

Question: Explain the structure of the LR parsers and the difference between LR and LL parsers. (7 Marks)

Answer:

1. Structure of LR Parsers:

LR parsers are **bottom-up** parsers used in syntax analysis. They read input **Left-to-right** and produce a **Rightmost derivation in reverse**.

Main components of LR Parser:

1. **Input Buffer** – Holds the string to be parsed.
2. **Stack** – Stores grammar symbols and states.
3. **Parsing Table** – Contains:
 - **Action table:** For shift, reduce, accept or error.
 - **Goto table:** Used after reductions to jump to new states.
4. **Driver Program** – Controls parsing using stack, input buffer, and parsing table.

Working of LR Parser:

- LR parser checks **top of stack and current input**.
- Uses the **parsing table** to decide whether to:
 - **Shift** (move input to stack),
 - **Reduce** (replace RHS of production with LHS),
 - **Accept** (if input is valid), or
 - **Error**.

2. Difference between LR and LL Parsers:

Feature	LL Parser	LR Parser
Parsing Type	Top-down parsing	Bottom-up parsing
Input Direction	Left to right	Left to right
Derivation	Leftmost derivation	Rightmost derivation (in reverse)
Grammar Support	Less powerful (no left recursion)	More powerful (handles left recursion)
Stack Usage	Stores non-terminals	Stores both symbols and states
Error Detection	Weaker	Stronger
Example	Predictive Parser	SLR, LALR, Canonical LR

Question:

What is an LR(0) item? Construct an SLR parsing table for the grammar G:

$S \rightarrow L=R \mid R$
 $L \rightarrow *R \mid id$
 $R \rightarrow L$

Is it SLR(1) grammar?
[7M]

Answer:

1. What is an LR(0) item?

- An **LR(0) item** is a production with a **dot (•)** showing how much of the input we have seen.
- It helps us in building **parsing states** while constructing **DFA (canonical collection of items)**.

□ **Example:**

Production: $A \rightarrow \alpha\beta$

LR(0) items:

- $A \rightarrow \bullet\alpha\beta$ (before reading anything)
- $A \rightarrow \alpha\bullet\beta$ (after reading α)
- $A \rightarrow \alpha\beta\bullet$ (after reading everything)

2. Given Grammar:

Let's rewrite with a new start symbol S' :

$S' \rightarrow S$
 $S \rightarrow L = R \mid R$
 $L \rightarrow * R \mid id$
 $R \rightarrow L$

3. Step-by-step: Construct Canonical Collection of LR(0) items

□ **I0: Closure of $[S' \rightarrow \bullet S]$**

$S' \rightarrow \bullet S$
 $S \rightarrow \bullet L = R$
 $S \rightarrow \bullet R$
 $L \rightarrow \bullet * R$

$L \rightarrow \bullet id$
 $R \rightarrow \bullet L$

□ **GOTO(I0, S) = I1**

$S' \rightarrow S \bullet$

□ **GOTO(I0, L) = I2**

$S \rightarrow L \bullet = R$
 $R \rightarrow L \bullet$

□ **GOTO(I0, R) = I3**

$S \rightarrow R \bullet$

□ ***GOTO(I0,) = I4**

$L \rightarrow * \bullet R$
 $R \rightarrow \bullet L$
 $L \rightarrow \bullet * R$
 $L \rightarrow \bullet id$

□ **GOTO(I0, id) = I5**

$L \rightarrow id \bullet$

□ **GOTO(I2, =) = I6**

$S \rightarrow L = \bullet R$
 $R \rightarrow \bullet L$
 $L \rightarrow \bullet * R$
 $L \rightarrow \bullet id$

□ **GOTO(I4, R) = I7**

$L \rightarrow * R \bullet$

□ **GOTO(I6, R) = I8**

$S \rightarrow L = R \bullet$

□ **GOTO(I6, L) = I9**

$R \rightarrow L \bullet$

□ **GOTO(I4, L) = I10**

$R \rightarrow L \bullet$

□ **GOTO(I6, id) = I11**

$L \rightarrow id \bullet$

□ ***GOTO(I6,) = I12**

$L \rightarrow * \bullet R$

$R \rightarrow \bullet L$

$L \rightarrow \bullet * R$

$L \rightarrow \bullet id$

□ **GOTO(I12, R) = I13**

$L \rightarrow * R \bullet$

□ **GOTO(I12, L) = I14**

$R \rightarrow L \bullet$

□ **GOTO(I12, id) = I15**

$L \rightarrow id \bullet$

4. FOLLOW Sets (For SLR Parsing Table)

$FOLLOW(S) = \{ \$, \}$

$FOLLOW(L) = \{ =, \$ \}$

$FOLLOW(R) = \{ \$, \}$

5. SLR Parsing Table

State	id	*	=	=	\$	S	L	R
I0	s5	s4				1	2	3
I1					acc			
I2			s6					
I3			r2	r2				
I4	s5	s4				10		
I5			r5	r5	r5			
I6	s11	s12				9	8	
I7			r3	r3	r3			
I8			r1	r1	r1			
I9			r4	r4				
I10			r2	r2	r2			

State	id	*	=	=	\$	S	L	R
I11					r5	r5	r5	
I12	s5	s4					14	13
I13					r3	r3	r3	
I14					r4	r4	r4	
I15					r5	r5	r5	

Q) What are different Intermediate Code Forms? Discuss different Three Address Code (TAC) types and implementations of Three Address Statements.

