.

**📘 COMPILER CONSTRUCTION — Mid 1 Topics Breakdown**

**1️⃣ Introduction to Compiler**

**🔹 What is a Compiler?**

A **compiler** is a program that translates high-level language (C, Java) into **machine code**.

**🔹 Compiler vs Interpreter**

|  |  |  |
| --- | --- | --- |
| **Feature** | **Compiler** | **Interpreter** |
| Translation | Translates entire code at once | Translates line by line |
| Output | Produces object code | No object code; executes directly |
| Speed | Faster after compilation | Slower due to real-time translation |
| Example | C, C++ | Python, JavaScript |

**2️⃣ Typical Language Processing System**

**Includes:**

1. **Preprocessor** – Handles macros, includes
2. **Compiler** – Analyzes and translates code
3. **Assembler** – Converts intermediate code to machine instructions
4. **Linker/Loader** – Links object files, loads into memory

**3️⃣ Compiler Design Phases**

1. **Lexical Analysis**
2. **Syntax Analysis**
3. **Semantic Analysis**
4. **Intermediate Code Generation**
5. **Code Optimization**
6. **Target Code Generation**

**Other Components:**

* **Symbol Table Manager** – Tracks variable/function declarations
* **Error Handler** – Detects and reports errors

**4️⃣ Classification of Compilers**

* **Single-Pass vs Multi-Pass**
* **Just-In-Time (JIT) Compiler**
* **Cross Compiler**
* **Incremental Compiler**
* **Optimizing Compiler**

**5️⃣ Lexical Analyzer**

**🔹 Introduction**

Breaks the input source code into **tokens** (identifiers, numbers, operators).

**🔹 Why Separate Lexical & Syntax Analysis?**

* Simpler parsing logic
* Reusable lexer for multiple languages

**🔹 Secondary Tasks**

* Remove whitespace/comments
* Track line/column for errors
* Communicate with symbol table

**🔹 Tokens, Patterns, Lexemes**

|  |  |
| --- | --- |
| **Term** | **Description** |
| Token | Unit of meaning (e.g., if, id) |
| Lexeme | Actual character sequence (while) |
| Pattern | Rule describing a token (e.g., [a-zA-Z]+) |

**🔹 Implementation Strategies**

* **Table-driven**
* **Ad-hoc (hard-coded if-else)**
* **Using Lex (lexer generator)**

**🔹 Input Buffering**

Improves performance with **two-buffer scheme**, especially with lookahead.

**🔹 Specification of Tokens**

Usually defined with **regular expressions**, e.g.,

id → letter (letter|digit)\*

**🔹 Recognition of Tokens**

Done via **Finite State Machines (FSM)** — either deterministic (DFA) or non-deterministic (NFA)

**🔹 Lex Tool**

* Generates lexical analyzers
* Input: rules + actions
* Output: lex.yy.c (C code for lexer)

**6️⃣ Syntax Analyzer**

**🔹 Introduction**

Checks **grammatical structure** based on CFG (Context-Free Grammar).

**🔹 Chomsky Hierarchy**

|  |  |
| --- | --- |
| **Type** | **Description** |
| Type 0 | Unrestricted |
| Type 1 | Context-sensitive |
| Type 2 | Context-free (used in parsing) |
| Type 3 | Regular grammar (used in lexing) |

**🔹 Grammar Representations**

* **Backus-Naur Form (BNF)**
* **Parse Trees / Derivations**

**🔹 Context-Free Grammar (CFG)**

Example:

E → E + T | T

T → T \* F | F

F → (E) | id

**🔹 Left Recursion**

Problem: causes infinite recursion in top-down parsers  
Example:

E → E + T | T

Fix (eliminate recursion):

E → T E'

E' → + T E' | ε

**🔹 Left Factoring**

Resolves **common prefixes** for predictive parsing  
Example:

A → if E then S else S | if E then S

Fix:

A → if E then S A'

A' → else S | ε

**🔹 Ambiguous Grammar**

A grammar is ambiguous if it has **more than one parse tree** for the same input.

**🔹 Removing Ambiguity**

Use:

* Operator precedence rules
* Associativity Rules
* Explicit grammar rewriting

**🔹 Simplification of Grammars**

* Eliminate:
  + **Unreachable symbols**
  + **Useless productions**
  + **Epsilon/Unit productions**

**🔹 Applications of CFG**

* Compilers
* Programming language design
* Natural language processing (NLP)
* XML/JSON schema validation

**7️⃣ Top-Down Parser**

**🔹 Introduction**

Parses input from **start symbol to leaves** using **leftmost derivation**.

