THEORY OF COMPUTATION AND COMPILERS

Unit - II

CONTEXT FREE GRAMMARS AND PARSING

- Introduction
- Context-Free Grammars - Derivation, Parse trees, Ambiguity
- Types of Parsers
- LL(K) grammars and LL(1) parsing
- Bottom-up Parsing - handle pruning
- LR Grammar Parsing
- **LALR parsing** 
- Parsing ambiguous grammars
- Error Recovery in Parsing
- YACC programming specification

Dr. R. Madana Mohana
Professor, Artificial Intelligence & Data Science | I/c-Head, Artificial Intelligence & Machine Learning
CHAITANYA BHARATHI INSTITUTE OF TECHNOLOGY
Hyderabad - 500 075, Telangana, INDIA
www.cbit.ac.in

Unit-II: Syntax Analysis (or) Parser
Constructing LALR Parsing Table

Outline:

- LALR-Parsing Table
- Algorithm for Constructing LALR-Parsing Table
- Example problem
- Compaction of LR Parsing Tables

LALR Parser

LALR parser is a **Look Ahead LR parser**. This method is commonly used in practice, because the tables obtained by these parsers are considerably smaller than the **CLR parsers** and at the same time most common syntactic constructs of programming languages can be expressed conveniently by an **LALR grammar**.

LALR Parser

Observe the following points:

- **SLR** and **LALR** tables for a grammar always have the same number of states.
- **CLR** parsing table will have more number of states when compared to **SLR** and **LALR**.
- It is easier and more economical to construct **SLR** or **LALR** parsers when compared to **CLR** parsers.

Algorithm for Constructing LALR-Parsing Table

The algorithm to construct **LALR** parsing table is shown below:

Algorithm: **LALR_PARSING_TABLE (G′)**

INPUT: An augmented grammar **G′**.

OUTPUT: The **LALR parsing table** consisting of **ACTION** and **GOTO**.

Algorithm for Constructing LALR-Parsing Table

METHOD: The procedure is shown below:

1. Obtain **LR(1) items** for the **augmented grammar G′** and store in **C**.
2. Group the items consisting of *common cores* and let the resulting items be **C′ = {I_0', I_1', I_2', …, I_n'}**. The states ***i*** of the parser are obtained form **I_i'**. The parsing **ACTION** values and **GOTO** values are computed in the same manner as is done for **CLR(1)** parser. The table using this algorithm is called **LALR** parsing table.

Algorithm for Constructing LALR-Parsing Table

METHOD:

2. a) If **[A → α.aβ, b]∈I_i** and
GOTO (I_i, a)= I_j then
ACTION [i, a] = "shift j" where **'a'** must be a terminal.

   b) If **[A → α., a]∈I_i**, then
ACTION [i, a] = "reduce A → α".

Algorithm for Constructing LALR-Parsing Table

METHOD:

2. c) If **[S′ → S., $]∈$I_i$**, then **ACTION [i, $] = accept**.

3. For each state **i**, **GOTO** transitions can be obtained as shown below:

   a) if **GOTO (I_i, A)= I_j** then **GOTO [i, A] = j**

   end if

   a) If **I_i** contains **[S′ → S., $]** then **'i'** is the initial or start state of the parser.

Algorithm for Constructing LALR-Parsing Table

Note: In **LALR parsing table**, if there are multiple entries in **ACTION**, then the grammar is not **LALR(1)** grammar.

The **parsing table** obtained using this algorithm is **LALR parsing table** and the **parser** which uses this table is called **LALR parser** and the **grammar** is said to be **LALR(1)**.

Constructing LALR-Parsing Table

Example:

Construct **LALR parsing table** for the given grammar **G** as shown below:

1. **S → CC**
2. **C → cC**
3. **C → d**

Constructing LALR-Parsing Table

Example: ***Solution***

Augmented Grammar **G′**:

1. **S′ → S**
2. **S → CC**
3. **C → cC**
4. **C → d**

Constructing LALR-Parsing Table

Example: ***Solution***

Step 1: The **LR(1)** items can be obtained as shown below:

I_0:
S′ → .S, $
S → .CC, $
C → .cC, c | d
C → .d, c | d

I_1: GOTO (I_0, S)
S′ → S., $
I_2: GOTO (I_0, C)
S → C.C, $
C → .cC, $
C → .d, $

Constructing LALR-Parsing Table

Example: ***Solution***

I_3: GOTO (I_0, c)
C → c.C, c | d
C → .cC, c | d
C → .d, c | d

I_4: GOTO (I_0, d)
C → d., c | d

I_5: GOTO (I_2, C)
S → CC., $

I_6: GOTO (I_2, c)
C → c.C, $
C → .cC, $
C → .d, $

I_7: GOTO (I_2, d)
C → d., $

I_8: GOTO (I_3, C)
C → cC., c | d

Constructing LALR-Parsing Table

Example: ***Solution***

GOTO (I_3, c) = I_3

C → c.C, c | d

C → .cC, c | d

C → .d, c | d

GOTO (I_3, d) = I_4

C → d., c | d

I_9: GOTO (I_6, C)