Answer:

Intermediate Code Forms:

Intermediate Code Generation is the phase in compiler design where the source code is translated into a machine-independent code. This helps in easy code optimization and portability.

The main **Intermediate Code Forms** are:

1. **Syntax Trees**
 - Tree representation of code where each node is an operator, and children are operands.
 - Useful for semantic analysis.
2. **Postfix Notation (Polish Notation)**
 - Operators follow operands (e.g., $a + b$ becomes $a \ b \ +$).
 - No need for parentheses.
3. **Three Address Code (TAC)**
 - Each instruction contains at most **three addresses** (two operands and one result).
 - Example: $t1 = a + b$
4. **Quadruples**
 - Each instruction is represented using 4 fields:
(operator, arg1, arg2, result)
5. **Triples**
 - Similar to quadruples, but **no separate result field**, instead positions are used.
(operator, arg1, arg2)
6. **Indirect Triples**
 - Uses a pointer table to point to the instruction locations of triples.

Three Address Code (TAC):

- A common intermediate representation.
- Each statement has at most **one operator** and **two operands**.
- Easy for code optimization and target code generation.

Types of Three Address Code:

1. Assignment Statements

- o $x = y \text{ op } z$

Example: $t1 = a + b$

2. Unary Assignments

- o $x = \text{op } y$

Example: $t2 = -a$

3. Copy Statements

- o $x = y$

Example: $t3 = b$

4. Conditional Jump

- o $\text{if } x \text{ relop } y \text{ goto } L$

Example: $\text{if } a < b \text{ goto } L1$

5. Unconditional Jump

- o $\text{goto } L$

Example: $\text{goto } L2$

6. Label Definition

- o $L:$

Example: $L1:$

7. Procedure Calls and Returns

- o $\text{param } x, \text{call } p, n, \text{return } x$

Implementation of Three Address Statements:

Three main ways to implement TAC:

1. Quadruples

Format: $(\text{op}, \text{arg1}, \text{arg2}, \text{result})$

Example: $(+, a, b, t1)$

2. Triples

Format: $(\text{op}, \text{arg1}, \text{arg2})$

Example: $(+, a, b)$

- o Uses position instead of variable names.

3. Indirect Triples

- o Pointer table used to refer to instruction positions.

Question:

Write a note on simple type checker and list the different types of type checking with examples. (7M)

Answer:

□ Simple Type Checker:

A **simple type checker** is a part of the compiler that verifies whether the program follows the rules of the language regarding **data types**.

It ensures that variables are used correctly and operations are performed on compatible data types.

- For example, it checks that we are not adding a number and a string like:

```
int a = 5 + "hello"; // Error: incompatible types
```

□ Purpose of Type Checking:

- Prevents type errors
- Increases program safety
- Helps in early detection of bugs during compilation

□ Types of Type Checking:

1. Static Type Checking:

- Done at **compile-time**
- Detects errors before program runs
- Used in languages like C, C++, Java
- **Example:**
- `int a = "text"; // Error: assigning string to int`

2. Dynamic Type Checking:

- Done at **run-time**
- Errors are found during program execution
- Used in languages like Python, JavaScript
- **Example (Python):**
- `a = 5`
- `a = a + "hello" # Error occurs during execution`

3. Strong Type Checking:

- Strict checking of data types
- No implicit type conversions allowed
- Example languages: Python, Java
- **Example:**
- `a = 5 + "10" # Error: can't add int and string`

4. Weak Type Checking:

- Allows implicit conversions
- May lead to unexpected results
- Example languages: JavaScript, C
- **Example (JavaScript):**
- `var a = 5 + "10"; // Output: "510" (number + string)`

Question: List and explain different types of LR Parsers. Differentiate LR(0) and LR(1) items and their associated parsers. (7 Marks)

Answer:

Types of LR Parsers:

LR parsers are bottom-up parsers that read input from **Left to right** and construct **Rightmost derivation in reverse**. There are four main types:

1. **LR(0) Parser:**
 - Simplest type of LR parser.
 - No lookahead symbols.
 - Uses **LR(0) items**.
 - Limited grammar support.
2. **SLR(1) Parser (Simple LR):**
 - Based on LR(0) items + FOLLOW sets.
 - More powerful than LR(0).
 - Easier to implement but may fail for complex grammars.
3. **LALR(1) Parser (Look-Ahead LR):**
 - Combines LR(0) states having same core.
 - Uses **lookahead** like LR(1) but with fewer states.
 - Used in tools like **YACC**.
4. **Canonical LR(1) Parser:**
 - Most powerful and accurate.
 - Uses **LR(1) items** (includes lookahead symbol).
 - More complex and requires more memory.

Difference between LR(0) and LR(1) Items:

Feature	LR(0) Item	LR(1) Item
Definition	Production with a dot (•) at some position	Production with a dot (•) + lookahead symbol
Example	$A \rightarrow \alpha \cdot \beta$	$A \rightarrow \alpha \cdot \beta, a$
Lookahead	No lookahead	One lookahead symbol
Power	Less powerful	More powerful (can handle complex grammars)
Parser Name	LR(0), SLR(1)	Canonical LR(1), LALR(1)
Parsing Table Size	Smaller	Larger (for Canonical LR(1))

Question: What do you mean by attributed grammars? Discuss the translation scheme for converting an infix expression to its equivalent postfix form.

Answer:

Attributed Grammar:

An **attributed grammar** is a context-free grammar that is extended by associating **attributes** with the grammar symbols. These attributes hold information that helps in **semantic analysis** or **intermediate code generation**.

