

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

**Lecture – 25**  
**Parser (Contd.)**

(Refer Slide Time: 00:15)

The slide is titled "Conflicts during shift reduce parsing". It contains the following content:

- Two kind of conflicts
  - Shift/reduce conflict
  - Reduce/reduce conflict
- Example:

stmt	→	if expr then stmt
		if expr then stmt else stmt
		other

Stack                      Input

... if expr then stmt      else ... \$

Handwritten notes on the right side of the slide:

- if E then S<sub>1</sub>
- else S<sub>2</sub>
- A box containing: if E<sub>1</sub> then S<sub>1</sub> else if E<sub>2</sub> then S<sub>2</sub> else S<sub>3</sub>

The slide also features a logo for "swayam" and a small video inset of a man in a red and white checkered shirt in the bottom right corner.

So, in our last class we were looking into the shift reducing parsing policies and we have seen that there are four actions that can occur with shift reduce parsing, shift reduce, accept and errors. So, these were the four operations that a shift reduce parser will do. Now, many a time we will see that it may happen that there will be conflict in because the as depending upon the grammar the parser may not be able to decide uniquely whether to do a shift operation or a reduce operation. So, that is known as a shift reduce conflict.

So, this is that shift reduce conflict. So, parser cannot decide whether to shift or to reduce and there is another conflict which is known as reduce reduce conflict where we can there are more than one rule by which we may try we may do the reduction. Of course in future what can happen is that as you see more tokens, so maybe one of these reductions are valid.

So, as a result if you do not do enough look ahead. So, there will there will be conflicts and those conflicts cannot be resolved at the first level it itself. So, that type of situation

will give us reduce reduce conflict. So, if we want to modify the grammar for removing this conflicts that is better if not then we have to take some default action. And the default action for shift reduce conflict is doing a shift and for a reduce reduce conflict the default action is whichever reduction rule comes first in the set of production rules.

So, that will be taken as the rule by which to do the reduction. So, there cannot be any shift shift conflict because in both the cases we are going to shift to the next input symbol. So, there is no problem with that. So, now, there is nothing like shift shift conflict and shift accept or reduce accept. So, this sort of conflicts also cannot occur because there we are already in the accept state. So, there is no further action to be taken

Now, let us take some example and try to see how this shift reduce conflicts it can occur like in this case. So, what we have is say this particular grammar then if else grammar. So, if statement producing a producing if expression then statement or if expression then statement else statement or other statements; now see at some point of time the situation may be like this that in the stack we already have these tokens.

If expression then statements so up to this much we have seen, so in this expression and statements. So, these are non terminals and if and then they are terminal. So, as we know that the stack can contain the all grammar symbols both terminals and non terminals. So, suppose at some point of time this is the situation where statement is at the top of the stack. And then the next input that we have is the else, next token that we have is else.

Now what to do? So, one possibility is that we so this else is a part of this if then statement. So, this else has to be shifted into the step. Other possibility is so it is so there was a nested if. So, there the situation is like this. So, there was a there is a nested if. So, I have got some if expression then S 1 else S 2, so this is one possibility. Other possibility is that if E then say if E 1 then S 1 if E 2 then S 2 else S 3.

So, this is the situation on which there will be a shift reduce conflict because when we see these particular else. Now what to do? So, whether it should be shifted because whether it is giving to it is going to give me this if then else statement or. So, I will reduce up to this, I will reduce up to this, and these else becomes a part of this outer E outer if. So, thus so that is the conflict ok. So in that case the parser will not be able to take a unique decision whether to do a shift or a reduce.

And as I said that default action is to do shift. So, if you shift it then what happens is that this else S 3 so, this becomes a part of the inner most if. So, and most of the programming languages they also tell that way that the else is always associated with the inner most if and in that case shift is a valid action. So, that is one example of shift reduce conflict. Next we will be looking into the reduce reduce conflict.

