

**Compiler Design**  
**Prof. Santanu Chattopadhyay**  
**Department of E & EC Engineering**  
**Indian Institute of Technology, Kharagpur**

**Lecture – 30**  
**Parser (Contd.)**

So, next we will be looking into the closure algorithm. So, for this LR 1 items the closure algorithm will be similar to what we have the closure algorithm for this LR SLR parsing.

(Refer Slide Time: 00:29)

**Closure algorithm**

```
SetOfItems CLOSURE(I) {  
    J=I;  
    repeat  
        for (each item  $[A \rightarrow \alpha.B\beta;a]$  in J)  
            for (each production  $B \rightarrow \gamma$  of G and each terminal  $b$  in  $\text{First}(\beta a)$ )  
                if ( $[B \rightarrow \gamma;b]$  is not in J)  
                    add  $[B \rightarrow \gamma;b]$  to J;  
    until no more items are added to J on one round;  
    return J;  
}
```

So, here but only thing is that this items have got this additional part. So, it has got this look ahead token part, so, this look ahead tokens will also appear in the closure state. So, for an item I so, if we are trying to get a closure so, for I we are trying to get a closure we initialize this set J to I and for each item a producing  $\alpha.B\beta;a$  in J in. So, this should be in I this is not in J in I. So, we have to see whether there is a grammar rule which has got the form B producing gamma.

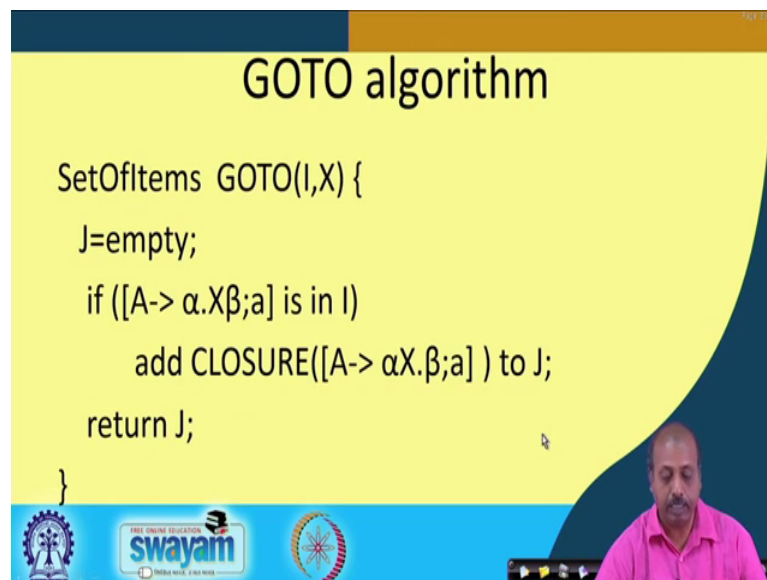
So, if there is a rule like B producing gamma then each terminal for each terminal B in the first of beta a. So, when should we go by this rule? When should we go by this rule? So, this go to B; so, this go to B we will see we will like to see if we can we see that there is something starting with beta followed by a. So, if there is something like that then only we will be using that particular transition in the parser. So, that is what is done

here, so, for every production rule so, which is of the form  $B \rightarrow \alpha \beta$  then will be so for each then for each terminal  $B$  in the first of  $\beta$  a so we are doing this  $B \rightarrow \alpha \beta$  producing dot  $\beta$  semicolon  $b$  it is not in  $J$  this is not yet added into the into the set  $J$ . So, we add this  $B \rightarrow \alpha \beta$  producing dot  $\beta$  semicolon  $b$  to  $J$ .