C → cC., $

GOTO (I_6, c) = I_6

C → c.C, $

C → .cC, $

C → .d, $

GOTO (I_6, d) = I_7

C → d., $

DFA for LR(1) items

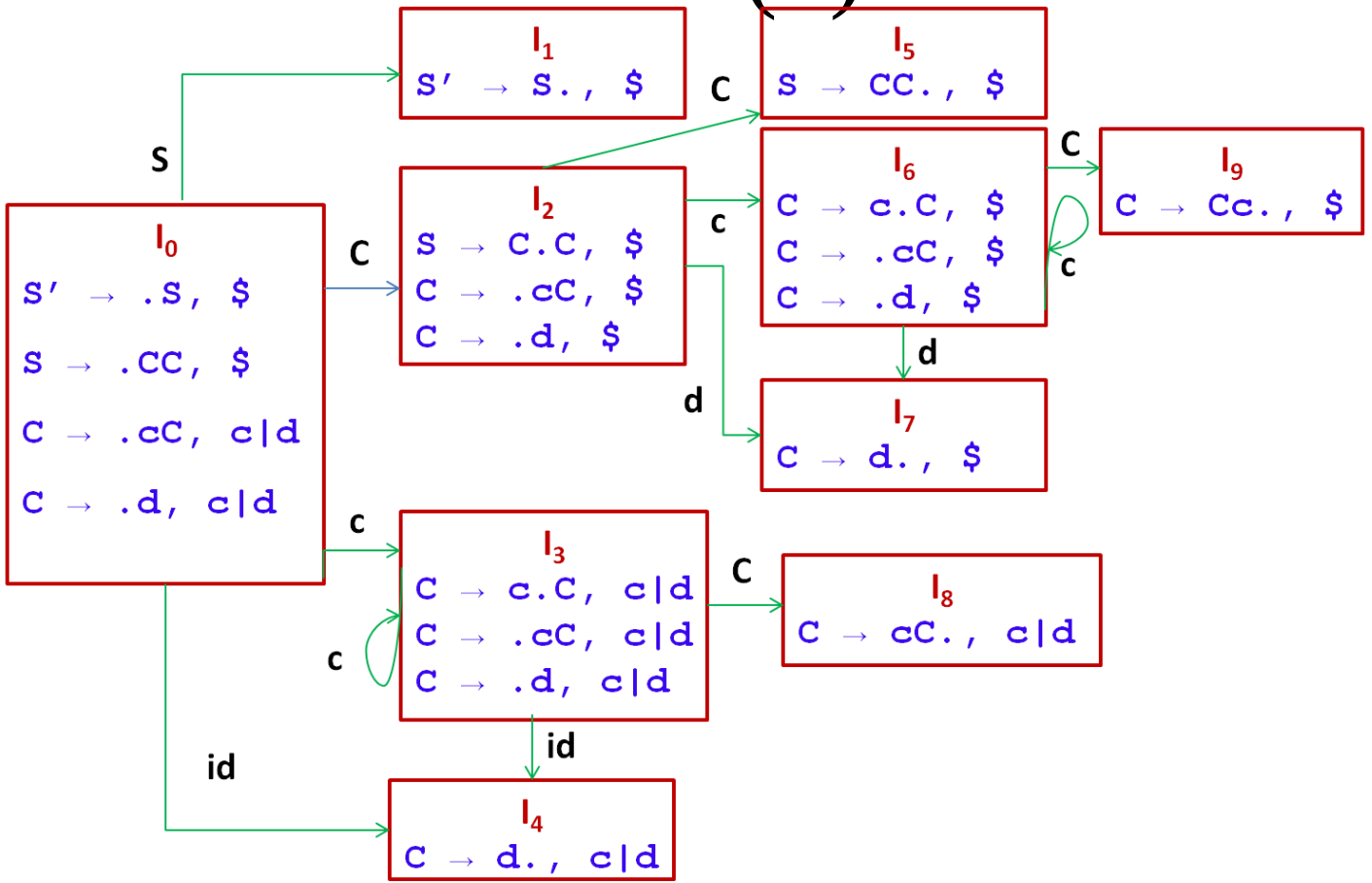

Fig: DFA for LR(1) items

Constructing LALR-Parsing Table

Example: ***Solution***

Observe that the items I_3 & I_6, I_4 & I_7 and I_8 & I_9 can be grouped as they consist of common cores and the resulting **LR(1)** items are shown below:

