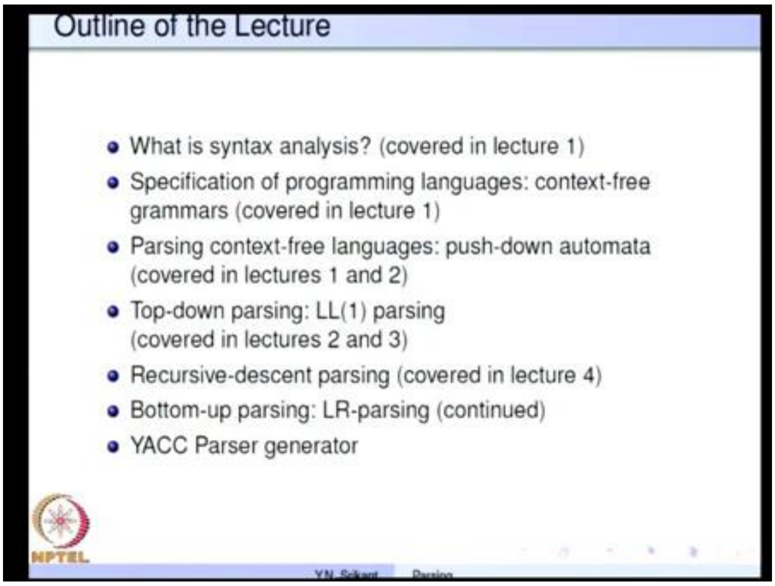
Principles of Complier Design

Prof. Y. N. Srikant
Department of Computer Science and Automation
Indian Institute of Science, Bangalore

Lecture - 11

Syntax Analysis: Context-free Grammars, Pushdown Automata and Parsing Part – 7

(Refer Slide Time: 00:23)



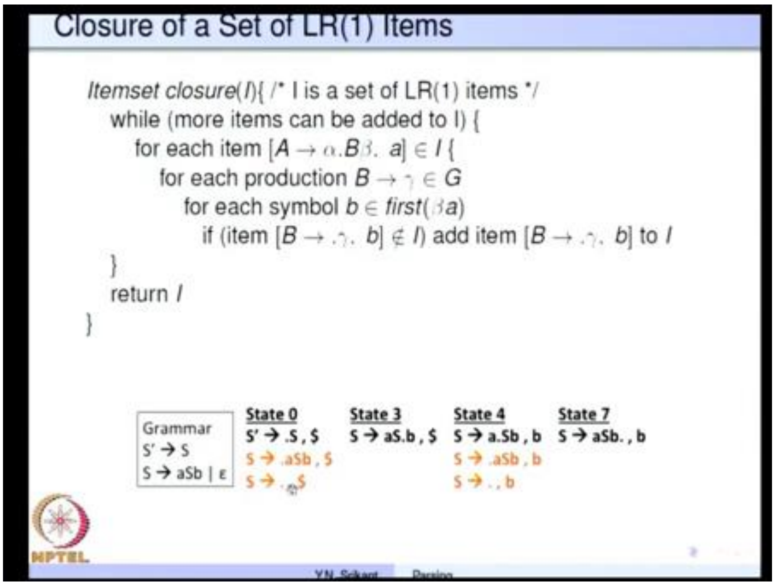
Outline of the Lecture

- What is syntax analysis? (covered in lecture 1)
- Specification of programming languages: context-free grammars (covered in lecture 1)
- Parsing context-free languages: push-down automata (covered in lectures 1 and 2)
- Top-down parsing: LL(1) parsing (covered in lectures 2 and 3)
- Recursive-descent parsing (covered in lecture 4)
- Bottom-up parsing: LR-parsing (continued)
- YACC Parser generator

Y.N. Srikant Parsing

Welcome to the lecture number 7 in parsing part 7. So far we have seen you know various topics is syntax analysis on including, context free grammars, push down automata, LL parsing, recursive descent parsing and parts of LR parsing. Today we will continue with LR parsing and also look at the commercial YACC parser generator.

(Refer Slide Time: 00:45)



Closure of a Set of LR(1) Items

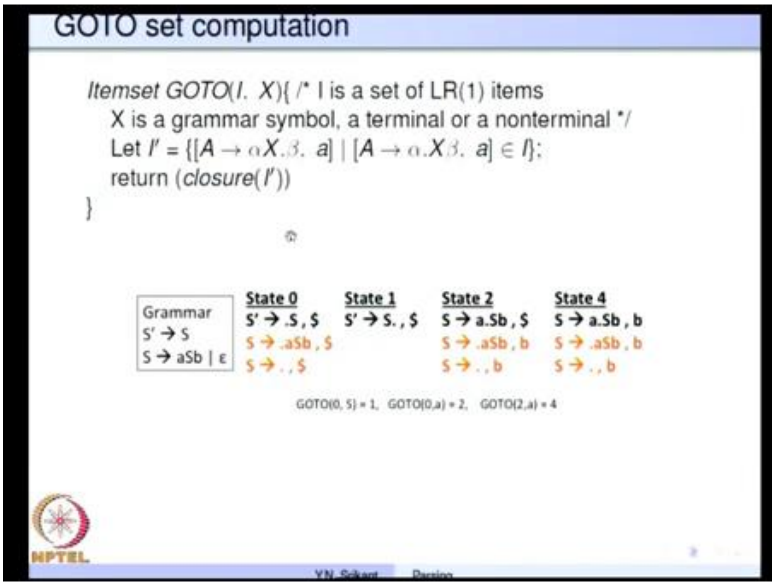
```
Itemset closure(I){ /* I is a set of LR(1) items */
  while (more items can be added to I) {
    for each item [$A \rightarrow \alpha.B\beta,\ a$] $\in I$ {
      for each production $B \rightarrow \gamma \in G$
        for each symbol $b \in first(\beta a)$
          if (item [$B \rightarrow .\gamma,\ b$] $\notin I$) add item [$B \rightarrow .\gamma,\ b$] to $I$
  }
  return I
}
```

Grammar	State 0	State 3	State 4	State 7
S' → S S → aSb | ε	S' → .S , \$ S → .aSb , \$ S → . , \$	S → aS.b , \$	S → a.Sb , b S → .aSb , b S → . , b	S → aSb. , b

Y.N. Srikant Parsing

So, just to do a bit of recap, we discussed LR 1 items, LR 1 item construction, etcetera in the last lecture. So, here we define two operations item set closure, which really includes all items B going to dot gamma comma b, whenever there is an item of the form A going to alpha dot B beta comma a in certain item set. So, if you take this example s prime going to s and s going to a s b or Epsilon, to be begin with the initial item set s prime going to dot s comma dollar. And then because of this s we add a going to dot s b comma dollar and then we add s going to dot comma dollar etcetera, etcetera.

(Refer Slide Time: 01:40)



GOTO set computation

```
Itemset GOTO(I, X){ /* I is a set of LR(1) items
  X is a grammar symbol, a terminal or a nonterminal */
  Let $I' = \{[A \rightarrow \alpha X.\beta,\ a] \mid [A \rightarrow \alpha.X\beta,\ a] \in I\}$;
  return (closure(I'))
}
```

Grammar	State 0	State 1	State 2	State 4
S' → S S → aSb | ε	S' → .S , \$ S → .aSb , \$ S → . , \$	S' → S. , \$	S → a.Sb , \$ S → .aSb , b S → . , b	S → a.Sb , b S → .aSb , b S → . , b

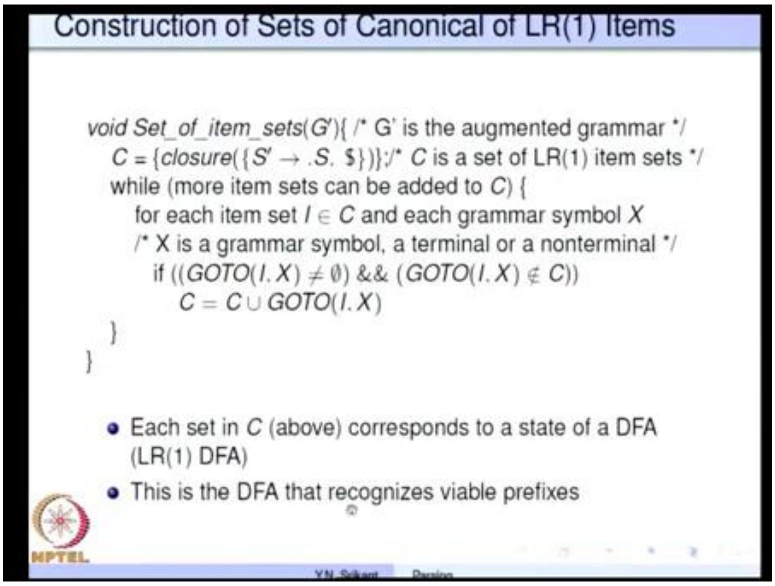
GOTO(0, S) = 1, GOTO(0,a) = 2, GOTO(2,a) = 4

Y.N. Srikant Parsing

So, and then we also define another operation go to of I comma X, I is a set of LR 1 items and X is a grammar symbol, either a terminal or a non terminal. So, here we have an item of the form A going to alpha dot X beta comma a and we advance the dot beyond this non terminal or terminal X. So, that gives us an item A going to alpha X dot beta comma a and then after these items are obtain we also take the closure.

So, for example, here from the state you know s, which contains s going to dot a s b comma dollar on little a we get the state containing the item s going to a dot s b comma dollar. Now, we take the closure and add the two items s going to dot a s b comma b and s going to dot comma b, similarly in the state four as well. So, this repeated application of advancing this dot gives us several go to states and taking the closure of that will you know add more items into those states.

(Refer Slide Time: 02:53)



Construction of Sets of Canonical of LR(1) Items

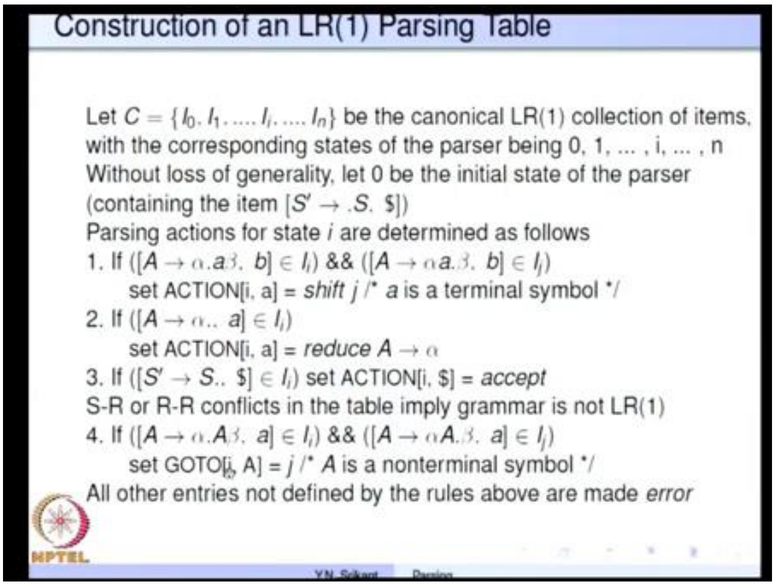
```
void Set_of_item_sets(G'){ /* G' is the augmented grammar */
  C = {closure({S' → .S, $})};/* C is a set of LR(1) item sets */
  while (more item sets can be added to C) {
    for each item set I ∈ C and each grammar symbol X
    /* X is a grammar symbol, a terminal or a nonterminal */
      if ((GOTO(I, X) ≠ ∅) && (GOTO(I, X) ∉ C))
        C = C ∪ GOTO(I, X)
  }
}
```

- Each set in C (above) corresponds to a state of a DFA (LR(1) DFA)
- This is the DFA that recognizes viable prefixes

Y.N. Srikant Parsing

So, to put together these two operations we define another function set of items sets G prime. So, we begin with the initial item s prime going to dot s comma dollar take it is closure and then apply the go to operation repeatedly and collect the sets into the same, you know collection of sets C. So, each set in this C corresponds to a state of the LR 1 DFA and this is the DFA that really recognizes the viable prefixes, this is what we learnt in the last lecture.