So, that way it is just otherwise it is same only thing is that for. So, previously we did not for the LR 0 item we did not have look ahead, so, there was this part was absent, so, this part was absent. So, we were just doing it like this for each production  $B \rightarrow \alpha \beta$  producing gamma of  $J$  we were adding this  $B \rightarrow \alpha \beta$  producing dot gamma into  $J$ . So, that is what we were doing, but now I have to tell what is the look ahead part also. So, just to do that, so, look ahead part is decided by this by you compute the first of  $\beta$  a and whatever symbols are coming in the first of  $\beta$  a, so, we have to add this thing into this set  $J$ .

So, this way the process will continue till no more items can be added to  $J$  then will be returning  $J$ . So, that is the closure computation algorithm, so, the we can use this for getting the closure of items.

(Refer Slide Time: 03:08)



## GOTO algorithm

```

SetOfItems GOTO(I,X) {
    J=empty;
    if ([A->  $\alpha$ .X $\beta$ ;a] is in I)
        add CLOSURE([A->  $\alpha$ X. $\beta$ ;a] ) to J;
    return J;
}

```

And then we can see the GOTO part. So, GOTO part is also a similar. So, there is initially, so, we are trying to find this GOTO of  $I X$ . So,  $I$  is an set of items and  $X$  is the grammar symbol then goes initially  $J$  is equal made equal to empty. Then if there is an item like  $A \rightarrow \alpha \cdot X \beta$  semicolon  $a$  in  $I$ , then we take the closure of  $A \rightarrow \alpha X \cdot \beta$  semicolon  $a$  and then we add that item to  $J$ . So, this way we

take the closure and add all those items to J and then it will be returning J. So, this way we can constitute the closure set the GOTO part for the this LR 1 items.

(Refer Slide Time: 04:05)

**Constructing LR(1) Parsing Table**

- Method
  - Construct  $C = \{I_0, I_1, \dots, I_n\}$ , the collection of LR(1) items for  $G'$
  - State  $i$  is constructed from state  $I_i$ :
    - If  $[A \rightarrow \alpha \cdot a \beta; b]$  is in  $I_i$  and  $\text{Goto}(I_i, a) = I_j$ , then set  $\text{ACTION}[i, a]$  to "shift  $j$ "
    - If  $[A \rightarrow \alpha \cdot a]$  is in  $I_i$ , then set  $\text{ACTION}[i, a]$  to "reduce  $A \rightarrow \alpha$ "
    - If  $[S' \rightarrow S \cdot \$]$  is in  $I_i$ , then set  $\text{ACTION}[i, \$]$  to "Accept"
  - If any conflicts appears then we say that the grammar is not LR(1).
  - If  $\text{GOTO}(I_i, A) = I_j$  then  $\text{GOTO}[i, A] = j$
  - All entries not defined by above rules are made "error"
  - The initial state of the parser is the one constructed from the set of items containing  $[S' \rightarrow \cdot S; \$]$

*Handwritten note:*  $\text{GOTO}(S \rightarrow S)$

Next we will see how, next we will see how to construct the LR 1 passing table. So, for constructing LR 1 parsing table so, the policy is same as we have done in LR 0 parsing table. So, we first construct the collection of LR 1 items  $C$  is the  $I_0, I_1$  up to  $I_n$  for the augmented grammar  $G$  dash and then state  $i$  is constructed from this item  $I_i$  like this. So, if  $A$  producing  $\alpha$  dot  $a$  beta comma semicolon  $b$  is in the item is in the set  $I_i$  and  $\text{GOTO } I_i a$  is  $I_j$  then action  $i, a$  is said to be equal to shift  $j$ . So, this part there is no change, so, this is exactly same as what we have for the LR 0 parser, the SLR parser.

But, this one it says that if you have got a rule like  $A$  producing  $\alpha$  dot semicolon  $a$  is in  $I_i$  then set action  $i, a$  to reduce by  $A$  producing  $\alpha$ . So, here you note that we are not looking into the follow set of  $A$ . So, follow set of  $A$  whether it can contain small  $a$  or not etcetera those things are not need not be computed. Definitely the follow set will if it is a valid thing then the follow of  $A$  will contain small  $a$ , if the if my string is valid then the follow of capital  $A$  will definitely contain small  $a$  that is why this after doing this reduction. Because the string was something like this so, this part was  $\alpha$  and then one look ahead token was  $a$  and we said that we will reduce by this whole thing by  $A$ ; that means, in some sentential form this small  $a$  will appear after capital  $A$ .