Constructing LALR-Parsing Table

Example: I_3 & I_6, I_4 & I_7 and I_8 & I_9

I_3: GOTO (I_0, c)
C → c.C, c | d
C → .cC, c | d
C → .d, c | d
I_4: GOTO (I_0, d)
C → d., c | d
I_8: GOTO (I_3, C)
C → cC., c | d

I_6: GOTO (I_2, c)
C → c.C, $
C → .cC, $
C → .d, $
I_7: GOTO (I_2, d)
C → d., $
I_9: GOTO (I_6, C)
C → cC., $

Constructing LALR-Parsing Table

Example: FINAL LR(1) ITEMS

I_0:

S′ → .S, $

S → .CC, $

C → .cC, c | d

C → .d, c | d

I_1: GOTO (I_0, S)

S′ → S., $

Constructing LALR-Parsing Table

Example: **FINAL LR(1) ITEMS**

I_2: GOTO (I_0, C)

S → C.C, $

C → .cC, $

C → .d, $

I_{36}: GOTO (I_0, c)

C → c.C, c | d | $

C → .cC, c | d | $

C → .d, c | d | $

Constructing LALR-Parsing Table

Example: **FINAL LR(1) ITEMS**

I_{47}: GOTO (I_0, d)

C → d., c | d | $

I_5: GOTO (I_2, C)

S → CC., $

I_{89}: GOTO (I_{36}, C)

C → cC., c | d | $

Constructing LALR-Parsing Table

Example: ***Solution***

Step 2(a): Construction of **LALR parsing table**

The **ACTION** entries for terminals can be obtained as shown below: ***Algorithm Rule 2.a***

Transition GOTO $(I_i, a) = I_j$	ACTION [i, a] = shift j
$I_0, c = I_{36}$	$[0, c] = s_{36}$
$I_0, d = I_{47}$	$[0, d] = s_{47}$
$I_2, c = I_{36}$	$[2, c] = s_{36}$

Constructing LALR-Parsing Table

Example: ***Solution***

Step 2(a):

Transition GOTO $(I_i, a) = I_j$	ACTION [i, a] = shift j
$I_2, d = I_{47}$	$[2, d] = s_{47}$
$I_{36}, c = I_{36}$	$[36, c] = s_{36}$
$I_{36}, d = I_{47}$	$[36, d] = s_{47}$

Constructing LALR-Parsing Table

Example: ***Solution***

Step 2(b): Construction of **LALR parsing table**

The **ACTION** entries for the items ending with dot (.) are shown below: ***Algorithm Rule 2.b***

$[A \rightarrow \alpha.,a] \epsilon I_i$	ACTION [i, a] = r $A \rightarrow \alpha$
$[C \rightarrow d., c\|d\|\$] \epsilon I_{47}$	[47, {c, d, $}] = r $C \rightarrow d$ (r_3)
$[S \rightarrow CC., \$] \epsilon I_5$	[5, {$}] = r $S \rightarrow CC$ (i.e., r_1)
$[C \rightarrow cC., c\|d\|\$] \epsilon I_{89}$	[89, {c, d, $}] = r $C \rightarrow d$ (r_2)

Constructing LALR-Parsing Table

Example: ***Solution***

Step 2(c): $[S' \rightarrow S., \$] \in I_i$ then **ACTION [i, $] = accept**: ***Algorithm Rule 2.c***

$[S' \rightarrow S., \$] \in I_i$	**ACTION [i, $] = accept**
$[S' \rightarrow S., \$] \in I_1$	**[1, $] = accept**

Constructing LALR-Parsing Table

Example: ***Solution***

Step 2 (d): The **GOTO** states can be computed using *rule-3* are shown below: ***Algorithm Rule 3***

Transition GOTO (I_i, A) = I_j	Table GOTO [i, A] = j
I_0, S = I_1	[0, S] = 1
I_0, C = I_2	[0, C] = 2
I_2, C = I_5	[2, C] = 5
I_{36}, C = I_{89}	[36, C] = 89

Constructing LALR-Parsing Table

Example: ***Solution***

The final LALR parsing table:

	ACTION			GOTO	
	c	d	$	S	C
0	S_{36}	S_{47}		1	2
1			acc		
2	S_{36}	S_{47}			5
36	S_{36}	S_{47}			89
47	r_3	r_3	r_3		
5			r_1		
89	r_2	r_2	r_2		

Constructing LALR-Parsing Table

Example: ***Solution***

In the above **LALR parsing table**, there are no multiple entries in **ACTION**, then the grammar is **LALR(1)** grammar.

Note: Since there are no multiple entries in the parsing table, the above parsing table is **LALR(1)** parsing table and the parser which uses this table is called **LALR** parser and the grammar is said to be **LR(1)** grammar.