(Refer Slide Time: 05:01)

**Reduce/reduce conflict**

$\text{stmt} \rightarrow \text{id}(\text{parameter\_list})$  ← procedure call  
 $\text{stmt} \rightarrow \text{expr} := \text{expr}$   
 $\text{parameter\_list} \rightarrow \text{parameter\_list}, \text{parameter}$   
 $\text{parameter\_list} \rightarrow \text{parameter}$   
 $\text{parameter} \rightarrow \text{id}$  ← array  
 $\text{expr} \rightarrow \text{id}(\text{expr\_list})$  ← array  
 $\text{expr} \rightarrow \text{id}$   
 $\text{expr\_list} \rightarrow \text{expr\_list}, \text{expr}$   
 $\text{expr\_list} \rightarrow \text{expr}$

Stack: ... id(id)      Input: ... id(id) ... \$

Handwritten notes on the slide include:  $\text{proc}(x, y, z)$ ,  $\text{Arr}(x, y, z)$ , and  $\text{id}(\dots)$  with arrows pointing to the corresponding parts of the grammar rules.

So, like this suppose I have got a grammar that has got both that has got both so, this is a procedure call. So, this is a procedure call and then this may be some array list, this may be an array. So, if there is a procedure call so say proc 1 there I can pass this parameters say x, y and z maybe it has got 3 parameters. Now there may be another array Ar and there also I have got the, I have got the arguments or the array indices they are also expressions. So, that is also say x, y, and z some expression.

Now, in so this x, y, z as far as tokens are concerned so, all of them will be taken as id. So, in one case you have got this situation that id followed by id followed by ideally I id followed by this parameter list where this parameter list is again an expression. So, this is parameter list giving parameter and the parameter is ultimately giving id, so ultimately it is giving id comma id like that. And other situation is that we have got this array and that after that array also we have got this array name is coming as id. And then we have got this expression list which is the array indices. So, these array indices so they also come as id.

Now, if you are at a situation like this. So, we have seen id within bracket id and the next symbol is a next token is a comma. Now what to do? So, there can be one possible reduction like this by this one parameter giving id and there is another reduction expression giving id. So, which one to do we really do not know. So, until and unless we know that very recently we have seen a procedure call then in that case this reduction should be by this.

And if you on the other hand if you assume that very recently we have seen one array call array array name as the portion before the open parenthesis then we know that this is going to be an expression list. So, you should do go by expression producing id. So, there is a confusion so just by looking into this top of the stack and this comma you cannot take a decision by which rule to do the reduction. So, this gives rise to reduce reduce conflict.

So, these conflicts are to be avoided because if a grammar has got this type of conflicts. Then in the parsing process then the parser will not be able to proceed properly and there will be difficulty in parsing the input sequence. So, default rules are there, but it is not mandatory that default rules will always be play applicable. So, depending upon the language, so they may not be applicable also. So, we have to be very careful about these conflicts. So one of the basic responsibilities for this parser designer is to modify the grammar so that this conflicts can be resolved.

(Refer Slide Time: 08:19)

The slide is titled "Bottom-Up Parsing" in a large, bold, black font. Below the title, there is a bulleted list:

- Operator Precedence Parsing
- LR Parsing

Handwritten in black ink next to "LR Parsing" is a diagram showing three arrows pointing downwards from "LR Parsing" to "SLR", "Canonical LR", and "LALR".

The slide has a yellow background with a blue border at the top and bottom. At the bottom, there is a blue banner with the "swayam" logo and text. In the bottom right corner, there is a small video feed of a man with a beard and a red and white checkered shirt.

So we will be looking into two types of bottom up parsing strategies one is known as operator precedence parsing and another is a class of parsers known as LR parsers out of these two operator precedence parsing this is very simple. So, for a particularly for language that just accept expressions; so, basically the expressions expression grammar, so they can be parsed using this operator precedence parsing.

So, we will see there are certain rules that will define what is an operator grammar and all. And in general other parsing method so we have got this LR parsing and this LR parsing we will see that it can further be divided into number of categories. We will look into something called SLR or simple LR parsing then we will look into something called canonical LR canonical LR, so which is more generic in nature.

So, SLR is pretty simple out of these three alternatives that we have in error parsing SLR is going to be pretty simple and many a times. So, we can we can construct the parser by hand. On the other hand this canonical LR. So, it is difficult to construct by hand and it has got a large number of states compared to an SLR parsers. So, it has got a large number of states.