**🔹 Types of Top-Down Parsers:**

* **Recursive Descent Parser**
* **Predictive Parser (LL)**

**🔹 Recursive Descent Parser**

* Each non-terminal has a recursive function
* Can be **backtracking** or **non-backtracking**

**🔹 Predictive Parser**

* Uses **lookahead** to decide next rule
* Requires **LL(1) Grammar**

**🔹 LL(1) Grammar**

* L: Left-to-right input scan
* L: Leftmost derivation
* 1: One-token lookahead

LL(1) requirements:

* No left recursion
* Left factoring applied
* Disjoint FIRST and FOLLOW sets for any non-terminal

**🔹 Predictive Parsing Table Construction**

**Steps:**

1. Compute FIRST and FOLLOW sets
2. Fill table:
   * If A → α and a ∈ FIRST(α), then M[A,a] = A → α
   * If ε ∈ FIRST(α) and b ∈ FOLLOW(A), then M[A,b] = A → ε

Parser uses table to drive decisions during parsing.

Chapter Slides

📘 CS4031 — Compiler Construction

🧠 Lecture 1: Introduction to Compiler

**1️⃣ What is a Compiler?**

**🔹 Definition:**

A **compiler** is a program that translates a source code written in a **high-level language** (e.g., C, C++) into a **target language** (typically low-level machine code).

**🔹 Purpose:**

* Translate source code to machine code
* Check for errors (semantic, type)
* Optimize performance
* Enable portability

**2️⃣ Why High-Level Languages?**

|  |  |
| --- | --- |
| Reason | Explanation |
| Closer to human thinking | Easier to express logic |
| Error detection | Compiler spots common mistakes |
| Shorter code | Less effort and redundancy |
| Portability | Compile once, run on multiple systems |

**3️⃣ Interpreter vs Compiler**

|  |  |  |
| --- | --- | --- |
| Feature | Compiler | Interpreter |
| Input | Entire program | Line-by-line |
| Speed | Faster after compilation | Slower due to real-time translation |
| Output | Object code generated | No object code |
| Memory | More (stores object code) | Less |
| Errors | Shown after full parse | Shown instantly line-by-line |
| Debugging | Harder | Easier |
| Examples | C, C++ | Python, Ruby |

**4️⃣ Skills Needed to Build a Compiler**

* **Programming Concepts**: memory, scope, parameter passing
* **Automata Theory**: FSM, regular expressions
* **Context-Free Grammars**: syntax rules
* **Algorithms/Data Structures**: graphs, hashing, dynamic programming
* **Computer Architecture**: registers, assembly code
* **Software Engineering**: design and modularity

**5️⃣ Analysis-Synthesis Model of Compilation**

**🔹 Analysis Phase:**

* Break source code into structure
* Output: **intermediate representation (IR)**

**🔹 Synthesis Phase:**

* Converts IR into target machine code

**6️⃣ Why Study Compilers?**

* Core concept in **Computer Science**
* Helps understand **programming tools**
* Useful in building **DSLs (domain-specific languages)**
* Compiler techniques apply in:
  + **Interpreters**
  + **Static analyzers**
  + **Code generators**

**7️⃣ Preprocessors, Compilers, Assemblers, Linkers**

**📌 Preprocessor:**

* Handles #define, #include
* Removes macros, replaces code
* Output: pure C code

**📌 Compiler:**

* Translates code to **assembly or object code**

**📌 Assembler:**

* Converts assembly code to **machine code**

**📌 Linker:**

* Combines object files & libraries
* Produces final **executable**

**8️⃣ Compiler Flags to View Each Stage**

|  |  |
| --- | --- |
| Flag | Purpose |
| -E | Stop after preprocessing |
| -S | Stop after compilation (keep .s) |
| -c | Stop after assembling (keep .o) |

**9️⃣ Phases of Compilation**

1. **Lexical Analysis** → Tokens
2. **Syntax Analysis** → Syntax tree
3. **Semantic Analysis** → Type and scope checking
4. **Intermediate Code Generation** → IR like Three Address Code
5. **Code Optimization** → Speed/space efficiency
6. **Target Code Generation** → Machine code

