UNIT I – Introduction

---

1. Objective, Scope, and Outcome of the Course

Introduction:
Compiler Design is a core subject in computer science that explains how high-level programming languages are translated into machine-understandable code. It combines concepts of **automata theory**, **programming languages**, and **system software design**.

Objective:
The main objectives are:

- To understand the **phases of compilation** from source to target code.
- To study **lexical, syntax, and semantic analysis** techniques.
- To learn **parsing algorithms** (top-down and bottom-up).
- To understand **code generation and optimization** processes.
- To design **syntax-directed translation schemes**.
- To develop **error detection and recovery** methods.

Scope:
Compiler design has vast applications. It helps in:

- Developing new **programming languages**.
- Building **interpreters** and **assemblers**.
- Understanding **machine architecture** and **code optimization**.
- Enhancing **execution speed** and **memory management** of programs.
- Supporting **cross-platform development** using cross-compilers.

Outcome:
After studying compiler design, a student can:

- Identify different compiler phases and their working.
- Build lexical analyzers and parsers.
- Generate intermediate and target machine code.
- Understand error-handling and recovery techniques.
- Optimize the code for better performance.

Conclusion:
Thus, the compiler design course provides a **complete understanding of program translation** and helps students build efficient language processors. It forms the foundation for advanced system software and language development.

---

2. Compiler

Definition:
A **compiler** is a system software that converts the entire source program written in a high-level language into an equivalent target program, usually in machine code or assembly.

Working Principle:
The compiler works in several **phases**, each responsible for a specific task in the translation process. The phases are divided into **front-end** (analysis) and **back-end** (synthesis).

Phases of a Compiler:

1. **Lexical Analysis:**
   Converts a stream of characters into meaningful tokens using a lexical analyzer.
2. **Syntax Analysis (Parsing):**
   Uses grammar rules to form a parse tree from tokens.
3. **Semantic Analysis:**
   Checks the meaning and type correctness of the program.
4. **Intermediate Code Generation:**
   Produces intermediate code (e.g., Three Address Code).
5. **Code Optimization:**
   Improves performance by removing redundant operations.
6. **Code Generation:**
   Translates intermediate code into target machine code.
7. **Error Handling:**
   Detects and manages lexical, syntax, and semantic errors.

Advantages of Compiler:

- Translates entire program at once, leading to faster execution.
- Optimizes code automatically.
- Helps identify complex errors before execution.

Examples:

- GCC (GNU Compiler Collection) for C/C++
- Java Compiler (javac)
- Rust Compiler (rustc)

Conclusion:
A compiler is the heart of language processing. It bridges the gap between human-readable source code and machine-executable instructions, ensuring efficiency and accuracy in program execution.

---

3. Translator

Definition:
A **translator** is a general term for any software that converts code from one language to another. It is an umbrella term for **assemblers, interpreters, and compilers**.

Types of Translators:

1. **Assembler:**
   Converts assembly language into machine language.
   Example: MASM, NASM.
2. **Interpreter:**
   Translates and executes code line-by-line.
   Example: Python interpreter.
3. **Compiler:**
   Converts entire source code to object code at once.
   Example: GCC compiler.

Functions of a Translator:

- Checks syntax and semantics of the source program.
- Reports and handles errors.
- Converts code efficiently into target representation.
- Maintains **symbol tables** for identifiers and variables.
- Performs **optimization** and **linking** for final execution.

Importance:

- Enables **portability** across platforms.

- Reduces human effort in writing machine code.
- Provides **error diagnostics** to improve code quality.

Difference between Compiler and Interpreter:

Feature	**Compiler**	**Interpreter**
Execution	Entire program at once	Line by line
Speed	Faster execution	Slower
Output	Object code generated	No object code
Error handling	Displays all errors after compilation	Stops at first error

Conclusion:
Translators play a vital role in software development by converting human-friendly code into hardware-executable instructions. Understanding translators is key to understanding how programming languages function internally.

---

4. Interpreter Definition

Definition:
An **interpreter** is a program that translates and executes code **line by line** instead of compiling the entire source at once. It is commonly used in scripting and dynamic languages such as Python, JavaScript, and Ruby.

Working:

1. Reads a single line or instruction from the source code.
2. Converts it into an intermediate form.
3. Executes it immediately.
4. Continues this process until the end of the program.

Advantages:

- Easier for debugging and testing.
- Platform-independent (no machine code file).
- Useful for beginners and rapid application development.