On the other hand this a LALR there is another parser known as LALR which is a full form is look ahead LR. So, this parser, so this will have less number of states in fact, the number of states that LALR parser will have is same as the number of states that you have in SLR, but it can be more powerful than the SLR, so that way it is better.

So, most of the automated tools that we have that we have talked about previously like Yak, Bison etcetera; so, they generate LALR parser for a grammar; however, LALR is difficult to learn for our class. So, we will be we will be first learning SLR and then go towards the other categories. So, let us start with this operator precedence parsing.

(Refer Slide Time: 10:31)

## Operator Grammar

- No  $\epsilon$ -transition
- No two adjacent non-terminals

Eg.

$$E \rightarrow E \text{ op } E \mid \text{id}$$
$$\text{op} \rightarrow + \mid *$$

*→ Not operator grammar*

The above grammar is not an operator grammar

but:

$$E \rightarrow E + E \mid E * E \mid \text{id}$$

So, operator precedence parsing is applicable for operator grammars. So, a grammar will be said to be an operator grammar. If it does not have any epsilon transition and in no production rule right hand side you will have two adjacent non terminals like say this particular grammar, so E producing E of E. So, you see this operator so, this is this operator we can mod. So, this grammar whether it is it operator grammar so it is not an operator grammar because you see this E and op. So, they are two non terminals. So, they are appearing side by side. So, this is not a operator grammar. So, this is not operator grammar

However we can modify this grammar a bit we can modify this grammar a bit and we can write it like this. So if we substitute this o p in the first rule, so you get a grammar like this and here you see we do not have this situation that is two adjacent non terminals so that condition never occurs. So, it is always separated by a terminal symbol. So, this is an operator grammar and also it does not have any epsilon transition. So, this is an operator grammar. So, this is fine. So, we can use this grammar for operator precedence parsing.

So, once so given a grammar you first check whether this is an operator grammar or not if it is not an operator grammar we have to check by doing some simple modification to the grammar is it possible to convert it into an operator grammar. So, if we can do that

then we can try to frame the operator precedence parsing table and follow the operator the operator. We can follow the operator precedence parsing.

(Refer Slide Time: 12:25)

## Operator Precedence

- If a has higher precedence over b;  $a > b$
- If a has lower precedence over b;  $a < b$
- If a and b have equal precedence;  $a = b$

Note:

- id has higher precedence than any other symbol
- \$ has lowest precedence.
- if two operators have equal precedence, then we check the **Associativity** of that particular operator

*a > b < c*

So, what do you mean by precedence of operators? So, from our mathematics classes we know that whenever we have got some operator, so there are some precedence. For example, addition and multiplication out of that in general multiplication has got the higher precedence than addition. So, in case of grammar so we will be talking about precedence between the terminal symbols; so, suppose a and b they are two terminal symbols. If a has higher precedence over b we will denote it like this. So, this is just a notation. So, we will read it as a a has higher precedence than b.

Another possibility is if a has lower precedence over b. So, we will be writing as a less than then a dot then b made and we will read it as a has lower precedence than b. And if a and b are of equal precedence then we will be writing like this a with a dot on the on top of an equality sign and then b. You see that many a times for our for the sake of simplicity we will simply write simply say a greater than b or a less than b like that or a equal to b.

But in general we will be we will be following we will be meaning this thing that is when I say greater than b. So, we I really mean that a is of higher precedence than b. So, so these are certain rules like I identifier has got higher precedence than any other symbol, dollar has the lowest precedence. And if two operators have equal precedence

then we check the associativity rule of that operator. So, this is in general for expressions. So, this is true for expressions that I any identifier it will have higher precedence over any other symbol and the dollar will have the lowest precedence and we have to follow associativity to decide the precedence.

So, if two operators are equal precedence like say a plus b plus c. So, a plus b plus c then this a b a a. So, this plus and this plus they are of equal precedence. So, we have to follow associativity rule in that case. So, anyway so for grammars that involves only arithmetic expressions. So, these rules are valid, but in general how to decide this precedence and also that we will see as we proceed in the lecture.

