. A **compiler** is a program that converts high-level language to assembly language. Similarly, an **assembler** is a program that converts the assembly language to machine-level language.

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

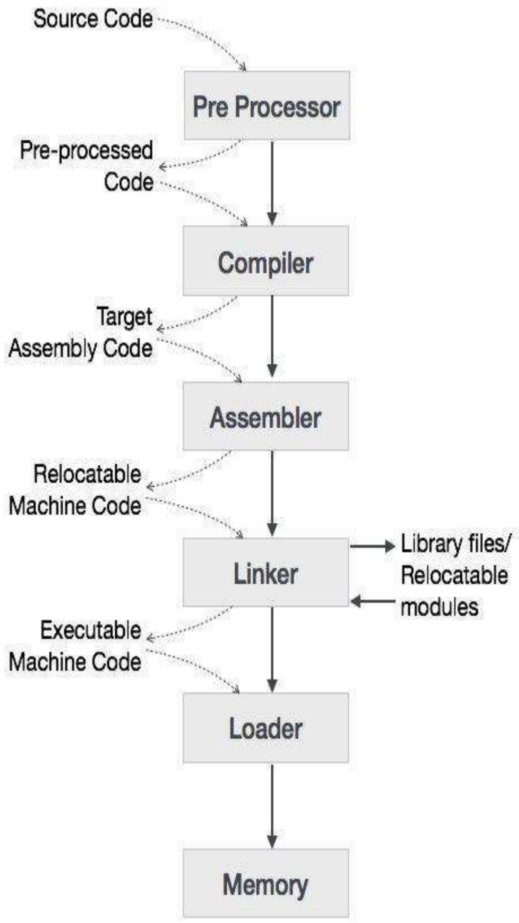
- **User writes a program in C language (high-level language).**
- **The C compiler, compiles the program and translates it to assembly program (low-level language).**
- **An assembler then translates the assembly program into machine code (object).**
- **A linker tool is used to link all the parts of the program together for execution (executable machine code).**
- **A loader loads all of them into memory and then the program is executed.**



- **High-Level Language:** If a program contains pre-processor directives such as #include or #define it is called HLL. They are closer to humans but far from machines. These (#) tags are called preprocessor directives. They direct the pre-processor about what to do.
- **Pre-Processor:** The pre-processor removes all the #include directives by including the files called file inclusion and all the #define directives using macro expansion. It performs file inclusion, augmentation, macro-processing, etc. For example: Let in the source program, it is written #include “Stdio. h”. Pre-Processor replaces this file with its contents in the produced output.
- **Assembly Language:** It’s neither in binary form nor high level. It is an intermediate state that is a combination of machine instructions and some other useful data needed for execution.



- **Assembler:** For every platform (Hardware + OS) we will have an assembler. They are not universal since for each platform we have one. The output of the assembler is called an object file. Its translates assembly language to machine code.
- **Compiler:** The compiler is an intelligent program as compared to an assembler. The compiler verifies all types of limits, ranges, errors, etc. Compiler program takes more time to run and it occupies a huge amount of memory space. The speed of the compiler is slower than other system software. It takes time because it enters through the program and then does the translation of the full program.
- **Interpreter:** An interpreter converts high-level language into low-level machine language, just like a compiler. But they are different in the way they read the input. The Compiler in one go reads the inputs, does the processing, and executes the source code whereas the interpreter does the same line by line. A compiler scans the entire program and translates it as a whole into machine code whereas an interpreter translates the program one statement at a time. Interpreted programs are usually slower concerning compiled ones.



- **Relocatable Machine Code:** It can be loaded at any point and can be run. The address within the program will be in such a way that it will cooperate with the program movement.
- **Loader/Linker:** Loader/Linker converts the relocatable code into absolute code and tries to run the program resulting in a running program or an error message (or sometimes both can happen). Linker loads a variety of object files into a single file to make it executable. Then loader loads it in memory and executes it.
  - **Linker:** The basic work of a linker is to merge object codes (that have not even been connected), produced by the compiler, assembler, standard library function, and operating system resources.
  - **Loader:** The codes generated by the compiler, assembler, and linker are generally re-located by their nature, which means to say, the starting location of these codes is not determined, which means they can be anywhere in the computer memory. Thus the basic task of loaders to find/calculate the exact address of these memory locations.

Types of Compiler

The four main types of compilers are as follows –

1. **Single-Pass Compiler** – A single-pass compiler processes the source code in a single pass, from start to finish, generating machine code as it goes. It is efficient but may not catch all errors or perform extensive optimization.

2. **Multi-Pass Compiler** – A multi-pass compiler makes multiple passes over the source code, analyzing it in different stages. This allows for more thorough error checking and optimization but can be slower than a single-pass compiler.

3. **Just-In-Time (JIT) Compiler** – A JIT compiler translates code into machine language while the program is running, on-the-fly. It is used in languages like Java and JavaScript to improve performance by converting code as needed during execution.

4. **Ahead-of-Time (AOT) Compiler** – An AOT compiler translates code into machine language before the program is run, producing an executable file. This approach is common in languages like C and C++, providing fast execution but requiring compilation before running the program.

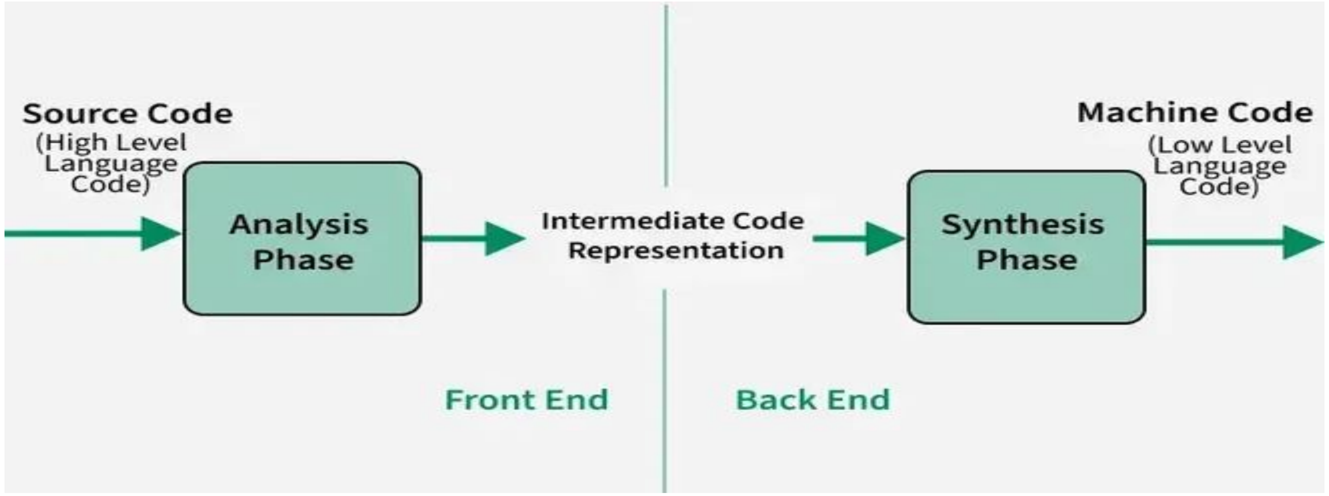
Other types of Compiler

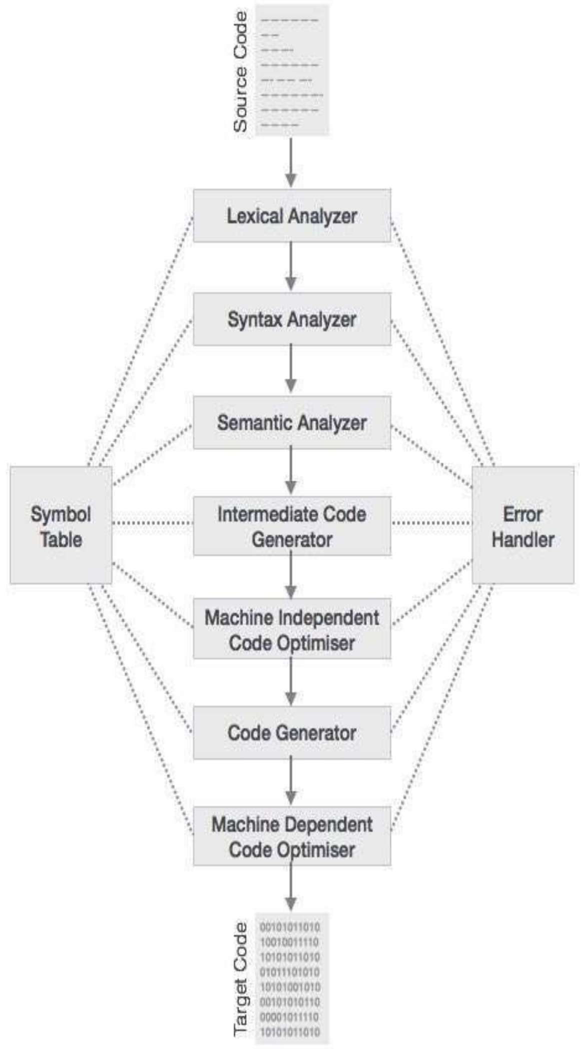
- **Self Compiler:** When the compiler runs on the same machine and produces machine code for the same machine on which it is running then it is called as self compiler or resident compiler.
- **Cross Compiler**: The compiler may run on one machine and produce the machine codes for other computers then in that case it is called a cross-compiler. It is capable of creating code for a platform other than the one on which the compiler is running.
- **Source-to-Source Compiler:** A Source-to-Source Compiler or transcompiler or transpiler is a compiler that translates source code written in one programming language into the source code of another programming language
- **Two Pass Compiler:** Two-pass compiler is a compiler in which the program is translated twice, once from the front end and the back from the back end known as Two Pass Compiler.
- **Incremental Compiler:** It compiles only the parts of the code that have changed, rather than recompiling the entire program. This makes the compilation process faster and more efficient, especially during development.

