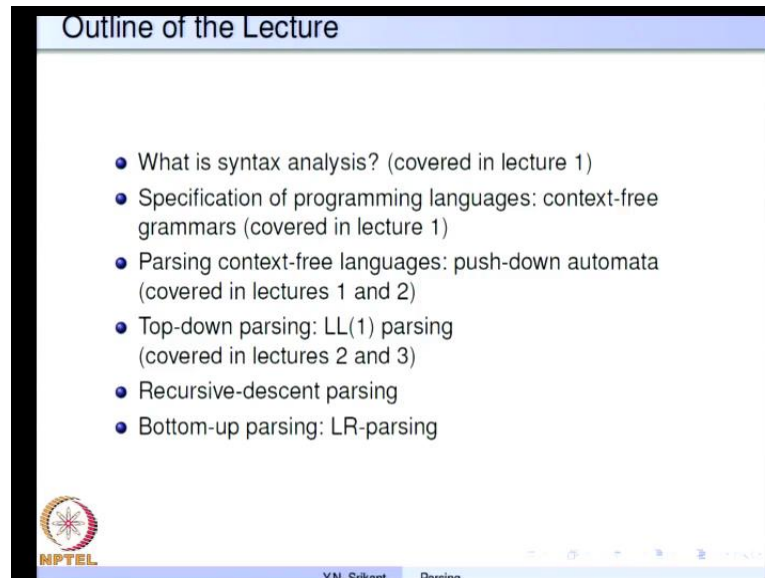


Principle of Compiler Design
Prof. Y. N. Srikant
Department of Computer Science and Automation
Indian Institute of Science, Bangalore

Lecture - 8

Syntax Analysis: Context-free Grammars, Pushdown Automata and Parsing Part-4

(Refer Slide Time: 00:20)

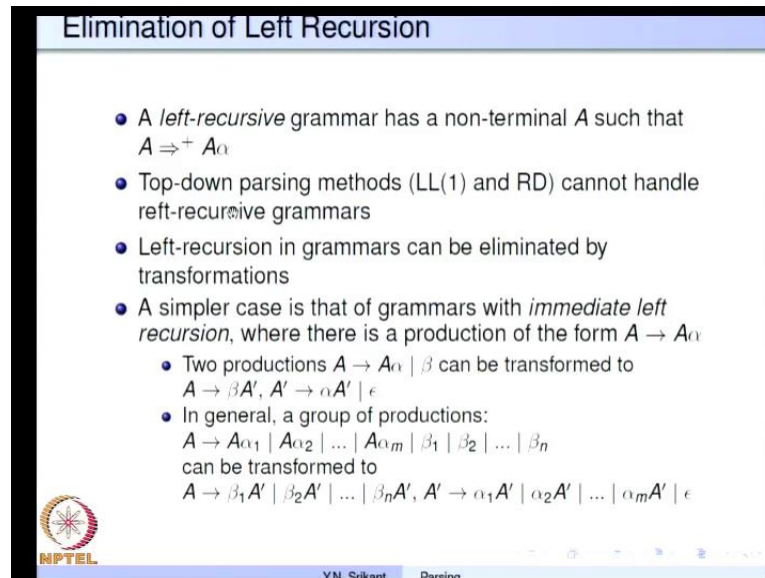


The slide is titled "Outline of the Lecture" and contains a bulleted list of topics. The list includes: "What is syntax analysis? (covered in lecture 1)", "Specification of programming languages: context-free grammars (covered in lecture 1)", "Parsing context-free languages: push-down automata (covered in lectures 1 and 2)", "Top-down parsing: LL(1) parsing (covered in lectures 2 and 3)", "Recursive-descent parsing", and "Bottom-up parsing: LR-parsing". The slide also features the NPTEL logo in the bottom left corner and a footer with the text "Y.N. Srikant Parsing".

- 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
- Bottom-up parsing: LR-parsing

Welcome to the lecture on syntax analysis part four. So, in the previous lectures we covered basics of syntax analysis context, free grammars, context free language and top down parsing. Today we will continue our discussion on parsing with recursive descent parsing and then move on to bottom up parsing.

(Refer Slide Time: 00:39)



Elimination of Left Recursion

- A *left-recursive* grammar has a non-terminal A such that $A \Rightarrow^+ A\alpha$
- Top-down parsing methods (LL(1) and RD) cannot handle left-recursive grammars
- Left-recursion in grammars can be eliminated by transformations
- A simpler case is that of grammars with *immediate left recursion*, where there is a production of the form $A \rightarrow A\alpha$
 - Two productions $A \rightarrow A\alpha \mid \beta$ can be transformed to $A \rightarrow \beta A', A' \rightarrow \alpha A' \mid \epsilon$
 - In general, a group of productions:
 $A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \dots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$
can be transformed to
 $A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \dots \mid \beta_n A', A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \dots \mid \alpha_m A' \mid \epsilon$

MPTEL
Y.N. Srikant Parsing

So, we looked a transformation known as useless symbol elimination and now we also look at two more transformations. One on left recursion elimination and another one known as left factoring, the left recursive grammars post problems to L L 1 parsers a generation. The problem is that left recursive grammars do not satisfy the L L 1 condition.

So, let us understand what exactly left recursive grammars are. So, if a grammar has a non terminal A such that A in more than one application of the productions of A produces $A\alpha$. So, you can see that the first symbol of this sentential form is also A so that means, we can produce as many, you know we can go and applying the productions of A as many times as we want and this is known as left recursion. So, our L L 1 parsing method and also the one we are going to see the recursive descent parsing method cannot handle left recursive grammars. So, left recursion grammars can be of course, eliminated, in other words we can convert left recursion to right recursion and similarly right recursion can be converted to left recursion.

So, first of all, if you look at what is known as immediate left recursion then we need to handle productions of the form A going to $A\alpha$, remember left recursion implies A derives $A\alpha$ in one or more applications of productions of A . Whereas, here the production itself has non terminal A on the left most in the left most position. So, if there is a production of the form A going to $A\alpha$ then the grammar is set to have

immediate left recursion. So, let us see what we can do with it. So, assume that there are two productions $A \rightarrow A\alpha$ and $A \rightarrow \beta$, the assumption is $A \rightarrow \beta$ will not have A in the left most position.

So, if that is so, then we can transform these two productions into these three productions. First of all $A \rightarrow \beta$ A' and then $A' \rightarrow \alpha A'$ or $A' \rightarrow \epsilon$, A' is a new non terminal, which did not exist before. So, if you look at this, you know this application produces exactly $\beta A'$ and nothing else, but then A' can be you know it has right recursion. So, we can use it as many times as we want. So, $A' \rightarrow \alpha A'$ can be used many times to produce many instances of A . So, if you look at this production we produce as many instances of α as we wanted and then A was replaced with β . So, here also A' produces as many instances of α as we want. And finally, A' goes to ϵ there by producing nothing.


But we have already produced a β to begin with. So, we have produced β followed by many α s which is exactly the same as what A produced. So, this is immediate left recursion elimination this grammar, you know does not have left recursion. So, if we have a group of productions say $A \rightarrow A\alpha_1$, $A \rightarrow A\alpha_2$, etcetera, $A \rightarrow A\alpha_m$. And then the other alternates without left recursion would be β_1 , β_2 , etcetera, β_n . So, if this is the case then we can transform these productions to in a very similar way to this $A \rightarrow \beta_1 A'$, $\beta_2 A'$, etcetera, $\beta_n A'$. And then the new non terminal A' will have productions $\alpha_1 A'$, $\alpha_2 A'$, etcetera, $\alpha_m A'$ and an extra ϵ as well. So, this is just a generalization of this transformation and is show to you know remove left recursion, immediate left recursion.

