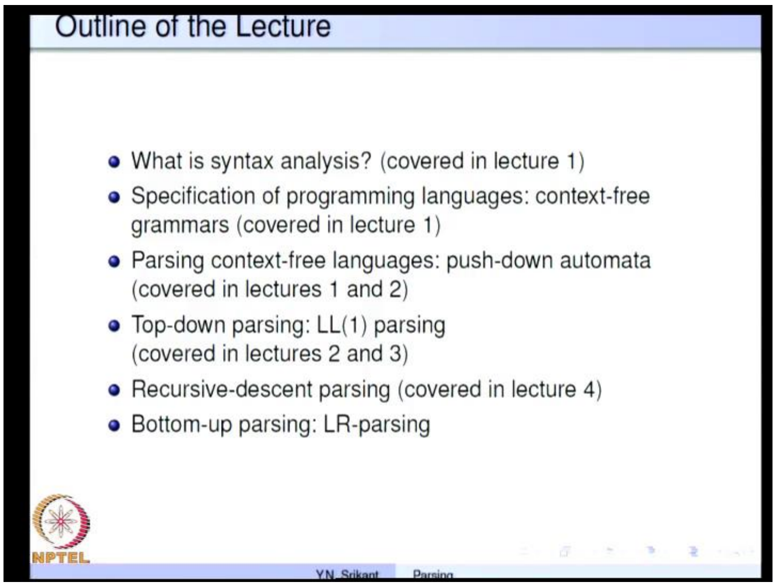
Principles of Compiler Design
Prof. Y.N. Srikant
Department of Computer Science and Automation
Indian Institute of Science, Bangalore

Lecture - 9
Syntax Analysis: Context-free Grammars, Pushdown Automata and Parsing Part 5

(Refer Slide Time: 00:23)



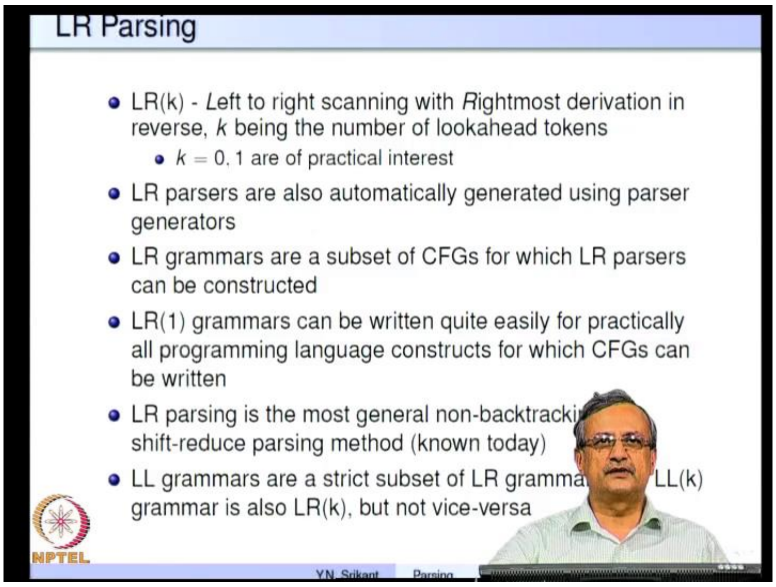
Outline of the Lecture

- 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

Y.N. Srikant Parsing

Welcome to part 5 of the lecture series on Syntax Analysis. So far we are have covered you know basics of syntax analysis and top down parsing, recursive descent parsing and a bit of bottom up parsing, let us a continue with L R parsing techniques today.

(Refer Slide Time: 00:37)



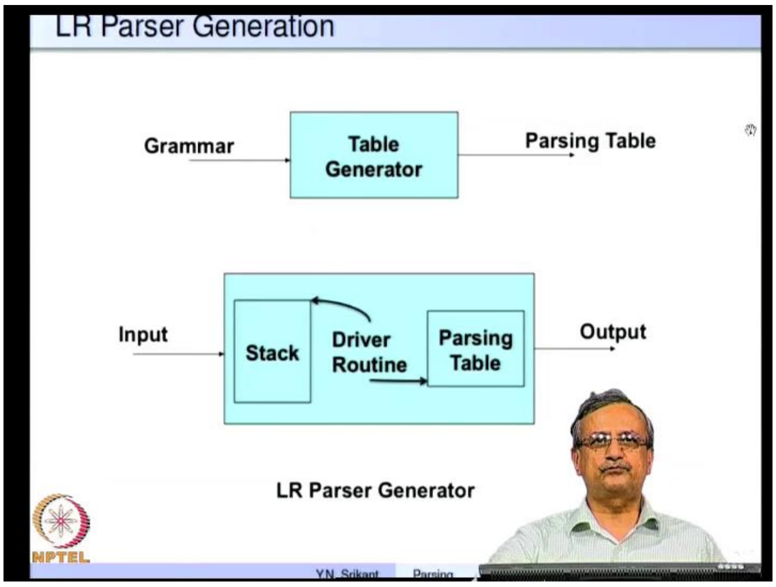
LR Parsing

- LR(k) - *L*eft to right scanning with *R*ightmost derivation in reverse, *k* being the number of lookahead tokens
  - *k* = 0, 1 are of practical interest
- LR parsers are also automatically generated using parser generators
- LR grammars are a subset of CFGs for which LR parsers can be constructed
- LR(1) grammars can be written quite easily for practically all programming language constructs for which CFGs can be written
- LR parsing is the most general non-backtracki shift-reduce parsing method (known today)
- LL grammars are a strict subset of LR gramma LL(k) grammar is also LR(k), but not vice-versa

Y.N. Srikant Parsing

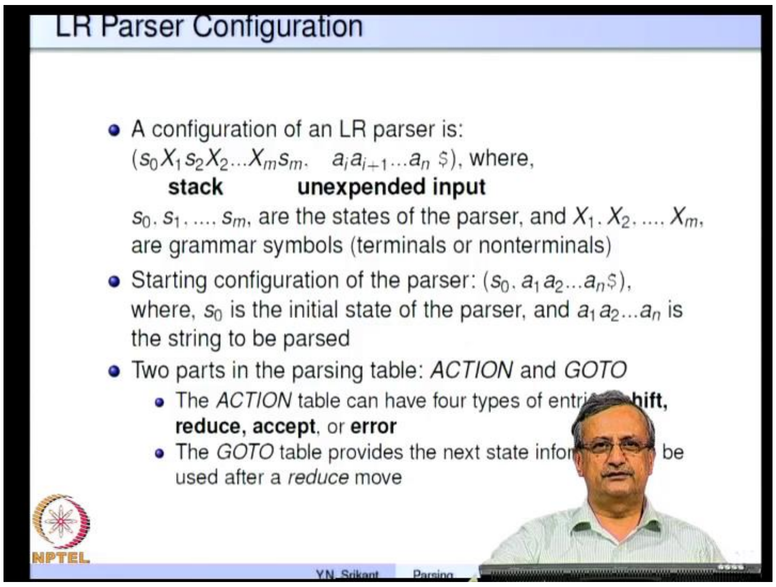
So, as I explained in the previous lecture, L R parsing is a method of bottom up parsing and it is stands for left to right scanning with rightmost derivation in reverse and k is the number of look ahead tokens. Of course, L R 0 and L R 1 are of great interest to us in practical sense, L R parsers of course, are important, because they can be generated automatically using parser generators and they are a subset of context free L R grammars are subset of context free grammars, for which such of parsers can be constructed. So, it is easy to write L R grammars and that is reason why there are very popular today.

(Refer Slide Time: 01:29)



So, let us look at the parser generator, the generator is a very simple device it takes grammar as input and generates a parsing table called the L R parsing table. And the parser table is fit into another box, containing a stack and a driver routine, this whole set of is the parser. So, here for example, so stack driver routine and parsing table together make the parser, it takes the program as input and delivers as output possibly a syntax free or something else.

(Refer Slide Time: 02:14)



LR Parser Configuration

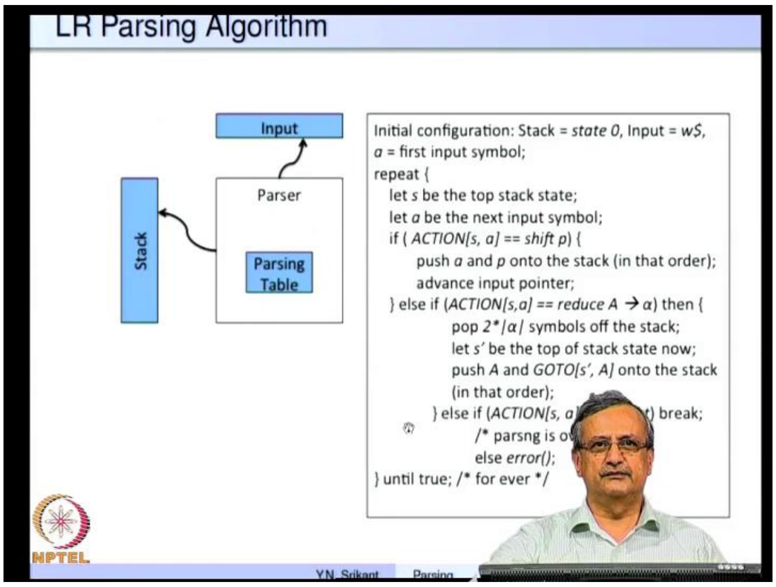
- A configuration of an LR parser is: $(\underbrace{s_0 X_1 s_2 X_2 \ldots X_m s_m}_{\textbf{stack}},\ \underbrace{a_i a_{i+1} \ldots a_n \$}_{\textbf{unexpended input}})$, where, $s_0, s_1, \ldots, s_m$, are the states of the parser, and $X_1, X_2, \ldots, X_m$, are grammar symbols (terminals or nonterminals)
- Starting configuration of the parser: $(s_0, a_1 a_2 \ldots a_n \$)$, where, s_0 is the initial state of the parser, and $a_1 a_2 \ldots a_n$ is the string to be parsed
- Two parts in the parsing table: *ACTION* and *GOTO*
  - The *ACTION* table can have four types of entri **hift, reduce, accept**, or **error**
  - The *GOTO* table provides the next state infor be used after a *reduce* move

Y.N. Srikant Parsing

So, let us look at the parser operation, so to understand it we need to know what exactly is a configuration of L R parser. The configuration has two parts, one is the stack, the other is the unexpended or unused input and to begin with the stack as only the start symbol or the initial state of the parser and the unexpanded input as the entire input, you know terminated with a end of file or dollar mark. So, somewhere in the middle a configuration will consist of a number of the states, inter mix with a grammar symbols and then the rest of the input.

