UNIT 1: Introduction to Compilers

1. Compiler

A **compiler** is a software program that converts code written in a **high-level programming language** (source language) into an **equivalent low-level language** (assembly language) to make it executable by a computer. If the source code is **error-free**, it is successfully translated into object code; otherwise, the compiler reports errors with their respective line numbers.

Operations of a Compiler

- **Breaking Source Programs**: Divides the source code into smaller, manageable parts.
- **Symbol Table Creation**: Maintains a table to store variable names, data types, and memory locations.
- **Intermediate Code Representation**: Converts the source code into an intermediate representation for further processing.
- **Code Compilation & Error Detection**: Detects and reports syntax, semantic, and logical errors.
- **Saving Codes and Variables**: Stores program variables and their values for future reference.
- **Program Analysis & Translation**: Reads, analyses, and converts the entire program.
- **Conversion to Assembly Code**: Translates human-readable source code into assembly-code instructions.

Advantages of Compiler Design

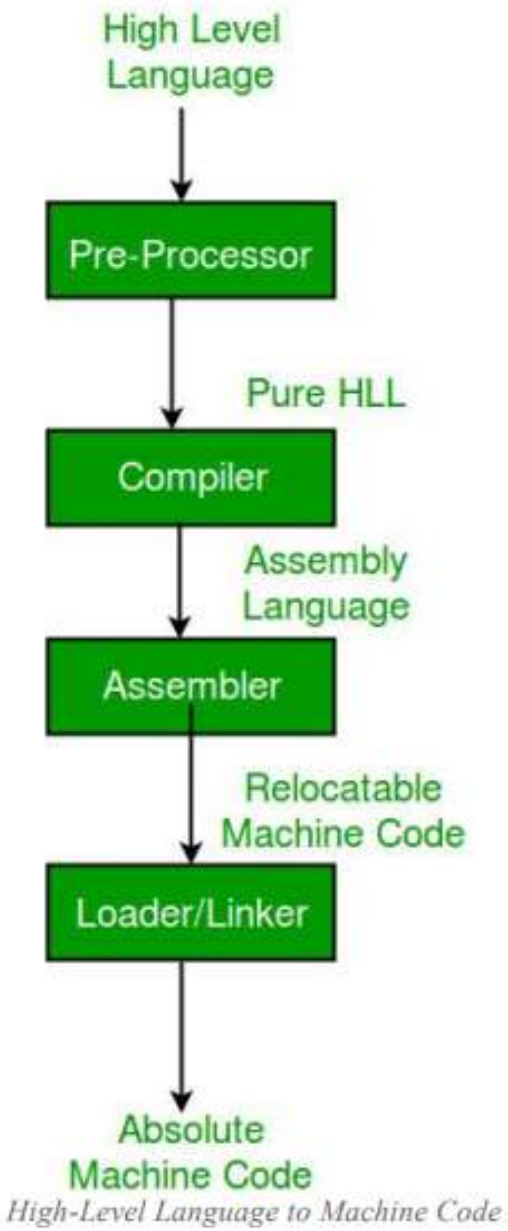
- **Efficiency**: Produces optimized machine code, ensuring faster program execution.
- **Portability**: Compiled programs with compatible hardware and OS can run on any system.
- **Error Checking**: Identifies errors before execution, reducing runtime failures.
- **Optimizations**: Improves performance by optimizing redundant and inefficient code.

Disadvantages of Compiler Design

- **Longer Development Time**: Writing a compiler is complex and requires expertise.
- **Debugging Challenges**: Machine code is hard to understand, making debugging difficult.
- **Lack of Interactivity**: Requires compilation before execution, slowing down development.

2. Language Processing System

A **Language Processing System** converts a **high-level program** into machine-executable code through multiple transformation stages, ensuring correctness, efficiency, and optimization.



High-Level Language to Machine Code

Stages and Process Flow

1. **High-Level Language (HLL) - Writing Source Code**
   - A program is written in a human-readable programming language (e.g., C, Java, Python).
   - Contains **preprocessor directives** like #include and #define.
   - The source code is fed into the **preprocessor**.
2. **Pre-Processor - Preparing the Code**
   - Handles **macro expansion, file inclusion, and language extensions**.
   - Replaces #include directives with the actual file content.
   - Processed code is passed to the **compiler**.
3. **Compiler - Translating to Assembly Language**
   - Converts **high-level code into assembly language**.
   - Performs **lexical analysis, syntax analysis, semantic analysis, and optimization**.
   - Detects errors and reports them.
   - If no errors, the compiler generates an **assembly language program**, which moves to the **assembler**.
   - If errors exist, the compiler stops execution and reports them.
4. **Assembler - Converting to Machine Code**
   - Translates **assembly language** into **machine code (object file)**.
   - Machine code consists of binary instructions that the computer understands.
   - The generated **object file** is passed to the **linker**.
5. **Linker - Merging Object Files**
   - Combines **multiple object files** (compiled code, standard libraries, OS resources).
   - Resolves external references and generates an **executable file**.
   - The **executable file** is passed to the **loader** for execution.
6. **Loader - Executing the Program**
   - Loads the **executable file into memory** and prepares it for execution.
   - Assigns memory locations for code and data.
   - The program starts executing in the computer's CPU.

3. Structure/ Phases of Compiler

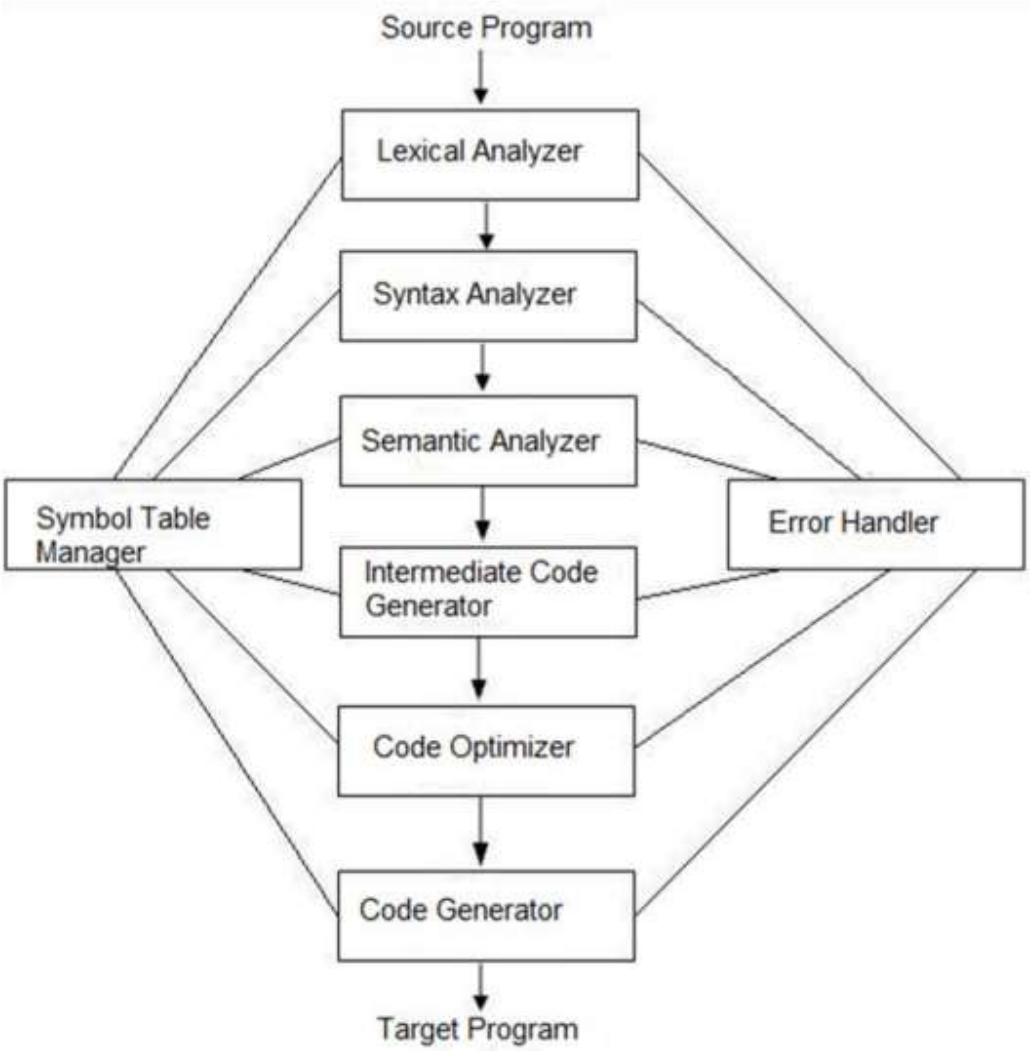
The compilation process consists of **two main parts**:

1. **Analysis Phase (Front-end)**: This phase **analyses** the source program, breaks it into smaller components, and generates an **intermediate representation** (IR).

   - Lexical Analysis
   - Syntax Analysis
   - Semantic Analysis
   - Intermediate Code Generation

2. **Synthesis Phase (Back-end)**: This phase creates an **equivalent target program** from the **intermediate representation.**

   - Code Optimization
   - Code Generation



1. Lexical Analysis (Scanning)

The **lexical analyser (scanner)** is the **first phase** of a compiler. It reads the **source code character by character** and converts it into **tokens**.

Key Functions:

- **Tokenization:** Breaks the input into **tokens** (the smallest meaningful units).
- **Removes Whitespaces & Comments:** Ignores spaces, newlines, and comments.
- **Handles Reserved Words:** Recognizes **keywords**, **identifiers**, **operators**, and **punctuation**.
- **Error Detection:** Detects invalid symbols (e.g., @, # in C code).

Key Definitions:

- **Token:** A sequence of characters forming a meaningful unit (e.g., int, x, +, ;).
- **Lexeme:** The actual character sequence that matches a token.
- **Pattern:** The rule defining how lexemes form tokens.

Example:

For the statement: sum = num1 + num2 * 5;

Token	Type
sum	Identifier
=	operator
num1	Identifier
+	Operator
num2	identifier
*	Operator
5	Constant
;	punctuation

Output: A **stream of tokens** is passed to the **Syntax Analyzer**.

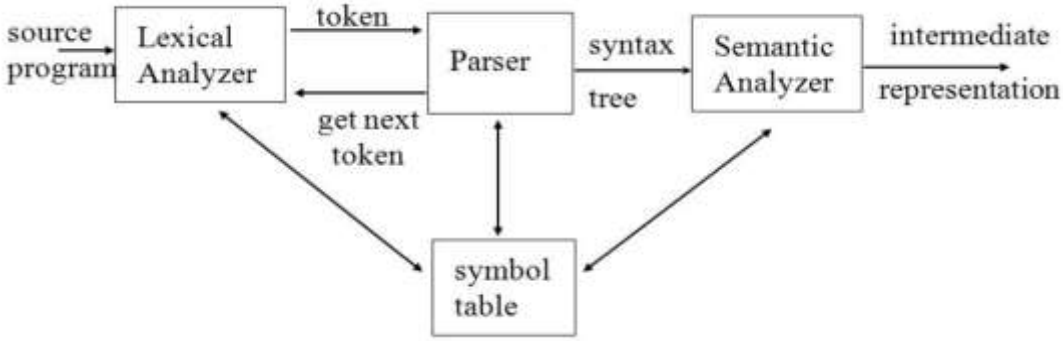
2. Syntax Analysis (Parsing)

The **syntax analyser (parser)** is the **second phase** of the compiler. It **checks whether the tokens follow the syntax rules** of the programming language.

Key Functions:

- **Constructs a Parse Tree:** Organizes tokens into a hierarchical structure.
- **Verifies Syntax Rules:** Ensures correct **parentheses**, **operators**, **loops**, etc.

- **Reports Syntax Errors:** E.g., missing semicolons or unmatched brackets.



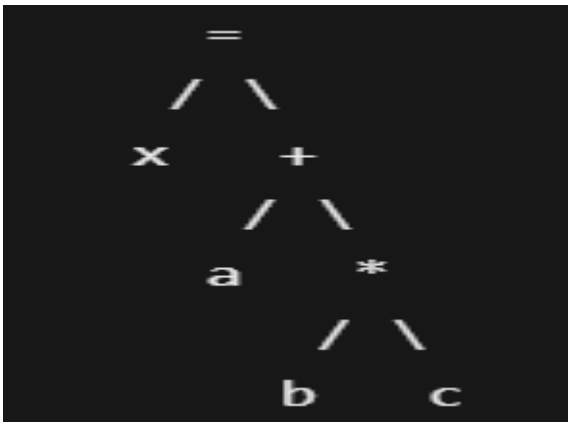
Key Definitions:

- **Parse Tree:** A tree representation of syntax.
- **Grammar:** The set of rules defining the language syntax.
- **Production Rule:** Defines how symbols can be replaced.

Example:

For the statement: x = a + b * c;

The **parse tree** is:



If we write: x = a + * b;

The parser detects an **error** because + * b is **invalid syntax**.

Output: A **Parse Tree** is passed to **Semantic Analysis**.

3. Semantic Analysis

The **semantic analyser** checks for the **meaningful correctness** of statements.

Key Functions:

- **Type Checking:** Ensures data type compatibility (e.g., int x = "hello"; is invalid).

- **Function Parameter Check:** Ensures correct function calls.
- **Variable Scope Checking:** Ensures variables are declared before use.
- **Error Detection:** Detects type mismatches, undeclared variables, etc.

Key Definitions:

- **Type Checking:** Ensures variables and functions use the correct data types.
- **Scope:** The area where a variable is valid.

Example:

```
int x;

float y;

x = y + 2.5;  // Error! Cannot assign float to
```

Output: A **modified syntax tree** with **semantic information**.

4. Intermediate Code Generation

This phase generates an **Intermediate Representation (IR)** of the source code, which is **not machine code**, but a code **easy to analyse and transform**.

Key Functions:

- **Generates Intermediate Code (IR)**: IR is a simplified version of the source code.
- **Platform-Independent**: IR does not depend on the machine architecture. Same for all platforms.
- **Acts as a Bridge:** Connects the compiler's front-end (analysis) and back-end (synthesis).
- **Supports Optimization:** IR is easier to optimize than high-level source code.
- **Facilitates Reusability:** You don't need to build a compiler from scratch. You can reuse IR from existing compilers and just change the back-end for new machines**.**

Key Definitions:

- **Three-Address Code (TAC):** Uses temporary variables to break down expressions.
- **Abstract Syntax Tree (AST):** A simplified tree form of code.

Example:

For x = a + b * c; The IR might be:

```
T1 = b * c

T2 = a + T1

x = T2
```

Output: An **intermediate code representation** for optimization.

5. Code Optimization

The **Code Optimization** phase makes the intermediate code **better** in terms of **execution time**, **memory usage**, or both — **without changing its meaning**.

Key Functions:

- **Improves Efficiency:** Transforms code to make it run faster and use fewer resources.
- **Preserves Semantics:** The meaning of the code remains the same, only the structure is improved.
- **Removes Redundancy:** Eliminates unnecessary or repeated instructions.
- **Simplifies Expressions:** Makes complex operations simpler and faster to execute.

Types of Optimization:

- **Machine Independent Optimization:**
  - Performed **before machine code generation**.
  - Works on the intermediate representation (IR).
  - Example: Removing dead code, constant folding.
- **Machine Dependent Optimization:**
  - Performed **after machine code generation**.
  - Depends on the **specific CPU architecture**.
  - Example: Choosing the fastest available instruction for a given operation.

Key Definitions:

- **Constant Folding:** Replaces constant expressions with their computed values at compile time. *E.g.,* a = 3 + 4; → a = 7;
- **Dead Code Elimination:** Removes code that **doesn't affect** the program output. *E.g.,* unused variables or unreachable statements.
- **Loop Optimization:** Improves the performance of loops. Examples:
  - **Code motion:** Moves constant calculations out of the loop.
  - **Strength reduction:** Replaces expensive operations like * with cheaper ones like +.
- **Peephole Optimization:** Looks at **small sets of instructions** and replaces them with faster equivalents.
- **Loop Unrolling:** Reduces the number of iterations by repeating the loop body multiple times per iteration.

Example:

Before Optimization:

```
for (int i = 0; i < 100; i++) {
    sum = sum + 5 * 10;
}
```

After Optimization:

```
int value = 5 * 10;
for (int i = 0; i < 100; i++) {
    sum = sum + value;
}
```

Output: An **optimized intermediate representation**.

6. Code Generation

This is the **final phase of compilation**, where the **optimized intermediate representation (IR)** is translated into **assembly instructions**.

Key Functions:

- **Instruction Selection:** Converts intermediate code (IR) into assembly-level instructions using the instruction set of the target machine.
- **Register Allocation:** Assigns variables and temporary values to available CPU registers to improve performance**.**
- **Machine Code Generation:** Converts the final optimized code into a machine-understandable format.
- **Assembler Dependent Output:** The generated code depends on the assembler being used (platform-dependent output).

7. Symbol Table Management

A **symbol table** is a **data structure** that stores information about identifiers (variables, functions, objects, etc.) used in the program.

Key Functions of the Symbol Table

- **Stores Information About Identifiers:**

- **Variable Names**
- **Function Names**
- **Data Types**
- **Memory Locations**
- **Scope (Local or Global)**
- **Access Modifiers (Public, Private, Protected)**

- **Ensures Efficient Variable Lookup:** Helps in the **quick identification** of identifiers.
- **Helps in Type Checking:** Ensures **correct usage** of data types.
- **Aids in Optimization:** Helps in **register allocation** and **removing redundant variables**.
- **Supports Code Generation:** Helps in **assigning memory locations**.

Structure of a Symbol Table

Identifier	Type	Scope	Memory Address	Value
x	int	Global	1000	10
sum	float	Local	1020	0.0
main()	function	Global	-	-

Uses in Different Phases:

- **Lexical Analysis:** Stores identifier names.
- **Syntax Analysis:** Helps in parsing variables and functions.
- **Semantic Analysis:** Checks for **type mismatches** and **undeclared variables**.
- **Intermediate Code Generation:** Helps in creating **temporary variables**.
- **Code Optimization:** Helps in **register allocation**.
- **Code Generation:** Assigns memory locations.

8. Error Handling

Errors in a program can appear at different stages of compilation. The compiler should be able to **detect** and **handle** errors **without stopping the compilation process**.

Types of Errors in Compilation

Error Type	Detected In	Example	Meaning
Lexical Error	Lexical Analysis	int @x = 10;	Invalid character @
Syntax Error	Syntax Analysis	if (x > 5	Missing)
Semantic Error	Semantic Analysis	int x = "hello";	Type mismatch (int cannot store a string)
Runtime Error	Code Execution	x = 5 / 0;	Division by zero
Logical Error	No detection	if (x = 5)	Wrong assignment instead of comparison

4. Phases and Passes in a Compiler (Not directly mentioned in syllabus)

What is a Phase?

- A **phase** is a specific stage in the compilation process.
- Each phase performs a well-defined task and passes its output to the next phase.
- The compilation process is divided into several phases for simplicity and modularity.
- **Example**: Lexical Analysis, Syntax Analysis, Semantic Analysis, etc., are all different phases.

What is a Pass?

- A **pass** refers to the compiler's complete traversal of the source program.
- A pass may include one or more phases.
- It reads the source or intermediate code from start to end, processes it, and stores the result.

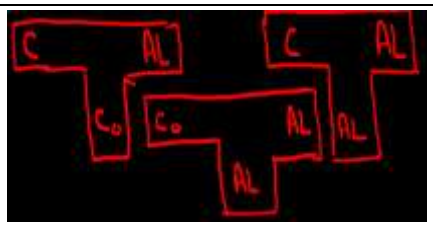
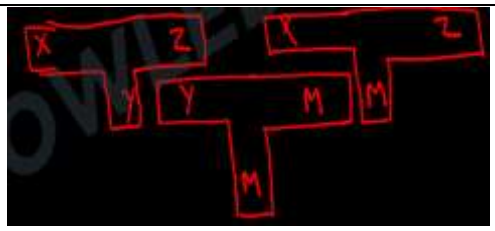
Single-Pass vs Multi-Pass Compilers

Aspect	Single-Pass Compiler	Multi-Pass Compiler
Compilation Speed	**Faster** as it compiles the source code in **one pass**.	**Slower** because it goes through the source code **multiple times**.
Memory Usage	**Less memory-intensive**, doesn't store much intermediate data.	**More memory-intensive**, as it needs to store intermediate representations.
Error Detection	May not detect all errors in the first pass due to limited context.	**Better at error detection**, as it analyses the code in-depth over multiple passes.

Aspect	Single-Pass Compiler	Multi-Pass Compiler
Optimization	**Limited optimization** opportunities because of the single pass nature.	**Better optimization** as it performs advanced transformations in multiple passes.
Language Complexity	Suited for **simpler languages** with less complex syntax and semantics.	**Better for complex languages** with intricate syntax and semantics.
Key Features	- **One-time processing** of the code. - **Faster** compilation. - **Memory efficient**. - **Limited optimization**. - Best for **simpler languages**.	- **Multiple passes** of the source code. - **Advanced optimizations**. - **Context awareness** for better error checking and optimization. - Requires **more memory**.
Examples	Early **FORTRAN Compilers**, **Pascal** (Original Version), **Tiny C Compiler (TCC)**.	Modern compilers like **GCC (GNU Compiler Collection)** and **LLVM**.

5. Bootstrapping vs Cross Compiler (Not directly mentioned in syllabus)

Aspect	Bootstrapping	Cross Compiler
Definition	A process of writing a compiler using the same language that the compiler is meant to compile.	A compiler that runs on one machine (host) but generates code for another machine (target).
Initial Stage	Starts with a simple compiler written in a low-level language (e.g., Assembly or C).	A compiler is created using another language to support development on different machines.
Process	1. Write a small compiler in a different language. 2. Rewrite the compiler in its language. 3. Compile it using the earlier version. 4. Repeat this to improve the compiler.	1. Write the compiler on a host machine. 2. Generate code for a different target machine. 3. Transfer compiled code to the target for execution.
Purpose	To make the compiler self-hosting and self-compiling.	To support development for different platforms that cannot compile code directly.
Key Idea	The compiler is written in the **same** language it is compiling.	The compiler compiles code for a **different** machine than the one it runs on.

Aspect	Bootstrapping	Cross Compiler
Common Use Cases	- Compiler development - Language improvement - Self-hosting compiler creation	- Embedded systems - Game consoles - IoT devices - OS development
Advantages	- Deep understanding of language features - Easier maintenance and upgrades - Language is self-supported	- Target systems don't need their compilers - Faster development on powerful machines - Enables multi-platform support
Disadvantages	- Time-consuming and complex process - Debugging is difficult if the compiler has bugs	- Needs accurate setup of target machine architecture - Toolchain complexity
Example		
Main Focus	Compiler building and language bootstrapping.	Platform independence and embedded system development.
Dependency	Needs an initial version written in another language.	Needs a working compiler for the host machine.

6. Functions and Roles of the Lexical Phase

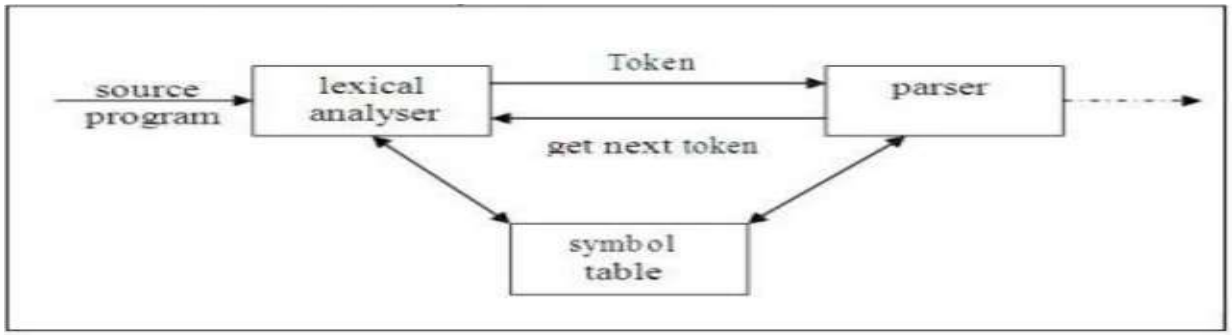
The **lexical analysis phase** is the first phase of a compiler, responsible for converting the raw source code into meaningful tokens that the parser can understand. It acts as a bridge between the source program and the syntactic analysis phase.