Structure and Phases of Compiler

A compiler performs this transformation through several phases, each with a specific role in making the code efficient and correct. Broadly, the compilation process can be divided into two main parts:

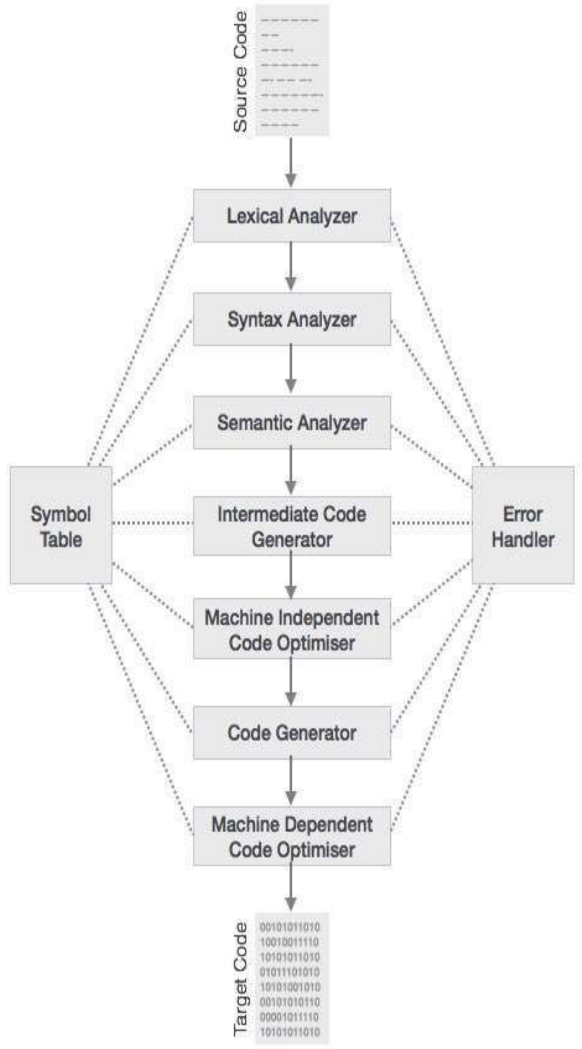
1. **Analysis Phase:** The analysis phase breaks the source program into its basic components and creates an intermediate representation of the program. It is sometimes referred to as front end.
2. **Synthesis Phase:** The synthesis phase creates the final target program from the intermediate representation. It is sometimes referred to as back end.


Source Code
(High Level Language Code)
Analysis Phase
Intermediate Code Representation
Synthesis Phase
Machine Code
(Low Level Language Code)
Front End
Back End



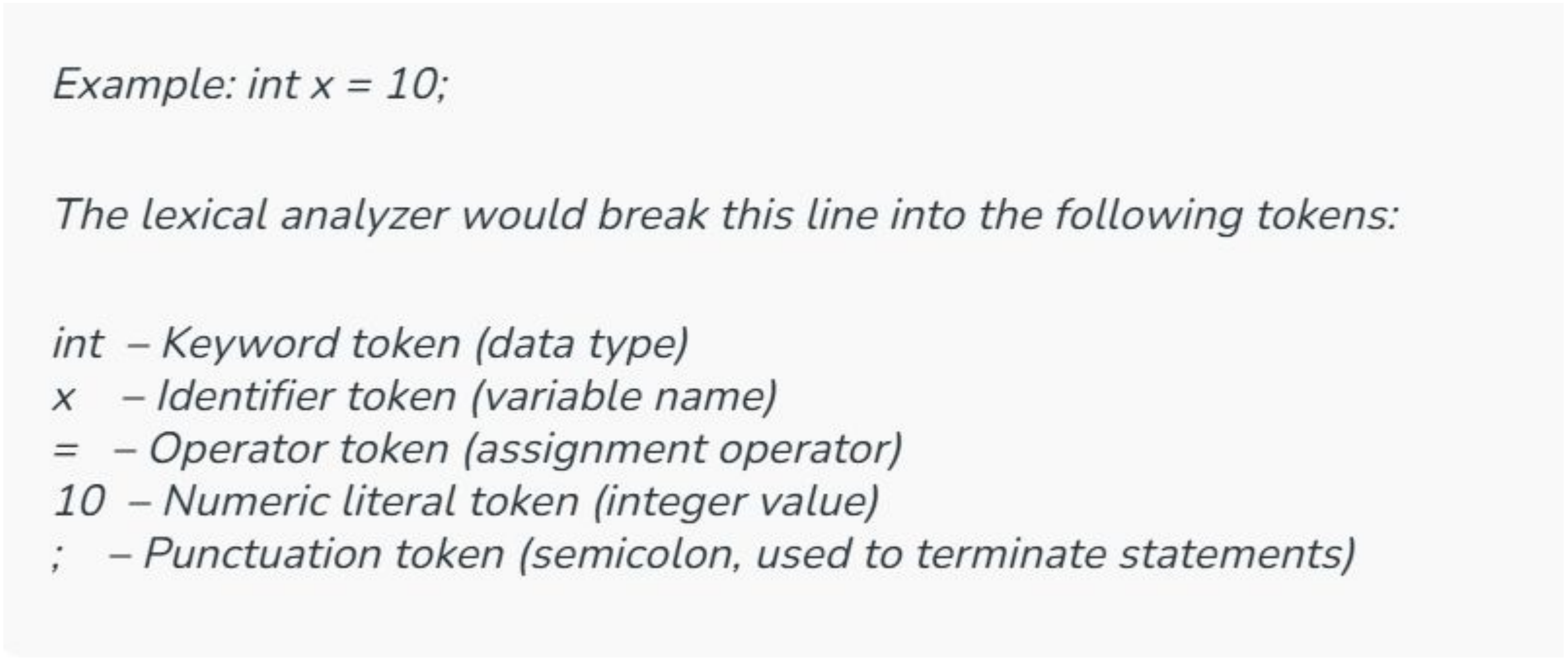
Phases of a Compiler

- The compiler consists of two main parts: the front-end and the back-end. The front-end includes the **lexical analyzer, syntax analyzer, semantic analyzer, and intermediate code generator**. The back-end takes over from there, **handling optimization, code generation, and assembly.**
- 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.



1. **Lexical Analysis:** The first phase, where the source code is broken down into tokens such as keywords, operators, and identifiers for easier processing.
2. **Syntax Analysis or Parsing:** This phase checks if the source code follows the correct syntax rules, building a parse tree or abstract syntax tree (AST).
3. **Semantic Analysis:** It ensures the program's logic makes sense, checking for errors like type mismatches or undeclared variables.
4. **Intermediate Code Generation:** In this phase, the compiler converts the source code into an intermediate, machine-independent representation, simplifying optimization and translation.
5. **Code Optimization:** This phase improves the intermediate code to make it run more efficiently, reducing resource usage or increasing speed.
6. **Target Code Generation:** The final phase where the optimized code is translated into the target machine code or assembly language that can be executed on the computer.

1. Lexical Analysis

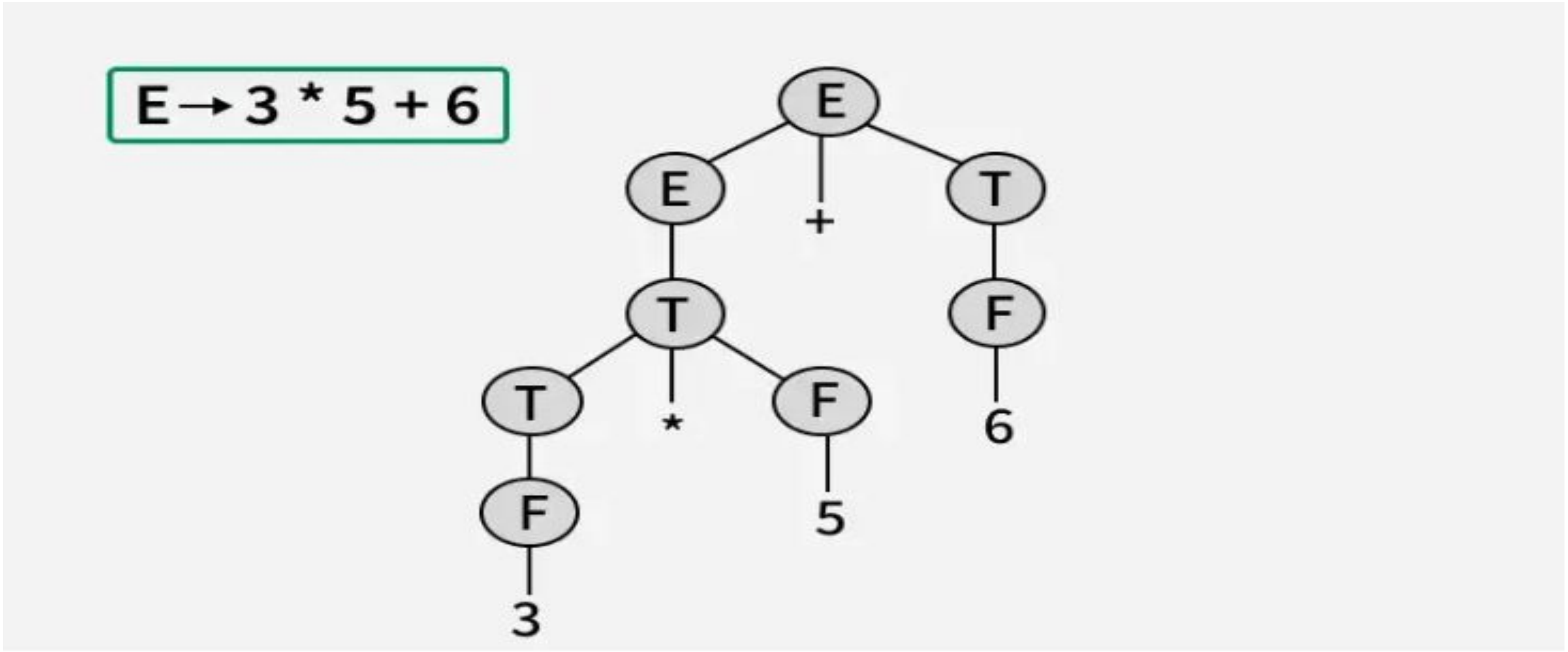


Example: int x = 10;