(Refer Slide Time: 03:33)



Construction of an LR(1) Parsing Table

Let $C = \{I_0, I_1, \ldots, I_i, \ldots, I_n\}$ be the canonical LR(1) collection of items, with the corresponding states of the parser being 0, 1, ... , i, ... , n

Without loss of generality, let 0 be the initial state of the parser (containing the item $[S' \rightarrow .S, \$]$)

Parsing actions for state i are determined as follows

1. If $([A \rightarrow \alpha.a\beta,\ b] \in I_i)$ && $([A \rightarrow \alpha a.\beta,\ b] \in I_j)$
   set ACTION[i, a] = *shift j* /* *a* is a terminal symbol */
2. If $([A \rightarrow \alpha.,\ a] \in I_i)$
   set ACTION[i, a] = *reduce* $A \rightarrow \alpha$
3. If $([S' \rightarrow S.,\ \$] \in I_i)$ set ACTION[i, $] = *accept*

S-R or R-R conflicts in the table imply grammar is not LR(1)

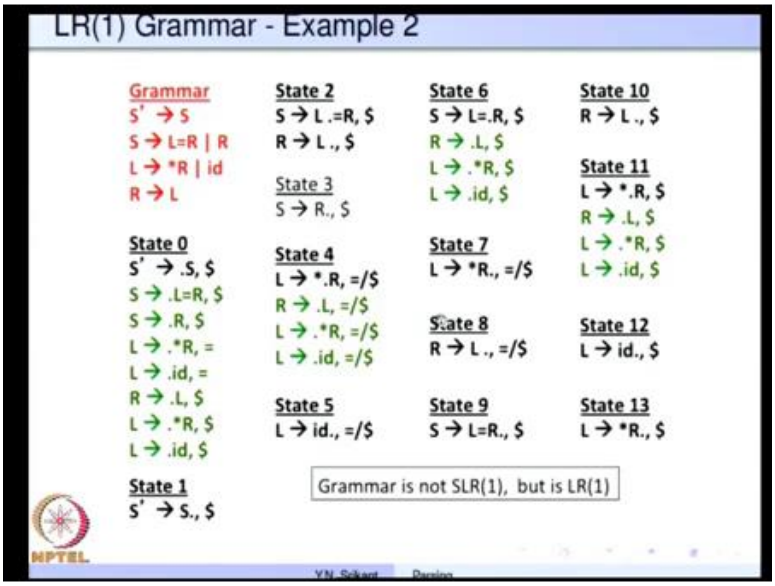
4. If $([A \rightarrow \alpha.A\beta,\ a] \in I_i)$ && $([A \rightarrow \alpha A.\beta,\ a] \in I_j)$
   set GOTO[i, A] = *j* /* *A* is a nonterminal symbol */

All other entries not defined by the rules above are made *error*

So, now construction of the LR 1 parsing table is quite straight forward, so we defined two parts in the table, action and go to for the action part. So, whenever A going to alpha dot little a beta comma b is an item, in a state and there is another state containing A going to alpha a dot beta comma b, we add action of i comma a 2 as shift j. Similarly, for the reduce whenever there is an item of the form A going to alpha dot gamma a we make it a reduce by A to alpha.

So, the particular entry in action i comma a is may reduce A to by alpha, so then we add accept and then you know the go to an error. So, the S R and R R conflicts in the table imply the grammar is not LL 1 and to add to the go to table, whenever we have a non terminal after the dot. And there is another state containing the non terminal, you know after advancement of the dot, we add a go to i comma A equal to j, so all other entries are error.

(Refer Slide Time: 04:55)



LR(1) Grammar - Example 2

Grammar
S' → S
S → L=R | R
L → *R | id
R → L

State 0
S' → .S, $
S → .L=R, $
S → .R, $
L → .*R, =
L → .id, =
R → .L, $
L → .*R, $
L → .id, $

State 1
S' → S., $

State 2
S → L .=R, $
R → L ., $

State 3
S → R., $

State 4
L → *.R, =/$
R → .L, =/$
L → .*R, =/$
L → .id, =/$

State 5
L → id., =/$

State 6
S → L=.R, $
R → .L, $
L → .*R, $
L → .id, $

State 7
L → *R., =/$

State 8
R → L ., =/$

State 9
S → L=R., $

State 10
R → L ., $

State 11
L → *.R, $
R → .L, $
L → .*R, $
L → .id, $

State 12
L → id., $

State 13
L → *R., $

Grammar is not SLR(1), but is LR(1)

Y.N. Srikant Parsing

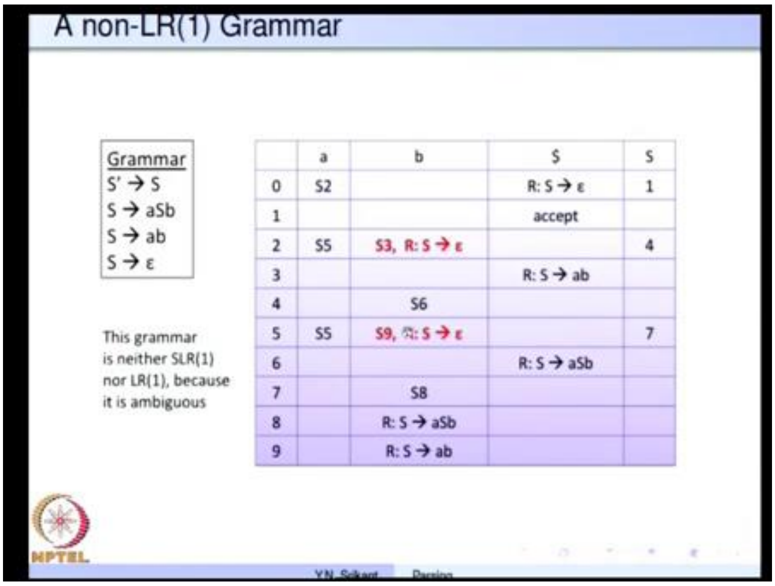
So, this is a another example here is the grammar, so this grammar is not SLR 1, but is definitely LR 1 as we can see. Let me demonstrate on this particular state how exactly it is constructed and all that is are similar, so s prime going to dot s comma dollar is the initial item. So, and because of this s we after the closure I adds s all the productions of s as a initial items, s going to dot L equal to R comma dollar and s going to dot R comma dollar.

And, because of the L and R we add the items of L and item of R as well and R gives you another item R going to dot L comma dollar. Whereas, here you know we had s going to dot R comma dollar, so here there are two sets of items corresponding to the productions of L. So, this was defined because of this item these two and their you know look ahead would be equal to because what follows L in this item is equal. Whereas, here what follows L is actually epsilon, so only dollar comes into picture and we have L going to dot star R comma dollar and L going to dot i d comma dollar.

So, observe that there are you know two items with L going to dot i d, but with different look ahead's. So, this is quite you know possible, it is not necessary to have a single items with a single look ahead etcetera, etcetera, corresponding to a production in any particular state. This particular state number 2 giving us a conflict in the SLR 1 case, so here it does not give a conflict because the reduction R going to from, I you know using the production R going to L will happen only on dollar.

Whereas, the shift happens on equal to this is possible because we have been carrying the local follow, which is you know equal to after L etcetera, etcetera, in the items as well. So, this makes sure that you know there is no possibility of a conflict in this state, so you should also observe that there are many states with more than one look ahead for example, I know a item with items, which have more than one look ahead, so R going to dot L and equal to dollar both. So, here also these two items could have been return in a similar manner, so L going to dot i d comma equal to slash dollar etcetera, etcetera, so this grammar does not have any conflicts in any states and therefore, it is LR 1.

(Refer Slide Time: 07:49)



A non-LR(1) Grammar

Grammar
S' → S
S → aSb
S → ab
S → ε

This grammar is neither SLR(1) nor LR(1), because it is ambiguous

	a	b	$	S
0	S2		R: S → ε	1
1			accept	
2	S5	S3, R: S → ε		4
3			R: S → ab	
4		S6		
5	S5	S9, R: S → ε		7
6			R: S → aSb	
7		S8		
8		R: S → aSb		
9		R: S → ab		

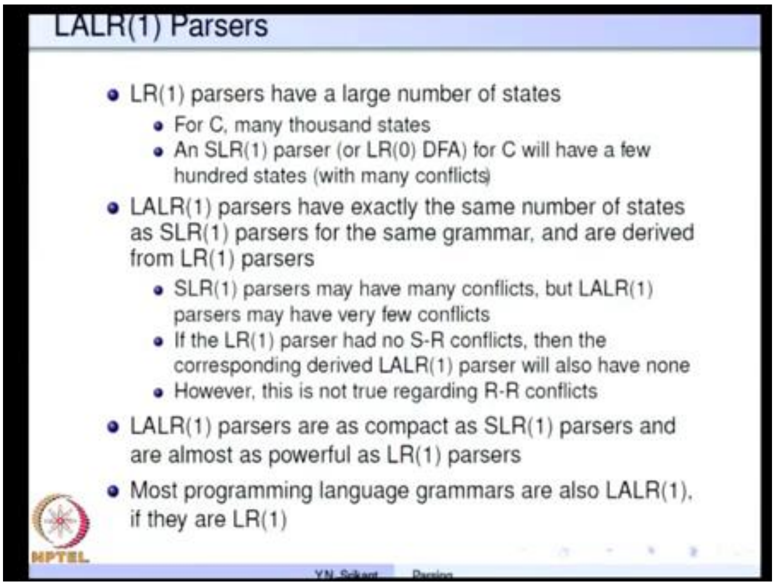
Y.N. Srikant Parsing

So, now we have seen two examples of LR 1 grammars, so it is time to go ahead and see there are grammars, which are not LR 1. So, there is a very famous theorem, which says the deterministic context free languages have LR 1 grammars and all LR 1 grammars, you know generator only SDCFS. But here we have a grammar s prime going to s, s going to a s b, s going to a b, s going to Epsilon, unfortunately this is ambiguous. So, the reason for ambiguity is we can go on producing as many a's and b's as we want.

And then stop using a either s going to a b or s going to Epsilon both are equally possible. So, therefore, you know for example, for the simple string a b, we can say s prime going to s and then s going to a b or we can say s prime going to s, s going to a s b and s going to Epsilon, so two parse trees are possible. All ambiguous grammars will fail the LR 1 test and show does this grammar as well, so if the construction of sets of items is left as home work.

So, here when we try to fill the parsing table we get two entries s 3 and then R which is a reduction s going to epsilon and here again a shift action s 9 and the reaction reduce by s 2 Epsilon. So, clearly between shift and reduce there is a conflict here and a conflict as well and this grammar is therefore, not LR 1.

(Refer Slide Time: 09:35)



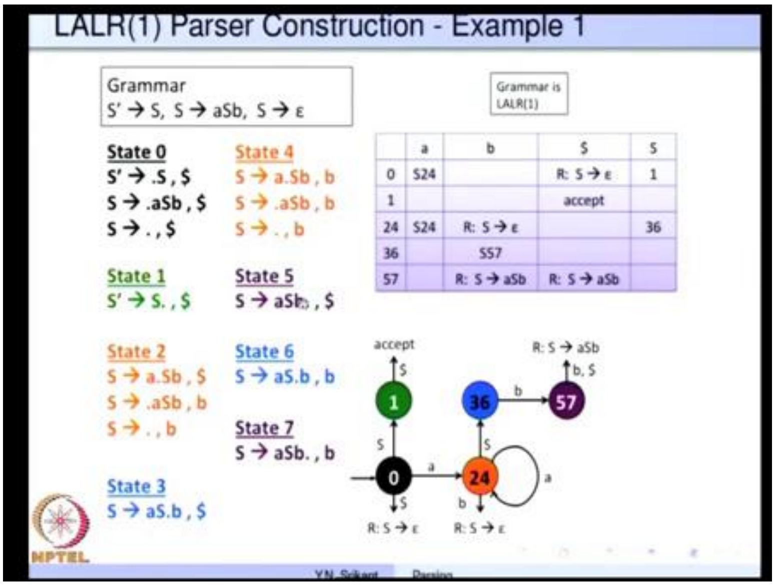
LALR(1) Parsers

- LR(1) parsers have a large number of states
  - For C, many thousand states
  - An SLR(1) parser (or LR(0) DFA) for C will have a few hundred states (with many conflicts)
- LALR(1) parsers have exactly the same number of states as SLR(1) parsers for the same grammar, and are derived from LR(1) parsers
  - SLR(1) parsers may have many conflicts, but LALR(1) parsers may have very few conflicts
  - If the LR(1) parser had no S-R conflicts, then the corresponding derived LALR(1) parser will also have none
  - However, this is not true regarding R-R conflicts
- LALR(1) parsers are as compact as SLR(1) parsers and are almost as powerful as LR(1) parsers
- Most programming language grammars are also LALR(1), if they are LR(1)

Y.N. Srikant Parsing

So, the next class of grammars is the LALR 1 grammars and they corresponding LALR 1 parsers, so what has been observed.