- Each grammar symbol can have one or more attributes.
- **Synthesized attributes** are computed from the attributes of child nodes.
- **Inherited attributes** are computed from parent or sibling nodes.

Translation Scheme for Infix to Postfix:

To convert an **infix expression** (e.g., $a + b$) to **postfix** (e.g., $ab+$), we use a **syntax-directed translation scheme**.

Grammar:

```
E → E + T    { print('+') }
E → E - T    { print('-') }
E → T
T → T * F    { print('*') }
T → T / F    { print('/') }
T → F
F → (E)
F → id       { print(id.name) }
```

Explanation:

- This is a **post-order traversal** translation.
- When we reach the **operator**, we have already printed both operands.
- `id.name` refers to the identifier like `a`, `b`, etc.

Example:

Input (Infix): $a + b * c$

Postfix Output: $abc*+$

Steps:

- $b * c \rightarrow bc*$
- $a + (bc*) \rightarrow abc*+$

This translation scheme helps the **compiler** to easily generate postfix code which is useful for **evaluation using stack**.

Question:

Define LR(k) parser. Explain the model of LR parser and various functions used in it for parser construction.

Answer:

Definition of LR(k) Parser:

LR(k) parser is a **bottom-up shift-reduce parser** that reads input **Left to right** and produces a **Rightmost derivation in reverse**.

Here,

- **L** = Scanning input from Left to right
- **R** = Producing Rightmost derivation
- **k** = Number of lookahead symbols used for making parsing decisions

Usually, **LR(1)** is used, which means one lookahead symbol.

Model of LR Parser:

An LR parser uses the following components:

1. **Input buffer** – Stores the string to be parsed.
2. **Stack** – Holds grammar symbols and states.
3. **Parsing Table** – Contains two parts:
 - **ACTION table** – Guides shift, reduce, accept, or error moves
 - **GOTO table** – Used for state transitions after reductions
4. **Driver Program** – Controls parsing by reading input, accessing the table, and updating stack.

Working of LR Parser:

- Starts with **initial state** in the stack.
- Reads **input symbols** one by one.
- Uses **ACTION and GOTO** tables to perform:
 - **Shift:** Push input symbol and next state
 - **Reduce:** Replace RHS of production with LHS and update state using GOTO
 - **Accept:** Successfully parsed input
 - **Error:** Input is not valid according to grammar

Functions used in LR Parser Construction:

1. **Closure(I):**
 - Adds all possible productions to the item set that can be derived from a non-terminal after dot (.)
 - Helps in building complete item sets
2. **Goto(I, X):**
 - Defines transitions from one item set to another by moving dot (.) over symbol X
 - Used to construct DFA (Deterministic Finite Automaton) of item sets
3. **Items(G):**
 - Generates all item sets (states) for grammar G
 - Combines closure and goto functions

4. Construct Parsing Table:

- Using items, closure, and goto, we build ACTION and GOTO tables
- Used in LR parsing

Conclusion:

LR(k) parsers are efficient and can handle **large class of grammars**.

They are used in tools like **YACC** for compiler design.

Common types: **SLR, Canonical LR, LALR**.

Q) How to handle ambiguity through LR parsers? Discuss about the Dangling-Else ambiguity.

Ans:

- **Ambiguity** occurs when a grammar can generate more than one parse tree for the same input string.
- **LR parsers** are powerful bottom-up parsers that can **detect ambiguity** during parsing.

Handling Ambiguity in LR Parsers:

- LR parsers handle ambiguity using **parser actions like shift and reduce**.
- When both **shift** and **reduce** actions are possible for a given input, it leads to **shift/reduce conflict**.
- LR parser **prefers “shift” over “reduce”** to resolve such conflicts, especially in **dangling-else** situations.

Dangling-Else Ambiguity:

- It is a common ambiguity in programming languages like C, C++, Java.

Example grammar:

```
stmt → if expr then stmt  
      | if expr then stmt else stmt  
      | other
```

- Input: if E1 then if E2 then S1 else S2
- This input can be parsed in **two ways**:
 1. else is associated with **inner if**
 2. else is associated with **outer if**→ **Ambiguous!**

Solution:

- LR parsers **resolve** it by always associating **else with the nearest unmatched if**.

- This removes ambiguity by applying the **shift operation** (i.e., shift "else" instead of reducing).

Conclusion:

- LR parsers **prefer shift over reduce** to solve the dangling-else problem.
- This technique ensures **correct parsing** of conditional statements without ambiguity.

Sure Manasa! Here's a clear and JNTUK-style answer for your question on Syntax Directed Translation Scheme (SDTS) for a **Simple Desk Calculator**:

Question:

Give Syntax Directed Translation Scheme (SDTS) for a Simple Desk Calculator.

Answer:

A **Syntax Directed Translation Scheme (SDTS)** for a **Simple Desk Calculator** evaluates arithmetic expressions using grammar rules with associated semantic actions.

This calculator supports operations like addition, subtraction, multiplication, and division.

Grammar with Semantic Rules:

$E \rightarrow E + T$	$\{ E.val = E1.val + T.val \}$
$E \rightarrow E - T$	$\{ E.val = E1.val - T.val \}$
$E \rightarrow T$	$\{ E.val = T.val \}$
$T \rightarrow T * F$	$\{ T.val = T1.val * F.val \}$
$T \rightarrow T / F$	$\{ T.val = T1.val / F.val \}$
$T \rightarrow F$	$\{ T.val = F.val \}$
$F \rightarrow (E)$	$\{ F.val = E.val \}$
$F \rightarrow \text{digit}$	$\{ F.val = \text{digit.lexval} \}$

Explanation:

- E, T, and F are non-terminals representing expressions, terms, and factors.
- digit is a terminal representing numeric input.
- .val is an attribute used to compute values during parsing.
- Semantic actions are written in curly braces {}.

