

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

Lecture – 24
Parser (Contd.)

So, next we will be taking on Parsing example of this Boolean expression.

(Refer Slide Time: 00:25)

"id or id and not id"

Stack	Input	Action
B	id or id and not id	Parse $B \rightarrow TB'$
B'	"	$T \rightarrow FT'$
B'T	"	$F \rightarrow id$
B'TF	"	Advance input
B'TFid	"	$T \rightarrow \epsilon$
B'TFid	or id and not id	$B' \rightarrow \text{or } TB'$
B'TFid	"	Advance input
B'TFid	"	$T \rightarrow FT'$
B'TFid	id and not id	$F \rightarrow id$
B'TFid	"	Advance input
B'TFid	"	$T \rightarrow \text{and } FT'$
B'TFid	and not id	Advance input
B'TFid	"	$F \rightarrow \text{not } B$
B'TFid	not id	Advance input
B'TFid	"	$B \rightarrow TB'$
B'TFid	id	$T \rightarrow FT'$
B'TFid	"	$F \rightarrow id$
B'TFid	"	Advance input
B'TFid	"	

Suppose we take the exam id or id, id or id and not of id. Suppose this is the Boolean string that is given B or id and not of id ok. Now as we know that this predictive parsing processes it maintains stack and the input. So, we will be writing it like this, so there is a stack part there will be the input as it is there and then the corresponding action part ok. So, initially stack top will contain the start symbol of the grammar B and the input will be the whole string id or id and not id, this whole thing will be there in the input.

(Refer Slide Time: 01:37)

Handwritten notes:

$$\begin{aligned}
 B &\rightarrow TB \\
 B' &\rightarrow or\ T\ B' \\
 T &\rightarrow FT' \\
 T' &\rightarrow and\ FT' \mid \epsilon \\
 F &\rightarrow not\ B \mid (B) \mid true \mid false \mid id
 \end{aligned}$$

LR(0) Item Set Table:

	or	and	not	()	true	false	id	\$
B	$B \rightarrow or\ TB'$								
B'	$B' \rightarrow or\ TB'$								
T		$T \rightarrow and\ FT'$							
T'		$T' \rightarrow and\ FT'$							
F			$F \rightarrow not\ B$	$F \rightarrow (B)$	$F \rightarrow true$	$F \rightarrow false$	$F \rightarrow id$		

Handwritten First and Follow sets:

$$\begin{aligned}
 First(B) &= not, (, true, false, id \\
 First(B') &= or \\
 First(T) &= not, (, true, false, id \\
 First(T') &= and \\
 First(F) &= not, (, true, false, id \\
 Follow(B) &=), \$ \\
 Follow(B') &=), \$ \\
 Follow(T) &= or,), \$ \\
 Follow(T') &= or,), \$ \\
 Follow(F) &= and, or,), \$
 \end{aligned}$$

Now we have to see what was the rule for B id? So, if you look into this grammar this table it says B id says you have to go by B producing TB dash by this rule ok; so, by B going by B producing TB dash. So, it says parse B producing TB dash and they will be put into the stack. So, B is popped out from the stack and the stack will now contain B dash and T and input remains unaltered same as this one.

Now, I have to see what is T id? So, T id T id is stilling go by T producing F T dash ok. So, it will be so it says that parse by T producing F T dash. So, this T is going out of the stack now the stack will have B dash, T dash and F input remains unchanged. So, B dash T dash F now F now stack top is F and the input symbol id ok. So, F id you see F id says that you go by F producing id. So, it says that you go by F producing id and as a result it will be B dash T dash id, this remain same and then this. So, these two ids will now match this and this id will match. So, naturally the corresponding action is advance input pointer.

So, now the situation that we have is B dash T dash on the stack and or id. So, there is a dollar at the end we input is assumed to be ended by a dollar and not and dollar, so that is the situation. Now I have to see what is T dash or. So, T dash or is T dash producing epsilon. So, it says that the corresponding thing is T dash producing epsilon. So, T dash will go out of the stack, so now the stack will become B dash and input will be like this only.

Now B dash or so B dash or is telling me go by B dash producing or TB dash. So, it says the action is B dash producing or T B dash. So, they will be put into the stack. So, this B dash goes out and this B dash comes in, so B dash T or input remains same. So, now these 2 ors match, so we advance the input advance input. So, this or goes out now the stack contains B dash T and this has got id, input has got id and not id dollar fine.