(Refer Slide Time: 09:53)



LALR(1) Parser Construction - Example 1

Grammar
S' → S, S → aSb, S → ε

Grammar is LALR(1)

State 0
S' → .S , $
S → .aSb , $
S → . , $

State 1
S' → S. , $

State 2
S → a.Sb , $
S → .aSb , b
S → . , b

State 3
S → aS.b , $

State 4
S → a.Sb , b
S → .aSb , b
S → . , b

State 5
S → aSb. , $

State 6
S → aS.b , b

State 7
S → aSb. , b

	a	b	$	S
0	S24		R: S → ε	1
1			accept	
24	S24	R: S → ε		36
36		S57		
57		R: S → aSb	R: S → aSb	

Y.N. Srikant Parsing

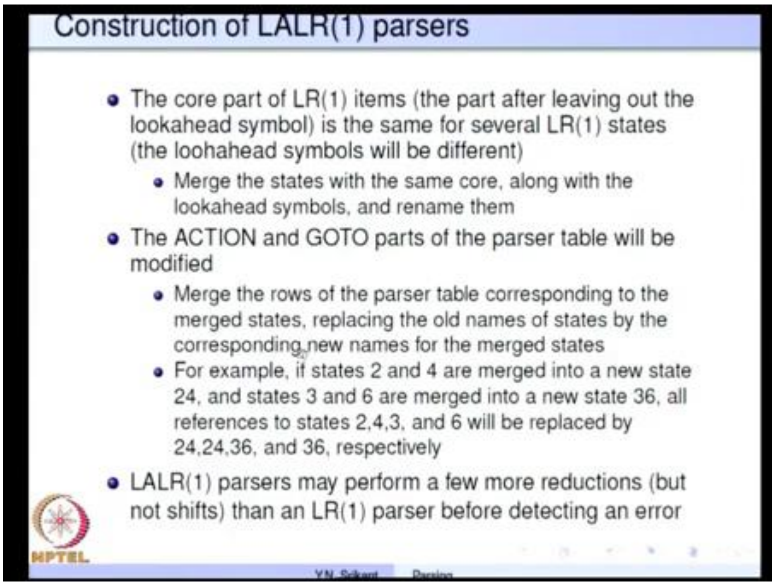
So, let me show you a picture before we discuss further, so for example, what has been observed when we construct the LR 1 sets of items is that, many states have you know the same sets of items, but the look ahead's are different. So, for example, in this case the state 0 of course, does not have any other states with the same set of items with different look ahead's, the state 1 also does not have any, but state 2 and 4, if you observe they have the same you know core item, but the look ahead's are different.

So, this part without the look ahead is called as the cornel, so the cornel of the state's 2 and 4 is the same and the look ahead's are different. Similarly, the states 3 and 6 have the same cornel s going to a s dot b, but the look ahead's are different and exactly ditto first states 5 and 7, they have a same cornel s going to a s b dot, but with different look ahead's. ((Refer Time: 11:01)) So, if this happens you know the reason why we are looking at all this is LR 1 parsers have a very large number of states.

So, for C grammar for example, there are many thousand sates whereas, if you consider an SLR 1 parser or LR 0 parser, for C it will have just a few hundred states, but there will be many conflicts. So, with the grammar is not useable, so there is a class of grammar between SLR 1 and LR 1 called as the look ahead LR 1, the advantage of these parsers is that they have the same number of states as SLR 1 parsers for the same grammar.

And they are derived from LR 1 parsers of course, there will not have as many conflicts as the SLR 1 parsers of course. So, the SLR 1 parsers may not have many conflicts, but LALR 1 parsers will have very few conflicts and of course, if the LR 1 parser had no shift reduce conflict, they corresponding derived LALR 1 parser will also have none. But for a R R conflicts this is not true, we will see the more of this later, the LALR 1 parsers are very compact as compact as the SLR 1 parsers. And are all most as powerful as the LR 1 parsers, theatrically they are not the same as LR 1. So, they are actually between SLR 1 and LR 1 and fortunately most programming language grammars happen to be LALR 1 if they or indeed LR 1.

(Refer Slide Time: 12:42)



Construction of LALR(1) parsers

- The core part of LR(1) items (the part after leaving out the lookahead symbol) is the same for several LR(1) states (the loohahead symbols will be different)
  - Merge the states with the same core, along with the lookahead symbols, and rename them
- The ACTION and GOTO parts of the parser table will be modified
  - Merge the rows of the parser table corresponding to the merged states, replacing the old names of states by the corresponding new names for the merged states
  - For example, if states 2 and 4 are merged into a new state 24, and states 3 and 6 are merged into a new state 36, all references to states 2,4,3, and 6 will be replaced by 24,24,36, and 36, respectively
- LALR(1) parsers may perform a few more reductions (but not shifts) than an LR(1) parser before detecting an error

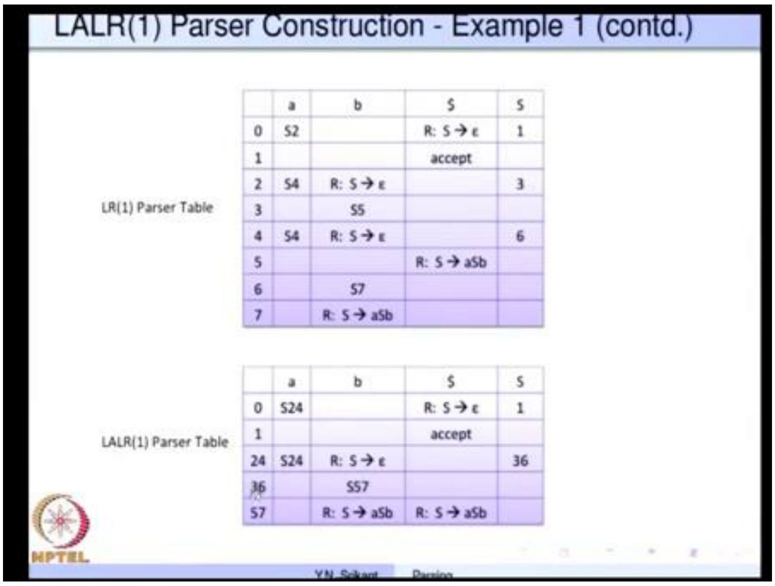
Y.N. Srikant Parsing

So, how exactly is the construction of LALR 1 parsers none, so let me again demonstrate it using the same picture. ((Refer Time: 12:55)) So, what we really do is take the states with the same cornel merge them into a single state, so 2 and 4 become a new state called 2 4, 3 and 6 become a new state called 3 6, 5 and 7 become a new state called 5 7. So, from the LR 1 DFA and the LR 1 sets of items, we get the new LALR 1 sets of items in the set of states for the DFA.

What happens to the look ahead of course, now we are going to actually merge the look ahead as well. So, in the new state 2 4 the items are going to be s going to a dot s b comma b slash dollar, s going to dot a s b comma b and s going to dot comma b, so the items are you know the look ahead's will be merged. Because, the look ahead for these two is the b, we are not going to repeat the items with the same look ahead, but for this particular item there is a dollar here and a b here.

So, we are going add the you know both the items into that set, so the look ahead is either b or dollar. So, two items will be present and that will be denoted as s going to a dot s b comma b slash dollar, so similarly the states 5 and 7 will have, you know will become a new state 5 7 with the item s going to a s b dot comma dollar slash b and 3 and 6 will become a new state merge into new state 3 6 with the item s going to a s dot b comma dollar slash b, so this is the new DFA and from this we construct the LALR 1 table.

(Refer Slide Time: 14:50)



LALR(1) Parser Construction - Example 1 (contd.)

LR(1) Parser Table

	a	b	$	S
0	S2		R: S → ε	1
1			accept	
2	S4	R: S → ε		3
3		S5		
4	S4	R: S → ε		6
5			R: S → aSb	
6		S7		
7		R: S → aSb		

LALR(1) Parser Table

	a	b	$	S
0	S24		R: S → ε	1
1			accept	
24	S24	R: S → ε		36
36		S57		
57		R: S → aSb	R: S → aSb	

NPTEL
Y.N. Srikant
Parsing

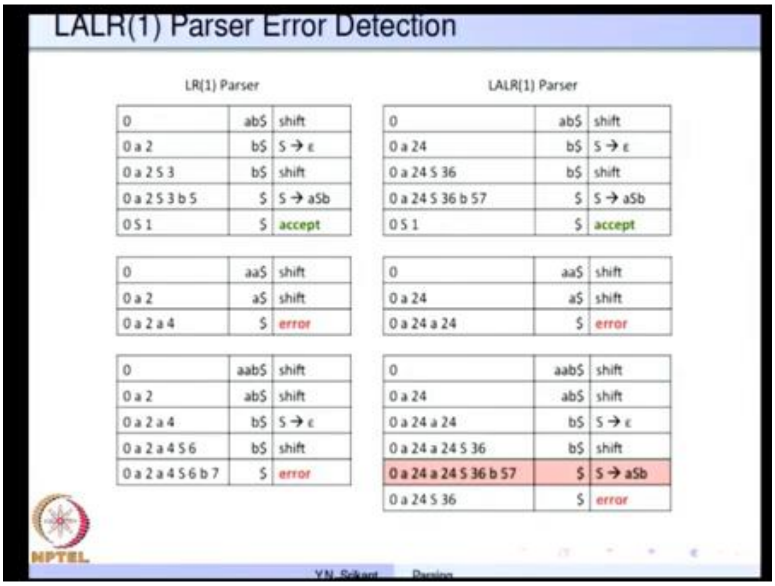
So, intuitively the rate is constructed can be demonstrated by merging the rows, so suppose we have the LR 1 parser table and now we have merge the two states 2 4, 3 6 and 5 7. So, they have become the new states and now what happens in the LR 1 parsing table is the rows correspond into 2 and 4 are actually combined, so this has s 4 s 4 in both of them. So, it becomes you know 2 and 4 will have shift action and since this state is 4, it will 2 4 has the state number here.

And then there is a reduce action here and that reduce action is the same in both 2 and 4, so we will have a reduction on this item. And then this 3 and 6; obviously, merge to 3 6, so a property a similar is the same happens with the 3 and 6 as well, so we merge 3 and 6. So, this becomes s 5 and s 7 they become the new action s 5 7, because 5 and 7 have merge and the rest of it, you know remains the same, so 3 6 have been merge 5 7 have been merge, so the number of rows here reduces.

So, what really happens is this, ((Refer Time: 16:19)) so the merge the states with the same core, along with the look ahead symbols and rename them. Now, construction of the parser action and go to tables, as I just now demonstrated will take place like this, merge the rows of the parser table corresponding to the merged states, replacing the old names of the states by the corresponding new names for the merge states. So, this is got a just now demonstrated, ((Refer Time: 16:49)) so the advantage of this particular thing is the parser has a LALR 1 parser has fewer states.

So, you can observe that if we merge the states, you know respective of the look ahead we are merging them. So, we really get the same set of states has the SLR 1 or LR 0, absolutely whenever the cornel is a same we merge all those state and the look ahead's as well. So, we really have states with unique cornels that is nothing, but the SLR 1 or LR 0 parser, but the look ahead's is also present, unfortunately the look ahead is not as strong as in the LR 1 case. But fortunately it is not also as weak as in the SLR 1 case, so this parser is stronger than the SLR 1, but we weaker than the LR 1.

(Refer Slide Time: 17:44)



LALR(1) Parser Error Detection

LR(1) Parser

0	ab$	shift
0 a 2	b$	S → ε
0 a 2 S 3	b$	shift
0 a 2 S 3 b 5	$	S → aSb
0 S 1	$	accept

0	aa$	shift
0 a 2	a$	shift
0 a 2 a 4	$	error

0	aab$	shift
0 a 2	ab$	shift
0 a 2 a 4	b$	S → ε
0 a 2 a 4 S 6	b$	shift
0 a 2 a 4 S 6 b 7	$	error

LALR(1) Parser

0	ab$	shift
0 a 24	b$	S → ε
0 a 24 S 36	b$	shift
0 a 24 S 36 b 57	$	S → aSb
0 S 1	$	accept

0	aa$	shift
0 a 24	a$	shift
0 a 24 a 24	$	error

0	aab$	shift
0 a 24	ab$	shift
0 a 24 a 24	b$	S → ε
0 a 24 a 24 S 36	b$	shift
0 a 24 a 24 S 36 b 57	$	S → aSb
0 a 24 S 36	$	error

Y.N. Srikant Parsing

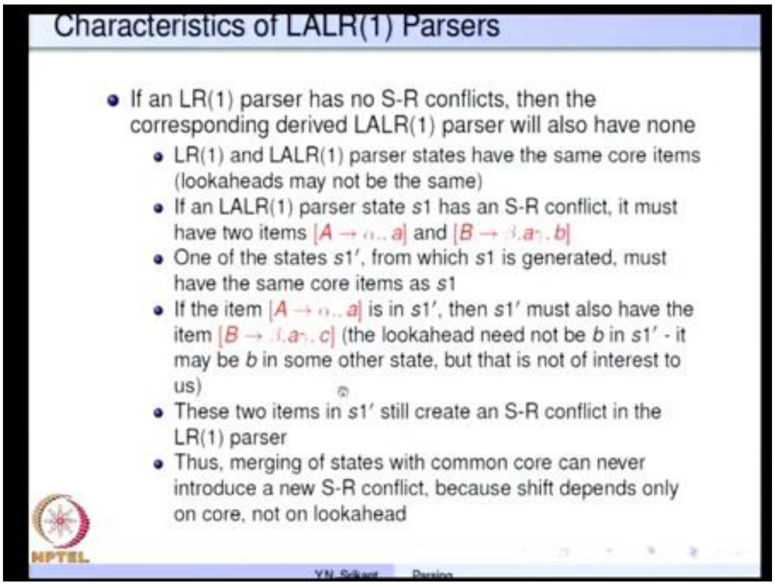
The what is the effect of this we will see now, the effect is on error detection, so let us go through these you know parser steps carefully. The action of the LALR 1 parser is identical to that of the LR 1 parser, whenever the a string is accepted, so for example, the parser on input a b dollar. So, it does a shift in the LR 1 parser case and shift in the LALR 1 parser case as well, so with the new state here is 2 and the corresponding state is 2 4 here.