The parsing table is a little more complex it has two parts, the action part and the GOTO part. The action part has four types of entry, shift, reduce, accept and error, the GOTO table is used to provide the next state information, which is actually necessary after a reduce move.

(Refer Slide Time: 03:29)

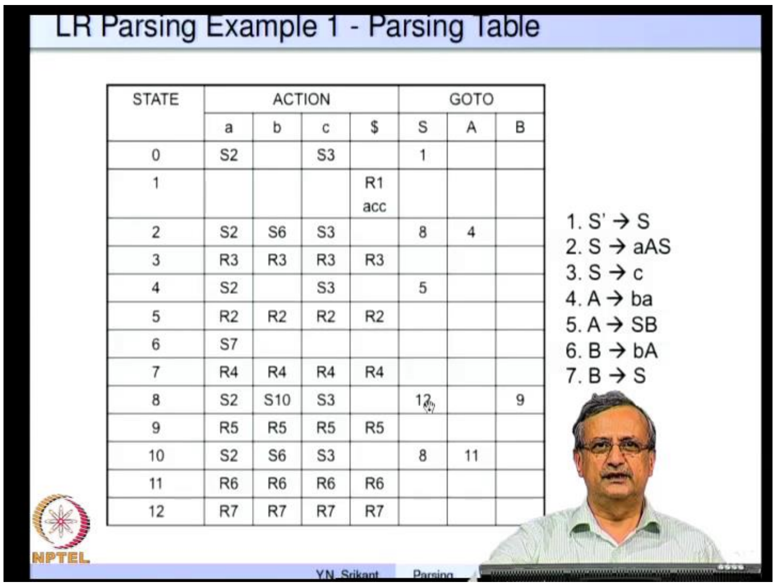


LR Parsing Algorithm

```
Initial configuration: Stack = state 0, Input = w$,
a = first input symbol;
repeat {
   let s be the top stack state;
   let a be the next input symbol;
   if ( ACTION[s, a] == shift p) {
         push a and p onto the stack (in that order);
         advance input pointer;
   } else if (ACTION[s,a] == reduce A → α) then {
              pop 2*|α| symbols off the stack;
              let s' be the top of stack state now;
              push A and GOTO[s', A] onto the stack
              (in that order);
         } else if (ACTION[s, a t) break;
                 /* parsng is o
                 else error();
} until true; /* for ever */
```

So, here is the parsing algorithm.

(Refer Slide Time: 03:36)



LR Parsing Example 1 - Parsing Table

STATE	ACTION				GOTO		
	a	b	c	$	S	A	B
0	S2		S3		1		
1				R1 acc			
2	S2	S6	S3		8	4	
3	R3	R3	R3	R3			
4	S2		S3		5		
5	R2	R2	R2	R2			
6	S7						
7	R4	R4	R4	R4			
8	S2	S10	S3		12		9
9	R5	R5	R5	R5			
10	S2	S6	S3		8	11	
11	R6	R6	R6	R6			
12	R7	R7	R7	R7			

1. $S' \rightarrow S$
2. $S \rightarrow aAS$
3. $S \rightarrow c$
4. $A \rightarrow ba$
5. $A \rightarrow SB$
6. $B \rightarrow bA$
7. $B \rightarrow S$

So, before that let us look at this parser table to understand the action and go to entry is little better. So, the parser table is indexed by the state numbers are one side and on the other side for the action table it is indexed by the tokens, present in the input and for the go to table it is indexed by the non terminals present in the grammar. Entries such as S 2 indicate that there is a shift operation and next state that we parser enters is 2, similarly S 3 indicates shift and the next state be in 3 and so on.

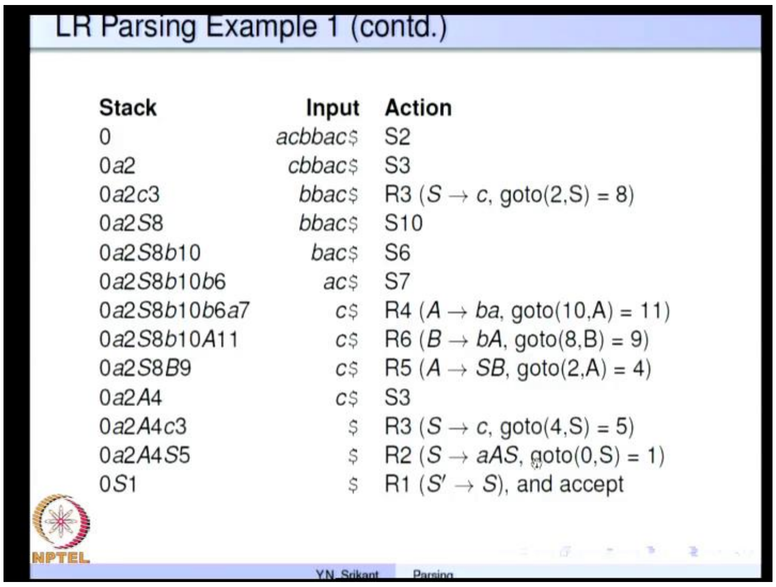
Entry such as R 3 indicate that the next move is a reduction move and the production number used is number 3. So, all the productions in the grammar or number sequentially and the production number tells you, which production is to be used for reduction, in this case R 3 is the production S to c and we use the production S to c and to reduction the perform reduction here. After the reduction is, but the during reduction some of the stack symbols are removed and we expose a state.

So, after the state is exposed the non terminal on the left hand side of the production used and the state combination looked up in the GOTO table, tells us which next state we need to GOTO. ((Refer Time: 05:11)) So, now let us look at the parsing algorithm, which basically explains the actions I was describing right now in more detail, the initial configuration of course, the stack has state 0 and the input is the entire input and let us say a is the next input symbol. So, now, there is a repeat until loop, which goes on forever unless it is interrupted by either error or accept action.

So, let S be the top of stack state and let a be the next input symbol, so now, as I said the parser looks up the action part, if the action part says shift p, then it pushes a and p onto the stack in that order. And the input pointer is advance to pick up the next input symbol, if the action part says reduce by a production, so this is the number that I indicated in the parser table. It pops off 2 into alpha symbols off the stack, the reason we are popping 2 times alpha length of alpha is the number of symbols on the right hand side of the production.

And we have the state symbol as well inter mix therefore, we need to remove to alpha symbols. The state S prime is exposed, now the left hand side A here and the GOTO off S prime comma A or pushed on to the stack, so this is a way we GOTO table is used, if the action is accept then its gets out of the infinite loop, otherwise there is an error and the error recovery routine is called.

(Refer Slide Time: 07:01)



LR Parsing Example 1 (contd.)

Stack	Input	Action
0	$acbbac\$$	S2
$0a2$	$cbbac\$$	S3
$0a2c3$	$bbac\$$	R3 ($S \to c$, goto(2,S) = 8)
$0a2S8$	$bbac\$$	S10
$0a2S8b10$	$bac\$$	S6
$0a2S8b10b6$	$ac\$$	S7
$0a2S8b10b6a7$	$c\$$	R4 ($A \to ba$, goto(10,A) = 11)
$0a2S8b10A11$	$c\$$	R6 ($B \to bA$, goto(8,B) = 9)
$0a2S8B9$	$c\$$	R5 ($A \to SB$, goto(2,A) = 4)
$0a2A4$	$c\$$	S3
$0a2A4c3$	$\$$	R3 ($S \to c$, goto(4,S) = 5)
$0a2A4S5$	$\$$	R2 ($S \to aAS$, goto(0,S) = 1)
$0S1$	$\$$	R1 ($S' \to S$), and accept

Y.N. Srikant Parsing

So, let us a look at an example to understand all the operations that I described just now, the stack contains 0, the state number and input is a c b b a c, the parser says shift. So, this is the parsing table that we have in mind and these are the productions that we are using, I have actually written down all the productions which are necessary for reduction etcetera in this slide itself. So, that we do not have to go back and forth, the entry is 0 comma a would tell us that is a S 2 action, so just for this let us look up 0 and a it indeed says S 2.

So, the symbol a shifted on to the stack along with the state 2, so this is the state right now the parser is in, so 2 and c the table would say S 3. So, we shift c and the next state 3 also on to the stack, so the combination of 3 and b is reduce by production 3 that is S to c. So, when we reduce we take out the 3 and c, this is the twice the length of the right hand side, so there is a one symbol is here, so we taking out the state symbol and the c itself.

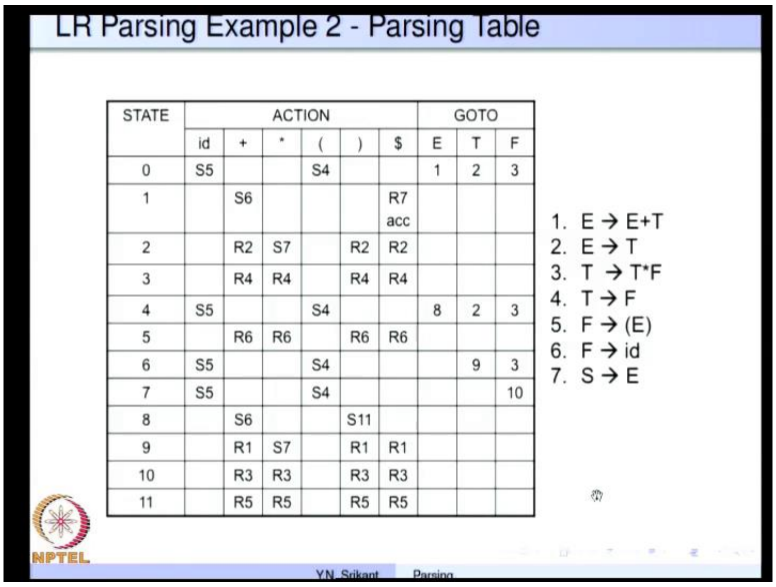
The state number 2 is exposed, so we lookup GOTO of 2 and the non terminal S here, this is will give us 8 in the table. So, the non terminal S and the state 8 the new state are pushed on to the stack, the combination of 8 and b says it is shift and next state is 10, so b and 10 go on to the stack, 10 and b say shift 6, so b and 6 go on to stack. Again it is shift time shift 7, so a and 7 also go on to the stack, now it is time for reduction by the production a to b a, so there are two symbols here.