Disadvantages:

- Slower execution compared to compiled programs.

- Each time the program runs, translation occurs again.

Examples:

- Python Interpreter
- Node.js for JavaScript
- Ruby Interpreter

Difference from Compiler:
While compilers translate the entire program into machine code before running, interpreters directly execute each line. Hence, interpreters are ideal for scripting and rapid development, whereas compilers are better for performance-critical applications.

Conclusion:
Interpreters simplify execution and debugging, making them indispensable in modern development environments, though they trade off speed for flexibility.

---

5. Phase of Compiler

Introduction:
The compiler does not translate source code into machine code in a single step. It performs translation in several well-defined stages called **phases of the compiler**. Each phase performs a specific function and passes its output to the next phase.

The compiler is broadly divided into **two parts**:

- **Analysis (Front End)** — breaks the source code into components and checks correctness.
- **Synthesis (Back End)** — constructs the target program from the analyzed information.

Phases of Compiler:

1. **Lexical Analysis:**
    - Converts characters into a stream of tokens.
    - Removes white spaces and comments.
    - Example: int a = 5; → Tokens: [int], [a], [=], [5], [;]
    - Tool used: **LEX**

2. **Syntax Analysis (Parsing):**
    - Checks if tokens follow the grammar of the language.
    - Builds a **parse tree** or **syntax tree**.
    - Example: Detects missing brackets or misplaced semicolons.
3. **Semantic Analysis:**
    - Checks for **meaning and logic** of the code.
    - Verifies type compatibility, variable declarations, etc.
    - Example: Prevents assigning float to int without casting.
4. **Intermediate Code Generation:**
    - Produces an intermediate representation (IR) like **Three Address Code (TAC)**.
    - It is independent of machine architecture.
    - Example: a = b + c * d →
    - t1 = c * d
    - a = b + t1
5. **Code Optimization:**
    - Improves performance by removing redundancies.
    - Example: Common sub-expression elimination, dead code removal.
6. **Code Generation:**
    - Converts intermediate code into machine code.
    - Allocates registers and generates efficient instructions.
7. **Code Linking and Loading:**
    - Links different modules and loads the final program into memory.

Conclusion:
Each phase plays a vital role in ensuring correctness, optimization, and efficiency of the compiled program. Understanding compiler phases helps in building reliable language processors and interpreters.

---

6. Bootstrapping

Definition:
Bootstrapping is the process of writing a compiler (or any system software) in the same programming language it is meant to compile. In simple terms, it means *a compiler compiling itself*.

Explanation:
Suppose you want to write a compiler for the language **L**. You have three choices:

1. Write it in **machine language** — difficult and time-consuming.
2. Write it in another existing language — feasible.
3. Write it in **L itself** — known as *bootstrapping*.

Process of Bootstrapping:

1. **Step 1:** Write a simple version of the compiler in another language (like Assembly).
2. **Step 2:** Use that compiler to compile a more advanced version written in L.
3. **Step 3:** Repeat until the compiler can compile its own source code.

Example:
The C compiler was originally written in Assembly language.
Later, developers rewrote the compiler in C itself and used the old compiler to compile the new one — this is **bootstrapping**.

Advantages:

- Easier maintenance and development.
- Demonstrates trust in the target language.
- Simplifies porting the compiler to new platforms.

Disadvantages:

- Requires an initial compiler or translator to start the process.
- Debugging can be complex because errors affect both compiler and source.

Conclusion:
Bootstrapping is an elegant and self-sustaining approach to compiler creation. It

represents the maturity of a programming language when it can define and compile itself.

---

7. Review of Finite Automata

Introduction:
Finite Automata (FA) are mathematical models used in the **Lexical Analysis phase** of compilers to recognize patterns like keywords, identifiers, numbers, and operators. They form the foundation for **token recognition**.

Types of Finite Automata:

1. **Deterministic Finite Automata (DFA):**
    - For each state and input symbol, there is **exactly one** transition.
    - Easy to implement in compilers.
    - Used by lexical analyzers for recognizing tokens.
2. **Non-Deterministic Finite Automata (NFA):**
    - Can have **multiple transitions** for the same input symbol.
    - Easier to construct but harder to implement directly.
    - Usually converted to DFA using **subset construction** method.

Finite Automata Components:

- **States (Q)** – possible positions in the automaton.
- **Input symbols (Σ)** – valid input characters.
- **Transition function (δ)** – rules for state changes.
- **Start state (q_0)** – where input processing begins.
- **Final states (F)** – accepting or valid token states.