So, small  $a$  is definitely there in the follow set of  $A$ , but  $a$  may be some symbols for some cases it will be something different. So, we will be doing this reduction only when we are getting this  $i, a$ . So, in case of LR 0 parser so, this reduction rule was put for all the symbols which are in follow of  $A$  follow of capital  $A$ , but in this LR 1 parser. So, we are adding the rule only to the case where the input symbol is small  $a$ .

And then  $S \dashv$  producing  $S \cdot$  semicolon dollar; so, if it is there in some  $i$  sets of set of item then, we said this action  $i$  dollar to accept. So, this is the; this is the simple way the same it is same as that SLR parsing and if there is any conflict that appear then we say that the grammar is not it is not a SLR so, this is not LR 1. So,  $S$  should not be there, so, if there is any conflict even after doing this if there is a conflict then the grammar is not LR 1 grammar.

So, but unfortunately we do not know grammars which are parsers which are more powerful than this  $S$  the canonical LR parser. So, if this thing fails; that means, you really need to work with your grammar and try to do something or bring some context sensitivity into the parsing process. So, you by means of context free grammar, so, you will not be able to generate a parser for the particular language.

So, we will see that if you look into this parser generated tool Yacc then Yacc, bison etcetera. So, those tools so, they apart from this LALR or LR 1 parsing table generation. So, it they also allow several context dependent features and those context dependant features may be used for resolving the conflicts that are occurring in an LR 1 parser. But, anyway with a for the time being we assume that once we have come to this LR 1 parser since it is very powerful so, it has it will be able to resolve whatever constructs we have in our programming language.

And then the GOTO part is same as what we have in a LR 0 parser. So, if GOTO  $i, A$  is equal to  $I_j$  then we say this GOTO  $i, A$  equal to  $j$  and all other entries which are not defined. So, they are they will be marked as error and initial state of the parser is the one that is constructed from this one,  $S \dashv$  producing  $\cdot S$  semicolon dollar. So, this dollar is the um end of string character or the so, then this is the previously in case of LR 0 parsing table construction so, we took the item  $S \dashv$  producing  $\cdot S$  and we took the closure of that. But, here I have to tell the look ahead token also, so, here the look ahead token is dollar. So, that is the additional thing that we have.

So, this way we can have this LR 1 parsing table constructed and we can have otherwise the parsing algorithm is same, so, that is same with the what we have in the LR the parsing 1.

(Refer Slide Time: 09:28)


### Example LR(1) Parser

goal -> expr  
 expr -> term + expr  
 expr -> term  
 term -> factor \* term  
 term -> factor  
 factor -> id

```

I0 : [goal -> expr, $], [expr -> term + expr, $], [expr -> term, $], [term -> factor * term, {+, $}],
      [term -> factor, {+, $}], [factor -> id, {+, $}]
I1 : [goal -> expr, $]
I2 : [expr -> term, $], [expr -> term + expr, $]
I3 : [term -> factor, {+, $}], [term -> factor * term, {+, $}]
I4 : [factor -> id, {+, $}]
I5 : [expr -> term + expr, $], [expr -> term + expr, $], [expr -> term, $],
      [term -> factor * term, {+, $}], [term -> factor, {+, $}], [factor -> id, {+, $}]
I6 : [term -> factor * term, {+, $}], [term -> factor * term, {+, $}], [term -> factor, {+, $}],
      [factor -> id, {+, $}]
I7 : [expr -> term + expr, $]
I8 : [term -> factor * term, {+, $}]
      
```

LR(1) items



So, let us take an example suppose we have got a set of grammar rules like this. So, this goal producing expression producing term plus expression producing term and then this term producing expression a factor star term produce a factor produce id. So, this is the grammar G and we have added this additional rule goal producing expression to make the grammar G dash the augmented grammar G dash.