So, we take out 4 symbols from the stack 7 a 6 and b, so now state 10 is exposed and that on the non terminal A, which is the left hand side of the production gives us state 11. So, we push the non terminal A and state 11 on to the stack, so here please observe that now this is the process of handle proven in, so we saw that in the shift reduce parsing algorithm as well. So, whenever there is a right hand side that is always at the top, so we remove the right hand side of the production and push the non terminal on to the stack.

But, as I explained there is a DFA whose states are being pushed on to the stack as well, so this is the state of the parser and the DFA. The state on the top tells us whether it is time for reduction or a shift, so for example, again 11 and c tell us that it is a reduction by 6, so B to b A takeout 4 symbols. So, 11 A 10 and b, so state 8 it again 8 and B give us state 9, so state 9 goes on to the stack along with B, B and c again is a reduction, so this time again remove 4 symbols.

So, state 2 is exposed 2 and A give us 4, so A and 4 go on to the stack 4 and c tell us it is time for shift. So, c 3 goes on to the stack again we reduce by S to c that gives us 4 S 5, again it is reduce by production number 2, so that expose state 0 and state 0 on S gives us state 1. So, 0 S 1 goes on to the stack and finally, 1 and dollar tell us that it is the start production plus accept action, so the whole process ends.

(Refer Slide Time: 11:04)



LR Parsing Example 2 - Parsing Table

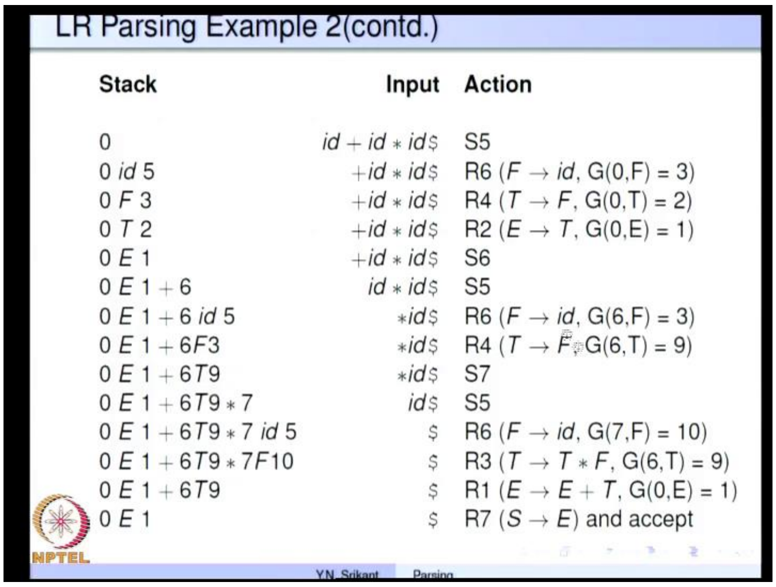
STATE	ACTION						GOTO		
	id	+	*	(	)	$	E	T	F
0	S5			S4			1	2	3
1		S6				R7 acc			
2		R2	S7		R2	R2			
3		R4	R4		R4	R4			
4	S5			S4			8	2	3
5		R6	R6		R6	R6			
6	S5			S4				9	3
7	S5			S4					10
8		S6			S11				
9		R1	S7		R1	R1			
10		R3	R3		R3	R3			
11		R5	R5		R5	R5			

1. $E \rightarrow E+T$
2. $E \rightarrow T$
3. $T \rightarrow T*F$
4. $T \rightarrow F$
5. $F \rightarrow (E)$
6. $F \rightarrow id$
7. $S \rightarrow E$

Y.N. Srikant Parsing

So, there is another example the familiar E to E plus T, E to T, T to T star F, T to F, F to parenthesis E parenthesis F to i d and S to E. So, here is the parsing table for this particular grammar, observe that this is the start symbol S.

(Refer Slide Time: 11:24)



LR Parsing Example 2(contd.)

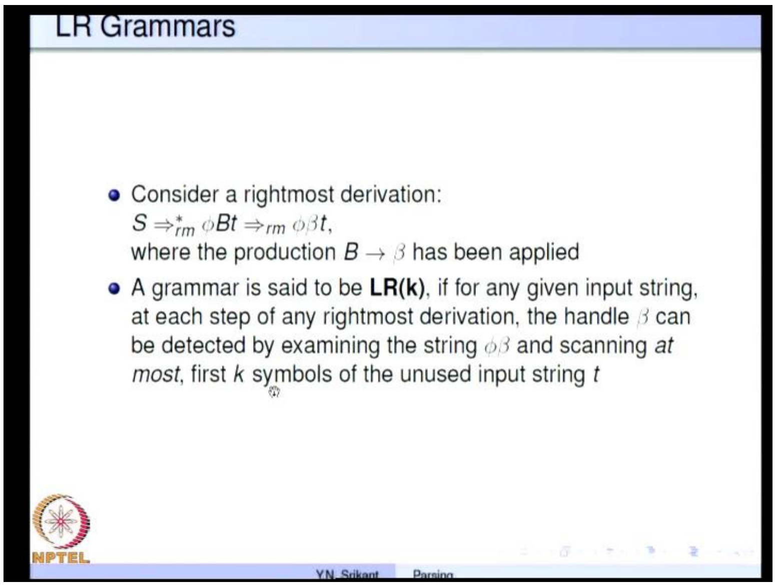
Stack	Input	Action
0	$id + id * id\$$	S5
0 id 5	$+id * id\$$	R6 ($F \rightarrow id$, G(0,F) = 3)
0 F 3	$+id * id\$$	R4 ($T \rightarrow F$, G(0,T) = 2)
0 T 2	$+id * id\$$	R2 ($E \rightarrow T$, G(0,E) = 1)
0 E 1	$+id * id\$$	S6
0 E 1 + 6	$id * id\$$	S5
0 E 1 + 6 id 5	$*id\$$	R6 ($F \rightarrow id$, G(6,F) = 3)
0 E 1 + 6F3	$*id\$$	R4 ($T \rightarrow F$, G(6,T) = 9)
0 E 1 + 6T9	$*id\$$	S7
0 E 1 + 6T9 * 7	$id\$$	S5
0 E 1 + 6T9 * 7 id 5	$\$$	R6 ($F \rightarrow id$, G(7,F) = 10)
0 E 1 + 6T9 * 7F10	$\$$	R3 ($T \rightarrow T * F$, G(6,T) = 9)
0 E 1 + 6T9	$\$$	R1 ($E \rightarrow E + T$, G(0,E) = 1)
0 E 1	$\$$	R7 ($S \rightarrow E$) and accept

Y.N. Srikant Parsing

Here is a simple example, let us quickly run through it to see what it says, so here the string is i d plus i d star dollar observe that this grammar is unambiguous, so it can be passed exactly in one way. So, on if a state 0 on i d is a shift action, then it is a reduce action and then again it is a reduce action, so finally, one more reduction and we get 0 even plus i d star i d dollar. So, there are two more shifts here and this is the stack at this point, now there is a reduce by 6, so production number 6 that is F to i d.

So, we reduce and push 6 F 3 on to the stack, then again there are two shifts and there is reduced by 6 5 on dollar is a reduce. So, if we reduce by F to i d and state 7 is exposed, so G of GOTO of 7 comma F is 10, so we push F 10 on to the stack, then that is time for reduction again another reduction and finally, accept. So, the L R parser is nothing, but a shift reduce parser as you know the actions are exactly the same, but at the time of reduction and push, we also push the state numbers on to the stack along with the symbols the either it terminal or non terminal symbols.

(Refer Slide Time: 12:59)



LR Grammars

- Consider a rightmost derivation: $S \Rightarrow^*_{rm} \phi B t \Rightarrow_{rm} \phi \beta t$, where the production $B \rightarrow \beta$ has been applied
- A grammar is said to be **LR(k)**, if for any given input string, at each step of any rightmost derivation, the handle β can be detected by examining the string $\phi\beta$ and scanning *at most*, first k symbols of the unused input string t

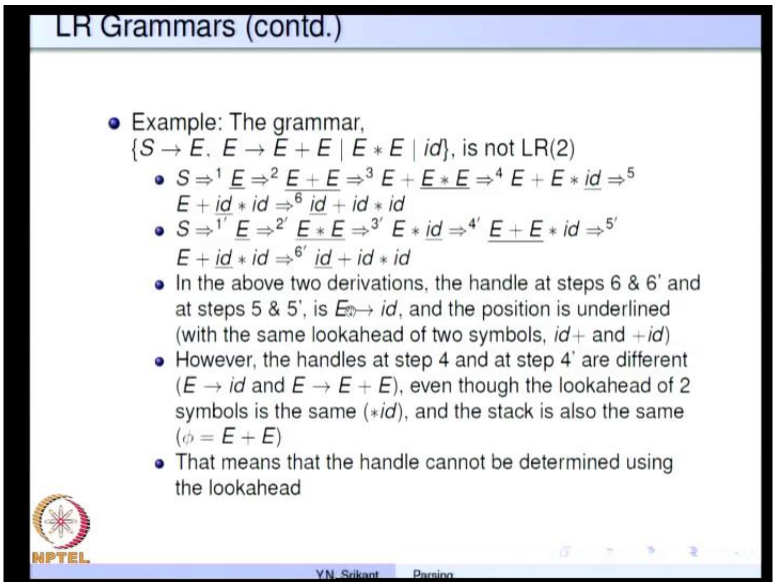
Y.N. Srikant Parsing

So, now it is time to understand how to build the parser table that we seen in the operation of the parser given the table, to do that we must understand what exactly makes a grammar an L R grammar. So, let us consider a right most derivation S derives in 0 or many steps phi B t, which in turn derives phi beta t, in other words in the last step B to beta is the production which has been applied. So, the basic idea in an L R grammar is we should be able to you know determine the handle uniquely, so beta is the handle here B to beta is the production.