Example:
To recognize identifier tokens (e.g., variable names starting with a letter):

State q0 → if letter → q1

State q1 → if letter/digit → q1 (loop)

Final state: q1

This DFA accepts strings like x, sum1, temp, etc.

Applications in Compiler Design:

- Used by **Lexical Analyzer** to detect valid tokens.
- Foundation for **regular expressions** used in token definitions.
- Helps in **pattern matching** and **token classification**.

Conclusion:
Finite Automata are the theoretical backbone of lexical analyzers. They ensure that only valid token patterns are recognized, making the compilation process structured and reliable.

---

8. Lexical Analyzer

Definition:
A **Lexical Analyzer** (or *scanner*) is the first phase of the compiler. It reads the source code character by character and groups them into meaningful sequences called **tokens**.

Functions of Lexical Analyzer:

1. **Scanning:** Reads the source program character by character.
2. **Token Generation:** Combines characters into tokens like keywords, identifiers, operators, and literals.
3. **Removing Whitespaces & Comments:** Ignores irrelevant characters.
4. **Symbol Table Maintenance:** Keeps record of identifiers and their attributes.
5. **Error Detection:** Identifies lexical errors (e.g., invalid symbol or missing quote).

Example:

Source Code:

int a = 5;

Tokens produced:
<int, keyword>, <a, identifier>, <=, operator>, <5, constant>, <;, delimiter>

Tools:

- **LEX (Lexical Analyzer Generator):** Automatically generates lexical analyzers from regular expression patterns.

Advantages:

- Simplifies the compiler’s design by separating lexical concerns.
- Enhances speed and efficiency of parsing.
- Provides error recovery at the token level.

Conclusion:
The lexical analyzer forms the foundation of the compilation process. It ensures the source program is broken into valid tokens, allowing syntax and semantic analysis to proceed smoothly.

---

9. Input

Introduction:
The **input** to the compiler is the **source program** written in a high-level language such as C, C++, or Java. The input is treated as a stream of characters, which are processed in multiple stages to produce tokens, syntax trees, and finally, machine code. Understanding how a compiler handles input is fundamental because it determines the accuracy and efficiency of the entire compilation process.

Role of Input in Compiler:

1. **Initial Source Code:**
   The compiler starts by reading the source file. The lexical analyzer treats it as a continuous stream of characters.

Example:

int sum = a + b;

The input stream contains characters like i, n, t, s, u, m, =, a, +, b, ;

2. **Input Buffering:**
   To make character reading efficient, the compiler uses **buffers**.
   - Buffers store chunks of characters to reduce disk I/O.
   - Two buffers are often used alternately to speed up scanning.
     This technique ensures smooth reading without repeatedly accessing the disk.
3. **Lexeme Identification:**
   The lexical analyzer reads the input and divides it into **lexemes** — sequences of characters forming valid tokens.
   For example, in int sum = a + b;, the lexemes are int, sum, =, a, +, b, ;
4. **End-of-File (EOF) Detection:**
   A special end marker (EOF) signals that the compiler has reached the end of input.
   Without this, the analyzer might continue reading unintentionally.

Challenges in Input Handling:

- Dealing with whitespaces, tabs, and comments.
- Handling multi-line strings and escape characters.
- Managing buffer overflows and end markers correctly.

Conclusion:
The input phase is the **starting point** of compilation. Efficient input buffering and management ensure that subsequent compiler phases operate smoothly and quickly without wasting time on repeated I/O operations.

---

10. Recognition of Tokens

Definition:
Token recognition is the process of identifying meaningful sequences of characters (lexemes) in the source code and classifying them into **token types** like keywords, identifiers, literals, operators, and punctuation marks.

Structure of a Token:

A token is generally represented as:

<token-name, attribute-value>

Example:
<id, 1>, <num, 5>, <operator, +>

Categories of Tokens:

1. **Keywords:** Predefined words (e.g., int, return, while).
2. **Identifiers:** User-defined names (e.g., x, sum, total).
3. **Constants:** Numeric or character values (e.g., 10, 'A', "Hello").
4. **Operators:** Symbols representing operations (e.g., +, -, *, /).
5. **Punctuations/Delimiters:** Characters like ;, ,, (,), {, }.