Now, I set that the first rule the first state or the first set of item I 0 is the closure of this goal producing dot expression and the symbol the look ahead token is dollar. So, I am expecting a string I am expecting to see a string expression and that will be followed by dollar. So, that the next input symbol will be dollar; so that way this is done and then you have to take the closure of that.

So, this dot expression is there. So, expression producing dot term plus expression so, this will be created these dollar is carried forward then this expression producing dot term is taken, so, this dollar is carried forward. Term producing now expression producing dot term is there. So, based on that this term producing dot factor star term and but here now this if you look into this one this expression producing dot term factor dot term comma dollar then this from this dot term you will see that it will be producing

dot factor star term and then from this dot factor it will be giving me this also it will be giving me a term producing factor and then this term producing dot factor and then this expression producing term. So, this will be giving me expression producing dot term, so, this one will come.

And many of these symbols like plus and dollar. So, they will come both of them will come because another item will be created where this plus will also be coming because their core part will become same that way it will come.

Similarly, from I1, so, this is goal producing expression dot will come. From I2 it will be expression producing term dot and then. So, from this rule expression producing dot term dollar, so, it will giving me expression producing term dot and this expression it will give me from this item. So, it will give me expression producing term dot plus expression dollar, so, this way this items the LR 1 items will be constructed for this particular grammar.

(Refer Slide Time: 12:26)

### Example LR(1) Parser (Contd.)

```

graph TD
    S0((S0)) -- term --> S2((S2))
    S0 -- factor --> S3((S3))
    S0 -- expr --> S1((S1))
    S0 -- id --> S4((S4))
    S2 -- "+" --> S5((S5))
    S3 -- "*" --> S6((S6))
    S3 -- factor --> S3
    S3 -- id --> S4
    S5 -- expr --> S7((S7))
    S6 -- term --> S8((S8))
    S6 -- factor --> S3
    S7 -- id --> S4
    S8 -- id --> S4
    S4 -- id --> S4
    
```

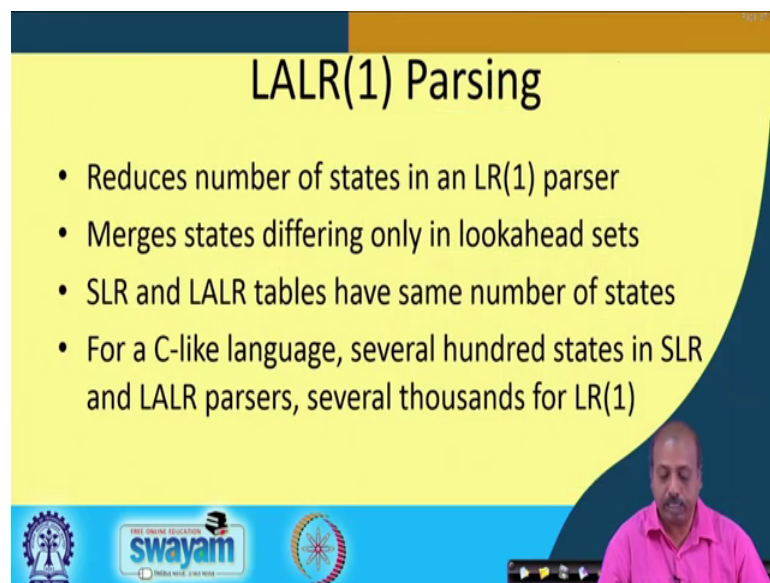
	id	+	*	\$	Expr	Term	factor
0	S4				1	2	3
1					Acc		
2		S5			R3		
3		S5	R6		R5		
4		R6	R6	R6			
5	S4				7	2	3
6	S4					8	3
7					R2		
8		R4			R4		

Next once this items have been constructed so, we can construct the corresponding table so, it corresponding automata will be like this. So, S0, S1, S2 so, these are the various states and then based on this rule that we have seen previously that parsing table construction rule so we can construct this table. So, this way so, for from state 0; so, on id so it is so, it from so it is going to state it is on id it is coming to state 4. So, this is the

that is id it is shift 4 and then on this term factor and expression so, it is going to the states 1, 2 and 3, so, accordingly this GOTO part is put like this.