So, we should be able to look at the first case symbols of this t in any derivation of a the grammar and determine, which production was applied at that particular point. So, a grammar is said to be L R k, if for any input string at each step of any rightmost derivation, the handle beta can be detected by examining the string phi beta and scanning at most first k symbols of the unused input string t. So, this phi beta is on the stack, so if we go back 1 step. ((Refer Time: 14:37))

So, this is the stack content and this is exactly what we mean by phi beta of course, the state numbers are extra. So, the finite state automaton whose states are been tracked by the stack gives us a method of examining the string phi beta and by looking at the a first case symbols of t and looking at the top of stack state, the L R parser will be able to determine, which production was used at this point, if it is a reduction, otherwise it will determine that it is a shift action.

(Refer Slide Time: 15:24)



LR Grammars (contd.)

- Example: The grammar, $\{S \rightarrow E,\ E \rightarrow E + E \mid E * E \mid id\}$, is not LR(2)
  - $S \Rightarrow^{1} \underline{E} \Rightarrow^{2} \underline{E + E} \Rightarrow^{3} E + \underline{E * E} \Rightarrow^{4} E + E * \underline{id} \Rightarrow^{5}$ $E + \underline{id} * id \Rightarrow^{6} \underline{id} + id * id$
  - $S \Rightarrow^{1'} \underline{E} \Rightarrow^{2'} \underline{E * E} \Rightarrow^{3'} E * \underline{id} \Rightarrow^{4'} \underline{E + E} * id \Rightarrow^{5'}$ $E + \underline{id} * id \Rightarrow^{6'} \underline{id} + id * id$
  - In the above two derivations, the handle at steps 6 & 6' and at steps 5 & 5', is $E \rightarrow id$, and the position is underlined (with the same lookahead of two symbols, $id+$ and $+id$)
  - However, the handles at step 4 and at step 4' are different ($E \rightarrow id$ and $E \rightarrow E + E$), even though the lookahead of 2 symbols is the same ($*id$), and the stack is also the same ($\phi = E + E$)
  - That means that the handle cannot be determined using the lookahead

Y.N. Srikant Parsing

So, here is a an example the grammar is ambiguous S to E, E to E plus E or E star E or i d, where we want to show that this is not L R 2 that is even with k look ahead of 2 we will not be able to determine the handle uniquely. If there are two derivations that I have shown here, S derives E, E derives E plus E, E plus E derives E plus E star E, then we derive E plus E star i d E plus i d star i d and i d plus i d star i d. So, this is a rightmost derivation, another rightmost derivation S to E E to E star E.

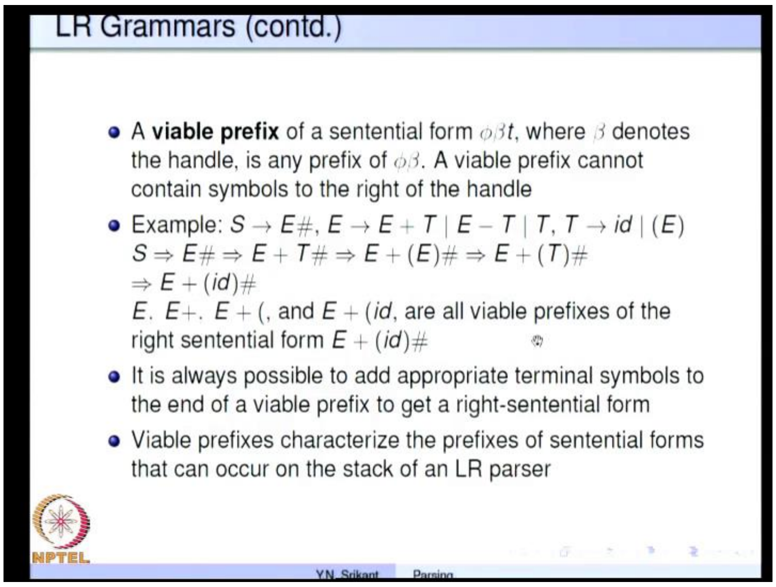
So, instead of E plus E we have E star E then E star i d, then this becomes E plus E star i d E plus i d star i d and i d plus i d star i d. So, the same string i d plus i d star i d has two right most derivations because, this is ambiguous, but the point we want to you know emphasis here is that we cannot determine the handle uniquely. So, consider this step 6 prime and a step 6, so the handle here is i d, the handle here is also i d, the same symbol first symbol and the production used of course, is E to i d.

So, we have E plus i d star i d as the sentential form in one step backward, here also i d E plus i d star i d is the sentential form when we traverse one step backward, the reduction from i d to E gives us this sentential form. The next step 5 prime and the corresponding step 5, again the handle is i d and the production, which has been used is again E to i d and in both cases we get the sentential form E plus E star i d E plus E star i d, which are identical.

Then, the difference surfaces for the step 4 and step 4 prime, the handle in this derivation is i d whereas, the handle in this derivation is E plus E. So, the look ahead in this case, you know the unexpanded input is star i d, here also the unexpanded input is star i d, so if we look at two symbols, we would be looking at star i d in both cases. So, that is the same look ahead string, but by looking at the string we are not able to uniquely determine whether the L R parser must take E plus E as the handle or E plus E as the handle or i d as the handle.

So, this is precisely the ambiguity and a grammar happens to be non L R 2 in this case, so the handle cannot be determined using the look ahead and of course, the derivation. So, because the stack content is identical, so you can see that E plus E is the stack in you know both cases, the unexpended input is star i d in both cases. So, by looking at the stack we are not able to say that the handle is E plus E here and i d here, so therefore, the grammar is not L R 2.

(Refer Slide Time: 19:13)



LR Grammars (contd.)

- A **viable prefix** of a sentential form $\phi\beta t$, where β denotes the handle, is any prefix of $\phi\beta$. A viable prefix cannot contain symbols to the right of the handle
- Example: $S \rightarrow E\#$, $E \rightarrow E + T \mid E - T \mid T$, $T \rightarrow id \mid (E)$
  $S \Rightarrow E\# \Rightarrow E + T\# \Rightarrow E + (E)\# \Rightarrow E + (T)\#$
  $\Rightarrow E + (id)\#$
  E, $E+$, $E + ($, and $E + (id$, are all viable prefixes of the right sentential form $E + (id)\#$
- It is always possible to add appropriate terminal symbols to the end of a viable prefix to get a right-sentential form
- Viable prefixes characterize the prefixes of sentential forms that can occur on the stack of an LR parser

Y.N. Srikant Parsing

Now, let us move on and understand how to build the automaton deterministic finite state automaton, which tracks the parser states and tells us when to shift and when to reduce, to do that we must go through some terminology. So, the first you know term that we need to define is a viable prefix of a sentential form, so a viable prefix of a sentential form phi beta T, where beta denotes the handle is any prefix of phi beta. So, in other words a viable prefix cannot contain symbols to the right of the handle.

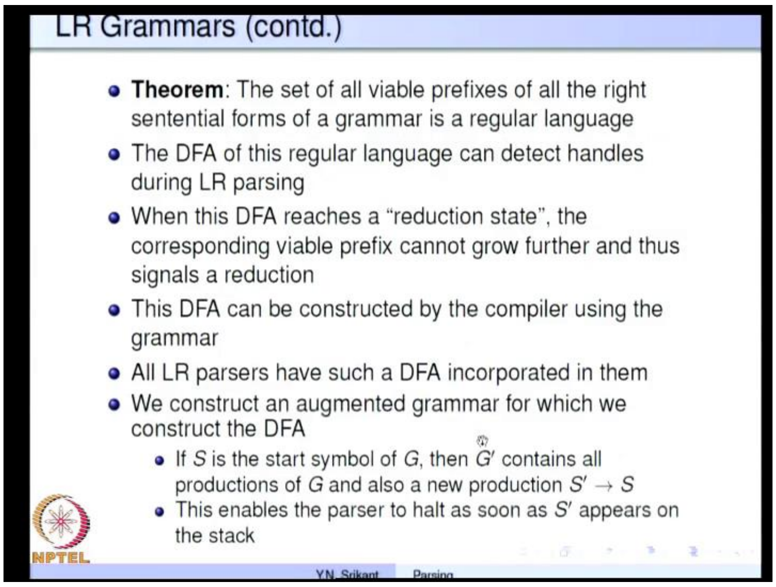
So, let us take an example, S the grammar is S to E hash E going to E plus T or E minus T or T, T going to i d or E parenthesis E parenthesis. So, let us look at a rightmost derivation S to E hash, now we apply the production E to E plus E, so E plus T, so we get E plus T hash, now we for T we apply T to parenthesis E parenthesis. So, we get E plus parenthesis E parenthesis hash now E to T is applied, so E plus T hash.

Now, T to i d is applied, so we get T E plus i d hash, in this sentential form the handle at this point is i d because, we applied the production T to i d to get this sentential form. So, the any prefix of E plus parenthesis i d is a viable prefix, so E E plus E plus parenthesis and E plus parenthesis i d are all viable prefixes of this particular sentential form. So, the property of a viable prefix is given a viable prefix, we should be able to add appropriate symbols to right end of the viable prefix to get a right sentential form.

So, for example, here you know we are able to if you take this viable prefix, then you know we can add a right parenthesis here and a hash here to get the right sentential form. So, in this case for example, you know if you look at this particular E plus parenthesis T parenthesis hash and so on. So, make sure once we derive all the terminal symbols from T we add whatever T derives and then we retain this E plus parenthesis, let us say T derives i d. So, if we add i d parenthesis and hash to E plus parenthesis we get a right sentential form.

So, similarly you can consider any one of these say E plus or something like that, so for E plus we add whatever T hash derives. So, that would include parenthesis i d parenthesis hash and we get a right sentential form, so this is the characteristic of a viable prefix, viable prefixes characterize the prefixes of sentential form that can occur on the stack of an L R parser. So, when we go from the terminal symbol terminal string to the start symbol, lot of reductions take place and during these reductions the stack contains parts of the sentential forms and the viable prefixes are exactly those you know parts which lie on the stack of an L R parser.

(Refer Slide Time: 23:25)



LR Grammars (contd.)