Now, T and id, so T and id you see T and id says go by T producing F T dash this rule. So, it says go by T producing F T dash. So, this T goes out of the stack and T dash and F they are put into the stack input remains unchanged. Now F id, F id is we have seen previously that it will tell me to go by this rule F producing id. So, F will be popped out from the stack and this id will come in input will remain unchanged. Now these 2 ids these 2 inputs will these 2 ids will match. So, the action will be advance input it will be advance input. So, this stack will become B dash T dash and the input will now have and not id dollar.

Now, I have got T dash and now T dash and it says it go by T dash producing and F T dash. So, it says go by T dash producing and F T dash. So, this T dash is popped out from the stack then this new T dash F and they are put into the stack input remains unchanged. So, again these 2 ands match, so we have to advance input we have to advance input. So, it becomes B dash T dash F and then this and is has been consumed. So, it is not id dollar.

Now, F and not F and not says you go by F producing not B. So, it says go by F producing not B. So, this F is taken out from the stack. So, B dash T dash B and not input is as it is. Now these 2 not's will match, so it will be advancing input; it will be advancing input. So, this will become B dash T dash B and then it says that I have got id and dollar fine.

Now, B id, so B id says that go by B producing TB dash this rule. So, it says go by B producing T B dash. Now so this B will be going out, so the stack will contain B dash T dash B dash T and then this is id and this is dollar. Now T id so, T id says go by T producing F T dash ok. So, it says go by T producing F T dash.

So, if we do that then the stack content will become something like this. This T goes out and this T dash F comes to the stack and this remains the input remains unchanged. Now I have got F and id and F and id will tell me to go by F producing id. So, this will be

modified to B dash T dash B dash T dash id input will be this one. So id, id will match so I will be advancing the input advance the input pointer.

So, my new stack will be B dash T dash B dash T dash this will be the new stack and here the input will be dollar. So, T dash dollar, so T dash dollar says go by T dash producing epsilon. So, it says go by T dash producing epsilon. So, T dash will go out now the stack will be B dash T dash B dash and dollar and this will take this is a B dash dollar tells me that go by B dash producing epsilon. So, this goes by B dash producing epsilon, so this B dash goes out. So, I have got B dash T dash and dollar.

Now, again T dash dollar will tell me T dash producing epsilon. So, this says you go by T dash producing epsilon. So, T dash will go out now I will have B dash and dollar and here it says B dash dollar says go by B dash producing epsilon. So, go by B dash producing epsilon, so this B dash goes out. So, stack is now empty. So, this stack is now empty and this has the input pointer is also at dollar; that means, the given string is a valid string ah. So, by using this predictive parsing method, so we can construct the corresponding parse tree.

So, this way given a grammar, so you can first construct the first the first and follow sets and from there you will be able to construct the parsing table and once the parsing table is made, so this parsing process is automated. So, you whether to check whether a given string belongs to the language or not? So, you can have this parsing algorithm. So, which will be run and then you can find out whether it is input is come in to dollar input entire input is consumed and the stack is also empty.

So, if you a come if you can come to that configuration staring with the configuration that stack has the start symbol in it and the entire input is there with the pointer at the beginning of the string. So, if you can come to that configuration then you can be you can tell that the string is accepted by the language. So, next we will be looking into a new topic which is known as bottom up parsing. So, bottom up parsing, so this top down parsing that we looked into. So, this actually try to construct the parsing parse tree starting with the start symbol of the grammar and from there it was trying to go in a top down fashion to derive the final string. So, another approach that we can have is that we can start constructing from the bottom. So, starting with the input, so we try to

constructing from the bottom and then we can ultimately merge onto converge onto the start symbol of the grammar.

Now, the problem with the top down parsing was that so you have to predict like at some point of time looking at the next input symbol. So, if it is L L L grammar, so looking at the current input symbol only you have to tell like which rule to follow that sometimes becomes difficult. And if you are if you try to modify that look ahead if you want to make it L L k then the parser will become very complex.

So, that way bottom up parsing is in some many cases bottom up parsing will be better because it will be able to construct from the bottom so it knows the input string that is there and it tries to construct from there. So, that way many many a time you will find that for certain grammars. So, you will be able to formulate the bottom up parsing algorithms, but not the top down parsing. So, the parsing table you can you can construct for bottom up parsing but not for top down parsing. So, that makes bottom up parsing more powerful than the top down parsing.

