MODULE-II

SYNTAX ANALYSIS

2.1 ThE ROLE OF THE PARSER

The **parser** is a critical component of a compiler's **syntax analysis** phase. After the lexical analyzer (lexer) breaks the input source code into tokens, the parser takes these tokens and determines their syntactic structure according to the grammar of the programming language. In other words, the parser checks if the sequence of tokens forms valid expressions and statements, and builds an internal representation of the program, typically a **parse tree** or **abstract syntax tree (AST)**.

Key Responsibilities of the Parser:

1. **Syntax Analysis**:
    - The parser ensures that the sequence of tokens provided by the lexical analyzer conforms to the grammatical rules of the source language.
    - The **grammar** defines the valid structure of statements and expressions in the language. The parser checks the tokens against this grammar and verifies whether they can be combined in valid ways.
2. **Building the Abstract Syntax Tree (AST)**:
    - Once the parser verifies the syntactic correctness of the input, it constructs a data structure representing the hierarchical structure of the program, called the **abstract syntax tree (AST)**.
    - The AST is a simplified, tree-like representation of the program that captures its logical structure without worrying about unnecessary syntactic details (like parentheses or commas).
    - For example, the expression 3 + 5 * 2 would be represented in the AST as a tree where * is a node with two children (5 and 2), and + is a parent of the * node and 3.
3. **Error Detection**:
    - The parser is also responsible for detecting **syntax errors**. If the sequence of tokens does not adhere to the grammar, the parser throws an error, providing useful feedback about where and what went wrong in the source code.
4. **Syntax Tree Construction**:
    - The parser typically constructs a **parse tree** or **abstract syntax tree (AST)**. The **parse tree** is a complete representation of the grammatical structure of the input, whereas the **AST** is a more abstract, simplified version that omits unnecessary details.

Types of Parsers:

There are two main types of parsers used in compilers:

1. **Top-Down Parsers**:
    - These parsers begin at the root of the parse tree and work down to the leaves, trying to match the grammar rules against the input tokens.

- **Recursive Descent Parser**: This is a common type of top-down parser that uses a set of recursive procedures to process each non-terminal in the grammar. It requires the grammar to be in **LL(1)** form (i.e., no ambiguity and no left recursion).
- **Predictive Parsing**: This is a more efficient version of recursive descent parsing, which uses a lookahead symbol to make decisions about which rule to apply.

2. **Bottom-Up Parsers**:
   - Bottom-up parsers start at the leaves (tokens) and work their way up to the root of the parse tree. These parsers reduce the input tokens to non-terminals according to the production rules of the grammar.
   - **LR Parsers**: These are a class of bottom-up parsers that can handle a wide range of grammars. They are more powerful than top-down parsers and can process **LR(1)** grammars (i.e., grammars that can be parsed with a single lookahead token).
   - **Shift-Reduce Parsing**: A common technique used by LR parsers. It involves shifting tokens onto a stack and then reducing them to higher-level constructs as the parsing process progresses.

Parsing Techniques:

1. **LL Parsing (Top-Down)**:
   - An LL parser reads input from **left to right** and applies **leftmost derivation**.
   - LL parsers are usually simpler but have restrictions, such as the need for **predictive parsing** and the inability to handle left-recursive grammars without transformation.
2. **LR Parsing (Bottom-Up)**:
   - An LR parser reads input from **left to right** and constructs a **rightmost derivation** in reverse.
   - LR parsers are more powerful than LL parsers and can handle a broader class of grammars, including most programming language grammars.
   - They are typically implemented using **shift-reduce algorithms**, where the input is shifted onto a stack, and reductions are performed when applicable grammar rules are matched.

Parse Tree vs Abstract Syntax Tree (AST):

- **Parse Tree**:
  - The parse tree (also known as a **concrete syntax tree**) represents all the syntactic details of the source code, adhering to the grammar of the language.
  - Every grammar rule is explicitly represented in the parse tree.
  - A parse tree is more detailed and is used primarily during parsing to validate the input.
- **Abstract Syntax Tree (AST)**:
  - The AST is a more abstract, simplified version of the parse tree that omits extraneous syntactic details (like parentheses, commas, etc.) that are not necessary for the program's meaning.

- The AST focuses on the logical structure of the program, making it easier for subsequent phases (such as semantic analysis and code generation) to operate on the program's logic.
- For example, the expression 3 + 5 * 2 would be represented in the AST with + as the root, 3 as a left child, and * as the right child with 5 and 2 as its children.

Example of Parsing:

Consider the following simple arithmetic expression:

3 + 5 * 2

Parse Tree: The parse tree for this expression (according to typical arithmetic grammar) would look like this:

```
  +
 / \
3   *
   / \
  5   2
```

Abstract Syntax Tree (AST): The AST for this expression, omitting unnecessary syntactic elements, would look like this:

```
  +
 / \
3   *
   / \
  5   2
```

In this case, the AST might look similar to the parse tree because the grammar for basic arithmetic is simple. However, in more complex expressions, the AST would omit details like parentheses or intermediate nodes.

Error Detection in the Parser:

- **Syntax Errors**: The parser can detect errors in the source code based on the grammar. If a sequence of tokens does not match any valid rule, the parser will throw a syntax error, typically indicating the position where the error occurred.

For example, an expression like:

3 + * 5

would be flagged by the parser as an error because the * operator is misplaced.

- **Recovery**: Parsers often have strategies for recovering from errors to continue parsing the rest of the input. For example, an error recovery strategy might skip invalid tokens or try to synchronize with the next valid construct.

The Role of the Parser in Compiler Phases:

1. **Input**: The parser receives tokens from the lexical analyzer.

2. **Processing**: It checks if the sequence of tokens follows the syntax rules of the language and constructs a parse tree or AST.
3. **Output**: The parser outputs the **abstract syntax tree (AST)** or **parse tree** that will be passed to the next phase of the compiler, such as semantic analysis or intermediate code generation.

2.2 ELIMINATING AMBIGUITY

Ambiguity in the context of programming languages and compiler design refers to situations where a grammar can generate more than one valid parse tree for a given input string. Ambiguity can lead to confusion and problems in both language parsing and interpretation, as it becomes unclear which structure or meaning the string is supposed to represent.

In a compiler, **eliminating ambiguity** is a critical task, because an ambiguous grammar can make it impossible to consistently and correctly parse programs. Ambiguity in a grammar can lead to multiple interpretations of a given piece of source code, making the behavior of the program unpredictable.

Causes of Ambiguity

Ambiguity arises when the grammar allows multiple interpretations of the same string. For example, in arithmetic expressions, the expression 3 + 4 * 5 could be interpreted in two ways:

1. **(3 + 4) * 5** (First add, then multiply)
2. **3 + (4 * 5)** (First multiply, then add)

The ambiguity comes from the lack of clear precedence rules between addition (+) and multiplication (*). If the grammar allows both interpretations, then it is ambiguous.

Example of an Ambiguous Grammar

Consider the following simple grammar for arithmetic expressions:

$E \rightarrow E + E$

$E \rightarrow E * E$

$E \rightarrow (E)$

$E \rightarrow id$

This grammar is ambiguous because there is no specification for the precedence of operators like + and *. For example, the string id + id * id could be parsed in two ways:

- **(id + id) * id** (where addition happens first)
- **id + (id * id)** (where multiplication happens first)

Both interpretations are valid, but they give different results for the same expression.

Methods to Eliminate Ambiguity

There are several techniques to eliminate ambiguity from a grammar:

1. Rewriting the Grammar (Refactor the Grammar)

A common approach to eliminating ambiguity is to **rewrite the grammar** to clarify precedence rules and avoid ambiguous constructs.

For example, the ambiguous grammar for arithmetic expressions can be rewritten to make operator precedence explicit:

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

In this grammar:

- **E** stands for an expression.
- **T** stands for a term (which handles multiplication).
- **F** stands for a factor (which could be an identifier or a subexpression).

This modified grammar clearly separates the addition (+) and multiplication (*) operators, assigning higher precedence to multiplication (T), so that the string id + id * id is interpreted as **id + (id * id)**, rather than (id + id) * id.

2. Operator Precedence and Associativity Rules

Operator precedence and associativity rules can be explicitly incorporated into the grammar to ensure that expressions are interpreted in the intended way. For example:

- **Precedence**: Multiplication has higher precedence than addition.
- **Associativity**: If two operators of the same precedence appear in an expression, the associativity determines how they should be grouped. For most operators like + and *, the associativity is **left-to-right** (left associative).

This can be done by creating separate non-terminals for different levels of precedence, as shown in the previous example.