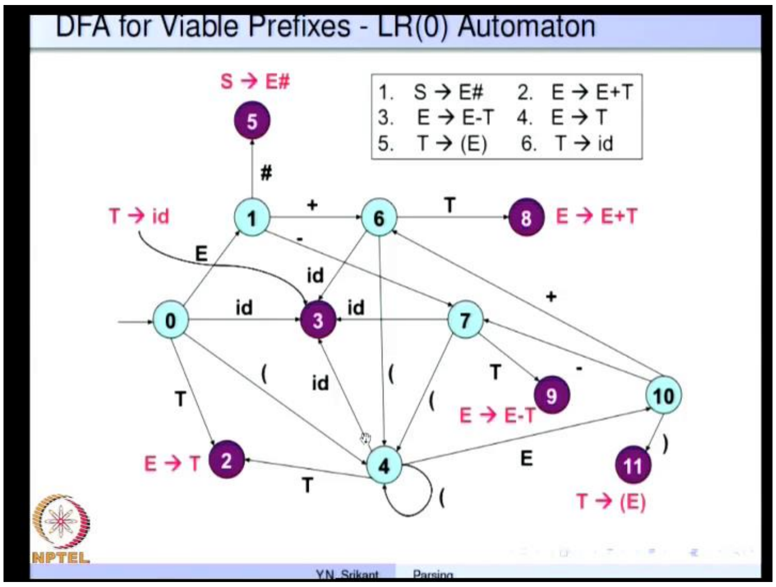
- **Theorem**: The set of all viable prefixes of all the right sentential forms of a grammar is a regular language
- The DFA of this regular language can detect handles during LR parsing
- When this DFA reaches a "reduction state", the corresponding viable prefix cannot grow further and thus signals a reduction
- This DFA can be constructed by the compiler using the grammar
- All LR parsers have such a DFA incorporated in them
- We construct an augmented grammar for which we construct the DFA
  - If S is the start symbol of G, then G' contains all productions of G and also a new production $S' \rightarrow S$
  - This enables the parser to halt as soon as S' appears on the stack

Y.N. Srikant Parsing

A major theorem in L R parsing theory is that, the set of viable prefixes of all the right sentential forms of a grammar, the set forms a regular language. So, the DFA of this language can detect handles during L R parsing, so we will be seeing that very soon, the point is the DFA reaches a, so called reduction state and signals that the viable prefix cannot grow further. So; that means, there is a reduction that is necessary at this point, I will show you what exactly is a reduction state after this slide.

This sort of a DFA can be constructed by the complier using the grammar and we are going to discuss that procedure a little later. All L R parsers have such a DFA incorporated in them, this is the heart of an L R parser really, so to do that we construct an augmented grammar and if S is the start symbol of G then G prime contains all the productions of G, along with a new production S prime going to S. The reason we do that is there could be productions from S with S on the right hand side as well. But, we want to make sure that the start symbol is unique and we want to halt the parser as soon as S prime appears on the stack, so to do that we add an extra start symbol and make it an augmented grammar.

(Refer Slide Time: 25:11)


DFA for Viable Prefixes - LR(0) Automaton

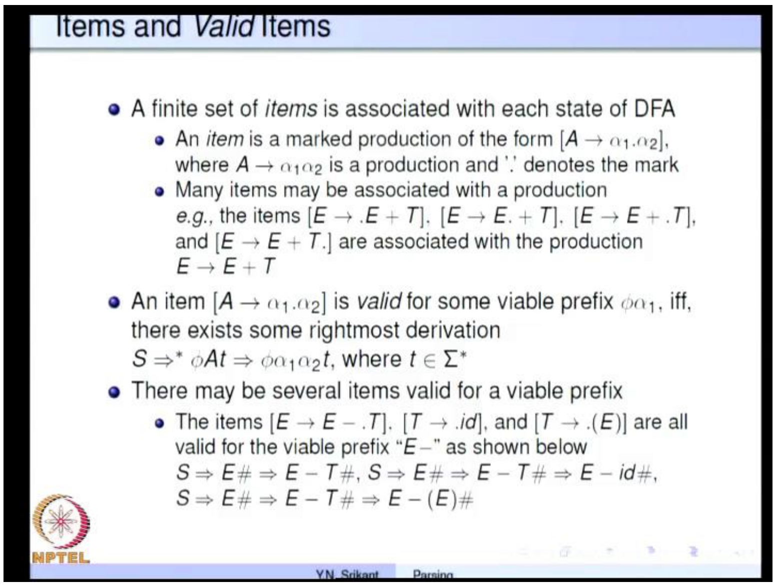
1. S → E#   2. E → E+T
3. E → E-T   4. E → T
5. T → (E)   6. T → id

So, here is an example of a DFA for this particular grammar, this is the L R 0 DFA for this particular grammar. So, let us understand what exactly this is there are some states, which are in a greenish blue and there are some other states which are in violet, the states which are marked in violet also have a production associated with them and these are the reduction states. So, 5, 8, 3, 2, 9 and 11 they are all reduction states and the other states 0, 1, 6, 7, 10 and 4, which are in greenish blue color are all shift states.

So, when the parser enters parser DFA enters one of these reduction states, then a reduction by this particular production is bound to happen. Whereas, if it is in any other state, then the upon an input signal symbol a shift would happen and it would go to an appropriate state. For example, from the state 6 on in the set of input i d it goes to state 3 whereas, in state 4 on the input parenthesis it remains in the same state 4 and it can possibly you know go to state 3 on i d, so this is the L R parser, this is the parser DFA here is a start state.

So, there is only one difference between a an actual DFA as we defined in lexical analysis and the DFA which I have written here. The difference is this particular DFA there is no explicitly defined final state, in fact, all the states are final states the reason being, in this DFA from the start state it does not matter which path you take that would form a viable prefix. So, for example, 0 to 1 form you know we have E, so E is the viable prefix 0, 1, 6, E plus is a viable prefix, E plus T is also a viable prefix and so on and so forth.

(Refer Slide Time: 27:45)



Items and *Valid* Items

- A finite set of *items* is associated with each state of DFA
  - An *item* is a marked production of the form $[A \rightarrow \alpha_1.\alpha_2]$, where $A \rightarrow \alpha_1\alpha_2$ is a production and '.' denotes the mark
  - Many items may be associated with a production *e.g.*, the items $[E \rightarrow .E + T]$, $[E \rightarrow E. + T]$, $[E \rightarrow E + .T]$, and $[E \rightarrow E + T.]$ are associated with the production $E \rightarrow E + T$
- An item $[A \rightarrow \alpha_1.\alpha_2]$ is *valid* for some viable prefix $\phi\alpha_1$, iff, there exists some rightmost derivation $S \Rightarrow^* \phi At \Rightarrow \phi\alpha_1\alpha_2 t$, where $t \in \Sigma^*$
- There may be several items valid for a viable prefix
  - The items $[E \rightarrow E - .T]$, $[T \rightarrow .id]$, and $[T \rightarrow .(E)]$ are all valid for the viable prefix "$E-$" as shown below $S \Rightarrow E\# \Rightarrow E - T\#$, $S \Rightarrow E\# \Rightarrow E - T\# \Rightarrow E - id\#$, $S \Rightarrow E\# \Rightarrow E - T\# \Rightarrow E - (E)\#$

Y.N. Srikant Parsing

So, then to construct this DFA there must be some algorithm written out and what we are going to do now is, we provide a method of constructing this DFA using what are known as items, the procedure is quite simple. And then state some results, which link this set you know the DFA constructed using items to the DFA which recognizes viable prefixes. So, let us define items a finite set of items is associated with each state of a DFA, remember we are now defining possibly some other DFA, we will provide an algorithm to construct this DFA using these items.

And then link this DFA and mention results that say this is a DFA which recognizes viable prefixes. So, what exactly is an item, an item is a very simple you know take a production put a dot anywhere on the right hand side of the production and you get a an item. So, for example, if you consider the production E to E plus T, then you can put a dot just before E plus T, just after the E, just after the plus and after the t, so you can actually form 4 items from this single production E to E plus T.

So, the general form of a production is A going to alpha 1 dot alpha 2 with either alpha 1 or alpha 2 or both being Epsilon. So, items are denoted by actually they are enclosed in 2 square brackets, now a little more terminology, an item A going to alpha 1 dot alpha 2 is said to be valid for some viable prefix phi alpha 1. So, now we are trying to link an item and viable prefix, so we have a viable prefix phi alpha 1, please see that the alpha 1 there and the alpha 1 in the production are same.

Secondly, observe that alpha 1 is the portion of the right hand side of the production just before the dot. So, it says A go into alpha 1 dot alpha 2 is valid for some viable prefix phi alpha 1, if and only if there exists some rightmost derivation, so what is the derivation S derives phi A t, and then you apply the production A go into alpha 1 alpha 2 at this point. So, you get phi alpha 1 alpha 2 t, so if this is the case, if we are able to get the production A go into alpha 1 apply alpha 2 applied.

and the viable prefix at this point is indeed phi alpha 1, then we say that the item A go into alpha 1 dot alpha 2 is valid for this particular viable prefix. You can also observe here, that the item A going to dot alpha 1 alpha 2 is actually valid for the viable prefix phi, why the same rightmost derivation can be used to show that, you know you have the sentential form phi alpha 1 alpha 2 t. So, alpha 1 alpha 2 is what you have here and the item would be A going to dot alpha 1 alpha 2 and the viable prefix is phi. So, A going to dot alpha 1 alpha 2 is valid for the viable prefix phi.

Similarly, if you consider the item A going to alpha 1 alpha 2 dot, you know then the viable prefix for which it is valid is phi alpha 1 alpha 2 that is very trivial from the same rightmost derivation again. So, you can consider this entire thing as the viable prefix and then after the dot there is nothing here, so just the T in the right sentential form. So, A going to alpha 1 alpha 2 dot will be valid for the viable prefix phi alpha 1 alpha 2.

