Chapter 05: Bottom-Up Parsing

Introduction

Bottom-up parsing is a fundamental approach in syntax analysis that constructs the parse tree from the input tokens up to the start symbol. This chapter introduces the principles of LR parsing and explores its main variants: LR(0), SLR(1), LR(1), and LALR(1).

1. Definition

Bottom-up parsing is a syntax analysis technique that builds a parse tree from the leaves (input tokens) up to the root (start symbol) of the grammar. It attempts to reconstruct the **rightmost derivation in reverse**, using a process called **reduction**: it repeatedly replaces sub-strings of the input that match the right-hand side of a grammar production with the corresponding left-hand side non-terminal. This process continues until the entire input is reduced to the start symbol or an error is detected.

Example : Consider the grammar:

S → aABe
A → Abc | b
B → d

Let's derive 'abbcde' using **rightmost derivation** :

$S \Rightarrow aA\underline{\mathbf{B}}e$
$\quad \Rightarrow a\underline{\mathbf{A}}de$
$\quad \Rightarrow a\underline{\mathbf{A}}bcde$
$\quad \Rightarrow abbcde$

Bottom-up parsing aims to reverse these steps, finding the right-hand side of a production in the current string and replacing it with the left-hand side. In other words, it starts from the input string and **reduce handles** (*i.e.* matches of right-hand sides of rules) in the **reverse order** of a rightmost derivation until you get S.

The string “abbcde” can be reduced to “S” as follows :

⇒ abbcde

⇒ a**A**bcde (replace b by A using A → b)

⇒ aAde (replace Abc by A using A → Abc)

⇒ aA**B**e (replace d by B using B → d)

⇒ **S** (replace aABe by S using S → aABe)

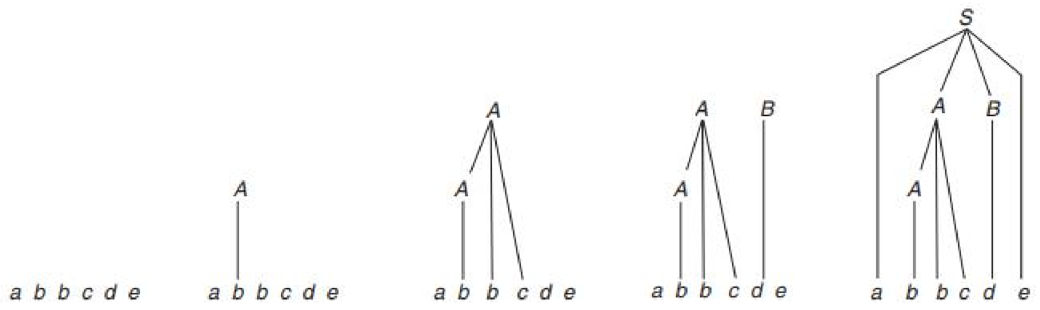


Figure 1. Parse Tree of the Input String : abbcde

2. Bottom-Up Parsing Process

The core of bottom-up parsing, particularly **shift-reduce parsing**, involves using a **stack** to store processed symbols and an **input buffer** for remaining tokens. The parser operates by repeatedly choosing between two fundamental actions: **shift** and **reduce**.

The bottom-up parsing process follows these steps:

1) **Scan**: Read input tokens from **left to right**
2) **Shift**: Push tokens onto a stack
3) **Reduce**: When the top of the stack matches the right-hand side of a production rule, replace it with the left-hand side non-terminal
4) **Accept**: When the stack contains only the start symbol and input is exhausted
5) **Error**: When no valid shift or reduce action is possible

3. General Algorithm

A general shift-reduce parser operates as follows:

```
Initialize stack as empty
Append $ to end of input string $a_1a_2a_3...a_n$
Repeat
    If top of stack matches handle $\beta$ in some production $A \rightarrow \beta$ then
        Pop $\beta$ from stack
        Push $A$ onto stack
    Else if input is not empty then
        Shift next input symbol $a_i$ onto stack
    Else
        Report error and halt

Until stack == [$S$] and input == $
```

Note **:** A **handle** is sub-string of the input that matches the right-hand side of a some production $A \rightarrow \beta$ and whose reduction represents one step in the reverse derivation of the input string.

Example : Consider the previous grammar. Let's trace the parsing process for the input string "**abbcde**":

Stack	*Input*	*Action*
$	abbcde$	**shift** a
$a	bbcde$	**shift** b
$ab	bcde$	**reduce** by A → b
$aA	bcde$	**shift** b
$aAb	cde$	**shift** c
$aAbc	de$	**reduce** by A → Abc
$aA	de$	**shift** d
$aAd	e$	**reduce** by B → d
$aAB	e$	**shift** e
$aABe	$	**reduce** by S → aABe
$S	$	**accept**

4. Types of Bottom-Up Parsers

The primary challenge for bottom-up parsers lies in deterministically deciding which action to take at each step. This leads to potential conflicts:

- **Shift/Reduce Conflicts**: When the parser cannot decide whether to shift the next token or reduce the current stack contents.
- **Reduce/Reduce Conflicts**: When multiple production rules could apply for reduction.

These challenges are addressed by different types of bottom-up parsers, which vary in their parsing table construction and lookahead capabilities. Key types include:

1) **Operator-Precedence Parsers :** A simpler bottom-up method for operator grammars, using precedence rules.
2) **LR Parsers (Left-to-right, Rightmost derivation) :** A powerful family of parsers that use state machines and lookahead :
   - **LR(0)** : The simplest form, but limited in practical use due to lack of lookahead.
   - **SLR (Simple LR)** : Improves LR(0) by using follow sets for conflict resolution.
   - **CLR (Canonical LR)** : More precise than SLR but requires larger parsing tables.
   - **LALR (Look-Ahead LR)** : Balances power and efficiency, widely used in parser generators (*e.g.,* Yacc/Bison).

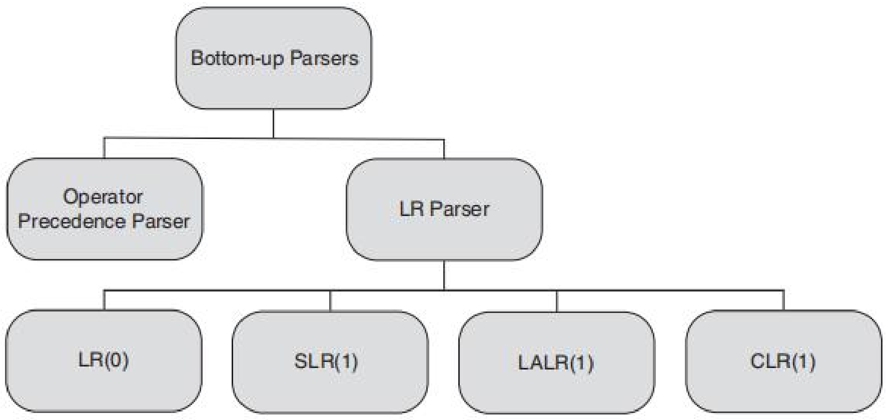


Figure 2. Types of Bottom-Up Parsers

5. LR Parsing

LR parsing is a deterministic bottom-up parsing technique for context-free grammars that reads input **Left-to-right (L)** and produces a **Rightmost derivation (R) in reverse**. It systematically constructs a parse tree by shifting input symbols onto a stack and applying reductions based on a sequence of grammar productions, guided by a pre-constructed **parsing table**.