(Refer Slide Time: 14:59)

## Precedence Table

	id	+	*	\$
id		·>	·>	·>
+	<·	·>	<·	·>
*	<·	·>	·>	·>
\$	<·	<·	<·	·>

Example:  $w = \$id + id * id\$$   
 $\$ < \cdot id \cdot > + < \cdot id \cdot > * < \cdot id \cdot > \$$

So, this is a precedence table; so, following the previous rule that we have so we can say that this is the precedence rule. So, I we said that id has identifier has got higher precedence than any other symbol. So, here is so identifier is having higher precedence than any other symbol. Then it says that in case of plus so plus has got lower precedence than identifier and then this is I have to follow associativity. So, if I have got a plus b plus c then we do a plus b first and then do plus c.

So, plus has got the higher president over plus, similarly plus has lower precedence than star and plus has got higher precedence than dollar. So, all the terminal symbols they have got higher precedence than dollar. Similarly star it has got multiplication it has got lower precedence than identifiers, higher precedence than plus, higher precedence than

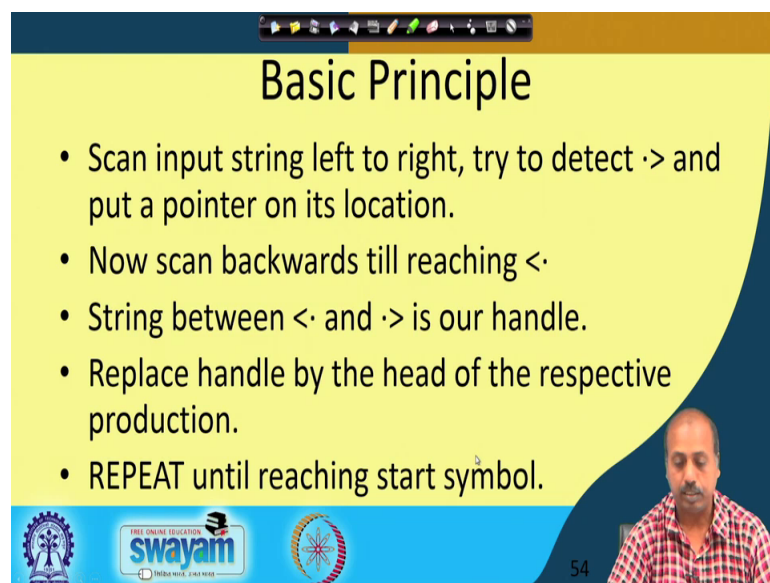


star, and higher precedence than dollar. And dollar has got lower precedence than everybody accepting dollar. So, this is the precedence table. So, this particular table we have framed by taking into consideration the arithmetic expression rules.

Now we will see later how to do it for a general grammar. Now, suppose we have got an example to be parse. So, id plus id star id. So, what we do? So, we put a dollar at the beginning and a dollar at the end and between any two terminals. So, we insert the corresponding precedence where precedence value.

So, if you consider this rule dollar and id. So, dollar is less than id. So, we are put less than then id and plus. So, id is greater than plus. So, this is greater than so that way we do it. Now it is said that anything that comes within this less than and greater than. So, that is a handle. So, this is a handle similarly if we have got this thing. So, this part will be a handle. So, that way it can identify the handles in the language in the thing.

(Refer Slide Time: 17:03)



### Basic Principle

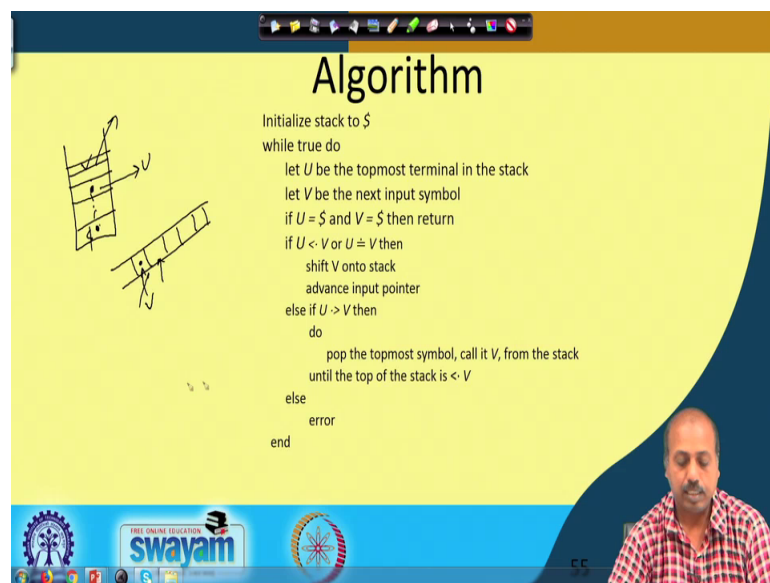
- Scan input string left to right, try to detect  $\cdot >$  and put a pointer on its location.
- Now scan backwards till reaching  $< \cdot$
- String between  $< \cdot$  and  $\cdot >$  is our handle.
- Replace handle by the head of the respective production.
- REPEAT until reaching start symbol.