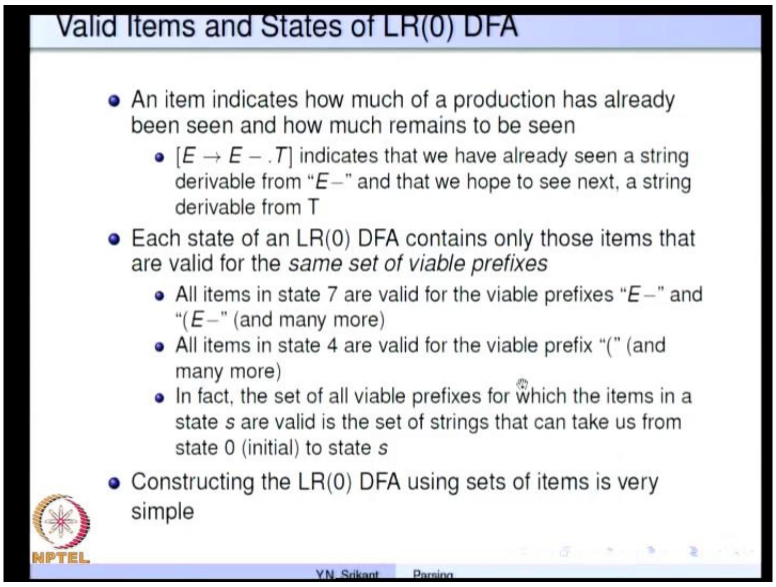
So, there may be several items valid for a single viable prefix let us see how, consider the derivations given below S derives E sharp and then E that derives E minus T sharp, then S derives E sharp which intern derives E minus T sharp and which again intern derives E minus i d sharp finally, we get S E sharp E minus T sharp and E minus parenthesis E parenthesis sharp, so 3 derivations.

Now, the viable prefix that we are considering is E minus, so in this sentential form again E minus is a viable prefix. In this sentential form also E minus is a viable prefix and in this sentential form also E minus is a viable prefix, the let us consider the 3 items the statement says all these 3 items are valid for the viable prefix E minus. So, let us take the first item E going to E minus dot t, so E minus is the alpha part, T is the alpha 1 part and T is the alpha 2 part.

So, we take the sequence A going to E hash going to E minus T hash, so E minus is our viable prefix that corresponds to our alpha 1 and then T is alpha 2. So, this item is valid for E minus T going to dot i d, so we consider this, so again E minus is our viable prefix. So, and alpha 1 is null here and i d is alpha 2, so E minus i d is here, so in this application, in this derivation we see the T minus is the viable prefix and therefore, T to i d is valid at this point i d is indeed the handle as well.

Third is T going to dot parenthesis E parenthesis, so we again take this derivation, so we have alpha 2 which is parenthesis E parenthesis E minus is our viable prefix here. So, this derivation shows that this is indeed valid for the viable prefix E minus, so there may be many items valid for a single viable prefix.

(Refer Slide Time: 35:19)



Valid Items and States of LR(0) DFA

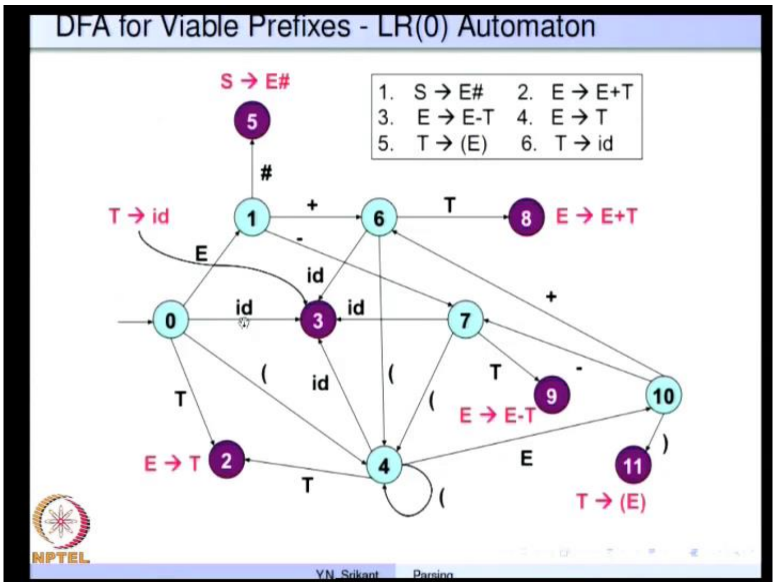
- An item indicates how much of a production has already been seen and how much remains to be seen
  - $[E \rightarrow E - .T]$ indicates that we have already seen a string derivable from "$E-$" and that we hope to see next, a string derivable from T
- Each state of an LR(0) DFA contains only those items that are valid for the *same set of viable prefixes*
  - All items in state 7 are valid for the viable prefixes "$E-$" and "($E-$" (and many more)
  - All items in state 4 are valid for the viable prefix "(" (and many more)
  - In fact, the set of all viable prefixes for which the items in a state *s* are valid is the set of strings that can take us from state 0 (initial) to state *s*
- Constructing the LR(0) DFA using sets of items is very simple

Y.N. Srikant Parsing

Then, what does an item indicate you know in a grammar and a derivation sequence, an item indicates how much of a production has already been seen and how much really remains to be seen. So, if you consider the production E going to E minus dot T, it says we have already seen the string of course, I am assuming that you know this item is valid for some viable prefix. So, in the derivation we would have already seen some part of the string input string derived from E minus and the input string derivable from T is yet to be seen, so this is the interpretation.

So, before the dot it indicates the past and after the dot it indicates the future and each state of the L R 0 DFA contains only those items that are valid for the same set of viable prefixes. So, you know I have still not told you how to construct the state of the DFA using these items, but a state will contain many of these items, the point is all the items here will be valid for the same set of viable prefixes. So, all the items in state 7 are valid for the viable prefixes E minus and parenthesis E minus.

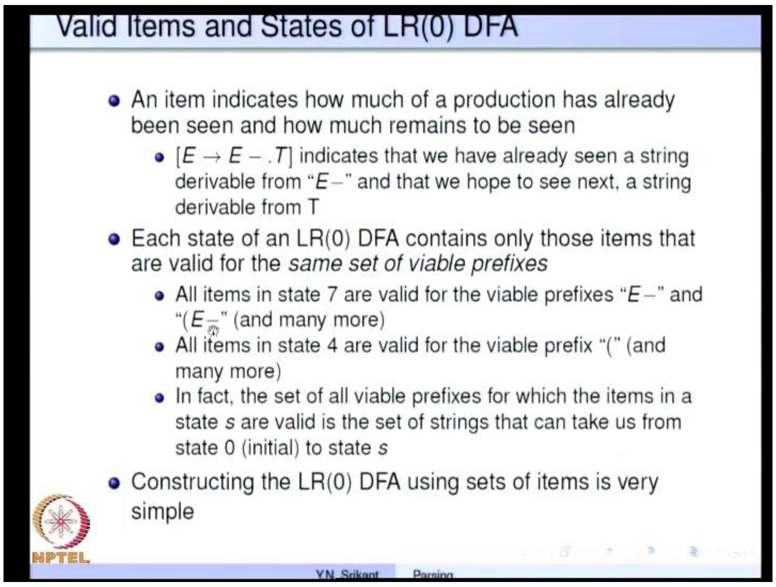
(Refer Slide Time: 37:12)



So, let us look at that, so E minus this is the state number 7, so you should observe that this the path we take from the initial state 0 from 0 we go to 1 and then 2 7 and labels we accumulate as E minus. So, similarly let us see how to go state 7 using some other path, so from 0 we can go to 1, then we go 6 and then we go state 10 and finally, we go to state 7. So, we have E plus you know, so E minus is what brings you here and we cannot go to from state 6, we cannot go to state 10 directly, we need to go state number 4 and then go to state number 10 the arc is in the reverse direction.

So, we get E plus and then parenthesis and then you know this side is again another E and finally, a minus. So, if you accumulate the labels on a path which takes you to any particular state, so that is very significant and that is what this is trying to say.

(Refer Slide Time: 38:32)



Valid Items and States of LR(0) DFA

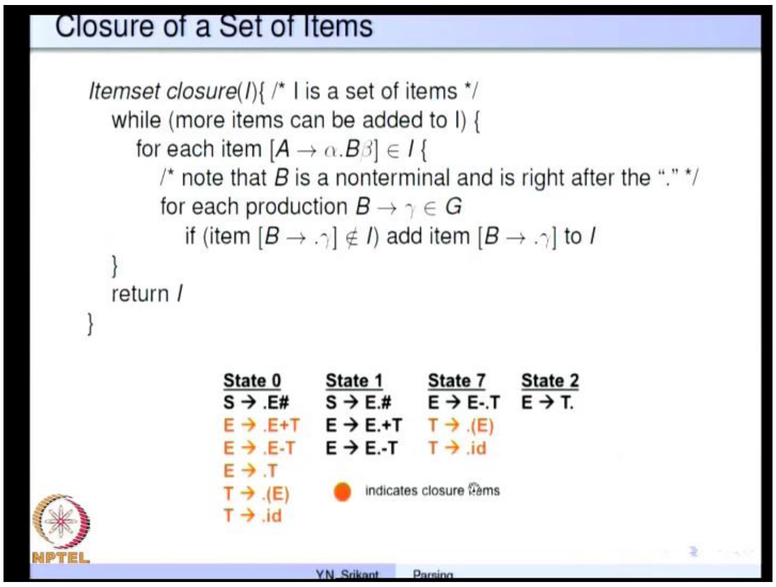
- An item indicates how much of a production has already been seen and how much remains to be seen
    - $[E \to E - .T]$ indicates that we have already seen a string derivable from "$E-$" and that we hope to see next, a string derivable from T
- Each state of an LR(0) DFA contains only those items that are valid for the *same set of viable prefixes*
    - All items in state 7 are valid for the viable prefixes "$E-$" and "$(E-$" (and many more)
    - All items in state 4 are valid for the viable prefix "(" (and many more)
    - In fact, the set of all viable prefixes for which the items in a state *s* are valid is the set of strings that can take us from state 0 (initial) to state *s*
- Constructing the LR(0) DFA using sets of items is very simple

Y.N. Srikant Parsing

So, all the items in state 7 are valid for the viable prefixes E minus and parenthesis E minus and many more of course. Similarly, all the states in items in state 4 are valid for the viable prefix parenthesis and many more of course, so the basic idea is what I was trying to describe just now. The set of all viable prefixes for which the items in a state S are valid is the set of strings that can take us from the state 0 to the state S, ((Refer Time: 39:10)) so here, so find out all the strings which can take you to state 7, those are the viable prefixes for the you know for which the items in state 7 are valid.

So, similarly if you take state 10 all the strings which can take you from 0 to 10 are the viable prefixes for which the items in state 10 are valid. So, that is about the validity of items, constructing L R 0 DFA using sets of items is very simple, so let us look at that procedure now and then look at the relevance of this two our problem.