(Refer Slide Time: 05:27)

Left Recursion Elimination - An Example

$$A \rightarrow A\alpha \mid \beta \Rightarrow A \rightarrow \beta A', A' \rightarrow \alpha A' \mid \epsilon$$

- The following grammar for regular expressions is ambiguous:
 $E \rightarrow E + E \mid E E \mid E^* \mid (E) \mid a \mid b$
- Equivalent left-recursive but unambiguous grammar is:
 $E \rightarrow E + T \mid T, T \rightarrow T F \mid F, F \rightarrow F^* \mid P, P \rightarrow (E) \mid a \mid b$
- Equivalent non-left-recursive grammar is:
 $E \rightarrow T E', E' \rightarrow + T E' \mid \epsilon, T \rightarrow F T', T' \rightarrow F T' \mid \epsilon,$
 $F \rightarrow P F', F' \rightarrow * F' \mid \epsilon, P \rightarrow (E) \mid a \mid b$

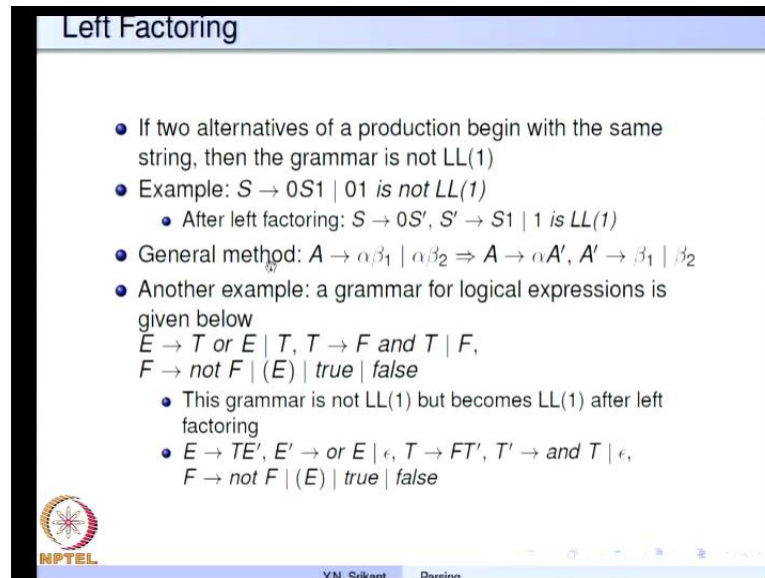


YN, Srikant Parsing

Let us take an example. So, the rule is given here, A going to $A\alpha$ or β will be transform to A going to $\beta A'$, A' going to $\alpha A'$ or ϵ . So, this grammar has left recursion because, E goes to $E + E$ or $E E$ or E^* or parenthesis E parenthesis or a or b . So, in one two and three these three productions we have E as a left most non terminal and therefore, there is left recursion. So, if we eliminate the left recursion, but remove the ambiguity as well this is ambiguous. So, sorry, if we want to eliminate the left recursion.


First of all, let us try to write a left recursion, but unambiguous grammar because ambiguous grammars do not you know help us in parsing. So, this is the unambiguous version of that grammar E plus T or T , E goes to $E + T$ or T , T goes to $T F$ or F , F goes to F^* or P , P goes to parenthesis E parenthesis or a or b . So, this is the, you know generates the same language as the previous one, but is unambiguous, but this is still left recursive. So, if we remove it you know the left recursion then we get E going to $T E'$ prime exactly like this, you know β a prime right. So, E going to T is the other productions. So, we have E going to $T E'$ prime E' prime going to plus $T E'$ prime. So, this is the part plus T and then or epsilon, T going to $F T'$ prime, T' prime going to $F T'$ prime or epsilon, F going to $P F'$ prime, F' prime going to star F' prime or epsilon and P remains as it is. Of course, we could try eliminating left recursion from this grammar as well and see what happens. So, that I leave it you as a home work.

(Refer Slide Time: 07:38)



Left Factoring

- If two alternatives of a production begin with the same string, then the grammar is not LL(1)
- Example: $S \rightarrow 0S1 \mid 01$ is not LL(1)
 - After left factoring: $S \rightarrow 0S'$, $S' \rightarrow S1 \mid 1$ is LL(1)
- General method: $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \Rightarrow A \rightarrow \alpha A'$, $A' \rightarrow \beta_1 \mid \beta_2$
- Another example: a grammar for logical expressions is given below
 $E \rightarrow T \text{ or } E \mid T$, $T \rightarrow F \text{ and } T \mid F$,
 $F \rightarrow \text{not } F \mid (E) \mid \text{true} \mid \text{false}$
 - This grammar is not LL(1) but becomes LL(1) after left factoring
 - $E \rightarrow TE'$, $E' \rightarrow \text{or } E \mid \epsilon$, $T \rightarrow FT'$, $T' \rightarrow \text{and } T \mid \epsilon$,
 $F \rightarrow \text{not } F \mid (E) \mid \text{true} \mid \text{false}$

 Y.N. Srikant Parsing

So, that was about left recursion elimination. As I said a left factoring is the other transformation that we want to study. If there are two alternatives beginning with the same string then the grammar is not LL(1), example is here. So, S going to $0S1$ or 01 so, this is definitely not a LL(1) grammar. And the idea intuition behind left factoring is remove the common portion of the alternatives. And then have a separate non terminal for the rest of it. So, for example, here zero is the common portion.

So, that is factored out and for the rest of the portion, we have a new non terminal S' . So, S' goes to $S1$, which is the rest of it here and or 1 , which is the rest of it in the other alternate. This grammar is of course, is definitely LL(1). So, this transformation you know using this transformation is possible some times to change a grammar which is not LL(1) to LL(1) form of course, left recursion was another as well. General method would be, if there are two alternatives for a production A going to $\alpha\beta_1$ or $\alpha\beta_2$ of course, there could be many alternatives with the same prefix as well. So, the method remains similar. So, we remove α . So, A becomes $\alpha A'$ and new non terminal A' goes to β_1 or β_2 .

So, here is another example for logical expressions. So, E going to T or E or T , T going to F and T or F and F goes to $\text{not } F$ or (E) or true or false . So, this grammar of course, is definitely not LL(1), but because it starts with the same non terminal T for both the options, here also it starts with the same non terminal F for both

options. Once we do left factoring we get E going to T prime, E prime going to or E or epsilon, T going to F T prime, T prime going to and T or epsilon, F going to not F or parenthesis E parenthesis true or false. So, this definitely is L L 1. So, it is not as if these are hypothetical examples, these you know, this expression grammar of course, occurs in the programming language the, you know in the grammar for programming language c or similar languages.

(Refer Slide Time: 10:39)

Grammar Transformations may not help!

Original Grammar

$$S' \rightarrow SS$$

$$S \rightarrow \text{if id } S \mid \text{if id } S \text{ else } S \mid a$$

LL(1) Parsing Table for modified grammar

	if	else	a	\$
S'	$S' \rightarrow SS$			
S	$S \rightarrow \text{if id } S \mid S1$		$S \rightarrow a$	
$S1$		$S1 \rightarrow \epsilon$ $S1 \rightarrow \text{else } S$		$S1 \rightarrow \epsilon$

Left-Factored Grammar

$$S' \rightarrow SS$$

$$S \rightarrow \text{if id } S S1 \mid a$$

$$S1 \rightarrow \epsilon \mid \text{else } S$$

tokens: if, id, else, a

dirsyb(SS) = {if, a}; dirsyb(a) = {a}

dirsyb(if id S S1) = {if}

dirsyb(else S) = {else}


dirsyb(ϵ) = { ϵ , S}

Grammar is not LL(1)

$dirsyb(\text{if id } S S1) \cap dirsyb(a) = \emptyset$

$dirsyb(\epsilon) \cap dirsyb(\text{else } S) \neq \emptyset$

Choose $S1 \rightarrow \text{else } S$ instead of $S1 \rightarrow \epsilon$ on lookahead *else*.
This resolves the conflict. Associates *else* with the innermost *if*