**🔟 Lexical Analysis**

**🔹 Role:**

* Tokenizes the input
* Removes whitespace/comments

**🔹 Example:**

Code: sum = 2 + c;  
Tokens:

css

CopyEdit

[sum] → Identifier

[=] → Assignment operator

[2] → Constant

[+] → Operator

[c] → Identifier

**1️⃣1️⃣ Syntax Analysis**

**🔹 Also Known As:**

**Parsing**

**🔹 Output:**

**Abstract Syntax Tree (AST)**

**🔹 Example:**

Code: sum = 2 \* c  
AST:

=

/ \

sum \*

/ \

2 c

**1️⃣2️⃣ Semantic Analysis**

* Checks **type correctness**
* Verifies variables and scope
* Annotates tree with semantic info

**1️⃣3️⃣ Intermediate Code Generation**

**🔹 Purpose:**

Create a low-level, platform-independent code.

**🔹 Example:**

Input: sum = 2 \* c

IR:

t1 = 2 \* c

sum = t1

**1️⃣4️⃣ Code Optimization**

* Removes redundant code
* Rearranges for efficiency

Example:

int x = 5;

int y = 5;

int z = x + y;

→ optimize to: z = 10;

**1️⃣5️⃣ Target Code Generation**

**🔹 Output:**

Actual **machine instructions**

**🔹 Example:**

MOV R1, 2

MOV R2, c

MUL R1, R2

MOV sum, R1

**1️⃣6️⃣ Code Generation Example**

**Code:**

Price = old + Rate \* 50;

**Intermediate Code:**

t1 = Rate \* 50

Price = old + t1

**1️⃣7️⃣ Target Code Generation Example**

MOV old, R1

MOV Rate, R2

MOV 50, R3

MUL R2, R3

ADD R1, R2

MOV R1, Price

**1️⃣8️⃣ Compiler Types**

**📌 Single-Pass Compiler**

* All phases done in **one pass**
* Fast but limited
* Example: Pascal

**📌 Two-Pass Compiler**

* **Front-end**: Lexing, Parsing, Semantic
* **Back-end**: Optimization, Code generation

**📌 Multi-Pass Compiler**

* Runs source code through **multiple stages**
* Each stage outputs to file for the next
* Better for optimization and modularity

**✅ MCQs (5 Quick Review Questions)**

1. **Which phase converts source code into tokens?**  
   ✅ A) Lexical Analysis
2. **Which component links object code with libraries?**  
   ✅ B) Linker
3. **What is the output of syntax analysis?**  
   ✅ C) Abstract Syntax Tree
4. **Which compiler type processes code in a single pass?**  
   ✅ A) Single-Pass Compiler
5. **What is the role of a preprocessor?**  
   ✅ B) Expand macros and includes

📘 Lecture 2: Lexical Analysis (CS4031 – Compiler Construction)

**🔹 1. Lexical Analyzer: Role and Definition**

**💡 What is Lexical Analysis?**

Lexical analysis is the **first phase of a compiler**. It reads characters from the source program, groups them into **lexemes**, and produces a sequence of **tokens** as output.

* **Lexeme**: The actual string of characters from source code.
* **Token**: The categorized output of the lexeme. Each token has a name and an optional attribute.

**🧠 Why Is It Needed?**

* Removes whitespace and comments.
* Converts source code into meaningful symbols for the next phase (syntax analysis).

**🔹 2. Lexeme, Tokens, Patterns**

**✅ Lexeme**

A sequence in the source code that matches a pattern for a token.

📌 Example:

int c = 5;

|  |  |
| --- | --- |
| Lexeme | Token |
| int | Keyword |
| c | Identifier |
| = | Assignment Operator |
| 5 | Constant |
| ; | Symbol |

**✅ Token**

* A pair: **(token-name, optional-attribute-value)**
* Types: Identifier, Keyword, Constant, Operator, Separator, String

**✅ Pattern**

* Describes the form of a lexeme. Example: [a-zA-Z\_][a-zA-Z0-9\_]\* matches identifiers.

**🔹 3. Interaction with Parser and Symbol Table**

**Symbol Table**

Stores metadata about identifiers:

Name | Type | Size | Line No. | Scope

------------------------------------------

semester | int | 4 | 0 | local

**🔹 4. Ad-hoc Lexer**

A manual method (not using tools like Lex) to tokenize source code.