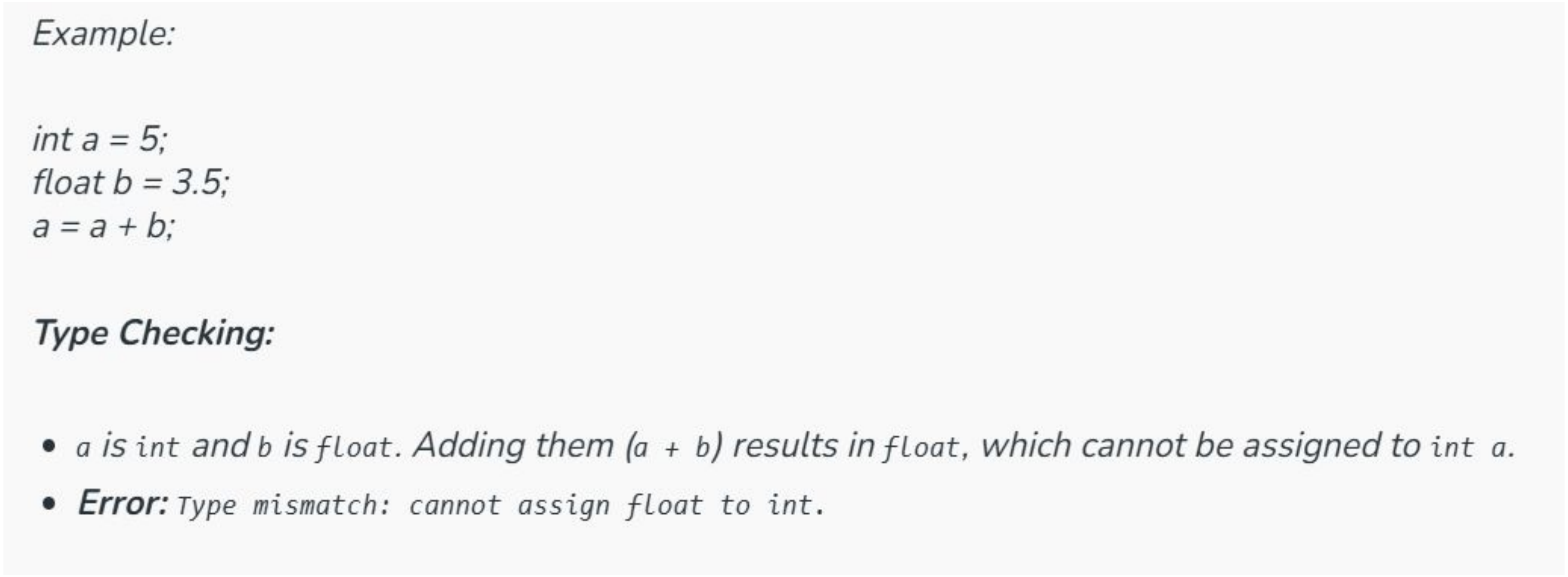
The lexical analyzer would break this line into the following tokens:

int – Keyword token (data type)
x – Identifier token (variable name)
= – Operator token (assignment operator)
10 – Numeric literal token (integer value)
; – Punctuation token (semicolon, used to terminate statements)

2. Syntax Analysis



3. Semantic Analysis



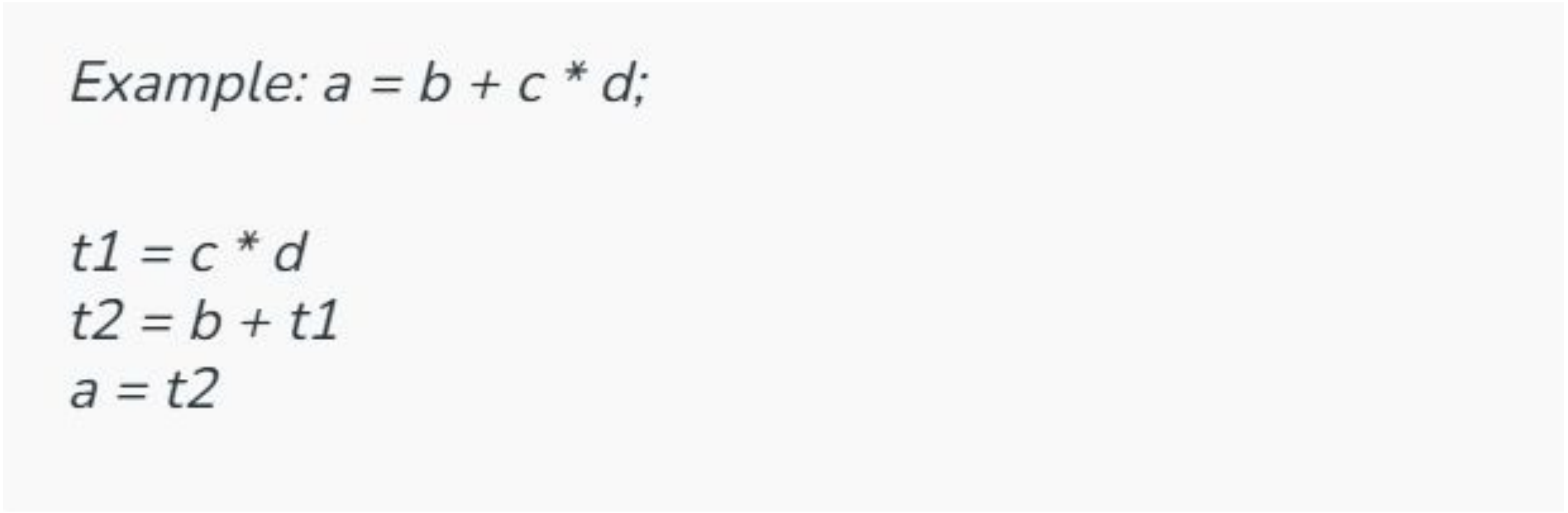
Example:

```
int a = 5;
float b = 3.5;
a = a + b;
```

Type Checking:

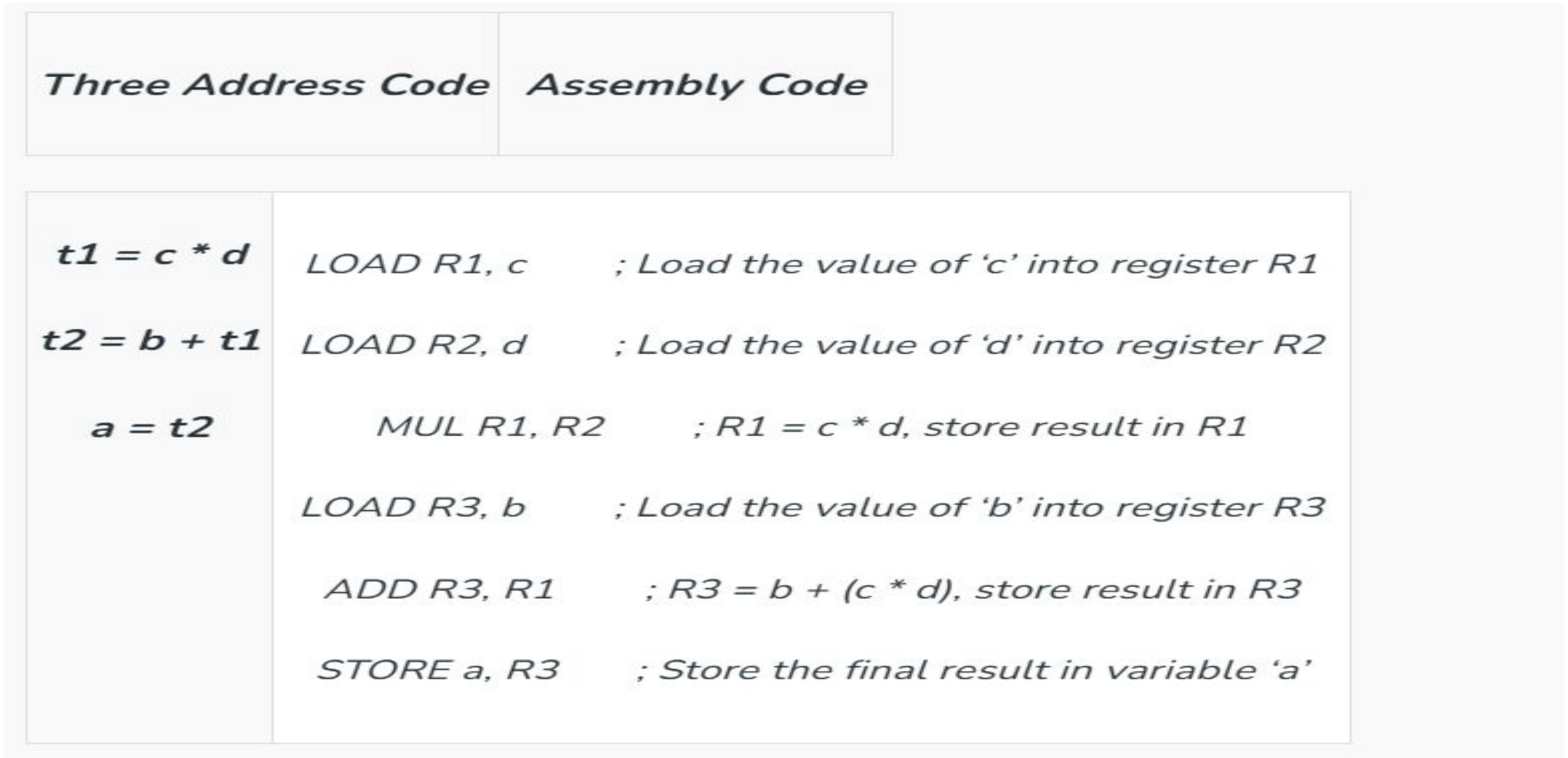
- *`a` is `int` and `b` is `float`. Adding them (`a + b`) results in `float`, which cannot be assigned to `int a`.*
- ***Error:*** `Type mismatch: cannot assign float to int.`

