

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

Lecture – 20
Parser (Contd.)

So, the recursive descent parsing technique that you are discussing; so, there are some problems. The first problem is that the general recursive descent parsing so, this requires backtracking.

(Refer Slide Time: 00:26)

Recursive descent parsing (cont)

- General recursive descent may require backtracking
- The previous code needs to be modified to allow backtracking
- In general form it cannot choose an appropriate production easily.
- So we need to try all alternatives
- If one fails, the input pointer needs to be reset and another alternative has to be tried
- Recursive descent parsers cannot be used for left-recursive grammars

The slide includes a diagram of a stack with a pointer and a small video inset of the professor in the bottom right corner. The Swayam logo is visible in the bottom left.

So, the point is that as I said that there may be alternatives. So, I suppose I have got 2 alternatives A producing alpha is an alternative and beta is another alternative. So, first I will try with this rule A producing alpha now while doing this so, the input that we have so, input pointer was say. So, this is my input sequence and the pointer was somewhere here, when I was when I started the procedure for A.

Now, while trying out this alternative alpha, this input pointer has advanced to some extent. So, it as gone to this and then suppose, we find that no A producing alpha is not the correct option to try out. So, it does not derive the final string. So, we need to come back and try the next alternative a producing beta, but how do we do this? Because by this time the lexical analyzer so, it has advanced its input pointer already to this point.

So, it has to be taken back to this point before, we start this A producing beta alternative. So, this is difficult because, we have to take back the input pointer to come back to this point. So, that is what we you are telling it here, that the last code that we have seen for the procedure A.

So, that needs to be modified to allow backtracking. In general form, it cannot choose an appropriate production easily because, we may have this left recursion, left factoring and also though it may be a left recursive grammar or they it may require left factoring. So, if those are not done then I cannot make a top down parser strategy, recursive descent parsing strategy using that grammar.

So, we cannot choose an appropriate option very easily whether to try out alpha or beta. So, there is no immediate guidelines. So, even if this left factoring and left recursion elimination has been done still, it may be difficult to make a wise choice between alpha and beta. So, only when alpha fail so, we try out beta. So, exhaustively all the alternatives maybe may have to be tried out. So, you need to try all alternatives; if one fails the input pointer needs to be reset and another alternative has to be tried.

So, these I have just discussed. So, you need to take back the input pointer for the lexical analysis tool to the point from where, we have to started the procedure A. So, this is another issue and the recursive descent parsers they cannot be used for left recursive grammar. So, if the grammar has got left recursion then we cannot use this recursive descent parsing.

(Refer Slide Time: 03:12)

The slide is titled "Example". It contains the following content:

- Production rules:
 $S \rightarrow cAd$
 $A \rightarrow ab \mid a$
- Input: cad
- Two parse trees for S:
 - Tree 1: S expands to c, A, and d. A expands to a and b.
 - Tree 2: S expands to c, A, and d. A expands to a.
- Handwritten annotations on the right side:
 - $S()$
 - { consume c
 - $A()$
 - { consume b
 - $A()$
 - { consume a

The slide also features a Swamyam logo and a small video inset of a man in a pink shirt in the bottom right corner.

So, we will see some solutions to this strategies, this drawbacks in our next slides, but before that we take an example suppose, we have got a set of production rules S producing $c A d$ and A producing $a b$ or a . So, you see that we do not have any left recursion here because, here it starts with a terminal symbol c , here also it starts with a terminal symbol a . So, there is no left recursion. Similarly, there is no left factoring is also required because, because we do not have any non-terminal big coming on the right hand side.

So, it is not that I have got this a then something that you can take out a and like that suppose the input that we have is $c a d$. So, the way it proceeds it first starts with first try out we start with the a star symbol of the grammar and I do not have any option here there is only one rule $c A d$.

