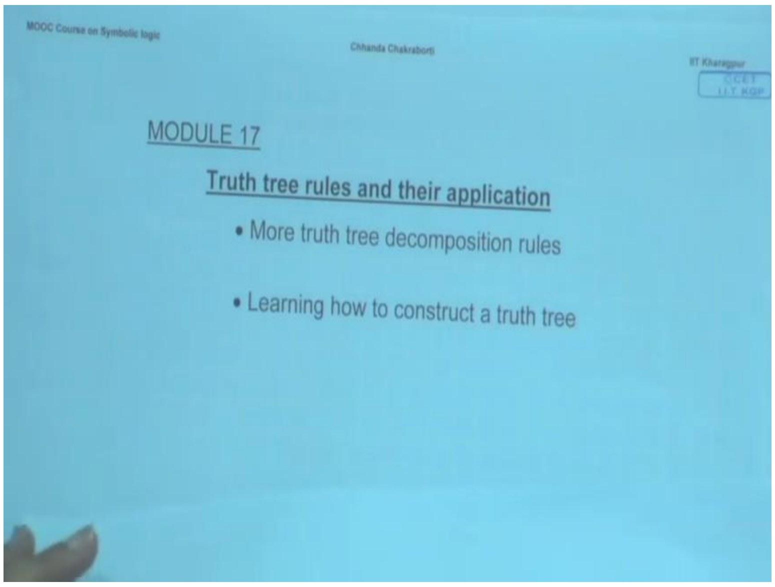
Symbolic Logic

Prof. Chhanda Chakraborti
Department of Humanities and Social Sciences
Indian Institute of Technology, Kharagpur

Lecture - 17
Truth-Tree Rules and their Application
More Truth-Tree Decomposition Rules
Learning how to Construct a Truth-Tree

Hello and welcome to this module! This is our module 17 of symbolic logic course. So, we remember, I hope, that we were learning the truth tree procedure and I just got you started on that the truth tree procedure.

(Refer Slide Time: 00:43)


MOOC Course on Symbolic logic
Chhanda Chakraborti
IIT Kharagpur

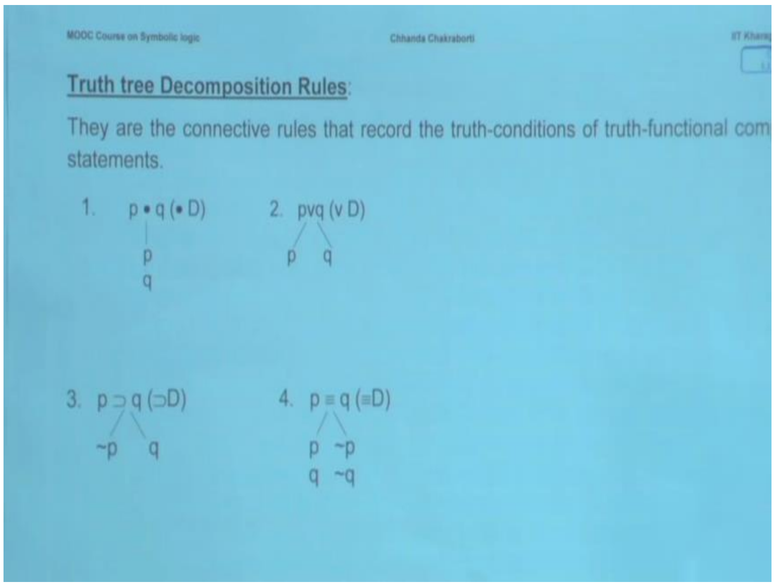
MODULE 17

Truth tree rules and their application

- More truth tree decomposition rules
- Learning how to construct a truth tree

So, we will continue with the procedure a little bit in this module also. This is our module 17 and where are going to learn more about the truth tree rules and their applications. So, I have introduced you only four of these decomposition rules. But as there are more connectives left, so we need to learn a little bit more on this rules. And as we do that, after that comes how to construct a truth tree. So, we'll go together on that and slowly try to construct trees together. So, together means that you can… you can watch me doing it, but then again it is better that if you also have a pen and pencil and a paper ready to, if necessary, to immediately start doing this along with me. Because the results are in front of you, so you can check the result as well as your performance vis-a-vis what is shown. So, we are going to learn about truth tree rules and the decomposition rules.