(Refer Slide Time: 12:49)

Introduction

- Constructs parse tree for an input string beginning at the leaves (the bottom) and working towards the root (the top)
- Example: $\text{id} * \text{id}$

$E \rightarrow E + T \mid T$
 $T \rightarrow T * F \mid F$
 $F \rightarrow (E) \mid \text{id}$

The slide illustrates the bottom-up parsing process for the input string $\text{id} * \text{id}$. It shows a partial parse tree where the root node E is at the top, and the leaf nodes are id and id . The tree structure is as follows:

```

graph TD
    E[E] --> T1[T]
    E --> F1[F]
    T1 --> T2[T]
    T1 --> F2[F]
    T2 --> T3[T]
    T2 --> F3[F]
    T3 --> id1[id]
    T3 --> id2[id]
    F3 --> id3[id]
    F1 --> id4[id]
  
```

The slide also includes a small video inset of a person in the bottom right corner.

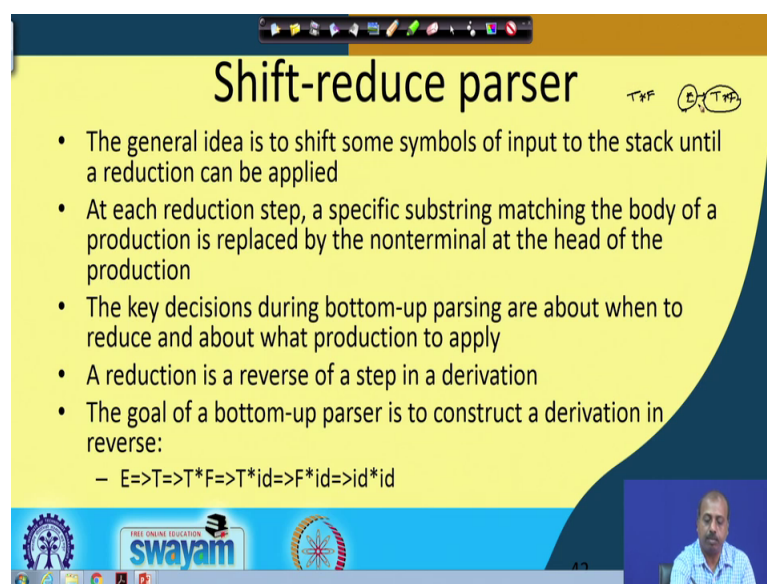
So, to start with this bottom up parsing strategies to the construct parse tree for an input string beginning at the leaves that is at the bottom most level of the parse tree. So, parse tree at the leaf level it has got only the terminal symbols that is the given the input string. So, it will let us start with that bottom level thing and then try to construct the parse tree and finally, reaching the root so which is supposed to be the start symbol of the grammar.

So, if you can do this, so if you can have a if you can show that the entire string can be reduced to the start symbol of the grammar then we say that the string is accepted by the language. So, an example suppose we have got that expression grammar which is given by this E producing E plus T or T T producing T star F or F and F producing within bracket E or id. And we try to see whether id star id is a valid string of this language or not.

So, we will do it in a bottom of fashion. So, first id will be replaced by F ok. So, we get this part then this F can be replaced by T ok. So, this is we have got T into id then this second id is replaced by F and then this T into F. So, that part is replaced by F then T into F is that F F is replaced by T it should be F should be replaced by T here and there is a jump step jump at this point. So, ideally they should not be like this. So, this E should give me T and T should give me F because E cannot give me F. But anyway so this is (Refer Time: 14:45) are make the space less. So, it has been shown like that, but they should be ideally like this E to T to F.

So, you see starting with the input string. So, we can construct the entire parse tree and go to the root the start symbol of the grammar. So, if you can do this thing then we say that we have got a bottom up parsing strategy for the grammar.

(Refer Slide Time: 15:15)



Shift-reduce parser

- The general idea is to shift some symbols of input to the stack until a reduction can be applied
- At each reduction step, a specific substring matching the body of a production is replaced by the nonterminal at the head of the production
- The key decisions during bottom-up parsing are about when to reduce and about what production to apply
- A reduction is a reverse of a step in a derivation
- The goal of a bottom-up parser is to construct a derivation in reverse:
 - $E \Rightarrow T \Rightarrow T * F \Rightarrow T * id \Rightarrow F * id \Rightarrow id * id$

swayam

So, next so they are so this bottom up parser. So, they are also known as shift reduce parsers. Why it is named? Because these parsers they work on two operation mainly on

two operations one is called a shift operation, another is called a reduce operation. So, that is a general idea is to shift some symbols of input to the stack until we can find a situation where reduction can be applied.