Now, there is a reduction in both of them and then there is a shift again there is a shift here as well. Now, there is a reduction here and a reduction here, finally there is a accept here and an accept here as well, in this example for the wrong string a a dollar there is a shift shift and error and same is true for this as well. But in certain cases for erroneous inputs for example, a a b this does a shift then a shift then a reduce then a shift and then it finds that there is no error on dollar.

Here, it does a shift it another shift exactly like in the case of LR 1, then a reduction exactly as in the case of LR 1, another shift again exactly as in the case LR 1. Now, this LR 1 parser actually declared an error whereas, this LALR 1 parser now says there is a reduction possible by s going to a s b. So, it reduces and then declares an error, so this is the difference between the LR 1 parser and the LALR 1 parser, what happens is in the case of correct inputs, the number of steps will be identical to that of the LR 1 parser.

But, in the case of LALR 1 you know erroneous inputs, it is possible that there would be more reductions then the tough LR 1 parser and then the error would be declared. But it is certain that there will be no shift, you know extra shift that is carried out even in the LALR 1 parser. In other words, the wrong symbol which is causing the error will never be shifted on to the stack, there be a few more erroneous reductions, but never a shift. That means, the error point will remain the same in both the parsers it will never shift to the next symbol erroneously.

(Refer Slide Time: 20:32)



Characteristics of LALR(1) Parsers

- If an LR(1) parser has no S-R conflicts, then the corresponding derived LALR(1) parser will also have none
  - LR(1) and LALR(1) parser states have the same core items (lookaheads may not be the same)
  - If an LALR(1) parser state $s1$ has an S-R conflict, it must have two items $[A \rightarrow \alpha., a]$ and $[B \rightarrow \beta.a\gamma, b]$
  - One of the states $s1'$, from which $s1$ is generated, must have the same core items as $s1$
  - If the item $[A \rightarrow \alpha., a]$ is in $s1'$, then $s1'$ must also have the item $[B \rightarrow \beta.a\gamma, c]$ (the lookahead need not be b in $s1'$ - it may be b in some other state, but that is not of interest to us)
  - These two items in $s1'$ still create an S-R conflict in the LR(1) parser
  - Thus, merging of states with common core can never introduce a new S-R conflict, because shift depends only on core, not on lookahead

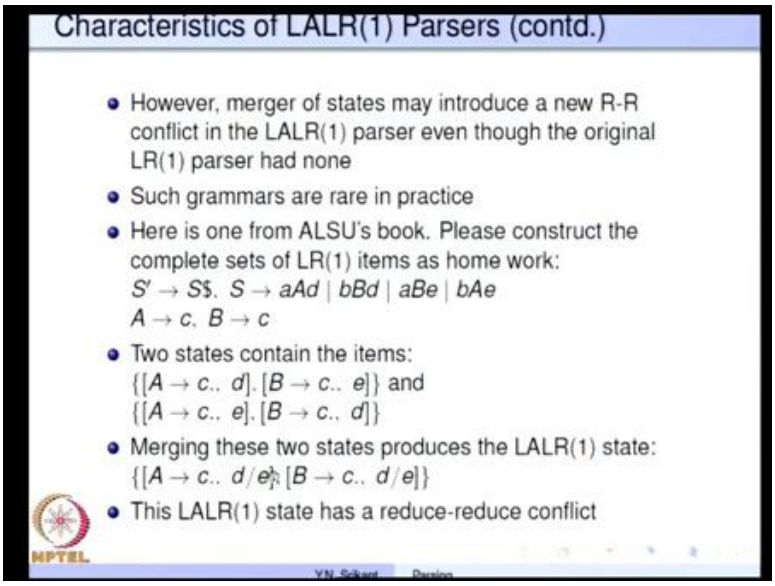
Y.N. Srikant Parsing

So, now let us look at the characteristics of LALR 1 parsers, the most important property is if a LR 1 parser has no shift reduce conflicts, then the derived LALR 1 parser will also have none, this is actually quite easy to prove. So, let us go through the simple proof here, both LR 1 and LALR 1 parser states have the same core items or cornel, look ahead's of course, may not be the same in each of the LR 1 states, but the cornel will be the same we merge them.

So, if a LALR 1 parser state s 1 has a shift reduce conflict, then it would have a reduce item and a shift item, the reduction also happens on a the shift also happens on a. So, if these two items are actually in the state s 1, they would have actually come to state s 1, because of let us say some other state s 1 prime. So, in the s 1 prime would have the same core as that is a going to alpha dot and b going to beta dot a gamma, but with different possible look ahead's.

So, this is the possible in this state s 1 prime, so if state s 1 prime had a going to alpha dot comma a and it also must have another item B going to beta dot a comma, but with a different look ahead c. So, if this happens this state s 1 prime also had exactly the same shift reduce conflict in the LR 1 parser, so if the LALR 1 parser has a shift reduce conflict, the LR 1 parser would also had the same shift reduce conflict. So, merging states to produce a compact LALR 1 parser will not increase a number of shift reduce conflicts.

(Refer Slide Time: 21:31)



Characteristics of LALR(1) Parsers (contd.)

- However, merger of states may introduce a new R-R conflict in the LALR(1) parser even though the original LR(1) parser had none
- Such grammars are rare in practice
- Here is one from ALSU's book. Please construct the complete sets of LR(1) items as home work:
  $S' \rightarrow S\$,\ S \rightarrow aAd \mid bBd \mid aBe \mid bAe$
  $A \rightarrow c,\ B \rightarrow c$
- Two states contain the items:
  $\{[A \rightarrow c., d], [B \rightarrow c., e]\}$ and
  $\{[A \rightarrow c., e], [B \rightarrow c., d]\}$
- Merging these two states produces the LALR(1) state:
  $\{[A \rightarrow c., d/e], [B \rightarrow c., d/e]\}$
- This LALR(1) state has a reduce-reduce conflict

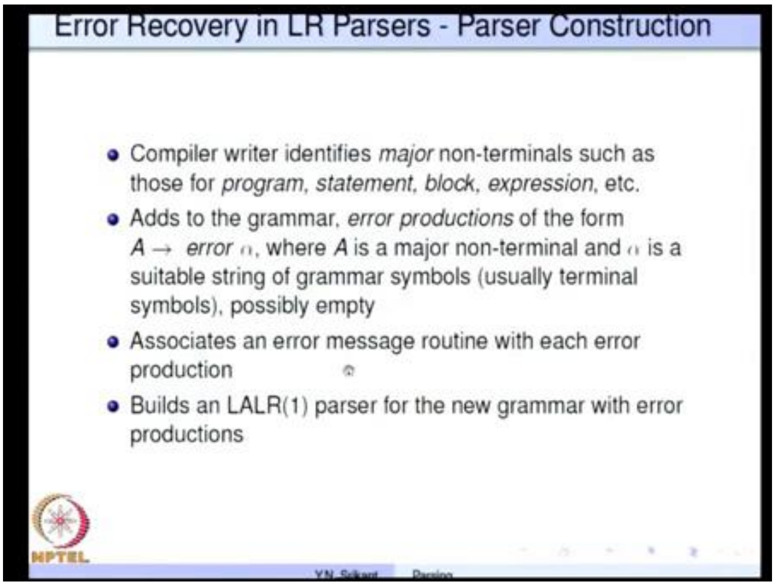
Y.N. Srikant Parsing

But, this is not, so as for the reduce reduce conflicts are concerned, the LALR 1 parser may have a reduce reduce conflicts, even if the original LR 1 parser had none. So, this is you know the point, so if this was also not true then the LALR 1 and LR 1 parsers would have had the same power almost. But because LALR 1 is weaker it is possible that LR 1 parser is clean, but the LALR 1 parser has reduce reduce conflicts, fortunately such grammars are very, very rear.

And it is very difficult to find practical grammars, which have reduce reduce conflicts in the LR 1 parser. So, let me take an example from the dragon book, you know constructing the sets of items LR 1 sets of items for this grammar is an left as an exercise, when you actually construct the LR 1 sets of items, you get two states one containing A to c dot comma d, B to c dot comma e. The other one containing A to c dot comma e and B to c dot comma d, so the cornel is the same, the core is the same, so when you merge you get A to c dot comma d slash e and B to c dot comma d slash e.

So, this parser has you know reduce reduce conflict, this also says reduce, this also says reduce and the look ahead is also identical. So, this is a an example of the LALR 1 parser having a introducing, you know a reduce reduce conflict, but the original LR 1 parser really had none.

(Refer Slide Time: 24:20)



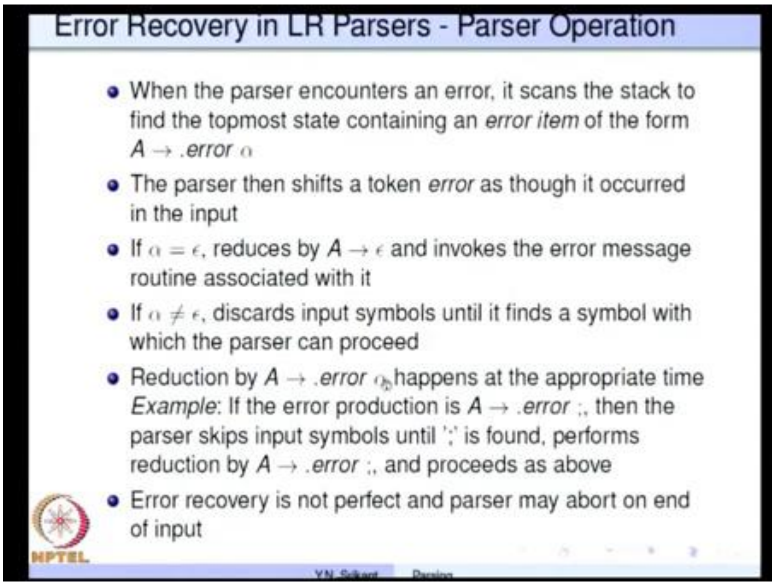
Error Recovery in LR Parsers - Parser Construction

- Compiler writer identifies *major* non-terminals such as those for *program*, *statement*, *block*, *expression*, etc.
- Adds to the grammar, *error productions* of the form $A \rightarrow$ *error* α, where A is a major non-terminal and α is a suitable string of grammar symbols (usually terminal symbols), possibly empty
- Associates an error message routine with each error production
- Builds an LALR(1) parser for the new grammar with error productions

So, then let us move on to error recovery in LR parsers, so error recovery very sophisticated error recovery is very difficult in LR parsers. So, basically this is a practical approach, the compiler writer identifies what are known as major non terminals, corresponding to the important construction a programming language. Such as, the non terminal program, non terminal statement, non terminal block, expression generating the important program constructs in the language.

And then for each of these major non terminals, error productions are added to the grammar. So, each error production is of the form A going to error alpha, where A is a major non terminal and alpha is a suitable string of grammar symbols, usually terminal symbols. Now, for each of these error productions there is also an error message routine, which print outs a suitable error and now after this modification of the grammar build the LALR 1 parser and go ahead with the parser operation.