(Refer Slide Time: 39:49)



Closure of a Set of Items

```
Itemset closure(I){ /* I is a set of items */
    while (more items can be added to I) {
        for each item [$A \rightarrow \alpha.B\beta$] $\in I$ {
            /* note that B is a nonterminal and is right after the "." */
            for each production $B \rightarrow \gamma \in G$
                if (item [$B \rightarrow .\gamma$] $\notin I$) add item [$B \rightarrow .\gamma$] to I
    }
    return I
}
```

State 0	State 1	State 7	State 2
S → .E#	S → E.#	E → E-.T	E → T.
E → .E+T	E → E.+T	T → .(E)	
E → .E-T	E → E.-T	T → .id	
E → .T			
T → .(E)			
T → .id			

● indicates closure items

Y.N. Srikant Parsing

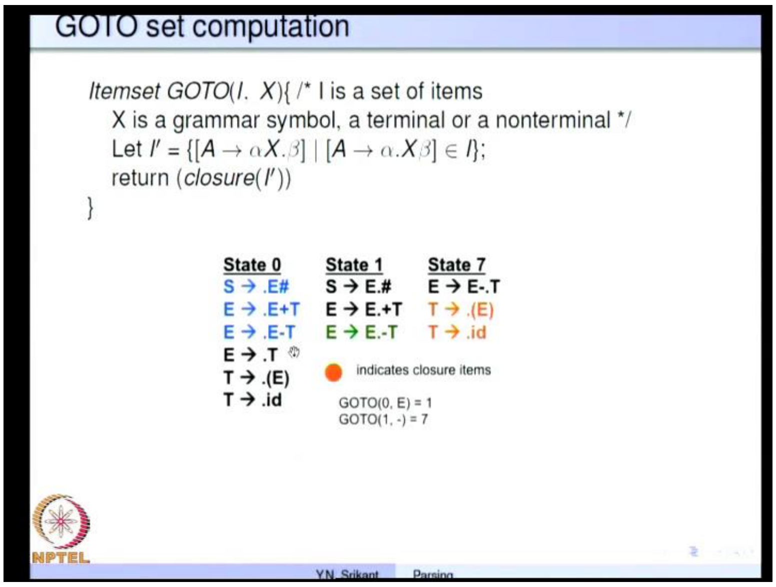
So, there is an operation called closure, so first let me explain the closure operation with respect to these examples and then look at the algorithm itself. So, let us say we are given an item S going to dot E hash, closure says look at the symbol after the dot if that say non terminal then add all the productions of that particular non terminal with a dot on the and put a dot in the left most position, for E there are 3 productions E to E plus T E to E minus T and E to T.

So, all these 3 items have been added with a dot in the left most position, now do this for the item that we just now added, again we have a symbol E here and there is nothing more to add for E. But, this gives us a new symbol T, so add all the productions and the item associated with it to the state, so T going to parenthesis E parenthesis and T going to i d with a dot in the left most position. So, these orange color once are items which we added because of the closure.

So, if here these are the item which are given to us, but it is, so happens that the symbol after the dot is a terminal symbol; obviously, there are no productions corresponding to terminal symbols and so we cannot expand this state further using closure. State number 7 we have E going to E minus dot T, T is a non terminal, so we can add two more items for this closure I know and E to T dot adds nothing. So, the closure process is very simple item set closure I, I is the set of items which are given to you, while more items cannot be added to I.

For each item A to alpha dot B to beta in I, so observe that we are looking at the non terminal B after the dot. And for each production B to gamma we add the item B going to dot gamma, if it is not already present in the item set to I. So, this is what we did here and when we considered these to is they give us the same items, so we did not add them a second time, so this is the closure operation.

(Refer Slide Time: 42:27)



GOTO set computation

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

State 0	State 1	State 7
$S \rightarrow .E\#$	$S \rightarrow E.\#$	$E \rightarrow E-.T$
$E \rightarrow .E+T$	$E \rightarrow E.+T$	$T \rightarrow .(E)$
$E \rightarrow .E-T$	$E \rightarrow E.-T$	$T \rightarrow .id$
$E \rightarrow .T$		
$T \rightarrow .(E)$		
$T \rightarrow .id$		

● indicates closure items

GOTO(0, E) = 1
GOTO(1, -) = 7

Y.N. Srikant Parsing

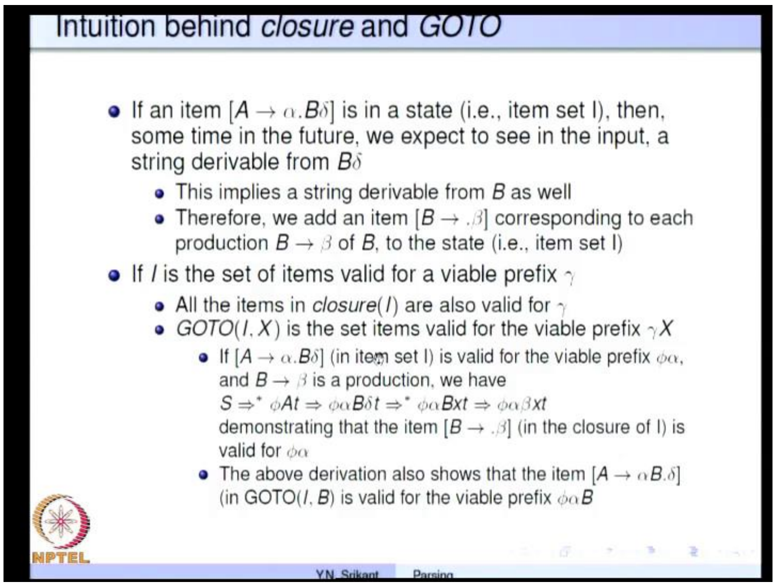
Then, there is a another operation called go to, so again let us consider you know the blue items which are in state 0. All the 3 blue items have the non terminal E just after the dot, the others have different symbols of course, the go to set computation tries to advance the dot by one position. So, whenever the symbol after the dot is exactly the same, we take all the items with that symbol on the after the dot in the same state.

Advance the dot by 1 position that gives us a couple of items for example, this gives us S going to E dot hash, this gives us S going to E dot plus T and this gives us E going E dot minus T. So, we really add them into another state, if the state if no other state has these items we create a new state and add this items to that particular state, so here again you know we check whether it is possible to do a closure operation. So, in this case all the symbols after the dot or terminal symbols, so there are no more items that we can add by a closure operation.

So, now for this again let us form the go to state go to set or go to state, so of just before the, you know minus there is a dot here. So, let us and this is only item which has a minus after the dot, so we consider only this particular item, advance the dot by 1 position, so we get the item E going to E minus dot T add it to a new state. Now, the symbol after the dot is a non terminal, so add the 2 items which can be derived by the closure operation to this state.

So, the go to set computation is very simple procedure go to of I comma X, I is a set of item, X is a grammar symbol either is a terminal or a non terminal. In this case it was a non terminal and in this case it is a terminal symbol, the new state or item set we get is I prime A to it contains all although the set or a item set contains A to alpha X dot beta. So, what we had, you know was A going to alpha dot X beta, so now, we advance the dot by one position, so alpha dot X beta was already in I, so we form the new item alpha X dot beta and put it into a new state I prime. If I prime was already there we do not form a new state, we just use the same you know it do not do anything more for that particular go to set. Now, form a closure of I prime and return it as the result, so this is what the go to set computation is...

(Refer Slide Time: 45:44)



Intuition behind *closure* and *GOTO*

- If an item $[A \rightarrow \alpha.B\delta]$ is in a state (i.e., item set I), then, some time in the future, we expect to see in the input, a string derivable from $B\delta$
  - This implies a string derivable from B as well
  - Therefore, we add an item $[B \rightarrow .\beta]$ corresponding to each production $B \rightarrow \beta$ of B, to the state (i.e., item set I)
- If I is the set of items valid for a viable prefix γ
  - All the items in *closure*(I) are also valid for γ
  - *GOTO*(I, X) is the set items valid for the viable prefix γX
    - If $[A \rightarrow \alpha.B\delta]$ (in item set I) is valid for the viable prefix $\phi\alpha$, and $B \rightarrow \beta$ is a production, we have $S \Rightarrow^* \phi At \Rightarrow \phi\alpha B\delta t \Rightarrow^* \phi\alpha Bxt \Rightarrow \phi\alpha\beta xt$ demonstrating that the item $[B \rightarrow .\beta]$ (in the closure of I) is valid for $\phi\alpha$
    - The above derivation also shows that the item $[A \rightarrow \alpha B.\delta]$ (in GOTO(I, B) is valid for the viable prefix $\phi\alpha B$

Y.N. Srikant Parsing

So, now what is the intuition behind closure and go to why should we do all these, if an item A going to alpha dot B beta is in a stay in a particular state or items set. Then some time in the future we expect to see in the input a string derivable by from B delta that we already know. So, the implication is if these string is derivable from B delta, there should be a small part of that big string, which is derivable by a from the non terminal B as well.

So, this implies a string derivable from B as well, this is the reason for adding the item B going to dot beta corresponding to the production B to beta of B to the state that we already have. So, if the state contained A going to alpha dot B beta and we expect that we see a string derivable from B delta, we must correctly add the items B going to dot beta as well to announce that a small part of this string is derivable from B, which is nothing, but the string derivable from beta.

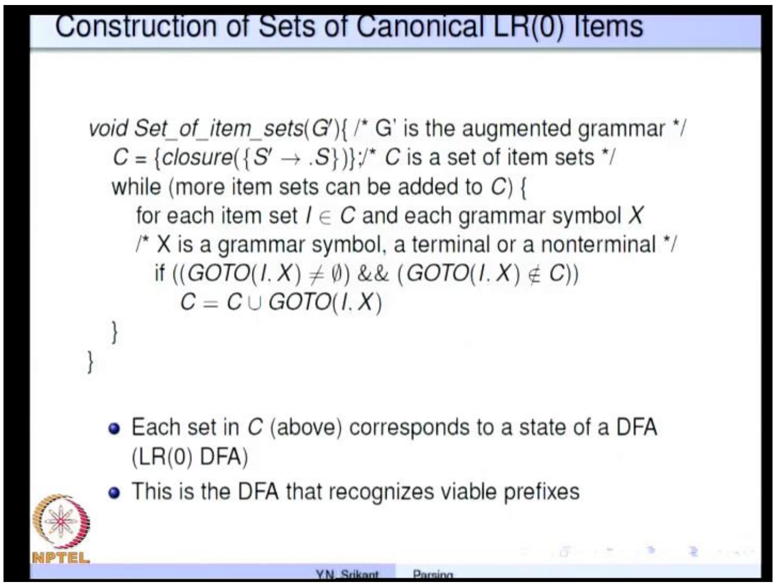
Now, if this is about the closure, so in summary when we add something because of the closure operation, we are only announcing that parts of this big, you know sentential form are derivable form the non terminals that are present in the sentential form. If I is the set of items valid for a viable prefix gamma, then it is important to know that all the items in the closure I are also valid for gamma.