(Refer Slide Time: 01:46)



MOOC Course on Symbolic logic Chhanda Chakraborti IIT Kharag

Truth tree Decomposition Rules:

They are the connective rules that record the truth-conditions of truth-functional com statements.

1. $p \bullet q$ ($\bullet$D)

   p

   q

2. $p \vee q$ ($\vee$D)

   p q

3. $p \supset q$ ($\supset$D)

   $\sim p$ q

4. $p \equiv q$ ($\equiv$D)

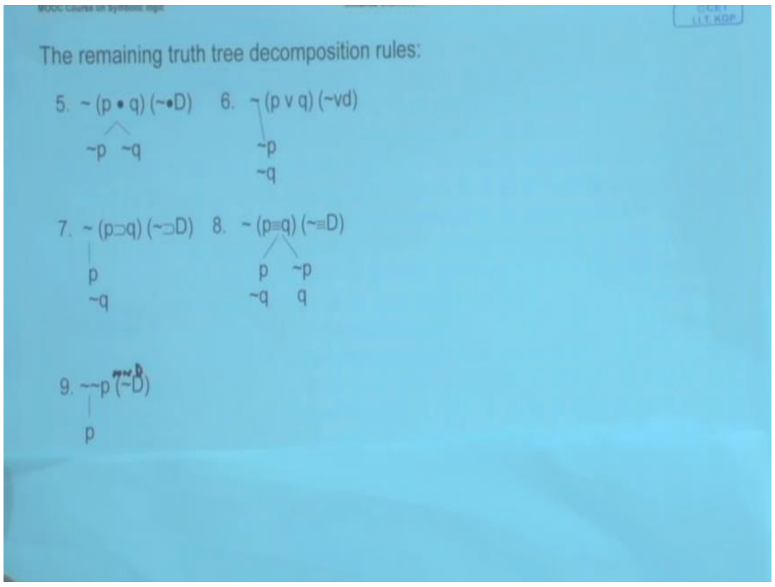
   p $\sim p$

   q $\sim q$

We started this. So, I will just freshen your memory a little bit and say that, you know what we were looking last were truth tree decomposition rules; how to decompose from propositions in a truth tree. And we found out, for example, that these are four rules which tells us how to decompose a dot proposition (•), a wedge proposition (∨), a horseshoe proposition (⊃) and a triple bar proposition (≡). So, the main connectives, as you can see, are all over and each of these, as I have referred to you and I have explained to you, will be referred to by this kind of names : the • D or the ∨ D or the ⊃ D decomposition or the ≡ D decomposition rules. So, try to understand them, instead of memorising them try to understand the rules so that they stick to you better and then learn to apply them. As we go along we'll take examples and will try to show you.

But this is not the entire set of decomposition rules. You need to know it more because there are many more connectives still left. So, let us take a look into those. Here are the remaining truth tree decomposition rules.

(Refer Slide Time: 03:09)



The remaining truth tree decomposition rules:

5. ~ (p • q) (~•D)
~p ~q

6. ~ (p v q) (~vd)
~p
~q

7. ~ (p⊃q) (~⊃D)
p
~q

8. ~ (p≡q) (~≡D)
p ~p
~q q

9. ~~p (~~D)
p

The next two talks about when you have a situation, when you have the negation of a dot. This one talks about when you have a negation of wedge and apparently these rules are also called that. This is negation • D, different from the earlier seen rule of • D. What are we asking? We are asking when is the negation of a '•' dot true? Ask yourself. So, when is negation of p • q true means when is p • q false? Correct? And we know that p • q is false when either one of them is false, one of the conjuncts is false or when both of them are false. So, you are capturing the truth condition as saying this: that either way not-p true or not-q true, right? So, this is your tilde dot (~ •) decomposition rule, where you are saying when is tilde dot true. This is the rule called tilde dot decomposition.