Y.N. Srikant Parsing

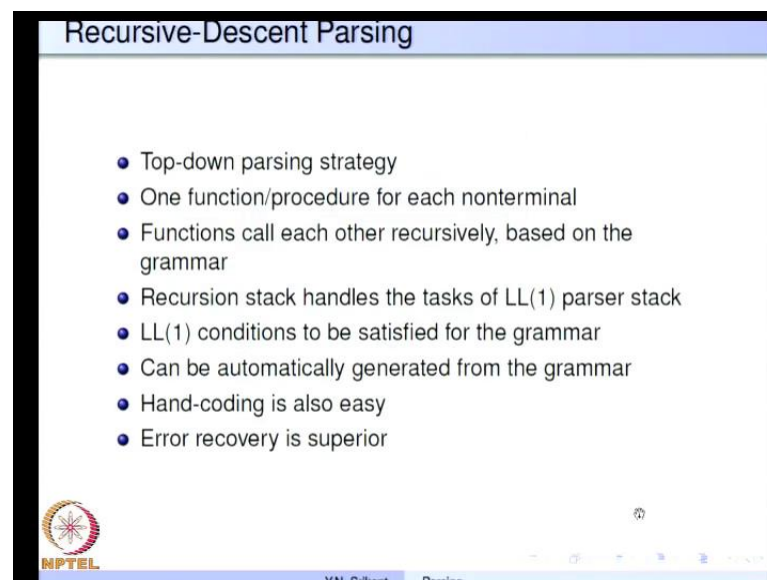
So now, an example to show that grammar transformation may not help every time. Here is our, you know controversial if then else grammar, we know that this is ambiguous. So, even if we do and of course, it has the same left side as well if i d S and if i d S because, it is ambiguous, even if we do left factoring. Let us say S prime going to S dollar, S going to if i d S S 1 or a, S 1 going to epsilon or else s. So, this is a factored grammar. So, there is no common part between the left side and at the two right hand sides of S, but once we compute the directions symbol sets and try to fill up the table.

We have seen this example before so, I will not go into too many details again, for the symbol S 1 and else we get two productions. So, the grammar does not happen to be L L 1. So, left factoring has not really helped in this case because, the underline grammar was ambiguous. Suppose you know, practically if the grammar produces, you know such a table and we have a problem with it. In such a case the programmer can be given a choice to choose the correct entry and eliminate the other entries correct in the sense

intuit inventory. So, in this case, if we choose $S \rightarrow \epsilon$ instead of $S \rightarrow \text{else } S$ on the look at else. This resolves the conflict because we have eliminated one of the productions here. And this associates else with the innermost if then which is actually the correct choice.

So, the matched statement grammar that I gave you couple of lectures ago, does this as well. So, the choice of eliminating $S \rightarrow \epsilon$ and retraining $S \rightarrow \text{else } S$ fills this table correctly makes this operatable as an LL(1) parser, but it will associate the else with the innermost if.

(Refer Slide Time: 13:11)



The slide is titled "Recursive-Descent Parsing" and contains a list of seven bullet points. At the bottom left, there is an NPTEL logo. At the bottom center, the text "Y.N. Srikant" and "Parsing" is visible. At the bottom right, there are navigation icons for a presentation slide.

- Top-down parsing strategy
- One function/procedure for each nonterminal
- Functions call each other recursively, based on the grammar
- Recursion stack handles the tasks of LL(1) parser stack
- LL(1) conditions to be satisfied for the grammar
- Can be automatically generated from the grammar
- Hand-coding is also easy
- Error recovery is superior

So, let us move on to recursive descent parsing. Recursive descent parsing is a top down parsing strategy and basically instead of a table driven, you know table driven parser as in the case of LL(1) parser. Here this is a, you know inlined program as such. So, we are going to have one function or procedure for each non terminal.

So, these are hard coded programs rather than table driven parsers. The function call each other recursively that is why this is called a recursive descent parser based on the grammar. I will show you an example as well. The recursion stack handles the tasks of the LL(1) parser stack. So, there is no other stack necessary like in the case of LL(1) parser the programming language stack which handles recursion will also take care of the job of the LL(1) parser stack. The exactly the same LL(1) conditions need to be satisfied for the grammar here as well for the grammar to be eligible for a recursive descent

parsing. Just like the LL(1) parser, we can generate recursive descent parsers also automatically from the grammar. Hand coding is also very easy even, if you do not generate them automatically. The advantage of recursive descent parsers is that error recovery in such grammars such parsers is superior to the though LL(1) parsers.

(Refer Slide Time: 14:51)

An Example

```
Grammar: S' → S$. S → aAS | c, A → ba | SB. B → bA | S

/* function for nonterminal S' */
void main() { /* S' --> S$ */
    fs(); if (token == eof) accept();
        else error();
    }
/* function for nonterminal S */
void fs() { /* S --> aAS | c */
    switch token {
        case a : get_token(); fA(); fs();
                break;
        case c : get_token(); break;
        others : error();
    }
}
```

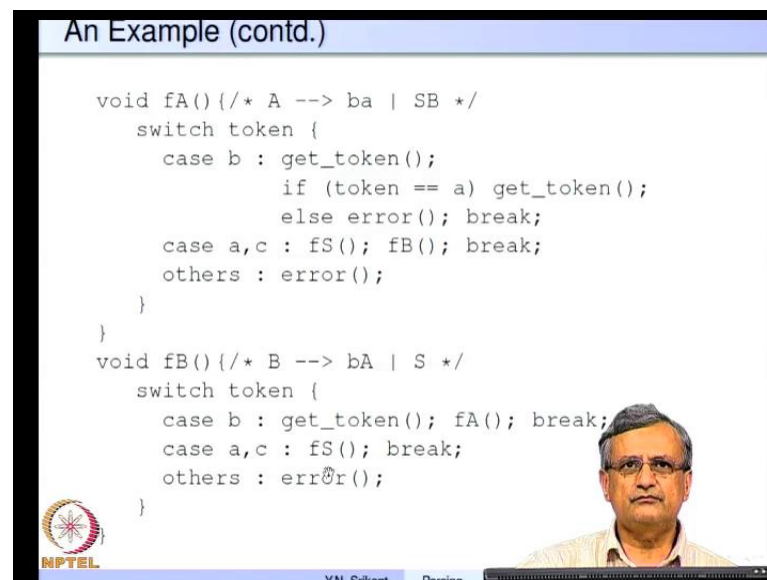
So, let us take an example and then move on to automatic generation of recursive descent parsers, here is a very simple grammar $S' \rightarrow S\$, S \rightarrow aAS \mid c, A \rightarrow ba \mid SB, B \rightarrow bA \mid S$. As I said there is one function for each non terminal. So, this is the function for non terminal S' then we have a function for non terminal S and then we are going to have a, you know function for non terminal a and another one for non terminal b as well. So, let us look at the function for non terminal S' . So, this is the main program because, it corresponds to the start symbol of the grammar. So, on the right hand side we have the non terminal S followed by the end of file symbol dollar. So, they flow within the program in the within the parser would be call the function fS corresponding to the non terminal S .

And once it completes that the control comes here. So, the next token is checked whether it is E or F not, if it E or F parsing is over so, you accept otherwise you show an error message. So, this is the for the non terminal S' which is very intuitive the whenever there is a non terminal you call the function for the non terminal and whenever there is a terminal check whether the next token actually corresponds to the terminal present in the

production. So, let us take the function for the non terminal S. So, all the alternatives are combined in the function. So, we really have many, you know cases of the switch statement one corresponding to each of the, I know alternatives of the production. The first alternative is a A S and it begins with little a, the second alternative is little c.

So, we have on the token a switch statement to check whether it is little a or little c and the grammar as been assumed to be L L 1. So, these two alternatives will never begin with the same symbol. So, little a assures that the production to be used for expansion in L L 1 parsing is S going to a A S. And so, we are going expand using the production S going to a A S here as well. So, the token a is matched already because, we came to case a, we get the next token then here is a function of a, here is a non terminal A. So, we call the function f A. The next one is a non terminal S so, we call the function f S then that case is completed. So, we have a break. So, that we get out of the case switch statement for the case c, we already, you know we have match the c with this because, if we take a c the matching automatically happens. So, you get the next token and then get out of the switch statement. Any other symbol in the input implies an error.