4. Intermediate Code Generation



```
Example: a = b + c * d;

t1 = c * d
t2 = b + t1
a = t2
```

6. Code Generation



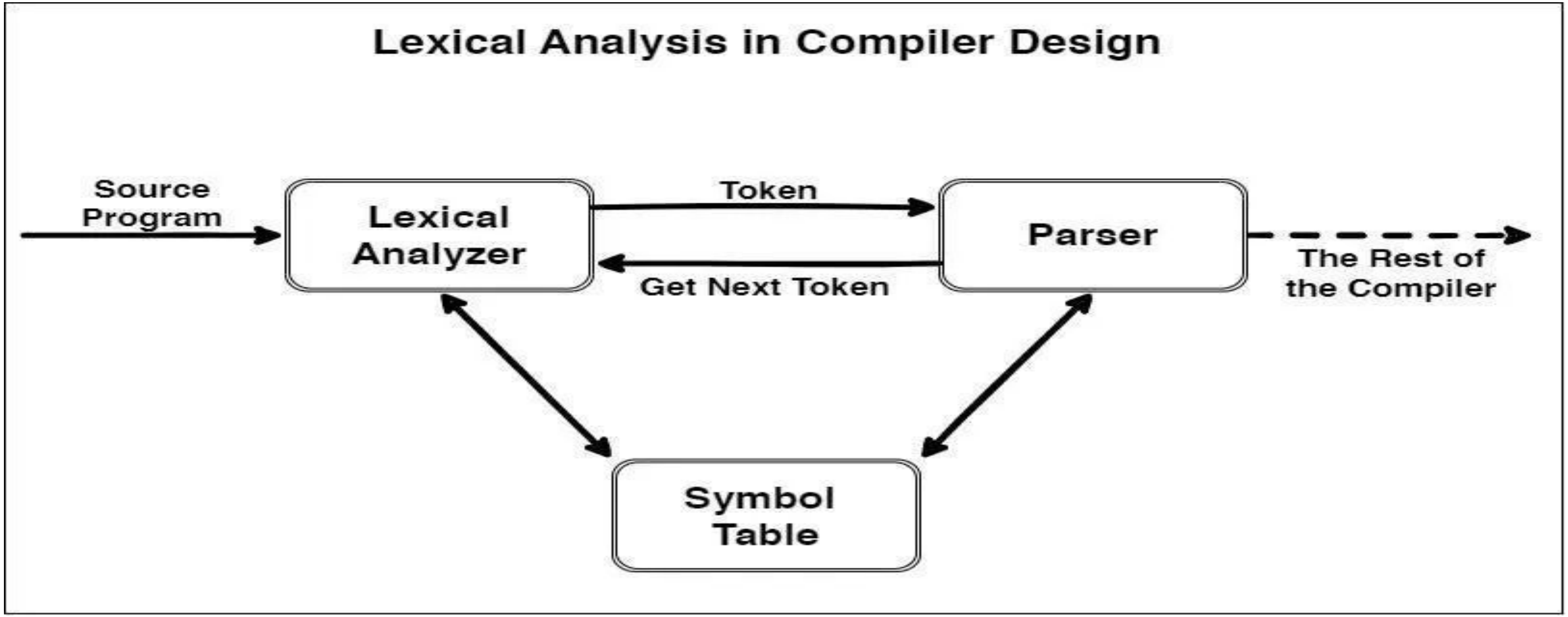
Three Address Code	Assembly Code

t1 = c * d
t2 = b + t1
a = t2
	LOAD R1, c ; Load the value of ‘c’ into register R1
	LOAD R2, d ; Load the value of ‘d’ into register R2
	MUL R1, R2 ; R1 = c * d, store result in R1
	LOAD R3, b ; Load the value of ‘b’ into register R3
	ADD R3, R1 ; R3 = b + (c * d), store result in R3
	STORE a, R3 ; Store the final result in variable ‘a’

Symbol Table

It is a data structure being used and maintained by the compiler, consisting of all the identifier’s names along with their types. It helps the compiler to function smoothly by finding the identifiers quickly.

Unit 2: Scanner(Lexical Analysis)



Scanner(Lexical Analysis)

Lexical analysis is the first phase of a compiler. It takes modified source code from language preprocessors that are written in the form of sentences. The lexical analyzer breaks these syntaxes into a series of tokens, by removing any whitespace or comments in the source code.

Tokens are meaningful sequences of characters. There are usually only a small number of tokens for a programming language including constants (such as integers, doubles, characters, and strings), operators (arithmetic, relational, and logical), punctuation marks and reserved keywords.

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.

Tokens

In programming language, keywords, constants, identifiers, strings, numbers, operators and punctuations symbols can be considered as tokens.

For example, in C language, the variable declaration line

```
int value = 100;
```

contains the tokens:

```
int (keyword), value (identifier), = (operator), 100 (constant) and ;
(symbol).
```

2.1 Role of Scanner

In compiler design, the scanner (also called a lexical analyzer or tokenizer) functions to identify meaningful sequences of characters in the source code and convert them into tokens. These tokens are then passed to the parser for further processing.

Here's a more detailed breakdown of the scanner's functions:

1. Input Processing:

- The scanner takes the source code as a stream of characters.
- It reads this stream character by character.
- It may use input buffering to read larger chunks of input at a time.

2. Tokenization:

- The scanner identifies and extracts meaningful units called tokens.
- Examples of tokens include keywords, identifiers, operators, literals, and punctuation.
- Tokens can be described using regular expressions.
- The scanner uses a deterministic finite automaton (DFA) to recognize these tokens.

3. Token Classification:

- The scanner assigns a type (e.g., keyword, identifier, operator) to each recognized token.
- It may also store information about the token, such as its line number or the text it represents.

4. Error Handling:

- The scanner detects and reports errors in the input, such as invalid characters or unexpected symbols.
- It might generate an error message or code to indicate the type of error.

5. Token Output:

- The scanner produces a stream of tokens, which are then passed to the parser.
- The parser uses these tokens to build a parse tree and eventually generate intermediate code.

2.2 Recognizing Words

In lexical analysis (also called lexing or scanning), recognizing words means identifying meaningful sequences of characters from the input source code—these sequences are called lexemes, and they are matched against patterns defined for tokens.

What is a “word”?

A "word" is usually a token—a basic building block of the source code. Examples of words include:

1. Keywords

Reserved words in the programming language.

- Examples: `if`, `else`, `while`, `return`, `int`, `for`, `class`

2. Identifiers

Names defined by the programmer.

- Examples: `sum`, `totalMarks`, `myFunction`, `x`, `i`
- Rules vary by language (e.g., must start with a letter or underscore)

3. Literals (Constants)

Fixed values written directly in the code.

- **Integer literals**: `10`, `0`, `-99`
- **Float literals**: `3.14`, `0.0`, `-2.5`
- **Character literals**: `'a'`, `'1'`, `'\n'`
- **String literals**: `"hello"`, `"Anil"`

4. Operators

Symbols that perform operations on operands.

- **Arithmetic**: `+`, `-`, `*`, `/`, `%`
- **Relational**: `==`, `!=`, `<`, `>`, `<=`, `>=`
- **Logical**: `&&`, `||`, `!`
- **Assignment**: `=`, `+=`, `-=`

5. Separators / Delimiters

Symbols that separate code elements.

Examples: `(`, `)`, `{`, `}`, `[`, `]`, `,`, `;`, `:`

MAY OR MAYNOT BE A TOKEN

6. Comments (sometimes treated as tokens)

Used for code documentation; ignored during execution.

- Examples: `// this is a comment`, `/* block comment */`

7. Whitespace and Newlines

Usually ignored but can be important in some languages (like Python).

- **Whitespace**: spaces, tabs
- **Newlines**: `\n`

How are words recognized?

1. **Regular expressions** are used to define patterns for each type of token.
2. The lexer (or scanner) scans the input character by character.
3. When it finds a sequence that matches a pattern, it groups those characters into a **token** and classifies it.

Example: For this code:

```
int x = 10;
```

The lexer might recognize the following tokens:

- `int` → **Keyword**
- `x` → **Identifier**
- `=` → **Assignment Operator**
- `10` → **Integer Literal**
- `;` → **Semicolon**

What are "complex words"?

“complex words” typically refer to:

- Multi-character identifiers: `userName123`
- Compound operators: `+=`, `==`, `&&`
- Floating-point numbers: `3.14159`, `0.001`
- Strings with escape sequences: `"Hello\nWorld"`
- Comments (which might span lines or contain symbols)
- Keywords vs. identifiers (e.g., `int` is a keyword, `intVar` is an identifier)

Regular Expression

The lexical analyzer needs to scan and identify only a finite set of valid string/token/lexeme that belong to the language in hand. It searches for the pattern defined by the language rules.

Regular expressions have the capability to express finite languages by defining a pattern for finite strings of symbols. The grammar defined by regular expressions is known as **regular grammar**. The language defined by regular grammar is known as **regular language**.

Regular expression is an important notation for specifying patterns. Each pattern matches a set of strings, so regular expressions serve as names for a set of strings. Programming language tokens can be described by regular languages. The specification of regular expressions is an example of a recursive definition. Regular languages are easy to understand and have efficient implementation.