Process of Token Recognition:

1. **Input Stream Reading:**
   The lexical analyzer reads one character at a time.
2. **Pattern Matching:**
   The sequence of characters is compared with patterns (regular expressions).
   Example: [a-zA-Z_][a-zA-Z0-9_]* → Identifier.
3. **Token Formation:**
   When a match is found, the analyzer forms a token and sends it to the parser.
4. **Symbol Table Update:**
   Identifiers and constants are stored in the symbol table for later reference.
5. **Error Handling:**
   If a lexeme does not match any pattern, a lexical error is reported.

Example:
For the input code:

int a = b + 10;

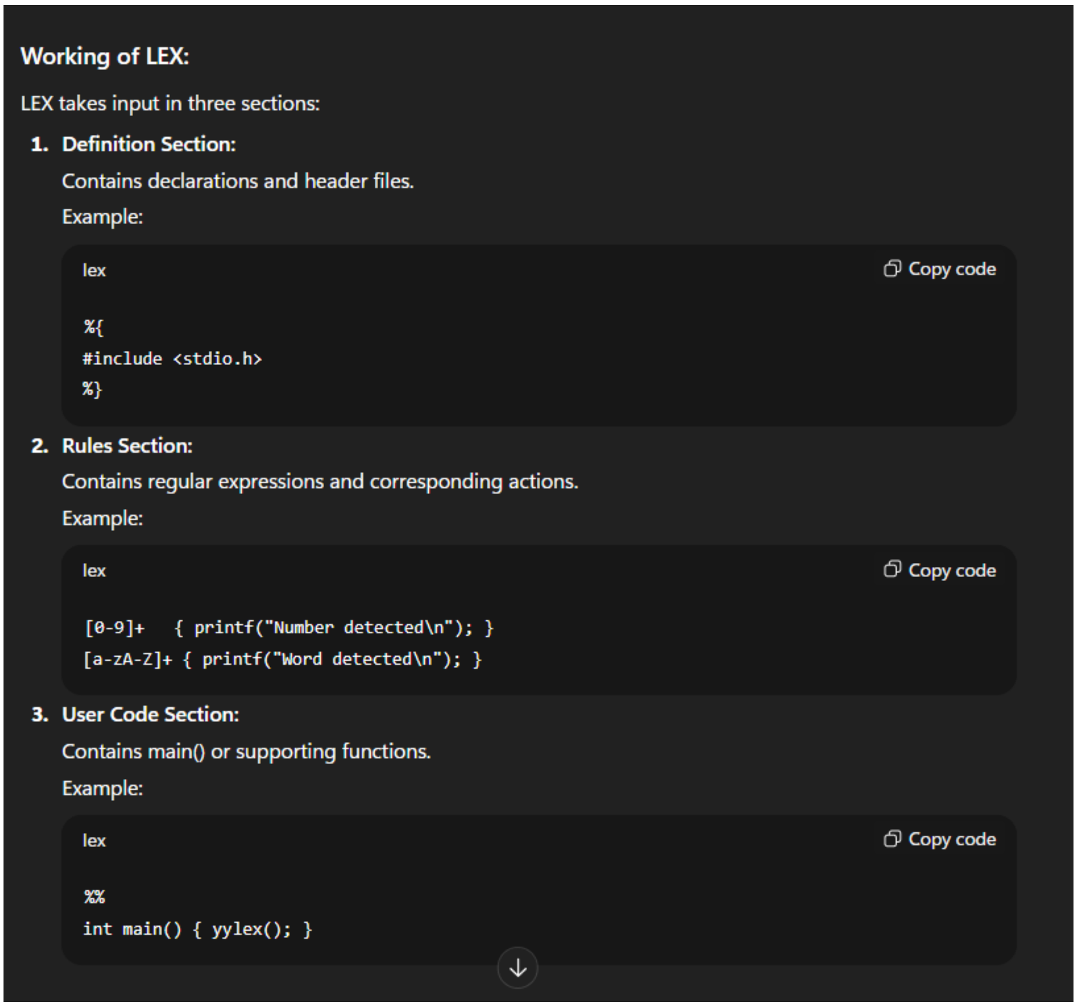
Tokens recognized are:
<int, keyword>, <a, identifier>, <=>, operator>, <b, identifier>, <+, operator>, <10, constant>, <;, delimiter>

Conclusion:
Token recognition is the **foundation of lexical analysis**. Without correctly identifying tokens, the syntax and semantics of the program cannot be understood, making this step essential for accurate compilation.