So, it expands it by $c A d$, then it tries out the first alternative S producing $a b$. So, if it does this S producing $a b$ then if you see that it has derived the strings $ca bd$, which is not cad . So, it cannot be converted to cad . So, it will fail the parser the this when it tries does this. So, it cannot go to the input string cad . So, it will not match. So, it will when procedure for A is called, it will first call the actually when the procedure for S is called. So, the procedure S is something like this.

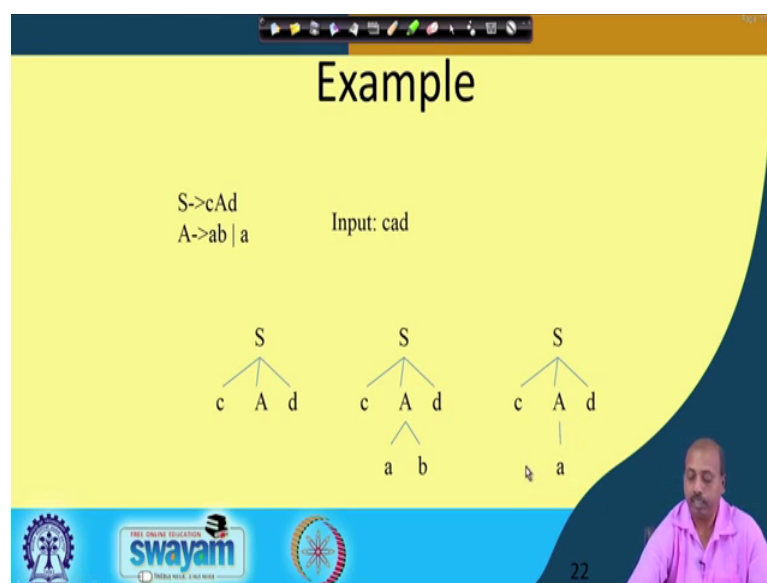
So, consume c . So, it is consume c then it will called procedure A then it will be consume d , it will be something like this and the procedure for A is something like this there are 2

alternatives, first alternative is consume a and then consume b. So, this is one alternative and a other alternative that, we have is consume a this is a second alternative. So, first it will try out the first alternative for a. So, this will be successful because, when it is c a d initially the input pointer is here.

So, procedure for S is call so, it will consume c. So, input pointer will advanced it will come to a now it calls the procedure A so, it comes here consumes a and then pointer is advanced. So, it tries to consume b, but it finds that there is a d there. So, there is a mismatch. So, it understand that this alternative is not a valid for this particular derivation. So, input pointer is taken back by one position from where this procedure a was called and then it will find the other alternative says consume a.

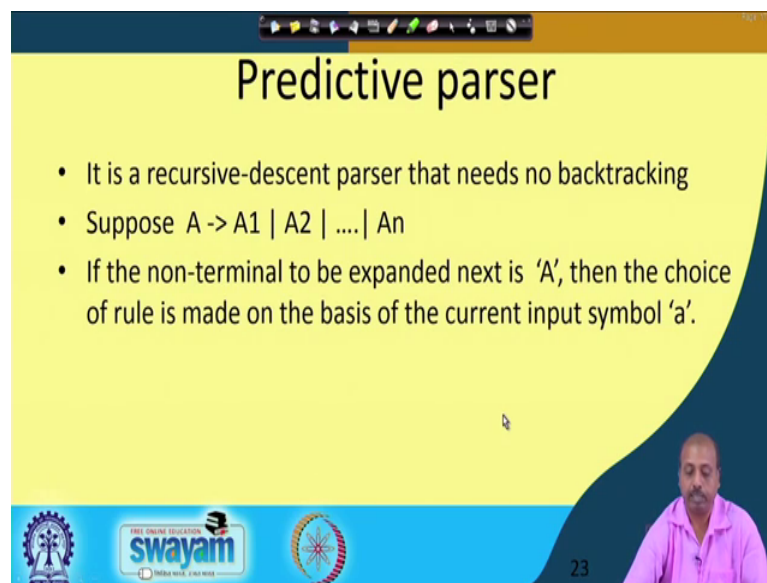
So, input pointer will be advance and then A part is done. So, it will come back to this point and now it will say consume d from the procedure a. So, this will be consumed. So, that will be the third thing and then it. So, it can proceed like that.