**💻 Example in C++:**

class Scanner {

InputStream s;

char next;

Lexer(InputStream \_s) {

s = \_s;

next = s.read(); // lookahead character

}

Token nextToken() {

if (isIdentifier(next)) return readId();

if (isNumber(next)) return readNumber();

...

}

}

**🔹 5. Token Generation via Program**

**💡 Function to read identifiers:**

Token readId() {

string id = "";

while (true) {

char c = input.read();

if (!isIdChar(c)) return new Token("TID", id);

id += c;

}

}

**🔹 6. C++ Example for Identifiers & Keywords**

**📌 Keyword Function:**

int isKeyword(char buffer[]) {

char keywords[32][10] = {"auto", "break", "case", ...};

for (int i = 0; i < 32; ++i)

if (strcmp(keywords[i], buffer) == 0)

return 1;

return 0;

}

**📌 Operator Check:**

char operators[] = "+-\*/%=";

for (int i = 0; i < 6; ++i) {

if (ch == operators[i])

cout << ch << " is operator\n";

}

**🔹 7. Regular Expressions (RE) for Tokens**

**💡 RE Syntax:**

|  |  |
| --- | --- |
| Notation | Meaning |
| R|S | Either R or S |
| RS | R followed by S |
| R\* | Zero or more repetitions of R |
| R+ | One or more repetitions |
| R? | Zero or one occurrence |

**📌 RE Examples:**

* **Integer**: [0-9]+
* **Identifier**: [a-zA-Z\_][a-zA-Z0-9\_]\*

**🔹 8. Acceptors**

A mechanism that checks if string w ∈ L (language of RE).

input → acceptor → yes/no

**🔹 9. Finite Automata (FA)**

**🤖 What is FA?**

An abstract machine used to recognize patterns (used in lexical analyzers).

**Components:**

* **Input Alphabet (Σ)**
* **Set of States (Q)**
* **Start State (q0)**
* **Transition Function (δ)**
* **Set of Accepting States (F)**

**📘 Example:**

Accept any number of 1’s followed by a single 0:

States: q0 (start), q1 (accept)

Transition:

q0 --1--> q0

q0 --0--> q1

**🧠 MCQs (15)**

**1. Which of the following best defines a token?**

**A)** A string from source  
**B)** A regular expression  
**C)** A classified output from a lexeme ✅  
**D)** None

**2. What does the lexical analyzer remove?**

**A)** Errors  
**B)** Comments and whitespaces ✅  
**C)** Functions  
**D)** Keywords

**3. A lexeme for printf is classified as?**

**A)** Literal  
**B)** Keyword  
**C)** Identifier ✅  
**D)** Operator

**4. What is a pattern?**

**A)** Sequence of machine code  
**B)** Grammar rule  
**C)** Description of form of lexeme ✅  
**D)** None

**5. A DFA:**

**A)** Accepts context-free languages  
**B)** Accepts regular languages ✅  
**C)** Generates machine code  
**D)** Optimizes IR

**📌 Summary of Lecture 2**

* Lexical analysis is the first compiler phase, converting code into tokens.
* Tokens include identifiers, keywords, operators, constants, and more.
* Lexemes are actual text segments, matched by patterns (described via regular expressions).
* Ad-hoc lexers manually parse tokens; Lex tool can automate this.
* Finite Automata (FA) is used to implement token recognition via state transitions.
* Regular expressions are used to define patterns accepted by tokens.

**CS4031 Compiler Construction - Lecture 4 Full Notes**

**Practice: Define the Lex Program for Given Code Lexemes**

**Example 1:**

int main ( ) {

int x, a=2, b=3, c=5;

x = a + b \* c;

printf("the value of x is %d", x);

return 0;

}

This code consists of identifiers, operators, delimiters, keywords, numbers, and strings which need to be captured by tokens in a Lex specification.

**Example 2:**

int main ( ) {

a = b + d;

int a, b, d;

printf("sum of given is %d"; a);

}

Here, misplaced semicolon and misordering of declarations make token recognition important.

**Example 3:**

int main ( ) {

a = b++++++===--;

}

This complex expression demonstrates the need for proper handling of compound operators in lexical analysis.

**Syntax Analysis**

* Parses the syntactic structure of code based on CFG.
* Output: Parse tree
* A syntax error results in error messages.

**Syntax Tree**

* A tree representing syntactic structure using grammar rules.
* For example: id1 = id2 + id3 \* 60; turns into an expression tree.