Thus, to implement an LR parser, it is essential to understand the elements that work together during parsing:

- **Input buffer**: Stores the input string to be parsed followed by an end-of-input marker ($). The parser reads symbols from left to right using an input pointer.
- **Stack**: A pushdown stack that stores grammar symbols and state numbers alternately. It starts with state 0 :
  - ✓ **Symbols**: The grammar symbols (terminals and non-terminals) that appear on the stack.
  - ✓ **States**: Represent sets of LR items.
- **Parsing table** : is formally a two-dimensional array containing ACTION and GOTO entries.
  - ✓ **ACTION table**: Tells the parser whether to shift, reduce, accept, or report an error based on the current state and input symbol (terminal).
  - ✓ **GOTO table**: Tells the parser which state to go to after a reduction based on the current state and non-terminal.

These components work together to manage how the input is parsed and guide the parser through decisions.

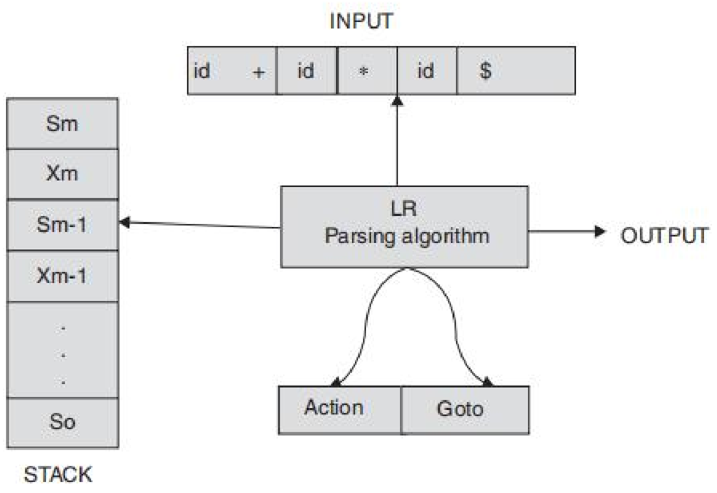


Figure 3. Model of LR Parsing

5.1. Constructing LR Parsing Table

The construction of parsing tables for LR parsers follows a systematic procedure that builds upon the concept of **LR items** and **canonical collections**. The complete process is as follows:

1) **Augment the grammar**: Add a new start production $S' \rightarrow S$, where S is the original start symbol. This ensures a unique accept state.

2) **Generate LR items**: For every production, create items by inserting **a dot** (•) at every possible position in the right-hand side.
3) **Build the canonical collection of item sets (states)** using the CLOSURE and GOTO operations.
4) **Construct the ACTION and GOTO tables**: For each state and grammar symbol, determine the correct parsing action: shift, reduce, accept, or error.

These construction steps are the same across all LR parsers; however, they differ in the type of **item sets** used (LR(0) *vs.* LR(1)) and in **the rules for filling the ACTION table**, particularly how and when reduce actions are applied.

5.2. LR Parsing Stack-based Algorithm

All LR parsing variants (LR(0), SLR, LALR(1), CLR(1)) use the identical parsing algorithm and driver program. The variants differ only in their table construction methods, not in table usage. The general parsing algorithm is:

1) Initialize the stack with **state 0**.
2) Repeat the following steps until the input is accepted or an error is detected:
   - Let s be the state on top of the stack and a be the current input symbol.
   - Consult $\boldsymbol{ACTION[s, a]}$:
     - If $\boldsymbol{ACTION[s, a] = shift\ i}$: push a then state i onto the stack, and advance the input.
     - If $\boldsymbol{ACTION[s, a] = reduce\ A \to \beta}$: pop $\underline{\boldsymbol{2 * |\beta|}}$ <u>symbols</u> from the stack, let $\boldsymbol{s'}$ be the state now on top, then push A and the state $\boldsymbol{GOTO[s', A]}$.
     - If $\boldsymbol{ACTION[s, a] = accept}$: parsing is successful.
     - If $\boldsymbol{ACTION[s, a] = error}$: report a syntax error and halt.

6. LR(0) Parsing

LR(0) parsing is the simplest form of LR parsing. It introduces the essential ideas used in all LR parsers. Although not powerful enough for practical use with real-world programming languages due to their inability to use lookahead symbols, LR(0) form the foundation for more powerful LR parsing methods and can handle a subset of context-free grammars without ambiguity.

In this section, we will illustrate the LR(0) parsing process through a step-by-step example. We begin by defining the basic concepts, such as augmented grammar and LR(0) item sets, and proceed to construct the canonical collection of LR(0) items using the key operations: closure and GOTO. We then explain how to use these item sets to build a deterministic finite automaton (DFA) representing parser states and transitions. From this DFA, we derive the parsing tables (ACTION and GOTO). Finally, we demonstrate the complete parsing of an input string using these tables and the stack-based LR(0) parsing algorithm.

Note : A grammar is said to be LR(0) if it can be parsed by an LR(0) parser.

6.1. Augmented grammar

An augmented grammar is a modified version of a context-free grammar where a new start symbol is added with a corresponding production.

Given a grammar with G start symbol S, the **augmented grammar** G' adds:

- A new start symbol: S' (not in the original set of non-terminals),
- A new production: $\boldsymbol{S'} \rightarrow \boldsymbol{S}$

The aim of this new starting production is to allow the parser to recognize when the entire input has been successfully parsed.

6.2. LR(0) Items

An LR(0) item is a production rule with a dot (•) at some position in its right-hand side. It indicates how much of a production's right-hand side has been matched so far. Formally:

Example 1 : for a production S → ABC:

S → •ABC (nothing recognized yet)
S → A•BC (read A, yet to read BC)
S → AB•C (read B, yet to read C)
S → ABC• (read C, entire RHS matched, ready to reduce)

Note : The production $A \rightarrow \varepsilon$ generates only one item, $A \rightarrow \bullet$

6.3. Closure of Item Sets

The **Closure** operation expands a set of LR(0) items by adding all items that can be derived from **non-terminals** immediately following a dot (•). This ensures that all possible parsing paths are considered.