Key Functions of Lexical Analysis

- **Reading Source Code** – Scans the program character by character.
- **Token Generation** – Groups characters into meaningful tokens.
- **Skipping White Spaces and Comments** – Ignores unnecessary spaces and comments.
- **Error Detection and Reporting** – Identifies and reports lexical errors with file and line number.
- **Providing Tokens to the Parser** – Supplies tokens when requested.

Interaction of Lexical Analyzer with Parser

- The parser requires structured **tokens** instead of raw characters.
- The lexical analyser scans the input and provides tokens.
- A **symbol table** stores identifiers, keywords, and other symbols.



- **Simpler Compiler Design** – Separating tokenization simplifies parsing.
- **Improved Efficiency** – Buffering techniques speed up token processing.
- **Portability** – The lexical analyser can be reused across different languages.

Tokens, Patterns, and Lexemes

- **Token:** A category of words in a programming language. Examples include:
  - **Keywords** (e.g., if, while, return)
  - **Operators** (e.g., +, -, *, /)
  - **Identifiers** (e.g., variable names, function names)
  - **Constants** (e.g., numeric and string literals)
  - **Punctuation** (e.g., ;, {}, ())
- **Pattern:** A rule defining how a valid token is structured (e.g., an identifier starts with a letter and may contain numbers).
- **Lexeme:** The actual sequence of characters that matches a token pattern in the source code.

Attributes of Tokens

- Every token has associated **attributes** that provide additional details for translation.
- These attributes help in symbol table management and code generation.
- Example:
  - An identifier token may have an attribute linking to its **symbol table entry**, where information like data type and memory location is stored

Lexical Errors and Error Recovery

The lexical analyser detects and handles errors that arise due to invalid characters or unexpected sequences.

Common Lexical Errors

1. **Illegal characters:** Using symbols that don't belong to the language (e.g., @ in a C program).
2. **Unterminated string literals:** A missing closing quote in printf("Hello; will cause an error.
3. **Invalid number format:** Writing numbers incorrectly (e.g., 12.3.4 is not a valid float).
4. **Misspelled keywords:** Writing inte instead of int will not be recognized as a valid keyword.

Error Recovery Strategies

1. **Deleting extra characters:** Removing unwanted symbols to allow parsing to continue.
2. **Inserting missing characters:** Adding a missing quote, semicolon, or operator.
3. **Replacing incorrect characters:** Correcting an illegal symbol with a valid one.
4. **Swapping adjacent characters:** Fixing mistyped words like itn instead of int.

7. Input Buffering

In compiler design, **input buffering** is used to efficiently scan and process the source code. The **lexical analyser** reads the input character by character, which can be slow and inefficient. Input buffering improves performance by reading larger chunks of data at once, reducing system calls and processing time.

Need for Input Buffering

- **Slow character-by-character reading** can be inefficient, especially for large programs**.**
- **System calls** for each character are time-consuming.
- **Input buffering** reduces the overhead by **bulk-reading characters**, making the process faster and more efficient.

Working of Input Buffering

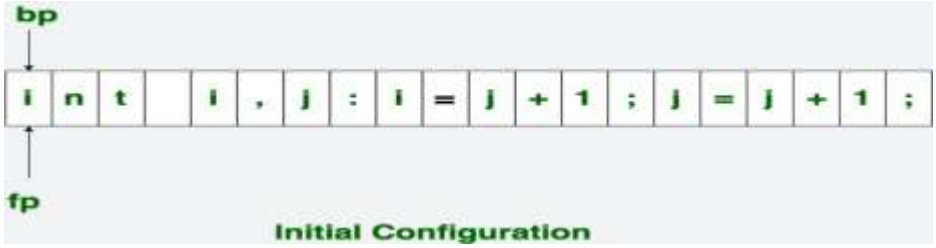
Pointers Used

- **Begin Pointer (bp)**: Marks the start of the token.

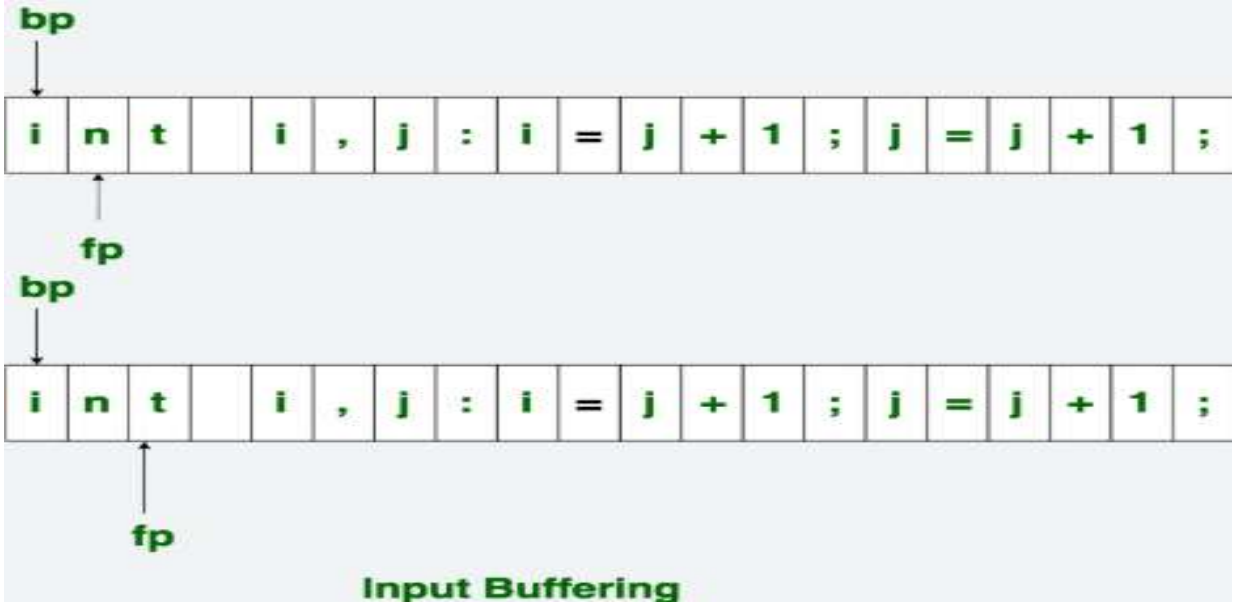
- **Forward Pointer (fp)**: Moves ahead to find the end of the token.

How It Works:

1. Initially, both bp and fp point to the first character.



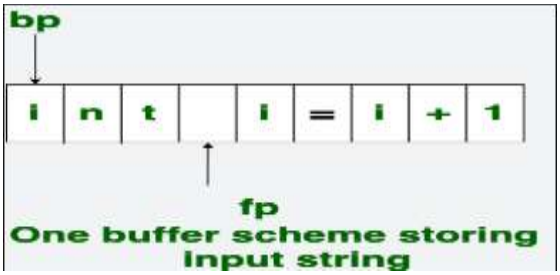
2. fp moves forward until it encounters a **delimiter** (like whitespace).
3. The string between bp and fp forms a **lexeme** (token).
4. The fp then **skips whitespace** and moves to the next token.
5. The bp is updated to the position of fp, and the process repeats.



Buffering Techniques

1. One-Buffer Scheme

- Uses **a single buffer** to store input characters.
- If a **long lexeme crosses the buffer boundary**, the buffer must be refilled.



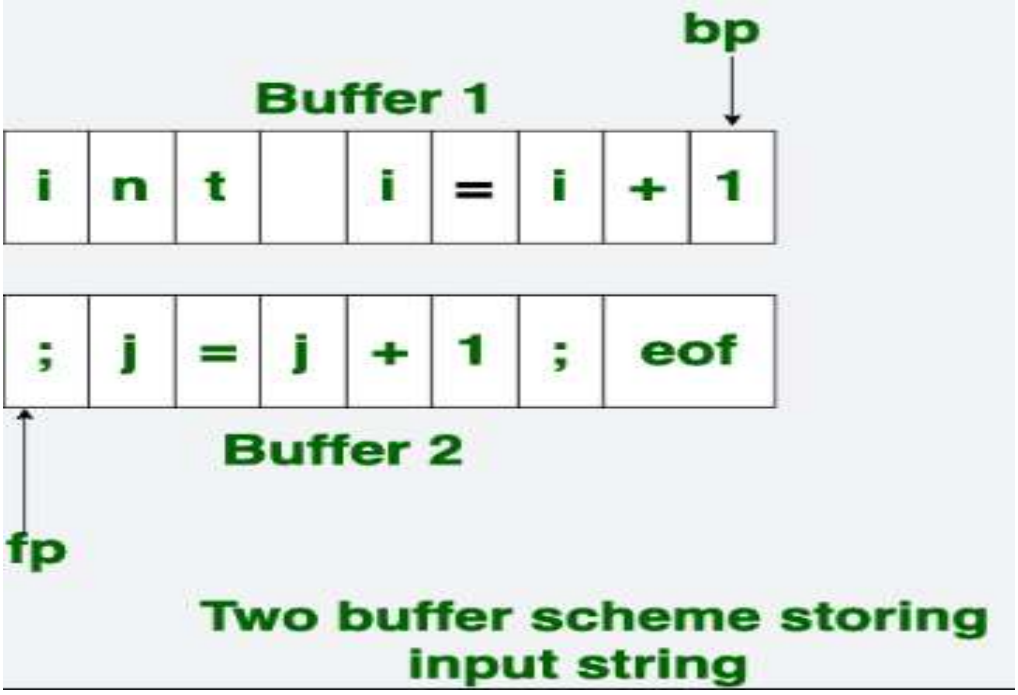
- **Problem**: Overwriting occurs when refilling the buffer, leading to **loss of data**.
- **Not efficient** for handling large lexemes.

2. Two-Buffer Scheme

To solve the problem of **overwriting in the one-buffer scheme**, a **two-buffer approach** is used.

Working of the Two-Buffer Scheme

- The input buffer is divided into **two halves** of size N.
- Both halves are **filled alternately** with input characters.



- If fp reaches the end of the first buffer, the **second buffer** is filled, and reading continues.
- If fp reaches the end of the second buffer, the **first buffer** is refilled again.
- This process continues until the **entire source code is processed**.

Sentinel Character in Buffering

A **sentinel** (special EOF character) is placed at the end of each buffer half to:

- **Mark the buffer end** and **avoid repeated boundary checks**.
- **Indicate when to refill** the buffer efficiently.

Using sentinels, the compiler only **checks if fp points to EOF** instead of performing multiple boundary checks, improving efficiency.

Example:

Let's take a simple source code input:

```
int a = 10;
```

Buffer Size: 6 characters per buffer (for simplicity).

Step 1: Load Buffers

- **Buffer 1** will contain:

```
[i][n][t][ ][a][ ]
```

Sentinel at the end:

```
[i][n][t][ ][a][ ] [$]
```

- **Buffer 2** will contain:

```
[=][ ][1][0][;][\n]
```

Sentinel at the end:

```
[=][ ][1][0][;][\n] [$]
```

Step 2: Pointer Movement

Step	fp Char	Action	Notes
1	**i**	fp moves forward	First character in Buffer 1.
2	**n**	fp moves forward	
3	**t**	fp moves forward	
4	**' '**	Token: int (bp to fp-1)	Lexeme formed: int
5	**a**	fp moves forward	Next token
6	**' '**	Token: a (bp to fp-1)	Lexeme formed: a
7	**=**	fp moves forward	Next token
8	**' '**	Token: = (bp to fp-1)	Lexeme formed: =

Step	fp Char	Action	Notes
9	**1**	fp moves forward	
10	**0**	fp moves forward	
11	**;**	fp moves forward	
12	**\n**	Token: 10; (bp to fp-1)	Lexeme formed: 10;

Advantages and Disadvantages of Input Buffering

Feature	Advantages	Disadvantages
Performance	Reduces system calls, improving efficiency.	A large buffer size can consume too much memory.
Ease of Implementation	Simplifies compiler design by reducing input management code.	If not managed properly, it can lead to errors.
Token Recognition	Enables fast and efficient token processing.	If lexemes are longer than the buffer size, scanning issues may arise.

8. Specification of Tokens

In compiler design, a **token** is a fundamental unit of meaning recognized by the lexical analyser. The specification of tokens is defined using three key components:

1. Strings

A **string** is a finite sequence of symbols (characters). These symbols can be alphabets, digits, or special characters. There is also an **empty string (ε)**, which represents no characters.

Operations on Strings

- **Prefix**: A prefix of a string is obtained by removing zero or more characters from the end.
    - Example: If **S = "NINJA"**, a prefix can be **"NIN"** (by removing "JA").

    - **Proper Prefix**: A prefix that is neither the string itself nor empty (ε).
- **Suffix**: A suffix of a string is obtained by removing zero or more characters from the beginning.
    - Example: If **S = "NINJA"**, a suffix can be **"JA"** (by removing "NIN").
    - **Proper Suffix**: A suffix that is neither the string itself nor empty (ε).
- **Substring**: A substring is obtained by removing a prefix and a suffix.
    - Example: If **S = "AYUSHI"**, a substring can be **"US"** (removing "AY" and "HI").
    - **Proper Substring**: A substring that is neither the string itself nor empty (ε).
- **Subsequence**: A subsequence is obtained by removing zero or more characters from anywhere in the string (not necessarily consecutive).
    - Example: If **S = "NINJAMANIA"**, a subsequence can be **"NIJAANIA"**.
- **Concatenation**: The operation of joining two strings together.
    - Example: If **S = "Cod"** and **T = "ing"**, then **S + T = "Coding"**.

2. Language

A **language** is a set of strings formed using a given set of symbols or alphabets.

Operations on Languages

Languages support various operations in lexical analysis:

- **Union** (**L ∪ S**): The union of two languages **L** and **S** includes all strings from both.
    - Example: If **L = {a, b}** and **S = {c, d}**, then **L ∪ S = {a, b, c, d}**.
- **Concatenation** (**LS**): The concatenation of two languages **L** and **S** forms new strings by combining elements of each.
    - Example: If **L = {L', L"}** and **S = {S', S"}**, then **LS = {L'S', L'S", L''S, L''S"}**.
- **Kleene Closure** (**L***): It includes all possible concatenations of **L**, including the empty string **ε**.
    - Example: If **L = {a, b}**, then **L* = {ε, a, b, aa, bb, aaa, bbb, …}**.
- **Positive Closure** (**L+**): It is similar to Kleene closure but does not include the empty string **ε**.
    - Example: If **L = {a, b}**, then **L+ = {a, b, aa, bb, aaa, bbb, …}**.

3. Regular Expressions

A **regular expression** is a pattern used to define a language. It consists of characters and symbols to form valid strings.

Regular Expression Symbols

- **Asterisk (*)**: Matches zero or more repetitions of the preceding character.
  - Example: **ab*c** → Strings: **ac, abc, abbc, abbbc, …**
- **Plus (+):** Matches one or more repetitions of the preceding character.
  - Example: **ab+c** → Strings: **abc, abbc, abbbc, …**
- **Wildcard (.)**: Matches any single character.
  - Example: **a.c** → Matches **"abc", "axc", "a@c"**, etc.
- **Character Class [ ]**: Defines a set of allowed characters.
  - Example: **[a-z]** matches any lowercase letter from 'a to z'.

Rules for Regular Expressions

- **ε** is a regular expression denoting the language **L(ε) = {ε}**.
- **A single character 'a'** in Σ is a regular expression denoting **L(a) = {a}**.
- If **r** and **s** are regular expressions, then:
  - **r | s** denotes **L(r) ∪ L(s)** (Union).
  - **(r)(s)** denotes **L(r) · L(s)** (Concatenation).
  - *(r) denotes (L(r))* (Kleene closure).**
  - **(r)+ denotes (L(r))+ (Positive closure).**

9. Recognition of Tokens

In a **lexical analyzer**, **tokens** are recognized through a series of steps, typically involving **pattern matching** against predefined regular expressions and converting those patterns into an executable form, like a **Finite Automaton** (FA). The basic process of recognizing tokens involves scanning the input source code and classifying sequences of characters into categories known as **tokens**.

Steps to Recognize Tokens in a Lexical Analyzer:

1. Input Scanning

- The lexical analyser takes the **source code** as input and **scans** it character by character or in chunks (using input buffering).
- The input is processed sequentially, starting from the beginning of the source code.

2. Pattern Matching with Regular Expressions

- Each **token** in the source code is defined by a **regular expression**. Regular expressions describe the structure of the token.
    - For example:
        - **Keywords**: if, else, and while are typically recognized by exact matches.
        - **Identifiers**: A sequence starting with a letter (e.g., a, b, sum, etc.).
        - **Operators**: +, -, =, *, etc.
- **Patterns** are defined for each type of token (keywords, operators, identifiers, literals, etc.).

- Example:
    - **Identifier Pattern**: [a-zA-Z][a-zA-Z0-9]* (starts with a letter, followed by any number of letters or digits).
    - **Keyword Pattern**: if| else | while (match exactly one of these keywords).

3. Finite Automaton (FA) for Token Recognition

- A **Finite Automaton (FA)**, either **Nondeterministic (NFA)** or **Deterministic (DFA)**, is used to implement the regular expressions.
- The lexical analyser converts the regular expressions into a **finite state machine (FSM)** that can match tokens efficiently.
    - **NFA** is constructed first from the regular expression.
    - Then, it is converted into a **DFA**, which is more efficient for runtime matching.
- **Example**: The regular expression for identifiers ([a-zA-Z][a-zA-Z0-9]*) can be turned into an NFA. This NFA is then converted into a DFA to recognize valid identifiers.

4. Token Classification

- As the lexical analyser scans the input, it uses the **automaton** to **match** sequences of characters against the predefined patterns.
- When a match is found, a **token** is generated.
    - For example, if the string int is matched by the **keyword pattern**, the token (keyword, int) is generated.
    - If sum123 matches the identifier pattern, the token (identifier, sum123) is generated.

5. Handling Special Characters and Whitespace

- **Whitespace** (spaces, tabs, newlines) is typically **ignored** by the lexical analyser, as it doesn't contribute to the meaning of the program.
- **Special characters** like parentheses (), semicolons ;, or commas , are recognized as **separate tokens**.
- These are classified as **punctuation tokens**.

6. Tokenization Example

- Input source code: int sum = 10;
- Step-by-step:
    1. **'i'**: Matches the first part of an identifier, so the automaton moves into a state that expects an identifier.
    2. **'n'**: Matches the next part of the identifier.
    3. **'t'**: Completes the identifier int. The token (keyword, int) is generated.
    4. **' '** (whitespace): Ignored.
    5. **'s'**: Starts a new identifier sum, and the token (identifier, sum) is generated.
    6. **'u'**: Matches the next part of the identifier sum.
    7. **'m'**: Completes the identifier sum. The token (identifier, sum) is generated.
    8. **' '** (whitespace): Ignored.
    9. **'='**: Matches the operator pattern. The token (operator, =) is generated.
    10. **' '** (whitespace): Ignored.
    11. **'1'**: Starts a number literal.
    12. **'0'**: Completes the number literal 10. The token (literal, 10) is generated.
    13. **';'**: Matches the punctuation pattern. The token (punctuation, ;) is generated.

Tokens Recognized: (keyword, int), (identifier, sum), (operator, =), (literal, 10), (punctuation,;)

7. Error Handling

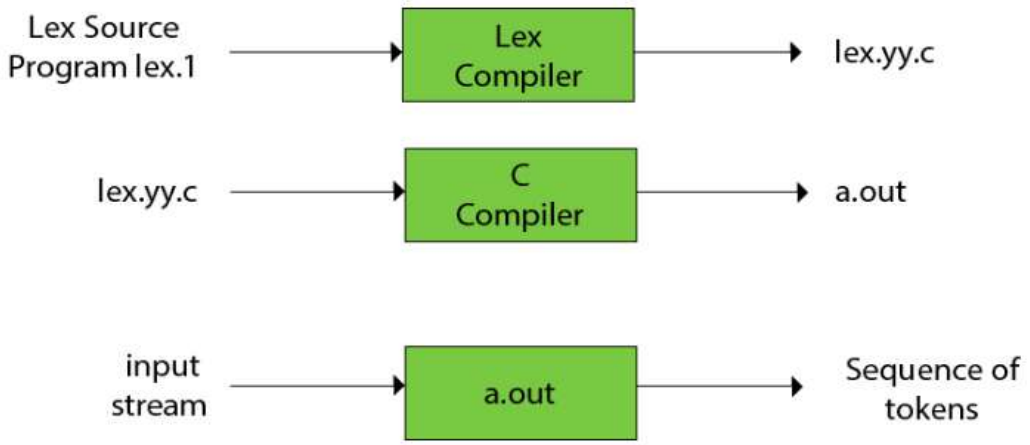
- If the lexical analyser encounters characters that do not match any token patterns, it **reports an error**.
- For example, if a character such as @ appears in a language that doesn't support it, the lexical analyser will flag it as an **invalid character**.

10. LEX (Lexical Analyzer Generator)

- **LEX** is a tool used to generate a lexical analyser (lexer) from given input rules.
- It is commonly used with **YACC (Yet Another Compiler Compiler)** to build compilers.
- The lexer reads an input stream and converts it into a sequence of **tokens**.
- It is mainly used in **compiler design** to identify meaningful components (tokens) from the source code.

Working of LEX

- LEX processes an input file containing lexical rules and generates a C program for token recognition.
- The process follows these steps:
    1. **Writing the Lex Program**: Create a Lex file (lex.l) containing token definitions.
    2. **Lex Compiler Execution**: The Lex compiler processes lex.l and generates a C program (lex.yy.c).
    3. **C Compilation**: The C compiler compiles lex.yy.c into an executable (a.out).
    4. **Lexical Analysis Execution**: Running a.out processes an input stream and outputs identified tokens.



Lex File Format

- A Lex file consists of **three sections**, separated by %%:
    1. **Definitions**: Declare constants, variables, and regular expressions.
    2. **Rules**: Define token patterns and associated actions.
    3. **User Subroutines**: Additional functions required by actions.

```
{ definitions }
%%
{ rules }
%%
{ user subroutines }
```

Explanation of Each Section

(a) Definitions Section

- Includes:
  - **Constants** (e.g., #define PI 3.14)
  - **Global Variables**
  - **Regular Definitions** (e.g., digit [0-9])

(b) Rules Section

- Specifies **patterns** and corresponding **actions**.
- The format: pattern { action }

(c) User Subroutines Section

- Contains helper functions used in the rules section.

Working of the Lexical Analyzer

1. The lexer reads input character by character.
2. It checks if any **pattern** from the rules section matches.
3. If a match is found, it performs the **specified action**.
4. It continues until the entire input is processed.

Features of LEX

- Supports **regular expressions** for pattern matching.
- Generates **efficient C-based** lexical analysers.
- Works seamlessly with **YACC** for full parser implementation.
- Can handle **whitespace and special characters**.

Advantages of LEX

- **Automates lexical analysis** instead of writing it manually.

- **Easy integration** with compilers and parsers.
- **Fast processing** due to compiled C output.
- **Customizable token actions** through C functions.

Limitations of LEX

- Cannot handle **complex syntax analysis** (requires YACC).
- Not suitable for **context-sensitive grammars**.
- Debugging can be difficult due to the **generated C code complexity**.

NOTE: FINITE AUTOMATA, REGULAR EXPRESSIONS TO AUTOMATA, AND MINIMIZING DFA ARE COVERED IN FLAT.