54

So, so basic principle for this parsing is like this we scan the input string from left to right and we try to detect the greater than. So, this string that we have so I scan from left to right and try to see where first this greater than occurs. So, this is the first greater than. So, we detect the greater than and put a pointer on its location. So, we keep a note that we have seen a greater than at this point. So, keep a note will give a pointer here and then now we have to scan backwards till we reach the previous less than.

So, this one is done so we scan backwards to see what is the, we scan backwards to see where this less than appears. And once we find it the string between this less than and greater than is the handle we replace this handle by the head of the respective production. So, this handle will be replaced by E now so this is this. So, because E producing id was a rule, so E producing id was a rule. So, this handle will be replaced by E now. So, and then we repeat this process until we reach the start symbol. So, this way the parsing process will continue like this.

(Refer Slide Time: 18:31)



## Algorithm

```

Initialize stack to $
while true do
  let U be the topmost terminal in the stack
  let V be the next input symbol
  if U = $ and V = $ then return
  if U < V or U ≠ V then
    shift V onto stack
    advance input pointer
  else if U > V then
    do
      pop the topmost symbol, call it V, from the stack
      until the top of the stack is < V
    else
      error
  end

```

So, this is the overall algorithm. So, initialize the stack to dollar and we then we consider a at any point of time what we do is that we have got the stack. So, say we look into the top of the stack and if this is the input stream so we look into the input symbol. So, we compare this top of the stack element with the first input symbol. So, here we have put a dollar.

So, we compare dollar with the first input if U be the topmost terminal in the stack. So, this is the so what is the topmost terminal in stack so that we find out and V is the next input symbol. So, there may be some there may be some non terminal symbols, but suppose this is the topmost terminal symbol that we have here. So, this is our U, and this is the V. So, we compare between U and V so, if U is of equal precedence than dollar, if U is dollar or V is dollar in that case we have successfully parsed the string. So, we will come out.

If not if U is less than V U is of lower precedence than V or U is of equal precedence than like V then we will be shifting V into the stack. So, in that case V will be put into the stack and we will be advancing the input pointer to the next place and if U is greater than V precedence of U is more than the precedence of V then we pop out the topmost symbol ok.

Then we were from the stack call it V from the stack until the top of the stack is less than V. So, whatever symbol we pop out. So, until that popped out symbol is having a less than relationship with top of the stack has got a less than relationship with V. So, till that much we pop out. So, that will be popping out enough entries to that is that will pop that will be popping out a complete handle. Otherwise so if that also does not happen then there is an error.

So, error condition occurs when we some table entries are undefined like say here in this table, so id id is undefined, so and as we can intuitively understand. So, if there are two identifiers coming one after the other. So, that is meaningless because in an expression between two identifiers some operator must be there. So, if the operator is not there; that means, there is some error that is there is some error. So, this entry is a is an error entry. So, we can have many such error entries. So, if you find that there is no precedence relationship between U and V; that means, there is an error.