So, this way we can construct this table. So, this table construction is same as what we have in the previous case like this the SLR parsing table for that how we whatever way it is constructed so, that they will be coming like that.

(Refer Slide Time: 13:29)



**LALR(1) Parsing**

- Reduces number of states in an LR(1) parser
- Merges states differing only in lookahead sets
- SLR and LALR tables have same number of states
- For a C-like language, several hundred states in SLR and LALR parsers, several thousands for LR(1)

The slide features a yellow background with a dark blue curved border on the right. At the bottom, there is a blue banner with logos for 'swayam' and 'MOE, GOVT OF INDIA'. A small video inset in the bottom right corner shows a man in a pink shirt speaking.

So, once this is done so, we can this table is ready, but you see that the number of states will be much more compared to this LR 0 parser or SLR parser. Of course, for this particular case we do not have much number of states, but it may so happen in many a cases that number of states are much higher.

So, this is LALR parsing algorithm, so, that is also doing a look ahead by one token. So, they are called LR 1 they are basically LR 1 parsing, but with look ahead they are called look ahead LR that is why it is called LALR. It reduces number of states in an in LR LR 1 parser by merging states that differ only in the look ahead sets. So, if there are a number of states that they differ only in the look ahead sets then it will merge all those states together and call them as one state.

So, as a result this SLR and LALR tables they may they will have the same number of states. Though we are not proving this result formally so, it can be shown that this SLR and LALR parsers they will have more number of states whereas, the LR 1 parser will



may have large number of states. So, for a C-like language; so, there are several hundred states in the SLR and LALR parsers whereas, for LR 1 parser so, it may be one order of magnitude more. So, several thousand states may be there in the LR 1 parser.

So, this way we can have this thing we can have this LR 1 parsers converted to LALR parser. So, we will see how to do that so that the number of states are reduced, ok.

(Refer Slide Time: 15:22)

**Example**

$S' \rightarrow S$   
 $S \rightarrow CC$   
 $C \rightarrow cC \mid d$

**LR(1) items**

- $I_0 : [S' \rightarrow \cdot S, \$], [S \rightarrow \cdot CC, \$], [C \rightarrow \cdot cC, \{c, d\}], [C \rightarrow \cdot d, \{c, d\}]$
- $I_1 : [S' \rightarrow S, \$]$
- $I_2 : [S \rightarrow C \cdot C, \$], [C \rightarrow \cdot cC, \$], [C \rightarrow \cdot d, \$]$
- $I_3 : [C \rightarrow c \cdot C, \{c, d\}], [C \rightarrow \cdot cC, \{c, d\}], [C \rightarrow \cdot d, \{c, d\}]$  ✓
- $I_4 : [C \rightarrow d \cdot, \{c, d\}]$
- $I_5 : [C \rightarrow cC \cdot, \$]$
- $I_6 : [C \rightarrow c \cdot C, \$], [C \rightarrow \cdot cC, \$], [C \rightarrow \cdot d, \$]$  ✓
- $I_7 : [C \rightarrow d \cdot, \$]$
- $I_8 : [C \rightarrow cC \cdot, \{c, d\}]$
- $I_9 : [C \rightarrow cC \cdot, \$]$

States  $I_4$  and  $I_7$ ,  $I_3$  and  $I_6$ ,  $I_8$  and  $I_9$  can be merged