UNIT 2: Syntax Analysis

1. Role of Parser (Mentioned on page 5 in syntax analysis)

2. Grammar

A formal grammar is defined by a 4-tuple: G = (V_N , Σ , P , S)

1. V_N– Variables (Non-terminals) - A finite, non-empty set of symbols used to build strings.

2. Σ– Terminals - A finite, non-empty set of actual symbols of the language. These are the final characters of generated strings.

3. S – Start Symbol - A special symbol from V_N from which derivations begin.

4. P – Production Rules - A finite set of rules to replace symbols. Each rule is of the form: α→β

- **α** must have at least one non-terminal.
- **β** can be any string of terminals and/or non-terminals.

Example:

Let

- $V_N = \{S, A\}$
- $\Sigma = \{a, b\}$
- $S = S$
- $P = \{S \to aA, A \to Sb, S \to ab\}$

Derivation:

- $S \to aA \to aSb \to aabb$

3. Context Free Grammar

CFG is a type-2 grammar in the Chomsky Hierarchy. It generates Context-Free Languages. These languages are accepted by a Push Down Automata (PDA) (non-deterministic PDA by default).

Production Format

- Rule: **A → β**
    - **A**: A single variable (non-terminal)
    - **β**: Any combination of variables and terminals

- LHS has only one non-terminal — no left or right context.
- Hence, it is called context-free.

Examples

- Languages: ALGOL-60, PASCAL
- Example CFG: S → a S b | a b S

  S → a S b → a a S b b → a a a b b b

Capabilities of CFG

1. Describes Programming Language Syntax - CFGs are used to define the structure of programming languages. They organize elements like Statements**,** Expressions, **and** Declarations. Example: if (condition) { block }

2. Helps in Efficient Parser Construction -CFGs make it easy to automatically generate parsers. Parsers are used in compilers to analyze and understand code. Tools like YACC, Bison use CFG rules to build parsers.

3. Handles Associativity and Precedence - CFG can manage how operators (like +, *) should behave: Associativity: Left-to-right or right-to-left evaluation, **and** Precedence: Order of operations. Ensures expressions like a + b * c is correctly interpreted.

4. Supports Nested Structures - CFG is very good at handling nested or recursive structures: Balanced parentheses: (( )), Matching blocks: begin...end, {...}, **and** Nested control statements: if...else, loops inside loops**.** This is very important in real-world programming languages.

4. Applications of Formal Grammar in Syntax Analysis

1. **Defines Programming Language Syntax** - Specifies how valid code should be written.
2. **Used by Syntax Analyzer (Parser)** - Checks if code follows grammar rules.
3. **Helps in Building Parse Trees** - Shows the structure of the code for further processing.
4. **Detects Syntax Errors** - Reports mistakes in code structure.
5. **Used in Parser Generators** - Tools like YACC and ANTLR use grammar to generate parsers.
6. **Aids in Language Design** - Helps in designing new programming languages.
7. **Supports IDE Features** - Enables syntax highlighting, error hints, and auto-complete.

5. Basic Definitions

1. Derivation - A derivation is the process of generating a string by repeatedly applying production rules, starting from the start symbol, until only terminals remain.

Types of Derivations

1. Leftmost Derivation (LMD) - In each step, the **leftmost** non-terminal is replaced first. Used in **top-down parsers**.

2. Rightmost Derivation (RMD)- In each step, the **rightmost** non-terminal is replaced first. Used in **bottom-up parsers**.

Example:

1. CFG : X → X + X | X * X | a , String = a + a * a

LMD :

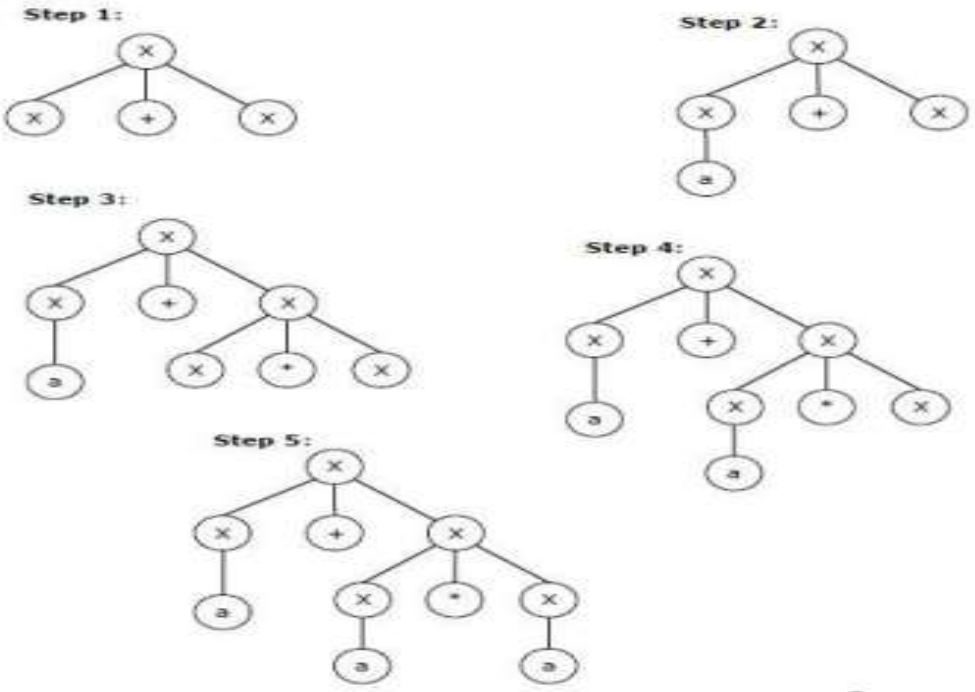
X → X + X

X → a + X

X→ a + X * X

X→ a + a * X

X→ a + a * a



RMD:

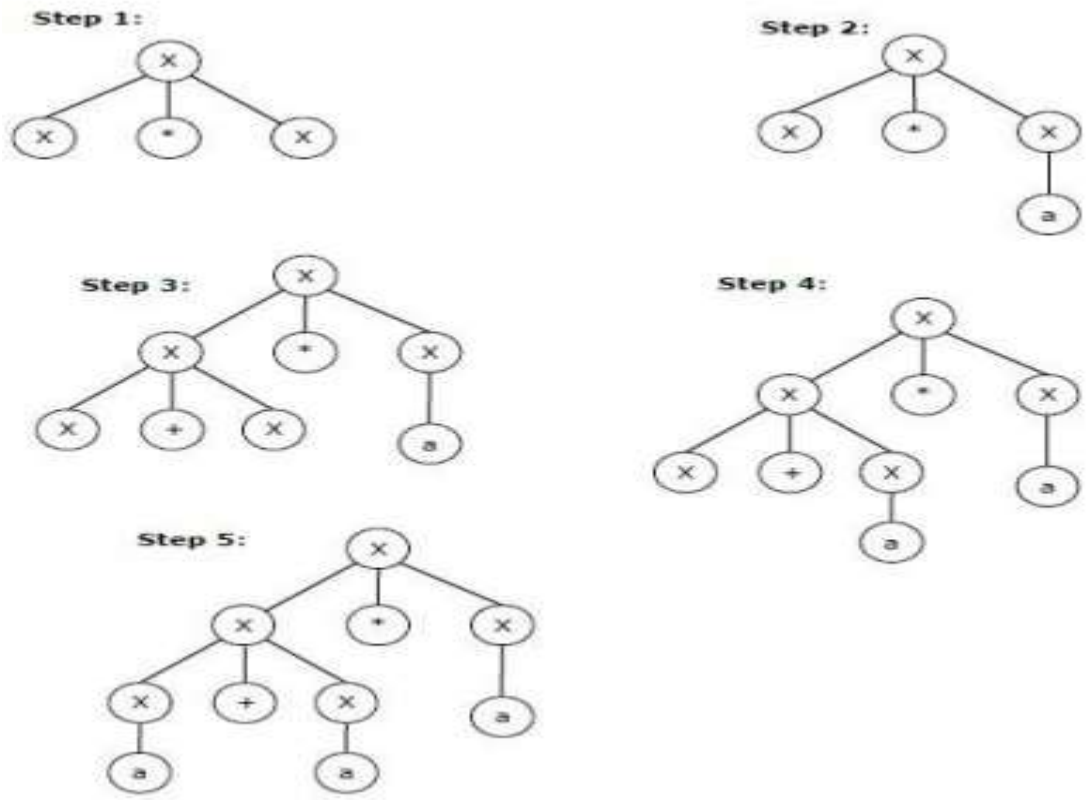
X → X * X

X→ X * a

X→ X + X * a

X→ X + a * a

X→ a + a * a



2. CFG : S → a B | b A , A → a S | b A A | a , B → b S | a B B | b , String = aaabbabbba

LMD:

S → a B

S→ aaBB (B → aBB)

S→ aaaBBB (B → aBB)

S→ aaabBB (B → b)

S→ aaabbB (B → b)

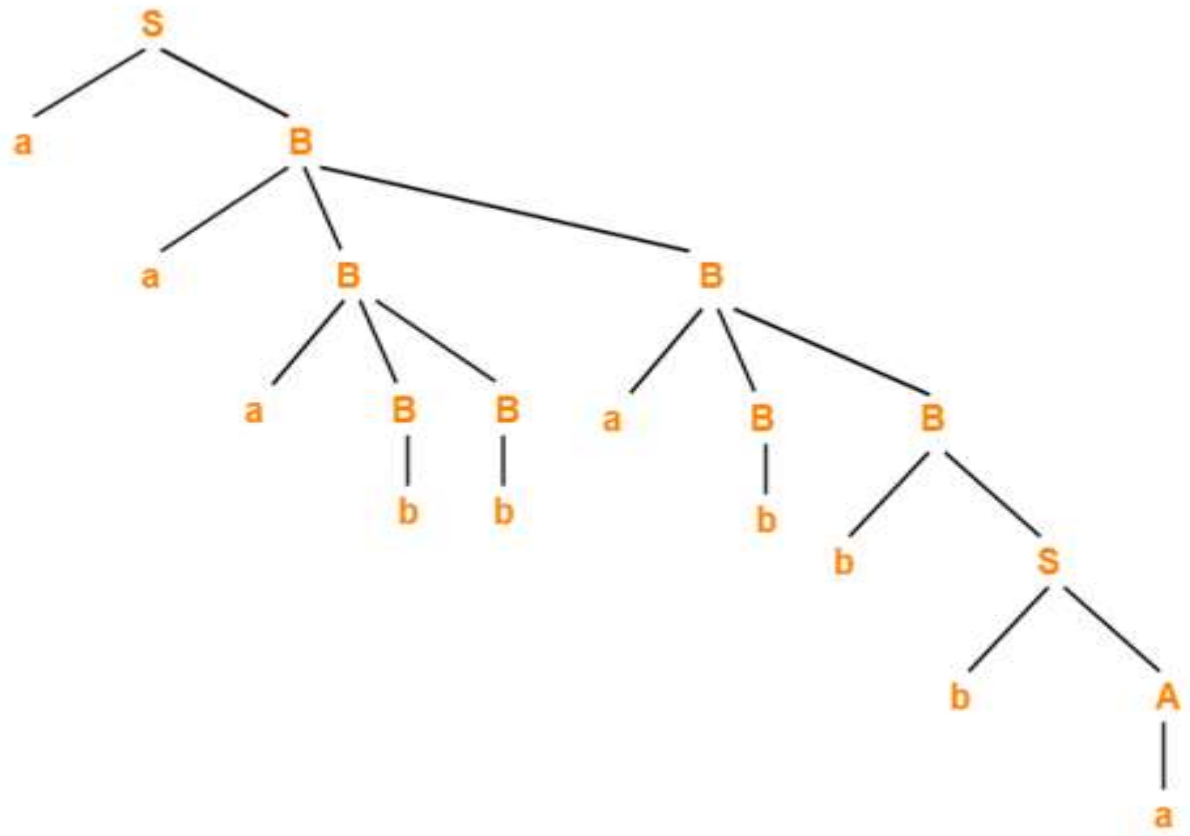
S→ aaabbaBB (B → aBB)

S→ aaabbabB (B → b)

S→ aaabbabbS (B → bS)

S→ aaabbabbbA (S → bA)

S→ aaabbabbba (A → a)



Leftmost Derivation Tree

RMD:

S → aB

S→ aaBB (B → aBB)

S→ aaBaBB (B → aBB)

S→ aaBaBbS (B → bS)

S→ aaBaBbbA (S → bA)

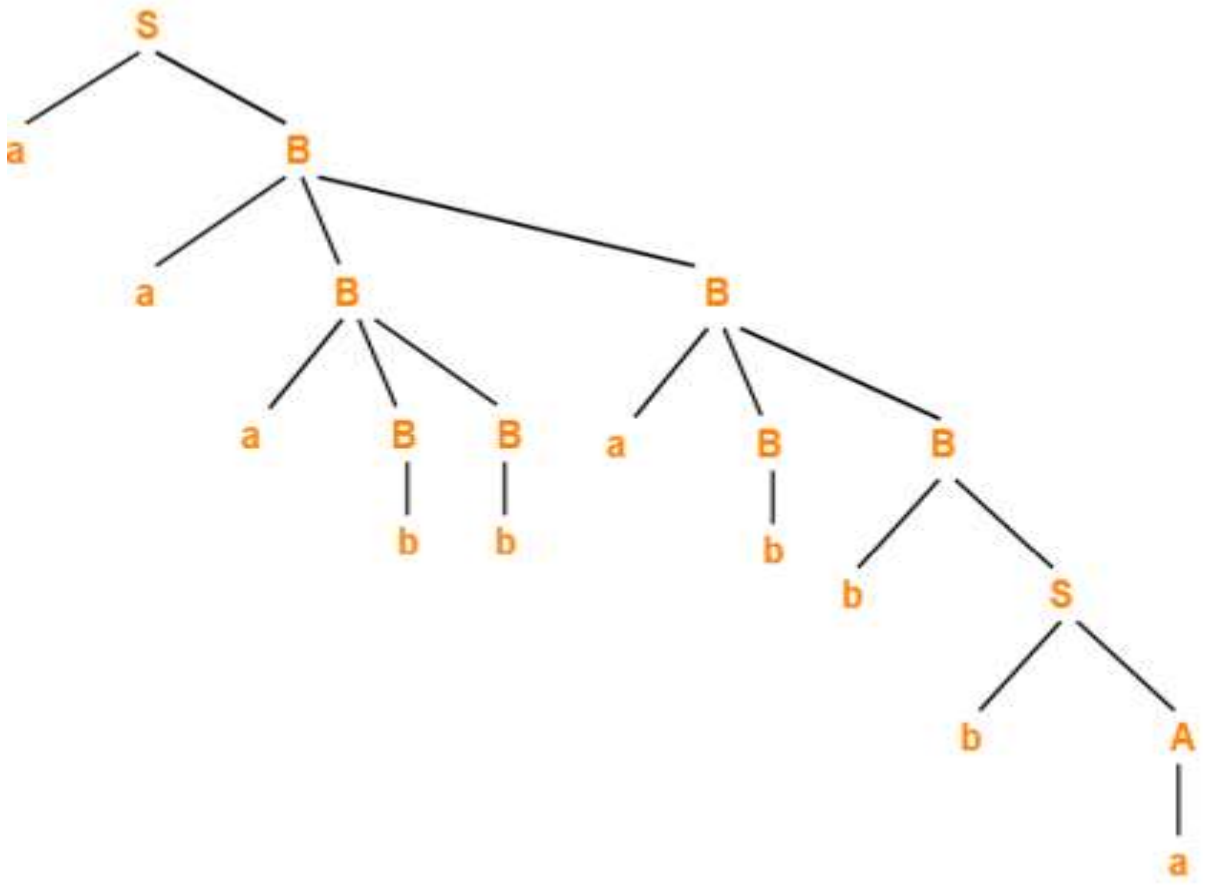
S→ aaBaBbba (A → a)

S→ aaBabbba (B → b)

S→ aaaBBabbba (B → aBB)

S→ aaaBbabbba (B → b)

S → aaabbabbba (B → b)

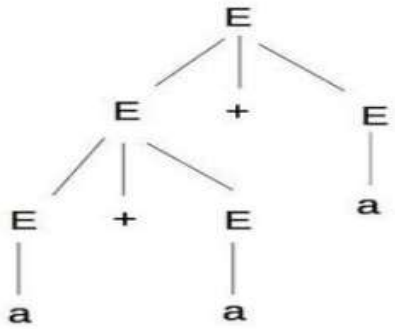


Rightmost Derivation Tree

2. Parse /Syntax / Derivation Tree - It is a **diagram** showing how a string is derived from the **start symbol** using grammar rules. Reading leaves **left to right** gives the **final string** (called **yield**).

- **Root node** → Start symbol
- **Internal nodes** → Non-terminals
- **Leaf nodes** → Terminals

Example: Grammar: E → E + E | E * E | a , **String:** a + a + a



3. Non-Deterministic Grammar - A grammar is **non-deterministic** if two or more productions for a non-terminal start with the same prefix.

Form: $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$. This causes backtracking during parsing, which is slow and inefficient.

Parsers (especially top-down parsers) cannot decide which rule to choose just by looking at the next input symbol. So, we need to convert a non-deterministic grammar to a deterministic one.

4. Left Factoring - Left Factoring is the method used to remove common prefixes and make the grammar deterministic.

Before (Non-Deterministic): $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$

After (Deterministic using Left Factoring): $A \rightarrow \alpha A'$, $A' \rightarrow \beta_1 \mid \beta_2$

This helps the parser choose a rule without backtracking.

5. Ambiguous Grammar - A grammar is ambiguous if a single string can have: More than one parse tree, or More than one Leftmost Derivation (LMD), or More than one Rightmost Derivation (RMD)

Example - E → E + E / E x E / id , String = id + id x id

E → **E** + E
→ id + **E**
→ id + **E** x E
→ id + id x **E**
→ id + id x id

Leftmost Derivation-01

E → **E** x E
→ **E** + E x E
→ id + **E** x E
→ id + id x **E**
→ id + id x id

Leftmost Derivation-02

E → E + **E**
→ E + E x **E**
→ E + **E** x id
→ **E** + id x id
→ id + id x id

Rightmost Derivation-01

E → E x **E**
→ **E** x id
→ E + **E** x id
→ **E** + id x id
→ id + id x id

Rightmost Derivation-02

6. Unambiguous Grammar - A grammar is said to be unambiguous if every string generated by it has: Exactly one Parse Tree, exactly one Leftmost Derivation (LMD), and exactly one Rightmost Derivation (RMD).

Example - X → AB , A → Aa | a , B → b

7. Recursive Grammar - A grammar is recursive if it contains at least one recursive production.

Examples: S → aS | a , S → Sa | a and S → aSb | ab

Types of Recursion:

1. Left Recursive Grammar - The leftmost symbol on the RHS is the same as the LHS variable. Example: A → Aα

2. Right Recursive Grammar - The rightmost symbol on the RHS is the same as the LHS variable. Example: A → αA

How to Eliminate Left Recursion?

Convert: A → Aα | β **To**: A → βA' , A' → αA' | ε

6. Error Handling and Recovery Strategies

Error handling is the process of detecting, reporting, and recovering from errors in a program during compilation. The goal is to help programmers find and fix mistakes without stopping the compilation completely. A good compiler should continue analysing the rest of the code even after finding errors.

Types of Errors in Compilation

1. Lexical Errors - Related to the incorrect formation of tokens. Example: Using an invalid variable name like 9var.

2. Syntax Errors - Occur when the structure of a statement does not follow grammar rules. Example: Missing parentheses in if (x > 5 {.

3. Semantic Errors - Related to the meaning of a statement being incorrect. Example: Assigning a string to an integer variable.

4. Logical Errors - The Program runs but gives incorrect output. Example: Using the wrong formula for calculations.

Syntax Errors (Handled by Parser)

Syntax errors are specifically handled during the syntax analysis phase. The parser's job is to detect these errors, report them, and try to recover so it can continue parsing the rest of the code.

Goals of Syntax Error Handling

1. Detect errors as early as possible.
2. Report errors clearly with useful messages.
3. Recover and continue parsing the remaining code.
4. Avoid reporting multiple errors for a single mistake.

Error Recovery Strategies

1. Panic Mode Recovery - Skips tokens until a synchronizing token (like; }, etc.) is found. **It is** simple and fast. May skip large portions of code. **It's** commonly used in practical compilers.

2. Phrase-Level Recovery - Attempts to correct errors by inserting, deleting, or replacing tokens. **It** focuses on small, local fixes. Helps the parser continue parsing without skipping large sections. Example: Automatically inserting a missing semicolon.

3. Error Productions - Special grammar rules are added to handle common errors. Helps in giving specific and meaningful error messages. Useful for catching known patterns of user mistakes.

4. Global Correction - Tries to make the smallest possible change to the input to fix errors. Most accurate but computationally expensive. Not practical for real-world compilers.

Error Reporting

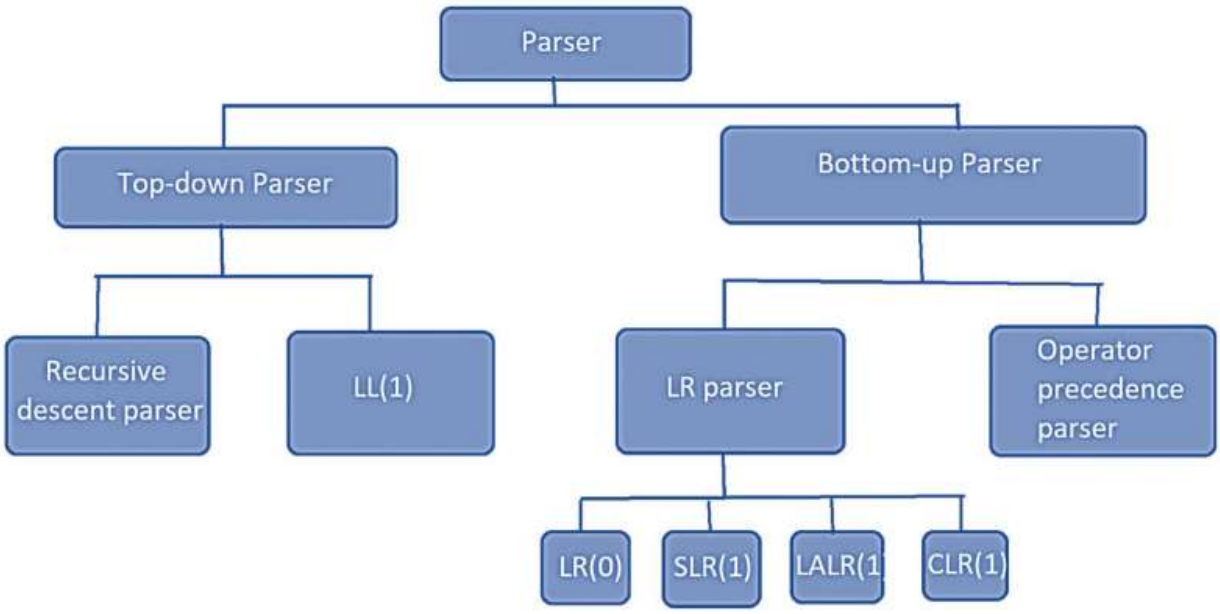
- The compiler should report:
  - What the error is.
  - Where it occurred (line number and position).
  - How it can be fixed (if possible).
- Good error messages help developers correct issues quickly.