Let I be a set of items for a grammar G. $\boldsymbol{Closure(I)}$ is computed as follows:

1) Start with all items in I (initial set).
2) Repeat the following until no more items can be added:
   - For each item $[A \rightarrow \alpha \bullet B\, \beta]$ in the set, where B is a **non-terminal** immediately after the dot :
     - ✧ For each production $B \rightarrow \gamma$ in the grammar :
       - ➢ Add the item $[B \rightarrow \bullet\gamma]$ to the set I if it is not already present.

Example : Consider the augmented grammar:

S' → S
S → A
A → aA
A → b

Let's compute the closure for $\{S' \rightarrow \bullet S\}$

S' → •S contains •S, and S → A is the only rule for S ⟹ **add** [S → •A]

S → •A has •A, and A → aA, A → b are productions ⟹ **add** [A → •aA], [A → •b]

So we get: $I_0 = Closure(\{S' \rightarrow \bullet S\}) = \{ S' \rightarrow \bullet S, S \rightarrow \bullet A\, , A \rightarrow \bullet aA\, , A \rightarrow \bullet b\}$

6.4. The GOTO Function

The GOTO function is used to define the transitions in the LR(0) automaton for a grammar. The states of the automaton correspond to sets of items, and $\boldsymbol{GOTO(I, X)}$ specifies the transition from the state for I under symbol X (terminal or non-terminal) in the input.

Given a set of items I and a grammar symbol X, $\boldsymbol{GOTO(I, X)}$ is computed as:

1) Initialize an empty set J.
2) For each item $[A \rightarrow \alpha \bullet X\, \beta]$ in I, add $[A \rightarrow \alpha\, X \bullet \beta]$ to J (*i.e.*, move the dot over X).
3) Compute the closure of J: $Closure(J)$

6.5. Building the Canonical Collection of LR(0) Item Sets

The **canonical collection of LR(0) items** is the complete set of all valid LR(0) item sets (*i.e.*, parser states) that can be constructed from a given augmented grammar using the **Closure** and

GOTO operations. It forms the foundation of the LR(0) parsing automaton (DFA) and is essential for building the ACTION and GOTO parsing tables.

The canonical collection $C = \{I_0, I_1, I_2, \ldots, I_n\}$ is constructed using the following procedure:

1) Start with the **closure** of the augmented start production:
   - Begin from the item $S' \to \bullet S$ and compute its **closure**.
   - This becomes the **initial item set** I_0.
2) For each item set I and each grammar symbol X, compute:
   - $GOTO(I, X)$: advance the dot over symbol X and compute the closure.
   - If this results in a new set of items, add it to the collection as a new state.
3) Repeat until no new item sets can be generated.

Example : Consider the previous grammar :

S' → S
S → AA
A → aA
A → b

The initial state I_0 is obtained using the **closure** function.

$I_0 = Closure(\{S' \to \bullet S\}) = \{S' \to \bullet S, S \to \bullet AA, A \to \bullet aA, A \to \bullet b\}$

The remaining elements in set C are obtained by the GOTO function. We compute $GOTO(I, X)$ for every symbol X (terminal or non-terminal) that appears immediately after the dot in some item of the set I.

$I_1 = GOTO(I_0, S) = \{S' \to S\bullet\}$

$I_2 = GOTO(I_0, A) = \{S \to A\bullet A\} \cup \{A \to \bullet aA, A \to \bullet b\} = \{S \to A\bullet A, A \to \bullet aA, A \to \bullet b\}$

$I_3 = GOTO(I_0, a) = \{A \to a\bullet A\} \cup \{A \to \bullet aA, A \to \bullet b\} = \{A \to a\bullet A, A \to \bullet aA, A \to \bullet b\}$

$I_4 = GOTO(I_0, b) = \{A \to b\bullet\}$

I_1 and I_4 have only one final item, there is no need to apply the GOTO function.

Consider I_2 :

$I_5 = GOTO(I_2, A) = \{S \to AA\bullet\}$

$GOTO(I_2, a) = \{A \rightarrow a \bullet A\} \cup \{ A \rightarrow \ \bullet aA , A \rightarrow \ \bullet b\} = \{A \rightarrow a \bullet A, A \rightarrow \bullet aA , A \rightarrow \ \bullet b\} = I_3$

$GOTO(I_2, b) = \{A \rightarrow b \bullet\} = I_4$

Consider I_3 :

$I_6 = GOTO(I_3, A) = \{A \rightarrow aA \bullet\}$

$GOTO(I_3, a) = \{A \rightarrow a \bullet A\} \cup \{A \rightarrow a \bullet A, A \rightarrow \bullet aA, A \rightarrow \bullet b\} = \{A \rightarrow a \bullet A, A \rightarrow \bullet aA , A \rightarrow \bullet b\} = I_3$

$GOTO(I_3, b) = \{A \rightarrow b \bullet\} = I_4$

Therefore the canonical collection of LR(0) items for our grammar is $C = \{I_0, I_1, I_2, I_3, I_4, I_5, I_6\}$.
The following table summarizes the aforementioned results :

States			**Transitions**			
States	**Items**		**S**	**A**	**a**	**b**
I_0	$\{ S' \rightarrow \ \bullet S, S \rightarrow \ \bullet AA , A \rightarrow \ \bullet aA , A \rightarrow \ \bullet b\}$		I_1	I_2	I_3	I_4
I_1	$\{ S' \rightarrow S \bullet\}$					
I_2	$\{S \rightarrow A \bullet A, A \rightarrow \ \bullet aA , A \rightarrow \ \bullet b\}$			I_5	I_3	I_4
I_3	$\{A \rightarrow a \bullet A, A \rightarrow \ \bullet aA , A \rightarrow \ \bullet b\}$			I_6	I_3	I_4
I_4	$\{A \rightarrow b \bullet\}$					
I_5	$\{S \rightarrow AA \bullet\}$					
I_6	$\{A \rightarrow aA \bullet\}$					

6.6. Constructing the LR(0) DFA Using Item Sets

Once the canonical collection of LR(0) items is constructed using the Closure and GOTO operations, we can organize these sets of items into a **deterministic finite automaton (DFA)** that models the behavior of the parser. This DFA guides the parser through valid parsing decisions based on the input symbols and grammar structure.

In this DFA:

- **States** correspond to sets of LR(0) items (also called **configurations**).
- **Transitions** between states are labeled by grammar **symbols** (terminals and non-terminals).
- **The initial state** is the closure of the item containing the augmented start production with the dot at the beginning ($[S' \ \rightarrow \ \bullet S]$).
- For each state (set of items), and each grammar symbol X that appears immediately after a dot in one or more items of that state, we compute a transition using the GOTO function. $GOTO(I, X)$ defines the next state by advancing the dot over X and applying closure to the result.

The resulting DFA represents all valid **configurations** of the parser as it processes an input string. Each path through the DFA corresponds to a possible derivation of a string in the language. This DFA serves as the foundation for constructing the ACTION and GOTO parsing tables used by the LR(0) parser.