Example:

For input: $3 + 4 * 2$

- It will be parsed as:
- $E \rightarrow E + T$
- $E \rightarrow T$
- $T \rightarrow F$
- $F \rightarrow \text{digit } (3)$
- $T \rightarrow T * F$
- $T \rightarrow F \rightarrow \text{digit } (4)$
- $F \rightarrow \text{digit } (2)$
- Computed as: $3 + (4 * 2) = 11$

Conclusion:

This SDTS helps in evaluating expressions **during parsing** by attaching actions to grammar rules.

It is used in **desk calculators** and **compiler design** for **expression evaluation**.

Question: What are the rules used to construct an SLR (Simple LR) Parser? Explain with an example.

Answer:

The construction of an SLR Parser involves the following main steps:

1. **Grammar Specification:**
 - The grammar must be in Context-Free Grammar (CFG) format. This means it should have non-terminal symbols, terminal symbols, production rules, and a start symbol.
2. **Constructing the Canonical Collection of LR(0) Items:**
 - The parser starts by constructing the set of LR(0) items, which are augmented productions of the grammar.
 - An LR(0) item is a production rule with a dot (.) indicating how much of the production has been read. For example, a production $A \rightarrow BC$ can have three possible items: $A \rightarrow .BC$, $A \rightarrow B.C$, and $A \rightarrow BC..$
3. **Constructing the Parsing Table:**
 - The SLR parser uses two tables:
 - **Action Table (Action[i, a]):** Specifies the action to take for each terminal symbol (shift, reduce, accept, or error).
 - **Goto Table (Goto[i, A]):** Specifies the state transition for non-terminal symbols.
 - The action table is filled by applying the following rules:
 - **Shift:** If an item has a dot before a terminal symbol, shift to a new state.

- **Reduce:** If the dot is at the end of a production, reduce the rule.
 - **Accept:** If the dot is at the end of the start production, accept the input.
 - **Error:** If none of the above actions apply, report an error.
4. **Handling Conflicts:**
- In an SLR parser, reduce conflicts are resolved based on the follow set of the non-terminal involved in the reduction. This is done by looking at the lookahead symbol (the symbol after the dot).
5. **Parsing Process:**
- The parsing process starts with the initial state and input symbols. Based on the action table, the parser performs shift and reduce operations until the input is completely parsed or an error occurs.

Example:

Consider the following grammar:

$S \rightarrow AA$
 $A \rightarrow aA \mid b$

Step 1: Augment the grammar:

$S' \rightarrow S$
 $S \rightarrow AA$
 $A \rightarrow aA \mid b$

Step 2: Construct LR(0) items:

- For $S' \rightarrow S$, the items are:
 - $S' \rightarrow .S$
 - $S' \rightarrow S.$
- For $S \rightarrow AA$, the items are:
 - $S \rightarrow .AA$
 - $S \rightarrow A.A$
 - $S \rightarrow AA.$
- For $A \rightarrow aA$, the items are:
 - $A \rightarrow .aA$
 - $A \rightarrow a.A$
- For $A \rightarrow b$, the items are:
 - $A \rightarrow .b$

Step 3: Create the action and goto tables:

- The action table will have shift operations for terminals (a, b), and reduce operations when the dot reaches the end of a production.
- The goto table will contain state transitions for non-terminals (S, A).

Question:

Generate the three-address code for the following code fragment:

```
a = b + 1
x = y + 3
y = a / b
a = b + c
```

Answer:

The three-address code for the above code fragment is:

```
t1 = b + 1          // a = b + 1
a = t1
t2 = y + 3          // x = y + 3
x = t2
t3 = a / b          // y = a / b
y = t3
t4 = b + c          // a = b + c
a = t4
```

This three-address code uses temporary variables (t_1 , t_2 , t_3 , t_4) to hold intermediate results and represents each operation in a simple and efficient manner. This approach is widely used in compiler design for generating intermediate code.

Question: Explain the LR Parsing Algorithm with steps.

Answer:

LR Parsing is a bottom-up parsing technique used to process deterministic context-free grammars (CFGs). It reads input from left to right and constructs a rightmost derivation in reverse.

Here's how LR parsing works:

1. **Start with an initial state:**
 - The parsing begins in the initial state (usually denoted as S_0). The input string and the parse stack are initialized.
2. **Read the input symbol:**
 - The parser reads the input string symbol by symbol, starting from the leftmost symbol.
3. **Use a parsing table:**
 - The parser uses an LR parsing table, which is made up of two components:
 - **Action Table:** Tells the parser what action to take (shift, reduce, accept, or error) based on the current state and the next input symbol.
 - **Goto Table:** Specifies the next state to transition to when a reduction is made.

4. **Shift operation:**
 - If the action table suggests "shift," the parser moves the next symbol from the input to the stack and transitions to the next state as defined by the Goto table.
5. **Reduce operation:**
 - If the action table suggests "reduce," the parser pops the right-hand side of a production from the stack and pushes the non-terminal (left-hand side of the production) onto the stack. The new state is determined from the Goto table.
6. **Acceptance:**
 - If the parser reaches a state where the stack contains the start symbol and the input is empty, the parsing is complete, and the string is accepted.
7. **Error handling:**
 - If none of the actions in the table apply, the input string is rejected as not belonging to the language.