(Refer Slide Time: 06:29)



So, it fails at this point. So, it tries out the other alternative A producing a and giving c A d.

(Refer Slide Time: 06:37)



Predictive parser

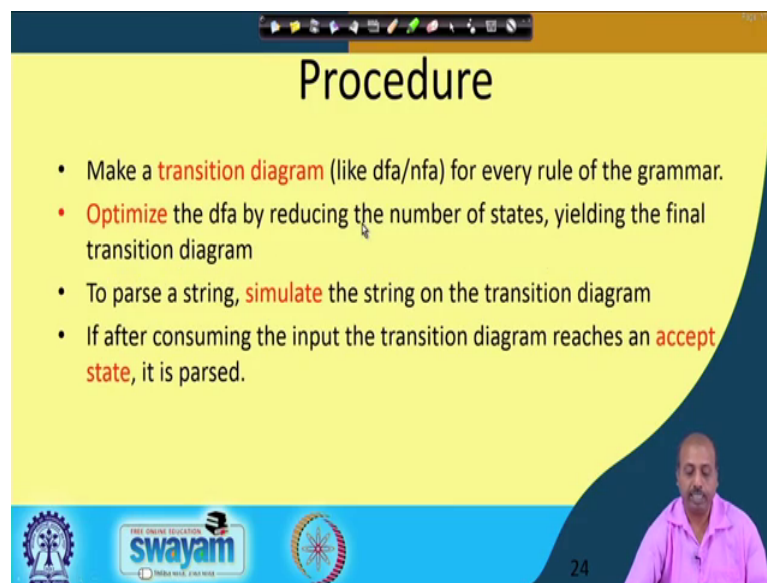
- It is a recursive-descent parser that needs no backtracking
- Suppose $A \rightarrow A_1 \mid A_2 \mid \dots \mid A_n$
- If the non-terminal to be expanded next is 'A', then the choice of rule is made on the basis of the current input symbol 'a'.

23

So, this way this recursive descent parsing works, but sometimes we do not like this type of backtracking because, then we have got the difficulty, because we have to see like how much has to be taken back and all. So, this predictive parsing so, this is a recursive descent parsing that needs no backtracking. So, it will be a recursive descent parsing only, but without any backtracking. Suppose, I have got a rule A producing A_1, A_2, A_n . So, if the non-terminal to be expanded next is A then the choice of the rule is made on the basis of the current input symbol. So, if we have got a number of suppose, these are the alternatives A_1, A_2, A_n . So, these are the alternatives.

So, which alternative to follow so, based on the current input symbols. So, it will take a decision ok. So, if we can take the decision properly then we can make a predictive parser for the grammar. So, if we cannot make a decision for all the productions then predictive parser designed for the particular grammar is not possible.

(Refer Slide Time: 07:45)



Procedure

- Make a **transition diagram** (like dfa/nfa) for every rule of the grammar.
- **Optimize** the dfa by reducing the number of states, yielding the final transition diagram
- To parse a string, **simulate** the string on the transition diagram
- If after consuming the input the transition diagram reaches an **accept state**, it is parsed.

24

So, in many cases it will be possible and in many cases it will not be possible. So, we will see the situations for both. So, procedure for technique for designing this long recursive version or predictive parsing mechanism is to make a transition diagram for every rule of the grammar like dfa or nfa deterministic finite automata, non deterministic finite automata. So, we make a transition diagram and then we try to optimize the transition diagrams by reducing number of states yielding the final set of diagram.

So, we first draw a set of diagrams based on some optimization of the diagram. So, we come to the final stage of diagram, whenever you are trying to parse a string, we simulate the string on the transition diagram. So, as if we are making transitions to the transition diagram using the inputs from the string, if we are we have consumed the input transition after consuming the input, when you are consuming the entire input if the transition diagram reaches an accept state then it is taken as parsed ok.