Example: Syntax Error: Missing ')' at line 3 Suggestion: Add ')' to complete the if-statement condition

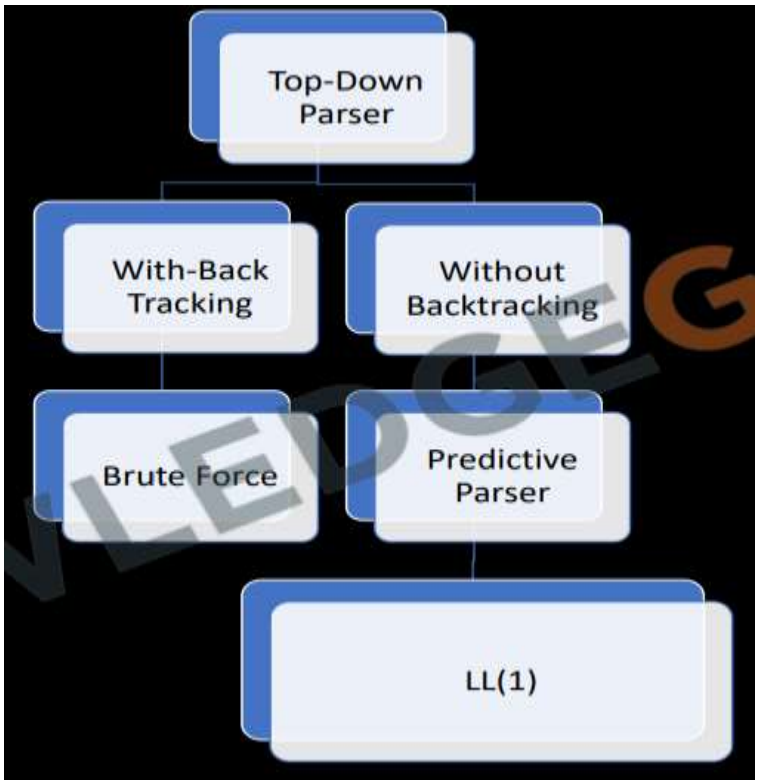
Parser-Level Error Handling Flow

1. The parser reads the source code input.
2. If the input does not match any grammar rule, a syntax error is detected.
3. The parser applies an error recovery strategy.
4. The error is reported to the user.
5. The parser resumes parsing the remaining input.

7. Type of Parser



1. Top Down Parser - A Top-Down Parser is a type of parser used to analyse and construct the parse tree of a given input string starting from the start symbol of a grammar and working towards the terminals (actual symbols in the input string).



- Top-down parsers are most efficient when working with **context-free grammars** (CFGs) that are **non-left-recursive** and **unambiguous**.

- Top-down parsers typically produce a leftmost derivation, meaning the leftmost non-terminal is expanded first.

1. Recursive Descent Parser (Brute Force Parser) - A **Recursive Descent Parser** is a **top-down parsing** technique. It recursively expands non-terminals in the grammar and tries to match them with the input string.

- **When the grammar is simple and deterministic** (e.g., LL(1) grammar).
- **Avoid ambiguous or complex grammars** where backtracking would be too costly.

How It Works:

1. **Grammar Expansion:** Each non-terminal is represented by a function. The function tries all possible production rules (alternatives) for that non-terminal. If one alternative matches, it moves forward. If not, it backtracks and tries the next alternative.
2. **Backtracking:** If one alternative fails to match, the parser backtracks and tries another alternative. This backtracking makes it a **brute-force** approach, as it checks all possibilities.
3. **Success or Failure:** Parsing is successful if an alternative matches the input. If no alternative matches, parsing fails.

Advantages:

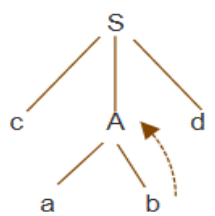
1. **Simple and Easy to Implement:** Directly follows the grammar structure.
2. **Good for Simple, Predictive Grammars:** Works well for grammars with no ambiguity and limited backtracking.

Disadvantages:

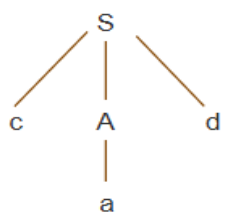
1. **Inefficient for Complex Grammars:** Backtracking increases time complexity ($O(2^n)$).
2. **Cannot Handle Left Recursion:** Left recursion leads to infinite recursion in recursive descent parsers.
3. **Hard to Debug and Optimize:** Due to backtracking, debugging and optimizing are difficult.

Example:

1. Grammar: S → c A d , A → ab | a and String = cad

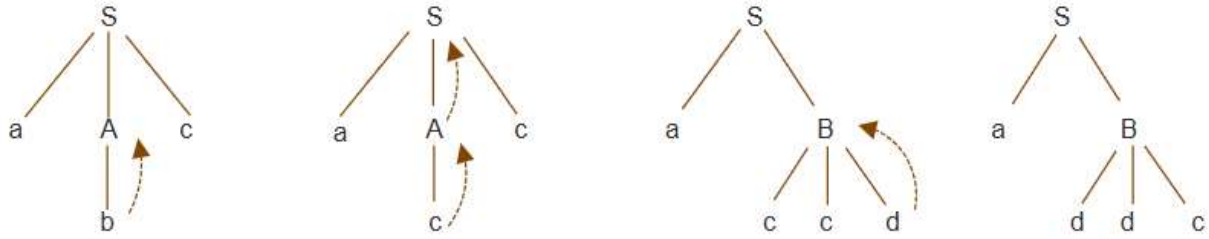


Backtrack as it does not match the input string cab



the input string matches with it

2.Grammar: S → aAc | aB , A → b | c , B → ccd | ddc and String = addc



2. Predictive Parser (LL (1) Parser) - A **Predictive Parser** is a type of **top-down parser** that works by **predicting** which grammar rule to apply, using a **lookahead symbol** (usually one symbol from the input). For a grammar to be used in a predictive parser, **no left recursion**, **no ambiguity,** and be **left-factored.**

First Set - **FIRST(α)** is the set of **terminals** that can appear at the **beginning** of any string derived from α. It is used in **predictive parsing** (LL(1) parsing) to help decide which production to use based on the next input symbol.

Rules to Calculate FIRST(α):

1. If α is a terminal: FIRST(α) = { α }

2. If α is a string of terminals (like abc): FIRST(abc) = { a } because only the first symbol matters when the string is made up of terminals.

3. If α is a non-terminal and one of its productions is α → ε, FIRST(α) includes ε.

4. If α is a non-terminal with production α → b (where b is a terminal): FIRST(α) = { b }

5. If α is a non-terminal with a production like α → X1 X2 X3 ...:

- If FIRST(X1) does **not** contain ε, then: **FIRST(α) = FIRST(X1)**
- If FIRST(X1) **contains** ε, then: **FIRST(α) = (FIRST(X1) ∪ FIRST(X2)) – { ε }**
- If both FIRST(X1) and FIRST(X2) contain ε, and X3 does **not** derive ε, then: **FIRST(α) = (FIRST(X1) ∪ FIRST(X2) ∪ FIRST(X3)) – { ε }**
- If X1, X2, and X3 all derive ε, then: **FIRST(α) = FIRST(X1) ∪ FIRST(X2) ∪ FIRST(X3) ∪ { ε }**

Follow Set – **FOLLOW(α)** is the set of **all terminals** that can **appear immediately to the right of** α in any valid derivation from the start symbol. FOLLOW sets are calculated only **for non-terminals.** They help in building the LL(1) parsing table, especially when a production leads to ε (empty string).

***Rules to Calculate FOLLOW*(α):**

1. If **S** is the start symbol, then **FOLLOW(S) = { $ }**

2. **If A → α B β**, where B is followed by some string β (could be terminals or non-terminals), then: **FOLLOW(B) = FIRST(β) – { ε }**

3. **If A → α B** Or if **A → α B β** and **FIRST(β) contains ε**, then: **FOLLOW(B) = FOLLOW(A)**

Examples:

1. **S → a | b | ε , First(S) =** { a, b, ε } and **Folow(S) =** { $ }
2. **S → aA | bB , A → ε , B → ε**

Non-terminal	FIRST	FOLLOW
S	{ a, b }	{ $ }
A	{ ε }	{ $ }
B	{ ε }	{ $ }

3. **S → AaB | BA , A → a | b , B → d | e**

Non-terminal	FIRST	FOLLOW
S	{ a, b, d, e }	{ $ }
A	{ a, b }	{ a , $ }
B	{ d, e }	{a, b, $ }

4. **S → AaB , A → b | ε , B → c**

Non-terminal	FIRST	FOLLOW
S	{ b, a }	{ $ }
A	{ b, ε }	{ a }
B	{ c }	{ $ }

5. **S → AB , A → a | ε , B → b | ε**

Non-terminal	FIRST	FOLLOW
S	{ a, b,ε }	{ $ }
A	{ a, ε }	{ b, $ }
B	{ b, ε }	{ $ }

6. S → ABCDE , A → a | ε , B → b | ε , C → c | ε , D → d , E → e | ε

Non-terminal	FIRST	FOLLOW
S	{ a, b, c, d }	{ $ }
A	{ a, ε }	{ b, $ }
B	{ b, ε }	{ c, $ }
C	{ c, ε }	{ d }
D	{ d }	{ e, $ }
E	{ e, ε }	{ $ }

7. E → T E', E' → + T E' | ε , T → F T', T' → * F T' | ε, F → (E) | id

Non-terminal	FIRST	FOLLOW
E	{ (, id }	{), $ }
E'	{ +, ε }	{), $ }
T	{ (, id }	{ +,), $ }
T'	{ *, ε }	{ +,), $ }
F	{ (, id }	{ *, +,), $ }

8. S → aBDh , B → cC , C → bC | ε , D → EF , E → g | ε ,F → f | ε

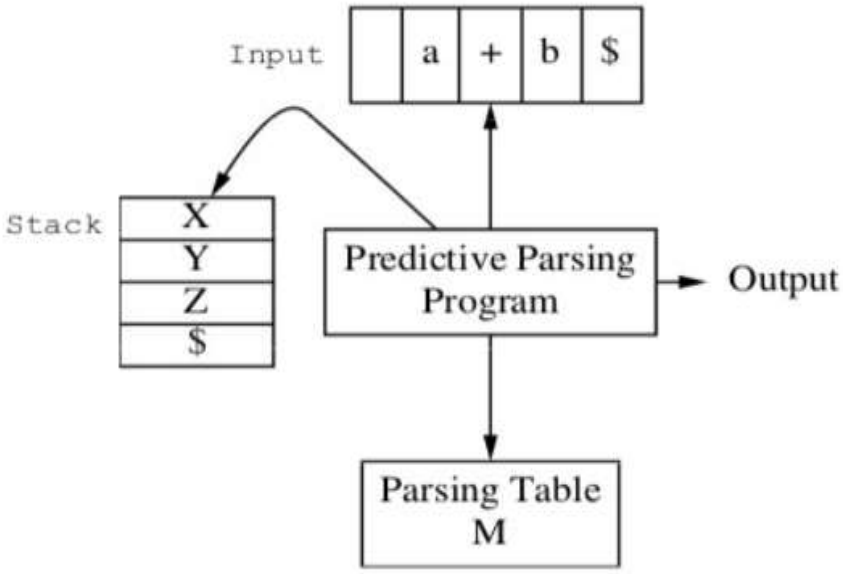
Non-terminal	FIRST	FOLLOW
S	{ a }	{ $ }
B	{ c }	{ g, f, h }
C	{ b, ε }	{ g, f, h }
D	{ g, f, ε }	{ h }
E	{ g, ε }	{ f, h }
F	{ f, ε }	{ h }

LL(1) Parser – **An LL(1) parser** is a type of **non-recursive, top-down parser.** The name **LL(1)** means:

- **First L**: Scans the input from Left to right.
- **Second L**: Produces a Leftmost derivation.
- **1**: Uses one lookahead symbol to make decisions.

LL(1) parsers are **called predictive parsers** because they predict the correct production by looking at the next input symbol.

Components of an LL(1) Parser



1. Input Buffer

- Holds the string to be parsed (e.g., id + id * id$).
- Divided into cells, each holding one input symbol.
- A pointer (header) points to the current lookahead symbol.
- After matching, the pointer moves to the next symbol.
- Ends with $, which is used as an end-of-input marker.

2. Parse Stack

- Holds grammar symbols (terminals and non-terminals).
- Initially contains $ (bottom-of-stack marker) and the start symbol (e.g., S).
- Parsing is performed using push and pop operations based on grammar rules.

3. Parse Table

- A 2D array of size m × n, where:
  - **m** = number of non-terminals
  - **n** = number of terminals + 1 (for $)
- Each cell M[A, a] contains a production A → α or is marked as an error.
- Used to determine which production to apply based on:
  - **A** = top symbol of the stack (non-terminal)
  - **a** = lookahead input symbol

How LL(1) Parser Works

1. Initialize the stack with $ and the start symbol S.
2. Read the current top of stack X and input symbol a.
3. Perform the following:

- If **X = a = $:** Parsing successful.
- **If X = a ≠ $:** Pop X from the stack and advance the input pointer.
- If **X is a non-terminal**:
  - Look up M[X, a]
  - If entry is **X → α**: Pop X, and push α onto the stack in reverse.
  - If entry is in error: Trigger error-handling routine.

<u>Steps to Construct LL(1) Parse Table</u>

1. For each terminal a in FIRST(α): **Add A → α to M[A, a].**
2. If ε is in FIRST(α): For each terminal b in FOLLOW(A)`: **Add A → α to M[A, b].**
3. If $ is in FOLLOW(A) and α → ε**: Add A → ε in M[A, $].**
4. All other cells are marked as errors.

G: $E \rightarrow TE'$
$E' \rightarrow +TE' | \varepsilon$
$T \rightarrow FT'$
$T' \rightarrow *FT' | \varepsilon$
$F \rightarrow id | (E)$

NON TERMINAL	FIRST	FOLLOW
E	{ (,id }	{),$}
E'	{+, ε}	{),$}
T	{ (,id }	{+,),$}
T'	{*, ε}	{+,),$}
F	{ (,id }	{*,+,),$}

LL(1) Parsing Table:

	id	+	*	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \varepsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$
F	$F \rightarrow id$			$F \rightarrow (E)$		

Note: All undefined entries are Errors.

25

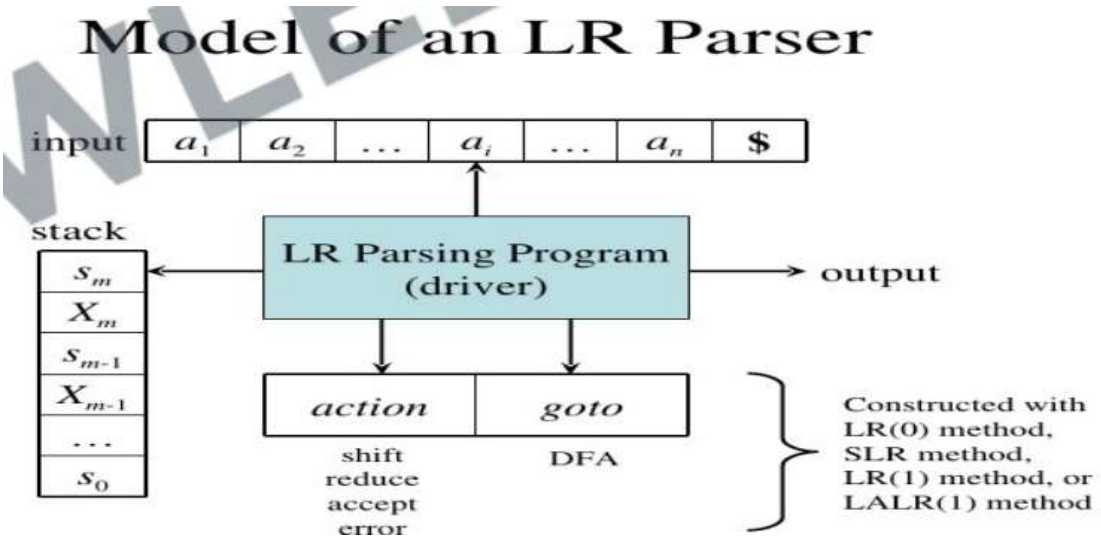
Parsing the input string "id+id*id" using LL(1) parser:

STACK	INPUT	OUTPUT
$E	id + id * id$	
$E' T	id + id * id$	$E \rightarrow TE'$
$E' T' F	id + id * id$	$T \rightarrow FT'$
$E' T' id	id + id * id$	$F \rightarrow id$
$E' T'	+ id * id$	
$E'	+ id * id$	$T' \rightarrow \varepsilon$
$E' T +	+ id * id$	$E' \rightarrow +TE'$
$E' T	id * id$	
$E' T' F	id * id$	$T \rightarrow FT'$
$E' T' id	id * id$	$F \rightarrow id$
$E' T'	* id$	
$E' T' F *	* id$	$T' \rightarrow *FT'$
$E' T' F	id$	
$E' T' id	id$	$F \rightarrow id$
$E' T'	$	
$E'	$	$T' \rightarrow \varepsilon$
$	$	$E' \rightarrow \varepsilon$

2. Bottom-Up Parser - A **Bottom-Up Parser** starts with the input and works towards the root of the parse tree. It constructs the parse tree by reducing sequences of symbols into non-terminals using production rules in reverse.

Bottom-up parsers are used for deterministic, recursive, and even ambiguous context-free grammars, making them ideal for complex programming language syntax. **Bottom-Up Parsers** simulate the **reverse of the rightmost derivation.** Faster and more efficient than Top-Down parsers.

Components of Bottom-Up Parser



1. **Input Buffer**:
   - Contains the string to be parsed.
   - Split into **cells**; each holds one input symbol.
2. **Stack**:
   - Holds grammar symbols.
   - Parser uses **Shift** and **Reduce** operations:
     - If the **handle is not on top → Shift**.
     - If the **handle is on top → Reduce**.
3. **Parse Table**:
   - Contains **Action** and **Goto** parts.

Operations

Operation	Meaning
Shift	Move the next input symbol to the stack.
Reduce	Replace the handle on the stack with the LHS of a production.
Accept	If the stack contains the start symbol and the input is empty → accept the string.
Error	Invalid string (i.e., not derivable from grammar).

1. LR Parser- **LR** stands for:

- **L**: Left-to-right scanning of the input.
- **R**: Construction of a Rightmost derivation in reverse.
- (**k**): k is the number of lookahead symbols.

Handle - A **substring** of the input that matches the **RHS (Right-Hand Side)** of a production rule. This is the part of the string we **reduce** during parsing.

Handle Pruning - The **process of finding a handle** and **replacing** it with its **LHS (Left-Hand Side)** variable.

LR(0) Item: A production with a dot (.) in RHS indicating parser position.

Closure(I): Adds all items starting from the non-terminal **after the dot**.

GOTO(I, X): Moves the dot over symbol **X** and returns a new item set.

LR Parser Construction Procedure

Step 1: Augmented Grammar - Add a new start production**:** If original: **S → AB,** then augmented**: S' → S, S → AB.**

Step 2: Create LR(0) Items - Insert a dot (•) at different positions in the RHS.

Step 3: Construct Canonical Collection - Denoted as: C = { I0, I1, I2, ..., In } where each Ii is a set of LR(0) items.

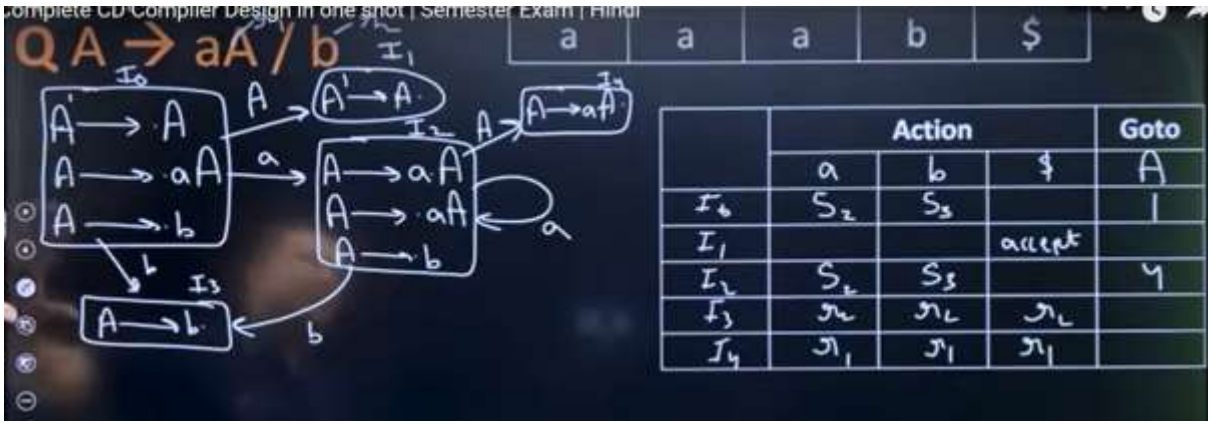
Step 4: Compute Closure and GOTO Functions - Closure(I): Adds new items based on the current items. If an item has A → α .Bβ and B → γ exists, add B → .γ. **Goto(I, X**): Moves the dot past symbol X:

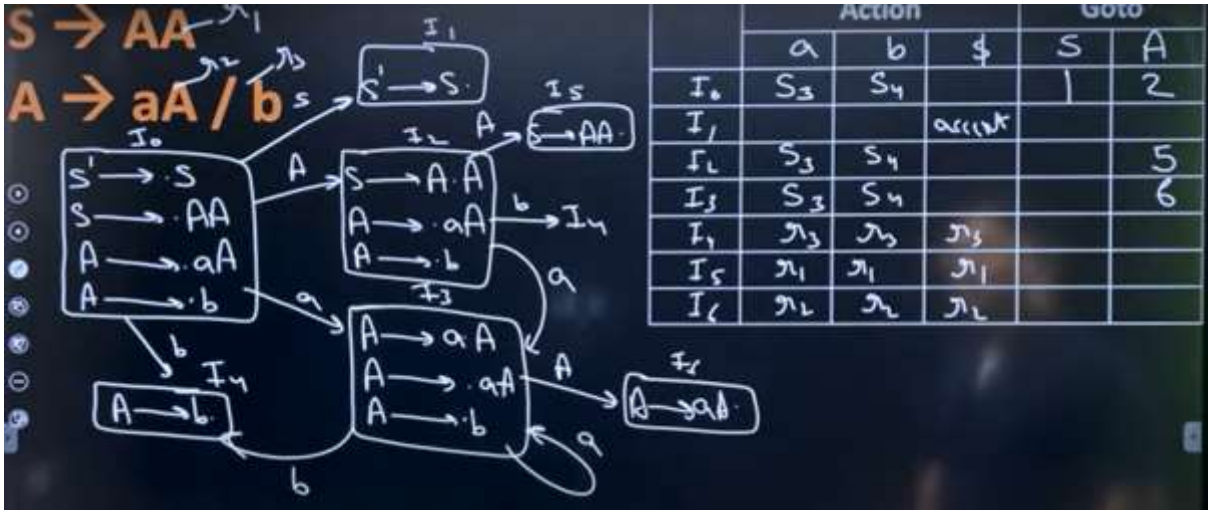
Step 5: Construct the Parse Table - For every item in every state:

- If item is A → α . a β and a is terminal **→ Action = Shift.**
- If item is A → α. → **Action = Reduce A → α.**
- If item is S' → S . → **Action = Accept.**
- For each non-terminal X, **use Goto(Ii, X) = Ij → Goto entry.**

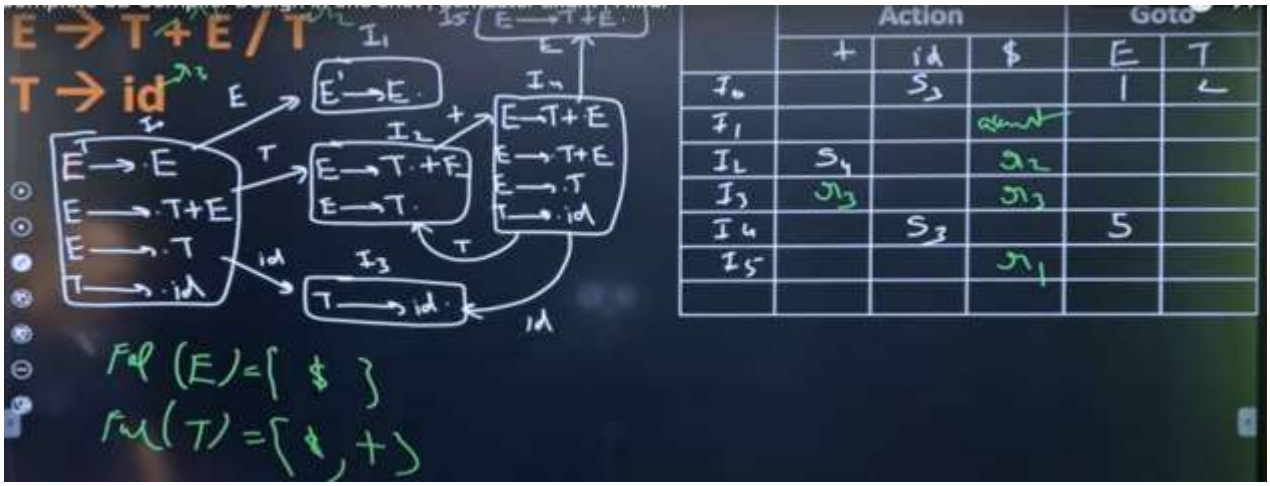
LR(0) parser - LR(0) parsers are a type **of bottom-up parser**. They are used to process input in a left-to-right manner and perform reverse right-most derivations. The "**0**" **in LR(0**) refers **to 0 lookahead symbols**, meaning the parser uses no lookahead information when making parsing decisions.