$I_{47} : [C \rightarrow d \cdot, \{c, d, \$\}]$   
 $I_{36} : [C \rightarrow c \cdot C, \{c, d, \$\}], [C \rightarrow \cdot c, \{c, d, \$\}], [C \rightarrow \cdot d, \{c, d, \$\}]$   
 $I_{89} : [C \rightarrow cC \cdot, \{c, d, \$\}]$

10-6 = 4+3 = 7

So, next we will see how an example of how to do this thing. So, first of all suppose this is the grammar that we have and for this grammar S dash producing S. So, this is the new rule that is added and this is S producing CC and C producing small c capital C and d. So, this small c and small c and small d so these are the terminal symbols of the grammar and rest are all non-terminal.

So, the set I 0 is constructed by which we are for which we are doing this thing. So, this we are we take this production S dash producing this item S dash producing dot S and dollar and then we have to take the closure of that. So, closure of that gives me S producing dot CC dollar. So, now dot CC dollar so, what I have to do is for taking the closure of this item so I have to consider the first set of.

So, you remember that if I have got a production like X producing dot X dot YZ semicolon alpha then what I have to do is I have to take the set first Y Z alpha and then I have to see what is coming there. So, for then the this as this they this look ahead token I



have to do it like this. So, if I have got a production rule like Y producing gamma then for that I have to see what is the first set of this YZ alpha and accordingly I have to see I have to add them to the first to this item.

So, here also so, this dot CC and there is a rule like C producing cC. So, when I am trying to use this rule, so, I have to see what can I what can be derived. So, what is the first set of this capital C? So, this first set of capital C contains this small c and small d, so, they are added. So, I have to see what is there in the first of this C dollar and this first of C dollar capital C dollar will have small c and small d, so, this is the item that is created.

Similarly, by this rule it will be C producing, so, I am looking into the C producing d, but I have to see like what are what are the first of this C, C dollar and the first of C dollar again having C and the small c and small d so, they are added to the look ahead token. In this way this look ahead tokens are calculated. So, this is a bit cumbersome, but that is the technique ok, so, we have to do it like this.

So, you see for the small grammar having only to say three grammar rules S producing C cC producing small c, capital C and or d, so, the three grammar rules. So, there are 10 states that I created that way it is a large number of states that have been created.

Now, what we look into this for converting into LALR consider the states I 4 and I 7, so, I 4 and I 7. So, you see that they are the core part is same. This is also C producing d dot, this is also C producing d dot. So, what is differing is only the look ahead part. So, here the look ahead part is C d, here the look ahead part is dollar. So, we will be combining them together and call them a new state I 47 and this has got C producing d dot semicolons that look aheads will have look aheads of both the states c, d and dollar. So, c, d and dollar they have been taken into consideration.

Then the I third then this 3 and 6; so, 3 and 6, so, this is 3 and this is 6. So, here also the look aheads are this a core part is same. So, it has got core like C producing small c dot capital C, so, this is matching; then C producing dot c capital C this is matching, C producing dot d this is also matching. So, again what is differing are the look ahead tokens. So, this look ahead tokens so, they will be merged together and we will be getting this I this state I 36 and this 8 and 9. So, here also this 8 and 9 so, they are matching. So, their core part is matching, so, we just merge them together.

So, essentially from this 1, 2, 3, 4, 5, 6 states so, we could reduce them to 3 states. So, total originally there were 10 states and now from the 6 states. So, 10 minus 6, 4 and 3 new states created, so, total number of states turns out to be 7. So, if we construct the SLR parsing table then you can check that this also gives rise to seven states. So, that way this is going to be this LALR parsing. So, this is going to help us of course, it is cumbersome in the sense that we are. So, we are the way that we have done it we have first produced the LR 1 items so and then we are trying to see whether there is any merging possibility between the items.