(Refer Slide Time: 18:23)



An Example (contd.)

```
void fA() /* A --> ba | SB */
{
    switch token {
        case b : get_token();
                if (token == a) get_token();
                else error(); break;
        case a,c : fS(); fB(); break;
        others : error();
    }
}

void fB() /* B --> bA | S */
{
    switch token {
        case b : get_token(); fA(); break;
        case a,c : fS(); break;
        others : error();
    }
}
```

NPTEL

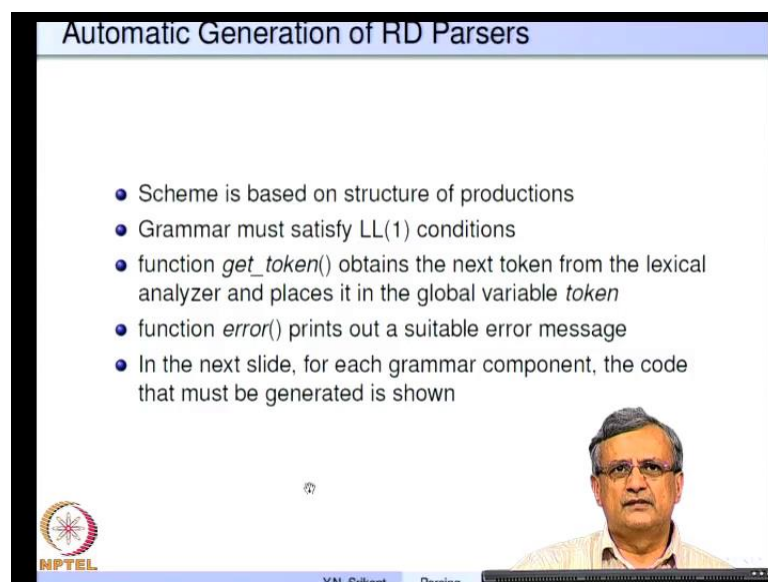
YN. Srikant Parsing

Then we have the function for the non terminal A. So, again there are two alternatives one corresponds to b a, the other one corresponds to S B, b a starts with little b. So, we have a case for little b, if the token is little b then get the next token. If the next token happens to be little a, that is the one symbol here again get the next token, otherwise an

error message is given and then we get out. For the symbol is here S so, to take this particular alternative you must make sure that the input symbol that is a token is either a or c which corresponds to the first of capital S.

So, in that case we call the function corresponding to the non terminal S then we call f B. The function corresponding to non terminal B and then break, for any other symbol it is an error. So, finally, the function for non terminal B, it again has two options b A and S. So, we again have a similar flow for b we get the next token and call f A and finally, break. For the first of S, which is a comma c we call f S and break, for others it is an error.

(Refer Slide Time: 20:00)

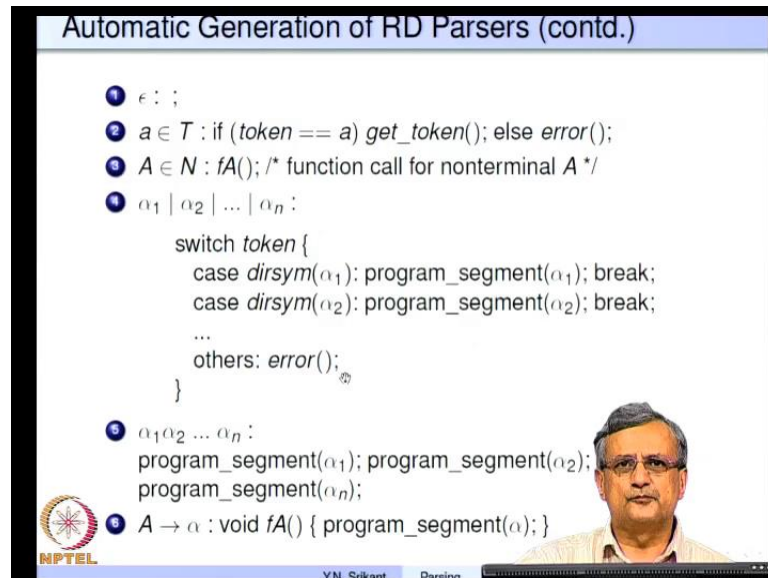


The slide is titled "Automatic Generation of RD Parsers" and contains a list of five bullet points. In the bottom right corner, there is a small video inset showing a man with glasses speaking. The NPTEL logo is in the bottom left corner, and the text "Y.N. Srikant" and "Parsing" is visible at the bottom of the slide.

- Scheme is based on structure of productions
- Grammar must satisfy LL(1) conditions
- function *get_token()* obtains the next token from the lexical analyzer and places it in the global variable *token*
- function *error()* prints out a suitable error message
- In the next slide, for each grammar component, the code that must be generated is shown

So, how do we generate recursive descent parsers automatically? Of course, the scheme is based on the structure of productions. So, the generator actually looks at the productions and generates the program we will we are going to look at some more details of this the grammar must of course, satisfy the L L 1 conditions. So, we are going to check the L L 1 condition and then call the generator. Function get token obtains the next token from the lexical analyzer, and places it in the global variable token this is the assumption function error prints out a suitable error message.

(Refer Slide Time: 20:45)



Automatic Generation of RD Parsers (contd.)

- 1 ϵ : ;
- 2 $a \in T$: if (*token* == *a*) *get_token*(); else *error*();
- 3 $A \in N$: *fA*(); /* function call for nonterminal A */
- 4 $\alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n$:
 switch *token* {
 case *dirsym*(α_1): *program_segment*(α_1); break;
 case *dirsym*(α_2): *program_segment*(α_2); break;
 ...
 others: *error*();
 }
- 5 $\alpha_1 \alpha_2 \dots \alpha_n$:
 program_segment(α_1); *program_segment*(α_2);
 program_segment(α_n);
- 6 $A \rightarrow \alpha$: void *fA*() { *program_segment*(α); }

MPTEL

Y.N. Srikant Parsing

And now, we are going to see the details. So, the way it is given here for if the, you know right hand side happens to be epsilon, if there is a production $A \rightarrow \epsilon$ and let us start with the last one first. So, there is a production $A \rightarrow \alpha$. So, then we are going to write a function *fA*, which returns nothing and does not take any parameters and the body of the function corresponds to the program segment generated for alpha.

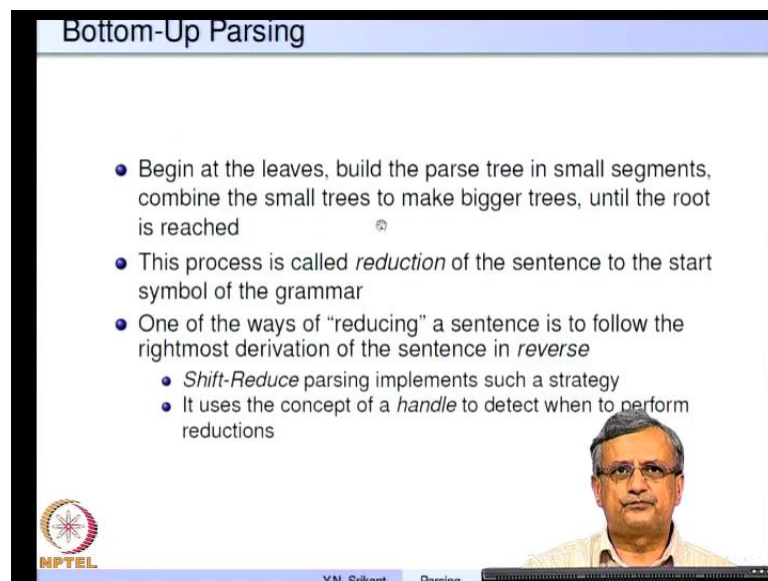
So now, we are going to see various possibilities for alpha and then mention the code which can be generated. So, if alpha is epsilon then we just generate the skip that is semicolon if alpha is a single non terminal a, say single terminal a then we generate the program segment if *token* equal to a *get_token* else *error*. So this is intuit, we saw this in the program segment example before. If A is non terminal then we generate a call *fA* function call for the non terminal A, if alpha consist of several alternatives alpha 1, alpha 2, alpha 3, etcetera, etcetera then we generate a switch on the token and for all the symbols in the directions symbols set of alpha 1 instead of first we use direction symbol because, alpha 1 or alpha 2 etcetera may also be epsilon and then the follow automatically kicks in.