So, if we so, transition diagram like dfa and nfa it will have an some initial state and a set of final states. So, if it reaches one of the final states after consuming all the inputs then you will say that the parser has accepted the input.

(Refer Slide Time: 09:17)


Example

The grammar is as follows

- $E \rightarrow E + T \mid T$
- $T \rightarrow T * F \mid F$
- $F \rightarrow (E) \mid id$

After removing left-recursion , left-factoring

The rules are :




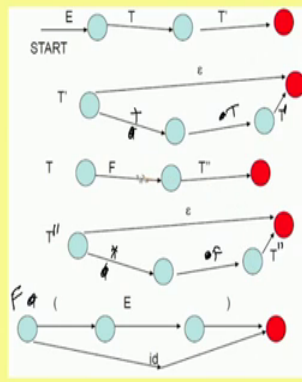
25

So, we will take some example. So, suppose we have again take that expression grammar E producing E plus T or T, T producing T star F for F F producing within bracket E or id then after removing this left recursion and left factoring the rules will be something like this E producing T T dash then T dash producing plus T T dash or epsilon F producing T producing FT double dash and T producing star FT double dash epsilon.

(Refer Slide Time: 09:33)

Rules and their transition diagrams

- $E \rightarrow T T'$
- $T' \rightarrow + T T' \mid \epsilon$
- $T \rightarrow F T''$
- $T'' \rightarrow * F T'' \mid \epsilon$
- $F \rightarrow (E) \mid id$



26

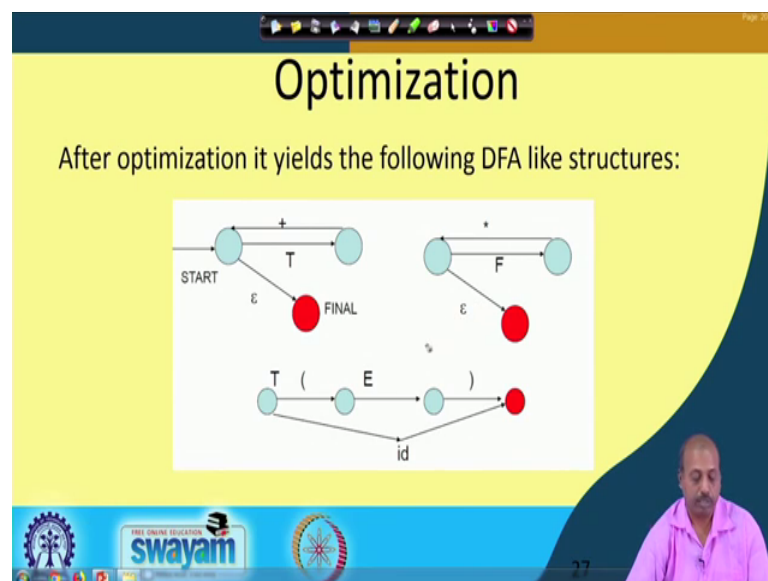
And then T producing within bracket E id now, so this is so, this is the for the first one. So, we have got a diagram like this E this is for E. So, produces it uses T and T dash. So,

on getting these 2 symbols, it will come to a final state similarly T dash. So, this is the red is the red is the final state. So, on getting a epsilon. So, it can come to final state or it can be it can be. So, this is our T. So, this is FT double dash then this is T dash.

So, T dash is actually this is something wrong. So, this should be plus this should be T and this should be T dash and then these T dash, this is FT double dash then, these T T dash producing this should be T double dash. So, this is T double dash producing. So, this should be T double dash producing a star FT double dash or epsilon and then this should be F producing within bracket E or id.

So, this is for F within bracket E or id. So, this is T dash. So, this should be plus this should be T and this is T dash then T producing FT double dash that is all right then this is for T double dash, T double dash producing epsilon and star this should be star FT double dash and F producing within bracket E and id. So, this is our final set of transition diagrams that we can have and then.