Example : The DFA of the previous grammar is illustrated in Figure 4.

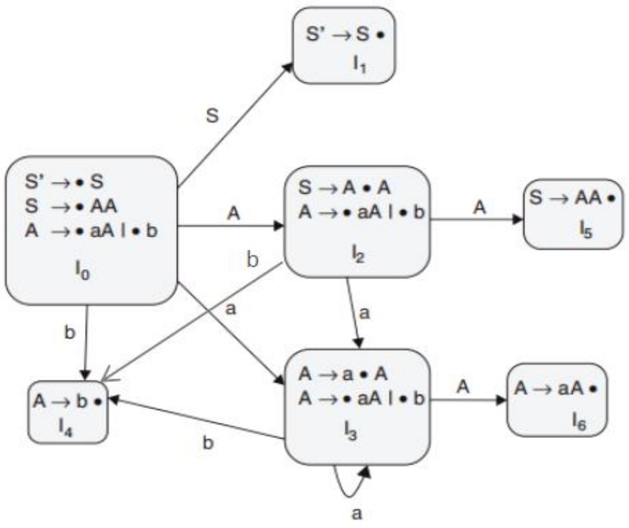


Figure 4. DFA for the grammar in Example.

6.7. Constructing Parsing Tables

Once the canonical collection of LR(0) items (or equivalently, the DFA of LR(0) item sets) is constructed, the next step is to build the parsing tables used by the LR(0) parser: **the ACTION table and the GOTO table**.

The ACTION table has dimensions of $\boldsymbol{states} \times \boldsymbol{terminals}$, including end-marker \$. Each entry contains one of four possible actions: **shift**, **reduce**, **accept**, and **empty cells** represent **syntax errors**. The GOTO table has dimensions of $\boldsymbol{states} \times \boldsymbol{non-terminals}$. Each entry either contains a **state number** indicating the next state to transition to after reducing to that non-terminal, or remains **empty** to indicate no valid transition exists.

❖ ***ACTION table construction rules***

For each state I_i in the **canonical collection of LR(0) item sets** :

➢ **Shift entries** : If $A \to \alpha{\bullet}a\beta$ is in I_i where a is a terminal, and $GOTO(I_i, a) = I_j$, then $\boldsymbol{ACTION[i, a] = "shift\, j".}$

This means: on seeing terminal a, the parser shifts and push state $\boldsymbol{j}$**.**

➢ **Reduce entries** : If $A \to \alpha\bullet$ is in I_i, and $A \neq S'$, then $\boldsymbol{ACTION[i, a]} = \boldsymbol{"reduce\, k"}$ for every terminals a (including \$), where k is the number of the production $\boldsymbol{A} \rightarrow \boldsymbol{\alpha}$ in the numbered grammar.

- **Accept entry** : If $S' \rightarrow S\bullet$ is in I_i, then $\boldsymbol{ACTION}[\boldsymbol{i}, \$] = "\boldsymbol{accept}"$

Note: Since LR(0) has no lookahead, reductions are applied on all terminals, which can lead to conflicts in ambiguous grammars.

❖ ***GOTO table construction rule***

For each non-terminal A: If $GOTO(I_i, A) = I_j$, then: $\boldsymbol{GOTO}[\boldsymbol{i}, \boldsymbol{A}] = \boldsymbol{j}$ (go to state j)

This tells the parser to transition to state $\boldsymbol{j}$ after reducing to non-terminal A.

Example : Given the previous grammar numbered :

(1) S $\rightarrow$ AA
(2) A $\rightarrow$ aA
(3) A $\rightarrow$ b

The following combined table represent its ACTION and GOTO tables.

The codes for the actions are :

- S_i means **shift** and stack state i
- R_k means **reduce** by the production numbered k
- acc means **accept**
- Blank means **error**

	ACTION				**GOTO**	
States	**a**	**b**	**\$**		**S**	**A**
0	S3	S4			1	2
1			**acc**			
2	S3	S4				5
3	S3	S4				6
4	R3	R3	R3			
5	R1	R1	R1			
6	R2	R2	R2			

Using the **LR parsing stack-based algorithm** presented in *section 5.2*, Let's parse the input string "aabb" :

Stack	**Input**	**Action**
\$	aabb\$	**Push state 0**

$0	aabb$	**S$_3$**: shift ‘a’ then state 3 and increment
$0a3	abb$	**S$_3$**: shift ‘a’ then state 3 and increment
$0a3a3	bb$	**S$_4$**: shift ‘b’ then state 4 and increment
$0a3a3b4	b$	**R$_3$**: reduce by A → b : pop 2 symbols, replace by A, push GOTO[3, A]=6
$0a3a3A6	b$	**R$_2$**: reduce by A → aA : pop 4 symbols, replace by A, push GOTO[3, A]=6
$0a3A6	b$	**R$_2$**: reduce by A → aA : pop 4 symbols, replace by A, push GOTO[0, A]=2
$0A2	b$	**S$_4$**: shift ‘b’ then state 4 and increment
$0A2b4	$	**R$_3$**: reduce by A → b : pop 2 symbols, replace by A, push GOTO[2, A]=5
$0A2A5	$	**R$_1$**: reduce by S → AA : pop 4 symbols, replace by S, push GOTO[0, S]=1
$0S1	$	**Accept**

Note : ACTION and GOTO tables can be extracted directly from the DFA of LR(0) items by following this procedure :

From each DFA **state** I_i:

- For each transition $GOTO(I_i, X) = I_j$:
  - If X is a **terminal**, set $ACTION[i, X] = "shift\ j"$
  - If X is a **non-terminal**, set $GOTO[i, X] = j$
- If I_i contains $A \rightarrow \alpha\bullet$ (dot at end and $A \neq S'$):
  - For all **terminals** a, set $ACTION[i, a] = "reduce\ by\ A \rightarrow \alpha"$ (or $reduce\ k$, where k is the number of production $A \rightarrow \alpha$ in the numbered grammar)
- If I_i contains $S' \rightarrow S\bullet$, set $ACTION[i, \$] = "accept"$

7. SLR(1) Parsing

SLR(1) (Simple LR with 1-token lookahead) parsing is an enhancement of LR(0) parsing. It uses the same set of LR(0) items to construct an automaton of LR(0) items but improves decision-making during **reductions** by using **FOLLOW sets** of non-terminals to restrict reduce actions, which avoids incorrect reductions and helps resolve some **shift/reduce and reduce/reduce parsing conflicts** that LR(0) cannot handle.

Note : A grammar is said to be SLR(1) if it can be parsed by an SLR(1) parser.

7.1. SLR(1) Table Construction Rules

SLR(1) follows the same parser construction procedure as LR(0), but differs by considering

FOLLOW sets to restrict reduce entries in the ACTION table to specific terminals.