So, we go on shifting some input symbols on to the stack and then try to replace that part replace some part of the stack by applying some reduction. And each and in each reduction step a specific substring matching the body of a production is replaced by the non terminal at the left hand side of the production.

For example, if I have got a in the stack if you find that you have got say T star F available in the stack and in the grammar there is a rule like E producing T star F. Then what we do? So we can replace this part of the stack by the symbol E, so that is exactly what is said here. So, we have at each so that is a reduction. So, we reduce some part of the stack by a non terminal.

So, whatever comes on the left hand side of the rule by which we are doing the reduction, so the stack now will now contain that symbol. So, stack will now contain that symbol and the portion on the right side. So, they will be they will be used to replace the portion of the stack. So, naturally the decisions the key decisions during bottom up parsing are about when to reduce? And what which production we should apply for reduction? So, like shifting so I can just get the next input and shift it into that the step, but I have to take a decision that at some point of time I will apply the reduction operation also the reduced operation also.

So, when to you apply this reduce? So, if you are not, but doing it correctly then you will be reducing at arbitrary point. So, that it will not be able to generate the parse tree even if the string is correct. So, that so I have to do it judiciously at which point I do the reduction and also there may be several rules by which we can do a reduction ok.

For some for some point of time so it may so happened that if I take set top most 3 symbols there can be reduction. And there can be two different rules by which we can do that reduction or if I look into top most 5 symbol so I can do a reduction. So, that way I have to take a decision like of which production rule we apply for doing the reduction and a reduction is a reverse of a step of derivation.

So, in a derivation we are going in a top down fashion. So, we are starting with the start symbol and then we are deriving the sentential forms till we are at the input stream and in the reduction process. So, this is just the reverse we are starting at the input stream and converting some parts of it into non terminals and ultimately the entire thing reduces to the start symbol.




So, this is just the reverse of the derivation process and the goal of bottom up parser is to construct a derivation in reverse. Like say actual derivation is say for this id star id the derivation is E to T to T star F to T star id to F to id to id star id. So, but the bottom up parser it will do it in this the direction. So, it will start with id star id then it will find this reduction of id to F, then it will find this reduction of F to T, then it will find this reduction of id to [FL], then it will reduce this T star F by T and then finally, it will reduce this T by e, so that is the thing. So, does it in the reverse direction compare to a top down parsing.

(Refer Slide Time: 19:23)


Handle pruning

- A Handle is a substring that matches the body of a production and whose reduction represents one step along the reverse of a rightmost derivation

Right sentential form	Handle	Reducing production
id*id	id	$F \rightarrow id$
F*id	F	$T \rightarrow F$
T*id	id	$F \rightarrow id$
T*F	T*F	$E \rightarrow T*F$

43



There is a concept called handle pruning. So, what is a handle? So a handle is a substring that matches the body of production and whose reduction represents one step along the reverse of a rightmost derivation. So, for example if I have got say this particular right sentential form id star id, then this id can be a handle because there is a production rule which says F producing id and this right hand side matches with this id. So, this id part is a handle.

Similarly this F producing F F into id then this F is a handle because there is a rule which says T producing F. So, this id is also can be a handle by because there is a rule like F producing id, but the problem is. So, it will give rise to something like F star F and then it may not be giving the actual a rightmost derivation. So, a handle it is with respect to a string.

So, it is not only that it should be the right hand side of some production, but also it should be part of the derivation process, it should be part of the rightmost derivation process arbitrarily we can take. So, if we can identify proper handles then we will be able to identify the proper derivation in the reverse. So, this rule, so this will be using this LR parsers or this bottom of parser; so, they will be using this handle pruning strategy ok.