3. Use of Semantic Actions

While the syntax of a language can be ambiguous, semantic rules can often help resolve ambiguities. This is done by defining the **meaning** of constructs that can be interpreted in different ways, based on the context. For example:

- **Context-sensitive parsing**: The context of the expression could dictate whether + or * should be parsed first, without needing to change the grammar. This approach involves adding semantic checks during parsing (e.g., based on the types of operands).

4. Left Recursion Elimination

In some cases, ambiguity can be caused by **left recursion** in a grammar. Left recursion occurs when a non-terminal symbol in a production rule references itself as the first symbol on the right-hand side.

For example, the following grammar is left-recursive and potentially ambiguous:

$E \rightarrow E + T$

$E \rightarrow T$

This can lead to infinite recursion in parsers that don't handle left recursion. To eliminate ambiguity, left recursion can be removed by restructuring the grammar, as shown below:

$E \rightarrow T\ E'$

$E' \rightarrow + T\ E' \mid \varepsilon$

$T \rightarrow id \mid (E)$

Here, **E'** handles the recursive part, and the grammar is now **right-recursive**, which is more easily handled by parsers.

5. Use of Parsing Strategies (LL and LR)

Certain parsing techniques can handle ambiguity more effectively:

- **LL(1) Parsers**: These parsers look ahead one token and use a top-down approach. By enforcing **left-to-right** parsing and **leftmost derivation**, these parsers can avoid ambiguity through proper grammar design.
- **LR(1) Parsers**: These parsers use a bottom-up approach, which is capable of handling more complex grammars and can deal with **operator precedence** and **associativity** more effectively.

By adjusting the grammar and designing parsing strategies, ambiguity can often be controlled.

6. Disambiguation Using Priorities or Precedence Tables

For more complex cases of ambiguity (e.g., operator precedence), **disambiguation rules** such as **precedence tables** or **priority values** can be applied. These tables or rules specify which production should be chosen when multiple alternatives are available for the same input.

For example, a precedence table might specify that multiplication has higher precedence than addition, ensuring that expressions like 3 + 5 * 2 are interpreted as 3 + (5 * 2).

By applying these methods, a grammar can be made unambiguous, enabling the compiler to accurately parse and understand source code. Ambiguity elimination ensures that programs are parsed consistently and that their meaning is clear to the compiler.

2.3 ELIMINATING OFLEFT RECURSION AND LEFT FACTORING

In the context of compiler design, **left recursion** and **left factoring** are two important concepts that can cause issues when constructing parsers, particularly **recursive descent parsers**. These issues can lead to infinite recursion or inefficient parsing. To ensure the grammar can be parsed efficiently, it is important to eliminate left recursion and apply left factoring where necessary.

Let's go through each concept and how to eliminate them.

1. Eliminating Left Recursion

Left recursion occurs when a non-terminal symbol in a production rule refers to itself on the left side of the rule, potentially causing infinite recursion during parsing. Left recursion is problematic for **top-down parsers** (like recursive descent parsers), which process the input from left to right and expand the leftmost non-terminal first. Left recursion can lead to infinite loops when trying to expand such non-terminals.

Example of Left Recursion

Consider the following grammar for arithmetic expressions:

$E \rightarrow E + T$

$E \rightarrow T$

In this grammar, the production $E \rightarrow E + T$ is left-recursive because the non-terminal E appears at the beginning of the right-hand side. If a parser starts expanding E by applying the rule $E \rightarrow E + T$, it will continue expanding E indefinitely.

Elimination of Left Recursion

To eliminate left recursion, we can rewrite the grammar to make it **right-recursive**. The basic idea is to introduce a new non-terminal that handles the recursive part in a rightward fashion. Here's how we can transform the left-recursive grammar:

Original Grammar (with left recursion):

$E \rightarrow E + T$

$E \rightarrow T$

Rewritten Grammar (without left recursion):

$E \rightarrow T\ E'$

$E' \rightarrow + T\ E' \mid \varepsilon$

In this transformed grammar:

- E now produces T followed by E'.
- E' handles the recursion, producing + T E' (for recursive cases) or ε (for the base case).
- This is a right-recursive structure that no longer leads to infinite recursion.

General Approach for Elimination of Left Recursion

To eliminate left recursion in a grammar rule of the form:

$A \rightarrow A\ \alpha \mid \beta$

Where:

- A is a non-terminal.
- α and β are sequences of grammar symbols (with α starting with A, causing left recursion).

We follow these steps:

1. Introduce a new non-terminal A'.
2. Rewrite the grammar as:

$A \rightarrow \beta\ A'$

$A' \rightarrow \alpha\ A' \mid \varepsilon$

This effectively moves the recursive part to the right-hand side, preventing the infinite recursion problem.

2. Left Factoring

Left factoring is a technique used to eliminate ambiguity in a grammar that can lead to inefficiency in predictive parsing (such as in **recursive descent parsing**). It occurs when there are two or more alternatives that begin with the same prefix, making it difficult for the parser to decide which rule to apply based on the first token.

Example of Left Factoring

Consider the following grammar for arithmetic expressions:

$E \rightarrow T + E$

$E \rightarrow T - E$

$E \rightarrow T$

Here, the non-terminal E has two alternatives (T + E and T - E) that both start with the same token T. This creates ambiguity because the parser will not know whether to apply T + E or T - E until it sees more tokens.

Elimination of Left Factoring

To eliminate left factoring, we introduce a new non-terminal that captures the common prefix (T in this case) and factors it out. Here's how we can transform the grammar:

Original Grammar (with left factoring):

$E \rightarrow T + E$

$E \rightarrow T - E$

$E \rightarrow T$

Rewritten Grammar (after left factoring):

$E \rightarrow T\ E'$

$E' \rightarrow + E \mid - E \mid \varepsilon$

In this transformed grammar:

- E first produces T, followed by E'.
- E' handles the remaining part, which can be + E, - E, or nothing (represented by ε).
- This makes the decision about whether to apply the + or - operator unambiguous, as it comes after the T.

General Approach for Left Factoring

To factor out a grammar rule of the form:

$A \rightarrow \alpha\ \beta \mid \alpha\ \gamma$

Where:

- A is a non-terminal.
- α is the common prefix in both alternatives.
- β and γ are the rest of the rules that follow the common prefix.

We rewrite the grammar as:

$A \rightarrow \alpha\ A'$

$A' \rightarrow \beta \mid \gamma$

This allows the parser to first match the common prefix α, and then decide between β or γ.

3. Practical Example

Let’s consider a more complex grammar and eliminate both left recursion and left factoring:

Original Grammar (with both left recursion and left factoring):

$E \rightarrow E + T \mid E - T \mid T$

$T \rightarrow id$

- The rule $E \rightarrow E + T \mid E - T$ is left-recursive (because E appears on the left side).
- The alternatives E + T and E - T start with the same token E, leading to a need for left factoring.

Step 1: Eliminate Left Recursion

First, eliminate the left recursion:

$E \rightarrow T\ E'$

$E' \rightarrow +\ T\ E' \mid -\ T\ E' \mid \varepsilon$

$T \rightarrow id$

Now, the grammar is no longer left-recursive, but we still need to address left factoring.

Step 2: Eliminate Left Factoring

In the rewritten grammar $E' \rightarrow +\ T\ E' \mid -\ T\ E'$, we have alternatives that start with the same token (+ and -). Left factoring can be done as follows:

$E \rightarrow T\ E'$

$E' \rightarrow (\ + \mid -\ )\ T\ E' \mid \varepsilon$

$T \rightarrow id$

Now, the grammar is **both** free from left recursion **and** left factoring issues, making it suitable for a top-down parser.

2.4 TOP-DOWN PARSING

Top-down parsing is a method of syntax analysis in which the parsing process starts from the **start symbol** (root) of the grammar and works its way down to the leaves (the terminals). It attempts to match the input string with the derivation of the grammar from the top to the bottom.

2.4.1 RECURSIVE DESCENT PARSING

Recursive descent parsing is one of the most straightforward and intuitive forms of top-down parsing. It involves a set of recursive functions that correspond to the non-terminal symbols in the grammar. The main idea is that each non-terminal has a recursive function that tries to match the corresponding production rules of the grammar.

Key Concepts of Recursive Descent Parsing

1. **Grammar**: Recursive descent parsing works with context-free grammars (CFGs), which are typically in **LL(1)** form. An LL(1) grammar means that it can be parsed with a left-to-right scan of the input and a **1-token lookahead**.
2. **Recursive Functions**: Each non-terminal in the grammar is represented by a function. The function tries to match the corresponding production rules.
3. **Backtracking**: Some recursive descent parsers can backtrack when a rule doesn't work (though this isn't always efficient). Backtracking is typically avoided in parsers for LL(1) grammars because such grammars don't require it.
4. **Lookahead**: The parser uses a lookahead token to decide which rule to apply. For LL(1) parsers, one lookahead token is enough to make a decision.