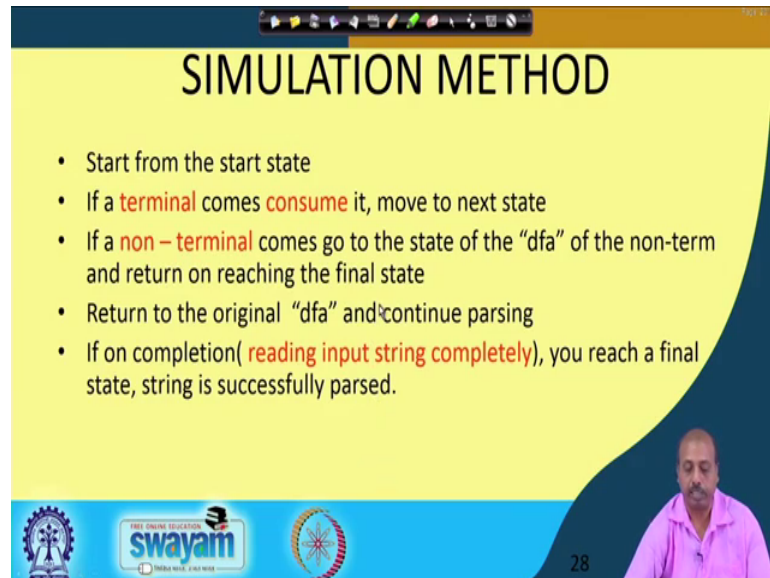
(Refer Slide Time: 12:50)



So, you can do some optimizations like starting with the star symbol. So on getting T, it comes to this state and so, this is the rule for this is the rule for T. So, that T is again coming back. So, you know coming back to the start state. So, if there is no expression after this. So, it can come to the final state or if there is a plus symbol so, it can come to this. Similarly, if for the for this f it can come to on getting epsilon, it can come here, but

if it gets F. So, it can come to this state and getting star, it can come back to this point and again proceed on epsilon to the next state. So, this can be done.

(Refer Slide Time: 13:37)



SIMULATION METHOD

- Start from the start state
- If a **terminal** comes **consume** it, move to next state
- If a **non – terminal** comes go to the state of the “dfa” of the non-term and return on reaching the final state
- Return to the original “dfa” and continue parsing
- If on completion(**reading input string completely**), you reach a final state, string is successfully parsed.

28

So, how do you simulate it? We start with the start symbol of the set of diagram start with the start state and if it is a terminal, we consume it and move to the next state, if it is a non terminal. So, we go to the state of the dfa of the non terminal and return on reaching the final state, when we return the final state of that non-terminal transition diagram, we return to the start state and we return to the original dfa and continue parsing.

So, this way when we are whenever you reach the any final state so, we come back to the original dfa and continue parsing on completion of input when the input has been seen completely. So, if we find that we have reached a final state then the string is successfully parsed. So, otherwise it is not.

(Refer Slide Time: 14:30)

Disadvantage

- It is inherently a recursive parser, so it consumes a lot of memory as the stack grows.
- To remove this recursion, we use LL-parser, which uses a table for lookup.

Handwritten notes:
leftmost derivation → LL-parser
left-to-right scan → LL-parser

So, the problem that we have is that inherently it is a recursive parser. So, it consumes lot of memory as the stack grows. So, we are calling the routines again and again. So, as a result we are doing it by means of transition diagram, but it is taking lot of time it may take lot of time, because of this recursive nature and to remove this recursion, we can use one type of parser, which is known as LL parser, which will use a table for the lookup procedure.

So, this LL so, these 2 L stand for the first L stands for leftmost derivation and this is the this is stands for left to right scan of the input, this will do a left to right scan and it will produce a left most derivation for the parsing.