So, that way the constructing the LR 1 items itself is costly and then if you so, then checking for the merging and all. So, that is why this LALR parsing construction parts of construction is going to be a costly affair though automation is no problem. So, automation can be done, so, rules are well defined, so, we can do the automation very easily.

(Refer Slide Time: 21:01)



**LALR Construction – Step-by-Step Approach**

- Sets of states constructed as in LR(1) method
- At each point where a new set is spawned, it may be merged with an existing set
- When a new state S is created, all other states are checked to see if one with the same core exists
- If not, S is kept; otherwise it is merged with the existing set T with the same core to form state ST

The slide features a yellow background with a blue header and footer. The footer contains logos for 'swayam' and 'INDIA RISE, CHINA RISE' along with a video inset of a man in a pink shirt speaking.

So, what can be the other way of constructing the LALR parser? So, this is known as step by step approach for this LALR parsing table construction this item construction.

So, sets of item sets of states constructed as in LR 1 method only; however, so, we do not construct the entire LR 1 set of items and then try to do the reduction, then try to merge the items instead of that at each point whenever a new set is spawned. So, we may see that whether it can be merged with an existing set or not.

So, in the previous example you see that when we had so, when we had generated this say for example, this state number this particular state 7. So, whenever we are generating this 7. So, whenever this is generated we see that it is matching with 4. So, we do not generate a new item or new set 7 rather we add this dollar to this set, so, that way it is going to be done. So, with the it will not generate large number of new state, so, number of states generated will remain same as the S LR parsing.

So, that is how this parsing table for LALR this set of items for LALR parser will be created that at each point when a new set will be spawned. So, you will see whether it can be merged with an existing set or not. Whenever a new state S is created all other states are checked to see if one with the same core exists. And if so, if it is not if there is no such thing exists then S will be kept otherwise it is merged with the existing set T and we name the core as name with the same core to form the state ST.

So, previously you have seen that we have named the states as 47; 47 the I 4 and I 7 were merged, so, we told the item to be I 47. So, similarly this 3 and 6 was merged, so, we called it I 36 or I 36 like that. So, that way this merging is done and after doing the merging so, we can get the after doing the merging we get the new set of states and the new set of states will be number of states will be same as the SLR number of states. So, this way we can step by step we can construct the LALR parser.

(Refer Slide Time: 23:26)

## Using Ambiguous Grammars

$E \rightarrow E+E$   
 $E \rightarrow E^*E$   
 $E \rightarrow (E)$   
 $E \rightarrow id$

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

$E \rightarrow E+T \mid T$   
 $T \rightarrow T^*F \mid F$   
 $F \rightarrow (E) \mid id$

I0:  $E' \rightarrow E$   
 $E \rightarrow E+E$   
 $E \rightarrow E^*E$   
 $E \rightarrow (E)$   
 $E \rightarrow id$

I2:  $E \rightarrow (E)$   
 $E \rightarrow E+E$   
 $E \rightarrow E^*E$   
 $E \rightarrow (E)$   
 $E \rightarrow id$

I4:  $E \rightarrow E+E$   
 $E \rightarrow E+E$   
 $E \rightarrow E^*E$   
 $E \rightarrow (E)$   
 $E \rightarrow id$

I1:  $E' \rightarrow E$   
 $E \rightarrow E+E$   
 $E \rightarrow E^*E$

I3:  $E \rightarrow id$   
 $E \rightarrow E+E$   
 $E \rightarrow E^*E$

I5:  $E \rightarrow E^*E$   
 $E \rightarrow (E)$   
 $E \rightarrow E+E$   
 $E \rightarrow E^*E$   
 $E \rightarrow (E)$   
 $E \rightarrow id$

I6:  $E \rightarrow (E)$   
 $E \rightarrow E+E$   
 $E \rightarrow E^*E$

I7:  $E \rightarrow E+E$   
 $E \rightarrow E+E$   
 $E \rightarrow E^*E$

I8:  $E \rightarrow E^*E$   
 $E \rightarrow E+E$   
 $E \rightarrow E^*E$

I9:  $E \rightarrow (E)$

$E \rightarrow E+T \mid T$   
 $T \rightarrow T^*F \mid F$   
 $F \rightarrow (E) \mid id$