**Front-End Parser Pipeline**

* Source Code → Scanner → Tokens → Parser → IR → Errors
* Ensures grammatical correctness and forms an intermediate representation.

**Natural Language Analogy**

* Sentence: "He wrote the program"
* Parts of speech → noun, verb, etc.
* Syntax Analysis aligns similarly in compilers.

**Syntax Analysis in Programming Language**

**Example:** if (b <= 0) a = b;

* Translates into if-statement, expression, and assignment.

**Faulty Code Example:**

Int \*compiler(int i, int j)) {

for(k = 0; i j;)

fi (i > j)

return j;

}

**Errors:**

* Extra Parenthesis
* Missing Expression
* Typo in fi instead of if

**Semantic Analysis**

**Code Example:**

int\* compiler(int i, int j ) {

for(k=0; i<j; j++)

if(i < j - 2)

float sum;

sum = sum + i;

return sum;

}

* Semantic issues: Improper declaration and use of variable sum.

**Role of Parser**

* Filters invalid token sequences.
* Constructs parse trees.

**Parsing**

* Matching string derivation using Grammar G
* Based on derivation rules (e.g., L(G))

**Grammar Definition**

G = (V, T, P, S)

* V: Non-terminals
* T: Terminals
* P: Productions
* S: Start Symbol

**Terminal Symbols:** lowercase letters, operators, punctuation, digits, keywords

**Non-Terminal Symbols:** Uppercase letters (A-Z), start symbol S, non-terminal strings like expr, stmt

**Chomsky Hierarchy**

1. **Type 0 (Unrestricted)**: General grammar
2. **Type 1 (Context-Sensitive)**: Length of RHS >= LHS
3. **Type 2 (Context-Free)**: LHS = Single Non-terminal
4. **Type 3 (Regular)**: RHS = Terminal or Terminal+Non-terminal

**BNF (Backus-Naur Form)**

* Used to formally describe CFGs
* Format: <lhs> ::= <rhs>
* Example:

<digit> ::= 0 | 1 | 2 | ... | 9

**Grammar Derivation Example:**

<expression> ::= <term> | <expression> - <term>

<term> ::= <factor> | <term> \* <factor>

<factor> ::= x | y | 2

**Leftmost Derivation:** Prioritize left child **Rightmost Derivation:** Prioritize right child

**Grammar Problems**

* **Left Factoring**: Needed when common prefixes exist
* **Left Recursion**: Problematic for Top-down parsers

**Elimination of Left Recursion:** From: A → Aα | β To: A → βA', A' → αA' | ε

**Ambiguity Removal**

* **Associativity Constraint**:
  + +, \* → Left associative
  + ^ → Right associative
* **Precedence Constraint**:
  + Lower production level = higher precedence

**First & Follow Sets**

* **First(A)**: terminals that start the strings derivable from A
* **Follow(A)**: terminals that can appear immediately after A

**Rules:**

1. For S (start symbol): Add $ to Follow(S)
2. A → αB: Follow(B) += Follow(A)
3. A → αBβ:
   * If ε ∉ First(β), Follow(B) += First(β)
   * Else: Follow(B) += (First(β) \ ε) ∪ Follow(A)

**Example:**

E → E+T | T

T → T\*F | F

F → id | (E)

**First & Follow Table Example**

|  |  |  |
| --- | --- | --- |
| **Production** | **FIRST** | **FOLLOW** |
| E → TE' | {id, (} | { $, ) } |
| E'→+TE'/ε | {+, ε} | { $, ) } |
| T → FT' | {id, (} | { +, $, ) } |
| T'→\*FT'/ε | {\*, ε} | { +, $, ) } |
| F → id/(E) | {id, (} | { \*, +, $, ) } |

**Example Grammar:**

S → ACB | Cbb | Ba

A → da | BC

B → g | ε

C → h | ε

**FIRST:**

* FIRST(S) = { d, g, h, ε, b, a }
* FIRST(A) = { d, g, h, ε }
* FIRST(B) = { g, ε }
* FIRST(C) = { h, ε }

**FOLLOW:**

* FOLLOW(S) = { $ }
* FOLLOW(A) = { h, g, $ }
* FOLLOW(B) = { a, $, h, g }
* FOLLOW(C) = { b, g, $, h }

**Parsing Techniques**

**Top-Down Parsers:**

* Recursive Descent
* LL(1), LL(0)