How Recursive Descent Parsing Works

Let’s consider a simple grammar for arithmetic expressions:

$E \rightarrow E + T \mid E - T \mid T$

$T \rightarrow id$

Step 1: Define Functions for Each Non-terminal

We define a recursive function for each non-terminal in the grammar. For example, for the non-terminal E, we have a function parse_E(), which corresponds to the production rules for E. Similarly, we have functions for T.

Step 2: Handle Each Rule in the Grammar

The recursive descent parsing function works by trying to match the input string with the grammar's rules. If a match is found, it recurses on the next part of the string; if no match is found, it backtracks or signals a parsing error.

For the grammar:

E → E + T | E - T | T

T → id

We define the following recursive functions:

1. **parse_E()**: This function will handle the rules for E.
2. **parse_T()**: This function will handle the rules for T.

Recursive Descent Parser for the Example Grammar

1. Function for E

We have the production rules:

E → E + T | E - T | T

This can be interpreted as:

- An expression E is either another expression E followed by + or - and a term T, or just a term T.

```
def parse_E():
    parse_T()  # Start by parsing the first T
    while lookahead == '+' or lookahead == '-':
        consume(lookahead)  # Consume the '+' or '-'
        parse_T()  # Parse the next term
```

2. Function for T

The rule for T is simpler:

T → id

This means a term is just an identifier (id), so we would define:

```
def parse_T():
    if lookahead == 'id':
        consume('id')  # Consume the identifier token
    else:
        error()  # If it's not an 'id', there's a syntax error
```

3. Token Consumption and Lookahead

The **lookahead** is the next token in the input string. The consume() function reads the current token and moves the pointer to the next token. The lookahead is updated to the next token after each consumption.

Example of How the Recursive Descent Parser Works

Consider the input id + id - id.

1. **parse_E()** is called, which first calls **parse_T()**.
2. **parse_T()** matches id and consumes it.
3. The lookahead token is now +, so the parser checks for the next operator in the parse_E() function.
4. The parser consumes + and calls **parse_T()** again to parse the next term (id).
5. This process repeats, and eventually, the parser consumes all tokens and successfully parses the entire input.

Advantages of Recursive Descent Parsing

1. **Simplicity**: Recursive descent parsing is simple and easy to implement. Each non-terminal in the grammar corresponds to a function, making the parser structure easy to understand and maintain.
2. **Direct Mapping to Grammar**: The parser closely follows the structure of the grammar. Each production rule is translated directly into a recursive function.
3. **Flexibility**: Recursive descent parsers can easily handle certain types of grammar (particularly LL(1) grammars), and they can be easily extended or modified.

Limitations of Recursive Descent Parsing

- **Left Recursion**: One major drawback of recursive descent parsers is that they cannot handle **left-recursive grammars**. Left recursion occurs when a non-terminal calls itself on the left side of its production (e.g., $E \rightarrow E + T$), which leads to infinite recursion.

Example:

$E \rightarrow E + T \mid T$

The parser would try to expand E indefinitely, leading to a stack overflow or infinite loop.

Solution: Left recursion can be eliminated by refactoring the grammar (as discussed previously in the context of eliminating left recursion).

- **LL(1) Restriction**: Recursive descent parsers are suitable for **LL(1)** grammars, which are grammars that can be parsed with a single lookahead token. They are not suitable for more complex grammars, such as **LR** grammars, which require more sophisticated parsing strategies (e.g., **LR parsers**).
- **Backtracking**: While some parsers support backtracking (attempting different parsing strategies if one fails), backtracking is inefficient and can lead to poor performance in larger grammars or more complex languages.

2.4.2 NON RECURSIVEPREDICTIVE PARSING

Non-recursive predictive parsing is a type of **top-down parsing** that constructs a parse tree for a given input string using a **predictive parser** without the need for recursion. This method is particularly useful for parsing grammars that are **LL(1)**, meaning they can be parsed using a single token lookahead, and it's

typically implemented using a **stack** and **table-based approach** rather than function calls like recursive descent parsing.

Key Concepts

1. **Predictive Parsing**: Predictive parsing is a form of **LL parsing** where the parser makes predictions about which production rule to apply based on the current input token and the current state. It uses a **lookahead** token (usually just one token) to decide which production to apply.
2. **Non-Recursive**: Unlike recursive descent parsing, where each non-terminal has a corresponding recursive function, non-recursive predictive parsing uses an explicit stack to track the parsing process, which avoids the risk of **stack overflow** or infinite recursion.
3. **LL(1) Grammar**: Non-recursive predictive parsing works on **LL(1) grammars**, which can be parsed with a single left-to-right scan of the input and a **one-token lookahead**. These grammars must be unambiguous, and every decision about which rule to apply can be made by inspecting the current input token.

How Non-Recursive Predictive Parsing Works

Non-recursive predictive parsing typically works using a **parse table** and a **stack** to simulate the recursive process in a non-recursive manner. The general approach involves using a **stack** to keep track of the current non-terminal being expanded and using a **parsing table** to determine which production rule to apply based on the current non-terminal and lookahead token.

Steps Involved:

1. **Initialize Stack**: The stack is initialized with the start symbol of the grammar.
2. **Start Parsing**:
    - The parser reads the input string from left to right.
    - The stack holds the current non-terminal that needs to be expanded, starting with the start symbol.
    - The **lookahead** is the current token from the input string.
3. **Predictive Decision**:
    - Based on the current non-terminal at the top of the stack and the lookahead token, the parser consults the **predictive parsing table** to find which production rule to apply.
    - If the top of the stack is a non-terminal, the parser uses the appropriate production rule from the table to expand it.
    - If the top of the stack is a terminal and matches the lookahead, the terminal is popped from the stack, and the input token is consumed (moved to the next token).
    - If there is no matching entry in the table, a **parsing error** is raised.
4. **Repeat** the process until the stack is empty, indicating the input string has been successfully parsed, or an error is encountered.

Example of Non-Recursive Predictive Parsing

Consider the following simple LL(1) grammar:

$E \rightarrow T\ E'$

E' → + T E' | ε

T → id

- **Start symbol**: E
- **Terminals**: id, +
- **Non-terminals**: E, E', T

We need to construct a **parsing table** to guide the non-recursive parsing process.

Parsing Table Construction

To create the parsing table, we need to compute the **FIRST** and **FOLLOW** sets for each non-terminal:

1. **FIRST sets**:
   - FIRST(E) = FIRST(T) = {id}
   - FIRST(E') = {+, ε}
   - FIRST(T) = {id}
2. **FOLLOW sets**:
   - FOLLOW(E) = {$, +} (end of input or +)
   - FOLLOW(E') = {$, +} (since E' is at the end of the production for E)
   - FOLLOW(T) = {+, $} (from the productions of E and E')

Using these sets, we can now construct the **parsing table**.

Non-terminal	`id`	`+`	`$`
`E`	`T E'`		
`E'`		`+ T E'`	`ε`
`T`	`id`		

The table entries represent the rules to apply for each non-terminal and lookahead combination.

Parsing Example

Let's parse the input string id + id using the parsing table.

1. **Initialization**:
   - Stack: [E] (start symbol)
   - Input: id + id
   - Lookahead: id
2. **Step 1**:
   - The top of the stack is E, and the lookahead is id.
   - According to the table, we apply E → T E'.
   - Stack: [T, E']
3. **Step 2**:

- The top of the stack is T, and the lookahead is id.
- According to the table, we apply T → id.
- Stack: [E']
- Input: + id (consume id)

4. **Step 3**:
   - The top of the stack is E', and the lookahead is +.
   - According to the table, we apply E' → + T E'.
   - Stack: [+, T, E']
5. **Step 4**:
   - The top of the stack is +, and the lookahead is +.
   - We match the lookahead with the top of the stack, and consume the +.
   - Stack: [T, E']
   - Input: id (consume +)
6. **Step 5**:
   - The top of the stack is T, and the lookahead is id.
   - According to the table, we apply T → id.
   - Stack: [E']
   - Input: `` (consume id)
7. **Step 6**:
   - The top of the stack is E', and the lookahead is $ (end of input).
   - According to the table, we apply E' → ε.
   - Stack: [] (empty)

The stack is empty, and the input has been fully consumed, so the input string id + id has been successfully parsed.

Advantages of Non-Recursive Predictive Parsing

1. **Efficiency**: Non-recursive predictive parsers avoid the overhead of recursion, making them more efficient in terms of both memory and performance, particularly for large inputs.
2. **Simple Implementation**: The use of a stack and a parsing table makes the implementation straightforward, and it's easy to understand compared to recursive descent parsers.
3. **Error Detection**: Predictive parsers can detect errors early by checking the parsing table for valid production rules. If the table does not provide a valid rule for a given non-terminal and lookahead token, an error is detected.
4. **No Backtracking**: Unlike recursive descent parsers, predictive parsers do not require backtracking, which can lead to inefficiency.