Examples:





SLR(1) Parser – An SLR(1) parser (Simple LR(1) parser) is an enhancement of the LR(0) parser. While the construction of the SLR(1) parse table is similar to LR(0), it introduces a key distinction in the handling of the reduce actions. Specifically, in SLR(1), **the parser uses follow sets t**o **determine when a reduction action** should be applied, which makes it more powerful than the LR(0) parser.



CLR(1) Parser – A CLR(1) parser, also known as Canonical LR(1), is an advanced form of the LR(1) parser. It is a more general and powerful parsing technique that handles a wider range of grammars. However, it is also more complex and requires more computational resources than other types of LR parsers like SLR(1) or LR(0).

Key Features of CLR(1) Parser

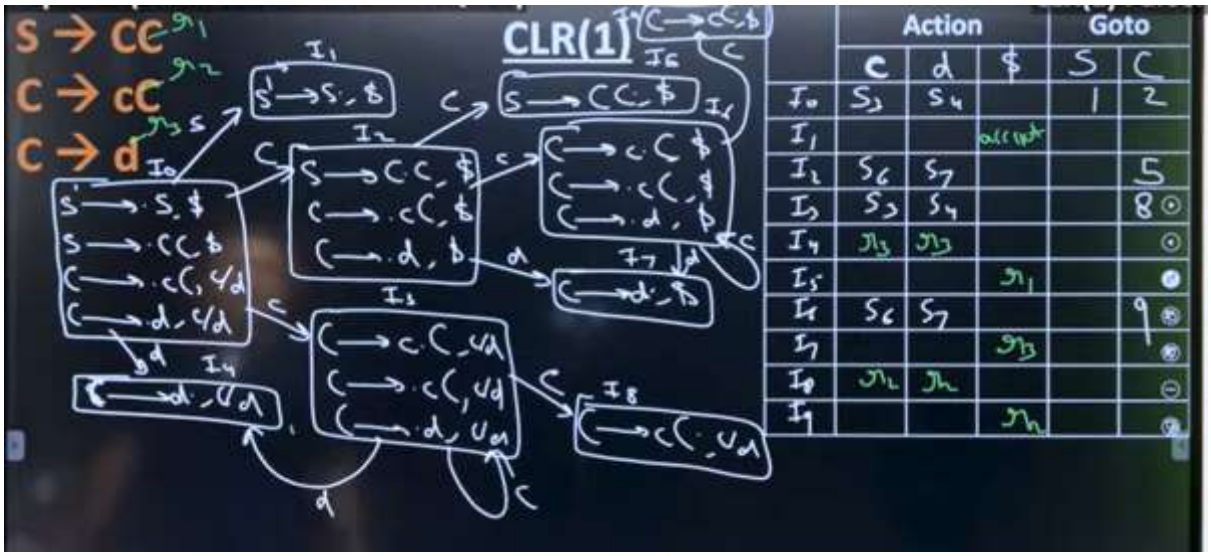
1. **Lookahead:** The LR(1) parser uses one lookahead symbol to make decisions. In CLR(1), the concept remains the same, but the parsing table construction becomes more detailed and sophisticated.
2. **Closure and Goto:**
   - Closure is used to determine which items (productions with a dot on the right-hand side) are part of the parser's state. The Goto operation determines the next state after reading a specific symbol.
   - The difference between CLR(1) and SLR(1) or LR(0) lies in how lookahead and follow sets are handled, especially when dealing with conflicts**.**

Closure Operation in CLR(1) **-** Closure in CLR(1) is similar to that in LR(1), but it has additional complexity due to the lookahead symbol: If an item is of the form A → α.Bβ (with the dot . before a terminal symbol or non-terminal B), and if β can derive something (i.e., β → б is in the grammar), then β can be added to the closure, with lookahead symbols derived from First(β) and Follow(A).

Characteristics of CLR(1) Grammar

- A CLR(1) grammar is a grammar for which the CLR(1) parse does not have any conflicts or multiple entries in the parse table. This means that for each state, there should not be more than one action (like shift or reduce) for any given lookahead symbol.
- CLR(1) grammar can handle more complex cases, including those that involve conflicts in parse tables, but it is more computationally expensive compared to SLR(1) or LR(1).

Example:



LALR (1) Parser - **LALR(1) (Lookahead LR(1))** is a modified version of the **CLR(1) parser**, which **merges states** with the same production **but different follow** sets to reduce the size of the parse table.

Main Idea:

- Combine states that have identical production parts but different follow sets.
- This reduces memory usage and improves efficiency.

Steps:

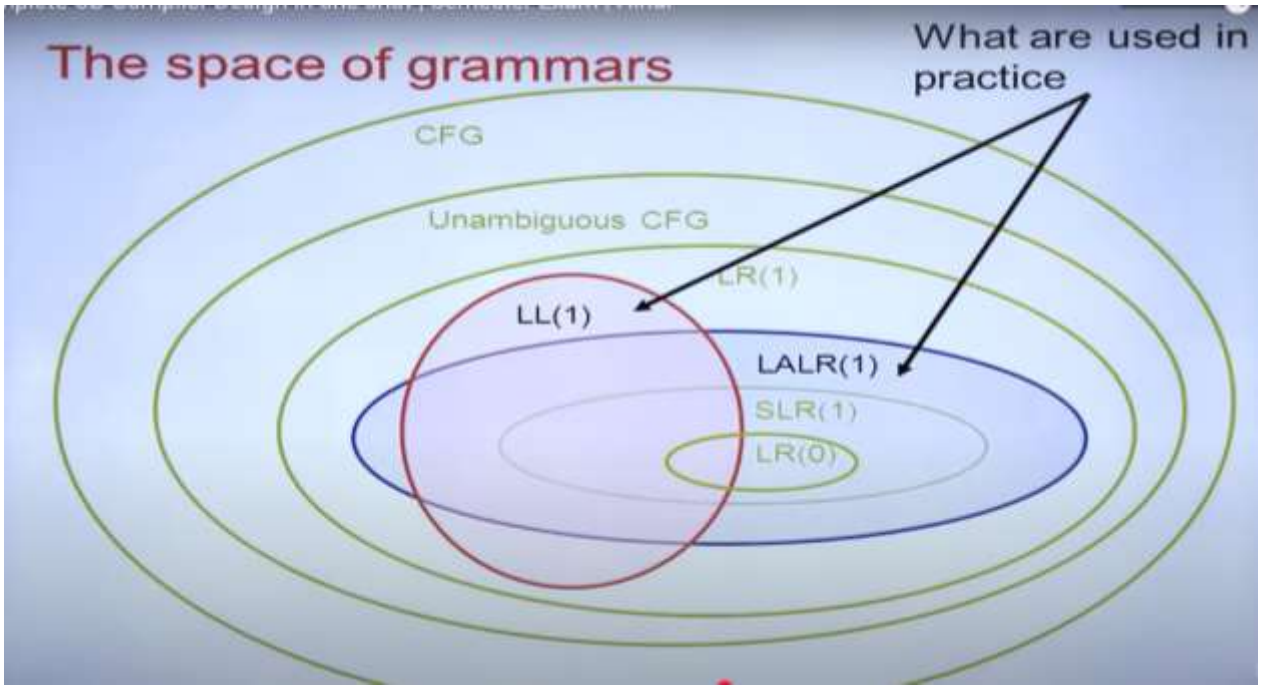
1. Start with CLR(1) construction (create the canonical collection of LR(1) items).
2. Merge states with identical productions but different follow sets.
3. Construct the LALR(1) parse table using merged states.

Advantages:

- **Space-efficient:** Fewer states and a smaller parse table.
- **Faster parsing:** Less memory means quicker parsing.

Disadvantages:

- **Less powerful than CLR(1):** Some grammars CLR(1) can handle, LALR(1) may not**.**
- **State merging complexity:** Merging states can introduce ambiguities.



Feature / Parser	LL(1)	LR(0)	SLR(1)	CLR(1)	LALR(1)
Type	Top-Down	Bottom-Up	Bottom-Up	Bottom-Up	Bottom-Up
Derivation	Leftmost (from root to leaf)	Rightmost in reverse (leaf to root)	Rightmost in reverse	Rightmost in reverse	Rightmost in reverse
Lookahead	1 symbol	0 symbols	1 symbol (uses FOLLOW)	1 symbol (uses FIRST & FOLLOW)	1 symbol (uses merged lookaheads)
Grammar accepted	Simple, non-left recursive	Very restricted	More than LR(0), less than CLR(1)	Most general context-free grammars	Nearly as powerful as CLR(1)
Conflict handling	Can't handle left recursion	Prone to conflicts	Fewer conflicts than LR(0)	Least conflicts	May have conflicts if merging is wrong
Table size	Small	Small	Small	Very large	Moderate
Ease of construction	Easy	Moderate	Moderate	Complex	Moderate
Error detection	Poor	Good	Good	Very good	Very good
Efficiency	Fast but less powerful	Slow and less powerful	Faster than LR(0), less than CLR(1)	Accurate but memory-heavy	Efficient + balanced
Power ranking	Lowest	Bit more	More powerful	Most powerful	Nearly same as CLR(1)

2. Operator Precedence Parser - It is a bottom-up parser. Mainly used to parse arithmetic expressions with operators like +, -, *, /. Based on operator precedence relations between terminals. It helps decide which part of the input to reduce first, using operator priority.

For a grammar to be parsed using an operator precedence parser, it must satisfy:

- **No ε-productions** (i.e., no rule like A → ε)
- **No two non-terminals should be adjacent in the RHS of any production**
  For example:
  - Allowed: S → aA, A → b
  - Not Allowed: S → AB

Such grammars are called **Operator Precedence Grammars.**

The parser uses three precedence relations between terminals (symbols):

Relation	Meaning	Example (assume * > +)
<.	Yields precedence (lower priority)	+ < *
=.	Equal precedence	+ = +
>.	Takes precedence (higher priority)	* > +

These symbols help to detect the handle in the input string for reduction.

Example: Used to decide which symbol takes priority.

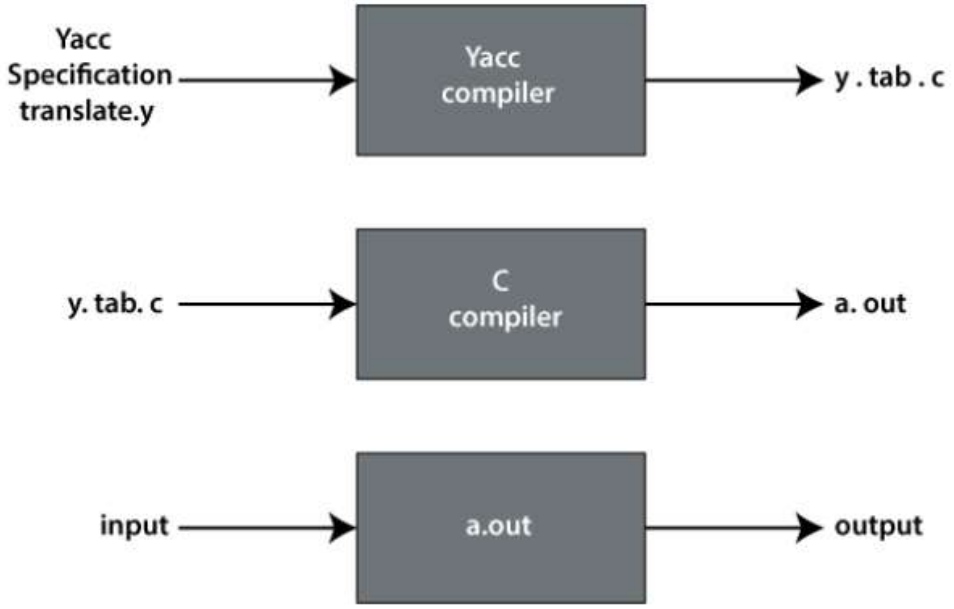
	+	*****	**id**	**$**
+	**>**	**<**	**<**	**>**
*****	**>**	**>**	**<**	**>**
id	**>**	**>**		**>**
$	**<**	**<**	**<**	

Advantages

- Simple for parsing arithmetic expressions.
- Can handle some ambiguous grammars (unlike LL parsers).
- Easy to implement in practice for expression evaluation.

8. YACC (Yet Another Compiler Compiler)

YACC is a tool used in compiler construction to automatically generate a parser. It takes a language's grammar and produces C code for a parser based on that grammar. YACC is used for building parsers in compilers and interpreters.



Key Aspects of YACC

1. **Function of YACC:**
   - YACC generates parsers from grammar specifications, allowing the creation of tools like compilers and interpreters.
   - It uses a given syntax specification and produces a C program that performs syntax analysis (parsing).
2. **Parser Type:**
   - The parser generated by YACC is typically an LALR(1) parser. This type of parser is efficient and works well for many grammars.
   - LALR(1) stands for Look-Ahead Left-to-Right parsing with 1 symbol of lookahead.
   - It is a bottom-up parser, meaning it starts from the leaves (input symbols) and works its way to the root (start symbol).
3. **Integration with Lex:**
   - YACC is often used together with Lex, a tool for generating a lexical analyzer.
   - Lex handles the tokenization of the input (splitting the input into meaningful symbols or tokens), while YACC focuses on the syntax analysis (parsing the structure of the input).

Structure of a YACC File

A YACC file generally has three parts:

1. **Declarations:**
   - The first part contains declarations such as token definitions and C code that needs to be included.
   - For example, tokens like DIGIT, KEYWORD, etc., are declared in this section using the %token keyword.
   - C code, like header files or constants, can also be included between %{ and %}.

2. **Rules (Grammar and Actions):**
    - This section contains grammar rules (productions) and the semantic actions that should be executed when a rule is matched.
    - The rules are written in a format like:
    - The semantic actions are C statements executed when the rule is applied. $$ refers to the value of the non-terminal on the left-hand side of the production, and $1, $2, etc., refer to the values of the symbols on the right-hand side.
3. **User Subroutines:**
    - This part contains any additional C functions needed for the parser.
    - A critical function that must be provided is yylex(), which is the lexical analyzer responsible for tokenizing the input.

Advantages of YACC

1. **Automates Parser Creation:** YACC automates the process of generating a parser from a grammar specification, saving significant time and effort in compiler construction.
2. **Simplifies Grammar Modification**: Changes to the grammar can be easily made in the YACC specification, which automatically updates the parser code, making it more maintainable.
3. **Separation of Syntax and Actions:** YACC allows a clear separation between syntax rules (grammar) and semantic actions (the associated C code), leading to cleaner and more modular code.

UNIT 3: Intermediate Code Generation

1. Syntax Directed Definition (SDD)

Syntax Directed Definition (SDD) combines **context-free grammar (CFG)** with **semantic rules** to translate a high-level language into intermediate or machine code**.** The grammar defines the **structure of the language,** while **the semantic rules** define the meaning of the language constructs.

Key Concepts in SDD:

1. **Grammar**: Specifies the syntactic structure of the language.
2. **Semantic Rules:** Attach meaning to the syntactic rules and guide the translation to intermediate representations or code.
3. **Semantic Actions:** The actions that are executed as part of translation rules, such as computation, assignment, or printing values during the parsing process.

Applications of SDD:

1. **Code Generation:** Converting high-level code to low-level machine code.
2. **Intermediate Code Generation:** Creating machine-independent intermediate code for optimization.
3. **Expression Evaluation:** Evaluating the expressions during parsing.
4. **Infix to Postfix Conversion:** Converting infix expressions to postfix notation.

Examples:

1. Compute the Value for the Expression using SDD
Grammar:

- E → E1 # T { E.value = E1.value * T.value }
- E → T { E.value = T.value }
- T → T1 & F { T.value = T1.value + F.value }
- T → F { T.value = F.value }
- F → num { F.value = num.value }

Input Expression: 2 # 3 & 5 # 6 & 4

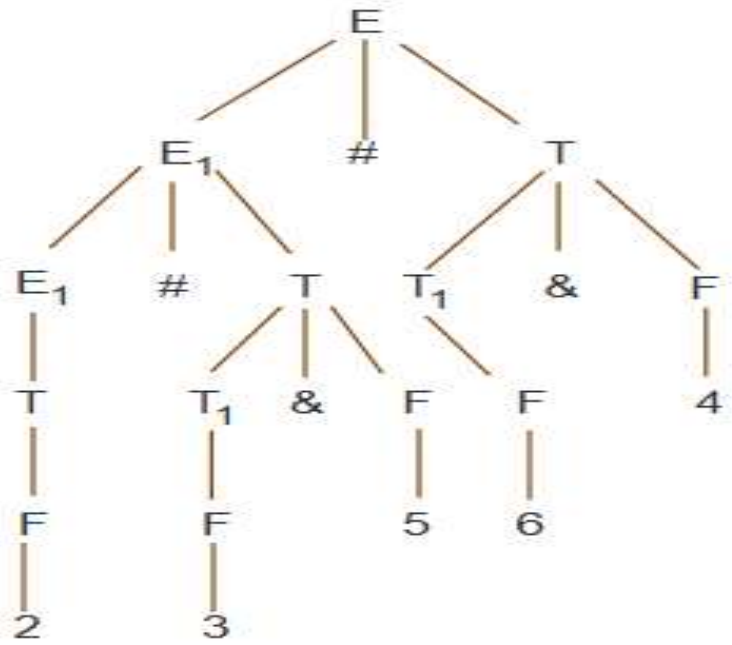
Basically, first 3 & 5 will be solved = 8

Then 6 & 4 will be solived = 10

Then 2 # 8 will be solved = 16

Then 16 # 10 will be solved = 160

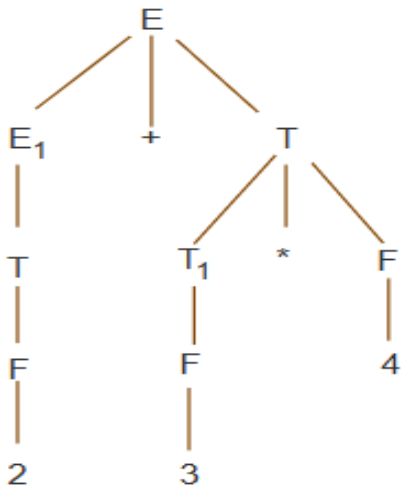
Hence, value = 160.



2. Code Generation for Infix Expression
Grammar:

- E → E1 + T { print('+') }
- E → T
- T → T1 * F { print('*') }
- T → F
- F → num { print(num.value) }

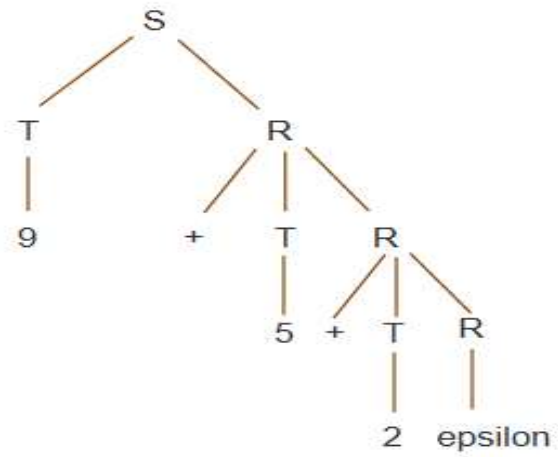
Input Expression: 2 + 3 * 4
So, the Output will be **234*+**



3. Translation Scheme for Simple Addition
Grammar:

- S → T R
- R → + T { print('+') } R | ε
- T → num { print(num.value) }

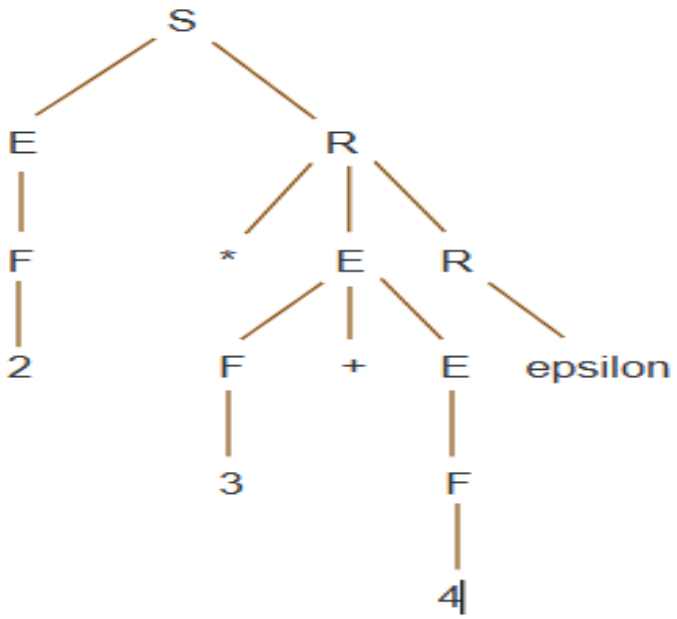
Input Expression: 9 + 5 + 2
So, The Output is **95+2+**



4. Translation Scheme for Multiplication and Addition
Grammar:

- S → E R
- R → * E { print("*") } R | ε
- E → F + E { print("+") } | F
- F → (S) | id { print(id.value) }

Input Expression: 2 * 3 + 4
So, the output is **234+***



2. Types of Attributes in SDDs

Attribute in the context of **Syntax Directed Definitions (SDDs)** refers to an additional piece of information associated with the grammar symbols (non-terminals and terminals) in a **parse tree**. These attributes can store various kinds of information, such as the **type of an expression**, the **value of a variable**, or a **memory location**.

Attributes are classified into **two types** based on how their values are computed:

1. Synthesized Attributes - A **synthesized attribute** is one whose value is computed **from the attribute values of its child nodes** in the parse tree.

Key Characteristics:

- **Flow of information**: From the **bottom-up** (from the children to the parent node).
- **Used for**: Typically employed in **bottom-up parsers** like **SLR, LALR, and Canonical LR parsers**.
- **Evaluation**: Synthesized attributes are computed after the attributes of the child nodes are known. This means that the parent node's value depends on its children's values.

***Example*: A → B C D .** A synthesized attribute might be: **A.val = f(B.val, C.val, D.val)**

Usage:

- These attributes propagate from the **leaves** of the parse tree (the children) **upwards** to the root (the parent).
- **No need for information from the parent or siblings**.

2. Inherited Attributes - An **inherited attribute** is one whose value is computed **from the attributes of its parent node or left siblings** in the parse tree.

Key Characteristics:

- **Flow of information**: From the **top-down** (from parent to children) or **left-to-right** (across siblings).
- **Used for**: Typically employed in **top-down parsers** like **LL parsers**.
- **Evaluation**: Inherited attributes are computed **before** visiting the corresponding child node. These attributes are often used to pass **context-sensitive** information, like data types or scope, from the parent to the children or from left siblings to the right children.