Example:

Consider the grammar:

S → AB
A → a
B → b

Steps:

- Start at state S₀, read the first symbol 'a'.
- Shift 'a' to the stack (S₁), move to the next state.
- Read 'b', shift 'b' to the stack (S₂).
- At this point, the parser reduces 'A → a' and 'B → b'.
- The final reduction gives 'S → AB', and the parser reaches the accepting state.

Question:

Explain the various types of LR Parsers. (7M)

Answer:

LR Parsers are used in syntax analysis (parsing) for deterministic context-free grammars (CFGs). These parsers read input from left to right and produce a rightmost derivation. The different types of LR parsers are:

1. **Simple LR (SLR) Parser:**
 - **Working:** It is the simplest form of LR parsing. It uses a parsing table, which consists of actions (shift, reduce) based on the current state and the input symbol.
 - **Limitations:** SLR parsers use a simplified method for creating the parsing table, which can lead to conflicts in some grammars, making it less powerful than other LR parsers.
2. **Canonical LR Parser:**

- **Working:** This parser is more powerful than the SLR parser. It constructs a more detailed parsing table by considering the full context of the grammar rules.
 - **Advantage:** It is capable of handling more complex grammars than SLR parsers, including grammars with more conflicts.
 - **Limitations:** Canonical LR parsers are more memory-intensive and slower compared to other LR parsers.
3. **Lookahead LR (LALR) Parser:**
- **Working:** The LALR parser is a variation of the Canonical LR parser that merges states with similar action sets to reduce memory usage while maintaining parsing power.
 - **Advantage:** It is more efficient than Canonical LR parsers in terms of memory, while still handling most of the complex grammars.
 - **Usage:** LALR parsers are commonly used in practical compilers, such as in the YACC parser generator.
4. **Extended LR (ELR) Parser:**
- **Working:** This parser handles grammars that might not be processed by a simple LR or LALR parser, extending the capabilities of LR parsing further by adding more complex features.
 - **Advantage:** It can handle even more complex situations where other LR parsers would fail.
 - **Limitations:** It is very complex and not often used due to its computational overhead.

Summary:

- **SLR:** Simple but limited.
- **Canonical LR:** Powerful but resource-heavy.
- **LALR:** Efficient, commonly used in practical applications.
- **ELR:** Handles more complex cases but is complex to implement.

Question:

Explain in detail about the Error recovery in LR Parsing. [7M]

Answer:

Error recovery in LR parsing ensures that the parser can handle syntax errors and continue parsing without crashing. It allows the parser to skip over erroneous parts of the input and continue processing the valid portions.

Key Concepts:

1. **Error Detection:**
 - Errors are detected when the input symbol doesn't match the expected symbol in the parsing table, triggering the need for recovery.
2. **Error Recovery Strategies:**

- **Panic Mode:** The parser discards tokens until it finds a synchronizing token (like semicolons or braces) and resumes parsing from there.
 - **Phrase-Level Recovery:** The parser attempts to correct errors locally by inserting or deleting tokens, which is more precise but complex.
 - **Error Productions:** Special grammar productions are defined to handle specific errors, allowing the parser to shift or reduce tokens and continue parsing.
3. **Error Recovery Table:**
 - A table guides the parser on actions to take when an error is encountered, such as shifting or reducing tokens to an error state.
 4. **Resynchronization:**
 - After error recovery, the parser skips over erroneous input and resumes parsing at the next valid part, ensuring the process continues smoothly.
 5. **Challenges:**
 - Ambiguous errors, efficiency concerns, and preserving meaningful error messages can complicate recovery.

Question: Differentiate LR and LL Parsers. [7M]

Feature	LR Parser	LL Parser
Definition	LR parser is a bottom-up parser that reads input from Left to Right and constructs a Rightmost derivation in reverse.	LL parser is a top-down parser that reads input from Left to Right and constructs a Leftmost derivation.
Parsing Direction	Left to Right and Rightmost derivation.	Left to Right and Leftmost derivation.
Type of Grammar	Suitable for a larger class of grammars including context-free grammars.	Suitable for simpler context-free grammars.
Efficiency	More efficient for large and complex grammars.	Less efficient for complex grammars.
Stack Usage	Requires a stack for maintaining parsing state.	Requires recursion or a stack-based approach.
Handling Ambiguity	Handles ambiguity by considering multiple paths using lookahead.	Ambiguity can be problematic and often requires modifications in the grammar.
Parsing Table Size	Requires large parsing tables.	Parsing tables are smaller compared to LR.
Error Handling	Error recovery can be complex.	Easier error recovery with a simpler structure.

Question:

What is Dangling Else ambiguity? Explain. [7M]

Answer:

Dangling Else ambiguity occurs in programming languages, particularly in languages with if-else statements, when there is an ambiguity in the association of an "else" with an "if." It happens when there are nested if-else conditions and the compiler is unsure which "if" an "else" should belong to.

In simple terms, the "else" statement may be linked to the nearest "if" statement or may be intended to match an outer "if" statement. This creates ambiguity in the program's control flow.

Example of Dangling Else:

```
if (condition1)
    if (condition2)
        statement1;
else
    statement2;
```

In the above code, it's unclear whether the `else` is associated with the inner `if (condition2)` or the outer `if (condition1)`. According to the C programming language rules, the `else` will be paired with the closest preceding `if`, which may lead to unexpected results.