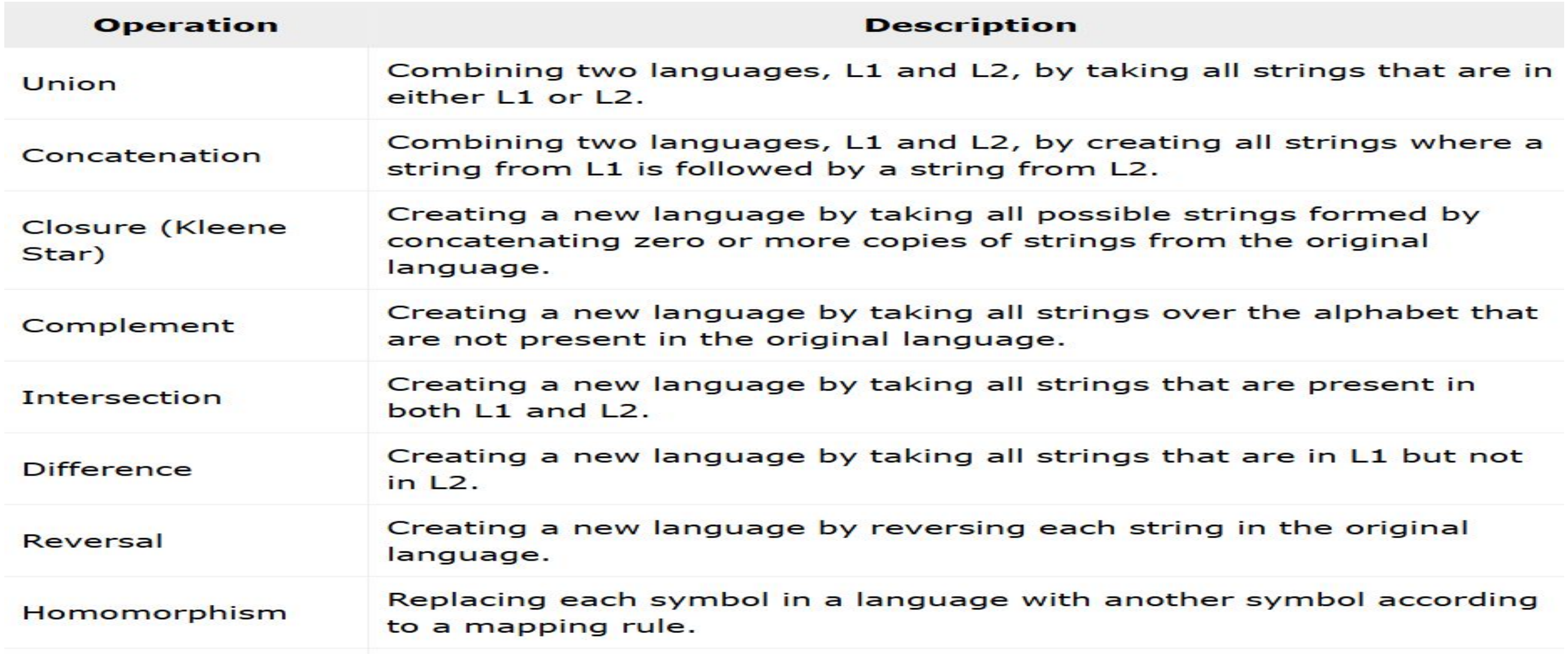
Difference between Null Strings and Empty Sets

A **null string** () is a valid string with no characters. It is like a blank piece of paper. For example, consider the language {, 0, 00}. It includes the null string and strings of zeros, but it is not an empty set because it contains elements.

An **empty set** (φ), on the other hand, contains no strings at all. It is like having no paper to write on.

Closure properties of Regular languages

Closure properties on regular languages are defined as certain operations on regular language that are guaranteed to produce regular language. Closure refers to some operation on a language, resulting in a new language that is of the same “type” as originally operated on i.e., regular.



Operation	Description
Union	Combining two languages, L1 and L2, by taking all strings that are in either L1 or L2.
Concatenation	Combining two languages, L1 and L2, by creating all strings where a string from L1 is followed by a string from L2.
Closure (Kleene Star)	Creating a new language by taking all possible strings formed by concatenating zero or more copies of strings from the original language.
Complement	Creating a new language by taking all strings over the alphabet that are not present in the original language.
Intersection	Creating a new language by taking all strings that are present in both L1 and L2.
Difference	Creating a new language by taking all strings that are in L1 but not in L2.
Reversal	Creating a new language by reversing each string in the original language.
Homomorphism	Replacing each symbol in a language with another symbol according to a mapping rule.

Reverse Homomorphism	The reverse operation of homomorphism.
Quotient	Creating a new language by dividing one language by another.
Initiate	Creating a new language by taking all prefixes of strings in the original language.
Substitution	Replacing each symbol in a language with another language according to a mapping rule.
Infinite Union	Creating a new language by combining an infinite number of languages.

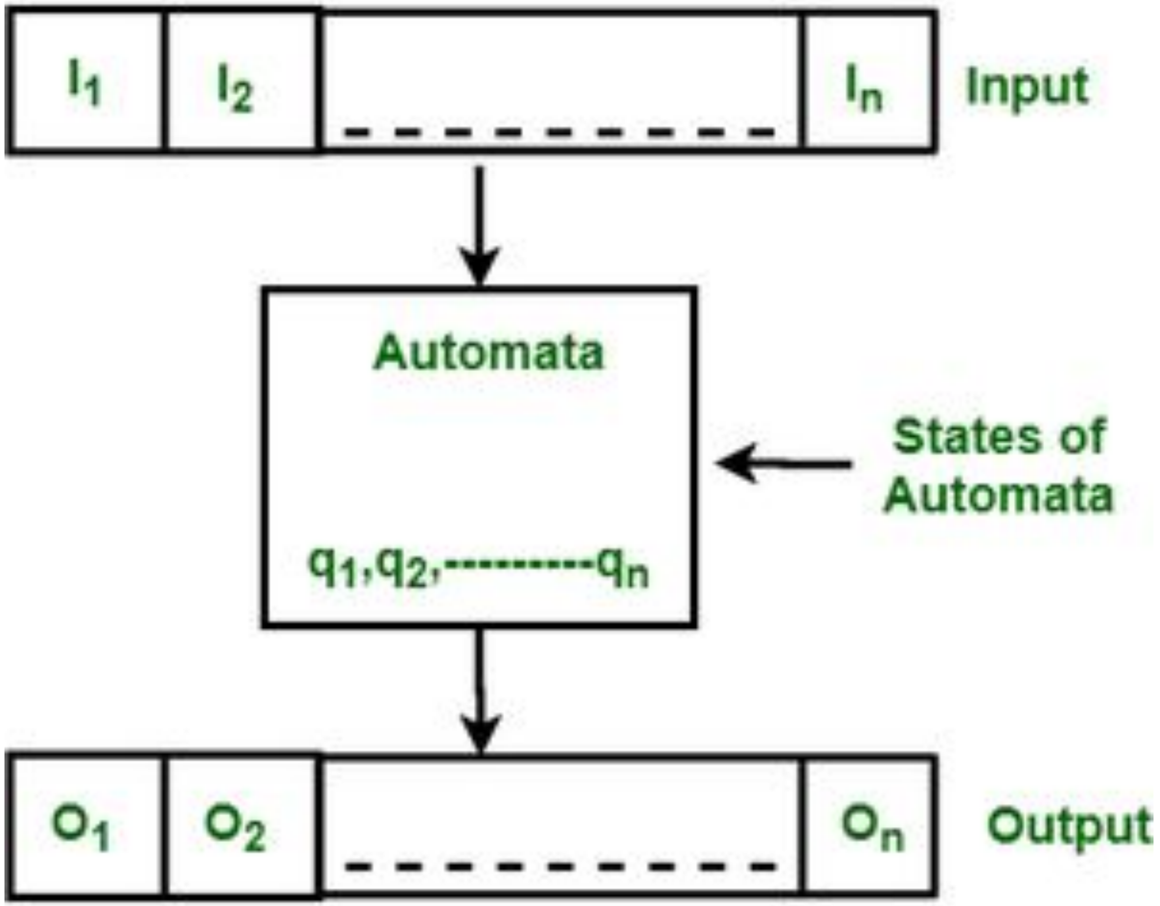
Finite Automata

Finite automata is a state machine that takes a string of symbols as input and changes its state accordingly.

Finite automata is a recognizer for regular expressions. When a regular expression string is fed into finite automata, it changes its state for each literal.

If the input string is successfully processed and the automata reaches its final state, it is accepted, i.e., the string just fed was said to be a valid token of the language in hand.

Finite automata are abstract machines used to recognize patterns in input sequences, forming the basis for understanding regular languages in computer science. They consist of states, transitions, and input symbols, processing each symbol step-by-step. If the machine ends in an accepting state after processing the input, it is accepted; otherwise, it is rejected.


I_1
I_2
I_n
Input
Automata
$q_1, q_2, \text{---------} q_n$
States of Automata
O_1
O_2
O_n
Output

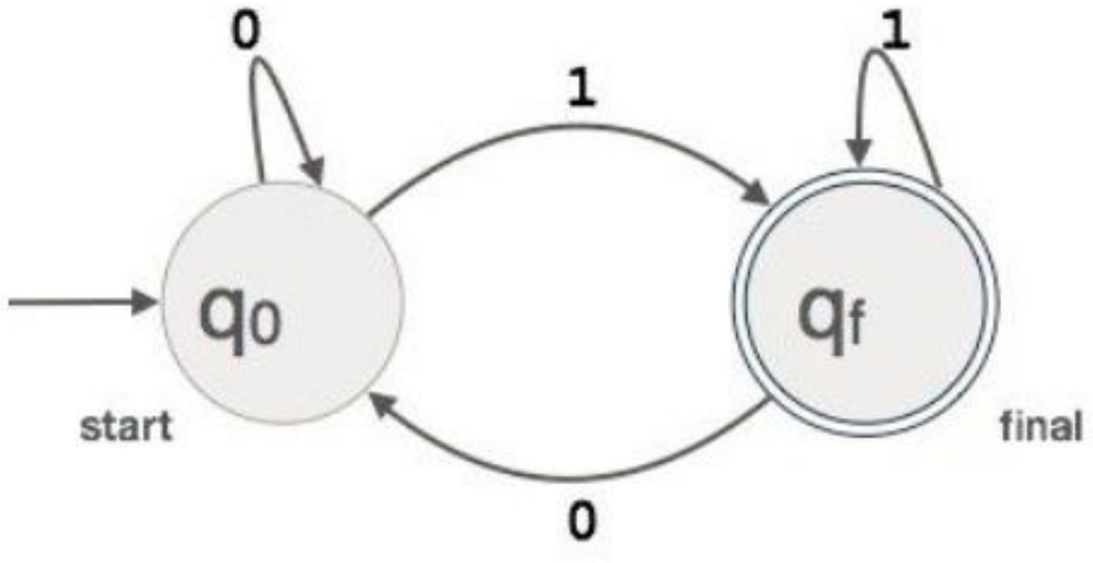
Mathematical Model of Finite Automata

A finite automaton can be defined as a tuple:

{ Q, ∑, q, F, δ }, where:

- Q: Finite set of states
- ∑: Set of input symbols
- q: Initial state
- F: Set of final states
- δ: Transition function

Example : We assume FA accepts any three digit binary value ending in digit 1.