Similarly, this is your tilde wedge (~ ∨) decomposition rule, where you are asking when is ~(p ∨ q) true, right? So, in other case when is (p ∨ q) false? And answer is very clear : When both the disjuncts are false; that's when the wedge is going to be false. In every other condition, it is going to come out as true. Remember from the truth table? . So, that is what we have read. So, this tilde is true when not-p is true, not-q is true. In other words, when p is false and q is false, and that is what is captured as a linear branch for the rule tilde wedge (~ ∨) decomposition. Got me?

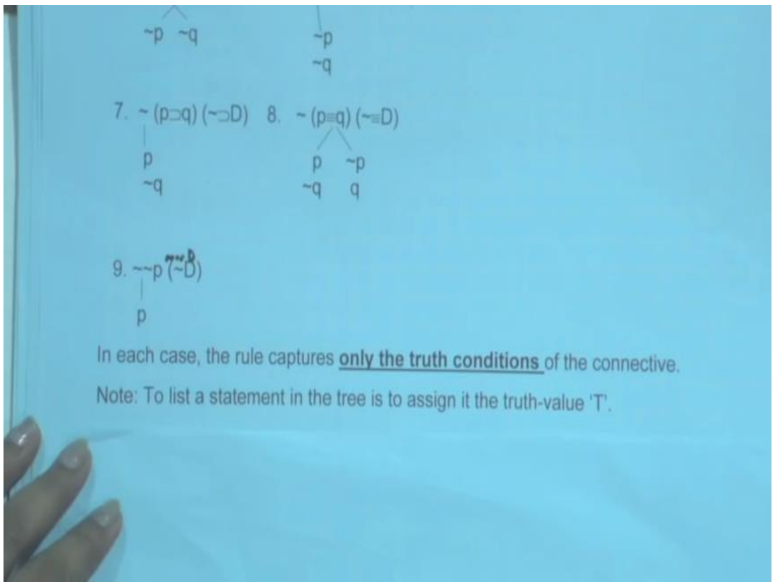
Why do we need to know this? Because you may encounter in your tree also the negation of the dot proposition, or the negation of a wedge as a proposition. So, when you have encountered this, you need a rule two decomposition that and here are the two rules to start you by.

Similarly, these two are negation of horseshoe and negation of triple bar situation. So, what we are asking here is that when is it that tilde p horseshoe q ~ (p ⊃ q) is true? In other words, when is p horseshoe q (p ⊃q) false? Do you remember the horseshoe (⊃) truth-table? If you do, you know that (p ⊃ q) is false when p is true, but q is false. So, that is what we have captured. When is tilde p horseshoe q ~ (p ⊃q) true? When p is true and not-q is true. Not-q is true is means q is false. So, this is your tilde horseshoe (~⊃) decomposition rule.

Here comes that tilde triple bar (~≡) decomposition rule. So, triple bar, when is it true? You have seen it in triple bar decomposition rule. But this is : When is negation of a triple bar true? In other words, when is (p ≡ q) false? So, whenever there is a value mismatch of the equivalent terms, that's when you have the triple bar as false. So, similarly, we have captured it in two sets of conditions. As you see there are two branches, either when p is true, or q is false which means not-q is true. So, that is one set, when, p is true and not-q is true; or when not-p is true which means p is false and q is true. Got me ? So, when p is true, but q is false or when p is false, but q is true. That is what we have captured in here in the negation of triple bar decomposition rule.

And finally, you needed this also: when is negation of a negation true? When you have the proposition. So, it sort of cancels each other. So, when is tilde tilde p ~~p is true? The answer is: When p is true. And this is going to be known as… sorry, that is a mistake. That should be tilde tilde decomposition. Ok? So, this is what will complete our total truth tree decomposition rules. You have 9 of these. Because there are nine possible kinds of connectives that you can encounter the truth tree, and when you do, you have the entire set of rules to apply to it.