Limitations of Non-Recursive Predictive Parsing

1. **LL(1) Restriction**: Non-recursive predictive parsers are limited to LL(1) grammars, which are grammars that can be parsed with a single token lookahead. These grammars must be unambiguous and should not require backtracking or multiple alternatives with the same prefix.
2. **Complexity of Grammar**: For more complex grammars (i.e., those that are not LL(1)), other parsing techniques like **LR parsing** or **recursive descent with backtracking** may be necessary.

2.4.3 LL (1) GRAMMARS

LL(1) grammars are a subset of **context-free grammars (CFGs)** that can be parsed using a **predictive parser** with a single lookahead token. The term **LL(1)** is derived as follows:

- **L**: Left-to-right scanning of the input.
- **L**: Leftmost derivation (the leftmost non-terminal is expanded first).
- **1**: A single token lookahead (the parser looks at the next input token to decide the parsing action).

An **LL(1) grammar** can be parsed by an **LL(1) parser**, which uses a single token of lookahead to decide which rule to apply at each step of parsing. These grammars have specific properties that make them suitable for such parsers.

Key Properties of LL(1) Grammars

1. **Left-to-right**: The input string is read from left to right, meaning the parser processes the input sequentially, starting with the first token.
2. **Leftmost Derivation**: In a leftmost derivation, the leftmost non-terminal in the current sentential form is always replaced first. This ensures the parser constructs the parse tree in a left-to-right order.
3. **1-token Lookahead**: At each step of parsing, the decision about which production to apply is made using only the next token in the input. This is why **LL(1)** grammars are relatively simple to parse with a single-token lookahead.
4. **No Ambiguity**: LL(1) grammars must be unambiguous, meaning there must be no ambiguity in the choice of production rules when parsing any prefix of the input.
5. **No Left Recursion**: LL(1) grammars cannot have left recursion. This is because left recursion can cause infinite recursion in top-down parsers (like recursive descent or predictive parsers). Left recursion needs to be eliminated for a grammar to be LL(1).

Formal Definition of LL(1) Grammar

A grammar is **LL(1)** if for every non-terminal **A** and for every pair of distinct input tokens **a** and **b**:

- For each production rule **A → X1 X2 ... Xn**, the parser should be able to choose the correct rule based solely on the next input token **a**.

The **first condition** of LL(1) grammar requires that for each non-terminal **A**, the sets of **FIRST** sets of the productions of **A** must be disjoint (i.e., there must be no overlap between the FIRST sets).

- **FIRST**(A) is the set of terminal symbols that can appear at the beginning of a string derived from **A**.

The **second condition** of LL(1) grammars requires that if a production **A → ε** (i.e., an epsilon production) exists for a non-terminal **A**, then **FOLLOW**(A) (the set of terminal symbols that can appear immediately to the right of **A**) must be disjoint with **FIRST**(A).

- **FOLLOW**(A) is the set of terminal symbols that can appear immediately to the right of **A** in any derivation.

FIRST and FOLLOW Sets

The **FIRST** and **FOLLOW** sets are critical to understanding and constructing LL(1) parsers.

1. **FIRST Set**: The FIRST set of a non-terminal **A**, denoted as **FIRST(A)**, is the set of terminals that appear first in the derivation of **A**. For example:
    - If **A → aB** is a production, then **a** is in **FIRST(A)**.
    - If **A → ε** (epsilon, the empty string) is a production, then **ε** is in **FIRST(A)**.
2. **FOLLOW Set**: The FOLLOW set of a non-terminal **A**, denoted as **FOLLOW(A)**, is the set of terminals that can appear immediately to the right of **A** in some derivation of the grammar. For example:
    - If there is a production **B → αAβ**, then all the symbols in **FIRST(β)** are added to **FOLLOW(A)**.
    - If **A** is the start symbol, then **$** (end-of-input marker) is added to **FOLLOW(A)**.

Constructing an LL(1) Parsing Table

An **LL(1) parsing table** guides the parsing process. It is a two-dimensional table where:

- The **rows** represent the non-terminals in the grammar.
- The **columns** represent the terminals (including the end-of-input marker $).
- Each entry in the table contains a **production** rule to apply for a given non-terminal and lookahead terminal.

To construct the LL(1) table:

1. For each non-terminal **A** and terminal **a**, find the production rule **A → X1 X2 ... Xn** such that **a ∈ FIRST(X1 X2 ... Xn)**. Place this production rule in the table at position **[A, a]**.
2. If **A → ε** is a valid production and **a ∈ FOLLOW(A)**, place **A → ε** in the table at position **[A, a]**.
3. If multiple entries are found in any table position, the grammar is **not LL(1)**.

Example of an LL(1) Grammar

Consider the following simple grammar:

E → T E'

E' → + T E' | ε

T → id

Step 1: Compute FIRST and FOLLOW Sets

1. **FIRST(E)** = **FIRST(T)** = {id}

2. **FIRST(E')** = {+, ε}
3. **FIRST(T)** = {id}

For the **FOLLOW** sets:

1. **FOLLOW(E)** = {$, +}
2. **FOLLOW(E')** = {$, +}
3. **FOLLOW(T)** = {+, $}

Step 2: Construct the Parsing Table

Non-terminal	id	+	$
E	T E'		
E'		+ T E'	ε
T	id		

Step 3: Parsing Example

To parse the input id + id, follow these steps:

1. Start with the stack: [E] and input: id + id$.
2. The top of the stack is E, and the lookahead is id. According to the table, apply E → T E'.
3. The top of the stack is now T, and the lookahead is id. Apply T → id.
4. After consuming id, the stack is now [E'], and the lookahead is +.
5. The top of the stack is E', and the lookahead is +. Apply E' → + T E'.
6. The stack is now [T, E'] with the lookahead +.
7. The top of the stack is T, and the lookahead is id. Apply T → id.
8. After consuming id, the stack is [E'], and the lookahead is $.
9. The top of the stack is E', and the lookahead is $. Apply E' → ε.
10. The stack is empty, and the input has been fully consumed, so the parsing is successful.

2.4.4 A TRADITIONAL TOP-DOWNPARSER GENERATOR—YACC

YACC (Yet Another Compiler Compiler) is a traditional **bottom-up parser generator** widely used for creating parsers in the context of compiler construction. Although YACC is mainly associated with generating **LR parsers**, it's commonly mentioned in discussions about parser generators and is sometimes considered alongside other parser generators like **Bison** or **ANTLR**, which are used for generating both top-down and bottom-up parsers.

YACC Overview

YACC is a tool used to generate parsers based on a formal grammar. It works by taking a **context-free grammar (CFG)** and generating a parser, typically of the **LR** (left-to-right, rightmost derivation) type, which is capable of parsing a wide range of context-free grammars that a top-down parser (like **recursive descent**) might struggle with. However, it can be used in conjunction with **lexical analyzers** to produce a full compiler or interpreter.

Characteristics of YACC

1. **Bottom-Up Parsing**: YACC generates parsers using **LR parsing**, a class of bottom-up parsers. These parsers construct the parse tree by starting from the leaves (input symbols) and working their way up to the root (start symbol).
2. **Table-Driven Approach**: YACC generates parsing tables (action and goto tables) that are used by an **LL(1)** or **LR(1)** parser to decide which production rules to apply at each step of parsing.
3. **Context-Free Grammars**: YACC can generate parsers for context-free grammars (CFGs). It is particularly suited for parsing grammars that involve more complex structures, such as **ambiguous grammars**, that a simple top-down parser like **recursive descent** cannot handle.
4. **LALR Parsing**: YACC uses **LALR(1) (Lookahead-LR)** parsing, which is a more efficient variant of **SLR(1)** and **LR(1)** parsing. LALR parsers combine the efficiency of LR parsers with the reduced size of parsing tables, making them faster and more memory-efficient.
5. **Integration with Lex**: YACC works well in combination with **Lex** (or **Flex**, a more modern alternative). Lex is used for lexical analysis (tokenizing the input), and YACC handles syntactic analysis (parsing). Together, they form the core components of many compilers and interpreters.

How YACC Works

YACC operates in two stages:

1. **Grammar Specification**: The user provides a formal grammar that defines the syntax of the programming language (or data format) to be parsed. The grammar is written in a format that resembles **BNF (Backus-Naur Form)** or **EBNF (Extended BNF)**.
2. **Parser Generation**: YACC generates a C code file that contains the parsing tables and the parser's logic. The generated parser can then be compiled and linked with a lexical analyzer (produced by Lex or Flex) to create a full parser for the specified grammar.