(Refer Slide Time: 20:55)

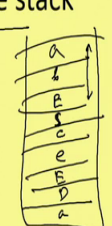
Shift reduce parsing

- A stack is used to hold grammar symbols
- Handle always appear on top of the stack
- Initial configuration:

Stack	Input
\$	w\$

- Acceptance configuration

Stack	Input
\$\$	\$



So, this is the general philosophy of a stack reduce shift reduce parsing policy. So, we will be using a stack to hold grammar symbol. So, grammar symbols means the symbol the terminals and non terminal, so both are grammar symbol. So, we will be having a stake that will hold the grammar symbols and as a result handle will appear on the top of the stack. So, if we have it like this, so if this is the stack. So, it has the grammar symbols in it may I have got say a, b, then some e, s then say c, e, so like that again another e like that.

Now, you see the where this capital uppercase letter. So, they are say non terminals and the lowercase letter, so they are say terminals. So, this may be say some d this may be a

like that. So, they are at, so this stack can contain both terminals and non terminals. Now what can be and handle like say this handle it will always appear on top of the stack. So, I can have a handle which stands over so these three entries so a, b, e. So, if there is a production rules like rule like a, b, e then we will say that this is matching the handle.

(Refer Slide Time: 22:29)


Shift reduce parsing

- A stack is used to hold grammar symbols
- Handle always appear on top of the stack
- Initial configuration:

Stack	Input
\$	w\$

- Acceptance configuration

Stack	Input
\$S	\$



The slide illustrates the initial and acceptance configurations for shift-reduce parsing. The initial configuration shows a stack with a single '\$' symbol and an input string 'w\$'. The acceptance configuration shows a stack with '\$S' and an input string '\$'. A diagram of a stack with 'S' on top and '\$' below, and an input string 'w' with a pointer at the end, is also shown.

So, initially the stack will contain dollar and the input will be the entire input string w and dollar. So, initially the stack has got dollar and this stack has got dollar and this input is the input string w . So, we will assume that there is a dollar character at the end of it, there is a dollar at the end. So, if the whole parsing process in it will start with this particular configuration.

And then it will be an acceptance it will go to an acceptance configuration if at the end you find that the stack has got dollar S in it and if the input is here the input pointer is here. Initially input pointer was here and after this parsing process, so input pointer has come to the end of the string and the stack contains S followed by dollar. So, if this configuration can be reached.

So, starting with this configuration by taking actions corresponding to the input symbols that you have in w ; so, if you can come to this configuration then we say that this is a valid string the given string w is a valid string of the language. So, in this way the shift reduce parser performs the basic actions are it will sometimes we will shift the next input symbol on to the stack, sometimes it will pop out some entries from stack and

apply a reduction operation on them the reduce operation on then. So, these two things will be doing. So, how do when do you take a shift decision and when do you take a reduce decision. So, that will be depicting the design of the parser or the parsing policy.

(Refer Slide Time: 24:09)

Shift reduce parsing (cont.)

- Basic operations:
 - Shift
 - Reduce
 - Accept
 - Error
- Example: $id * id$

Stack	Input	Action
\$	$(id * id)$	shift
\$id	$* id$	reduce by $F \rightarrow id$
\$F	$* id$	reduce by $T \rightarrow F$
\$T	$* id$	shift
\$T*	id	shift
\$T*id	\$	reduce by $F \rightarrow id$
\$T*F	\$	reduce by $T \rightarrow T*F$
\$T	\$	reduce by $E \rightarrow T$
\$E	\$	accept

So, this is a simple example. So, we have got 4 basic operations. So, we have just now we have talked about shift and reduce there are two more operation one is accept and another is error. So, accept is the situation where you have reached a configuration where the stack top contains the start symbol of the grammar, and the input is also input pointer is also pointing to dollar. So, that is the accept configuration.

And in between so if you are in a state where the parser does not know what to do the parsing table or the parsing policy does not tell clearly like what to do in that case that is an error operation. So, error operation so you can have some error message flashed or you can take some error recovery measure, so that the parser can continue parsing reporting the error.

So, let us take the same example the $id * id$. So, the initial configuration will be like this the stack will be having only dollar and this $id * id$ dollar. So, this will be the input. So, right now we do not know how do we take the shift and reduce decision, but suppose we know that we can we can choose we can make this actions we can make the actions shift and reduce and the choice is done in this fashion as it is shown here.

So, suppose we take a decision to do a shift then what will happen this id will come to the stack. Now the stack has got dollar and id with id at the top of the stack and the input will have star id dollar. Then somehow we come to a decision that will be following this reduction rule reduced by a producing id. So, this id will be popped out from the stack and then this left hand side that is F so it will be pushed into the stack.