***Example*: A → B C D .** Inherited attributes might be: B.in = A.in **,** C.in = B.out **and** D.in = C.out

Usage:

- These attributes help propagate information about **context**, like the data type expected at a node or the scope of a variable.
- The inherited attributes must be **calculated before** the child node is visited, as they rely on the parent or sibling nodes.

Aspect	Synthesized Attributes	Inherited Attributes
Definition	Attributes computed from the values of **child nodes**.	Attributes are computed from the values of the **parent or left siblings**.
Flow of Information	From **bottom-up** (children to parents).	From **top-down** (parent to children) or **left-to-right** (across siblings).
Evaluation Order	Evaluated after the children's attributes are computed.	Evaluated before visiting the corresponding child node.
Used in Parsing Techniques	Bottom-up parsers like **SLR**, **LALR**, and **Canonical LR**.	Top-down parsers are like **LL parsers**.
Typical Usage	Information propagation **upwards** from the leaves to the root.	Context-sensitive information is passed **downwards** or **across** the tree.
Need for Parent/Sibling Info	Does not require information from parents or siblings.	Requires context from the **parent** or **left siblings** to be computed.
Example	A → B C D; A.val = f(B.val, C.val, D.val)	A → B C D; B.in = A.in, C.in = B.out, D.in = C.out
Attributes Propagation	Propagates information from **child nodes** to the **parent node**.	Propagates context-sensitive information to **child nodes** or **siblings**.
Context-Sensitivity	Typically used for **evaluated values** of expressions or computed results.	Used for **context-related information** such as variable types, scopes, etc.

3. Evaluation Orders in SDDs

The **evaluation order** refers to the **sequence** in which the attributes of the nodes in the parse tree are evaluated. It ensures that all dependencies are met before any attribute is computed.

1. Evaluation Order for Synthesized Attributes

Evaluation Strategy:

- Synthesized attributes are computed using a **bottom-up** approach.
- The parse tree is traversed **from the leaves to the root**.

- For each node, the values of its **children's attributes** are evaluated first before the node's synthesized attribute can be computed.

Key Points:

- **No need to access parent or sibling information**; only children's information is needed.
- This evaluation order fits naturally with **bottom-up parsers**, such as **LR** and **LALR** parsers.
- The evaluation is often performed using **post-order traversal** of the parse tree (processing children before the parent).

2. Evaluation Order for Inherited Attributes

Evaluation Strategy:

- Inherited attributes require a **top-down** or **left-to-right** approach.
- The parse tree is usually traversed **depth-first** and **left-to-right**.
- Inherited attributes must be computed **before visiting** the corresponding child node in the parse tree.

Key Points:

- Inherited attributes require information from the parent node and sometimes from the left sibling.
- These attributes must be computed **before** the corresponding child node is visited.
- This evaluation strategy requires **careful control of the flow** of attribute values to ensure that all dependencies are met before an attribute is evaluated.

4. Types of Syntax-Directed Definitions Based on Attribute Usage

There are **two main types** of Syntax Directed Definitions (SDDs) based on how attributes are used during evaluation.

1. S-Attributed Definitions - An **S-attributed definition** uses **only synthesized attributes**. **Semantic actions** (rules for computing the attribute values) are placed at the **end** of the production.

Suitable for: **Bottom-up parsers**, such as **LR, SLR, and LALR** parsers.

***Evaluation*: Post-order traversal** of the parse tree (bottom-up). No dependencies on parents or siblings.

2. L-Attributed Definitions - An **L-attributed definition** uses **both inherited and synthesized attributes**. **Inherited attributes** must be computed using values from the **parent node or left siblings** only. **Semantic actions** can be placed **anywhere** in the production, not necessarily at the end.

Suitable for: **Top-down parsers**, such as **LL parsers**.

Evaluation: **Depth-first, left-to-right traversal** of the parse tree. Inherited attributes must be evaluated **before** visiting the child nodes.

Aspect	S-Attributed Definitions	L-Attributed Definitions
Definition	Uses only **synthesized attributes**.	Uses both **synthesized and inherited attributes**.
Type of Attributes Used	Only **synthesized attributes** (attributes computed from child nodes).	Both **synthesized** and **inherited** attributes (synthesized attributes from children and inherited attributes from parents or siblings).
Semantic Actions	Semantic actions (rules to compute attributes) are placed at the **right end** of the production.	Semantic actions can be placed **anywhere** on the right-hand side of the production.
Evaluation Order	Can be evaluated in a single **post-order traversal** (bottom-up) of the parse tree.	Attributes are evaluated using a **depth-first, left-to-right** traversal of the parse tree.
Parent-Sibling Information	**No need** for information from parents or siblings.	Inherited attributes require context from the **parent** or **left siblings** to be computed.
Use Case	Suitable for **bottom-up parsers** like **LR, SLR, LALR**.	Suitable for **top-down parsers** like **LL parsers**.
Example	A → B C D; {E. val = E1. val + T. val} (here, E. val is synthesized from E1. val and T. val).	A → B C D; {B. in = A. in, C. in = B.out, D. in = C.out, A. out = D.out} (uses both inherited and synthesized attributes).
Context Sensitivity	Less context-sensitive as it only uses synthesized attributes that do not depend on the parent or siblings.	More context-sensitive, as inherited attributes depend on the parent or left siblings.
Suitability	Simple and easier to implement, generally used for **bottom-up parsing**.	More complex, used for **top-down parsing** and handles more context-sensitive cases.

5. Intermediate Language

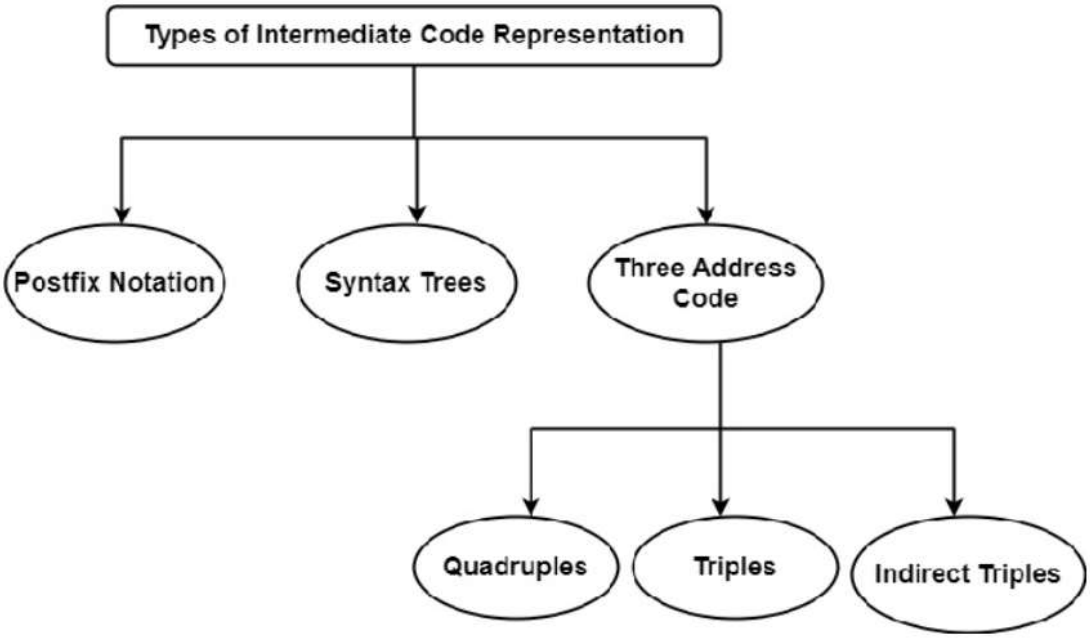
Intermediate Language (IL) is an abstraction between high-level programming languages and machine code**. It is a low-level, machine-independent code** that the compiler generates after parsing and semantic analysis. **The IL is an intermediate representation** of the source code that can **be further optimized and translated** into machine-specific code by the compiler's back-end.

Advantages /Purpose

1. **Portability**: Machine-independent, so only the back-end needs to change for new hardware.
2. **Optimization**: Easier to apply optimizations like constant folding, dead code elimination, and loop optimizations.
3. **Simplicity**: Breaks the compiler into three stages: **Front-end**: Parses the high-level code, **Middle-end**: Generates and optimizes IL, and **Back-end**: Converts IL to machine code.

Example: **High-level code**: x = y + z * 60;
Translation into Three Address Code (TAC):

1. t1 = z * 60
2. t2 = y + t1
3. x = t2



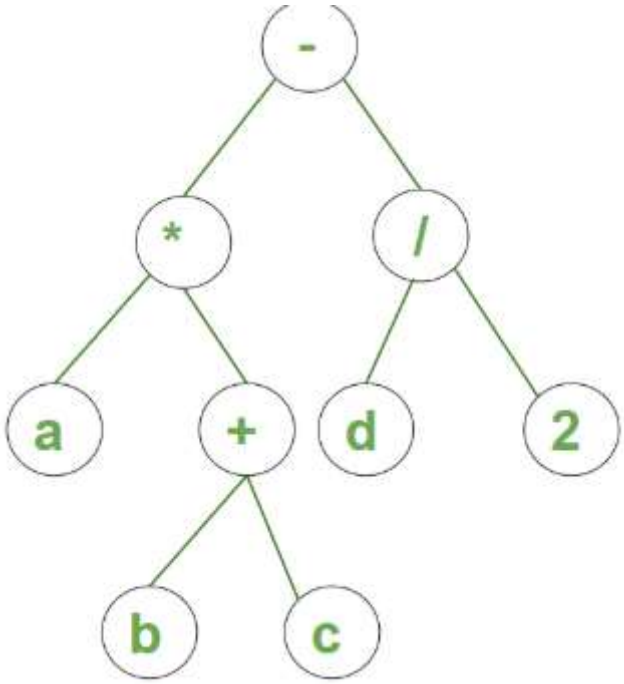
1. Syntax Tree (Abstract Syntax Tree - AST): A syntax tree (or abstract syntax tree) is a hierarchical representation of a program's syntactic structure, where **Internal nodes** represent operators (e.g., arithmetic or logical) and **Leaf nodes** represent operands (e.g., variables or constants).

Key Features:

- **Simplified Representation**: It condenses the parse tree, removing unnecessary details.
- **Operator Precedence**: Operators that are evaluated first are placed deeper in the tree.
- **Structure**: The root represents the final operation, and child nodes represent components of expressions.

Purpose: Used for **parsing**, **optimizing**, and **code generation** in compilers.

Example : a * (b + c) – d/2



2. Postfix Notation (Reverse Polish Notation - RPN): Postfix notation places the operator **after** the operands, eliminating the need for parentheses and operator precedence rules. It is a linear representation of an expression.

Key Features:

- **No Parentheses**: Parentheses are not required due to fixed operator positions.
- **Operator After Operands**: Operators always follow their operands.
- **Linear Representation**: It is a simplified form of a syntax tree, showing the order of operations clearly.

Evaluation Steps (Using Stack):

- Push operands (variables/constants) onto the stack.
- When an operator is encountered, pop operands from the stack, operate, and push the result back onto the stack.
- The final result remains in the stack after the entire expression is processed.

Advantages:

- **No Operator Precedence Rules**: The position of operators automatically defines their order of application.
- **Efficient for Stack-Based Evaluation**: It is particularly useful for evaluation in virtual machines or stack-based computations.

Example: a * (b + c) – d/2

1. a*(b c +) - d/2
2. a b c + * - d 2 /
3. a b c + * d 2 / -

3. Three Address Code (TAC): Three Address Code (TAC) is an intermediate representation used in compilers. It decomposes complex expressions into simple steps, each involving at most three addresses: two operands and one result. It helps simplify the translation process and makes optimizations easier during code generation.

Key Features of Three-Address Code:

- **Simplified Expressions**: TAC decomposes complex expressions into simpler ones.
- **Temporary Variables**: Results are stored in temporaries (compiler-generated variables).
- **Control Flow Representation**: TAC captures control flow and data dependencies.
- **Optimization**: TAC allows easier code optimization since operations are clear and modular.
- **Intermediate Representation**: It sits between high-level source code and machine-level code.

Applications of Three-Address Code:

1. **Optimization**: TAC is used during the optimization phase to improve the performance of the generated code. Optimizations like constant folding, loop unrolling, etc., are easier in TAC.
2. **Code Generation**: TAC helps in generating code specific to the target platform. It ensures that the generated code is efficient and correct.
3. **Debugging**: TAC can be used to debug and trace the program execution at an intermediate stage before final code generation.
4. **Language Translation**: TAC helps in translating source code from one programming language to another by converting to an intermediate representation.

Types of Three-Address Code Representations:

1. Quadruples: It consists of four fields:

- **op**: operator
- **arg1**: first operand
- **arg2**: second operand
- **result**: result of the operation

Advantages: Easy to rearrange code, access temporary variables via the symbol table.
Disadvantages: Generates many temporary variables.

2. **Triples**: Uses only three fields:
   - **op**: operator
   - **arg1**: first operand
   - **arg2**: second operand

Advantages: More compact, fewer temporary variables.
Disadvantages: Difficult to rearrange and optimize due to implicit temporary variables.

3. **Indirect Triples**: Uses a pointer to refer to the list of references, which is stored separately. Requires less space than quadruples and is easier to rearrange code.

Advantages: Less space compared to quadruples, easier to rearrange.
Disadvantages: Requires a separate list for references, and can be more complex to implement.

Example:

$-(a + b) * (c + d) + (a + b + c)$

1) $t_1 = a+b$
2) $t_2 = -t_1$
3) $t_3 = c+d$
4) $t_4 = t_2 * t_3$
5) $t_5 = a+b$
6) $t_6 = t_5 + c$
7) $t_7 = t_4 + t_6$

Quadruples

	Operator	$Operand_1$	$Operand_2$	Result
1)	+	a	b	t_1
2)	-	t_1		t_2
3)	+	c	d	t_3
4)	*	t_2	t_3	t_4
5)	+	a	b	t_5
6)	+	t_5	c	t_6
7)	+	t_4	t_6	t_7

1) $t_1 = a+b$
2) $t_2 = -t_1$
3) $t_3 = c+d$
4) $t_4 = t_2 * t_3$
5) $t_5 = a+b$
6) $t_6 = t_5 + c$
7) $t_7 = t_4 + t_6$

Triplet

	Operator	$Operand_1$	$Operand_2$
1)	+	a	b
2)	-	1	
3)	+	c	d
4)	*	2	3
5)	+	a	b
6)	+	5	c
7)	+	4	6

1) $t_1 = a+b$
2) $t_2 = -t_1$
3) $t_3 = c+d$
4) $t_4 = t_2 * t_3$
5) $t_5 = a+b$
6) $t_6 = t_5 + c$
7) $t_7 = t_4 + t_6$

Q Write the quadruples, triple and indirect triple for the following expression : **(x+y)* (y+z)+(x+y+z) ?**

$t_1 = x + y$
$t_2 = y + z$
$t_3 = t_1 + t_2$
$t_4 = t_1 + z$
$t_5 = t_3 + t_4$

Index	Operator	Arg1	Arg2	Result
1	+	x	y	t1
2	+	y	z	t2
3	*	t1	t2	t3
4	+	x	y	t4
5	+	t4	z	t5
6	+	t3	t5	t6

Q Write the quadruples, triple and indirect triple for the following expression : **(x+y)* (y+z)+(x+y+z) ?**

$t_1 = x + y$
$t_2 = y + z$
$t_3 = t_1 + t_2$
$t_4 = t_1 + z$
$t_5 = t_3 + t_4$

Index	Operator	Arg1	Arg2
0	+	x	y
1	+	y	z
2	*	0	1
3	+	x	y
4	+	3	z
5	+	2	4

Pointer	Index
p0	0
p1	1
p2	2
p3	3
p4	4
p5	5

Implementation of Three-Address Code:

1. **Source Code Parsing**:
   - **Lexical Analysis**: Converts source code into tokens (keywords, identifiers, operators).
   - **Syntax Analysis**: Builds the Abstract Syntax Tree (AST) to represent the structure of the code.
   - **Semantic Analysis**: Detects errors (type mismatches, undeclared variables) and creates a symbol table.
2. **TAC Instruction Generation**:
   - Traverse the AST to generate the corresponding TAC instructions.
   - Use temporary variables to store intermediate computations.
3. **Evaluation of Expressions**:
   - **Arithmetic Expressions**: Break complex expressions into simpler ones. **Control Flow**:
   - **If Statements**: Use conditional jumps.
   - **Loops**: Convert loops to equivalent while loops with conditional and unconditional jumps.
4. **Procedure Calls**:
   - **Argument Passing**: Push arguments onto the stack.
   - **Function Call**: Jump to the function entry point.
   - **Return Value**: Store the result in a temporary variable and pop the arguments off the stack.

Types of Three-Address Statements

1. **Assignment with Binary Operators:**
   - **Form**: x = y op z
   - **Description**: Assigns the result of a binary operation between operands y and z to variable x.
2. **Assignment with Unary Operators:**
   - **Form**: x = op y
   - **Description**: Assigns the result of applying a unary operation (op) to operand y to variable x.
3. **Copy Statement:**
   - **Form**: x = y
   - **Description**: Copies the value of operand y into variable x.
4. **Unconditional Jump:**
   - **Form**: goto L
   - **Description**: Unconditionally jumps to the statement labeled L. This is used for flow control in loops and branches.
5. **Conditional Jump:**
   - **Form**: if x relop y goto L

- **Description**: Applies a relational operator (relop) such as <, >, ==, !=, >=, <= to operands x and y, and if the condition holds, jumps to the statement labeled L.

6. **Procedure Call and Return:**
   - **Form**:
     - param x1, param x2, ..., param xn
     - call p, n
     - return y
   - **Description**: These statements handle procedure calls and passing parameters.
     - param x: Specifies the parameter x being passed to the procedure.
     - call p, n: Calls procedure p with n parameters.
     - return y: Returns the value y from the procedure.
7. **Indexed Assignments:**
   - **Form**: x = y[i] or x[i] = y
   - **Description**: These are used for array accesses.
   - x = y[i]: Assigns the value at the i-th index of array y to variable x.
   - x[i] = y: Assigns the value of y to the i-th index of array x.
8. **Address and Pointer Assignments:**
   - **Form**:
     - x = &y: Assigns the address of y to x.
     - x = *y: Dereferences pointer y and assigns the value to x.
     - *x = y: Dereferences pointer x and assigns y to the memory location pointed to by x.
   - **Description**: These statements deal with memory addresses and pointer dereferencing.

6. Types and Declarations

Types:: Types refer to the data structures used in the source programming language and how they are represented in the target language or intermediate code.

Examples of Types:

- **Primitive Types:** Integer (int), Floating-point (float), Character (char), Boolean (bool).
- **Compound Types**: Arrays, Structures, Classes.
- **User-defined Types:** Enums, Typedefs, etc.

Purpose: Types help the compiler understand how to treat data—whether it should be treated as a number, string, or something else. The type of a variable dictates the kind of operations that can be performed on it.

Translation of Types:

- During translation, the source language's type system is converted into the intermediate code (IC) type system.
- **Example:**
  - A float in a high-level language like C may be represented as a register or memory address in the target machine language.
  - This translation ensures that operations on variables respect their type, like performing addition only on compatible types (e.g., adding two integers or two floating-point numbers).

Declarations: Declarations in a program give information about variables, functions, or objects. They specify the type, scope, and memory allocation for those elements.

- **For variables:** The declaration tells the compiler the variable's type, where it will be stored (memory address), and how long it lasts (scope).
- **For functions:** The declaration specifies the function's return type and its parameters.
- **Memory Allocation and Scope:**
  - Declarations allow the compiler to allocate memory for variables at runtime (or during compilation).
  - The scope defines where the variable is accessible in the code (e.g., within a function or globally).

Storage Layout: The storage layout refers to how the memory is organized and allocated for different types of variables, arrays, structures, etc., during execution**. The layout dictates where each data item is stored and how it can be accessed during runtime.**

Memory Allocation Example for Arrays:

- Arrays have a contiguous block of memory, where each element is stored next to the other. To access an element in an array, we need to compute its memory address.
- The formula for accessing an element at index k in an array a[] is:
  - a[k] = B + W*k
  - B is the base address of the array.
  - W is the size of each element (e.g., 4 bytes for integers).
  - k is the index of the element.

Array Representation and Address Calculation: When accessing an array element, the base address of the array and the size of each element are used to compute its address.

Example: To access the kth element in an array, we calculate:

- **a**[k] = B + W * (k - Lower Bound)
- Where:
  - B is the base address of the array.
  - W is the size of each element.
  - k is the index of the element.
  - Lower Bound is the index of the first element of the array, ensuring we account for arrays that might start at an index other than 0.

7.Translation of Expressions

When we write arithmetic or boolean expressions in a high-level programming language like C or Java, the compiler has to **translate** these into **intermediate code** (e.g., Three Address Code - TAC) so the machine can understand and execute them.

1. Arithmetic Expressions Translation - These include operators like +, -, *, /.

Example: a = b + c * d;

Three Address Code (TAC):

1. t1 = c * d
2. t2 = b + t1
3. a = t2

2. Boolean Expressions Translation - Boolean expressions include AND, OR, NOT, comparisons (==, <, >).

Example: if (a < b || c < d)

TAC using Short-Circuit Evaluation:

```
if a < b goto L1

if c < d goto L1

goto L2

L1: // true block

L2: // false block
```

8. Translation of Control Flow Statements

Control flow statements like **if**, **while**, **for**, and **switch** are also converted to intermediate code using jumps and labels.

1. If-Else Statement

```
if (a < b) t = 1;

else t = 0;
```

TAC:

```
if a < b goto L1

t = 0

goto L2

L1: t = 1

L2:
```

2. While Loop

```
while (a < b) {

   a = a + 1;

}
```

TAC:

```
L1: if a < b goto L2
   goto L3
L2: a = a + 1
     goto L1
L3:
```

3. For Loop

```
for (i = 0; i < 10; i++) {

    S;

}
```

TAC:

```
i = 0

L1: if i < 10 goto L2

    goto L4

L2: S

i = i + 1

goto L1

L4:
```

4. Switch-Case Statement

```
switch (a) {

  case 1: x = 10;

          break;

  case 2: x = 20;

          break;

  default: x = 0;

}
```

TAC:

```
t1 = a

if t1 == 1 goto L1

if t1 == 2 goto L2

goto L3

L1: x = 10

   goto L4

L2: x = 20

   goto L4

L3: x = 0

L4:
```

9. Type Checking

Type checking is the process of verifying that **each operation** in a program is performed on **data types that are allowed** by the language. It ensures that type-related errors are **detected and reported** either at **compile time** or **runtime**.

Types of Type Checking

1) Static Type Checking

- Performed during **compile time**.
- The type of every variable is **known and fixed** at compile time.
- Common in languages like **C, C++, Java**.

Example:

```
int x = 5;

x = "hello";  // Error: assigning string to int
```

2) Dynamic Type Checking

- Performed during **runtime**.
- Variables **can change types** over time.
- Used in languages like **Python, JavaScript, Perl, Ruby**.

Example:

```
int x = 5;

x = "hello";  # Allowed in Python
```

Position of Type Checker in Compiler

The type checker works **after parsing** and **before intermediate code generation**.

Source Code → Token Stream → Syntax Tree → Type Checker → Intermediate Code

- **Input:** Syntax Tree
- **Output:** Annotated tree with type information or type errors

Type Checking of Expressions

Type expressions represent the **type of each sub-expression** in a syntax tree.