(Refer Slide Time: 15:41)

First and Follow

- **First(α)** is set of terminals that begins strings derived from α
- If $\alpha \Rightarrow^* \epsilon$ then ϵ is also in **First(α)**
- In predictive parsing when we have $A \rightarrow \alpha \mid \beta$, if **First(α)** and **First(β)** are disjoint sets then we can select appropriate A-production by looking at the next input
- **Follow(A)**, for any nonterminal A, is set of terminals a that can appear immediately after A in some sentential form
 - If we have $S \Rightarrow^* (\alpha A \beta)$ for some α and β then A is in **Follow(A)**
- If A can be the rightmost symbol in some sentential form, then $\$$ is in **Follow(A)**

Handwritten notes on the slide include:

- A diagram showing a derivation: $S \rightarrow \dots \rightarrow \alpha A \beta \rightarrow \dots$ with a box around $\alpha A \beta$.
- A definition: $\text{Follow}(A) = \{a, b, \dots\}$
- A diagram showing a string $\alpha \beta$ with a box around α and a box around β , with labels $a \in \text{First}(\alpha)$ and $a \in \text{First}(\beta)$.
- A diagram showing a string $S \rightarrow \dots A \$$ with a box around A and a box around $\$$.

The slide also features the Swayam logo and a small video feed of a person in the bottom right corner.

So, we will see how it can be design, but to come to that. So, we need to have some definition one is known as first set, another is known as follow set. So, these 2 definitions we have to learn by heart because, many times in our compiler course. So, will come to these 2 sets first and follow and we should be we should be able to compute them very confidently and particularly this follow set computation is slightly tricky. So, we have to be careful there ok. Let us try to understand what does it mean. So, first of alpha is a set of terminals that begins strings derived from alpha, as you know that alpha is a any string of terminals and non terminals.

So, if it is whatever be the string. So, from starting with this alpha, whatever string after whatever strings you can derive ok. So, in that for whichever symbol appears as the first symbol; so, that is the thus that is particular terminal symbol so, will be included in the first set. So, if alpha in 1 or 0 or more derivations give epsilon in epsilon is also in the first of alpha ok. Now, if I have got so, how does it, how is it going to help? It is going to help because, if I have got 2 productions, A producing alpha and A producing beta and at present suppose, I am looking in the current input symbol is A.

So, based on this we will be able to take a decision like which rule to follow. So, we look into the 2 sets first of alpha and first of beta. Now if the symbol A belongs to the set first of alpha and A does not belong to first of beta. So, in that case there is there is no point trying out the A producing beta alternative for this particular string.

So, we should try with A this A producing alpha type of rule. So, this way it will be helping us in predicting like which rule to follow. So, in predictive parsing whenever we have got A producing alpha or beta, if first of alpha and first of beta are disjoint sets then we can select appropriate a production by looking at the next input ok. So, this way it is going to help. Another definition is follow of A, so where A is a non terminal symbol. So, follow of a is a set of terminals A that can appear immediately after a in some sentential form.

So, sentential form means anything that you can derive with the start symbol of the grammar. Suppose, S is the start symbol of the grammar and you are applying some rules and. So, ultimately this rule leads to set of terminals. So, this is the final input string now, all the this intermediary things that you have so they will be called sentential forms because, they are actually derivable from S and will lead finally, to sum a string of the language.

So, any string of any string of symbols terminals and non terminals that can finally, lead to the finally, lead to some input string or involving terminals only and this is derivable from the start symbol of the grammar then that mix of mixed string of terminals and non terminals. So, there will be it will be called a sentential form so, and in that sentential form.