YACC Syntax

A typical YACC input file consists of three sections:

Definitions Section: This section includes declarations for tokens (produced by Lex), as well as macros or type definitions.

```
%token INTEGER PLUS MINUS

%type <int> expression
```

Rules Section: This section defines grammar rules. Each rule consists of a **non-terminal** followed by a **production** and associated **action**. The action is written in C and specifies what should happen when a rule is applied.

```
expression:
    expression PLUS term { $$ = $1 + $3; }
  | expression MINUS term { $$ = $1 - $3; }
  | term
;
```

- The **non-terminal** is on the left side (e.g., expression).
- The **production** is on the right side (e.g., expression PLUS term).

- The **action** specifies what the parser should do when the rule is matched. In this case, it performs addition or subtraction.

User Code Section: This section allows the inclusion of C code that is needed for initialization, error handling, or other specific operations.

```
int yyerror(char *s) {
   fprintf(stderr, "Error: %s\n", s);
   return 0;
}
```

Example: Simple Calculator

Consider a simple calculator grammar that can evaluate arithmetic expressions with addition and subtraction.

YACC Input Example

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

%token INTEGER PLUS MINUS
%type <int> expression

%%

expression:
     expression PLUS term { $$ = $1 + $3; }
   | expression MINUS term { $$ = $1 - $3; }
   | term
   ;

term:
     INTEGER { $$ = $1; }
   ;

%%

int yyerror(char *s) {
```

```
    fprintf(stderr, "Error: %s\n", s);
    return 0;
}

int main(void) {
    yyparse();
    return 0;
}
```

- The grammar defines expression as an arithmetic expression involving addition and subtraction (PLUS and MINUS), and term as a basic integer (INTEGER).
- The **actions** (e.g., $$ = $1 + $3;) specify how the results of parsing are calculated.
- The main() function calls yyparse(), which is the parsing function generated by YACC.

Lex Input Example

To pair with this YACC input, we also need a Lex file for tokenizing the input:

```
%{
#include "y.tab.h"
%}

%%

[0-9]+    { yylval = atoi(yytext); return INTEGER; }
"+"       { return PLUS; }
"-"       { return MINUS; }
[ \t\n]   { /* ignore whitespace */ }
.         { return 0; }

%%
```

- This Lex file defines rules for recognizing integers ([0-9]+), plus (+), and minus (-).
- The yylval is set to the integer value of the token (for use by the parser).

Generating the Parser

To generate and compile the parser using YACC and Lex, you would follow these steps:

1. **Run Lex** to generate the lexical analyzer:

```
lex calc.l
```

2. **Run YACC** to generate the parser:

yacc -d calc.y

This creates two files: y.tab.c (the parser code) and y.tab.h (the header file with token definitions).

3. **Compile** the generated C code along with Lex-generated code:

gcc -o calc y.tab.c lex.yy.c -ll -ly

4. **Run the calculator**:

./calc

You can then input arithmetic expressions like 3 + 4 - 2, and the calculator will evaluate them.

Advantages of YACC

1. **Efficient Parsing**: YACC generates **LR parsers**, which are very efficient for a large class of context-free grammars. They can handle left recursion and ambiguous constructs that are difficult for simple top-down parsers like recursive descent.
2. **Error Handling**: YACC provides built-in error handling mechanisms that help in detecting syntax errors during parsing.
3. **Extensibility**: YACC can easily be extended with user-defined actions (written in C) for performing semantic analysis, syntax-directed translation, or even code generation.
4. **Well-Established**: YACC has been in use for decades and has a vast ecosystem, with many tutorials, documentation, and resources available.

Limitations of YACC

1. **Bottom-Up Parsing**: While YACC is great for **bottom-up parsing** (using **LR parsing**), it doesn't generate **top-down parsers**, which are used in simpler parsing techniques like recursive descent.
2. **Complexity**: YACC can be difficult to use with more complex or highly ambiguous grammars. It also requires knowledge of **LALR** parsing and how to handle **shift/reduce** conflicts, which can be challenging for beginners.
3. **Requires Lex**: YACC typically works in conjunction with Lex (or its modern variant, **Flex**) for lexical analysis, adding an extra step in the compilation process.
4. **Limited Error Reporting**: While YACC can handle basic errors, it may not always provide detailed or intuitive error messages, especially for more complex grammars.

2.5 BOTTOM-UP PARSING

2.5.1 SHIFT REDUCE PARSING

hift-reduce parsing is a common bottom-up parsing technique used in **LR parsers**. It builds the parse tree starting from the leaves (tokens) and works its way upwards towards the root (start symbol). This contrasts with **top-down parsing**, which begins with the start symbol and tries to derive the input string.

In a **shift-reduce parser**, the process is governed by two primary operations:

1. **Shift**: Move the next input token onto the stack.
2. **Reduce**: Replace a sequence of symbols on the stack that corresponds to a right-hand side of a production rule with the left-hand side (the non-terminal) of that rule.

Shift-reduce parsers use a **stack** to store symbols and a **buffer** to hold the remaining input. The parser repeatedly applies shift and reduce operations until it has successfully parsed the entire input.

Key Components of Shift-Reduce Parsing

1. **Stack**: Holds the symbols (both terminals and non-terminals) that have been shifted from the input or reduced by production rules.
2. **Input Buffer**: Holds the remaining input tokens (starting from the first token not yet processed).
3. **Action**: The parser decides whether to apply the shift operation or the reduce operation based on the current state and the top of the stack.
4. **Parsing Table**: A table that helps the parser decide whether to **shift** or **reduce** based on the current state and the lookahead token.

Shift-Reduce Parsing Steps

1. **Shift Operation**:
   - The parser moves the next token from the input buffer onto the stack.
   - This action is taken when the parser is not in a state where a reduction can be made.
2. **Reduce Operation**:
   - The parser applies a **reduction** when the top of the stack matches the right-hand side of a grammar rule.
   - The sequence of symbols on the stack that matches the right-hand side of a production is replaced with the non-terminal on the left-hand side of the production.
   - This reduces the stack size, and the parser moves one step closer to the root of the parse tree.
3. **Accepting**:
   - The parser has successfully parsed the input if the stack contains only the start symbol and the input buffer is empty.
4. **Error Handling**:
   - If neither a shift nor a reduction is applicable, or if the parser encounters an unexpected symbol, an error occurs.

Example: Shift-Reduce Parsing

Consider a simple grammar:

$S \rightarrow A\ B$

$A \rightarrow a$

$B \rightarrow b$

Input String: a b

We will demonstrate how a shift-reduce parser processes this string using this grammar.

1. **Initial State**:
   - **Stack**: [] (empty)
   - **Input**: a b

- **Action**: Since the stack is empty and no reduction is possible, we perform a **shift** to move a onto the stack.

2. **After Shift (Move a onto the stack)**:
   - **Stack**: [a]
   - **Input**: b
   - **Action**: No reduction possible. Perform **shift** to move b onto the stack.
3. **After Shift (Move b onto the stack)**:
   - **Stack**: [a, b]
   - **Input**: (empty)
   - **Action**: Now, a **reduction** is possible. According to the rule B → b, we reduce b to B.
4. **After Reduction (B → b)**:
   - **Stack**: [a, B]
   - **Input**: (empty)
   - **Action**: Now, a **reduction** is possible again. According to the rule S → A B, we reduce [a, B] to S.
5. **After Reduction (S → A B)**:
   - **Stack**: [S]
   - **Input**: (empty)
   - **Action**: The stack contains the start symbol S, and the input is empty, so the parser **accepts** the input as valid.

Parsing Table for Shift-Reduce Parsing

A **shift-reduce parser** can be guided by a **parsing table**. In the case of **LR parsers**, the table typically contains two types of entries:

1. **Action Table**: It tells the parser whether to **shift** or **reduce** based on the current state and lookahead token.
   - **Shift** means move the lookahead symbol onto the stack.
   - **Reduce** means apply a grammar rule to reduce the symbols on the stack.
   - **Accept** means the parsing is successful.
   - **Error** means an error has occurred (no action is defined).
2. **Goto Table**: It tells the parser what state to transition to after a reduction.

Example of Parsing Table for a Simple Grammar

Let's define a parsing table for the following grammar:

S → A B

A → a

B → b

1. **States**:
    - State 0: The parser has just begun parsing.
    - State 1: The parser has seen a and is waiting for B after A.
    - State 2: The parser has seen b and can reduce using B → b.
    - State 3: The parser has successfully parsed S → A B.
2. **Action Table** (indexed by state and lookahead token):

State/Token	a	b	$ (end)
0	shift 1		
1		shift 2	
2			reduce B → b
3			accept

3. **Goto Table** (indexed by state and non-terminal):

State/Non-Terminal	A	B	S
0	1		
1		2	
2			3

Conflict in Shift-Reduce Parsing

In shift-reduce parsing, conflicts can occur, typically in the following two forms:

1. **Shift/Reduce Conflict**: This happens when the parser has to choose between shifting a token or reducing using a rule, but both actions are valid for the current state and lookahead token.

Example:

S → A | B

A → a

B → a

If the input starts with a, the parser could either shift a or reduce a to A or B. This leads to a conflict.

2. **Reduce/Reduce Conflict**: This occurs when there are multiple possible reductions for the same portion of the input.

Example:

S → A | B

A → a

B → a

After shifting a, the parser may encounter a choice between reducing it to A or reducing it to B.

To resolve conflicts, more sophisticated parsing techniques like **LALR parsing** (used by YACC) or **GLR parsing** can be employed.

Advantages of Shift-Reduce Parsing

1. **Efficient**: Shift-reduce parsers can efficiently parse most programming languages and are widely used in real-world compilers.
2. **Handles Ambiguity**: While LR parsers can handle certain ambiguities that recursive descent parsers cannot, shift-reduce parsers are robust in handling them.
3. **Memory Efficiency**: Shift-reduce parsing is typically more memory-efficient than top-down approaches like recursive descent because the parser does not need to maintain multiple function calls or recursion depth.

2.5.2 LR PARSERS

LR Parsing is a class of bottom-up parsing techniques where the parsing process builds the parse tree from the leaves (input tokens) up to the root (start symbol). It is particularly powerful because it can handle a large subset of context-free grammars that other parsers, like recursive descent parsers, cannot.

In an **LR parser**, the "L" stands for **left-to-right** scanning of the input, and the "R" stands for **rightmost derivation** in reverse. This means that an LR parser reads the input from left to right, and the derivation of the input is done starting from the rightmost non-terminal.

2.5.2.1 SIMPLELR PARSER

An **SLR parser** is a type of **LR parser** that is relatively simple and efficient. It is based on the idea of **lookahead** (using a single lookahead symbol) and a set of **LR(0)** items. The key difference between the **SLR parser** and other types of LR parsers (like **LALR** or **Canonical LR**) is that it uses a simplified method for determining the reduction rules during parsing.

How an SLR Parser Works

1. **Input Buffer**: The input is stored in a buffer, and the parser reads the input from left to right one symbol at a time.
2. **Stack**: The parser uses a stack to store symbols, which include both non-terminals and terminals. Initially, the stack is empty.
3. **Action Table**: The action table guides the parser in terms of what action to take (shift, reduce, accept, or error) at each step. The action is determined based on the current state and the current input symbol.
4. **Goto Table**: The goto table helps the parser transition from one state to another after a reduction step. This table is crucial for managing non-terminal reductions in the parse process.

Steps of SLR Parsing

1. **Shift Operation**:
    - If the current symbol on the input is part of the terminal symbols (and no reduction rule applies), the parser **shifts** the symbol onto the stack and moves to the next input symbol.
2. **Reduce Operation**:
    - If the stack contains a sequence of symbols that matches the right-hand side of a production rule, the parser **reduces** the sequence of symbols to the corresponding non-terminal.
3. **Accept**:

- If the parser reaches a state where the stack contains the start symbol and the input buffer is empty, the input is successfully parsed, and the parser **accepts** the input.

4. **Error**:
   - If neither a shift nor a reduce operation is valid, the parser encounters an **error**.

SLR Parsing Tables

The **SLR parser** relies on two main tables:

1. **Action Table**:
   - This table defines the **shift** and **reduce** actions based on the current state and the input symbol.
   - **Shift**: If the parser should move the next token to the stack.
   - **Reduce**: If the parser should apply a production rule to reduce the symbols on the stack.
   - **Accept**: When the parsing is successful.
   - **Error**: When no valid action is defined (parsing error).
2. **Goto Table**:
   - This table is used after a reduction to determine the next state based on the non-terminal that was reduced.

Example of Simple LR Parsing

Consider a grammar:

$S \rightarrow A\ B$

$A \rightarrow a$

$B \rightarrow b$

Let's generate the **SLR parsing tables** for this grammar and parse the input string a b.

1. **States**:
   - The states correspond to different configurations of the parser, based on the grammar and the current input symbol. We will compute the **LR(0) items** (which are sets of production rules with a dot showing how much of the rule has been parsed).
2. **LR(0) Items**:
   - Initially, the parser starts with a state that is derived from the start symbol S and the production rules.
3. **Action Table**: The **Action table** tells us what to do based on the current state and the current input symbol.

State	`a`	`b`	`$` (end)
0	shift 1		
1		shift 2	
2			reduce `B → b`
3			accept

- **State 0**: When we see a, we **shift** to state 1.
- **State 1**: When we see b, we **shift** to state 2.