(Refer Slide Time: 08:05)


~p ~q

~p
~q

7. ~ (p⊃q) (~⊃D)

p
~q

8. ~ (p≡q) (~≡D)

p ~p
~q q

9. ~~p (~~D)

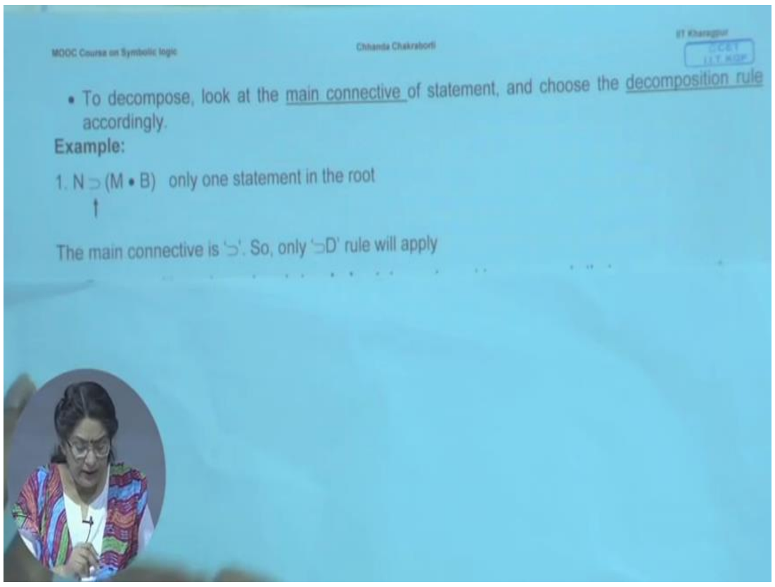
p

In each case, the rule captures **only the truth conditions** of the connective.

Note: To list a statement in the tree is to assign it the truth-value 'T'.

The matter that I will..I will repeat, because though I have said it, but it is important that you take it in is that the rule will capture only the truth conditions, not falsity conditions. So, you have to be creative in a binary logic domain to use the tilde and the literal to express the falsity conditions, but the rule will only show the truth conditions. And once again to remind you listing anything in a tree is to claim that it's truth-value is true. So, with that, we'll go forward with our further things that we can do in the truth tree. Let's try to understand this.

(Refer Slide Time: 08:48)


MOOC Course on Symbolic logic Chhanda Chakraborti IIT Kharagpur

- To decompose, look at the main connective of statement, and choose the decomposition rule accordingly.

Example:

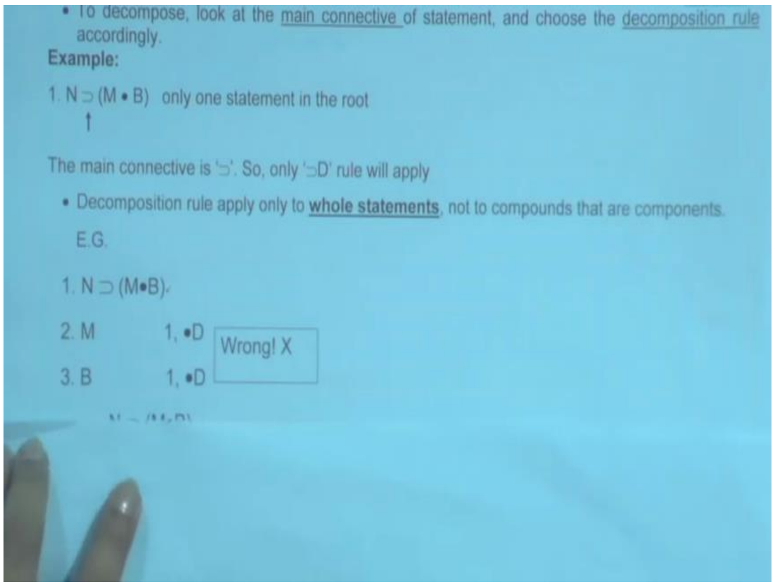
1. N ⊃ (M • B) only one statement in the root
↑

The main connective is '⊃'. So, only '⊃D' rule will apply

So, now you know when we said that how do we decompose? The answer is that you pick a decomposition rule that fits to your main connective. Right? So, if you are dealing with the main connective as the dot, then you all know that you need the dot decomposition rule here. If you are dealing with the wedge proposition, you need the wedge decomposition rule. As simple as that.