How to resolve Dangling Else ambiguity: To avoid such ambiguity, always use curly braces `{ }` to clearly define the scope of the `if-else` statements, even if there's only one statement in the block. This makes the code more readable and removes any ambiguity.

Correct version:

```
if (condition1) {
    if (condition2) {
        statement1;
    }
} else {
    statement2;
}
```

This resolves the ambiguity by making it clear that the `else` corresponds to the outer `if (condition1)`.

Question: Explain about SDD's (Syntax Directed Definitions) and also evaluation orders for SDD's. [7M]

Answer:

Syntax Directed Definitions (SDD): SDDs are a set of rules used to define the semantics of a programming language in conjunction with its syntax. They are used in syntax-directed translation, where attributes are associated with grammar rules. The main purpose of SDDs is to specify how the semantic rules of a language are evaluated based on its syntax.

An SDD consists of two key components:

1. **Syntax Rules:** These are the grammar rules that define the structure of the language.
2. **Semantic Rules:** These rules specify the meaning of the syntactic constructs. Each grammar rule is associated with semantic actions that compute the values of attributes.

Attributes in SDD:

- **Inherited Attributes:** These are passed down from parent to child nodes in a syntax tree. They are used to propagate information from the top of the tree to the leaves.
- **Synthesized Attributes:** These are computed from the children and are passed up to the parent node in the syntax tree.

Example of SDD: Consider the following SDD for a simple arithmetic expression:

Grammar:

```
E → E + T { E.val = E1.val + T.val }
E → T { E.val = T.val }
T → T * F { T.val = T1.val * F.val }
T → F { T.val = F.val }
F → ( E ) { F.val = E.val }
F → id { F.val = id.val }
```

In this example:

- `E.val`, `T.val`, `F.val`, and `id.val` represent the values of expressions, terms, factors, and identifiers respectively.
- The semantic rules define how the values of these components are calculated based on the arithmetic operations.

Evaluation Order for SDDs: The evaluation order refers to the sequence in which the semantic actions (rules) are applied during the parsing of a syntax tree. This order is crucial for determining how the attributes of nodes in the tree are computed.

There are two common evaluation orders:

1. **Top-Down Evaluation:** In this approach, inherited attributes are computed first. This is commonly used in situations where information needs to be propagated from the parent to the children.
2. **Bottom-Up Evaluation:** Here, synthesized attributes are computed first, and then the results are passed up to the parent node. This approach is used when the values of attributes depend on the children.

Example for Evaluation Order: Consider the expression $E \rightarrow E + T$. For this rule:

- First, compute the values of the left child (E) and the right child (T).

- Then, apply the semantic action to compute $E.val = E1.val + T.val$.

In the case of a bottom-up evaluation, we first evaluate $T.val$ and $F.val$, and then propagate these results upwards to compute the value for $E.val$.

Question:

Explain shift-reduce and reduce-reduce conflicts in the context of shift-reduce parsing. Provide an example.

Answer:

In **shift-reduce parsing**, the parser decides between two actions: shifting (moving to the next input symbol) or reducing (applying a production rule to a sequence of symbols on the stack). Conflicts arise when the parser encounters ambiguity in deciding which action to take.

1. **Shift-Reduce Conflict:** This occurs when the parser faces a situation where it can either:
 - **Shift:** Move the next input symbol onto the stack.
 - **Reduce:** Apply a production rule to reduce a sequence of symbols on the stack into a non-terminal.

A shift-reduce conflict happens when the parser is unsure whether to shift the next symbol or reduce the symbols on the stack.

Example: For the grammar:

$S \rightarrow A a$
 $A \rightarrow b$

Input: $b a$ The parser sees b and is uncertain whether to:

- **Shift** the a onto the stack, or
- **Reduce** b to A using the rule $A \rightarrow b$.

Here, a **shift-reduce conflict** occurs because the parser cannot decide whether to continue shifting or reduce the b to A .

2. **Reduce-Reduce Conflict:** This happens when the parser has multiple valid ways to reduce the symbols on the stack, but it cannot decide which production rule to apply.

Example: For the grammar:

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

Input: $b \ a$ After reading b , the parser can:

- **Reduce** b to A (using the rule $A \rightarrow b$) and then apply $S \rightarrow A \ a$, or
- **Reduce** b to B (using the rule $B \rightarrow b$) and then apply $S \rightarrow B \ b$.

Here, a **reduce-reduce conflict** arises because the parser cannot decide whether to apply $S \rightarrow A \ a$ or $S \rightarrow B \ b$.

Question:

Explain S-attributed and L-attributed definitions with an example. [7M]

Answer:

S-attributed Definitions:

An S-attributed definition is a type of attribute grammar where all the attributes are **synthesized** (computed from the children nodes in a parse tree). In this type of definition, the values of the attributes are calculated bottom-up, i.e., starting from the leaf nodes and moving towards the root.

- **Example:**

Consider the expression evaluation for a simple arithmetic expression $E \rightarrow E1 + E2$.

We can assign synthesized attributes to nodes in the parse tree. Suppose we are calculating the value of an expression.

$E \rightarrow E1 + E2$

Here, we assign a synthesized attribute val to E , $E1$, and $E2$:

- $E1.val = E1.value$
- $E2.val = E2.value$
- $E.val = E1.val + E2.val$

The values of $E1$ and $E2$ are computed first (from the leaf nodes), and then the value of E is computed based on those values. This is a classic example of an S-attributed definition.