If the symbols begin alpha 1 you know. So, that is directions symbol of alpha 1 all the symbols are listed here then the program segment corresponding to alpha 1 is generated and a break is generated as well. Similarly for alpha 2, alpha 3, etcetera, etcetera and for any other symbol it is error. So, we saw an example of these two already the first one,

you know begins the first option and the second set begins the second option etcetera, etcetera. If the string alpha can be of the form alpha 1, alpha 2, alpha n then we generate the program segments for alpha 1, alpha 2, alpha n and concatenate them in the same order.

So, this is the scheme for generating the program segment for you know a recursive descent parser. So, let us just go back and see one of the examples again. So, this is a production of the form $a \rightarrow ba$ or $S \rightarrow b$. So, there is a function for the non terminal a and then because, there are alternatives there are there is a switch as well. And within the switch the direction symbols set of ba is b and the direction symbols set of Sb is a comma c . So, that way we generate get token and then check the. So, this is the program segment generated for the string ba and this is the program segment generated for the string Sb . So, within this, this is of the form alpha 1, alpha 2 this is again of the form alpha 1, alpha 2. So, we go on applying the rules of generation and generate the appropriate program segments.

(Refer Slide Time: 24:32)



The slide is titled "Bottom-Up Parsing" and contains the following text:

- Begin at the leaves, build the parse tree in small segments, combine the small trees to make bigger trees, until the root is reached
- This process is called *reduction* of the sentence to the start symbol of the grammar
- One of the ways of "reducing" a sentence is to follow the rightmost derivation of the sentence in *reverse*
 - *Shift-Reduce* parsing implements such a strategy
 - It uses the concept of a *handle* to detect when to perform reductions

The slide also features the NPTEL logo in the bottom left corner and a portrait of a man in the bottom right corner. The footer of the slide reads "YN Srikant Parsing".

So, now we move on to bottom up parsing. So far, we have discussed the top down parsing strategy in which the starts symbol was expanded progressively and finally, we reach the leaves which correspond to the string to be parsed. In the case of bottom up parsing we begin at the leaves build the parse tree in small segments, combine the small trees to make bigger trees until the root is reached. So, that is why is called as a

bottom up strategy. So, this process is called reduction of the sentence to the start symbol of the grammar. So, one of the ways of reducing a sentence is to follow the rightmost derivation of the sentence in reverse. So, let me give you examples to show how this works. So and shifted using shift reduced parsing implements one such strategy and it uses the concept of what is known as a handle to detect when to perform such reductions.

(Refer Slide Time: 25:51)

Shift-Reduce Parsing

- **Handle:** A *handle* of a right sentential form γ , is a production $A \rightarrow \beta$ and a position in γ , where the string β may be found and replaced by A , to produce the previous right sentential form in a rightmost derivation of γ . That is, if $S \Rightarrow_{rm}^* \alpha A w \Rightarrow_{rm} \alpha \beta w$, then $A \rightarrow \beta$ in the position following α is a handle of $\alpha \beta w$.
- A handle will always eventually appear on the top of the stack, never submerged inside the stack.
- In S-R parsing, we locate the handle and reduce it by the LHS of the production repeatedly, to reach the start symbol.
- These reductions, in fact, trace out a rightmost derivation of the sentence in reverse. This process is called pruning.
- LR-Parsing is a method of shift-reduce parsing.

NPTEL
Y.N. Srikant, Parsing

So, let me show you some definitions and then we move on to examples. So, what exactly is a handle? A handle of a right sentential form γ is a production $A \rightarrow \beta$ and a position in γ . So, let us take a derivation of this form $S \Rightarrow_{rm}^* \alpha A w$. So, this is a rightmost derivation in many steps. And now, we replace αA by β and the context in which we apply is α on the left side and w on the right side.

So now, in this sentential form $\alpha \beta w$ we say that $A \rightarrow \beta$ in the position following α is a handle of the string or sentential form $\alpha \beta w$. So, basically we want to locate the right hand side of the production which is applicable at this point. So, that is why, that is the reason is called as a handle. Handle of right sentential form γ is a production $A \rightarrow \beta$ and a position in γ , where the string β may be found and replaced by the non terminal A , to produce the previous right sentential form in a rightmost derivation of γ . So, from here, if we replace β by A , we get this particular sentential form. Similarly we progressively replace handles by their left hand

side non terminals to reach start symbol S. This is the strategy which is used in shift reduced parsing.

So, handle will always eventually appear on the top of the stack and is never submerged inside the stack, this is a very important property. What it says, this is alpha is in this stack and this w is the rest of the input which is not parse. And now, we have beta also just on the top of the stack. So, when we want to remove beta and replace it with a we just have to pop an appropriate number of symbols from this stack. What it really says is it is not mixed up with alpha, but it is at the top of the stack. So, in shift reduce parsing we locate the handle and reduce it by the left hand side of the production repeatedly to reach the star symbol. So, we are going to see the examples of this. These reductions in fact, trace out a rightmost derivation of the sentence in reverse. So, this is known as handle pruning. And lr parsing is a method of shift reduce parsing we are going to study that in some detail later.


(Refer Slide Time: 28:59)

Examples

① $S \rightarrow aAcBe, A \rightarrow Ab \mid b, B \rightarrow d$
 For the string = *abcde*, the rightmost derivation marked with handles is shown below

$$\begin{aligned}
 S &\Rightarrow aAcBe \quad (aAcBe, S \rightarrow aAcBe) \\
 &\Rightarrow aAcde \quad (d, B \rightarrow d) \\
 &\Rightarrow aAbcde \quad (Ab, A \rightarrow Ab) \\
 &\Rightarrow abcde \quad (b, A \rightarrow b)
 \end{aligned}$$

The handle is unique if the grammar is unambiguous!


Y.N. Srikant Parsing

So, let us look at examples of what handles are and how they relate to rightmost derivation. Here is a grammar S going to a A c B e, A going to A b or b, B going to d. So, let us consider the string a b b c d e. So, the rightmost derivation is given here. So, for S we apply the production a A c B e and then you know we apply the production, B going d to get the form a A c d e then you know, we apply the production A going to A b

to get the form $a A b c d e$. And finally, we apply the production $A \rightarrow b$ to get the form $a b b c d e$. So, this is a rightmost derivation of the string $a b b c d e$.

Because, every time we replace only the rightmost non terminal, you know rather we expand the rightmost non terminal. So, now, let us traverse this particular derivation in reverse. So, the rightmost derivation runs in this order the reverse of it would be this particular order. So, this is our given input string $a b b c d e$ at the shift reduce parser let us not worry about the details of how. It locates that this second b is the handle. So, it says let me now reduce little b to capital a . If it does that then we get this sentential form in this, it locates the handle as $a b$. So, the production is applicable is a to $a b$, a to $a b$ you know is apply at this point. So, we replace $a b$ by a and we get this sentential form.

So, in this form it the shift reduce parser says d is the handle. So, and the production applicable at this point is determine to be b to d . So, this little d is replaced by b to get this sentential form. So, now, it determines that this entire right hand side is the handle and the production applicable is S going to $a a c b e$. So, it replaces this entire right hand side by S and since we have reach the start symbol the parsing is completed. So, this is how you know the shift reduce parser works it starts from the inputs string and reaches the start symbol. So, the handle is unique, if the grammar is unambiguous. if the grammar is ambiguous then you know it is not possible to use this strategy in a very simple way.

Another example, so, we have $S \rightarrow a A S$ or c , $A \rightarrow b a$ or $S B$, $B \rightarrow b A$ or S . So, let us take the string $a c b b a c$. So, the string is here and the rightmost derivation is in this order. So, you can observe that the non terminal on the rightmost at the, in the rightmost position has been expanded. So, S to c then b is expanded then a is expanded then S is expanded and finally, we get the string. So, the handles are handle is really is underlined here, c is the handle. So, the production is S to c so, we replace c by S we get this sentential form, $b a$ is the handle so, it is replaced by a so, we get this. Here again $b a$ is the handle. So, we get $b a$ replace $b a$ by b and we get this sentential form. S b is the handle so, that is replaced by a to get this form. c is the handle so, that is replaced by S because, S to c is the production to get this sentential form. And this entire thing is the handle and that would be reduce to S using the production $S \rightarrow a A S$



(Refer Slide Time: 33:25)

Examples (contd.)