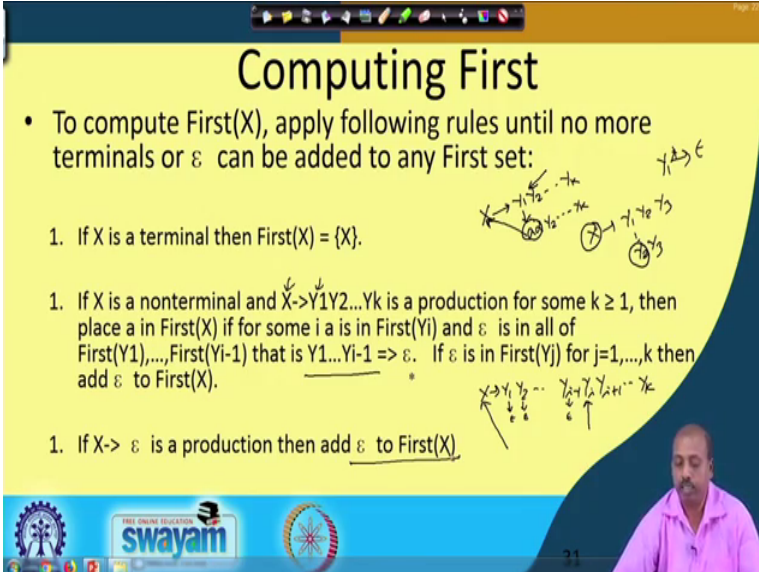
Suppose, I have got a sentential form here, I have got the symbol A and after that suppose this small a character, the terminals small a appears then we say that small a belongs to the follow of A. In another sentential form maybe, this A is followed by may be followed by B. So, B is also in the follow set of A. So, A can be followed by A or B. So, the so, the follow set of A will have these 2 symbols a b, some more may be there, but these 2 must be there.

So, if we have starting with S in 0 or more derivation steps. So, if you come to this type of sentential form alpha A A beta and the for some alpha beta then a is the follow of A. So, this particular small a is in follow of capital A, if a can be the right most symbol in some sentential form then dollar is in the follow of A. So, if you have if you start with S S and see that we can derive something. So, that a is the last symbol of that sentential form then it is assumed that this may be followed by the dollar symbols and dollar matches with the end of strings.

So, whenever we have got a string. So, the end of string is marked by the dollar symbol. So, that is a convention followed by these compiler designers. So, it assume that the last symbol of the input is the end of string symbol which is dollar. So, if this A happens to be the last symbol of any sentential form then this dollar will also be in the follow of A.

So, these are the definitions of first and follow, first is any string that can be derived from alpha, whatever can come in the first whatever terminal symbol can come at the beginning so, that is the first of a alpha. Follow means, you take any derivations from the start symbol of the grammar and all the intermediately derivations in all the intermediately derivations, if this any non terminal a can be followed by another terminal symbol small a then the small a is going to be in the follow of follow set of capital A. So, that is all these first and follower define.

(Refer Slide Time: 22:02)



Computing First

- To compute $\text{First}(X)$, apply following rules until no more terminals or ϵ can be added to any First set:

- If X is a terminal then $\text{First}(X) = \{X\}$.
- If X is a nonterminal and $X \rightarrow Y_1 Y_2 \dots Y_k$ is a production for some $k \geq 1$, then place a in $\text{First}(X)$ if for some i a is in $\text{First}(Y_i)$ and ϵ is in all of $\text{First}(Y_1), \dots, \text{First}(Y_{i-1})$ that is $Y_1 \dots Y_{i-1} \Rightarrow \epsilon$. If ϵ is in $\text{First}(Y_j)$ for $j=1, \dots, k$ then add ϵ to $\text{First}(X)$.
- If $X \rightarrow \epsilon$ is a production then add ϵ to $\text{First}(X)$.

The slide includes handwritten diagrams illustrating the computation of First sets. One diagram shows a nonterminal X deriving $Y_1 Y_2 \dots Y_k$ with arrows indicating the flow of symbols. Another diagram shows a production $X \rightarrow \epsilon$ with an arrow pointing to the ϵ symbol. A third diagram shows a production $X \rightarrow Y_1 Y_2 \dots Y_k$ with arrows pointing to each Y_i and a note that ϵ is in all $\text{First}(Y_i)$.

Now, how is it going to help us? So before going to that so, let us see how can you compute this first set and follow set. For computation of the first set, we apply the rules until no more terminals or epsilon can be added to any of the first set, what is the rule? So, if X is a terminal symbol. So, we are trying to compute the first of X , where X is a grammar symbol, it can be a terminal, it can be a non terminal.