(Refer Slide Time: 25:35)



Error Recovery in LR Parsers - Parser Operation

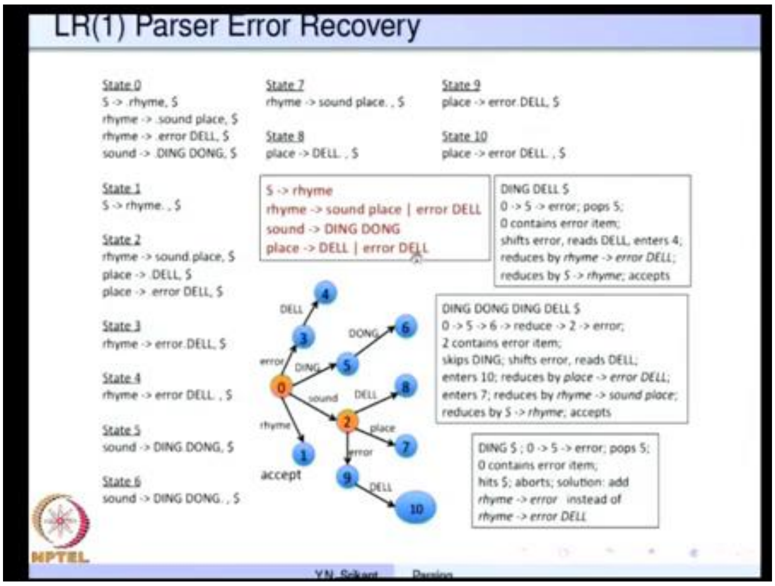
- When the parser encounters an error, it scans the stack to find the topmost state containing an *error item* of the form $A \rightarrow .error\ \alpha$
- The parser then shifts a token *error* as though it occurred in the input
- If $\alpha = \epsilon$, reduces by $A \rightarrow \epsilon$ and invokes the error message routine associated with it
- If $\alpha \neq \epsilon$, discards input symbols until it finds a symbol with which the parser can proceed
- Reduction by $A \rightarrow .error\ \alpha$ happens at the appropriate time *Example*: If the error production is $A \rightarrow .error\ ;$, then the parser skips input symbols until ';' is found, performs reduction by $A \rightarrow .error\ ;$, and proceeds as above
- Error recovery is not perfect and parser may abort on end of input

So, when the parser encounters an error, it really is you know cannot precedes further by shifting or reducing any items it can do, so only up to a particular point. So, then it stops, it now goes down the stack starting from the top and tries to find and item called an error item of the form A going to dot error alpha. So, that would be the first thing, then the parser shifts a token error as though it occurred in the input because this is the special token error.

So, there is a dot error alpha item, so; that means, the next input that is excepted is the token error. So, even though the error token does not occur in the input in the real case, it assumes that the token error has occurred and shifts it on to the stack, so if alpha is epsilon then; obviously, the production item would be of the form A going to dot error. So, a reduction by A to epsilon is called for, so it does that and then prints out the error message associated with it.

If alpha is not epsilon, then there must be a string here assume that these are all terminals symbols in alpha. So, it goes on discarding input symbols until it finds a symbol with which can proceed, so then the it finds all the symbols in alpha, so for example, if the item is A going to dot error semicolon, it skips all you know input symbols until it finds this semicolon. Then shifts the semicolon on to the stack reduces by the production A going to errors semicolon and then proceeds, so this is not perfect, but and the parser may also abort if it just gets the end of input.

(Refer Slide Time: 27:44)



LR(1) Parser Error Recovery

State 0
S -> .rhyme, $
rhyme -> .sound place, $
rhyme -> .error DELL, $
sound -> .DING DONG, $

State 1
S -> rhyme. , $

State 2
rhyme -> sound.place, $
place -> .DELL, $
place -> .error DELL, $

State 3
rhyme -> error.DELL, $

State 4
rhyme -> error DELL. , $

State 5
sound -> DING.DONG, $

State 6
sound -> DING DONG. , $

State 7
rhyme -> sound place. , $

State 8
place -> DELL. , $

State 9
place -> error.DELL, $

State 10
place -> error DELL. , $

S -> rhyme
rhyme -> sound place | error DELL
sound -> DING DONG
place -> DELL | error DELL

DING DELL $
0 -> 5 -> error; pops 5;
0 contains error item;
shifts error, reads DELL, enters 4;
reduces by *rhyme -> error DELL*;
reduces by S -> *rhyme*; accepts

DING DONG DING DELL $
0 -> 5 -> 6 -> reduce -> 2 -> error;
2 contains error item;
skips DING; shifts error, reads DELL;
enters 10; reduces by *place -> error DELL*;
enters 7; reduces by *rhyme -> sound place*;
reduces by S -> *rhyme*; accepts

DING $; 0 -> 5 -> error; pops 5;
0 contains error item;
hits $; aborts; solution: add
rhyme -> error instead of
rhyme -> error DELL

So, let me show you an example, so here is the a simple grammar s going to rhyme, rhyme going to sound place or we have also added the error production error dell, sound going to ding dong, place going to dell and another error production error dell. So, we constructs the LR 1 sets of items here and let us go through the error recovery procedure in this particular parser. So, the input is ding dell; obviously, this is erroneous, the correct input you know is not shown here because it any ways works well.

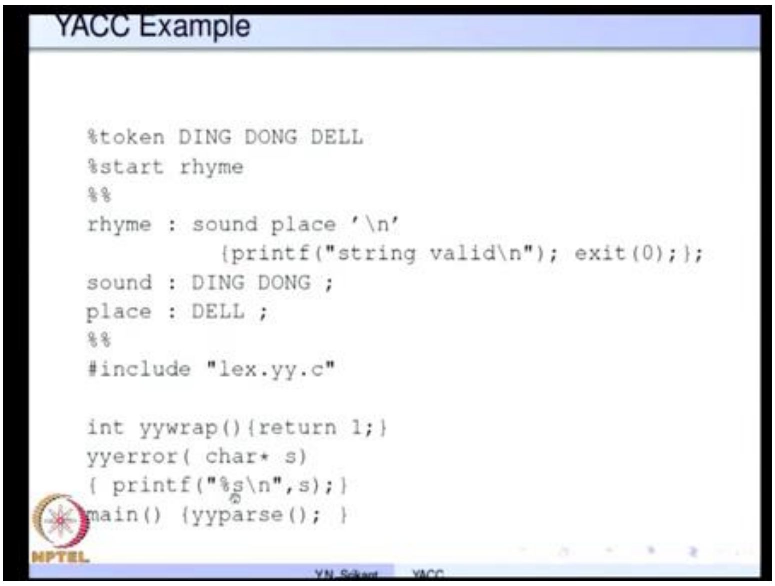
So, if there is an error here first a treats ding, so ding really takes it to state number 5 and then instead of dong there is dell. So, if there is error and it pops the state number 5 because state number 5 does not have the error item, it goes back to state 0, which has an error item. So, now it shifts error once it shifts error the next input symbol is dell, so it reads dell as the part of this, so if once it shifts error it goes to state 4 reads dell and then reduces by rhyme going to error dell.

Now, it is time for reduction by s going to rhyme and it accepts, so whereas, ding dong ding dell. So, it goes up to the second ding, you know then there is an error, so it has gone to state 2, fortunately two has an error item, you can see that here, so dot error dell. So, it shifts you know error item on to the stack, then ding is illegal, so it ignores ding reads dell and error has it enters state number 10 here. Now, reduce by place 2 error dell, enters state number 7, now it reduces by you know this rhyme going to sound place and finally, it reduces by s 2 rhyme and accepts.

Similarly here also, so here instead of any accept it has recovered and accepted the string the what is has really done is it has skip ding. And here it has assume the presence of dong whereas, in the case of just one ding and nothing else, it tries to pop 5 goes to 0 which contains an error items, but now the input is dollar. So, the parser aborts, so nothing can be done, it is possible to insert you know the error production has rhyme going to error, instead of rhyme going to error dell and then this error will go away.

But, that does not mean the error recovery in other cases will be good, so this the problem with error recovery, it can do a little bit and not everything, so let us move on to you know a practical LR parser generator called YACC.

(Refer Slide Time: 31:13)



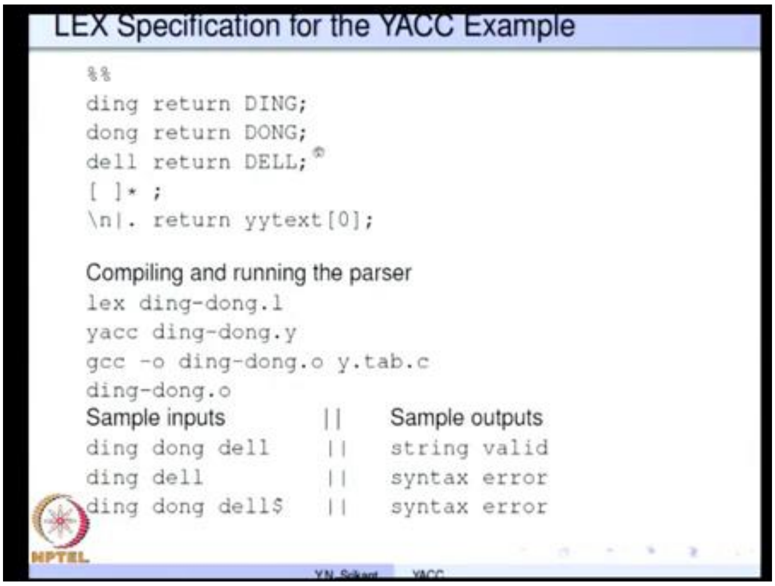
YACC Example

```
%token DING DONG DELL
%start rhyme
%%
rhyme : sound place '\n'
          {printf("string valid\n"); exit(0);};
sound : DING DONG ;
place : DELL ;
%%
#include "lex.yy.c"

int yywrap(){return 1;}
yyerror( char* s)
{ printf("%s\n",s);}
main() {yyparse(); }
```

Let me give you a simple example of how the YACC works, so just like YACC's the specification of YACC, you know is very similar. So, we actually describe all the tokens that appear in the grammar with a token directive, we show the start non terminal in start rhyme and then the productions of the grammar are return next. So, sound place, ding dong and dell, then there are some support routines which are necessary, so this is the basic structure of YACC.

(Refer Slide Time: 31:54)



LEX Specification for the YACC Example

```
%%
ding return DING;
dong return DONG;
dell return DELL;
[ ]* ;
\n|. return yytext[0];
```

Compiling and running the parser

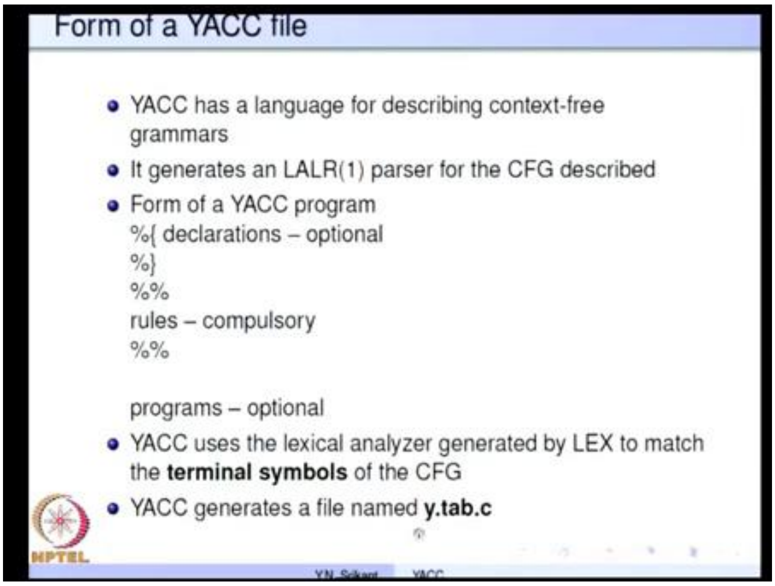
```
lex ding-dong.l
yacc ding-dong.y
gcc -o ding-dong.o y.tab.c
ding-dong.o
```

```
Sample inputs          ||    Sample outputs
ding dong dell         ||    string valid
ding dell              ||    syntax error
ding dong dell$        ||    syntax error
```

The corresponding lex specification for this YACC example, it is says ding dong and dell are the three tokens that we defined. For each of these it returns the capital ding dong and dell token as such, now blanks are all skipped, new line and any other character we just return y y text 0. So, that the parser can actually stop, so sound place new line it actually reduces and stops, compiling and running the parser is quite straightforward. So, you do a lex, which we know very well ding dong dot l do a YACC on ding dong dot y.

And then compile the file, which is produced in the case of you know l l the lexical analyzer, ((Refer Time: 32:50)) the lex dot y y dot c was the file produced and here it is y dot tab dot c and then we run the parser. So, here are the sample inputs ding dong dell is correct, so it prints string value, ding dell and ding dong dell dollar are all both illegal, so there is syntax error printed out.