• $E \rightarrow E + E, E \rightarrow E * E, E \rightarrow (E), E \rightarrow id$
 For the string = $id + id * id$, two rightmost derivation marked with handles are shown below

$E \Rightarrow \underline{E + E} (E + E, E \rightarrow E + E)$
 $\Rightarrow \underline{E + E * E} (E * E, E \rightarrow E * E)$
 $\Rightarrow E + \underline{E * id} (id, E \rightarrow id)$
 $\Rightarrow E + \underline{id * id} (id, E \rightarrow id)$
 $\Rightarrow \underline{id + id * id} (id, E \rightarrow id)$

$E \Rightarrow \underline{E * E} (E * E, E \rightarrow E * E)$
 $\Rightarrow E * \underline{id} (id, E \rightarrow id)$
 $\Rightarrow E + \underline{E * id} (E + E, E \rightarrow E + E)$
 $\Rightarrow E + \underline{id * id} (id, E \rightarrow id)$
 $\Rightarrow \underline{id + id * id} (id, E \rightarrow id)$

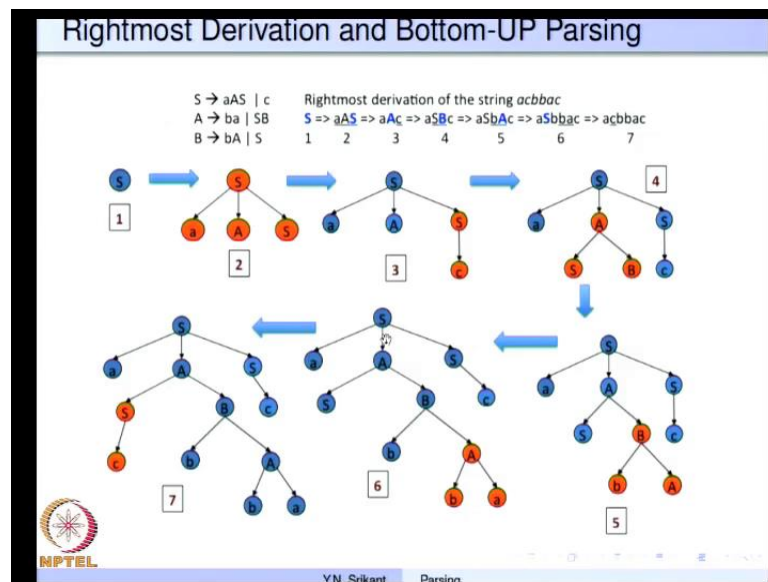
Y.N. Srikant Parsing

So, third example, this is the familiar expression grammar, this is ambiguous. So, let see what happens. So, the grammar is $E, E \rightarrow E + E, E \rightarrow E * E, E \rightarrow (E), E \rightarrow id$. So, we know very well that $E id + id * id$ can be passed in two ways because, this is an ambiguous we get two parsers trees. So, the first interpretation says $id + id$ first and then $* id$. The second interpretation says $id * id$ first and then added to two id . So, these are the two derivations. So, we let us see what happens you know, we have id here and that will be replaced by E because of this production id is the handle. The second id is the handle here and that would be replaced by E again. The third id is the handle here and that would be replaced by E again.

Now, $E * E$ is the handle that gets replaced by E and $E + E$ is the handle and that gets replaced by E . Whereas, if we had use the other rightmost derivation, we have the same string again $id + id * id$ at this point the handle is unique. So, we get this sentential form again, this handle is unique. So, we get $E + E * id$. The third handle is also unique so, we get this right, E sorry, the third this is a one step has been missed by mistake. So, this becomes $E + E + E * E$. So, we get you know, we replace $E + E * E$ by E to $E + c$ is replace by E . So, this is what we get $E * id$ is what we get and then that becomes $E * E$. And finally, we this entire thing is $E * E$ is the handle and that gets reduce to E . So, $E * E$ this becomes id and this becomes $E + E * id$.

And now, this E has been replaced by i d and finally, we get i d plus i d star i d. So, the thing is, this is not the handle here you know. So, this is the handle there is a minor error in this slide. So, we get from E plus E we reduce it to E star i d and then finally, E star E and finally, E. So, the problem with this is when we have a particular step we really do not know whether this you know derivation sequence was followed or this derivation sequence was followed. So, locating the handle uniquely may not be possible. So, in this case, if this was the derivation sequence the handle at this point. In the third step was i d whereas if we had used this derivation sequence the handle in the derivation sequence would E plus E. So, this is because of the ambiguous nature of the grammar and the modal of the story is if the grammar is a ambiguous then handles cannot be located uniquely.

(Refer Slide Time: 37:05)



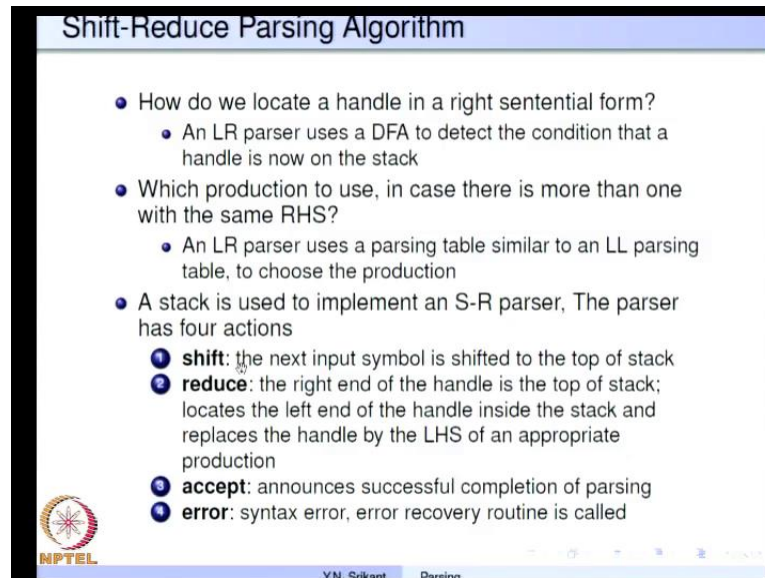
Here is an example to show how exactly the parsing happens with respect to parse trees. So, it is the same grammar again S going to a A S or c, A going to b a or S B, B going to b A or S. The rightmost derivation of the string a c b b a c is shown here and I have also underlined the, you know handle. So, here this is the rightmost derivation very simply like this and I am not going to discuss it in too much detail because it is a very simple thing whereas, what we discuss in detail is how exactly the parse tree is built up when it traverses the entire derivation in reverse. So, we start looking at the input symbol a to begin a c b b a c is the input string.

So, the first symbol is a and we can do no reductions here. So, we move on to the next step in which the little a remains as it is, but the next c gets replaced by S right. So, the second c replace by S. So, this is small pars tree which has been built here and this remains as it is. The third step, you know the, we have again replace this handle b a by a. So, the small pars tree corresponding to a going to b a has been added to this sequence. So, now we have a here, S to c another small pars tree the symbol be remains as it is and there is another small pars tree, which is has been built. So, this progresses the further and in the fourth step, we have replaced b a by b. So, this is the pars tree which has been built here. So, this little b and this pars tree have been combined to produce B going to using the production B going to b a.

So, here is a amalgamated pars tree and the other two remain as they are. Then step five we have S B being replaced by A. So, here is you know, the tree for S to c and here is the tree for b going to something. So, this S and this B are combined now by A. So, we have new pars tree in this form and the little a, which remain before also stays. and then we have step c in which this c is replaced by s. So, we have a little a, a small pars tree hanging by A and another small pars tree hanging by S. And finally, all these a a and s. So, this A, this entire pars tree routed at A and this S which is part of this pars tree are combined to produce this entire big pars tree and there is nothing more to do this is the root of the pars tree.