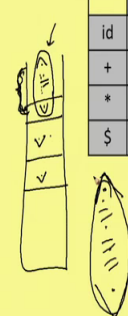
(Refer Slide Time: 21:11)

### Example

STACK	INPUT	ACTION/REMARK
\$	id + id * id \$	\$ < id
\$ id	+ id * id \$	id > +
\$	+ id * id \$	\$ < +
\$ +	id * id \$	+ < id
\$ + id	* id \$	id > *
\$ +	* id \$	+ < *
\$ + *	id \$	* < id
\$ + * id	\$	id > \$
\$ + *	\$	* > \$
\$ +	\$	+ > \$
\$	\$	accept

Stack:  
 id  
 +  
 \*  
 \$

	id	+	*	\$
id		>	>	>
+	<		<	>
*	<	>		>
\$	<	<	<	



So, let us see how this operation parsing process works. So, this is our input string, so id plus id star id. So, this is the input string so we have put a dollar at the end. So, we have put a dollar into the stack and then the rule says that you compare dollar with id and since you compare dollar and id. So, this is the a parsing table. So, dollar is less than id. So, dollar less than id so in that case the action is to shift id, so id has been shifted into the stack now it is id plus.

So, id plus so id is of higher precedence than plus. So, you so it is I so it is of higher precedence than plus. So, in that case it will be popping out the entries from idea from stack. So, id is popped out and the popped out symbol id has got top of the stack is dollar and that popped out symbol is id. So, if you look into this thing so top of the stack should have a less than relationship with the element popped out ok; so, that should be that case.

So, here you see when this id is popped out. So, dollar will be on the top of the stack and dollar has got less than relationship with id. So, it stops so this id is just popped out now between dollar plus. So, dollar plus dollar is of lower precedence than plus. So, plus will be shifted into the stack then we have got id. So, plus id so plus is of lower precedence than id. So, id is pushed into the stack. So, we have got star id here so, this id and star so id is of higher precedence than star.

So, we have to pop out entries. So, we pop out this id and so when we pop out between id and plus. So, id is of a higher prep, so as top of the stack now is now containing plus. So, plus is a of lower precedence than id, so it this popping out operation stops. So, we have got plus here and star here. So, between plus and star plus is of lower precedence than star so star is pushed into the stack then between star and id, star is of lower precedence than id, so id is pushed into the stack.

Now, between id and dollar so id is of higher precedence than dollar. So, so it will be id is of higher precedence. So, I have to pop out some entries so id is popped out between our now star is on the top between star and id star is of lower precedence than id. So, popping out operation stops now it is now the top of star and dollar so between the star and dollar. So, star is of higher precedence than dollar. So, it will be popping out the star star from the stack.

So, top of the stack contains plus and the symbol popped out is star and plus is of lower precedence than star. So, the popping out stops so it comes to this configuration between

plus and dollar plus is of higher precedence plus is of higher precedence than dollar. So, it will be popping out the plus symbol from the stack.

Now the top of the stack contains dollar and popped out symbol is plus. So, dollar is less than plus, so the popping out stops. So, at that at this point top of the stack contains dollar and the input is also dollar. So it comes to an accept state and the whole parsing process ends. So, if when you come to a situation that the top of the stack contains dollar, and the input is also having dollar; that means, we have seen the complete string.

So, it is called it is a shift reduce parsing because sometimes we are shifting the symbol whenever the next symbol that is coming on the input is whenever the top of the stack is of lower precedence than the next input symbol. So, we are pushing the symbol into the stack and whenever so that is a shift operation and whenever we are getting a situation where top of the stack is of higher precedence than the next input to come.

So, basically in the conceptually you can view it like this that as if. So, if this is my stack if this is my stack at some situation. So, we have got these symbols and there is that less than and greater than relationships. So, whenever the top of the stack is of lower precedence than id so it has been pushed into the stack. So, at any point of time if you have this all these symbols so they are of so there we have got the situation that this top of the stack is of lower precedence than id. So, this id the next symbol that has come, so this is of higher precedence; so, you have got a situation like this ok.

So, as if they are all of higher precedence then at some point of time if you get a symbol that is going to do it like this. So, whatever comes in between with all these equalities and all so this entire part is going to be one handle. So, this entire part is going to be one handle. So, this portion is popped out this portion is popped out from the stack and it is so that that is the reduction that we do. So, we can output it is explicitly not written that we do a reduction by that.

So, essentially it is doing a reduction by that rule and then we can, but then we can again proceed from this point. So, this way where so we basically try to figure out the situation where we have got in the stack a condition like this and in between we have got all equality things. So, this whole part becomes a handle and that handle is pruned by doing the reduction by the corresponding grammar rule.