(Refer Slide Time: 33:18)



Form of a YACC file

- YACC has a language for describing context-free grammars
- It generates an LALR(1) parser for the CFG described
- Form of a YACC program

```
%{ declarations – optional
%}
%%
rules – compulsory
%%

programs – optional
```

- YACC uses the lexical analyzer generated by LEX to match the **terminal symbols** of the CFG
- YACC generates a file named **y.tab.c**

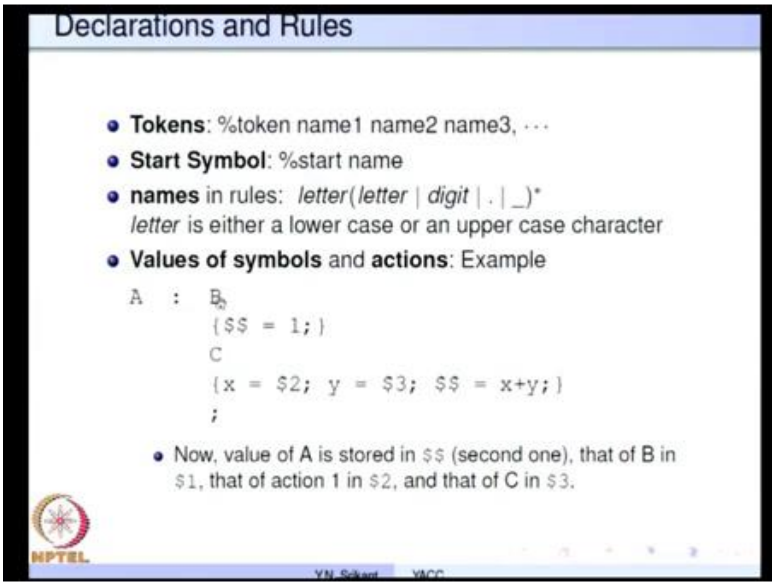
YN Srikant YACC

So, let us now go into details of the parser generator, so the YACC has a language for describing context free grammars, just has lex has a language for describing regular expressions. So, it generates an LALR 1 parser for the context free grammar, which it is described in side the file, the form of a YACC specification or YACC program, so you have percent and flower bracket, percent and flower bracket, inside this there are declarations, these are c declarations we are just copied into the output.

Then, there is a marker percent percent and another marker percent percent, so within this there are context free grammar rules and this is the compulsory part of the YACC specifications. After that there are several you know programs that is function, supporting functions etcetera which are all optional, but usually we require at least one or two to make the parser one. Now, the YACC does not define any you know regular expressions, it defines only context free grammar rules.

So, it uses the lexical analyzer generated by lex to match the terminal symbols of the context free grammar. So, we need to provide a lex specification and a YACC speciation has I showed you just now in order to make a parser and the parser generator YACC generates y dot tab dot c.

(Refer Slide Time: 34:50)



Declarations and Rules

- **Tokens**: %token name1 name2 name3, · · ·
- **Start Symbol**: %start name
- **names** in rules: *letter*(*letter* | *digit* | . | _)*
  letter is either a lower case or an upper case character
- **Values of symbols** and **actions**: Example

```
A   :   B
        {$$ = 1;}
        C
        {x = $2; y = $3; $$ = x+y;}
        ;
```

  - Now, value of A is stored in $$ (second one), that of B in $1, that of action 1 in $2, and that of C in $3.

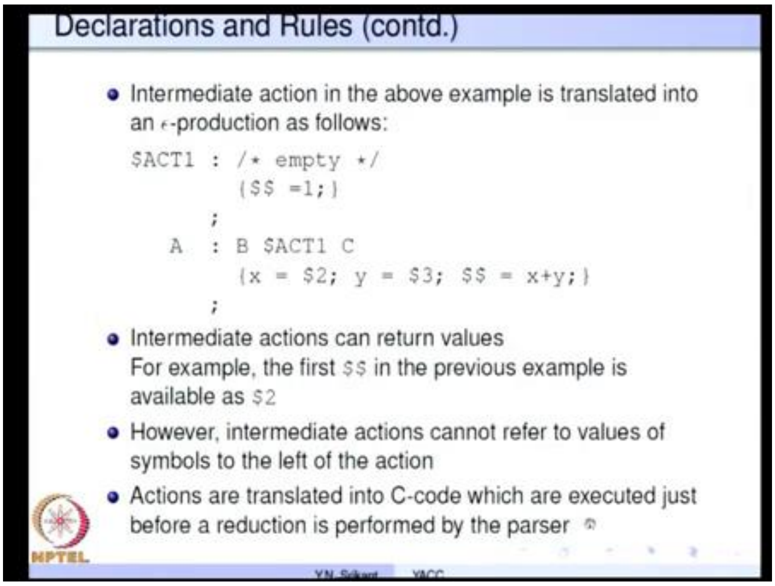
Y.N. Srikant YACC

So, let us look at the details of the specification, so tokens are defined as percent token name 1, name 2, name 3, etcetera. Start symbol is defined as percent start and the name of the non terminal, names in rules have the from letter, letter or digit or dot, underscore, star. So, this is how names of the non terminals can be and of course, letter is either lower case or upper case, so for example, A instead of write a arrow we have a colon here.

So, A colon B is a part of a rule, then we have this action and then another non terminal C another action followed by terminator of the production a semicolon. So, the production itself is A going to B C and there are some actions, which are interest first and mix with the right hand side. So, there are also values attached to the non terminals, so the left hand side non terminal has a value, which is denoted by dollar dollar, the right hand side non terminals have you know values dollar 1, dollar 2, dollar 3, etcetera, which are based on the position.

So, B is in the first position, so it is value will be available in dollar 1, let I will explain what these values are a little later, this is an action. So, it is value is available in dollar 2, C is the third symbol, so it is value will be available in dollar 3 and this is the fourth action. So, it theoretically it is available in dollar 4, but there is nothing more to use it, so this is how the specification of a rule would be, we are going to see more examples very soon. So, in this case there were actions which are in between, so each of these actions of course, context free grammar does not allow actions like this is directly.

(Refer Slide Time: 37:00)



Declarations and Rules (contd.)

- Intermediate action in the above example is translated into an ϵ-production as follows:

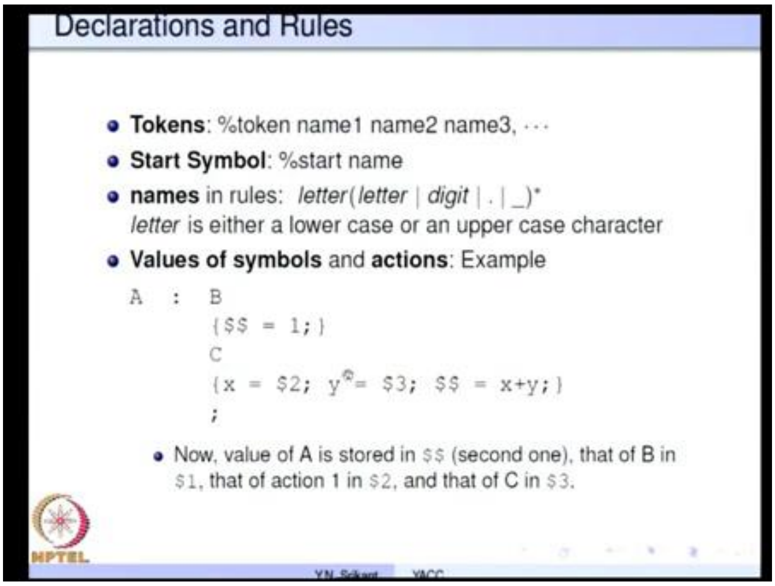
```
$ACT1 : /* empty */
          {$$ =1;}
        ;
    A   : B $ACT1 C
          {x = $2; y = $3; $$ = x+y;}
        ;
```

- Intermediate actions can return values
  For example, the first $$ in the previous example is available as $2
- However, intermediate actions cannot refer to values of symbols to the left of the action
- Actions are translated into C-code which are executed just before a reduction is performed by the parser

So, YACC really converts these actions into dummy productions going to Epsilon, so it introduces a new non terminal dollar act 1 to Epsilon and adds this, you know action to it. So, actions can be added only at the end of a production otherwise, so for example, now the production becomes A colon B dollar act 1 C, so this is a dummy non terminal produce by YACC and a action is at this end. So, now, A is dollar dollar B is dollar 1 dollar act 1 is dollar 2 and C is dollar 3.

So, what is the value that can be produced by dollar act 1 that is defined here, so this is dollar dollar and here this side there are no symbols. So, there are no values produced, so if you compute a value and assign it to dollar dollar, it will be available in the non terminal dollar act 1, when it is on the right hand side of some other production, this will become clearer as we go on. So, the intermediate actions cannot to refer to any values of symbols to the left of the action.

(Refer Slide Time: 38:17)



Declarations and Rules

- **Tokens**: %token name1 name2 name3, ···
- **Start Symbol**: %start name
- **names** in rules: *letter*(*letter* | *digit* | . | _)*
  letter is either a lower case or an upper case character
- **Values of symbols** and **actions**: Example

```
A  :  B
      {$$ = 1;}
      C
      {x = $2; y = $3; $$ = x+y;}
      ;
```

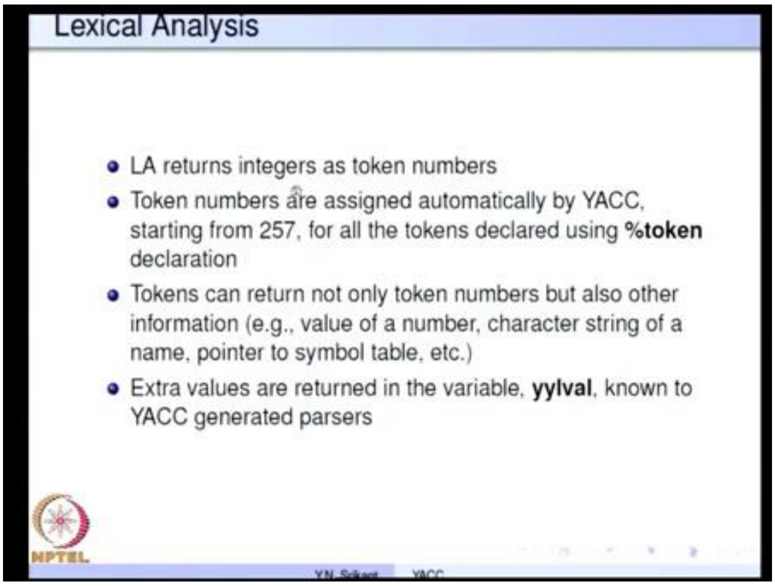
  - Now, value of A is stored in $$ (second one), that of B in $1, that of action 1 in $2, and that of C in $3.

Y.N. Srikant YACC

For example, here this action cannot use the values of B and this action cannot use the values, you know elsewhere and so on and so forth. So, this is at the end after the production is over, so a suppose you had another non terminal D, then this action could not have used the, you know values of B and C. But since this is the last action it can definitely use this values, which are available to the left of the production, left of the action.

So, dollar 2, dollar 3, etcetera can be used only when this is the last action in the whole production. ((Refer Time: 38:59)) Actions are translated into C code, which are executed just before a reduction is performed by the parser, this is extremely important, so the actions are performed only at the you know just before the reduction is performed by the parser.

(Refer Slide Time: 39:17)



Lexical Analysis

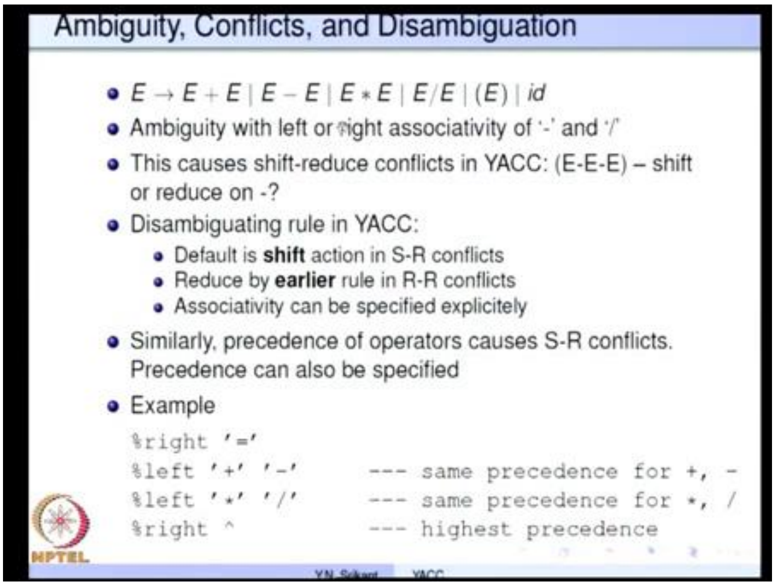
- LA returns integers as token numbers
- Token numbers are assigned automatically by YACC, starting from 257, for all the tokens declared using **%token** declaration
- Tokens can return not only token numbers but also other information (e.g., value of a number, character string of a name, pointer to symbol table, etc.)
- Extra values are returned in the variable, **yylval**, known to YACC generated parsers

So, now the connection between lexical analysis and parser, the lexical analyzer returns integers as token numbers, that is the normal case and the token numbers are assigned automatically by YACC starting from 257. So, for all the tokens declared using the percent token declaration, so if we provide a percent token declaration, YACC says now produce you know I am going to assume that the token numbers are 257 onwards. So, in the lexical analyzer we never define the token value as such, the parser defines it and the lex dot y y dot c will assume the values which are defined by the parser.

So, that is how the two interface, so remember in this simple specification ((Refer Time: 40:21)) here we had said hash include lex dot y y dot c. So, and we have defines the tokens ding dong and dell, but we never mention any value, so this is 257 258 into 259 and lex dot y y dot c will also use these ding dong and dell you can see that here. ((Refer Time: 40:40)) But we never define a value, automatically since the file lex dot y y dot c included in the parser, the definitions of ding dong and dell will be available to lex dot y y dot c.