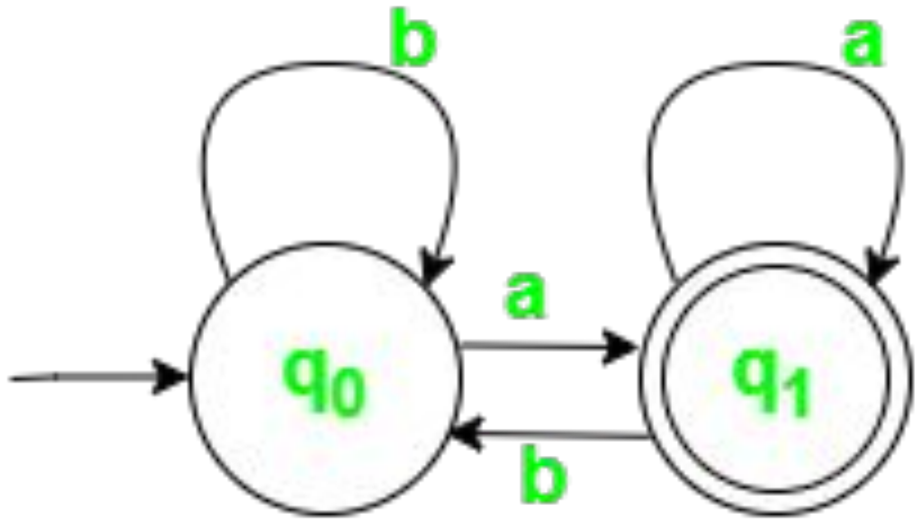


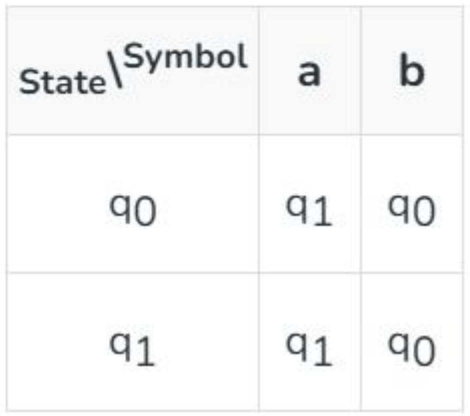
1. Deterministic Finite Automata (DFA)

A DFA is represented as {Q, ∑, q, F, δ}. In DFA, for each input symbol, the machine transitions to one and only one state. DFA does not allow any null transitions, meaning every state must have a transition defined for every input symbol.

```
DFA consists of 5 tuples {Q, Σ, q, F, δ}.

Q : set of all states.

Σ : set of input symbols. ( Symbols which machine takes as input )

q : Initial state. ( Starting state of a machine )

F : set of final state.

δ : Transition Function, defined as δ : Q X Σ --> Q.
```

Construct a DFA that accepts all strings ending with ‘a’.


$$\begin{tikzcd} \arrow[r] & q_0 \arrow[loop above, "b"] \arrow[r, bend left, "a"] & \boxed{q_1} \arrow[loop above, "a"] \arrow[l, bend left, "b"] \end{tikzcd}$$

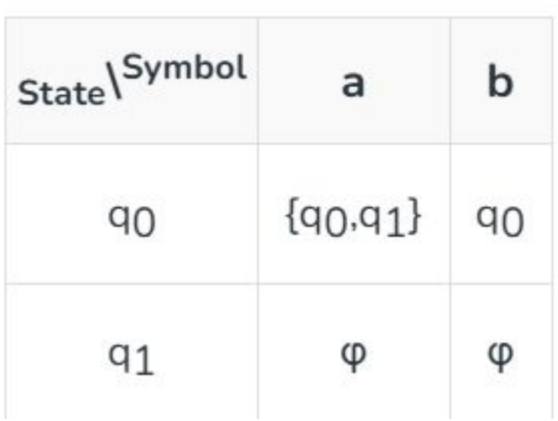


State\Symbol	a	b
q0	q1	q0
q1	q1	q0

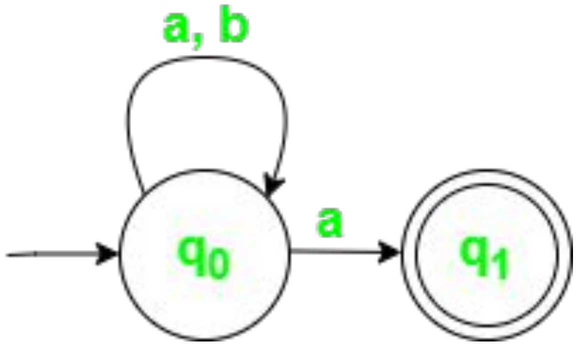
2) Non-Deterministic Finite Automata (NFA)

NFA is similar to DFA but includes the following features:

- It can transition to multiple states for the same input.
- It allows null (ϵ) moves, where the machine can change states without consuming any input.



State\Symbol	a	b
q0	{q0,q1}	q0
q1	φ	φ

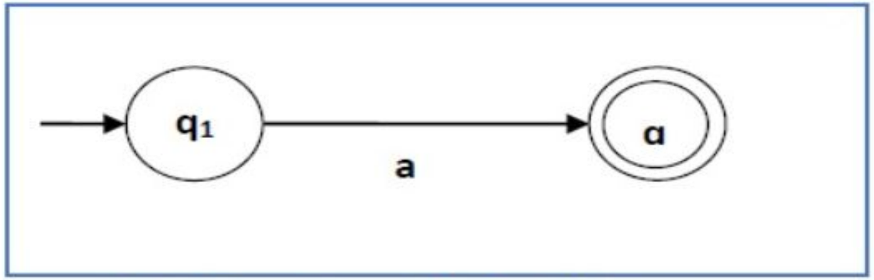


Draw a non deterministic finite automate which accept 00 and 11 at the end of a string containing 0, 1 in it, e.g., 01010100 but not 000111010

Regular Expression to NFA: Thomson’s Construction

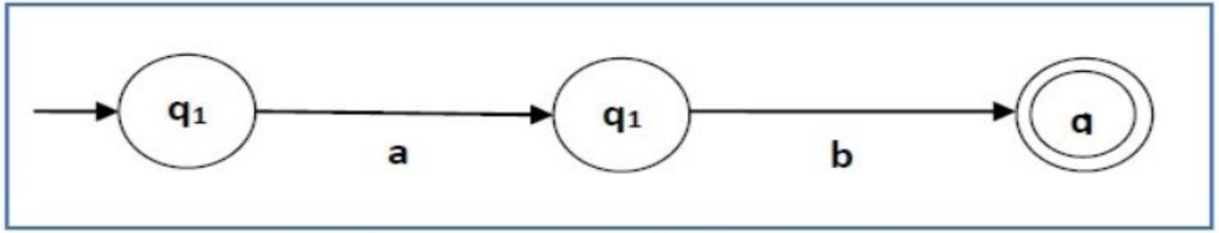
We can use Thompson's Construction to find out a Finite Automaton from a Regular Expression. We will reduce the regular expression into smallest regular expressions and converting these to NFA and finally to DFA.

Case 1 – For a regular expression a, we can construct the following FA –



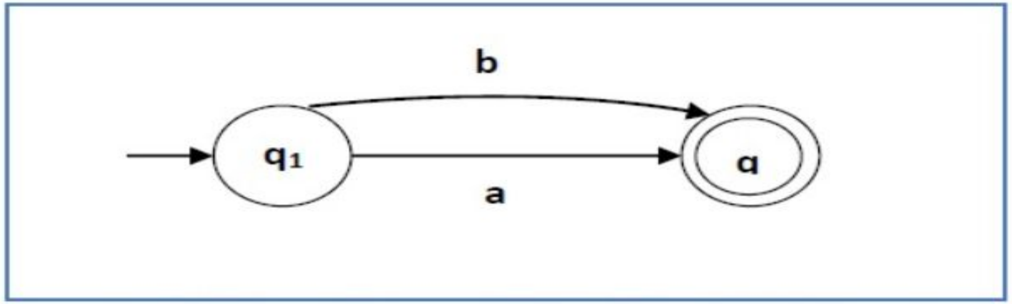
Finite automata for RE = a

Case 2 – For a regular expression ab, we can construct the following FA –



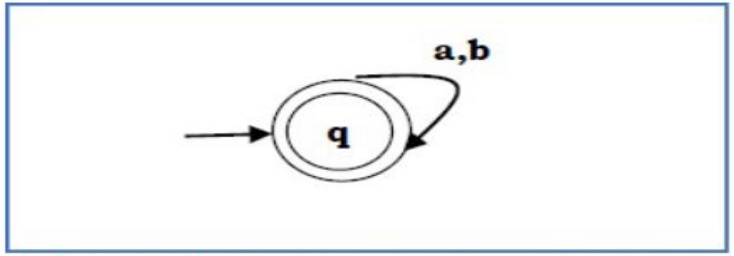
Finite automata for RE = ab

Case 3 – For a regular expression (a+b), we can construct the following FA –



Finite automata for RE= (a+b)

Case 4 – For a regular expression (a+b)*, we can construct the following FA –



Finite automata for RE= (a+b)*

Unit 3: Parser

Syntax Analyzer OR Parser

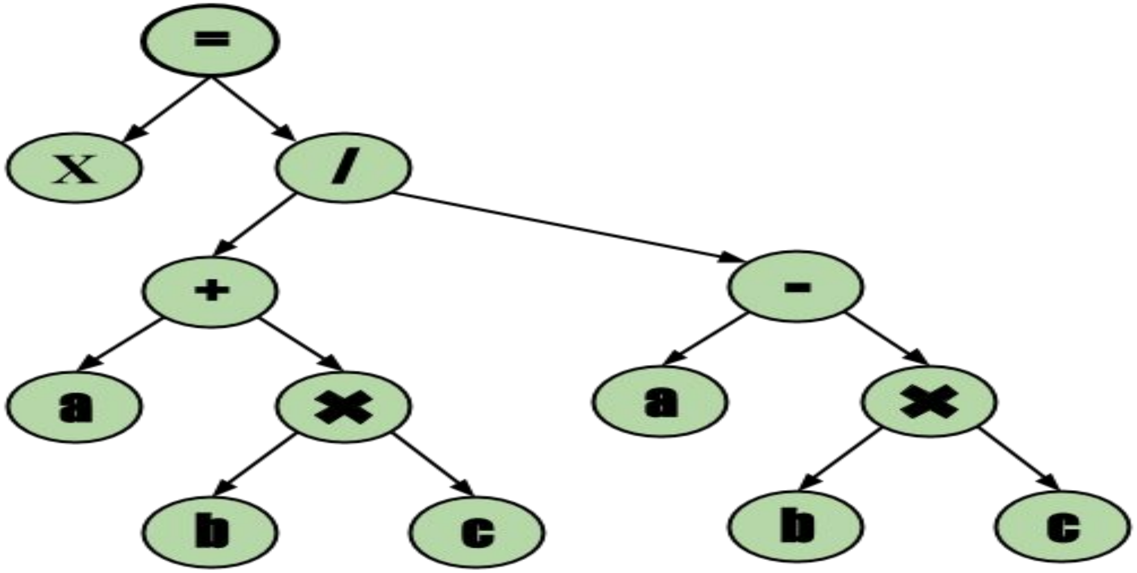
Parsing, also known as syntactic analysis, is the process of analyzing a sequence of tokens to determine the grammatical structure of a program. It takes the stream of tokens, which are generated by a lexical analyzer or tokenizer, and organizes them into a parse tree or syntax tree.