So, I already kind of showed you this before, but let me show it to you again, if A to alpha dot B beta is valid for the viable prefix phi alpha 1, then B beta b is a production, we consider the derivation S going to phi A t phi alpha B delta t and then phi alpha B x t and that becomes phi alpha beta x t. So, we are applying the production B to beta here and we are applying the production A going to alpha B delta here. So, this particular derivation shows that not only is the item A going to alpha dot B delta is valid for the prefix phi alpha, you know B to dot beta is also a valid for the prefix by alpha.

So, see this here this is our viable prefix and this beta is the handle and since dot is at the beginning of the item B going to dot beta, the dot is right here, so it is valid for this particular viable prefix phi alpha. So, phi alpha is here phi alpha is here, so both the items are valid, now what about the GOTO, GOTO of phi X is the set of items valid for the viable prefix gamma X. So, here the above derivation that is this derivation, also shows that the item A going to alpha B dot delta is valid for the viable prefix phi alpha b I already explain this before.

So, in the same production phi alpha B can be our viable prefix, so in that case the item would be A going to alpha B dot delta. So, that would be valid for the viable prefix phi alpha B, so this the intuition behind the construction of the automaton, so let us look at the entire algorithm.

(Refer Slide Time: 50:07)



Construction of Sets of Canonical LR(0) Items

```
void Set_of_item_sets(G'){ /* G' is the augmented grammar */
  $C = \{closure(\{S' \rightarrow .S\})\}$;/* C is a set of item sets */
  while (more item sets can be added to C) {
    for each item set $I \in C$ and each grammar symbol X
    /* X is a grammar symbol, a terminal or a nonterminal */
      if (($GOTO(I, X) \neq \emptyset$) && ($GOTO(I, X) \notin C$))
        $C = C \cup GOTO(I, X)$
  }
}
```

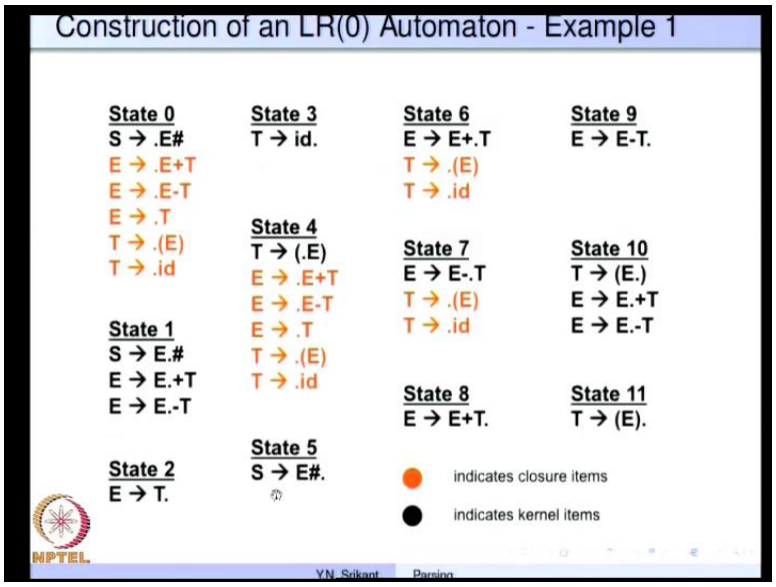
- Each set in C (above) corresponds to a state of a DFA (LR(0) DFA)
- This is the DFA that recognizes viable prefixes

Y.N. Srikant Parsing

So, how to construct set of items for the grammars G prime which is the augmented grammar, to begin with form just one item S prime going to dot S and take it closure. So, you get a set of items now, so now, you know until more sets can be added to that this thing, the item set you know rather set of item sets C, we form a GOTO and then you know if go to of I comma X not equal to phi and go to of I comma X is not in c, at that to the collection, C is the collection of set of items, now C union go to of i X.

Now, go back you have one more state are the item set in the collection, so we keep on you know applying the GOTO, GOTO in turn applies closure as well and this is how we keep doing it until we cannot get anymore new states. So, each set in C corresponds to a state of the L R 0 DFA and this is the DFA that recognizes viable prefixes.

(Refer Slide Time: 51:31)



Construction of an LR(0) Automaton - Example 1

State 0
S → .E#
E → .E+T
E → .E-T
E → .T
T → .(E)
T → .id

State 1
S → E.#
E → E.+T
E → E.-T

State 2
E → T.

State 3
T → id.

State 4
T → (.E)
E → .E+T
E → .E-T
E → .T
T → .(E)
T → .id

State 5
S → E#.

State 6
E → E+.T
T → .(E)
T → .id

State 7
E → E-.T
T → .(E)
T → .id

State 8
E → E+T.

State 9
E → E-T.

State 10
T → (E.)
E → E.+T
E → E.-T

State 11
T → (E).

● indicates closure items

● indicates kernel items

Y.N. Srikant Parsing

Let me explain the operation of set of item construction of the set of items, so to begin with this is our start state S going to dot E hash. So, the old start symbol was E, the new start symbol is S, so this is the augmented grammar, instead of S prime I have just used S here. So, S going to dot E hash is the first item, from which we are all these items because of the closure operation, so observe the E after the dot, so these two items get in and then the these items also gets in, so these are the productions for the E.

And because of the T these 2 items get in, so we have one state now, so let us advance the dot systematically for each item. So, now, the dot goes to the second position, so it becomes E dot hash, so this is the GOTO state you know go to of state 0 on E will be state 1. So, we have E dot hash and then E is also the symbol after the dot for these two items, so these two also go into the same state, none of the symbols after the dot or non terminal, so the state cannot grow further, because of the closure operation.

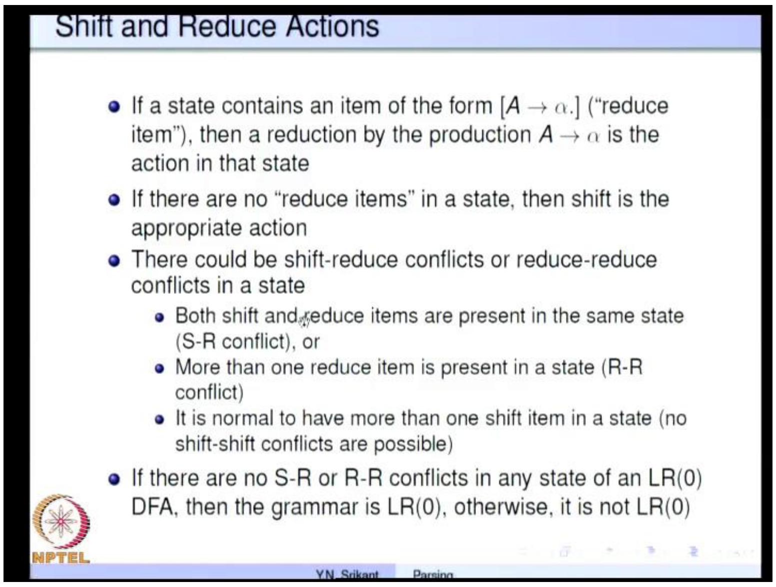
Consider this item, it gives you E to T dot and that is only item in the state 2, then we get T going to i d dot that is state 3. And we get state 4 from advancing the dot in this item, parenthesis dot E parenthesis, so E is the non terminal, so we add these three items and then because of this non terminal we add these two items. So, we have exhausted all the items in state 0 now, but we have added you know 4 new states state 1, state 2, state 3 and state 4.

And we need to apply the go to operation on the states as well, this gives us E hash dot that is state 5 and it cannot grow further. This gives us E plus dot T and the closure operation adds these 2, because of the T after the dot, E going to E minus dot T the dot advance you know the state go to of this would be E minus dot T, so that is the state and two more items are added, because of the closure operation the T after the dot. So, that exhaust this state, this cannot give us anymore, this also cannot give us anymore, this cannot give us anymore, but this can give us many more really.

So, the dot is advance the after the E, so parenthesis E dot parenthesis, so that gives us parenthesis E dot parenthesis. And these two are also added to the same state because, the non terminal E exist immediately after the dot, but we cannot grow the state further. Then E to T dot is already state available, this T to parenthesis dot E dot E parenthesis is the self state here and T do i d dot is a already state present. So, this is exhausted this does not give us any extra, but this can.

So, this gives us E plus T dot and that is state 9, these two do not generate any more new states, they generate just this state and this state respectively. This gives us a new you know this cannot this gives us a new state E minus T dot, these two cannot, this is already a state, which cannot grow further. And this particular state does not gives us only one extra state that is parenthesis E parenthesis dot and these two do not generate any extra state. They actually generate 6 and 7 respectively, so these are the entire sets of items that get generated, because of the you know closure and GOTO operations.

(Refer Slide Time: 56:03)



Shift and Reduce Actions

- If a state contains an item of the form $[A \rightarrow \alpha.]$ ("reduce item"), then a reduction by the production $A \rightarrow \alpha$ is the action in that state
- If there are no "reduce items" in a state, then shift is the appropriate action
- There could be shift-reduce conflicts or reduce-reduce conflicts in a state
  - Both shift and reduce items are present in the same state (S-R conflict), or
  - More than one reduce item is present in a state (R-R conflict)
  - It is normal to have more than one shift item in a state (no shift-shift conflicts are possible)
- If there are no S-R or R-R conflicts in any state of an LR(0) DFA, then the grammar is LR(0), otherwise, it is not LR(0)

Y.N. Srikant Parsing

And the shift and reduce actions are actually derived using this particular set of items, so we will look at this particular, you know method of filling the parser table using the sets of items in the next lecture.

Thank you.