Constructing LALR-Parsing Table

Sequence of moves:

The sequence of moves made by **LALR** parser for the string ***ccdd*** using the given grammar and **LALR** parsing table is shown below:

Stack	Input	Action
0	ccdd$	S_{36} => shift 36 onto the stack.
036	cdd$	S_{36} => shift 36 onto the stack.
03636	dd$	S_{47} => shift 47 onto the stack.

Constructing LALR-Parsing Table

Sequence of moves:

Stack	Input	Action
0363647	d$	r_3 => Reduce using 3rd production C → d. Pop |d| = 1 state from stack i.e., 47 and push GOTO (36, C) = 89 onto the stack.
0363689	d$	r_2 => Reduce using 2nd production C → cC. Pop |cC| = 2 states from stack i.e., 89 & 36 and push GOTO (36, C) = 89 onto the stack.

Constructing LALR-Parsing Table

Sequence of moves:

Stack	Input	Action
0363689	d$	r_2 => Reduce using 2nd production C → cC. Pop |cC| = 2 states from stack i.e., 89 & 36 and push GOTO (36, C) = 89 onto the stack.
03689	d$	r_2 => Reduce using 2nd production C → cC. Pop |cC| = 2 states from stack i.e., 89 & 36 and push GOTO (0, C) = 2 onto the stack.

Constructing LALR-Parsing Table

Sequence of moves:

Stack	Input	Action
0 2	d $	S_{47} => shift 47 onto the stack.
0 2 4 7	$	r_3 => Reduce using 3rd production C → d. Pop |d| = 1 state from stack i.e., 47 and push GOTO (2, C) = 5 onto the stack.
0 2 5	$	r_1 => Reduce using 1st production S → CC. Pop |CC| = 2 states from stack i.e., 5 & 2 and push GOTO (0, S) = 1 onto the stack.

Constructing LALR-Parsing Table

Sequence of moves:

Stack	Input	Action
01	$	ACCEPT, Parsing successful

Constructing LALR-Parsing Table

Practice Problem:

Consider the following grammar

1. **S → AA**
2. **A → Aa | b**

a) Construct sets of LR(1) items

b) Construct LALR(1) parsing table

c) Show the parsing steps for the string **“baaba”**

Compaction of LR Parsing Tables

- A typical programming language grammar with **50** to **100** terminals and **100** productions may have an **LALR** parsing table with several hundred states.
- The action function may easily have **20,000** entries, each requiring at least **8** bits to encode.
- On small devices, a more efficient encoding than a two-dimensional array may be important.
- We shall mention briefly a few techniques that have been used to compress the **ACTION** and **GOTO** fields of an **LR parsing table**.

Compaction of LR Parsing Tables

- One **useful technique** for compacting the action field is to recognize that usually many rows of the action table are identical.

	ACTION			GOTO	
	c	d	$	S	C
0	S_3	S_4		1	2
1			acc		
2	S_6	S_7			5
3	S_3	S_4			8
4	r_3	r_3			
5			r_1		
6	S_6	S_7			9
7			r_3		
8	r_2	r_2			
9			r_2		

Compaction of LR Parsing Tables

Example:

- In previous **CLR parsing table**, states **0** and **3** have identical action entries, and so do **2** and **6**. We can therefore save considerable space, at little cost in time, if we create a pointer for each state into a one-dimensional array. Pointers for states with the same actions point to the same location. To access information from this array, we assign each terminal a number from zero to one less than the number of terminals, and we use this integer as an offset from the pointer value for each state. In a given state, the parsing action for the ***i*th** terminal will be found ***i*** locations past the pointer value for that state.

Compaction of LR Parsing Tables

- Further space efficiency can be achieved at the expense of a somewhat slower parser by creating a list for the actions of each state.
- The list consists of (**`terminal-symbol`**, **`action`**) pairs.
- The most frequent action for a state can be placed at the end of the list, and in place of a terminal we may use the notation "**`any`**," meaning that if the current input symbol has not been found so far on the list, we should do that action no matter what the input is.
- Moreover, error entries can safely be replaced by reduce actions, for further uniformity along a row. The errors will be detected later, before a shift move.

Compaction of LR Parsing Tables

- We can also encode the **GOTO** table by a list, but here it appears more efficient to make a list of pairs for each nonterminal ***A***.
- Each pair on the list for ***A*** is of the form (*currentState*, *nextState*), indicating

GOTO[*currentState*, *A*] = *nextState*

- This technique is useful because there tend to be rather few states in any one column of the **GOTO** table.
- The reason is that the **GOTO** on nonterminal ***A*** can only be a state derivable from a set of items in which some items have ***A*** immediately to the left of a dot.

Summary...

Bottom-Up Parsing: **Constructing LALR Parsing Table**

- LALR-Parsing Table
- Algorithm for Constructing LALR-Parsing Table
- Example problem
- Compaction of LR Parsing Tables

Reading: Aho2, Section 4.7.4 to 4.7.6

Next Lecture: Using Ambiguous Grammar