So, then the tokens can return not only token numbers in the lex analyzer, but they can also return other information. For example, value of a number, character string of a name, pointer to a symbol table, etcetera, etcetera; obviously, these are all required the extra values are returned in the global variable y y l val, which is known to the YACC generated parsers, so we will see how this works very soon.

(Refer Slide Time: 41:27)



Ambiguity, Conflicts, and Disambiguation

- $E \rightarrow E + E \mid E - E \mid E * E \mid E / E \mid (E) \mid id$
- Ambiguity with left or right associativity of '-' and '/'
- This causes shift-reduce conflicts in YACC: (E-E-E) – shift or reduce on -?
- Disambiguating rule in YACC:
  - Default is **shift** action in S-R conflicts
  - Reduce by **earlier** rule in R-R conflicts
  - Associativity can be specified explicitely
- Similarly, precedence of operators causes S-R conflicts. Precedence can also be specified
- Example

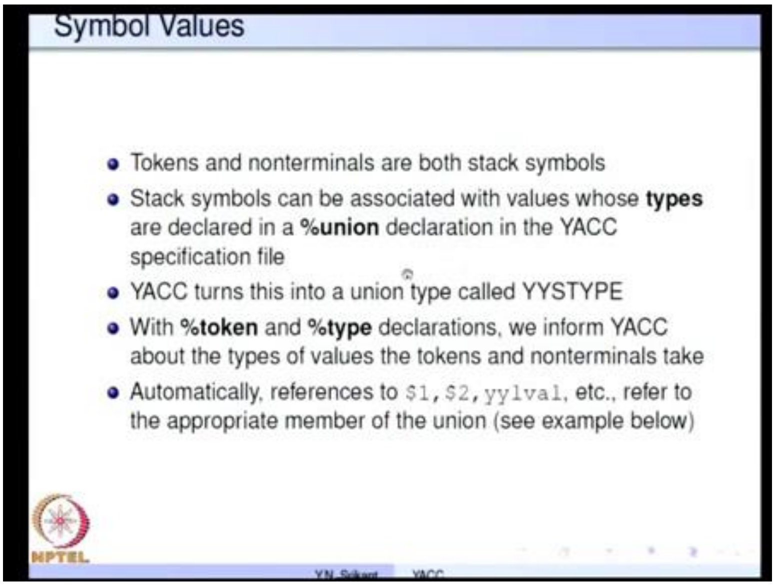
```
%right '='
%left '+' '-'       --- same precedence for +, -
%left '*' '/'       --- same precedence for *, /
%right ^            --- highest precedence
```

Then, how are ambiguity, you know conflicts, then disambiguation, etcetera carried out, so if you consider a grammar E going to E plus E E minus E E star E E slash E parenthesis E parenthesis i d, this a ambiguous. So, part from the problem with star and plus there is also a problem you know in plus and minus as well, for example, if you consider E minus E E minus E are even E plus E E plus E is this is a you know left associated operator or a right associated operator.

So, should we shift or reduce on minus this is a question, so even if you say E going to E minus E, then are we going to whether we expand this E or this E, we get the same E minus E minus E. So, with there is no indication of whether we should reduce or shift on this minus terminal, so this YACC gives you disambiguating rules, the default is a shift action, if there is a shift reduce conflict in the parser. And reduce by earlier rule in the case of reduce reduce conflicts and we can specify associativity and precedence explicitly.

For example, if we say percent right equal to then the equal to you know a token takes the associativity as right associative. Similarly, plus and minus become left associative and star and slash becomes left associative, exponentiation becomes right associative and the precedence we are going to increase the precedence as we go down words. So, plus and minus have the same precedence, but star and slash have higher precedence than plus and minus of course, exponentiation as the highest precedence equal to as the lowest precedence, so a precedence increases we go on down.

(Refer Slide Time: 43:48)



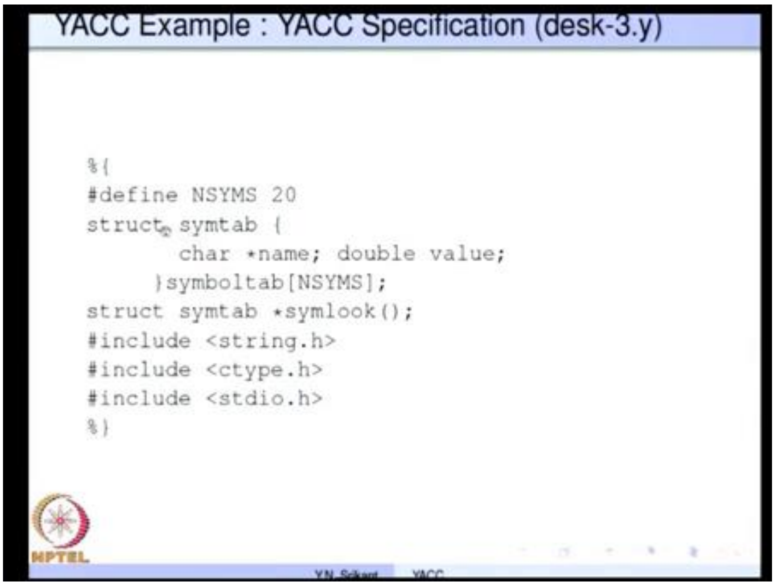
Symbol Values

- Tokens and nonterminals are both stack symbols
- Stack symbols can be associated with values whose **types** are declared in a **%union** declaration in the YACC specification file
- YACC turns this into a union type called YYSTYPE
- With **%token** and **%type** declarations, we inform YACC about the types of values the tokens and nonterminals take
- Automatically, references to `$1,$2,yylval`, etc., refer to the appropriate member of the union (see example below)

Y.N. Srikant YACC

Then, the symbol values, tokens and non terminals are both symbol stack symbols, so stack symbols can be associated with types and we are going to declare them in what is known as a percent union declaration.

(Refer Slide Time: 44:08)



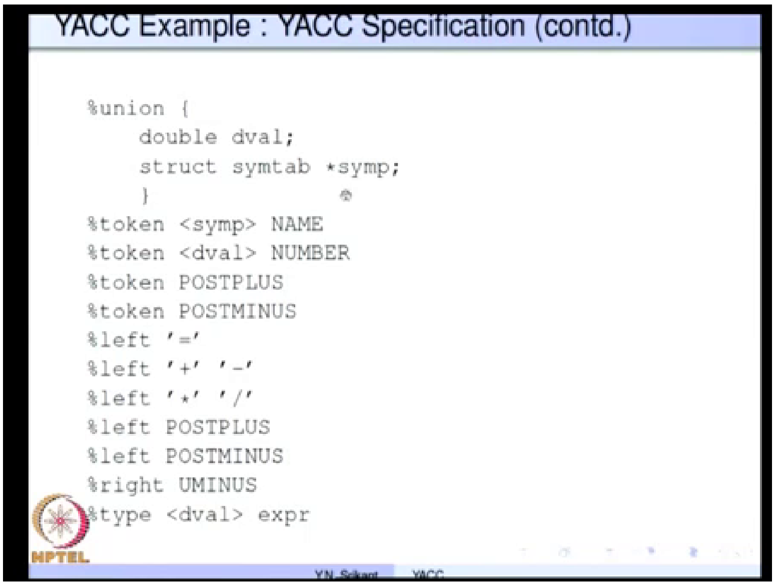
YACC Example : YACC Specification (desk-3.y)

```
%{
#define NSYMS 20
struct symtab {
       char *name; double value;
     }symboltab[NSYMS];
struct symtab *symlook();
#include <string.h>
#include <ctype.h>
#include <stdio.h>
%}
```

Y.N. Srikant YACC

So, let me show you what this is, so for example, this is a specification for a desk calculator. So, we have some of the program declarations here, the symbol table has 20 entry is possible, the structure of a symbol table is a structure with name, which is a pointer and you know character pointer and value which is called as double and the symbol table itself is a array of NSYMS and number of entries. So, the symlook is a function, which returns a pointer into the symbol table symtab star and there are of course, some library inclusions as well.

(Refer Slide Time: 44:56)



YACC Example : YACC Specification (contd.)

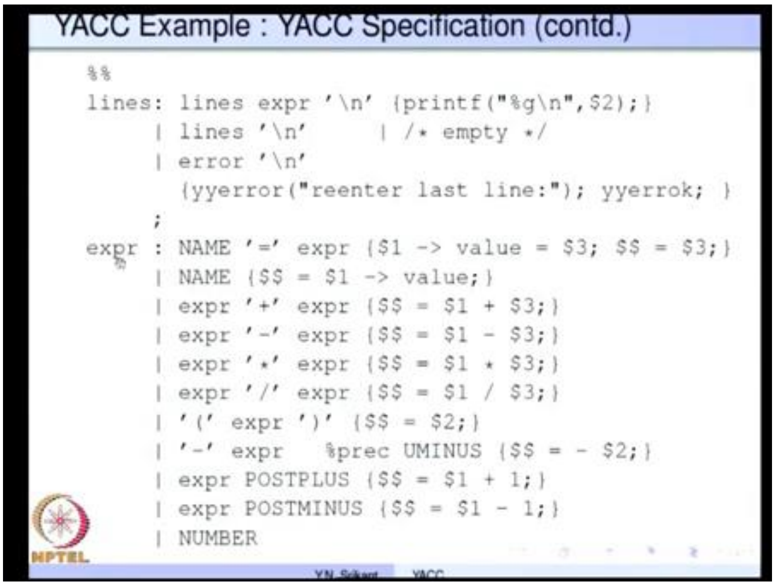
```
%union {
    double dval;
    struct symtab *symp;
    }
%token <symp> NAME
%token <dval> NUMBER
%token POSTPLUS
%token POSTMINUS
%left ’=’
%left ’+’ ’-’
%left ’*’ ’/’
%left POSTPLUS
%left POSTMINUS
%right UMINUS
%type <dval> expr
```

Y.N. Srikant YACC

Here is the percent union declaration, so double dval and struct symtab star symp, now the token name takes a value symp. So; that means, it is a pointer to the symbol table whereas, the token number takes a value dval, which is a double, so this is a way we are actually going to mention the values taken by the tokens, in the union declaration. And then refer them in the token declaration, these are not the only once which can get the values.

So, and then we have many of these tokens POSTPLUS, POSTMINUS, equal to plus minus, star slash, POSTPLUS, POSTMINUS and then percent right UMINUS. Here is the non terminal expression and we use a type declaration with union number dval, so; that means, expressions have values of type double that is the indication.

(Refer Slide Time: 46:04)



YACC Example : YACC Specification (contd.)

```
%%
lines: lines expr '\n' {printf("%g\n",$2);}
     | lines '\n'     | /* empty */
     | error '\n'
       {yyerror("reenter last line:"); yyerrok; }
     ;
expr : NAME '=' expr {$1 -> value = $3; $$ = $3;}
     | NAME {$$ = $1 -> value;}
     | expr '+' expr {$$ = $1 + $3;}
     | expr '-' expr {$$ = $1 - $3;}
     | expr '*' expr {$$ = $1 * $3;}
     | expr '/' expr {$$ = $1 / $3;}
     | '(' expr ')' {$$ = $2;}
     | '-' expr   %prec UMINUS {$$ = - $2;}
     | expr POSTPLUS {$$ = $1 + 1;}
     | expr POSTMINUS {$$ = $1 - 1;}
     | NUMBER
```

So, here is the context free grammar along with some of the actions, so let us look at the most important part of it first. So, here we have the production expression going to name equal to expression, so this desk calculator allows plus minus star slash and then parenthesis expressions of course, then the negation of an expression. It allows a post plus and post minus operation on expressions, it allows only numbers of course, it allows expression values to be placed in the symbol table and then used.

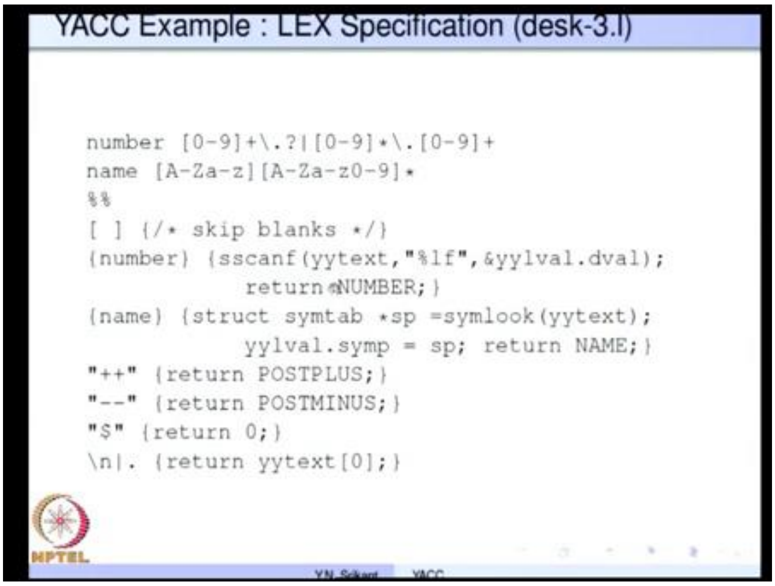
So, in each of these expressions we can actually use a name as well, so expression going to name is possible, if the name is already defined in the symbol table then that value is extracted and used here. If the name is not defined in the symbol table then of course, one could give an error I am not giving an error here, it just assume a value 0 and place it in the symbol table on the first occurrence. So, what about the computations here, let us look at these four computations first, expression going to expression plus expression.