L-attributed Definitions:

An L-attributed definition is a type of attribute grammar where attributes can be **synthesized** or **inherited**. It is a more generalized form of S-attributed definition. In an L-attributed grammar, **inherited** attributes can be passed from the parent node to its children, and **synthesized** attributes can be computed bottom-up. The "L" in L-attributed stands for "Left-to-right," meaning that attributes are propagated from left to right in the parse tree.

- **Example:**
Consider the grammar for assignment:

`S -> id = E`

Here, we have synthesized attributes (`val`) for `E` and inherited attributes (`env`) for `id`. The value of the `id` is inherited from the environment, and the value of the expression is synthesized.

`S -> id = E`
`env(id) = E.val`

- `E.val` is the synthesized attribute, which is computed from the subtrees of `E`.
- `env(id)` is the inherited attribute, which is passed down from the parent node to `id`.

In an L-attributed definition, inherited attributes can only be passed from left to right along the parse tree, making it efficient for left-to-right evaluation.

Summary:

- **S-attributed:** Only **synthesized** attributes, calculated bottom-up from leaf to root.
- **L-attributed:** Both **synthesized** and **inherited** attributes, where inherited attributes are passed from parent to child, following a left-to-right order.

Note: L-attributed grammars are more general and can be used to define more complex language features, while S-attributed grammars are simpler and more efficient for certain tasks, like evaluating arithmetic expressions.

Question: Justify the following statements:

- (i) If there is no shift-reduce conflict in CLR(1) parsing table, then there is no shift-reduce conflict in LALR(1) parsing table of the same grammar.
- (ii) Even if there is no reduce-reduce conflict in CLR(1) parsing table, there may be reduce-reduce conflict in LALR(1) parsing table of the same grammar.

Answer:

(i) If there is no shift-reduce conflict in CLR(1) parsing table, then there is no shift-reduce conflict in LALR(1) parsing table of the same grammar:

1. **CLR(1) Parsing Table:**
 - The CLR(1) (Canonical LR) parser uses the full set of items for each state and considers one lookahead symbol for decision making.
 - It produces a detailed parsing table where each state transition is defined precisely based on the lookahead symbol.
2. **LALR(1) Parsing Table:**

- The LALR(1) parser (Lookahead LR) uses a simplified set of items, merging similar items from the CLR(1) table, reducing the number of states.
 - Despite merging items, LALR(1) still uses one lookahead symbol for each parsing decision.
3. **No Shift-Reduce Conflict in CLR(1):**
 - If a shift-reduce conflict does not occur in the CLR(1) parsing table, it means the parser can unambiguously decide whether to shift or reduce for any given state and lookahead symbol.
 4. **Conclusion:**
 - Since LALR(1) parsing uses fewer states but still maintains the same decision-making process for shifts and reductions (with one lookahead symbol), if there is no shift-reduce conflict in the CLR(1) table, there will be no shift-reduce conflict in the LALR(1) table.

Therefore, the first statement is true.

(ii) Even if there is no reduce-reduce conflict in CLR(1) parsing table, there may be reduce-reduce conflict in LALR(1) parsing table of the same grammar:

1. **CLR(1) Parsing Table:**
 - CLR(1) parsing creates a fully expanded set of items, where each state explicitly represents the possible reductions that can occur for a particular lookahead symbol.
 - It ensures that each reduction decision is unambiguous for a given state.
2. **LALR(1) Parsing Table:**
 - LALR(1) parsing table merges similar states, reducing the number of states. However, this merging may introduce ambiguity in some cases, especially for reduce actions.
3. **Reduce-Reduce Conflict in CLR(1):**
 - If there is no reduce-reduce conflict in the CLR(1) parsing table, it means that no two reductions can occur simultaneously for the same lookahead symbol in the same state.
4. **Merging of States in LALR(1):**
 - When states are merged in the LALR(1) parsing table, it can combine two or more states that were previously distinct in CLR(1), leading to a situation where multiple reductions might be possible in the same state for the same lookahead symbol.
 - This can lead to a **reduce-reduce conflict**, where the parser cannot unambiguously decide which reduction to perform.
5. **Conclusion:**
 - Even if CLR(1) does not have a reduce-reduce conflict, the merging of states in LALR(1) can create a scenario where such a conflict arises.

Therefore, the second statement is true.

Final Conclusion:

1. (i) True: If there is no shift-reduce conflict in CLR(1), there will not be any shift-reduce conflict in LALR(1).
2. (ii) True: Even if there is no reduce-reduce conflict in CLR(1), there may be reduce-reduce conflict in LALR(1) due to state merging.

Question:

Write the quadruple, triple, and indirect triple for the statement:

$x = y * -z + y * -z$

Answer:

For the given statement $x = y * -z + y * -z$, we break it down into parts and convert it into the three representations.

1. Quadruple Representation:

A quadruple consists of (operator, operand1, operand2, result). We first need to represent each sub-expression:

- $y * -z$
- $y * -z$
- The sum of the two results.

Here's how it can be represented in quadruple form:

1. (*, y, -z, t1) // $t1 = y * -z$
2. (*, y, -z, t2) // $t2 = y * -z$
3. (+, t1, t2, t3) // $t3 = t1 + t2$
4. (=, t3, -, x) // $x = t3$

2. Triple Representation:

A triple consists of (operator, operand1, operand2), and the result is referred by index.

1. (*, y, -z) // $t1 = y * -z$
2. (*, y, -z) // $t2 = y * -z$
3. (+, t1, t2) // $t3 = t1 + t2$
4. (=, t3, -) // $x = t3$

3. Indirect Triple Representation:

An indirect triple uses the index of the previous triple as an operand reference.