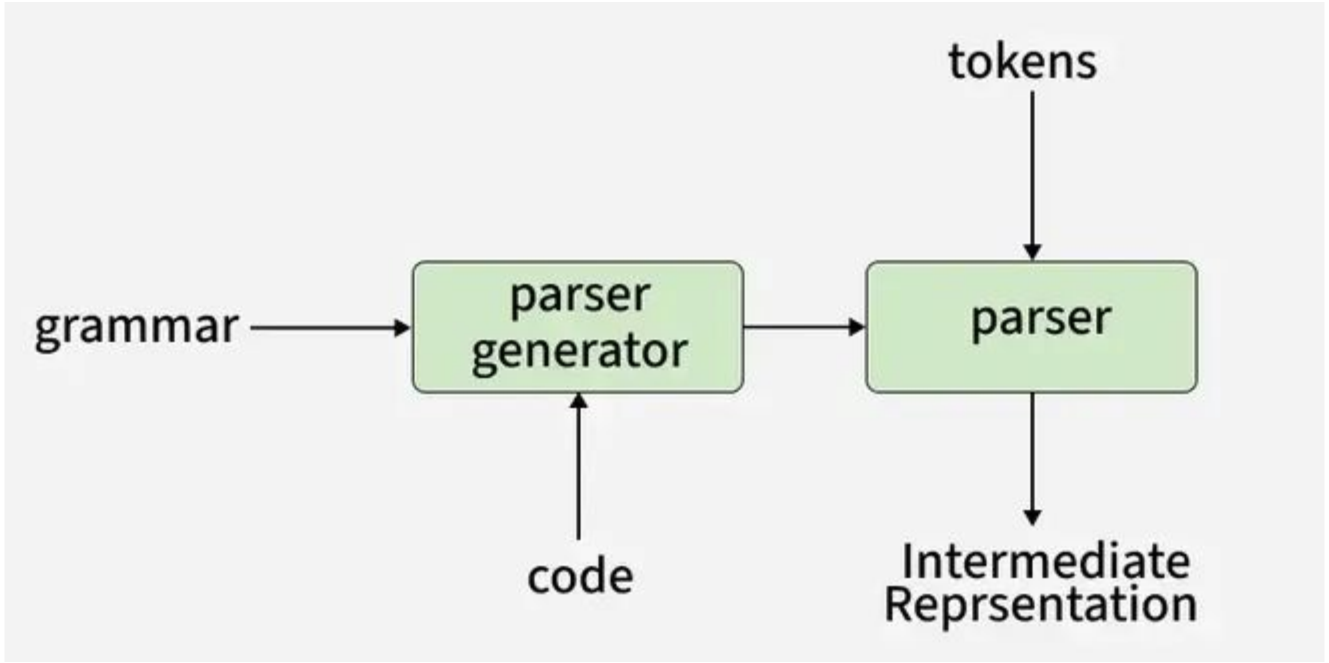
We have seen that a lexical analyzer can identify tokens with the help of regular expressions and pattern rules. But a lexical analyzer cannot check the syntax of a given sentence due to the limitations of the regular expressions. Regular expressions cannot check balancing tokens, such as parenthesis. Therefore, this phase uses context-free grammar (CFG), which is recognized by push-down automata.

The parse tree visually represents how the tokens fit together according to the rules of the language's syntax. This tree structure is crucial for understanding the program's structure and helps in the next stages of processing, such as code generation or execution.

$$X \underset{\downarrow}{=} (a + (b * c)) / (a - (b * c))$$

Operator Root



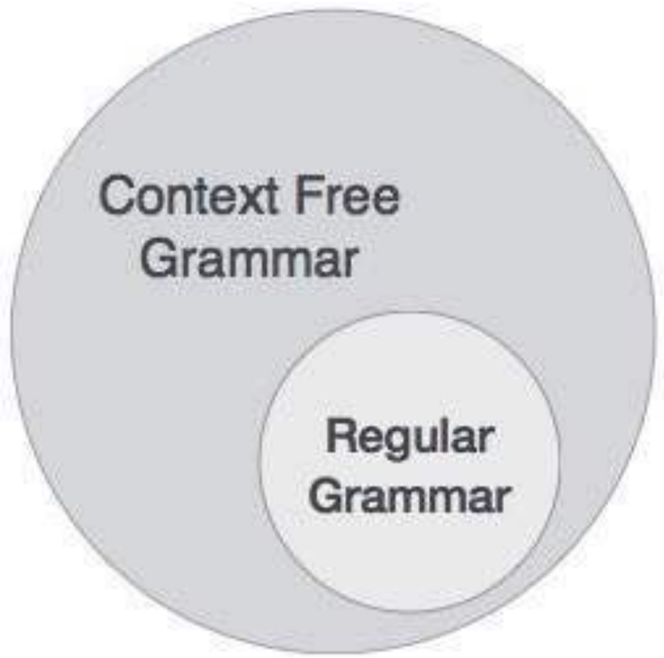

tokens
grammar
parser generator
parser
code
Intermediate Reprsentation

What is the Role of Parser?

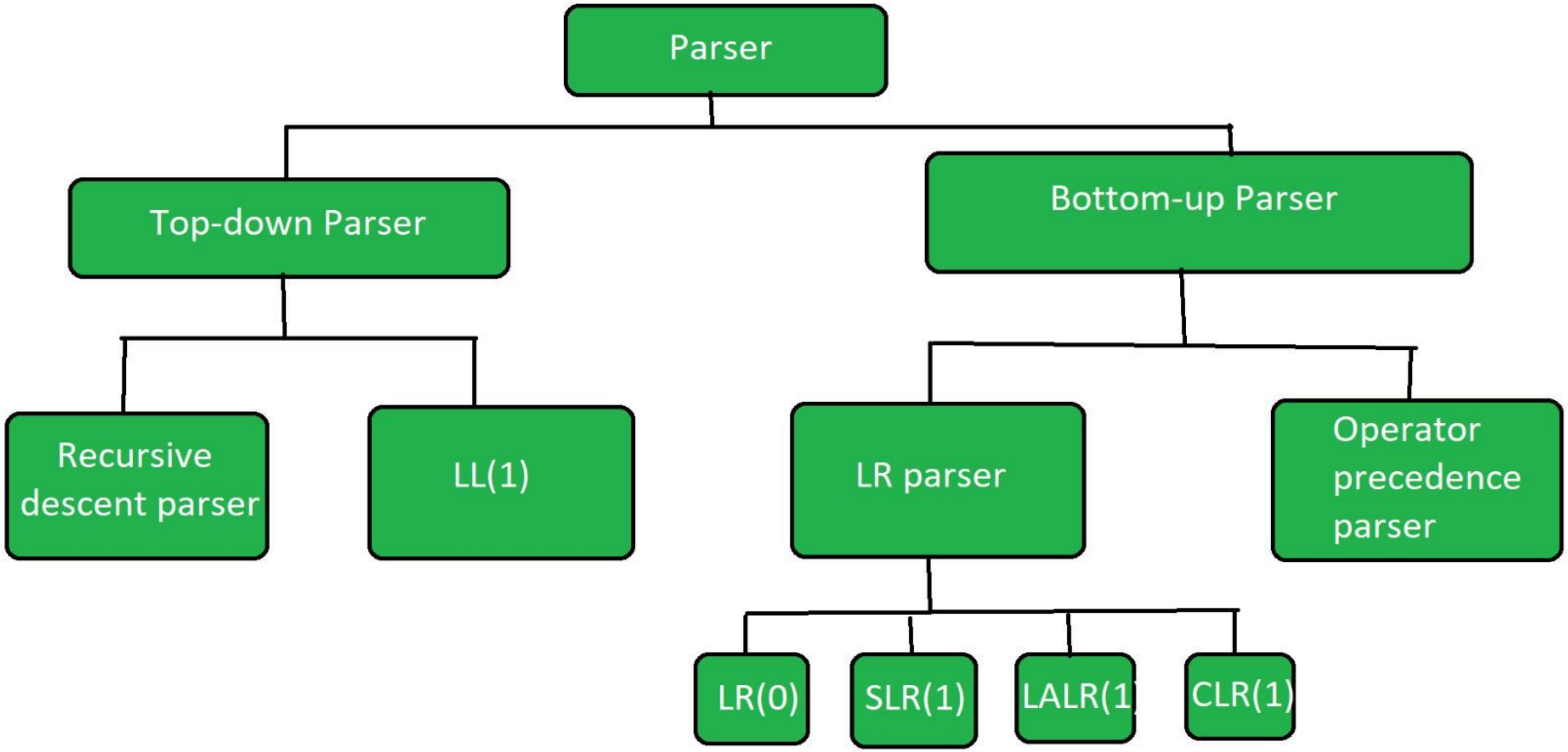
A parser performs syntactic and semantic analysis of source code, converting it into an intermediate representation while detecting and handling errors.