1) For each state I_i in the **canonical collection of LR(0) item sets** :

➢ **Shift entries** : If $A \rightarrow \alpha \bullet a\beta$ is in I_i where a is a terminal, and $GOTO(I_i, a) = I_j$, then $\boldsymbol{ACTION[i, a] = "shift\, j".}$

➢ **Reduce entries** : **If** $A \rightarrow \alpha\bullet$ is in I_i, and $A \neq S'$, then :

For each terminal $\boldsymbol{a \in FOLLOW(A)}$:

$\boldsymbol{ACTION[i, a]} = \boldsymbol{"reduce\ k"}$, where k is the number of the production $\boldsymbol{A \rightarrow \alpha}$ in the numbered grammar.

➢ **Accept entry** : If $S' \rightarrow S\bullet$ is in I_i, then $\boldsymbol{ACTION[i, \$] = "accept"}$

2) For each non-terminal A: If $GOTO(I_i, A) = I_j$, then: $\boldsymbol{GOTO[i, A] = j}$ (go to state j)

Example 1: Let's construct LR(0) and SLR(1) parsing tables for the augmented grammar :

E' → E
E → E + T | T
T → T * F | F
F → (E) | id

$I_0 = Closure(\{E' \rightarrow \bullet E\})$

$I_0 = \{ E' \rightarrow \bullet E, E \rightarrow \bullet E + T , E \rightarrow \bullet T, T \rightarrow \bullet T * F, T \rightarrow \bullet F, F \rightarrow \bullet(E), F \rightarrow \bullet id\}$

Below is the **canonical collection of LR(0) item sets** :

$I_1 = GOTO(I_0, E) = \{$
$E' \rightarrow E\bullet$
$E \rightarrow E\bullet + T$
$\}$

$I_2 = GOTO(I_0, T) = \{$
$E \rightarrow T\bullet$
$T \rightarrow T \bullet * F$
$\}$

$I_3 = GOTO(I_0, F) = \{$
$T \rightarrow F\bullet$
$\}$

$I_4 = GOTO(I_0, () = \{$
$E' \rightarrow (\bullet E)$
$E \rightarrow \bullet E + T$
$E \rightarrow \bullet T$
$T \rightarrow \bullet T * F$
$T \rightarrow \bullet F$
$F \rightarrow \bullet(E)$
$F \rightarrow \bullet id$
$\}$

$I_5 = GOTO(I_0, id) = \{$
$F \rightarrow id\bullet$
$\}$

$I_6 = GOTO(I_1, +) = \{$
$E \rightarrow E + \bullet T$
$T \rightarrow \bullet T * F$
$T \rightarrow \bullet F$
$F \rightarrow \bullet(E)$
$F \rightarrow \bullet id$
$\}$

$I_7 = GOTO(I_2, *) = \{$
$T \rightarrow T * \bullet F$
$F \rightarrow \bullet(E)$
$F \rightarrow \bullet id$
$\}$

$I_8 = GOTO(I_4, E) = \{$
$F \rightarrow (E\bullet)$
$E \rightarrow E\bullet + T$
$\}$

$GOTO(I_4, () = \{$
$F \rightarrow (\bullet E)$
$E \rightarrow \bullet E + T$
$E \rightarrow \bullet T$

$GOTO(I_4, T) = \{$
$E \rightarrow T\bullet$
$T \rightarrow T \bullet * F$
$\} = I_2$

$GOTO(I_4, id) = \{$
$F \rightarrow id\bullet$
$\} = I_5$

$T \rightarrow \bullet T * F$
$T \rightarrow \bullet F$
$F \rightarrow \bullet(E)$
$F \rightarrow \bullet id$
$\} = I_4$

$I_9 = GOTO(I_6, T) = \{$
$E \rightarrow\ E + T\bullet$
$T \rightarrow T\bullet * F$
$\}$

$GOTO(I_6, F) = \{$
$E \rightarrow F\bullet$
$\} =\ I_3$

$GOTO(I_6, id) = \{$
$F \rightarrow\ id\bullet$
$\} =\ I_5$

$GOTO(I_6, () = \{$
$E' \rightarrow\ (\bullet E)$
$E \rightarrow \bullet E + T$
$E \rightarrow \bullet T$
$T \rightarrow \bullet T * F$
$T \rightarrow \bullet F$
$F \rightarrow \bullet(E)$
$F \rightarrow \bullet id$
$\} = I_4$

$GOTO(I_4, F) = \{$
$E \rightarrow F\bullet$
$\} =\ I_3$

$I_{10} = GOTO(I_7, F) = \{$
$T \rightarrow T\ * F\bullet$
$\}$

$GOTO(I_7, (\,) = \{$
$F \rightarrow\ (\bullet E)$
$E \rightarrow \bullet E + T$
$E \rightarrow \bullet T$
$T \rightarrow \bullet T * F$
$T \rightarrow \bullet F$
$F \rightarrow \bullet(E)$
$F \rightarrow \bullet id$
$\} = I_4$

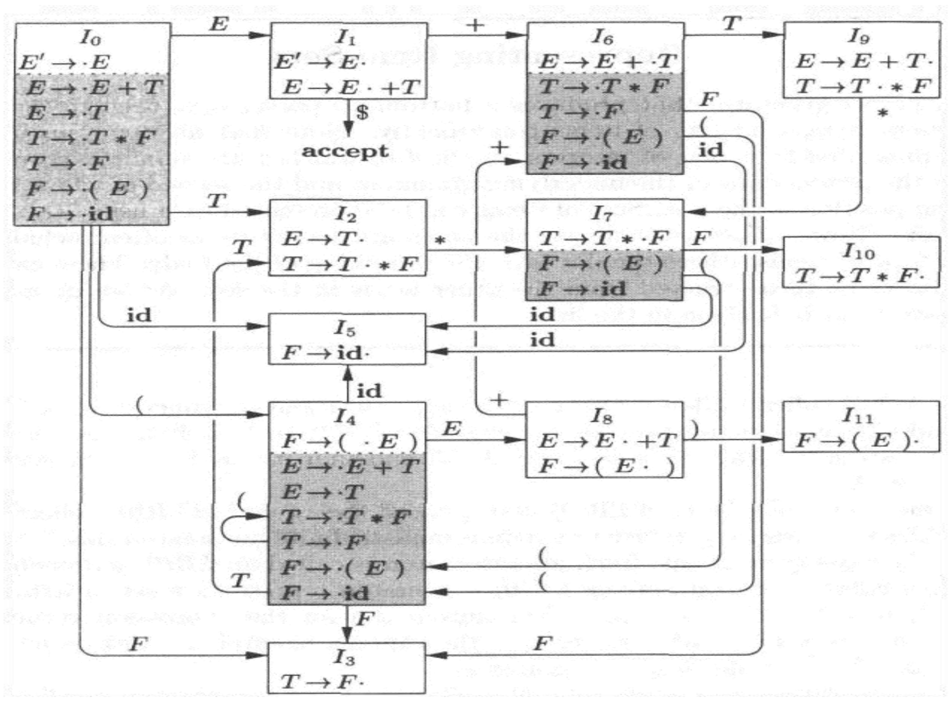
$GOTO(I_7, id) = \{$
$F \rightarrow\ id\bullet$
$\} =\ I_5$