**Bottom-Up Parsers:**

* LR Parser
* Operator Precedence
* SLR(1)
* LALR(1)
* CLR(1)

**CS4031 - Compiler Construction**

**Lecture 5: LL(1) Parsing & Ambiguity Removal**

**Lecturer: Mahzaib Younas, FAST NUCES CFD**

**Ambiguity in Grammars**

Ambiguity arises when a grammar produces more than one parse tree for a single input string. Common sources:

1. Logical Operators
2. Arithmetic Operators
3. If-else statements
4. Regular Expression Operators

**Key Concepts**

* **Precedence:** Order of operation
* **Associativity:** Direction of grouping (left or right)

**Arithmetic Operator Precedence & Associativity**

|  |  |  |
| --- | --- | --- |
| Operator | Precedence | Associativity |
| id | 1 | - |
| ^ | 2 | Right-to-left |
| \*, / | 3 | Left-to-right |
| +, - | 4 | Left-to-right |

**Logical Operator Precedence & Associativity**

|  |  |  |
| --- | --- | --- |
| Operator | Precedence | Associativity |
| id | 1 | - |
| ~ | 2 | Right-to-left |
| ^ | 3 | Left-to-right |
| ∨ | 4 | Left-to-right |

**Unambiguous Grammar Example for Logical Ops**

Ambiguous: E -> E ^ E

Unambiguous:

E -> E ∨ E1 | E1

E1 -> E1 ^ E2 | E2

E2 -> ~E2 | E3

E3 -> id | (E)

**If-Else Statement Problem**

Ambiguous Grammar:

S -> if E then S | if E then S else S | other

**Resolved Using:**

* **Matched Statements:** Non-if or fully formed if-then-else
* **Open Statements:** Incomplete if-then or dangling else

Example:

if E1 then if E2 then S1 else S2

Two possible parse trees exist -> Grammar is ambiguous.

**LL(1) Parsing**

**LL(1):**

* First 'L': Left-to-right scan of input
* Second 'L': Leftmost derivation
* '1': One lookahead symbol

**Conditions for LL(1):**

1. No left recursion
2. Left factoring
3. Disjoint FIRST and FOLLOW sets

**Sample Grammar**

E -> TE'

E' -> +TE' | ε

T -> FT'

T' -> \*FT' | ε

F -> id | (E)

**FIRST and FOLLOW**

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

**LL(1) Parse Table**

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

**Stack-Based LL(1) Parsing Example**

Input: id + id \* id $

**Stack Trace**

|  |  |  |
| --- | --- | --- |
| Stack | Input | Action |
| E$ | id+id\*id$ | E -> TE' |
| TE'$ | id+id\*id$ | T -> FT' |
| FT'E'$ | id+id\*id$ | F -> id |
| idT'E'$ | id+id\*id$ | Match id |
| T'E'$ | +id\*id$ | T' -> ε |
| E'$ | +id\*id$ | E' -> +TE' |
| +TE'$ | +id\*id$ | Match + |
| TE'$ | id\*id$ | T -> FT' |
| FT'E'$ | id\*id$ | F -> id |
| idT'E'$ | id\*id$ | Match id |
| T'E'$ | \*id$ | T' -> \*FT' |
| \*FT'E'$ | \*id$ | Match \* |
| FT'E'$ | id$ | F -> id |
| idT'E'$ | id$ | Match id |
| T'E'$ | $ | T' -> ε |
| E'$ | $ | E' -> ε |
| $ | $ | ACCEPT |

**Extra Examples for Practice**

**Example 2: Ambiguous Grammar**

S -> iEtSS' | a

S' -> eS | ε

E -> b

Not LL(1) because S and S' have overlapping FIRST and FOLLOW sets.

**Example 3: Simple Nullable Grammar**

S -> aABb

A -> c | ε

B -> d | ε

Satisfies LL(1), construct parse table with ε in A and B.

**Example 4: Nullable Right Recursion**

S -> W

W -> ZXY

Y -> c | ε

Z -> a | d

X -> Xb | ε

Use FIRST & FOLLOW to construct parse table.

**Summary**

* LL(1) grammars are simple to implement and fast to parse.
* Removing ambiguity, left recursion, and applying left factoring are essential preprocessing steps.
* Parse tables help in efficient parser creation.
* LL(1) parsing is limited but widely used due to predictability and ease of debugging.