Let's take an example. So, here is, for example, the only statement that is given is N ⊃ (M • B). There are two connectives here, but I hope you can immediately make out which one is the main connective. Which one is? The horseshoe (⊃) . Fine? So, in here the only appropriate rule of decomposition is going to be the horseshoe (⊃). Ok? On the other hand, if you do this, so only horseshoe decomposition rule will apply here. You will say that what about this dot (•)? How can I get into that? The answer is : Once you have decomposed the horseshoe, only after that the M • B, the dot inside M • B will become accessible to you; not before that. That is the order. You get at the step, a main connective in order to get into the sub connective.

(Refer Slide Time: 10:09)



- To decompose, look at the main connective of statement, and choose the decomposition rule accordingly.

Example:

1. N ⊃ (M • B)   only one statement in the root

$\uparrow$

The main connective is '⊃'. So, only '⊃D' rule will apply

- Decomposition rule apply only to **whole statements**, not to compounds that are components.

E.G.

1. N ⊃ (M•B)
2. M   1, •D
3. B   1, •D

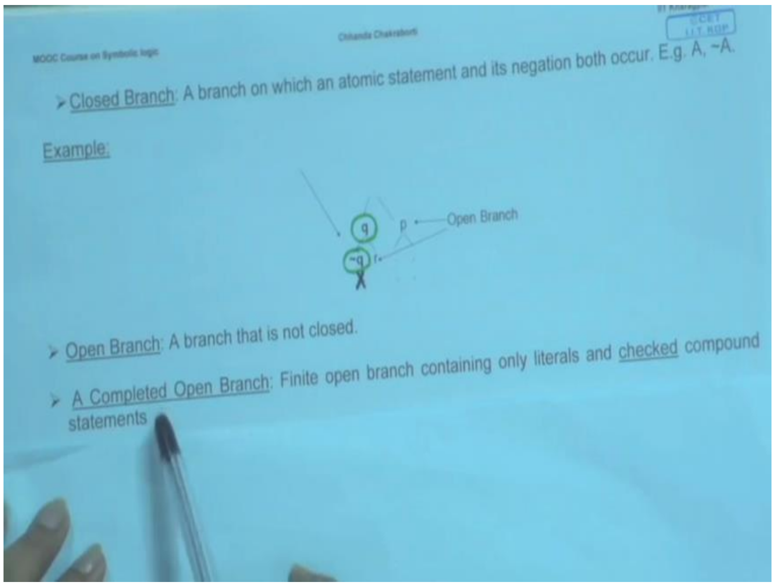
$\boxed{\text{Wrong! X}}$

So, that we have to remember. So, otherwise there will be errors. For example, say, this is what we also meant when we said earlier and remember I saidI will explain it, is that the decomposition rules apply only to whole statement and not to the parts, not to the parts of those compounds. There may be compounds which are components, but you cannot access these until you have broken open the main connective. Ok? Same point, it is the same point, but let’s try to see this.

So, here is for example, a proposition root. Now by mistake if you think that I want to get it here, and you say M and B. What are these? What you have done is to generate branches from here. By what? What you have done is operation on line one. This is one, by dot decomposition. Dot decomposition will give you M and B. So, two separate lines, as you see, two separate numbers and you are justifying it that I got it from one by dot decomposition. Is that correct move? The answer is: Wrong. For everything that I have explained in the last few seconds may be is that you cannot even access this dot (•) until you have broken the horseshoe (⊃) open. So, I will repeat that the only rule that will apply here, the decomposition rule, is the horseshoe decomposition rule. Why? Because horseshoe is the main connective here. Get me? So, in other words, you need to be careful about these kind of procedures that you cannot step ahead to make it convenient for you. You have to do it in a certain order, and the main connective is the right place to start with for the decomposition, and there are enough rules to take care of all kinds of main connectives.

It’s time now to go over the other technical parts of the tree, and this is going to be something to, sort of, get used to. We have seen so far that there will be the roots, then there will be the decomposition process following the decomposition rules that we have just learnt. As a result of decomposition from the root there will be branches, right?

(Refer Slide Time: 12:45)



MOOC Course on Symbolic logic

Chhanda Chakraborti

➢ Closed Branch: A branch on which an atomic statement and its negation both occur. E.g. A, ~A.

Example:

q
p
Open Branch
~q
X

➢ Open Branch: A branch that is not closed.