So, intuitively it is clear that the value of this expression, which is bigger than expression plus expression should be the some of these two. So, dollar dollar equal to dollar 1 plus dollar 3 this is dollar 1, this is dollar 2, this is dollar 3, similarly for minus star and slash as well. So, we compute it using the usual arithmetic operators, the perennisation does nothing, so the value is just copied the negation of course, negates the value of the expression, post plus adds one and post minus subtracts one.

Number does nothing, the value of the number is available in y y l val and that would be copied to dollar dollar automatically. When we actually get to expression going to a name, the searching the symbol table to get the value is not denoted here, it is actually done by the lexical analyzer itself. So, I will show you that in a minute, the production name equal to expression says, whatever is the value of the expression should be assign to name. And again the introduction of the name into the symbol table etcetera is not shown here are the value of also.

It simply says, dollar 1 pointer value equal to dollar 3, dollar 1 is the pointer into the symbol table for name. So, here also dollar dollar equal to dollar 1 pointer value and dollar dollar is dollar 3, so dollar 3 is value of the expression that is passed on to the left non terminal.

(Refer Slide Time: 49:05)



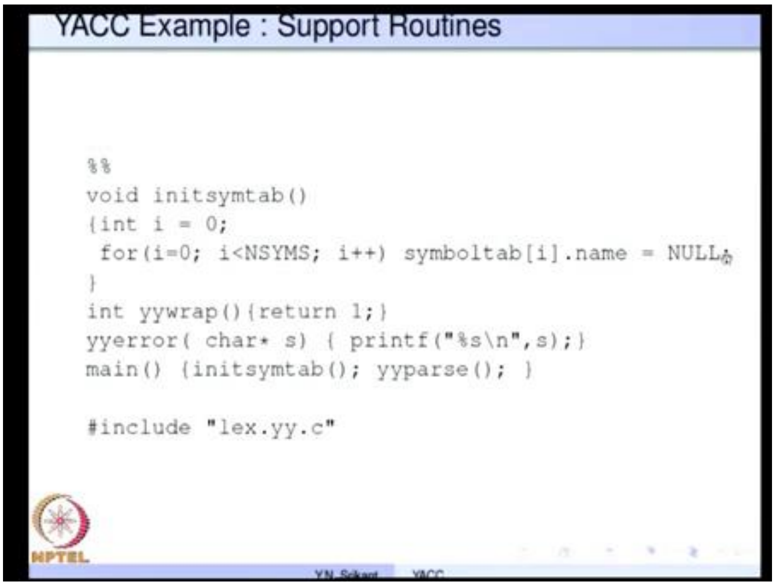
YACC Example : LEX Specification (desk-3.l)

```
number [0-9]+\.?|[0-9]*\.[0-9]+
name [A-Za-z][A-Za-z0-9]*
%%
[ ] {/* skip blanks */}
{number} {sscanf(yytext,"%lf",&yylval.dval);
            return NUMBER;}
{name} {struct symtab *sp =symlook(yytext);
            yylval.symp = sp; return NAME;}
"++" {return POSTPLUS;}
"--" {return POSTMINUS;}
"$" {return 0;}
\n|. {return yytext[0];}
```

So, now you can see in the lexical analyzer, whenever we find a number, which is nothing but what is described here. We read the token text, which is available in y y text using sscan f and place the value of that number in y y l val dot dval, so y y l val also takes both these values ((Refer Time: 49:35)) you know it is of this union type. So, it can either take dval or it can take symp both are possible, so in this case it takes the y y l val dot d val and behaves as if it is a double.

And then token returns is number, in the case of a name the symbol table actually the symbol table routine does a look up with this particular string which is nothing, but the name of the identifier. So, if it finds it, it returns a pointer to it, otherwise it inserts it and then returns a pointer to it, so y y l val dot symp is the pointer into the symbol table, so here is the assignment and returns a number, the rest are quite straightforward.

(Refer Slide Time: 50:25)



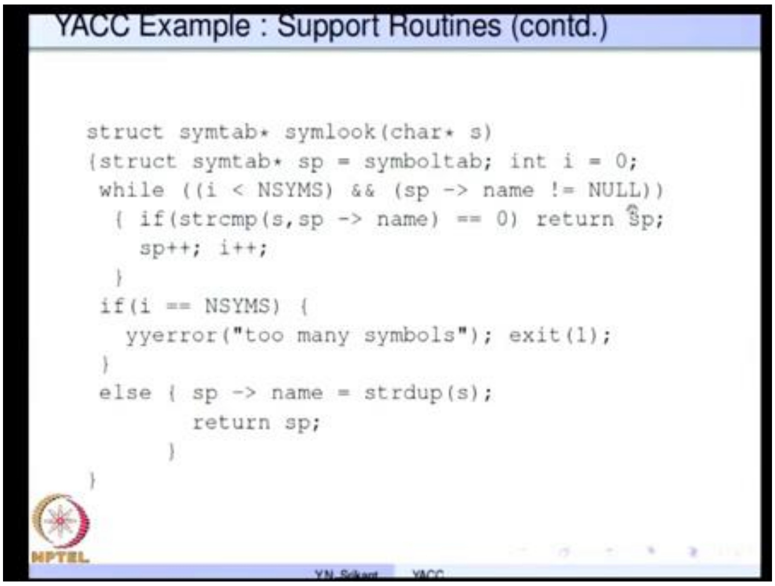
YACC Example : Support Routines

```
%%
void initsymtab()
{int i = 0;
 for(i=0; i<NSYMS; i++) symboltab[i].name = NULL;
}
int yywrap(){return 1;}
yyerror( char* s) { printf("%s\n",s);}
main() {initsymtab(); yyparse(); }

#include "lex.yy.c"
```

Here is the initialization, so name equal to null.

(Refer Slide Time: 50:32)

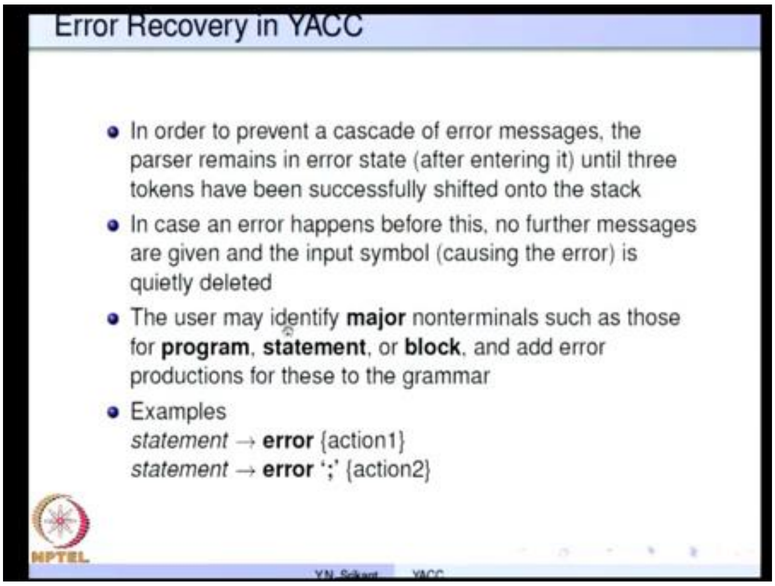


YACC Example : Support Routines (contd.)

```
struct symtab* symlook(char* s)
{struct symtab* sp = symboltab; int i = 0;
 while ((i < NSYMS) && (sp -> name != NULL))
  { if(strcmp(s,sp -> name) == 0) return sp;
    sp++; i++;
  }
 if(i == NSYMS) {
   yyerror("too many symbols"); exit(1);
 }
 else { sp -> name = strdup(s);
        return sp;
      }
}
```

And then here is the support routine, which inserts into the symbol table, so if the number of names in the symbol table is less than NSYMS and the name s p dot name not equal to null. That means, name is present in the symbol table and the name happens to be exactly the same as what we want, then it returns a pointer to the symbol table. Otherwise, it may come out of this loop the just because the number of symbols, number of names that can be put into the symbol table is two large. In which case it issues an error, otherwise it inserts the name into the symbol table and returns that particular pointer, so this is how the expression specification works.

(Refer Slide Time: 51:22)



Error Recovery in YACC

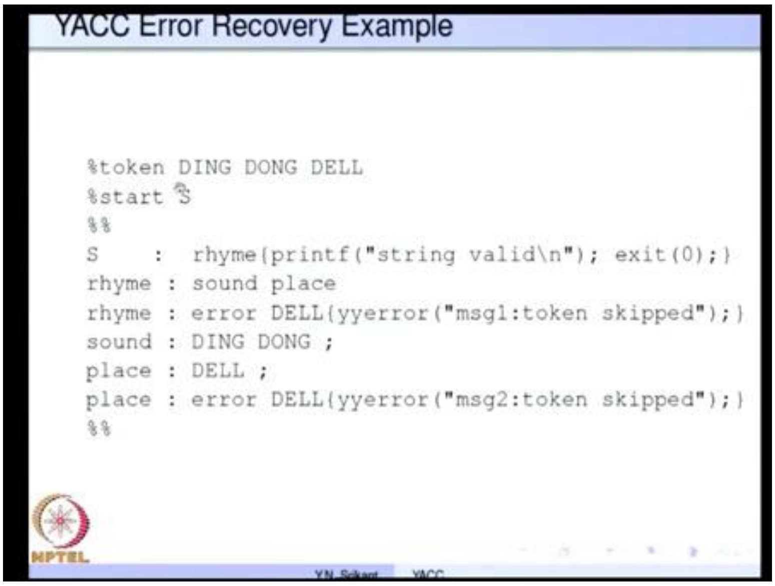
- In order to prevent a cascade of error messages, the parser remains in error state (after entering it) until three tokens have been successfully shifted onto the stack
- In case an error happens before this, no further messages are given and the input symbol (causing the error) is quietly deleted
- The user may identify **major** nonterminals such as those for **program**, **statement**, or **block**, and add error productions for these to the grammar
- Examples
  statement → **error** {action1}
  statement → **error** ';' {action2}

So, now let us look at the error recovery procedure in YACC, so in order to prevent a cascade of error messages, the parser really remains in error state, until three tokens have been successfully shifted on to the stack. So, the point is here is the it is an LALR 1 parser, so it actually you know goes on shifting and then reducing, enters some error state. Now, it will; obviously, try to do some error recovery, but until it has completely recovered from error and that is known only when a token is shifted successfully on to the stack.

Remember in an error state we never, you know shift items unnecessarily on to the stack, we must get out of the error state and only then we will be able to shift something on to the stack successfully. So, when it is in error state it goes on you know skipping symbols and until it handles three tokens successfully and shift them on to the stack, it will not you know giving you too many error messages. Because, the number of error messages otherwise would be just one too many, if an error happens before this no error further messages given and the input symbol is quietly deleted.

So, the user again it uses the major non terminal method, identifies major non terminals, programs, statement block, etcetera and adds you know error productions, so here state going to error, state going to error semicolon etcetera, etcetera.

(Refer Slide Time: 53:15)



YACC Error Recovery Example

```
%token DING DONG DELL
%start S
%%
S     :  rhyme{printf("string valid\n"); exit(0);}
rhyme : sound place
rhyme : error DELL{yyerror("msg1:token skipped");}
sound : DING DONG ;
place : DELL ;
place : error DELL{yyerror("msg2:token skipped");}
%%
```

Y.N. Srikant YACC

Here is an example of how it does, so this is percent token ding dong dell and the star is the non terminal s. So, for the production rhyme s going to rhyme there is no error message associated with it and there is no error production either, but for the production rhyme going to sound place, there is also an error production rhyme going to error dell. So, here there is an error message given as well y y error message 1 token skipped and for the production sound going to ding dong, there is no error production.

But, for the production place going to dell there is an error production place going to error dell. So, here the error message message 2 token skipped is issued, so this is the same you know examples as we saw in the case of the LR 1 error recovery and this is exactly what we had said as you know error productions etcetera. So, we can actually produce a grammar with error productions in this form in YACC, compile this grammar and then of course, run the parser associated with it. So, that gives you an error recovery parser as well, so thus now we come to the end of the lecture. So, we will continue with semantical analysis in the next lecture.

Thank you.