---

11. Idea about LEX (Lexical Analyzer Generator)

Introduction:
LEX is a widely used **tool for automatic lexical analyzer generation**. It takes the description of tokens in the form of **regular expressions** and produces a C program that performs lexical analysis for those tokens.



Working of LEX:

LEX takes input in three sections:

1. **Definition Section:**
   Contains declarations and header files.
   Example:

```lex
%{
#include <stdio.h>
%}
```

2. **Rules Section:**
   Contains regular expressions and corresponding actions.
   Example:

```lex
[0-9]+    { printf("Number detected\n"); }
[a-zA-Z]+ { printf("Word detected\n"); }
```

3. **User Code Section:**
   Contains main() or supporting functions.
   Example:

```lex
%%
int main() { yylex(); }
```

Process Flow:

1. Write token patterns using regular expressions in a .l file.
2. Run lex file.l to generate lex.yy.c.
3. Compile the generated file using cc lex.yy.c -ll.
4. Run the executable to perform lexical analysis.

Advantages of LEX:

- Automates token recognition and pattern matching.
- Reduces manual coding errors.
- Highly efficient and portable.
- Can be easily integrated with **YACC** (Yet Another Compiler Compiler) for syntax analysis.

Example Output:

For input:

```
123 abc
```

LEX output might be:

```
Number detected

Word detected
```

Conclusion:
LEX simplifies the process of writing lexical analyzers. Instead of manually coding pattern matchers, the developer defines rules using regular expressions, and LEX automatically generates efficient scanner code.

12. Error Handling

Introduction:
Error handling is a crucial part of compiler design. It ensures that the compiler can detect, report, and recover from errors encountered during different phases of compilation, such as lexical, syntax, or semantic errors.

Types of Errors in Compilation:

1. **Lexical Errors:**
   Occur during token generation.
   Example: Invalid character, unclosed string, or unknown symbol.
   → int #a = 5; (Invalid character #)
2. **Syntax Errors:**
   Violate grammatical rules.
   Example: Missing semicolon, unmatched brackets.
   → for(i=0 i<5 i++) (missing semicolon)
3. **Semantic Errors:**
   Related to meaning or logic.
   Example: Type mismatch or undeclared variable.
   → x = "Hello" + 10;

4. **Logical Errors:**
   Program runs but produces incorrect output. (Not detected by compiler.)
5. **Runtime Errors:**
   Occur during execution (division by zero, null pointer).

Error Detection and Recovery:

The compiler must **detect** the error, **report** it to the user, and, if possible, **recover** to continue compilation.

1. Panic Mode Recovery:

Skips input symbols until a synchronizing token is found (e.g., semicolon).
Simple but may skip too much.

2. Phrase Level Recovery:

Performs local correction, such as inserting or deleting symbols to continue parsing.

3. Error Productions:

Grammar is augmented with additional rules to handle common errors.

4. Global Correction:

Attempts to find the minimum number of changes to make the program valid (theoretical).

Error Reporting Methods:

- **Immediate Mode:** Displays error instantly with line number.
- **Batch Mode:** Collects all errors and displays after compilation.

Example Output:

```
Error at line 5: missing semicolon ';'

Error at line 8: undefined variable 'num'
```

Conclusion:
Effective error handling ensures the compiler remains robust, user-friendly, and able to guide programmers toward correcting mistakes without halting unnecessarily.

UNIT – 2: NUMBER SYSTEMS AND CODES

1. Review of CFG (Context-Free Grammar)

A Context-Free Grammar (CFG) is a formal grammar used to describe the syntax of programming languages. It defines how strings in a language can be generated using rules (productions).
A CFG consists of four components:

1. **V – A finite set of variables (non-terminals)**
2. **Σ – A finite set of terminals (tokens or symbols)**
3. **R – A set of production rules of the form *A* → *α*, where *A* is a non-terminal and *α* is a string of terminals and/or non-terminals**
4. **S – A start symbol**

Example:

S → aSb | ε

This grammar generates strings like: ab, aabb, aaabbb, etc.

CFGs are essential because they form the foundation for parsing, helping compilers understand the hierarchical structure of source code. They are powerful enough to represent nested structures such as expressions, loops, and function calls.

---

2. Ambiguity of Grammars

A grammar is ambiguous if there exists more than one parse tree or derivation for a single string in the language.

For example:

E → E + E | E * E | id

For the input id + id * id, we can derive two different parse trees:

- One where addition happens first.
- Another where multiplication happens first.