$I_{11} = GOTO(I_8,)) = \{$
$F \rightarrow (E)\bullet$
$\}$

$GOTO(I_8,\ +) = \{$
$E \rightarrow E + \bullet T$
$T \rightarrow \bullet T * F$
$T \rightarrow \bullet F$
$F \rightarrow \bullet(E)$
$F \rightarrow\ \bullet id$
$\} = I_6$

$GOTO(I_9,\ *) = \{$
$T \rightarrow T\ * \bullet F$
$F \rightarrow \bullet(E)$
$F \rightarrow\ \bullet id$
$\} = I_7$

The LR(0) DFA is illustrated in Figure 5.



I_0: $E' \rightarrow \cdot E$, $E \rightarrow \cdot E + T$, $E \rightarrow \cdot T$, $T \rightarrow \cdot T * F$, $T \rightarrow \cdot F$, $F \rightarrow \cdot (E)$, $F \rightarrow \cdot \mathbf{id}$
I_1: $E' \rightarrow E \cdot$, $E \rightarrow E \cdot + T$
$
accept
I_2: $E \rightarrow T \cdot$, $T \rightarrow T \cdot * F$
I_3: $T \rightarrow F \cdot$
I_4: $F \rightarrow (\cdot E)$, $E \rightarrow \cdot E + T$, $E \rightarrow \cdot T$, $T \rightarrow \cdot T * F$, $T \rightarrow \cdot F$, $F \rightarrow \cdot (E)$, $F \rightarrow \cdot \mathbf{id}$
I_5: $F \rightarrow \mathbf{id} \cdot$
I_6: $E \rightarrow E + \cdot T$, $T \rightarrow \cdot T * F$, $T \rightarrow \cdot F$, $F \rightarrow \cdot (E)$, $F \rightarrow \cdot \mathbf{id}$
I_7: $T \rightarrow T * \cdot F$, $F \rightarrow \cdot (E)$, $F \rightarrow \cdot \mathbf{id}$
I_8: $E \rightarrow E \cdot + T$, $F \rightarrow (E \cdot)$
I_9: $E \rightarrow E + T \cdot$, $T \rightarrow T \cdot * F$
I_{10}: $T \rightarrow T * F \cdot$
I_{11}: $F \rightarrow (E) \cdot$
E
T
F
id
(
)
+
*

Figure 5. DFA

Now, we construct both the LR(0) and SLR(1) parsing tables for the expression grammar. Let us begin by numbering the productions.

(1) E → E + T

(2) E → T
(3) T → T * F
(4) T → F
(5) F → (E)
(6) F → id

Table : LR(0) Parsing Table for Grammar Example

	ACTION							GOTO		
States	**id**	+	*	**(**	**)**	**$**		**E**	**T**	**F**
0	S5			S4				1	2	3
1		S6				**acc**				
2	R2	R2	R2/S7	R2	R2	R2				
3	R4	R4	R4	R4	R4	R4				
4	S5			S4				8	2	3
5	R6	R6	R6	R6	R6	R6				
6	S5			S4					9	3
7	S5			S4						10
8		S6			S11					
9	R1	R1	R1/S7	R1	R1	R1				
10	R3	R3	R3	R3	R3	R3				
11	R5	R5	R5	R5	R5	R5				

In the LR(0) parsing table, we observe a shift/reduce conflict in states 2 and 9 when the lookahead symbol is *. This conflict arises because LR(0) does not consider any lookahead information when making reduction decisions.

In contrast, the SLR(1) parser resolves this conflict by allowing reductions only when the current input symbol belongs to the FOLLOW set of the non-terminal being reduced.

Based on FOLLOW sets, we construct the SLR(1) parsing table, which is illustrated below.

We observe that the table contains no conflicts, confirming that the grammar is SLR(1)-compatible.

Follow (E') = {$}

Follow (E) = {+,), $}

Follow (T) = {+, *,), $}

Follow (F) = {+, *,), $}

Table : SLR(1) Parsing Table for Grammar Example

	ACTION							GOTO		
States	**id**	+	*	**(**	**)**	**$**		**E**	**T**	**F**
0	S5			S4				1	2	3
1		S6				**acc**				
2		R2	S7		R2	R2				

3		R4	R4		R4	R4				
4	S5			S4				8	2	3
5		R6	R6		R6	R6				
6	S5			S4					9	3
7	S5			S4						10
8		S6			S11					
9		R1	S7		R1	R1				
10		R3	R3		R3	R3				
11		R5	R5		R5	R5				

Using the **LR parsing stack-based algorithm** presented in *section 5.2*, Let's parse the input string "id*id+id" :

Stack	**Input**	**Action**
$	id*id+id$	**Push state 0**
$0	id*id+id$	**S_5**: shift 'id' then state 5 and increment
$0id5	*id+id$	**R_6**: reduce by F → id: pop 2 symbols, replace by F, push GOTO[0, F]=3
$0F3	*id+id$	**R_4**: reduce by T → F : pop 2 symbols, replace by T, push GOTO[0, T]=2
$0T2	*id+id$	**S_7**: shift '*' then state 7 and increment
$0T2*7	id+id$	**S_5**: shift 'id' then state 5 and increment
$0T2*7id5	+id$	**R_6**: reduce by F → id : pop 2 symbols, replace by F, push GOTO[7, F]=10
$0T2*7F10	+id$	**R_3**: reduce by T → T * F: pop 6 symbols, replace by T, push GOTO[0, T]=2
$0T2	+id$	**R_2**: reduce by E → T : pop 2 symbols, replace by E, push GOTO[0, E]=1
$0E1	+id$	**S_6**: shift '+' then state 6 and increment
$0E1+6	id$	**S_5**: shift 'id' then state 5 and increment
$0E1+6id5	$	**R_6**: reduce by F → id : pop 2 symbols, replace by F, push GOTO[6, F]=3
$0E1+6F3	$	**R_4**: reduce by T → F : pop 2 symbols, replace by T, push GOTO[6, T]=9
$0E1+6T9	$	**R_1**: reduce by E → E+T : pop 6 symbols, replace by E, push GOTO[0, E]=1
$0E1	$	**Accept**

8. CLR(1)/ LR(1) Parsing

CLR(1) parsing is a sophisticated bottom-up parsing technique that extends LR(0) by incorporating precise lookahead information directly into the parser states. **CLR** stands for **"Canonical LR"** and represents the full, canonical implementation of LR(1) parsing.