Translation Rules:

1. **Literal Constants**
   - Rule: E → literal
   - Meaning: If E is a character literal, then E.type = char.
   - If E is a numeric literal, then E.type = integer.
2. **Identifier**
   - Rule: E → id
   - Action: E.type := lookup(id.entry)
   - Meaning: The type of identifier id is looked up from the symbol table.
3. **Binary Operation (mod example)**
   - Rule: E → E1 mod E2
   - Action:

```
if E1.type = integer and E2.type = integer
   then E.type = integer
   else type_error
```

4. **Array Access**
   - Rule: E → E1 [E2]
   - Action:

```
if E1.type = array(s)
   and E2.type = integer
   then E.type = s
   else type_error
```

Type Checking of Statements

1. **Assignment Statement**

   - Rule: S → id := E
   - Action:

```
if id.type = E.type
   then S.type = void
   else type_error
```

2. **Conditional Statement (if-then)**

   - Rule: S → if E then S1
   - Action:

```
if E.type = boolean and S1.type = void
   then S.type = void
   else type_error
```

3. **While Statement**

   - Rule: S → while E do S1
   - Action:

```
if E.type = boolean and S1.type = void
   then S.type = void
   else type_error
```

4. **Sequence of Statements**

   - Rule: S → S1 ; S2
   - Action:

```
if S1.type = void and S2.type = void
   then S.type = void
   else type_error
```

Type Checking of Function Calls

- Rule: E → E1 (E2)
- Action:

```
if E1.type = s → t and E2.type = s
    then E.type = t
    else type_error
```

Role of Symbol Table in Type Checking

- Maintains:
  - Variable names
  - Their declared types
  - Scope information
- Used by the type checker to:
  - Lookup types
  - Verify correctness
  - Detect undeclared or mistyped identifiers

UNIT 4: Run-Time Environment and Code Generation

1. Symbol Table

A **symbol table** is a data structure used by a compiler to store and manage information about identifiers (such as variables, functions, constants, arrays, structures, and user-defined types) in a program.

Need for Symbol Table

- The symbol table helps the compiler to track and manage each identifier:
  - Name
  - Type
  - Size
  - Scope (local or global)
  - Lifetime
  - Memory location
- It is essential for various compiler tasks such as:
  - Semantic analysis
  - Type checking
  - Code generation

Role of Symbol Table in Compiler Phases

Compiler Phase	Role of Symbol Table
Lexical Analysis	Creates new entries for identifiers as they are encountered for the first time.
Syntax Analysis	Adds attribute information such as data types and scope to existing entries.
Semantic Analysis	Validates type correctness and checks for undeclared or redeclared identifiers.
Intermediate Code Generation	Stores and manages temporary variables used in expression translation.
Code Optimization	Supports optimization by providing variable usage, lifetime, and type details.
Code Generation	Supplies memory addresses and helps allocate registers or memory locations.

Information Stored in Symbol Table

A symbol table typically maintains the following information for each identifier:

- Name of the identifier
- Data type (e.g., int, float, char, etc.)
- Declaring a procedure or function
- Scope level (local, global, or block-level)
- Pointer to a structure or class definition (if applicable)
- Parameter passing mode (by value or by reference)
- Number and types of function arguments
- Base memory address or offset for storage

Functions of a Symbol Table

1. **Lookup -** Checks whether a specific identifier exists in the table. Useful during semantic checks and parsing.
2. **Insert -** Adds a new identifier with associated information (name, type, scope, etc.) into the table.
3. **Access -** Retrieves detailed information about a specific identifier.
4. **Modify -** Updates the existing information of an identifier (e.g., updating the memory address or type).
5. **Delete - Removes** identifiers from the table (e.g., when their scope ends).

Requirements for an Efficient Symbol Table

1. **Adaptive Structure -** Must support various types of identifiers and their attributes like functions, arrays, pointers, etc.
2. **Quick Lookup/Search -** Should allow fast searching and retrieval operations, especially important for large programs.
3. **Space-Efficient -** Should use memory efficiently and allow dynamic resizing as needed.
4. **Language Feature Accommodation** - Should support features of modern programming languages such as:
    - Block-structured scopes
    - Function overloading
    - Parameter passing rules
    - Implicit variable declarations

Data Structures for Symbol Table Implementation

1. Unordered List

Student Name	Age
John	20
Maria	19
Alex	21
Sarah	18
Brian	22

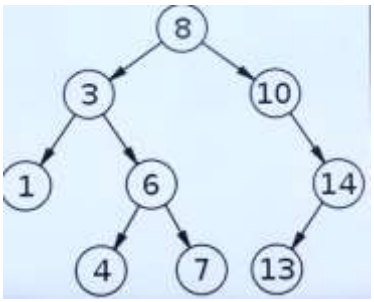
- **Implemented using**: Arrays or Linked Lists.
- **Lookup time**: O(n)
- **Insertion time**: O(1)
- **Advantages:** Very easy to implement, and Fast insertion.
- **Disadvantage**: Poor search performance for large programs.
- **Use Case**: Small symbol tables where insertions are more frequent than lookups.

2. Ordered List

Variable Name	Data Type
age	int
height	float
name	string
score	double
width	float

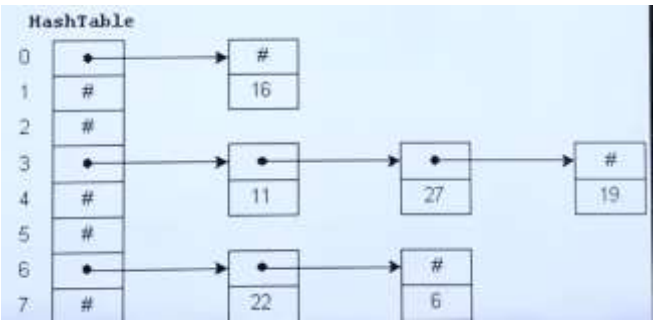
- **Implemented using**: Sorted arrays or linked lists.
- **Lookup time**: O(log n) using binary search.
- **Insertion time**: O(n) because list must be kept sorted.
- **Advantages**: Faster search than an unordered list.
- **Disadvantages**: Insertion is slower because of shifting elements to maintain order.
- **Use Case**: Suitable for fixed sets of names, such as reserved keywords.

3. Search Tree



- **Implemented using**: Binary Search Tree (BST), AVL Tree, or Red-Black Tree.
- **Average time complexity**: O(log n) for search, insert, and delete.
- **Advantages**: Balanced trees provide efficient access time and can dynamically adjust to insertions and deletions.
- **Disadvantages**: Tree needs to be balanced to ensure optimal performance, and are more complex to implement.
- **Use Case**: Dynamic symbol tables where entries are frequently added or removed.

4. Hash Table



- Uses a hash function to map identifier names to indices.
- **Average time complexity**: O(1) for insert and search.
- **Advantages**: Very fast for a large number of identifiers, and Efficient handling of frequent lookups.
- **Disadvantages:** Collision handling is required (using chaining or open addressing, and the Hash function must be well-designed to ensure uniform distribution.
- **Use Case**: Most commonly used in real-world compilers.

Entries in the Symbol Table

1. Variables

- Represent memory locations whose values can change.
- Symbol table stores name, data type, scope, and memory location.

2. Constants

- Immutable values known at compile time.
- Usually not allocated separate memory; values are embedded directly in code.

3. User-Defined Types

- Include structures, unions, typedefs, and enums.
- Symbol table records the type definition and all its components.

4. Classes

- Abstract data types with attributes and methods.
- Symbol table maintains information about constructors, destructors, inheritance, virtual tables, etc.

5. Records (Structures)

- Collection of fields with possibly different data types.
- Symbol table includes each field's name, type, and offset.

Scope and Scope Management

Scope refers to the region in a program where an identifier (such as a variable or function name) is valid and can be accessed. The symbol table must manage information across different scopes and must be able to distinguish between identifiers with the same name declared in different scopes.

1. Nested Scopes

- Inner blocks can redefine variables declared in outer blocks.
- Each block has its symbol table.

2. Scope Stack

- Symbol tables are managed using a stack.
- New scope → push new table.
- Exiting scope → pop the table.

3. Multiple Declarations

- The same identifier can be declared multiple times in different scopes.
- Symbol table must store each instance uniquely.

4. Scope Rules

- Identifiers declared in the same scope must have unique names.
- Lookup begins from the current scope and proceeds to outer scopes.

Lexical vs Dynamic Scoping

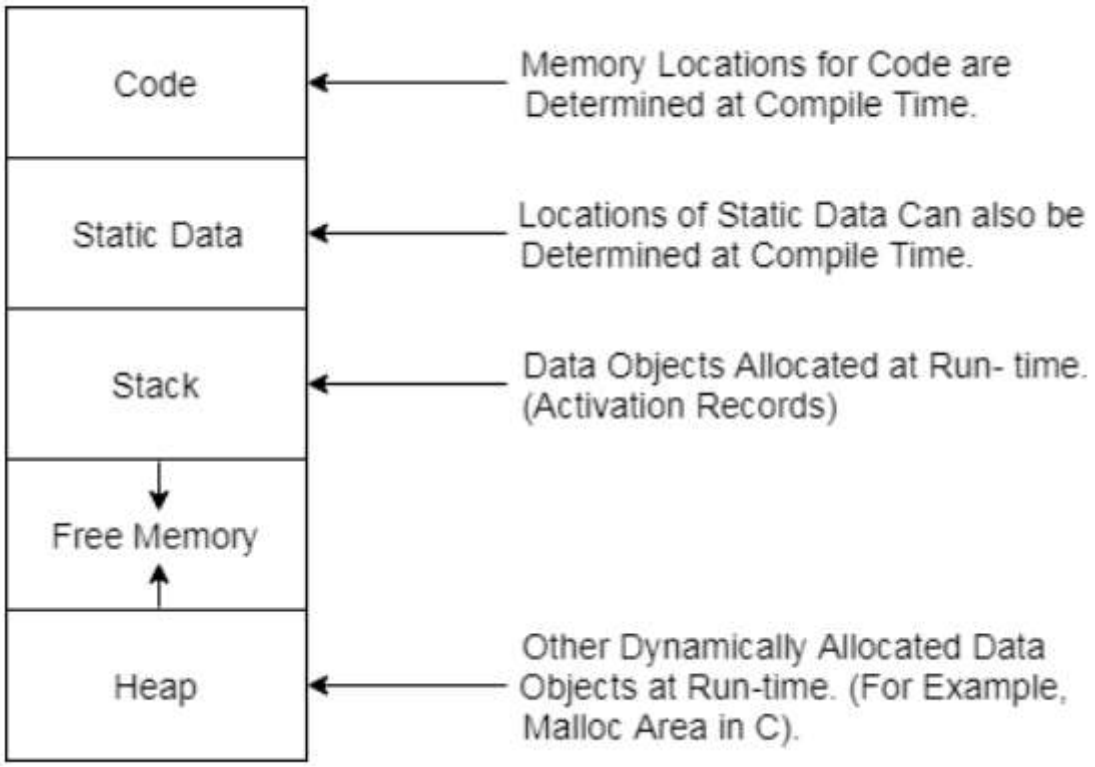
Aspect	Lexical Scoping	Dynamic Scoping
Determination Time	At compile time	At runtime
Basis of Scope	Structure of source code	Call stack or calling sequence
Predictability	High – easy to understand and debug	Low – depends on runtime behavior
Usage in Languages	Used in most modern languages (C, Java)	Found in older languages (Lisp)

2. Storage Organization

Storage organization refers to how memory is structured and managed during the execution of a program, especially in the context of a compiler. A compiler must ensure that appropriate memory is allocated to store:

- Code
- Variables (local, global, temporary)
- Function call information
- Dynamic data structures

There are several **memory allocation methods** used in compilation depending on the lifetime and accessibility of data.



1. Code Storage

- This section of memory holds the **machine-level target code** produced by the compiler.
- **Fixed-size and unchanging**: Once compiled, the code does not change.
- Code is usually loaded into a read-only section of memory to prevent accidental modification during program execution.
- Examples: Function definitions, jump statements, loop instructions.

2. Static Allocation

In static allocation, memory for all data variables is allocated **at compile time**. The size and memory location of every object are determined before the program runs.

Features:

- The compiler calculates and **reserves memory** during compilation.
- Each variable gets a **fixed memory location** throughout the program execution.
- Used for **global variables**, **static variables**, and **constants**.

Advantages:

- Efficient access because addresses are determined during compilation.
- No overhead of memory management during runtime.

Limitations:

- Cannot handle recursive function calls or dynamic memory needs.
- Not suitable for programs where the size of the data is unknown at compile time.

***Example*:** int a = 5; // Static allocation

3. Stack Allocation

Used to manage memory for **function calls and local variables** using a stack-based approach.

Key Component: Activation Record

- Also known as a **stack frame**.
- Contains local variables, return address, parameters, and bookkeeping information.

How it Works:

- On a function call, a new **activation record is pushed** onto the stack.
- When the function returns, the record is **popped** off, releasing memory.

Features:

- Memory for **local variables** is allocated on the stack.
- Fresh memory is allocated on each function call, even for recursive calls.
- Once the function ends, its stack space is reclaimed automatically.

Advantages:

- Efficient for memory reuse in function calls.
- Supports **recursion** and **nested calls**.

Limitations:

- Not suitable for variables that need to persist after the function returns.

Example:

```
void foo() {

   int x = 10; // Stored in activation record

}
```

4. Heap Allocation (Heap Management)

Heap allocation is used when memory is needed **dynamically** at runtime and the object lifetime **extends beyond the function** that created it.

Characteristics:

- Allocates memory from the heap area.
- Typically used for **objects**, **dynamic arrays**, and **data structures** like linked lists.

a. Reference Counting

- Each heap object maintains a **reference counter**.
- The counter tracks the number of references (or pointers) to that object.
- When a reference is removed, the counter decreases.
- When the counter becomes zero, the object is no longer accessible and can be freed.

Working:

- When a new reference is created to the object, the **counter is incremented**.
- When a reference is deleted or reassigned, the **counter is decremented**.
- If the counter becomes **zero**, it means no part of the program can access the object, and it is **safe to deallocate** its memory.

Advantages:

- Immediate deallocation when an object is no longer used.

- Simple to implement.

Limitations:

- Cannot handle **cyclic references** (e.g., object A references B, and B references A, but no external reference points to A or B — the cycle remains even though both are unreachable).

Example:

```
Node* p = new Node(); // reference count = 1

Node* q = p;        // reference count = 2

p = NULL;           // reference count = 1

q = NULL;           // reference count = 0, object can be freed
```

<u>b. Garbage Collection</u>

- Automatic process of **reclaiming unused memory**.
- Works by finding objects that are **no longer reachable** from any part of the program.
- These unreachable objects are marked as **garbage,** and memory is reclaimed.

How it works:

- Many techniques like **Mark-and-Sweep**, **Generational GC**, etc., are used.
- A **garbage collection bit** is often used to mark used and unused memory areas.

Advantages of Heap Allocation:

- Allows creation of dynamic and flexible data structures.
- Objects can persist beyond their creation scope.

Disadvantages of Heap Allocation:

- More overhead due to dynamic memory management.
- May lead to memory leaks if not managed properly (especially in manual allocation languages like C/C++).

Allocation Type	Allocation Time	Lifespan	Access Time	Examples
Code Storage	Compile time	Entire execution	Fast	Executable instructions
Static Allocation	Compile time	Entire execution	Fast	Global/static variables
Stack Allocation	Runtime	Until the function ends	Fast	Local variables, parameters
Heap Allocation	Runtime	Until explicitly freed or GC	Slower	Dynamic objects, linked lists, trees

3. Run-Time Administration / Stack Allocation Space

During program execution, the **run-time environment** manages various activities such as memory allocation, procedure call handling, and variable access. One of the most important aspects of this environment is how **procedure calls** are implemented using **activation records** and a **stack-based memory model**.

This method is called the **Simple Stack Allocation Scheme**, and it's used when a program has **nested or recursive function calls**.

An **activation record** (also known as a *stack frame*) is a block of memory that stores all the information needed to manage a **single execution** of a procedure (or function).

- Each time a function is called, a **new activation record is pushed** onto the run-time stack.
- When the function ends, its activation record is **popped off** the stack.

This mechanism helps the program **keep track of function calls**, their parameters, return values, and local variables.

Structure of an Activation Record

The typical activation record contains several fields. These fields may vary slightly depending on the language and implementation, but the common ones are:

1. Return Value

- Stores the **value returned** by the function.
- Space is reserved by the **caller**, and the **callee** fills in this value before returning.
- Used when functions have a return type (e.g., int, float, etc.).

2. Actual Parameters (Arguments)

- These are the **values passed** from the calling function to the called function.
- Stored in a **fixed order** so the callee knows how to retrieve them.
- In the activation record, these parameters are placed **by the caller** before control is transferred.

3. Control Link (Dynamic Link)

- Points to the **activation record of the caller**.
- Helps restore the **stack frame** of the calling function after the current function completes.
- It allows the system to return to the correct place in the stack after a function returns.

4. Access Link (Static Link)

- Points to the **activation record of the lexically enclosing function**.
- Useful for accessing **non-local variables** in **nested functions** (especially in languages that support nested scopes like Pascal or modern functional languages).
- Helps in **scope resolution** during variable access.

5. Saved Machine Status

- Stores information about the **state of the machine** just before the function was called.

- This includes:
    - **Return address**: where the control should return after the function completes.
    - **Saved registers**: important processor registers that might be overwritten by the function.

6. Local Data (Local Variables)

- Stores **all local variables** declared in the function.
- Allocated by the **callee** when the function starts executing.
- Each call to the function gets a **new, fresh copy** of local variables.

7. Temporaries

- Stores **intermediate values** calculated during the execution of expressions inside the function.
- May include:
    - Partial results of arithmetic operations
    - Temporary variables created during compilation (e.g., in TAC)

Benefits of Stack-Based Allocation

- **Simple and efficient**: Automatic memory management.
- **Supports recursion**: Each function call has a separate space.
- **Clear variable lifetime**: Variables live only during function execution.

Limitations

- Not suitable for **dynamic objects** that must persist beyond the function (like objects created using new).
- Cannot handle **coroutines** or **multithreading** without modifications.

4. Access to Non-Local Data on the Stack

- In many block-structured programming languages, functions can be **nested inside other functions**.
- A nested (inner) function may try to **access variables declared in an outer (enclosing) function**.
- These accessed variables are called **non-local variables** (not local to the current function).

- During execution, each function gets its **activation record (AR)** on the stack, so variables are separated in different stack frames.
- Since each function call creates a new activation record, the inner function cannot directly access the outer function's variables.
- The solution is to use **Access Links** (also called Static Links).
- An Access Link is a **pointer stored in each activation record**, which points to the **activation record of its immediately enclosing function**.
- They allow the inner function to **find and use variables from outer scopes**.
- When a function accesses a non-local variable, it **follows the access link chain** until it reaches the activation record where that variable is stored.

5. Issues in the Design of a Code Generator

A **code generator** is a key part of a compiler that converts **intermediate code** (e.g., abstract syntax trees, triples, quadruples) into **machine-level code**. Several factors need to be considered for its design to ensure **correctness**, **efficiency**, and **maintainability**.

1. Input to Code Generator

- **Input**: Intermediate code + Symbol table.
  - **Intermediate Code**: Triple/Quadruple/Abstract Syntax Trees.
  - **Symbol Table**: Contains information like variable addresses.
- **Challenges**: The input must be free from errors, as the code generator assumes prior phases (like type-checking) have already handled those.
- **Impact**: Poor quality input leads to incorrect or inefficient target code.

2. Target Program

- **Output Forms**:
  - **Absolute Machine Code**: Fast execution but inflexible.
  - **Relocatable Code**: Flexible, requires a loader, adds overhead.
  - **Assembly Code**: Easier to generate but needs an assembler to convert it to machine code.
- **Choice** depends on the execution environment and linking requirements.

3. Memory Management

- **Mapping**: Variables are assigned to memory using the symbol table.
- **Challenges**:

    - Efficient use of memory (static vs dynamic).
    - Avoiding memory conflicts or overlaps.
- **Optimization** is crucial for performance.

4. Instruction Selection

- **Goal**: Choose the most appropriate machine instruction for each intermediate code operation.
- **Challenges**:
    - Multiple ways to implement the same operation.
    - The wrong choice can make code slow or bulky.
- **Knowledge Needed**:
    - Available instruction set
    - Instruction costs (speed, size)
    - Processor addressing modes

5. Register Allocation

- **Why Important**: Registers are faster than memory, so using them efficiently improves performance.
- **Steps**:
    1. **Register Allocation**: Decide which variables stay in registers.
    2. **Register Assignment**: Assign specific registers to variables.
- **Challenges**:
    - Limited number of registers.
    - Handling **register spills** (when data is stored in memory due to insufficient registers).

6. Evaluation Order

- **What it Affects**: Register usage, number of instructions, and overall performance.
- **Challenges**:
    - Optimal order reduces memory and register usage.
    - Finding the best order can be computationally expensive and may require complex algorithms.

Disadvantages of a Code Generator

1. **Limited Flexibility**: Typically designed for a specific platform or language, making it hard to adapt.
2. **Maintenance Overhead**: Regular updates are required to stay compatible with new language features or platforms.
3. **Debugging Difficulties**: Generated code is harder to read and debug compared to hand-written code.
4. **Performance Issues**: May not generate as efficient code as manually written code.
5. **Steep Learning Curve**: Developers must understand the underlying compiler theory.
6. **Over-Reliance**: Developers may become too dependent on generated code and lose manual coding skills.

Approaches to Handle Code Generation Issues

A good code generator must meet the following goals:

1. **Correctness**: The generated code must accurately reflect the logic of the source program.
2. **Maintainability**: The design should be modular and easy to update or extend.
3. **Testability**: The code generator should be easy to test, with clear unit tests.
4. **Efficiency**: The generated code must be optimized for speed and memory usage, balancing both factors.

6. Design a Simple Code Generator

A simple **code generator** in a compiler translates intermediate representations (like Three-Address Code or TAC) into machine or assembly code. Here's how it works in brief:

1. **Input (IR)**: It takes an intermediate representation (e.g., TAC or AST) of the program.
2. **Instruction Mapping**: The code generator converts operations in the intermediate code into corresponding machine or assembly instructions.
3. **Register Allocation**: It assigns variables to machine registers or memory, optimizing register use.
4. **Expression Translation**: For expressions like a = b + c, it generates assembly instructions to load operands into registers, perform the operation, and store the result.

5. **Control Flow Translation**: It handles control structures (like loops and conditionals), generating jump and branch instructions.
6. **Function Calls**: For function calls, it saves the return address, passes arguments, and manages the call/return sequence.
7. **Output**: The code generator outputs machine/assembly code that can be executed on the target machine.

Example:

For t1 = a + b and t2 = t1 * c, the code generator generates assembly instructions like:

```
MOV R1, [a]
MOV R2, [b]
ADD R3, R1, R2
MOV [t1], R3
MOV R3, [t1]
MOV R4, [c]
MUL R5, R3, R4
MOV [t2], R5
```

UNIT 5: Code Optimization

1. Principal Sources of Optimization

Code optimization is a phase in compiler design where the code is transformed to make it more efficient without changing its functionality.

The main goals of optimization are:

- To make the code run faster
- To reduce memory usage
- To generate compact and efficient machine code
- To improve performance while maintaining the same output

Types of Code Optimization

Code optimization is broadly classified into two categories:

1. Machine-Independent Optimization - These optimizations are not tied to any specific hardware or machine architecture. They are applied to the intermediate code generated by the compiler.

***Purpose**:* To improve the logic, structure, and overall efficiency of the code before it is converted to machine-specific instructions.

***Key Features**:*

- Works on intermediate representation (IR)
- Independent of the target machine
- Improves code quality for any target platform
- Increases portability

***Common Techniques**:*

1. **Constant Folding**
    - Evaluates constant expressions at compile time.
    - Example: 4 * 5 is replaced by 20.
2. **Strength Reduction**
    - Replaces costly operations with cheaper ones.
    - Example: x * 2 is replaced with x + x.
3. **Common Subexpression Elimination**
    - Avoids recalculating the same expression multiple times.

- Example: If a + b is computed multiple times, compute it once and reuse the result.

4. **Dead Code Elimination**
   - Removes code that does not affect the output.
   - Example: Unused variable assignments.
5. **Loop Optimizations**

   These are applied to improve loop performance:
   - **Loop Unrolling**: Expands the loop body to reduce the number of iterations.
   - **Loop Fusion**: Combines two adjacent loops with the same range into one.
   - **Loop Invariant Code Motion**: Moves code that doesn't change within the loop to outside the loop.