Ambiguity creates confusion for the compiler — it won't know which interpretation is correct.
Therefore, ambiguous grammars are **undesirable** in compiler design.

Problems of Ambiguity:

- The compiler cannot decide the correct structure of a statement.
- Operator precedence and associativity become unclear.

Solution:

- Rewrite the grammar to remove ambiguity.
- Define precedence rules (e.g., * has higher precedence than +).
- Use **unambiguous grammars** or parser precedence tables.

---

3. Introduction to Parsing

Parsing is the process of analyzing a string according to grammar rules. The parser checks whether the given program is syntactically correct.

Phases in Parsing:

- **Lexical Analysis:** Breaks the code into tokens.
- **Syntax Analysis (Parsing):** Builds a parse tree using tokens and CFG.

Types of Parsers:

1. **Top-Down Parsing:** Begins from the start symbol and tries to reach the input string.
2. **Bottom-Up Parsing:** Begins with the input string and tries to reach the start symbol.

Parsing plays a critical role in detecting syntax errors and generating intermediate representations.

---

4. Top-Down Parsing

In **Top-Down Parsing**, we start from the start symbol and try to derive the input string by repeatedly applying grammar rules.

Key Features:

- Derivations are **leftmost**.
- Predictive and recursive descent parsing are common types.

Advantages:

- Easy to understand and implement manually.

- Works well for simple grammars.

Disadvantages:

- Cannot handle left recursion directly.
- Not suitable for all types of grammars.

---

5. LL Grammars and Parsers

An **LL parser** reads input **Left-to-right (first L)** and constructs a **Leftmost derivation (second L)**.

LL Grammar Requirements:

- Should be free of **left recursion**.
- Should not have **common prefixes** (use left factoring).

LL(1) Parser:
Uses one lookahead symbol to make decisions.
The parser uses a **predictive parsing table** with entries [Non-Terminal, Terminal].

Example:

E → TE'

E' → +TE' | ε

Here, LL(1) parsing table is used for efficient parsing.

---

6. Error Handling of LL Parser

Errors in parsing occur when the next input symbol does not match any expected symbol.

Error Handling Techniques:

1. **Panic Mode Recovery:** Skip symbols until a synchronizing token is found.
2. **Phrase-Level Recovery:** Modify the input minimally to fix errors.
3. **Error Productions:** Include known error rules in grammar.
4. **Global Correction:** Tries to find minimum changes needed to make input valid.

---

7. Recursive Descent Parsing

A **recursive descent parser** is a top-down parser built from a set of recursive procedures — one for each non-terminal.

Example:

E → TE’

E’ → +TE’ | ε

Functions:

E() { T(); E’(); }

E’() { if(lookahead == '+') { match('+'); T(); E’(); } }

Advantages:

- Simple to implement.
- Works for small grammars.

Disadvantages:

- Left recursion must be removed.
- Not efficient for complex grammars.

---

8. Predictive Parsers

Predictive parsing is a non-recursive form of top-down parsing.
It uses a parsing table to predict which rule to apply next using the lookahead symbol.

Algorithm:

1. **Use a stack initialized with start symbol.**
2. **Read input symbol.**
3. **Use parsing table to select production rule.**
4. **Replace non-terminal with rule’s right side.**
5. **Repeat until stack and input are empty.**

Advantages:

- **No backtracking needed.**
- **Efficient for LL(1) grammars.**

---

9. Bottom-Up Parsing

Bottom-Up Parsing starts with the input string and tries to reduce it to the start symbol by inverting production rules.

It constructs rightmost derivations in reverse.
Main types:

- Shift-Reduce Parsing
- LR Parsing

Advantages:

- Handles larger classes of grammars.
- Works well for real-world programming languages.

---

10. Shift-Reduce Parsing

This is a bottom-up parsing technique that uses a stack.
Steps:

- Shift: Move input symbol to stack.
- Reduce: Replace symbols on stack that match RHS of a rule with its LHS.
- Accept: When start symbol remains and input is empty.
- Error: If no valid rule applies.

Example:

Grammar: $E \rightarrow E + E \mid id$

Input: id + id

Stack operations simulate how the parser builds structure.

---

11. LR Parsers

LR parsers are the most powerful deterministic bottom-up parsers.
“L” means scanning input left-to-right, “R” means constructing a rightmost derivation in reverse.