- **State 2**: When we see the end of the input ($), we can **reduce** using B → b.
- **State 3**: The parser **accepts** when it has successfully parsed S → A B and the input is empty.

4. **Goto Table**: The **Goto table** is used to transition between states after a reduction.

State	A	B	S
0	1		
1		2	
2			3

- After reducing B → b, the parser goes to state 3 and **accepts** the input.
- After reducing S → A B, the parser transitions to the accepting state.

Parsing the Input String a b

We begin with an empty stack and the input a b. Here's how the parser would proceed:

1. **Initial State**:
   - **Stack**: []
   - **Input**: a b
   - **Action**: Shift a onto the stack.
2. **After Shifting a**:
   - **Stack**: [a]
   - **Input**: b
   - **Action**: Shift b onto the stack.
3. **After Shifting b**:
   - **Stack**: [a, b]
   - **Input**: (empty)
   - **Action**: Reduce using the production B → b.
4. **After Reducing B → b**:
   - **Stack**: [a, B]
   - **Input**: (empty)
   - **Action**: Reduce using the production S → A B.
5. **After Reducing S → A B**:
   - **Stack**: [S]
   - **Input**: (empty)
   - **Action**: Accept the input.

The input a b has been successfully parsed.

Advantages of SLR Parsing

1. **Simple and Efficient**: SLR parsers are simpler and more efficient than other types of LR parsers because they use a simpler method for constructing the **action** table and determining when reductions should occur.
2. **Bottom-Up Parsing**: SLR parsing is a bottom-up approach that can handle more complex grammars than top-down parsers (like recursive descent).
3. **Large Grammar Handling**: SLR parsers can handle a wide range of programming languages, especially those with a large number of constructs.

Limitations of SLR Parsing

1. **Limited Power**: SLR parsers are not as powerful as other LR parsers like **LALR** or **Canonical LR** parsers. SLR parsers can encounter **shift/reduce conflicts** and **reduce/reduce conflicts** in certain ambiguous grammars.
2. **Conflict Handling**: In cases where the grammar has conflicts (like ambiguity), an SLR parser might fail or require additional techniques (e.g., more lookahead or more advanced LR parsing methods) to resolve them.

2.5.2.2 CANONICAL LR PARSER

A **Canonical LR (CLR) parser** is a more powerful and general type of **LR parser** that can handle a broader range of grammars compared to simpler LR parsers, such as **SLR** or **LALR** parsers. The term "Canonical" refers to the fact that this parser uses the most precise and detailed method for determining what actions to take during parsing. It is capable of handling grammars with greater complexity, such as ambiguous or context-sensitive constructs, by maintaining a larger state machine and more precise parsing tables.

The **LR** in "Canonical LR" stands for:

- **L**: Left-to-right scanning of the input (processing the input string from left to right).
- **R**: Rightmost derivation in reverse (deriving the input string from right to left).

A **Canonical LR parser** is often considered the most powerful of the LR parsing techniques because it can handle any context-free grammar that can be parsed by an LR parser, with the additional benefit of being deterministic.

Key Components of a Canonical LR Parser

A **Canonical LR parser** uses a **state machine** to parse the input. The key components involved in the parsing process are:

1. **State Stack**: This stack stores the states during parsing. A state corresponds to a certain point in the parsing process and determines what actions to take next based on the current input token.
2. **Input Buffer**: This buffer stores the input string that the parser will process, typically along with a lookahead token.
3. **Action Table**: The **action table** is the key to making decisions during parsing. It provides instructions on whether to:
    - **Shift** (move the next input symbol onto the stack),

- **Reduce** (apply a production rule to reduce a sequence of symbols on the stack to a non-terminal),
- **Accept** (indicating that the input has been successfully parsed),
- **Error** (if no valid action is defined for a given state and input symbol).

4. **Goto Table**: The **goto table** is used after a reduction to determine the next state, depending on the non-terminal that has been reduced.

The **Canonical LR parser** uses both the **action** and **goto** tables to decide whether to **shift**, **reduce**, or take other actions during parsing. The state machine transitions between states based on the stack and input symbol.

Steps in Canonical LR Parsing

1. **Initial State**:
   - The parser starts with an initial state, typically corresponding to the start symbol of the grammar. The first token from the input buffer is used to decide the first action.
2. **Shift Operation**:
   - If the top of the stack does not match a production's right-hand side, the parser shifts the next input symbol onto the stack.
3. **Reduce Operation**:
   - If the top of the stack matches the right-hand side of a production rule, the parser reduces that sequence of symbols to the corresponding non-terminal on the left-hand side of the production.
4. **Acceptance**:
   - When the stack contains only the start symbol, and the input buffer is empty, the parser has successfully parsed the input.
5. **Error Handling**:
   - If the parser encounters a situation where no valid action is defined (i.e., no shift or reduce operation is applicable), it results in a parsing error.

Constructing the Canonical LR Parser

To build a **Canonical LR parser**, the following steps are generally taken:

1. Constructing LR(0) Items:

The first step in creating a Canonical LR parser is to generate **LR(0) items**. These items are derived from the production rules and represent the different stages in the parsing process.

Each LR(0) item consists of a production rule and a **dot** (.) that marks the position in the rule that has been parsed. For example, if we have a production:

$A \rightarrow X\ Y\ Z$

An LR(0) item could be:

- $A \rightarrow .\ X\ Y\ Z$ (indicating that nothing from this production has been parsed yet),
- $A \rightarrow X\ .\ Y\ Z$ (indicating that X has been parsed),
- $A \rightarrow X\ Y\ .\ Z$ (indicating that X and Y have been parsed), and so on.

2. Building the DFA (Deterministic Finite Automaton):

The next step is to build a **deterministic finite automaton (DFA)** using the LR(0) items. This DFA represents the state machine that will guide the parser. Each state in the DFA corresponds to a set of LR(0) items, and transitions between states are based on input symbols (both terminals and non-terminals).

The states of the DFA are constructed by taking the **closure** of the LR(0) items. The **closure** of a set of items includes all items that can be derived from the items in the set by applying productions.

3. Generating the Action and Goto Tables:

The **action table** is constructed based on the DFA. For each state and input symbol, the parser determines whether to shift, reduce, or accept based on the transitions in the DFA and the lookahead symbol.

The **goto table** is constructed by determining the state that the parser should transition to after a reduction is performed. It helps the parser manage transitions between non-terminals after reducing a sequence of symbols.

Example: Canonical LR Parsing for a Simple Grammar

Let's consider a simple grammar:

$S \rightarrow A\ B$

$A \rightarrow a$

$B \rightarrow b$

Step 1: Create LR(0) Items

We generate the following LR(0) items from the production rules:

- $S \rightarrow .\ A\ B$
- $A \rightarrow .\ a$
- $B \rightarrow .\ b$

Step 2: Construct the DFA

The DFA is constructed based on these LR(0) items. Each state corresponds to a set of items, and transitions are made based on the current input symbol.

- **State 0**: $\{S \rightarrow .\ A\ B, A \rightarrow .\ a, B \rightarrow .\ b\}$
  - Transition on a → State 1: $\{A \rightarrow .\ a\}$
  - Transition on b → State 2: $\{B \rightarrow .\ b\}$
- **State 1**: $\{A \rightarrow a\ .\}$
  - Reduce by $A \rightarrow a$.
- **State 2**: $\{B \rightarrow b\ .\}$
  - Reduce by $B \rightarrow b$.

Step 3: Build Action and Goto Tables

The **action table** shows what the parser should do at each state given a particular input symbol:

State	a	b	$ (end)
0	shift 1	shift 2	
1	reduce A→a		
2		reduce B→b	

The **goto table** manages transitions after reductions, based on non-terminal symbols:

State	S	A	B
0	3	1	2
1			
2			

Step 4: Parse Input

Consider the input string a b. The steps to parse are:

1. Start in **State 0**:
    - Input: a b
    - Action: Shift to **State 1** (shift a).
2. In **State 1**:
    - Input: b
    - Action: Reduce by A → a.
3. After reduction (replace a with A), transition to **State 3** (using the goto table).
4. In **State 3**:
    - Input: b
    - Action: Shift to **State 2** (shift b).
5. In **State 2**:
    - Input: (empty)
    - Action: Reduce by B → b.
6. After reduction (replace b with B), transition to **State 3**.
7. In **State 3**:
    - Input: (empty)
    - Action: Accept the input as valid.

Advantages of Canonical LR Parsing

1. **Powerful**: The Canonical LR parser is the most general and powerful type of LR parser, capable of handling any **LR(1)** grammar without conflicts.
2. **Deterministic**: It is deterministic, meaning that there is always a well-defined action to take for each state and input symbol. This avoids ambiguity during parsing.

3. **Error Detection**: It can detect syntax errors early during parsing, which is beneficial for compiler construction.

Disadvantages of Canonical LR Parsing

1. **Memory and Complexity**: Canonical LR parsers tend to have larger parsing tables and a more complex state machine compared to simpler parsers like **SLR** or **LALR**.
2. **Efficiency**: The size of the **action** and **goto** tables can become very large, especially for more complex grammars. This can lead to inefficiencies in both memory usage and table lookup.
3. **Construction Time**: Constructing a Canonical LR parser can be computationally expensive, as it requires building and processing large DFA tables.

2.5.2.3 LALR PARSER

An **LALR (Look-Ahead LR)** parser is a more efficient variant of the **Canonical LR (CLR)** parser. It aims to retain the power of an LR parser but reduce the size of the parsing tables, making it a practical choice for most programming language grammars. LALR parsers combine the simplicity of **SLR (Simple LR)** parsing with the power of **Canonical LR** parsing, thus providing a balance between efficiency and the ability to handle more complex grammars.

Key Concepts of LALR Parsing

- **LALR** stands for **Look-Ahead LR**, where "LR" refers to the **Left-to-right** scanning of the input and **Rightmost derivation** in reverse.
- **LALR** parsers use a **single lookahead symbol** (i.e., they can make decisions based on the current state and the next input symbol) to make parsing decisions.

The difference between **LALR** and **Canonical LR** lies in the way they handle **states**. LALR parsers **merge similar states** in the canonical LR state machine to reduce the size of the parsing tables, which makes them more memory efficient while still retaining the ability to handle many context-free grammars that can't be parsed with simpler parsers like **SLR**.

Steps in LALR Parsing

An **LALR parser** follows a similar process as other **LR parsers** but with certain optimizations:

1. **State Machine Construction**:
   - The LALR parser first builds the **Canonical LR state machine** (with **LR(0) items**).
   - It then **merges** certain states that have identical **action** entries (shift/reduce decisions) but might differ in the lookahead symbol used for decision-making. This merging reduces the number of states and the size of the parsing table.
2. **Action Table**:
   - The **action table** guides the parser on whether to **shift**, **reduce**, **accept**, or **error** based on the current state and the lookahead symbol.
3. **Goto Table**:
   - The **goto table** directs the parser to the next state after a reduction has occurred, based on the non-terminal that was reduced.