1. $t_1 = (*, y, -z)$ // First operation
2. $t_2 = (*, y, -z)$ // Second operation
3. $t_3 = (+, t_1, t_2)$ // Add the results of the previous operations
4. $t_4 = (=, t_3, -)$ // Assign the result to x

Thus, the indirect triple will reference the previous operation indices.

This concise form is the way you can break down the expression into more optimized intermediate code representations.

Question: Compare and contrast SLR(1), CLR(1), and LALR(1) parsers. [7M]

Feature	SLR(1) Parser	CLR(1) Parser	LALR(1) Parser
Full Form	Simple LR (1)	Canonical LR (1)	Look-Ahead LR (1)
Parsing Table Construction	Uses a simplified method to construct the parsing table	Uses a complete method with all possible states for the parsing table	Similar to CLR(1) but merges some states for optimization
Conflict Detection	Can handle shift-reduce conflicts easily, but may have reduced power for certain grammars	Resolves both shift-reduce and reduce-reduce conflicts more effectively	Similar to CLR(1) but with fewer states, reducing complexity while avoiding conflicts
Number of States	Fewer states than CLR(1), leading to a smaller parsing table	More states, leading to a larger parsing table with detailed information	Fewer states than CLR(1), optimized by merging similar states
Efficiency	Efficient but less powerful due to simplified state merging	Most powerful, but can be inefficient with large grammars due to large state space	More efficient than CLR(1) while maintaining most of its power
Grammar Handling	Can handle simpler context-free grammars with fewer conflicts	Handles more complex grammars with a better conflict resolution mechanism	Handles a wider range of grammars than SLR(1), with fewer states

Feature	SLR(1) Parser	CLR(1) Parser	LALR(1) Parser
Usage	Ideal for simple grammars with few conflicts	Suitable for complex grammars, but may be slower due to larger tables	A good compromise, suitable for most practical applications with a balance of power and efficiency

Question: Write the triples for the statement $c[i] = a[i] + b[i]$, where a , b , and c are arrays of integers.

Answer:

The statement $c[i] = a[i] + b[i]$ can be broken down into the following triples:

1. $(a[i], b[i], t1)$ – This represents the addition of elements at index i of arrays a and b , and stores the result in temporary variable $t1$.
2. $**(t1, c[i], =)$ – This represents assigning the value of temporary variable $t1$ to $c[i]$.

Thus, the triples are:

- $(a[i], b[i], t1)$
- $(t1, c[i], =)$

These triples represent the operations performed in the given statement.

Question: Compare and contrast top-down parsing and bottom-up parsing. [7M]

Aspect	Top-Down Parsing	Bottom-Up Parsing
Definition	In top-down parsing, the parser starts from the start symbol and works its way down to the terminals.	In bottom-up parsing, the parser starts from the input symbols and works its way up to the start symbol.
Approach	Uses a predictive approach based on grammar rules.	Uses a shift-reduce approach to reduce input to the start symbol.
Type of Grammar	Best suited for LL grammars.	Best suited for LR grammars.
Parsing Direction	Top to bottom, from the start symbol to the leaves.	Bottom to top, from the leaves to the start symbol.
Efficiency	Less efficient for ambiguous grammars or left recursion.	More efficient for handling a wide range of grammars.
Error Handling	Difficult to handle errors, especially in ambiguous cases.	Easier to handle errors due to shift-reduce operations.
Complexity	Simpler and easier to implement for small grammars.	More complex due to the need for managing multiple states.

Aspect	Top-Down Parsing	Bottom-Up Parsing
Backtracking	Often requires backtracking, leading to inefficiency.	Avoids backtracking, making it more efficient for larger grammars.

Question: Explain synthesized and inherited attributes with examples. [7M]

Answer:

In compiler design, attributes are used to store the information that is passed between different nodes in the syntax tree. There are two types of attributes:

1. **Synthesized Attributes:** These attributes are computed at the node itself based on its children. They are "bottom-up" in nature, meaning the value is passed from the leaves of the syntax tree to the root. Synthesized attributes are typically used to represent values that depend on the information derived from child nodes.

Example: Consider a simple arithmetic expression $a + b$. If a and b are leaves of the syntax tree (representing operands), then their values are passed up to the parent node (which represents the $+$ operator) as synthesized attributes.

- $\text{Value}(a) = 5$
- $\text{Value}(b) = 10$
- Parent node $+$ computes: $\text{Value}(+) = \text{Value}(a) + \text{Value}(b) = 5 + 10 = 15$

2. **Inherited Attributes:** These attributes are passed from the parent node to its children. They are "top-down" in nature, meaning the value is passed from the root of the tree to the leaves. Inherited attributes are used to convey context-specific information from the parent to the child nodes.

Example: Consider a simple assignment statement $x = a + b$. Here, the parent node (assignment operator) may have an inherited attribute, such as the type of the variable x . This type needs to be passed down to the child nodes representing a and b so that the addition operation can be performed correctly.

- $\text{Type}(x) = \text{int}$ (inherited from the parent node)
- $\text{Type}(a) = \text{Type}(x) = \text{int}$ (inherited)
- $\text{Type}(b) = \text{Type}(x) = \text{int}$ (inherited)

This ensures that the addition operation is valid for the given type.

Conclusion:

- Synthesized attributes are derived from the child nodes and passed upwards.
- Inherited attributes are passed from parent nodes to child nodes.