So, the parsing strategy starts from input build small pars tree combines them on the way and finally, reaches the start symbol. So, this is the big pars tree for this entire string. So, this is the process of bottom up parsing.

(Refer Slide Time: 40:48)



Shift-Reduce Parsing Algorithm

- How do we locate a handle in a right sentential form?
 - An LR parser uses a DFA to detect the condition that a handle is now on the stack
- Which production to use, in case there is more than one with the same RHS?
 - An LR parser uses a parsing table similar to an LL parsing table, to choose the production
- A stack is used to implement an S-R parser, The parser has four actions
 - 1 **shift**: the next input symbol is shifted to the top of stack
 - 2 **reduce**: the right end of the handle is the top of stack; locates the left end of the handle inside the stack and replaces the handle by the LHS of an appropriate production
 - 3 **accept**: announces successful completion of parsing
 - 4 **error**: syntax error, error recovery routine is called

 Y.N. Srikant Parsing

Now, let us study the shift reduce parsing algorithm. So, so far we said, we locate a handles and then the replace the handle by the right hand side, by the left hand side non terminal of the production and so on and so forth. We have not answer the question of how exactly do we locate a handle in a right sentential form. In the case of L R parser which we are going study later in detail it uses a deterministic finite state machine to detect the condition that handle is now on the stack. So, how to do that is actually F I complicated process and we will study that later.

The next question is which production to use in case there is more than one with the same right hand side, that is possible, but the question is answer by the L r parser using a parsing table similar to a L L parsing table to choose the production which is applicable. It so, happens that the state of the DFA tell us not only that it is a handle, but it also tells us the production which is applicable at that point. Then the third component is a stack a stack is used to implement a shift reduce parser and the shift reduce parser is nothing but an augmented deterministic push down automaton. It has several actions four to be precise. There is a shift action which shifts the next input symbol to the top of the stack. Then there is a reduce action the right hand of the handle is on the top of the stack.

So, it locates the left hand of handle inside the stack by looking at the number of symbols the right hand side is made up of, replaces the handle by the LHS of an appropriate production which is applicable at this point. So, you can observe that the reduction

process consist of popping symbols from the stack and pushing some symbols on to the stack. So, again these moves can be coded in terms of an ordinary deterministic push down automaton. So, that is why I said this is an augmented version of the dpda, which is more amenable for programming. Then it has an accept action, which says, yes parsing is completed and is successful and of course, it may announce an error it says the input the string is erroneous. So, an error recovery routine is called to make sure that parsing can be continued.

(Refer Slide Time: 43:52)

S-R Parsing Example 1		
<p>§ marks the bottom of stack and the right end of the input</p>		
Stack	Input	Action
§	acbbac§	shift
§ a	cbbac§	shift
§ ac	bbac§	reduce by $S \rightarrow c$
§ aS	bbac§	shift
§ aSb	bac§	shift
§ aSbb	ac§	shift
§ aSbba	c§	reduce by $A \rightarrow ba$
§ aSbA	c§	reduce by $B \rightarrow bA$
§ aSB	c§	reduce by $A \rightarrow SB$
§ aA	c§	shift
§ aAc	§	reduce by $S \rightarrow c$
§ aAS	§	reduce by $S \rightarrow aAS$
§ S	§	accept

So, let us take some examples to see, how exactly the parsers, shifted use parse works. So, we have the same a c b b a c input. So, in this case it shift, the action is supposed to be shift so, it shifts on to the stack. The next action is also shift. So, the parser shifts a c is also on to the stack. Now, the parser determines that reduction by S to c is in order. So, it replaces the top c by S. So, the next three symbols, next three actions are shift. So, we get a S b b on the stack and a c in the input, now the reduction is called for a. So, because the stack has b a as the handle and the production applicable is a to b a. So, b a is replaced by A and the only symbol which remains in the input is c. So, now, b a is again set to be a handle.

So, the replacement by b application of b to b a happens and b a is replaced by b. Again there is a handle here S B. So, S B is replaced by A and we get a A. Now, there is a shift and c is push down to the stack. Again c is reduce to S and A S finally, reduces to S and

here the shift reduce parser accepts. So, these are the exactly the same sequence of actions that were actually perform when we traverse the rightmost derivation in reverse it is just that they are coded in the form of the actions of the parser.

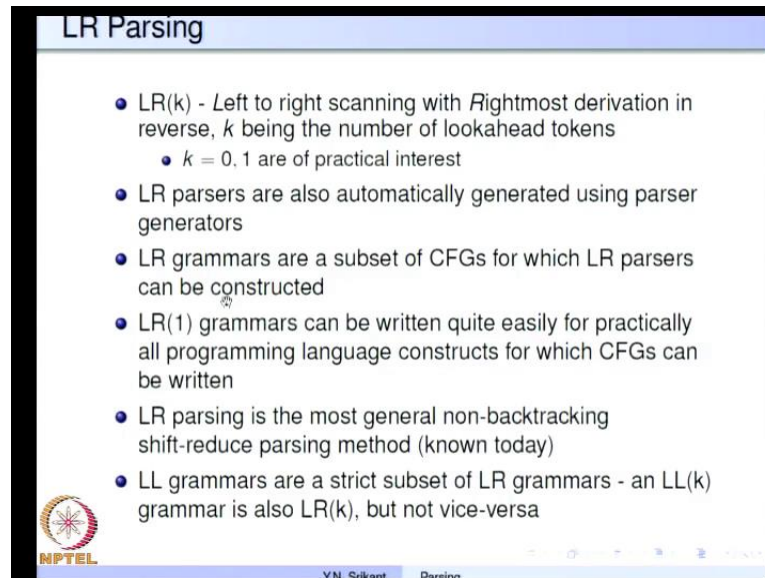
(Refer Slide Time: 46:02)

S-R Parsing Example 2		
§ marks the bottom of stack and the right end of the input		
Stack	Input	Action
§	$id_1 + id_2 * id_3 §$	shift
§ id_1	$+ id_2 * id_3 §$	reduce by $E \rightarrow id$
§ E	$+ id_2 * id_3 §$	shift
§ $E +$	$id_2 * id_3 §$	shift
§ $E + id_2$	$* id_3 §$	reduce by $E \rightarrow id$
§ $E + E$	$* id_3 §$	shift
§ $E + E *$	$id_3 §$	shift
§ $E + E * id_3$	§	reduce by $E \rightarrow id$
§ $E + E * E$	§	reduce by $E \rightarrow E * E$
§ $E + E$	§	reduce by $E \rightarrow E + E$
§ E	§	accept

So, that a program can be written for it, here is another example from our expression grammar and parser. So, $id_1 + id_2 + id_3$ combined with plus and start that is the input. So, id_1 is shifted on to the stack then a reduction to E happens then plus is pushed on to the stack.

And id_2 is also pushed on to the stack, again id_2 is reduce to E , star is pushed on to the stack, id_3 is also pushed on to the stack, again id_3 is reduce to E . Now, $E * E$ is reduce to E and finally, $E + E$ reduce to E and the parser accepts. So, this is second example of how the shift reduce parser really works.

(Refer Slide Time: 46:57)



The slide is titled "LR Parsing" and contains the following content:

- LR(k) - Left to right scanning with *Rightmost* 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-backtracking shift-reduce parsing method (known today)
- LL grammars are a strict subset of LR grammars - an LL(k) grammar is also LR(k), but not vice-versa

At the bottom left of the slide is the NPTEL logo. At the bottom center, the text "Y.N. Srikant" and "Parsing" is visible.

Let us move on to a specific type of shift reduce parser called the L R parser because, these are the parsers which are of practical interest to us. Of course, I must mention that there were in the olden days other types of shift reduced parsers known as operator precedence parsers, which were very popular at that time, but once the L R parsing strategy was proposed people found that this has much more a much wider application than operator precedence grammars.