2. Machine-Dependent Optimization - These optimizations depend on the specific target machine's architecture and instruction set.

Purpose: To exploit hardware features such as registers, pipelines, and special instruction sets to improve performance.

Key Features:

- Applied after the machine code is generated
- Tailored for a specific CPU or processor
- Not portable across different machines

Common Techniques:

1. **Register Allocation**
   - Assigns frequently used variables to CPU registers instead of memory to improve access speed.
2. **Instruction Scheduling**
   - Reorders instructions to avoid pipeline stalls and delays in execution.
3. **Peephole Optimization**
   - Examines a small set of instructions (a "peephole") and replaces inefficient instruction sequences with optimized ones.
   - Example: Replacing two instructions that cancel each other with a single, more efficient one.

4. **Use of Special Addressing Modes**
    - Uses machine-specific addressing capabilities like base+index or offset addressing for faster memory access.
5. **Pipeline Optimization**
    - Arranges instructions to make better use of CPU pipelines and avoid hazards such as delays due to dependencies.

Feature	Machine-Independent Optimization	Machine-Dependent Optimization
Based On	Program logic and structure	Machine architecture and instruction set
Code Level	Intermediate code	Target machine code
Portability	Highly portable across platforms	Not portable; machine-specific
Stage of Application	Before target code generation	After or during target code generation
Focus	Logical simplification	Efficient use of hardware resources
Common Techniques	Constant folding, loop optimization	Register allocation, instruction scheduling

2. Peephole Optimization

Peephole optimization is a technique used in compilers to optimize small sets of instructions generated by the compiler, referred to as a "peephole" or window. This small set of instructions is examined and optimized, usually for improving performance or reducing code size.

Key Points of Peephole Optimization

- **Post-code generation**: Peephole optimization is applied after the source code is converted into target machine code.

- **Machine-dependent optimization**: It is part of machine-dependent optimization, meaning it happens after the code is generated for a specific machine and utilizes machine-specific features like CPU registers or absolute memory references.
- **Repeated optimization**: This process is applied repeatedly on small sections of code (the peephole), improving code efficiency and performance in each pass.

Goals of Peephole Optimization

- **Improving performance**: The main aim is to enhance the execution speed of the compiled code.
- **Reducing memory footprint**: By optimizing the code, less memory is required to store the program.
- **Reducing code size**: Peephole optimization also reduces the size of the target code, making it more efficient.

Working of Peephole Optimization

1. **Identify the Peephole**:
    - The compiler identifies small sections of the generated code that may need optimization. The "peephole" refers to these small windows of code within which the compiler searches for optimization opportunities.
    - The window size varies depending on the optimization being applied.
2. **Apply the Optimization Rule**:
    - The compiler applies predefined optimization rules to the instructions inside the peephole. It searches for specific patterns, such as redundant code or unnecessary instructions.
3. **Evaluate the Result**:
    - After applying optimizations, the compiler checks whether the changes improve the code's performance (speed), memory usage, and overall size.
4. **Repeat**:
    - The compiler repeats these steps, optimizing smaller sections of the code until no further improvements can be made.

Objectives of Peephole Optimization

- **Increase code speed**: By removing redundant or unnecessary instructions.
- **Reduce code size**: By replacing longer sequences of instructions with more compact ones.

- **Eliminate dead code**: Remove unreachable code, redundant assignments, and other unnecessary operations that don't impact the program's output.
- **Simplify the code**: Make the code more understandable and efficient by removing complexities.

Techniques of Peephole Optimization

1. **Constant Folding**:
   - Constant folding involves evaluating constant expressions at compile time, instead of during runtime, to optimize performance.

Example:

Initial Code:

```
int x = 10 + 5;
int y = x * 2;
```

Optimized Code:

```
int x = 15;
int y = x * 2;
```

2. **Strength Reduction**:
   - Replaces expensive operations with cheaper ones, improving performance.

Example:

Initial Code:

```
int x = y / 4;
```

Optimized Code:

```
int x = y >> 2;  // Replace division by 4 with a right shift
```

3. **Redundant Load and Store Elimination**:
   - This technique reduces unnecessary memory accesses, especially when values are loaded multiple times but not changed.

Example:

Initial Code:

```
int x = 5;
int y = x + 10;
int z = x + 20;
```

Optimized Code:

```
int x = 5;
int y = x + 10;
int z = y + 10;  // Reuse 'y' instead of loading 'x' again
```

4. **Null Sequences Elimination**:
   - Eliminates sequences of instructions that do not affect the program's final output.

Example:

Initial Code:

```
int x = 5;
int y = 10;
int z = x + y;
x = 5;  // redundant instruction
```

Optimized Code:

```
int x = 5;
int y = 10;
int z = x + y;  // Remove the redundant 'x = 5'
```

3. Directed Acyclic Graph (DAG)

A **Directed Acyclic Graph (DAG)** is a data structure used to represent the flow of control and values within a program, especially for the transformation and optimization of **basic blocks** in the intermediate code of a compiler. It is an essential tool in modern compilers for performing optimizations such as common subexpression elimination and simplifying the control flow of a program.

Characteristics of Directed Acyclic Graph (DAG)

1. **Directed**: The edges of the graph have a direction, indicating the flow from one node to another.
2. **Acyclic**: There are no cycles or loops in the graph. This means that no node can be reached from itself by traversing the edges.
3. **Nodes**:
   - **Leaf nodes**: These represent variables or constants.

- **Interior nodes**: These represent operators (such as +, -, *, etc.).
- **Identifiers**: Each node may have an identifier, typically representing the computed value or the result of an operation.

4. **Topological Order**: A DAG can be topologically ordered, meaning the nodes can be sorted so that every directed edge goes from an earlier node to a later node in the sequence.
5. **Transitive Closure and Reduction**: These concepts apply to DAGs to optimize and simplify relationships between nodes.

Algorithm for Construction of Directed Acyclic Graph

DAG construction is based on three common cases encountered in three-address code generation. Here's how it is built:

1. **Case 1**: x = y op z
   - If y is not defined, create a node for y.
   - If z is not defined, create a node for z.
   - Create an operator node for the operation op and connect y and z as its children.
   - The result x is stored in the identifier for this operation node.
2. **Case 2**: x = op y
   - If there is already an operator node for op with one child (i.e., y), reuse that node.
   - If not, create a new node for op and connect y to it.
3. **Case 3**: x = y
   - If y is defined previously, reuse the node for y.

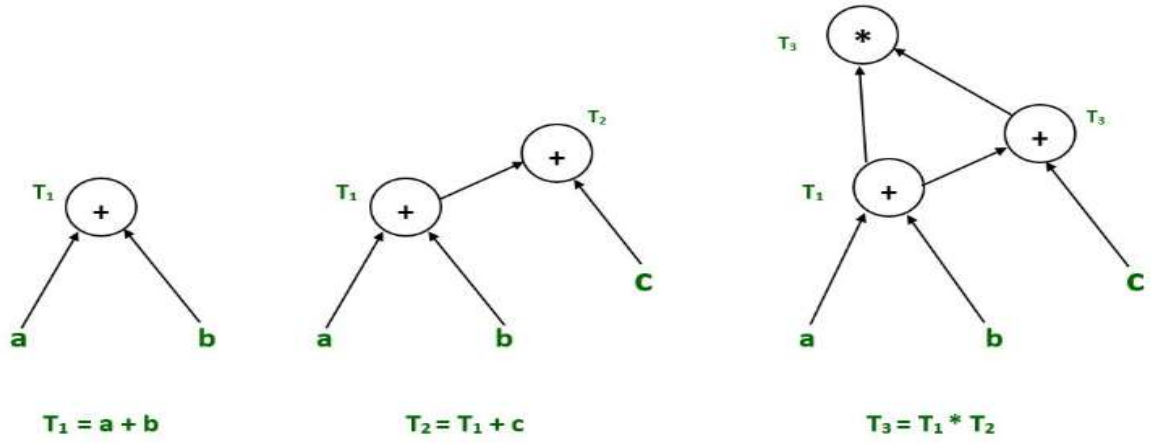
Applications of Directed Acyclic Graphs (DAG)

1. **Common Subexpression Elimination**: Identifies and reuses previously computed values to avoid redundant calculations.
2. **Optimization**: Enables techniques like algebraic simplification and constant folding within basic blocks.
3. **Data Flow Representation**: Efficiently shows the flow of values and computations between basic blocks.

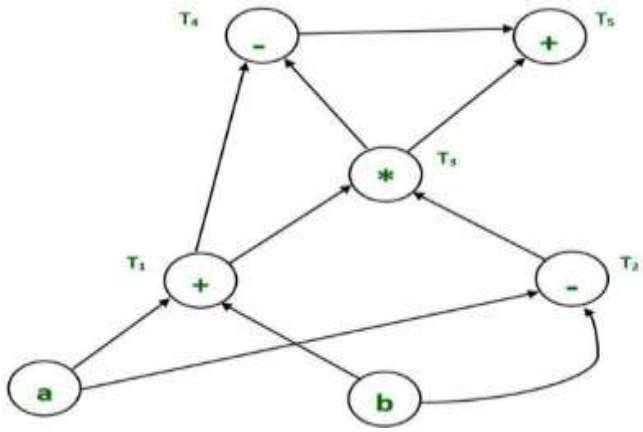
4. **Value Propagation**: Tracks how computed values are used in subsequent operations, removing unnecessary computations.
5. **Instruction Selection**: Helps select the most efficient machine instructions during code generation.
6. **Code Simplification**: Simplifies code by eliminating redundant operations and reducing complexity.

Example:

1. ***T1 = a + b***
   T2 = T1 + c
   T3 = T1 x T2

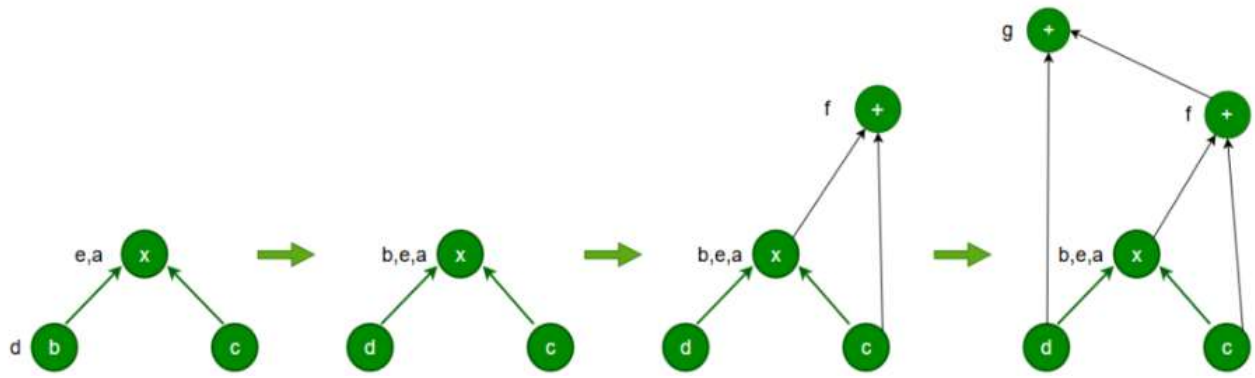


2. ***T1 = a + b***
   T2 = a – b
   T3 = T2 * T1
   T4 = T1 – T3
   T5 = T4 + T3



3.

a = b x c
d = b
e = d x c
b = e
f = b + c
g = f + d



4. Optimization of Basic Blocks

A basic block is a **sequence of consecutive three-address statements** that has:

- **Single entry point**: Control enters only at the first instruction.
- **Single exit point**: Control leaves only at the last instruction, without any jumps or halts in between.

Steps to Partition a Sequence into Basic Blocks

To divide a sequence of three-address code into basic blocks, follow this **algorithm**:

Step 1: Identify Leaders

A **leader** is the first statement of a basic block. The rules to find leaders:

1. **The first statement** in the code is a leader.
2. Any statement that is the **target of a jump (conditional/unconditional)** is a leader.
3. Any statement that **immediately follows a jump statement** is a leader.

Step 2: Identify Basic Blocks

- Once leaders are found, a basic block is formed from:

- A **leader** up to (but not including) the **next leader**.
- The last basic block ends at the last instruction.

Types of Basic Block Optimization

Optimizations within basic blocks are broadly classified into:

1. Structure-Preserving Transformations

2. Algebraic Transformations

1. Structure-Preserving Transformations: These are transformations that **do not change the structure** or **sequence** of code logic, but **reorganize, remove, or rename** parts to improve performance.

a) Dead Code Elimination

Dead code refers to code that:

- Is **never executed**, or
- Produces results that are **never used**

Goal: Remove such code to **reduce program size** and **execution time**.

Example:

```
int x = 2;
if (x > 2)
   cout << "Dead code";   // Never true
else
   cout << "Optimized";
```

Optimized:

```
int x = 2;
cout << "Optimized";   // Dead code eliminated
```

b) Common Subexpression Elimination (CSE): If an **expression** is **recomputed multiple times** and its operands **haven't changed**, compute it once and **reuse** it.

Tools Used: **DAG (Directed Acyclic Graph)** is used to detect and optimize common subexpressions.

Example:

```
t1 = a + b;
t2 = a + b;  // Same expression again
```

Optimized:

```
t1 = a + b;
t2 = t1;
```

c) Renaming of Temporary Variables: Temporary variables that hold intermediate values can be **renamed** for clarity, reuse, or to avoid **conflicts** in further transformations.

Example:

```
t = a + b;  // 't' is a temp
```

Optimized:

```
x = a + b;  // Renamed to 'x' without changing the value
```

d) Interchange of Independent Statements: If **two adjacent statements** are **independent** (i.e., no data dependency), they can be **reordered** for better performance or resource allocation.

Example:

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

Optimized:

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

2. Algebraic Transformations: These transformations use **algebraic rules** (like arithmetic identities) to simplify or change expressions into **more efficient** forms.

a) Constant Folding: Expressions involving **only constants** are evaluated at **compile-time**, instead of runtime.

Example:

```
x = 2 * 3 + y;
```

Optimized:

```
x = 6 + y;   // 2*3 evaluated at compile-time
```

b) Copy Propagation: Replaces the **occurrence of a variable** that is simply a **copy of another variable or constant**.

Two types:

1. **Variable Propagation**
2. **Constant Propagation**

Variable Propagation Example:

```
x = y;
z = x + 2;
```

Optimized:

```
z = y + 2;
```

Constant Propagation Example:

```
x = 3;
z = x + a;
```

Optimized:

```
z = 3 + a;
```

c) Strength Reduction: Replace **expensive operations** (like multiplication/division) with **cheaper alternatives** (like addition/bit shifts).

Examples:

```
x = 2 * y;      // Costly
x = y + y;      // Cheaper (Addition)
x = y << 1;     // Bit-shift (Even faster on hardware)
```

5. Loop Optimization Techniques

Loops are repeated multiple times, so **optimizing them** has a **large impact** on performance.

1. Code Motion (Loop-Invariant Code Motion): Move **invariant computations** (values that do **not change** inside the loop) **outside** the loop to reduce redundant calculations.

Example:

Before:

```
for (int i = 0; i < n; i++) {
    x = 10;        // Constant, doesn't change
    y = y + i;
}
```

After:

```
x = 10;
for (int i = 0; i < n; i++) {
    y = y + i;
}
```

2. Induction Variable Elimination: Remove **extra variables** in loops that simply follow a pattern and can be replaced with a simpler expression.

Example:

Before:

```
i1 = 0; i2 = 0;
for (int i = 0; i < n; i++) {
    A[i1++] = B[i2++];
}
```

After:

```
for (int i = 0; i < n; i++) {
    A[i] = B[i];    // Cleaner with one index
}
```

3. Loop Merging (Combining): If two loops **iterate over the same range** and don't interfere with each other, they can be **merged** into one.

Example:

Before:

```
for (i = 0; i < n; i++) A[i] = i + 1;
for (j = 0; j < n; j++) B[j] = j - 1;
```

After:

```
for (i = 0; i < n; i++) {
    A[i] = i + 1;
    B[i] = i - 1;
}
```

4. Loop Unrolling: Replace a loop with a **repeated sequence** of the loop body to **reduce the overhead** of loop control (especially useful for small loops).

Example:

Before:

```
for (i = 0; i < 3; i++) cout << "Cd";
```

After:

```
cout << "Cd";
cout << "Cd";
cout << "Cd";
```

6. Data Flow Analysis (DFA)

Data Flow Analysis (DFA) is a compiler technique used to analyze how data moves through different parts of a program. It focuses on tracking:

- Where values are defined
- Where they are used
- How they are propagated, modified, or killed (overwritten)

Importance of DFA

- Helps identify optimization opportunities
- Detects bugs such as using uninitialized variables

- Improves the efficiency and correctness of compiled code

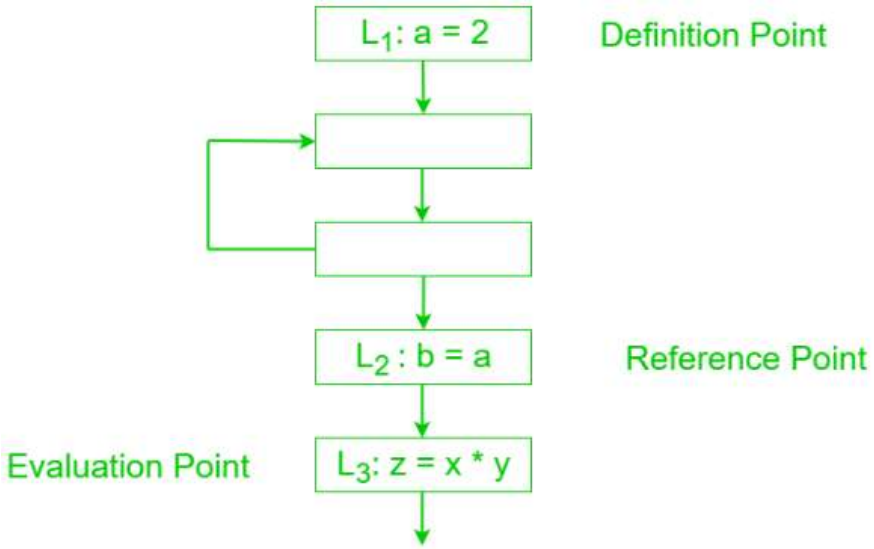
Program Representation – Control Flow Graph (CFG)

Before applying data flow analysis, the program is represented as a Control Flow Graph:

- **Nodes** represent basic blocks
- **Edges** represent the flow of control between blocks

Terminologies in Data Flow Analysis

Term	Meaning
Definition Point	A point in the program where a variable is assigned a value (e.g., x = 5;)
Reference Point	A point where a variable is used (e.g., y = x + 1;)
Evaluation Point	A point where an expression is evaluated



Types of Global Data Flow Analysis

1. Reaching Definitions (RD)

- A definition of a variable reaches a point p if there is a path from d to p such that d is not overwritten along the path
- Used in constant and variable propagation

Example:

```
1: a = 5;   // definition of 'a'
2: ...
3: b = a + 2;  // 'a' reaches this point if not redefined
```

2. Live Variable Analysis (LVA)

- A variable is live at point p if its value is used in the future before being overwritten
- Used for register allocation and dead code elimination

Example:

```
1: a = 10;
2: b = a + 1; // 'a' is live here
3: a = 7;     // after this, old value of 'a' is dead
```

3. Available Expressions (AE)

- An expression is available at a point if it has already been computed and none of its operands have changed
- Used for common subexpression elimination

Example:

```
1: t1 = a + b;
2: ...
3: t2 = a + b;  // expression is available → reuse t1
```

4. Constant Propagation

- Tracks values of variables that are constant and replaces them wherever possible
- Used for constant folding

Example:

```
1: x = 3;
2: y = x + 4;  // becomes y = 3 + 4 → y = 7
```

5. Busy Expression Analysis

- An expression is busy at a point if it will be used later along all paths and none of its operands are modified before then

- Used for code movement optimizations

Data Flow Properties Summary

Property	Description	Example Use Case
Reaching Definition	Definition reaches a use without being killed	Variable propagation
Live Variable	The variable is used in the future before redefinition	Register allocation
Available Expression	The expression has been computed earlier and is still valid	Common Subexpression Elimination
Constant Propagation	Replace variables with known constant values	Constant folding
Busy Expression	Expression will be used again soon without change	Code motion optimization

Features of Data Flow Analysis

1. Identifies dependencies between statements
2. Detects dead code that can be removed
3. Enables optimizations like constant folding and loop invariant code motion
4. Detects errors such as the use of uninitialized variables
5. Handles complex control flow like loops and conditionals
6. Can be extended to analyze across multiple functions (interprocedural analysis)
7. Can be scaled to large programs with thousands of lines

Advantages of Data Flow Analysis

- Improves performance and efficiency of compiled code
- Removes unnecessary computations
- Identifies and removes dead or redundant code
- Helps in better register usage and memory allocation
- Enables global optimizations across the control flow graph

7. Efficient Data Flow Algorithm

An Efficient Data Flow Algorithm is a systematic way to:

- Solve data flow equations over all basic blocks of a program
- Quickly compute accurate IN and OUT values for each block
- Minimize repeated computations
- Ensure that the algorithm works even for large and complex programs

This algorithm helps in optimizing the intermediate code by identifying useful or redundant data.

Term	Meaning
Basic Block (B)	A straight-line sequence of code with no branches except at the entry and exit
IN[B]	Information entering block B
OUT[B]	Information leaving the block B
GEN[B]	Information generated by block B
KILL[B]	Information invalidated by block B
Pred(B)	Predecessors of B in CFG
Succ(B)	Successors of B in CFG

General Algorithm Framework

Goal: Compute the IN[B] and OUT[B] values for every basic block B, based on the type of analysis.

Steps:

1. **Initialization**:
    - Set all IN[B] and OUT[B] to empty sets or universal sets based on the direction and type of analysis.
2. **Iterative Computation**:
    - For each basic block B, repeat until convergence:
        - Compute IN[B] and OUT[B] using:
            - Forward analysis:
                - IN[B] = Union of OUT[P] for all P in Pred(B)
                - OUT[B] = GEN[B] ∪ (IN[B] – KILL[B])

- Backward analysis:
  - OUT[B] = Union of IN[S] for all S in Succ(B)
  - IN[B] = USE[B] ∪ (OUT[B] – DEF[B])

3. **Stop Condition**:
   - When the IN and OUT sets of all blocks stop changing (i.e., when they reach a fixed point).

Types of Data Flow Analyses

Type of Analysis	Direction	Used For
Reaching Definitions	Forward	Constant/Variable Propagation
Live Variable Analysis	Backward	Register Allocation, Dead Code Elimination
Available Expressions	Forward	Common Subexpression Elimination
Busy Expressions	Backward	Code Motion Optimization

Optimizations for Efficiency

1. **Worklist Algorithm**:
   - Only recompute IN and OUT sets for blocks that may be affected by changes, reducing redundant computations.
2. **Bit Vector Representation**:
   - Represent sets (GEN, KILL, IN, OUT) as bit vectors for faster set operations using AND, OR, and NOT.
3. **Monotonic Transfer Functions**:
   - Ensure that repeated application leads to convergence (no infinite loops), which guarantees algorithm termination.

Why Is It Efficient?

Feature	Explanation
Scalability	Can be used on large programs with thousands of basic blocks
Precision	Solves data flow equations accurately until a fixed point is reached

Feature	Explanation
Speed	Uses bitwise operations and avoids unnecessary computations
Generality	Applicable to both forward and backward data flow problems

Applications in Compiler Optimization

Optimization Type	Based On
Dead Code Elimination	Live Variable Analysis
Register Allocation	Live Variable Analysis
Constant and Variable Propagation	Reaching Definitions
Common Subexpression Elimination	Available Expressions
Code Motion	Busy Expressions