So, if X is a terminal then first of X is definitely the X itself, because nothing more can be derived from a terminal symbol. So, that is there in the first of X , if X is a non terminal symbol and X producing Y_1, Y_2, Y_k is a production for some k greater or equal

1 then we place a in first of X , if for some i a is in first of Y_i and ϵ is in all of first of Y_1 first up to first of Y_{i-1} that is $Y_1 Y_2 \dots Y_{i-1}$, they produce ϵ .

So, let us try to explain this part. So, what it says suppose I have got the X is a non terminal symbol and we have got a rule which says it is Y_1, Y_2, \dots, Y_k . So, anything that you derive from X suppose, this from this Y_1 , I can get a derivation like a then Y_2 etcetera, this Y_1 can be expanded to a then definitely a has is there in the first of X . So, that is what the first part says, if Y_1 whatever is there in the first of Y_1 will be in the first of X ok. So, if we just take i equal to 1 then. So, this whatever is in first of Y_1 will be in first of X other. So, what about the terminals belonging to the first of Y_2 ? So, I will have this thing if Y_1 can give me ϵ .

So, if Y_1 can be reduced to ϵ then. So, I had got this Y_1, Y_2, Y_3 . Now, if this Y_1 can be reduced to ϵ then after sometime, I am getting the configuration like $Y_2 Y_3$ provided Y_1 can be reduced to ϵ by means of some steps ok. So, then this whatever is in first of Y_2 will also be in first of X provided Y_1 can give me ϵ .

So, in general so, if I have got say X producing $Y_1, Y_2, Y_{i-1} Y_i Y_{i+1}$ up to Y_k then whatever is in first of Y_i will come to the first of X provided each one of them can give me ϵ , that is first of all of them first of Y_1, Y_2 up to Y_{i-1} , all of them have got ϵ in them or I can say that is this Y_1, Y_2 up to Y_{i-1} can give me ϵ . So, this is the second rule and it says that if X producing ϵ is a production then we add ϵ also to the first of X . So, that is if there is a production rule X producing ϵ then it will come.

(Refer Slide Time: 25:46)

Example of First and Follow Sets

$E \rightarrow TE'$
 $E' \rightarrow +TE' \mid \epsilon$
 $T \rightarrow FT'$
 $T' \rightarrow *FT' \mid \epsilon$
 $F \rightarrow (E) \mid id$

	First	Follow
F	{(, id}	{+, *,), \$}
T	{(, id}	{+,), \$}
E	{(, id}	{), \$}
E'	{+, ϵ }	{), \$}
T'	{*, ϵ }	{+,), \$}

33

So, let us see how this first can be computed. So, this is say this is a set of gamma rules E producing T E dash E dash producing plus TE dash or epsilon T producing FT dash T dash producing star FT dash or epsilon F producing within bracket E or id for some of the rules. So, the first rule says that whatever is in first of T will be in first of E and if T can give me epsilon then whatever is in first of E dash will also be in first of E. Now, T does not give me epsilon. So, naturally I do not have that liberty.

So, whatever is in first of T will be in first of T. Now what is in first of T? First of T is equal to first of F because, by this rule you see that first of T is equal to first of F and what is first of F? So, this rule will tell that open parenthesis in first of F and i d is also in the first of F. So, first of F has got open parenthesis and id and from this rule it says that, whatever is in first of F is in first of T. So, first of T also has got open parenthesis and id and what about first of E? First of E by this rule it says that, whatever is in first of T is in first of E, first of T is open parenthesis id. So, first of E will also have open parenthesis and id now first of E dash from this rule you see it is plus TE dash.

So, it starts with a terminal symbol. So, it cannot have anything else. So, this plus is in first of E dash and E dash producing epsilon is a rule. So, epsilon is also in the first of E dash. So, E dash so, first of E dash has got has got plus and epsilon and finally, this T dash it can give me star and epsilon because, this can give me star and this is epsilon. So,

star and epsilon can come. So, in this way we can compute the first set for the gamma rules that we have here.