4. **Parsing**:

- The parsing proceeds similarly to a Canonical LR parser, with the parser shifting tokens onto a stack, reducing sequences of symbols to non-terminals, and making decisions based on the lookahead token.
- The **shift** and **reduce** operations are determined based on the **action table** and the **goto table**.

Example of LALR Parsing Process

Consider a simple grammar:

S → A B

A → a

B → b

Step 1: Construct Canonical LR Items

The initial LR(0) items from the grammar are:

- S → . A B
- A → . a
- B → . b

Step 2: Build the Canonical LR State Machine

From the LR(0) items, we build the Canonical LR state machine. The machine has states based on the items, transitions based on input symbols, and actions such as shift and reduce.

State	a	b	$ (end)
0	shift 1	shift 2	
1			reduce A→a
2			reduce B→b
3			accept

Step 3: Merge Similar States

The main step that differentiates **LALR** parsing from **Canonical LR** parsing is the **state merging** process. In **Canonical LR**, each state represents a unique set of LR(0) items. However, **LALR** parsers merge states that have identical **shift** and **reduce** operations, effectively reducing the number of states. This reduction leads to smaller parsing tables.

For example, states 1 and 2 from the Canonical LR table might be merged into one state if the shift and reduce actions are the same. The idea is to keep the parser's capability to make decisions based on the current state and lookahead symbol, while reducing the number of states and entries in the table.

Step 4: Construct the Action and Goto Tables

After merging similar states, the parser constructs the **action table** and **goto table** based on the **merged** states. The **action table** will show shift/reduce actions based on the current state and lookahead symbol, while the **goto table** will describe the state transitions after reductions.

Advantages of LALR Parsing

1. **Efficient Table Size**:

- **LALR parsers** reduce the size of the parsing tables compared to **Canonical LR parsers**. This is achieved by merging states that have the same shift/reduce decisions, leading to reduced memory usage.

2. **More Powerful than SLR**:
   - **LALR parsers** can handle a broader set of grammars than **SLR parsers** while keeping the table size manageable. It can handle many grammars that are too complex for **SLR** parsing.
3. **Widely Used in Compiler Construction**:
   - **LALR parsers** are commonly used in practical compilers (e.g., **Yacc**, **Bison**) because they provide a good trade-off between parsing power and table size. They are capable of handling most programming languages' syntaxes.
4. **Deterministic Parsing**:
   - Like all **LR parsers**, **LALR parsers** are deterministic, meaning there is only one action for each state and lookahead symbol combination, which eliminates ambiguity in parsing.

Disadvantages of LALR Parsing

1. **Limited Grammar Support**:
   - While **LALR parsers** handle most programming language grammars, they still cannot handle certain complex grammars that **Canonical LR parsers** can, such as those that involve more complicated lookahead situations or ambiguities.
2. **State Merging Conflicts**:
   - During the state merging process, conflicts might arise if two states can have different reduce actions for the same lookahead symbol. This can lead to **shift/reduce** or **reduce/reduce** conflicts, though these conflicts are much rarer than in **SLR parsing**.
3. **Complexity in Grammar Design**:
   - Designing grammars that work well with **LALR parsers** requires careful attention, as certain ambiguities or conflicts may not be detected until parsing. In some cases, manual adjustments or grammar transformations may be required.

Example of LALR Parser Construction

For the grammar:

$S \rightarrow A\ B$

$A \rightarrow a$

$B \rightarrow b$

Step 1: Canonical LR Items

The LR(0) items for the grammar would be:

- $S \rightarrow .\ A\ B$
- $A \rightarrow .\ a$
- $B \rightarrow .\ b$

Step 2: Create Canonical LR States

We would then create the states for the Canonical LR machine, representing different positions in the parsing process:

- **State 0**: {S → . A B, A → . a, B → . b}
  - Transition on a → **State 1**
  - Transition on b → **State 2**
- **State 1**: {A → a .}
  - Reduce by A → a
- **State 2**: {B → b .}
  - Reduce by B → b

Step 3: Merge States

If the transition tables for **States 1** and **2** are identical in terms of shift/reduce actions, these states are merged into one. This step is the core of **LALR** parsing. The merged states will reduce the number of total states in the table.

Step 4: Construct Action and Goto Tables

The **action table** and **goto table** would be built by referring to the merged states. The **action table** would include shift/reduce decisions, and the **goto table** would include transitions for non-terminals.

2.5.3 USING AMBIGUOUSGRAMMARS

Ambiguous grammars present a unique challenge for bottom-up parsers because such grammars can generate multiple parse trees for the same input string. This creates uncertainty in parsing, making it difficult for a parser to determine the correct derivation or action.

What Are Ambiguous Grammars?

A **grammar** is said to be **ambiguous** if there exists a string in the language generated by the grammar that has **more than one leftmost derivation**, **more than one rightmost derivation**, or **more than one parse tree**.

For example, consider the following grammar for simple expressions involving addition (+) and multiplication (*):

E → E + E

E → E * E

E → id

With this grammar, the string id + id * id can be parsed in two different ways:

1. **(id + id) * id** (Group the first two ids together and then apply multiplication).
2. **id + (id * id)** (Apply multiplication first and then addition).

Ambiguity in Bottom-Up Parsing

In **bottom-up parsing**, the parser works by reducing the input string to the start symbol of the grammar. At each step, the parser looks for a sequence of symbols in the input that matches the right-hand side of a production rule and reduces it to the left-hand side (i.e., applies a production rule in reverse).

For an ambiguous grammar, the bottom-up parser can encounter situations where it has multiple valid reductions to apply at the same point in the input. This can lead to **shift/reduce conflicts** or **reduce/reduce conflicts**, where the parser doesn't know which reduction to apply. The result is that there could be more than one valid parse tree, which is problematic because the parser may not know which one to choose.

Conflict Types in Ambiguous Grammars

Shift/Reduce Conflict: A **shift/reduce conflict** occurs when the parser can either **shift** (read more input) or **reduce** (apply a production rule) based on the current state and the lookahead token. In an ambiguous grammar, there may be multiple valid reductions, causing a conflict between shifting and reducing.

- Example: Consider the input string id + id * id for the ambiguous expression grammar:

$E \rightarrow E + E$

$E \rightarrow E * E$

$E \rightarrow id$

- After reading the first id, the parser has a choice:
  - It can shift the + token, or
  - It can reduce id to E and wait for the next token.

This results in a **shift/reduce** conflict because the parser could proceed either by shifting or reducing.

Reduce/Reduce Conflict: A **reduce/reduce conflict** occurs when the parser has more than one rule it can apply to reduce the same sequence of symbols in the input.

Example: Consider the grammar:

$A \rightarrow a$

$A \rightarrow b$

1.

- If the input string is a b, the parser might encounter a point where it has to reduce either a to A or b to A. Since both reductions lead to A, this creates a **reduce/reduce conflict**.

Handling Ambiguity in Bottom-Up Parsing

Bottom-up parsers like **LR parsers** (including **SLR**, **LALR**, and **Canonical LR**) may struggle with ambiguous grammars. To resolve ambiguity in these parsers, a few techniques can be used:

1. Grammar Transformation:

One way to handle ambiguity is by **transforming the grammar** to remove the ambiguity. These transformations may include:

- **Left factoring**: This involves restructuring the grammar to ensure that the parser can decide the next step without ambiguity.
- **Eliminating left recursion**: Left recursion can cause infinite recursion in some parsers, and transforming the grammar to remove left recursion helps prevent ambiguity in parsing.

For example, consider the ambiguous grammar:

$E \rightarrow E + E$

$E \rightarrow E * E$

$E \rightarrow id$

By introducing **precedence** and **associativity rules**, we can resolve the ambiguity and specify which operator has higher precedence (e.g., multiplication has higher precedence than addition).

The grammar might be rewritten to reflect these rules:

$E \rightarrow E + T$

$E \rightarrow T$

$T \rightarrow T * id$

$T \rightarrow id$

Now, we specify that multiplication (*) has higher precedence over addition (+), effectively resolving the ambiguity in the original grammar.

2. Precedence and Associativity:

In practice, many programming language parsers handle ambiguity by defining **precedence** and **associativity** rules. These rules give the parser guidance on how to resolve conflicts when multiple reductions or shifts are possible.

- **Precedence**: Defines the priority of operators (e.g., * has higher precedence than +).
- **Associativity**: Specifies how operators of the same precedence are grouped (e.g., + is left-associative, meaning a + b + c is parsed as (a + b) + c).

For example, consider the following precedence and associativity definitions for arithmetic expressions:

%left + -

%left * /

This indicates that the parser should treat + and - as having the same precedence (left-associative) and that * and / have higher precedence than + and -.

These rules ensure that the parser always knows which reduction to perform when faced with multiple options, helping to resolve ambiguity.

3. Using More Powerful Parsers (Canonical LR):

While **SLR** or **LALR** parsers can handle many grammars, **Canonical LR** parsers are more powerful and can handle a broader range of grammars, including some ambiguous ones, by explicitly managing more states and parsing decisions. However, even **Canonical LR** parsers are limited when it comes to grammars that are inherently ambiguous.

If ambiguity remains in the grammar, it may require more sophisticated methods (such as manually intervening with parsing strategies or transforming the grammar) to ensure deterministic parsing.

4. Error Recovery:

When a bottom-up parser encounters ambiguity that it cannot resolve based on its grammar, it may need to employ **error recovery** strategies. Error recovery involves making assumptions about where the parser should backtrack or accept an error in order to continue processing the rest of the input.

- **Backtracking**: The parser may try different paths or production rules, and if one fails, it backtracks and tries another.

- **Error production rules**: Some parsers include rules that guide the parser in dealing with unexpected input, such as when a conflict arises.

Example of Ambiguous Grammar and Bottom-Up Parsing

Consider the ambiguous grammar for simple expressions:

$E \rightarrow E + E$

$E \rightarrow E * E$

$E \rightarrow id$

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

1. **(id + id) * id**: Apply the $E \rightarrow E + E$ rule first, reducing id + id to E, then apply $E \rightarrow E * E$ to reduce the result and id to E.
2. **id + (id * id)**: Apply $E \rightarrow E * E$ to reduce id * id first, then apply $E \rightarrow E + E$ to reduce id + E.

The bottom-up parser will encounter a conflict when it has read id + id, as it must decide whether to shift (read the * symbol) or reduce id + id to E. Similarly, when it has read id * id, it must decide whether to reduce id * id to E or shift the + symbol.

By defining **precedence** rules (giving * higher precedence than +), the ambiguity is resolved, and the correct parse tree becomes clear.