1. **Context-free syntax analysis**: The parser checks if the structure of the code follows the basic rules of the programming language (like grammar rules). It looks at how words and symbols are arranged.
2. **Guides context-sensitive analysis**: It helps with deeper checks that depend on the meaning of the code, like making sure variables are used correctly. For example, it ensures that a variable used in a mathematical operation, like x + 2, is a number and not text.
3. **Constructs an intermediate representation**: The parser creates a simpler version of your code that's easier for the computer to understand and work with.
4. **Produces meaningful error messages**: If there's something wrong in your code, the parser tries to explain the problem clearly so you can fix it.
5. **Attempts error correction**: Sometimes, the parser tries to fix small mistakes in your code so it can keep working without breaking completely.

Context-Free Grammar



Types of Parsers



Transforming a grammar for top down parsing

To make a grammar suitable for top-down parsing, it needs to be transformed to remove left recursion and left-factoring.

These transformations ensure that the grammar can be parsed efficiently using a top-down parser like an LL(1) parser, which reads the input from left to right and constructs a left-most derivation.

1. Eliminating Left Recursion:

What is left recursion?

A grammar is left-recursive if a non-terminal can directly or indirectly derive a string that starts with itself.

Why is it problematic for top-down parsing?

Left-recursive grammars can lead to infinite loops in top-down parsers because the parser might repeatedly expand the same non-terminal without making progress.

How to eliminate it?

- **Direct left recursion:** If you have A -> A α | β, where α and β don't start with A, transform it into:
  - `A -> β A' `
  - A' -> α A' | ε (ε is the empty string)
- **Indirect left recursion:** If you have A -> B α and B -> A β, you'll need to eliminate the recursion through the B non-terminal first, and then apply the direct left recursion elimination to A.

2. Left Factoring:

What is left-factoring?

A grammar is left-factored if it has two or more productions with the same non-terminal on the left side that share a common prefix.

Why is it problematic for top-down parsing?

Left-factoring can make it difficult for a top-down parser to choose the correct production when it encounters a shared prefix in the input.

How to eliminate it?

- **Identify shared prefixes:** Find productions with the same non-terminal that start with the same symbols.
- **Introduce a new non-terminal:** Factor out the common prefix and introduce a new non-terminal (e.g., A') to represent the rest of the production.
    - For example, if you have `A -> α β | α γ`, you can transform it to `A -> α A' ` and `A' -> β | γ`.

Top-Down Parser

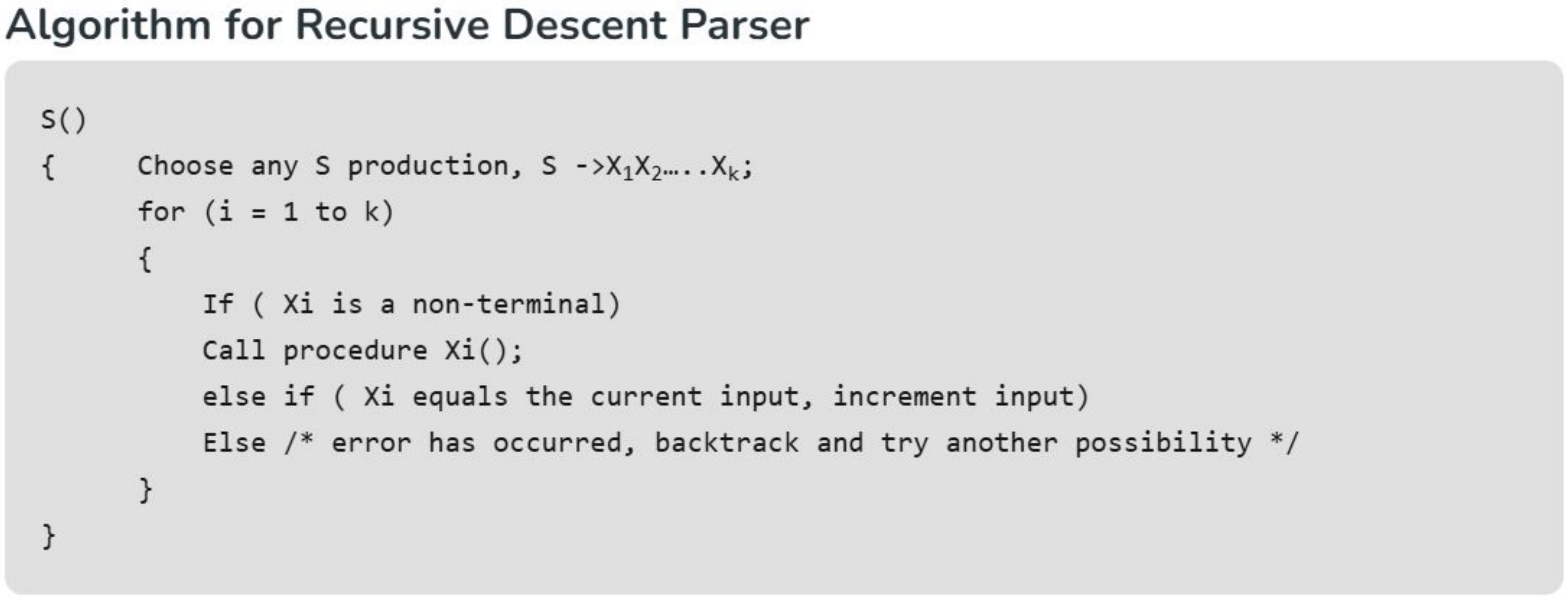
Top-down parser is the parser that generates parse tree for the given input string with the help of grammar productions by expanding the non-terminals. It starts from the start symbol and ends down on the terminals. It uses left most derivation.

Further Top-down parser is classified into 2 types:

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 techniques.
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.

Recursive Descent Parser

A **recursive descent parser** is a top-down parser that processes input based on a set of recursive functions, where each function corresponds to a grammar rule. It parses the input from left to right, constructing a parse tree by matching the grammar's production rules.



Algorithm for Recursive Descent Parser

```
S()
{     Choose any S production, S ->$X_1X_2$…..$X_k$;
      for (i = 1 to k)
      {
          If ( Xi is a non-terminal)
          Call procedure Xi();
          else if ( Xi equals the current input, increment input)
          Else /* error has occurred, backtrack and try another possibility */
      }
}
```

The given grammar is:

```
E  → i E'
E' → + i E' | ε
```

Function `E()`

```
E()
{
    if (input == 'i') {  // If the input is 'i' (identifier)
    input++;        // Consume 'i'
}
   E'();     // Call E' to check for further expressions
}
```

- It checks for `i` (identifier).
- If found, it moves the input pointer ahead.
- Calls `E'()` to check if a `+` operation exists.

Function `E'()`

```
void E`() {
    if (input == '+') {
        input++;           // Consume the '+'

        if (input == 'i') {
            input++;       // Consume the 'i'
        }

        E`();           // Recursively process more additions
    } else {
        return;            // If no '+', return (ε production)
    }
}
```

- It checks for `+ i`.
- If found, it consumes them and calls `E'()` recursively.
- If no `+`, it returns (ε production).

Main Function

```
Main()
{
    E();              // Start parsing from E
    if (input == '$') // If we reach end of input
    Parsing Successful;
}
```

- Calls `E()` to start parsing.
- Checks if the input ends with `$`, which indicates a successful parse.

What is LL(1) Parsing?

Here the 1st **L** represents that the scanning of the Input will be done from the Left to Right manner and the second **L** shows that in this parsing technique, we are going to use the Left most Derivation Tree. And finally, the **1** represents the number of look-ahead, which means how many symbols you will see when you want to make a decision.

Conditions for an LL(1) Grammar

To construct a working LL(1) parsing table, a grammar must satisfy these conditions:

- No Left Recursion: Avoid recursive definitions like A -> A + b.
- Unambiguous Grammar: Ensure each string can be derived in only one way.
- Left Factoring: Make the grammar deterministic, so the parser can proceed without guessing.

Algorithm to Construct LL(1) Parsing Table

Step 1: First check all the essential conditions mentioned above and go to step 2.

Step 2: Calculate First() and Follow() for all non-terminals.

1. **First():** If there is a variable, and from that variable, if we try to drive all the strings then the beginning Terminal Symbol is called the First.
2. Follow(): What is the Terminal Symbol which follows a variable in the process of derivation.

Step 3: For each production A –> α. (A tends to alpha)

1. Find First(α) and for each terminal in First(α), make entry A –> α in the table.
2. If First(α) contains ε (epsilon) as terminal, then find the Follow(A) and for each terminal in Follow(A), make entry A –> ε in the table.
3. If the First(α) contains ε and Follow(A) contains $ as terminal, then make entry A –> ε in the table for the $.

To construct the parsing table, we have two functions:

In the table, rows will contain the Non-Terminals and the column will contain the Terminal Symbols. All the **Null Productions** of the Grammars will go under the Follow elements and the remaining productions will lie under the elements of the First set.

Example 1: Consider the Grammar:

```
E --> TE'
E' --> +TE' | ε
T --> FT'
T' --> *FT' | ε
F --> id | (E)

*ε denotes epsilon
```

Step 1: The grammar satisfies all properties

Step 2: Calculate first() and follow().

Find their First and Follow sets:

	First	Follow
E –> TE’	{ id, (}	{ $,) }
E’ –> +TE’/ ε	{ +, ε }	{ $,) }
T –> FT’	{ id, (}	{ +, $,) }
T’ –> *FT’/ ε	{ *, ε }	{ +, $,) }
F –> id/(E)	{ id, (}	{ *, +, $,) }

Step 5: Make a parser table.

Now, the LL(1) Parsing Table is:

	id	**+**	*****	**(**	**)**	**$**
E	E –> TE’			E –> TE’		
E’		E’ –> +TE’			E’ –> ε	E’ –> ε
T	T –> FT’			T –> FT’		
T’		T’ –> ε	T’ –> *FT’		T’ –> ε	T’ –> ε
F	F –> id			F –> (E)		