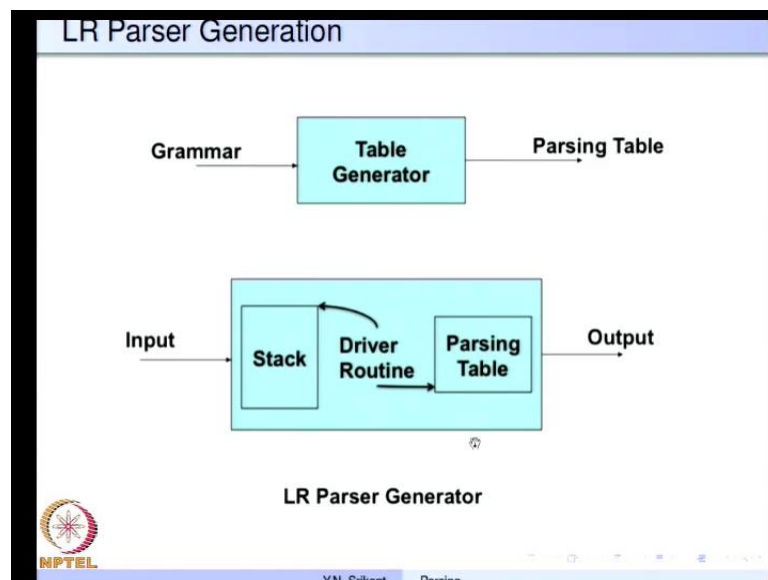
And therefore, in the recent years the parsers are all L R parsers and the tools that we have or the once we generate L R parsers automatically from the grammars. So, here again just like the L L k we have the L R k so, left to right scanning with rightmost derivation in reverse, k being the number of look ahead tokens. So, in the case of L L k was left to right scanning with leftmost derivation. Here it is left to right scanning with rightmost derivation in reverse not just rightmost derivation and k as usual is the number of look ahead tokens. So, of practical significant as just L R 0 and L R 1 really speaking L R 1 is a most important. These parsers can be automatically generated using parser generators and YACC is a an L R parser generator available under Unix. L R grammars are a subset of context free grammars for which L R parsers can be constructed.

So, remember just like the L L grammars the L R grammars are also subsets of general context free grammars. And a particular grammar for a language may not be L R 1 or L R 2 and we may be able to rewrite some of these grammars to become L R 1 or L R 2,

but it does not mean that every grammar that is written for a language will pass the LR 1 or LR 2 test we should be smart enough to write the grammar such that the test actually is completed successfully. So, LR 1 grammars fortunately can be written quite easily for practically all programming language constructs for which context free grammars can be written it just needs a little practice. And LR parsing happens to be the most general non backtracking shift reduce parsing method that is known today.

So, these LR grammars are a strict subset of LR grammars. So, an LR k grammar is also LR k, but an LR k grammar is need not necessarily be LR k. So, you can, that means, you can always find LR k grammars which are not LR k.

(Refer Slide Time: 50:27)



This is the block diagram of an LR parser generator. So, the grammar is input to a table generator this is the LR parsing table generator and outcomes a parsing table. And when we use the parsing table there is also a derive routine this thing is this block is entire parser. So, the parser consist of a stack, a derive routine and a and the parsing table which are generated automatically. So, when the input is given the derive routine reads the input manipulates the stack appropriately using the parsing table and generates some output in the form either a parser tree or error messages, etcetera, etcetera.

(Refer Slide Time: 51:20)

LR Parser Configuration

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

MPTEL
Y.N. Srikant Parsing

So, let us see what exactly is an L R parser configuration. So, the input you know is a 1, a 2, a 3, ... a n dollar and the starting state of the L R parser is assumed to be S 0. So, a configuration consist of you know a string of states and non terminals mixed here non terminals or terminals mixed here and unexpanded or unused input is the second part of the configuration. So, S 0, S 1, S m, etcetera are the sates of the parser and x 1, x 2, x m are the grammar symbols terminals or non terminals. So, the starting configuration is always S 0 and then the rest the entire input unused. So, there are two parts in the parsing table, the first part is called as the action part and the second part is called as the goto part. I will show you an example. The action table can have four types of entries shift, reduce, accept and error, which we have already seen and the goto table provides the next state information to be used after a reduce move.

(Refer Slide Time: 52:53)

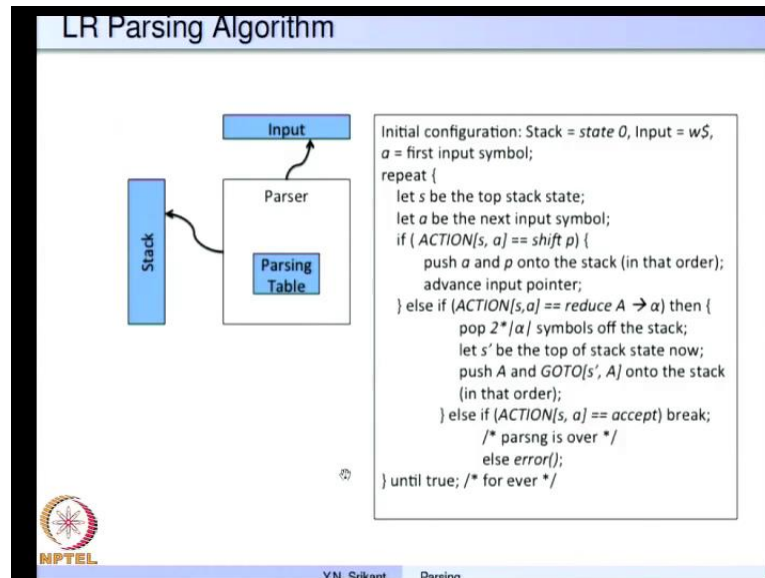
LR Parsing Example 1 - Parsing Table							
STATE	ACTION				GOTO		
	a	b	c	\$	S	A	B
0	S2		S3		1		
1				R1 acc			
2	S2	S6	S3		8	4	
3	R3	R3	R3	R3			
4	S2		S3		5		
5	R2	R2	R2	R2			
6	S7						
7	R4	R4	R4	R4			
8	S2	S10	S3		12		9
9	R5	R5	R5	R5			
10	S2	S6	S3		8	11	
11	R6	R6	R6	R6			
12	R7	R7	R7	R7			

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, let me show you a parsing table just to get the feel for it. So, the parsing table is indexed by the state number on this side. The action table action part is indexed by the tokens whereas, and of course, end of file and the goto part is indexed by the non terminals alone. So, for example, if you pick state two and a input symbol as b its says S 6. The interpretation of this is the action is shift action and straight to which we shift next is 6. So, what is done is, if the parser is in state two and the next input symbol is b, it shifts it on to the stack and also shifts the states number 6 on to the stack. Similarly suppose the state in which the parser is seven and for all the input symbols a b and c it says the action is R 4, R stands for reduction and the number 4 is not a state number, but it is the production number.

So, here it is a going to b a. So, the action which is done here is suppose the input symbol is a. The action is reduce using production 4, the implication is the handle b a is available on the stack. So, pop the handle from the stack it exposes some particular state. So, what we do is use that state along with the goto table to determine which state we should really go to. So, that is why, what is said here is the goto table provides the next state information to be used after a reduce move. This becomes clear when you take up a complete example.

(Refer Slide Time: 55:01)



But before that, Let us see how the LR parsing algorithm works. So, here is the parser it has a table, it has a stack and it looks at the input as well. It starts with the initial state 0 and the input is w dollar, a is the first input symbol. So, the whole thing is repeated forever until we get out of it. Let S be the top of stack state, a be the next input symbol. So, we look at S and a in the action part, if it is a shift P push a and P on to the stack in that order advance the input pointer. This is what I just now explained. So, if the action says reduced by A to α then pop two star α symbols off the stack the reason is the stack has a combination of both states and grammar symbols. So, there are two m symbols, if we the right hand side of the, if the handle is of the size n . So, we really pop two star α symbols of the stack. Now, the state S prime is exposed on the stack. So, push a and goto of S prime comma a on to the stack in that order.

So, this is how we determine the next state to be a jumped into. If the action is accept then we end otherwise, we announce an error and get out you know either do an error recovery or get out of the parser. So, we will stop at this point and continue in the next lecture.

Thank you.