CLR(1) parsing follows the same general procedure as LR(0) and SLR(1) for constructing the parsing table; however, it uses **LR(1) items** instead of LR(0) items, which leads to modifications in the Closure and GOTO operations, and consequently results in a more precise parsing table.

Note : A grammar is LR(1) (or Canonical LR(1)) if it can be parsed by a canonical LR parser.

8.1. LR(1) Items

An LR(1) item is a production of the grammar augmented with:

- ✓ A **dot** (•) indicating how much of the production has been recognized
- ✓ A **lookahead symbol**, which is a **terminal** that can legally follow the non-terminal being reduced.

Each item has the form: $[A \rightarrow \alpha \bullet \beta, a]$. It means: parsed α, expecting β next in the input, and reducing by $A \rightarrow \alpha\beta$ only if the next input symbol is a.

Example: Consider the first production in any augmented grammar : S' → S

An LR(1) item could be: [S' → •S, $]. This means we're just starting to parse S, and we expect the end of input ($) after S.

8.2. Closure of LR(1) Items

Given a set of LR(1) items I, the $Closure(I)$ is constructed by:

1) Initially placing all items in I into the closure set.

2) For each item $[A \rightarrow \alpha \bullet B\beta, a]$ in the set:

- For each production $B \rightarrow \gamma$ in the grammar
  - ◆ For each terminal b in $FIRST(\beta a)$:
    - ➢ Add $[B \rightarrow \bullet\gamma, b]$ to the set if it's not already included.

Note: $FIRST(\beta a)$ means we compute $FIRST$ of the string β followed by a. If β is **not nullable**: $FIRST(\beta a) = FIRST(\beta)$. If β is **nullable**: $FIRST(\beta a) = FIRST(\beta) \cup FIRST(a)$

Example: Consider the augmented grammar :

S' → S
S → CC
C → cC
C → d

Let's compute $Closure\ \{[S' \rightarrow \bullet S, \$]\}$

Since the **dot** is before S, and S → CC, we add: [S → •CC, $]

Now dot is before C, and C → cC | d. So we need to add :

[C → • cC, FIRST(C$)]; FIRST(C$) = FIRST(C)={c, d}

[C → • d, FIRST(C$)]; FIRST(C$) = FIRST(C)={c, d}

Thus:

$Closure[S' \to \bullet S, \$]$

$= \{[S' \to \bullet S, \$], [S \to \bullet CC, \$], [C \to \bullet cC, c], [C \to \bullet cC, d], [C \to \bullet d, c], [C \to \bullet d, d]\}$

8.3. The GOTO Function for LR(1) Items

While the structure of the GOTO function is similar to that in LR(0), the presence of lookahead symbols in LR(1) requires careful handling during the closure step, where lookaheads can propagate and vary.

Given a set of LR(1) items I and a grammar symbol X, $GOTO(I, X)$ constructs the next item set reachable by reading X. Formally:

If I contains items of the form $[A \to \alpha \bullet X\beta, a]$, then $GOTO(I, X) = Closure(\{ [A \to \alpha X \bullet \beta, a] \})$

The process involves two main steps:

- **Transition**: Move the dot over X in all applicable items. The lookahead symbol remains unchanged.
- **Closure**: Compute the closure of the resulting items. Here, new items may have different lookaheads, based on $FIRST(\beta a)$.

8.4. Canonical Collection of LR(1) Item Sets (C)

The procedure for constructing the **canonical collection of LR(1) items** closely mirrors that of LR(0). It follows the same overall steps: begin with the closure of the initial item set, then apply the GOTO function repeatedly to discover all reachable sets of items. The key difference lies in the nature of the items themselves — LR(1) items carry lookahead symbols, which are taken into account when applying the Closure and GOTO operations.

Example: Consider the augmented grammar :

S' → S
S → CC
C → cC
C → d

The initial state I_0 is obtained using the **closure** function.

$I_0 = Closure(\{[S' \rightarrow \bullet S, \$]\})$

$= \{[S' \rightarrow \bullet S, \$], [S \rightarrow \bullet CC, \$], [C \rightarrow \bullet cC, c], [C \rightarrow \bullet cC, d], [C \rightarrow \bullet d, c], [C \rightarrow \bullet d, d]\}$

The remaining elements in set C are obtained by the GOTO function.

$I_1 = GOTO(I_0, S) = \{ [S' \rightarrow S\bullet, \$]\}$

$I_2 = GOTO(I_0, C) = \{[S \rightarrow C\bullet C, \$]\} \cup \{[C \rightarrow \bullet cC, \$] , [C \rightarrow \bullet d, \$]\}$

$= \{[S \rightarrow C\bullet C, \$], [C \rightarrow \bullet cC, \$] , [C \rightarrow \bullet d, \$]\}$

$I_3 = GOTO(I_0, c) = \{[C \rightarrow c\bullet C, c], [C \rightarrow c\bullet C, d]\} \cup \{[C \rightarrow \bullet cC, c/d] , [C \rightarrow \bullet d, c/d]\}$

$= \{[C \rightarrow c\bullet C, c], [C \rightarrow c\bullet C, d], [C \rightarrow \bullet cC, c] , [C \rightarrow \bullet cC, d] , [C \rightarrow \bullet d, c], [C \rightarrow \bullet d, d]\}$

$I_4 = GOTO(I_0, d) = \{[C \rightarrow d\bullet, c], [C \rightarrow d\bullet, d]\}$

Consider I_2 :

$I_5 = GOTO(I_2, C) = \{[S \rightarrow CC\bullet, \$]\}$

$I_6 = GOTO(I_2, c) = \{[C \rightarrow c\bullet C, \$]\} \cup \{ [C \rightarrow \bullet cC, \$] , [C \rightarrow \bullet d, \$]\}$

$= \{[C \rightarrow c\bullet C, \$], [C \rightarrow \bullet cC, \$] , [C \rightarrow \bullet d, \$]\}$

$I_7 = GOTO(I_2, d) = \{[C \rightarrow d\bullet, \$]\}$

Consider I_3 :

$I_8 = GOTO(I_3, C) = \{[C \rightarrow cC\bullet, c], [C \rightarrow cC\bullet, d]\}$

$GOTO(I_3, c) = \{[C \rightarrow c\bullet C, c], [C \rightarrow c\bullet C, d], [C \rightarrow \bullet cC, c] , [C \rightarrow \bullet cC, d] , [C \rightarrow \bullet d, c], [C \rightarrow \bullet d, d]\}$

$= I_3$

$GOTO(I_3, d) = \{[C \rightarrow d\bullet, c], [C \rightarrow d\bullet, d]\} = I_4$

Consider I_6 :

$I_9 = GOTO(I_6, C) = \{[C \rightarrow cC\bullet, \$]\}$

$GOTO(I_6, c) = \{[C \rightarrow c\bullet C, \$], [C \rightarrow \bullet cC, \$] , [C \rightarrow \bullet d, \$]\} = I_6$

$GOTO(I_6, d) = \{[C \rightarrow d\bullet, \$]\} = I_7$

The DFA of LR(1) item sets is shown in Figure 6.