➢ A Completed Open Branch: Finite open branch containing only literals and checked compound statements

Now, some of these are going to be what is known as *closed branches*. Remember you have heard this term earlier also. Closed branches. What is a closed branch? A closed branch is a branch on which as a result of decomposition you may find a literal and its negation both. I will repeat that. So, a closed branch is a branch on which as a result of your decomposition, you might find a literal and its negation both. So, for example, you might find A and ~A both. Ok? This is when, when you have such a situation that in the same branch you have both a literal and its negation, you know that it is a closed branch. Try to understand this.

Remember what we said about listing. Whenever you are listing a statement in the truth tree, you are claiming it to be true. What is the situation in the closed branch? That you have found a literal and its negation both listed. So, you have found both A and ~A listed. Which means what? You are claiming that in this possibility or in this branch A is true, ~A is also true. Is that even possible in our binary logical world? The answer is : No. We cannot have both A and ~A true at the same time. That's our basic law in this binary logical world. So, which what happens then? If you find such an… rather absurd situation, logically absurd situation in binary propositional logic, then you can declare that this branch is closed. Closed for what? For any further logical operation in it. It means that you are ruling it out, terminating it and you are saying that this is… this is not something that we will continue. This is what closed branches are. We will just show you.

So, for example, that this is a small example, of course, that, you know, suppose you are doing the tree, here are the roots somewhere above, and then as a result of decomposition, please take a look closely here, is that what you have is that on the left hand side somehow you have generated q , and then later on again for decomposition you have generated not-q ~q. Can you see? So, when you have this situation, take a look, this branch is a closed branch, this branch is closed branch. Why is it closed branch ? Because in this both ~q and q have appeared as a result of decomposition from above.

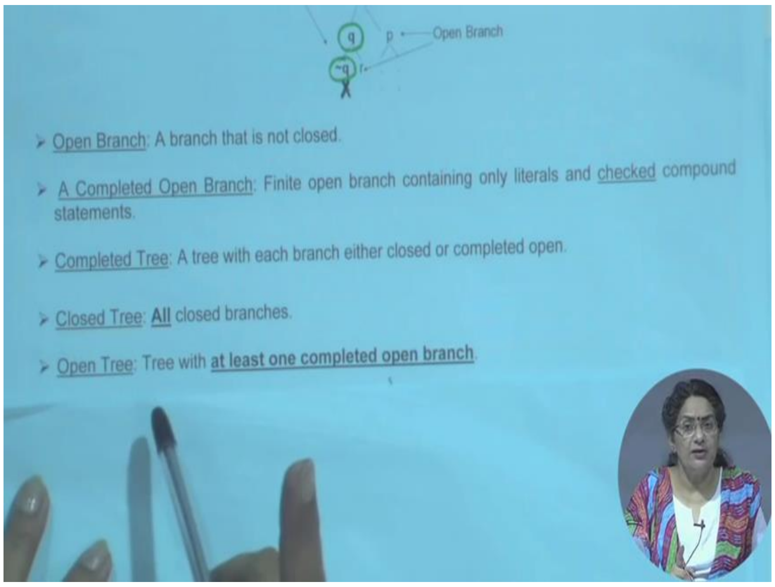
Note and when you do that, when it is closed branch, how do you indicate that it is closed? The visual way to do that is to put this cross, cross under it. So, this cross means that we are not going to do any more operations on this. This has to be put under the closed branch. And this circling is also important. To remind you, also your reader, that this is the reason this branch is closed. So, you encircle the components of the literals which have lead to the closure of the branch. Did you get that? So, follow the branch through and pick out the literals because of which the branch is closed down like so. So, this your closed branch example. But when you are doing this, it may happen and you need to understand this. Can we zoom it further to show this picture relatively clearer?

To show that this… when one branch closes down it is not necessary that every branch will closed down. Right? For example, here you have r. Can you see that? So, this bifurcation that has happened from q on the left hand side has generated not-q, but on the right hand side there is an r. Is that a closed branch? No, it is not. It is not. Because no such situation has happened yet on the branch. Similarly, take a look, at this point you have q coming here on the left hand side and p on the right hand side; and notice, that in here there is no effect of this closed branch. So, that is also possible, that is also possible that you may have some branches closed in the tree, whereas there may be the other branches which are remaining open, open as in not closed. So, that is our next thing to learn.