Types:

- SLR (Simple LR)

- Canonical LR
- LALR (Lookahead LR)

Features:

- Can handle a wide range of grammars.
- Detect errors quickly.

---

12. Construction of SLR, Canonical LR, and LALR Parsing Tables

1. SLR: Simplest LR parser, uses FOLLOW sets for reductions.
2. Canonical LR: Uses full LR(1) items; large but accurate.
3. LALR: Combines states of Canonical LR to reduce size while maintaining accuracy.

Construction Steps:

- Compute items and closure.
- Build Goto and Action tables.
- Use lookahead symbols for valid reductions.

---

13. Parsing with Ambiguous Grammar

Ambiguous grammars lead to multiple parse trees for the same input.
Example: E → E + E | E * E | id
To handle ambiguity:

- Rewrite grammar with precedence rules.
- Use operator precedence or LR parsing.
- Avoid ambiguity by restructuring productions.

---

14. Operator Precedence Parsing

A special type of bottom-up parsing based on operator precedence.
It assigns precedence and associativity to operators and parses accordingly.

Steps:

- Define precedence table.

- Shift or reduce based on operator precedence.
- No explicit stack of grammar rules.

Example:
* > +, so multiplication performed before addition.

---

15. Introduction of Automatic Parser Generator: YACC

YACC (Yet Another Compiler Compiler) is a tool that generates parsers automatically from grammar definitions.

- Input: Grammar rules written in BNF form.
- Output: A parser written in C code.

Features:

- Supports LR parsing.
- Handles complex grammars easily.
- Often used with LEX, the lexical analyzer generator.

---

16. Error Handling in LR Parsers

In LR parsing, errors are detected as soon as no valid action is available in the parsing table.

Common Recovery Methods:

- Panic Mode: Discard input symbols until a synchronizing token appears.
- Phrase Level Recovery: Try inserting/deleting symbols.
- Error Productions: Introduce specific rules for expected errors.
- Global Correction: Find minimal corrections to make input valid.

🧠 Unit III – Syntax Directed Translation and Intermediate Code

---

1. Syntax Directed Definitions (SDD)

A Syntax Directed Definition (SDD) is a formal way to attach semantic rules or actions to a grammar's production rules. It defines how meaning (semantics) is derived from the structure (syntax) of the program.

Key Concept:
Each grammar symbol (terminal or non-terminal) is associated with attributes, and each production rule has semantic rules to compute these attributes.

There are two types of attributes:

1. Synthesized Attributes: Computed from child nodes and passed upward in the parse tree.
2. Inherited Attributes: Computed from parent or sibling nodes and passed downward.

Example:

E → E1 + T

E.val = E1.val + T.val

Here, val is a synthesized attribute that stores the evaluated expression's value.

Uses:

- To specify type checking, code generation, symbol table management, and intermediate code construction.

Advantages:

- Provides a systematic way to integrate semantics with syntax.
- Forms the foundation for Syntax Directed Translation Schemes (SDT).

---

2. Construction of Syntax Trees

A Syntax Tree is a simplified representation of the structure of a source program, showing the hierarchical relationships of operators and operands. Unlike parse trees, it omits unnecessary nodes and focuses on meaningful syntactic constructs.

Example:
For the expression:

a + b * c

The syntax tree looks like:

```
   +
  /  \
 a    *
    / \
   b   c
```

Steps to Construct Syntax Tree:

1. Parse the input string according to grammar.
2. Create tree nodes for each operator and operand.
3. Connect nodes to represent operator precedence and associativity.

Purpose:

- Used for intermediate code generation, optimization, and semantic analysis.

---

3. S-Attributed Definition

An S-Attributed Definition is a type of SDD that uses only synthesized attributes. This means all attributes are computed from child nodes and passed upward.

Example:

E → E1 + T

E.val = E1.val + T.val

T → id

T.val = id.lexval

Properties:

- Very simple to implement.
- Suits bottom-up parsing.
- Commonly used in postfix code generation and arithmetic evaluation.

Advantages:

- Works well with LR parsers.

- **Implementation is straightforward since evaluation follows a bottom-up approach.**

---

4. L-Attributed Definition

An L-Attributed Definition allows both inherited and synthesized attributes, but with restrictions:

- **Inherited attributes of a symbol can depend only on:**
  - **Attributes of the parent, and**
  - **Attributes of siblings that appear to its left in the production.**

Example:

A → BC

B.inh = A.inh

C.inh = f(B.syn)

Uses:

- **Useful for type checking, symbol table management, and parameter passing.**

Advantages:

- **Supports top-down evaluation order.**
- **Works well with recursive descent parsers.**

---

5. Top-Down Translation

Top-down translation combines parsing and semantic analysis. It constructs the translation during leftmost derivation, moving from the start symbol to the terminals.