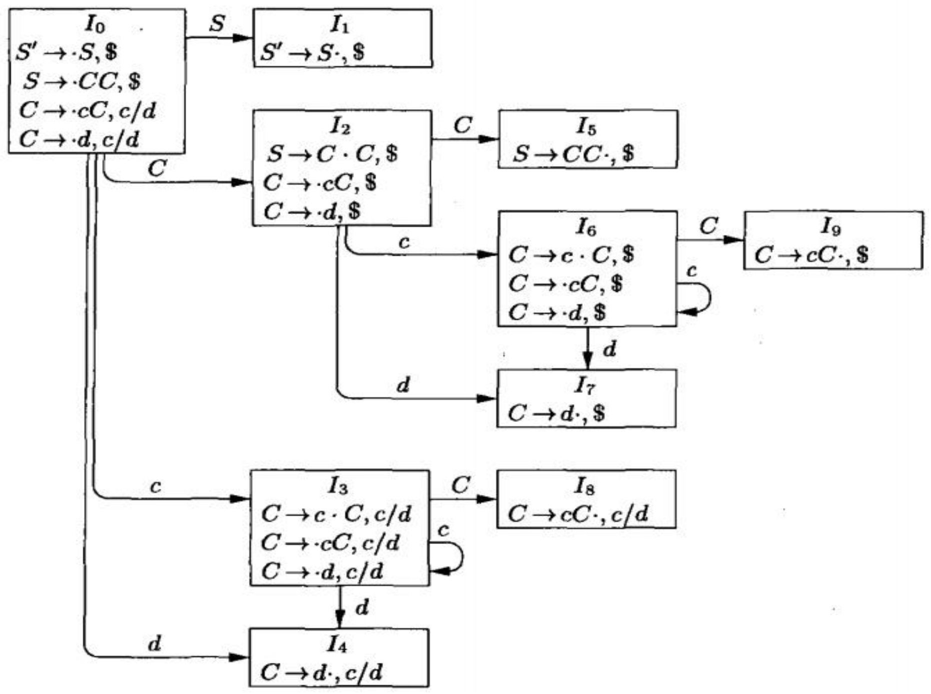


Figure 6. DFA

8.5. CLR(1) Table Construction Rules

1) For each state I_i in the **canonical collection of LR(1) item sets** :

- **Shift entries** : If $[A \rightarrow \alpha \bullet a\beta, b]$ is in I_i where a is a **terminal**, and $GOTO(I_i, a) = I_j$, then $\boldsymbol{ACTION[i, a] = "shift\,j"}$.
- **Reduce entries** : If $[A \rightarrow \alpha\bullet, a]$ is in I_i, and $A \neq S'$, then : $\boldsymbol{ACTION[i, a] = "reduce\ k"}$, where k is the number of the production $\boldsymbol{A \rightarrow \alpha}$ in the numbered grammar.
- **Accept entry** : If $[S' \rightarrow S\bullet, \$]$ is in I_i, then $\boldsymbol{ACTION[i, \$] = "accept"}$

2) For each non-terminal A: If $GOTO(I_i, A) = I_j$, then: $\boldsymbol{GOTO[i, A] = j}$

Example : Let's construct LR(1) parsing table for the grammar :

(1) S → CC
(2) C → cC
(3) C → d

Table : CLR(1) Parsing Table for Example

	ACTION				**GOTO**	
States	**c**	**d**	**$**		**S**	**C**
0	S3	S4			1	2
1			**acc**			
2	S6	S7				5
3	S3	S4				8

4	R3	R3				
5			R1			
6	S6	S7				9
7			R3			
8	R2	R2				
9			R2			

9. LALR Parsing

LALR(1) stands for **Look-Ahead LR(1)**. It is a type of LR parser that combines the power of CLR(1) with the table size efficiency of SLR(1). The key idea is to merge LR(1) item sets that have the same **LR(0) core**, thereby reducing the number of parser states and producing a smaller parsing table.

It is widely used in practice due to this balance between precision and efficiency. Tools like Yacc, Bison, and many compiler generators produce LALR(1) parsers.

Notes :

- A grammar is LALR(1) if it can be parsed by an LALR(1) parser.
- LR(0) ⊂ SLR(1) ⊂ LALR(1) ⊂ LR(1)/CLR(1).

Example : consider the canonical collection of LR(1) item sets of the previous grammar. Three pairs of item sets can be merged I_3 and I_6, I_4 and I_7, I_8 and I_9

$I_3 = \{[C \rightarrow c\bullet C, c], [C \rightarrow c\bullet C, d], [C \rightarrow \bullet cC, c], [C \rightarrow \bullet cC, d], [C \rightarrow \bullet d, c], [C \rightarrow \bullet d, d]\}$

$I_6 = \{[C \rightarrow c\bullet C, \$], [C \rightarrow \bullet cC, \$], [C \rightarrow \bullet d, \$]\}$

$$\boxed{\rightarrow \boldsymbol{I_{36}} = \{[C \rightarrow c\bullet C, c|d|\$], [C \rightarrow \bullet cC, c|d|\$], [C \rightarrow \bullet d, c|d|\$]\}}$$

$I_4 = \{[C \rightarrow d\bullet, c], [C \rightarrow d\bullet, d]\}$

$I_7 = \{[C \rightarrow d\bullet, \$]\}$

$$\boxed{\rightarrow \boldsymbol{I_{47}} = \{[C \rightarrow d\bullet, c|d|\$]\}}$$

$I_8 = \{[C \rightarrow cC\bullet, c], [C \rightarrow cC\bullet, d]\}$

$I_9 = \{[C \rightarrow cC\bullet, \$]\}$

$$\boxed{\rightarrow \boldsymbol{I_{89}} = \{[C \rightarrow cC\bullet, c|d|\$]\}}$$

After merging item sets with the same LR(0) core, the ACTION table may include additional reduce actions (one for each lookahead symbol now present), and the GOTO table reflects transitions from the merged state for each possible symbol after the dot, preserving the correct shift or goto destinations as in the original unmerged sets.

The LALR parsing table for the above grammar is shown in Table X.

Table : LALR(1) Parsing Table for Example

	ACTION				**GOTO**	
States	**c**	**d**	**$**		**S**	**C**
0	S36	S47			1	2
1			**acc**			
2	S36	S47				5
36	S36	S47				89
47	R3	R3	R3			
5			R1			
89	R2	R2	R2			