So, stack now contains dollar F, then looking at a F and star we take a decision that we will be doing a reduction by T producing F. So, this F is taken out from the stack and T is pushed into the stack. Now getting T and T at the top of the stack and star at the next input so, somehow we take a decision that will shift this star into the stack. So, the star is shifted into the stack and then this looking into this star and id this parser takes a decision that I will do a shift operation. So, this id is shifted into the stack.

Now, id and dollar then somehow we takes a decision that I will be following a reduction by F producing id. So, this id will be replaced by F, then this T star then by F and dollar looking at F and dollar. So, it will see it will take a decision that will reduce by T producing T star F. So, ultimately so this way it proceeds. So, ultimately we are at a configuration where we have got this start symbol E on the top of the stack and dollar at the input pointer. So, at that point we accept the input ok. So, that is the corresponding action is accept.

So, if there was an error in the expression they need some points so we will not be able to have any legal activity. So, at that time so we will flash that there is an error at this point. So, this can come to an error as quickly as possible. And in fact, this bottom up parsing on advantages that. So, it can come to these errors quite fast. But of course, how do you take this decision so that is the question ok. So, we will look into that as we proceed through the discussions.

(Refer Slide Time: 27:49)

Handle will appear on top of the stack

Stack Input

\$ α β γ yz\$

\$ α β γ yz\$

\$ α β γ z\$

Stack Input

\$ α γ xyz\$

\$ α Bxy z\$

The slide shows two parse trees and their corresponding stack and input states. In the first tree, the stack contains α , β , and γ , and the input is yz\$. In the second tree, the stack contains α and γ , and the input is xyz\$. The slide illustrates how a handle (Bxy) appears on top of the stack.

Now, handle will appear on top of the stack like say save this y z. So, this is the input; so, this y so this stack has got this thing this alpha, beta, gamma so this is the stack. Now it will find that this gamma. So, gamma will be replaced by B, so this gamma will be replaced by B. So, there so you see that gamma gamma comes at the top of the stack and that has been replaced by B.

Similarly, we have got this in the next step this y is shifted. So, this is B y, so this becomes it become B y and then this B y this beta B y. So, this gives me a handle and this beta B y is replaced by gamma. So, that way it can it can proceed ok. This is just an example how this handles can come at the top of the stack.

(Refer Slide Time: 28:53)

Conflicts during shift reduce parsing

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

Stack: ...if expr then stmt

Input: else ...\$

Production rules:

- $stmt \rightarrow if\ expr\ then\ stmt$
- $stmt \rightarrow if\ expr\ then\ stmt\ else\ stmt$
- $other$

So, there can be conflicts in the shift reduce parsing and there are two types of conflict like one is called shift reduce conflict, another is called reduce reduce conflict. So, shift reduce conflict means looking at the next input symbol and the top of the stack. So, there may be two possible actions. So, we can shift it shift the next input or we can apply some reduce rule we can reduce operation following some production rule and if the grammar is such that we cannot identify a single operation.

So, both the operations appears to be valid the shift and reduce operation then that is called a shift reduce conflict. Another thing is reduce reduce conflict so here the parser does not know which rule to apply for doing the reduction. So, it say it find there some reduction has to be done, but there may be more than one rule which qualifies for the reduction that is the right hand side of more than one rule matches with the handle. Then which rule to follow so that is reduce reduce conflict.

So, typical example is say this one. So, statement producing suppose we have got this particular rule so this is the grammar that we have. Now in the stack if we have this situation if expression then statement and in the input I have got else part. Now what to do? So, if you are if so one possibility is that so it will be lead to if expression then statement else statement type of thing so that is this one. So, in that case I should shift this else into the stack. So, that is a shift operation.

Other possibility is that this. So, this itself maybe the and if then statement. So, if we so we may try to reduce we may try to reduce by this rule statement producing if expression then statement. So, this is a shift reduce conflict on the input symbol else. If the input symbol is else and the stack contain something like this then it say shift it can give rise to a shift reduce conflict.

So, we will see how to take care of this as we proceed through the classes. And this is we will see that there are some of them can be handled or some of them can be resolved by taking help of the precedence of the operators. Some of them can be handled by modifying the grammar a bit and all that thing, but ultimately when you are designing the parser the parser should not have any conflict. So, it is the compiler designer's responsibility to do something so, that the parser does not face this type of conflicts.