Steps:

1. **Start from the root (start symbol).**
2. **Expand using production rules.**
3. **Compute attributes and perform actions during traversal.**

Example:
For an expression grammar:

E → T E’

E’ → + T { print('+') } E’ | ε

T → id { print(id) }

Input: a + b + c
Output (postfix): ab+c+

Advantages:

- **Easy to combine with predictive parsing.**
- **Efficient for syntax-directed translation.**

---

6. Intermediate Code Forms using Postfix Notation

Postfix (Reverse Polish Notation) represents expressions where operators follow their operands.
Example:

Infix: a + b * c

Postfix: a b c * +

Why Postfix?

- **No need for parentheses.**
- **Operator precedence is implicit.**
- **Easier to evaluate using stacks or generate machine code.**

Postfix Code Generation Steps:

1. **Convert infix expression to postfix.**
2. **Push operands to stack.**
3. **Pop operands when operator encountered and generate operation code.**

Benefits:

- **Easy to interpret and compile.**
- **Forms the foundation for three-address code.**

---

7. DAG (Directed Acyclic Graph)

A DAG is a graphical representation of expressions that helps detect common sub-expressions and optimize computations.
Each node represents an operation or operand, and edges represent data flow.

Example:
Expression:

a * b + a * b

DAG representation:

```
  +

 / \

 *   *

/ \ / \

a b a b
```

Optimization merges identical sub-expressions, reducing redundant calculations.

Uses:

- Used in code optimization and intermediate code generation.
- Helps the compiler reuse results of repeated computations.

---

8. Three Address Code (TAC)

Three Address Code (TAC) is an intermediate representation where each instruction contains at most three addresses (operands).

General Format:

x = y op z

Example:
For a + b * c, TAC:

t1 = b * c

t2 = a + t1

Types of TAC Statements:

1. Assignment: x = y op z
2. Unary: x = op y

3. **Copy: x = y**
4. **Conditional: if x relop y goto L1**
5. **Unconditional: goto L2**

Advantages:

- **Machine-independent representation.**
- **Simplifies code optimization and generation.**

---

9. TAC for Various Control Structures

a) If-Else:

```
if x < y goto L1
t1 = x - y
goto L2
L1: t1 = y - x
L2:
```

b) While Loop:

```
L1: if x < y goto L2
goto L3
L2: body of loop
goto L1
L3:
```

c) For Loop:

```
init
L1: condition
if false goto L3
body
update
goto L1
L3:
```

Advantages:

- **Makes flow control clear and structured.**
- **Simplifies translation into machine code.**

---

10. Representing TAC using Triples and Quadruples

TAC can be stored in two common data structures:

a) Triples:
Index-based representation; results are referred to by index.

```
(0) * b c

(1) + a (0)
```

b) Quadruples:
Use named temporary variables explicitly.

```
(0) (*, b, c, t1)

(1) (+, a, t1, t2)
```

Difference:

- **Triples save memory but are harder to modify.**
- **Quadruples are easier for optimization and debugging.**

---

11. Boolean Expressions and Control Structures

Boolean expressions are logical statements that result in true or false values.
Used in if, while, and for statements.

Example:

```
if (a < b and c > d)
```

TAC Representation:

```
if a < b goto L1

goto L2

L1: if c > d goto L3

L2:
```

Short-Circuit Evaluation:

- Used in AND (&&) and OR (||) operations.
- Avoids unnecessary computation once the result is known.

Purpose:

- Efficient conditional evaluation.
- Simplifies flow control during code generation.

✅ Summary of Unit III

Topic	Key Concept	Importance
SDD	Connects syntax and semantics	Foundation of compiler translation
Syntax Tree	Represents expression hierarchy	Used in code generation
S-Attributed	Synthesized-only attributes	Suits bottom-up parsing
L-Attributed	Inherited + synthesized	Supports top-down parsing
Postfix Notation	Operator after operands	Easy to evaluate
DAG	Removes redundancy	Used in optimization
TAC	Three operand code	Machine independent
Triples/Quadruples	TAC representation	Simplifies storage
Boolean Expressions	Logical evaluation	Control flow generation