$m \times 3$   
 $m \times 1$

So, after this so, since it is very difficult to do some exercise on this SLR and this canonical LR and LALR parsing. So, rest of our discussion will be constructing will be restricting ourselves to SLR parsers only. But, whatever discussions we do with SLR parser so, they will also be valid for this canonical LR and LALR parsers. So, the strategy will remain same only thing is that maybe for some grammar where this SLR parser is giving rise to conflicts this canonical LR and LALR will may not give you conflicts. But, there may be situation where the grammar is such that it also gives conflict there, so, in that case we have to resolve conflicts accordingly.

So, how to so, next important issue that we will be looking into is the usage of ambiguous grammars for this parser construction. So, ambiguous grammar means that there is ambiguity. So, like this is that famous ETF this expression grammar previously we had written like  $E \rightarrow E + T \mid T$  then  $T \rightarrow T * F \mid F$  and  $F \rightarrow (E) \mid id$ , so, this was the grammar.

Now, what do we do we do not take this extra non-terminal symbols this ETF. So, why this non terminals  $T$  and  $F$  were introduced they were introduced to ensure the precedence of the operators because until and unless you have done this until and unless you have reduced to a  $T$  so, you cannot reduce to  $E$ , but for reducing to  $T$  so, this multiplications are already taken care of. So, with this you can so, whenever you are doing a reduction to  $E$  either it can be a an expression that does not contain any multiplication or it is an expression where there is a addition operation and one part of it and the two parts of it they can contain addition or multiplication, but at the top most level I have got multiplication addition.

So, that means, the addition it has got the lowest precedence compared to this multiplication. So, this way similarly this  $F$  is put within bracket  $E$ . So,  $F$  is  $F$  has been introduced to ensure that this parentheses it has got the highest precedence over any other operator. So, this way this parser so, it could generate pars trees unambiguously, but the difficulty that we face is many a times what happens is that as you are increasing your number of non-terminals in your grammar so, the parsing table becomes larger, but in particular at least the GOTO part of the table that will become larger because GOTO part will have one entry for each of the non terminal in each of the states.

So, that GOTO part is going to have problem. Like if I say that this after constructing the item set, suppose if I see that if there are say  $m$  number of items and there are three non-terminal symbols. So, GOTO part will have at least  $M$  into three number of entries. Whereas, if somehow I can make this grammar that has got only say one non-terminal then that table of table size will become  $M$  into 1. So, that way the number of entries in the table becomes small, so, that table becomes simple.

So, what we do we many times this parser the parser generators they say that you we purposefully make the grammar ambiguous. So, that the parsing table size becomes small and then naturally once we do that so, it can give rise to conflicts. So, I can so, I have to reduce I have to dissolve this conflicts and we dissolve the conflicts in a particular fashion. So, depending on the grammar so from the knowledge about the language for which we are designing the parser. So, we can try to resolve the we can try to resolve the ambiguity by putting additional rules.

Like, if I substitute if I write this particular grammar without using this extra non-terminals  $T$  and  $F$  I always tell it is  $E$ , then we know that this addition has is of lower precedence than multiplication. So, if this I think is told to the parser so, when it is generating the parse tree so, if it has got a confusion that on getting the next input symbol, the input operator plus or star whether to shift or whether to reduce, so it can take a decision depending on what operator it has seen and what is the current situation, so, based on that, it can take a decision.

So, we will see that this ambiguous grammars it can lead to conflicts, but at the same time this ambiguous grammars so they can be used to make the parsing table small and that helps in the overall parsing process, so, makes the parsing process faster. So, that way, this sometimes we will see that we purposefully make it make the grammar ambiguous.