What is this open branch? An open branch is that branch which is not closed in this kind of sense. Alright? So, the branch is… this is what we'll mean by closed branch, this is what we'll mean by open branch. There is something about *completed open branch*. Ok? So, branches can remain open, and branches can be *open and completed*. There is a difference. What is the difference? A completed open branch is a finite open branch which contains only literals and checked compound statements. Checked means that a tick mark has been given against them. I’ll show you examples. But a completed open branch and let’s take it in. What is that? In which all decompositions have been finished. How do we know that? Because it contains only literals. Remember, the decomposition stops when you have come down to the level of literals. So, if this is an open branch, which contains only literals and whatever compound statements are there, if those are all checked, checked as in ticked. When do you tick it? When decomposition has been done on it. So, in a way, completed open branch is some open branch, in which all decompositions that were to be done on the branch, are over, finished, done; yet it has not closed down. Ok? So, that is what is known as completed open branch. Then you will see that in completed open branch, from a completed branch you actually can get a lot of information out.

So, there is a difference between *open branch* and *a completed open branch*; and a branch may remain open without being completed. That is the first thing to understand. This is why we are making this difference: that there is open branch, branches can remain open without being completed. But this is the situation, where it has remained open even when *all decompositions have been accomplished*. Get it inside your head that there is a major difference here and whatever logical information you need to recover we’ll show you how to… can be obtained from the completed open branch, not necessarily from the open branch. Why? Because the open branch may not remain open, all decomposition steps have not been done. It may close down, it may.. it is… the work is still unfinished on that. So, you have no reliable information to be obtained from that. You need information from this kind of branch: Completed open branch.

(Refer Slide Time: 20:29)



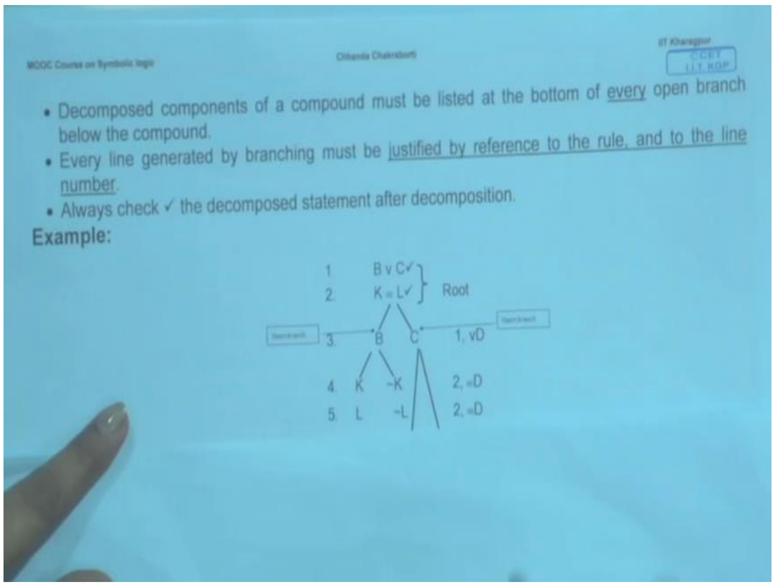
- Open Branch: A branch that is not closed.
- A Completed Open Branch: Finite open branch containing only literals and checked compound statements.
- Completed Tree: A tree with each branch either closed or completed open.
- Closed Tree: All closed branches.
- Open Tree: Tree with at least one completed open branch.

Ok. Then here comes… can we, can we now zoom out? So, then here comes the other thing: *Completed tree*. What is a completed tree? A completed tree is a tree where each branch is either closed or completed open. So, this is a tree where everything that had to be done is finished. How do we know? Either every branch is closed down. So, there is no operation to be done; or whatever operation was there has been finished, which is why you have completed open branch. And this is the kind of tree that we are looking for. You need to finish the work on the tree in order to qualify as a completed tree.

This is a strange tree, you know sometimes it may result and this is called the *closed tree*. A closed tree is one which has all closed branches, all closed branches. So, can that happen? Well, it depends on the root and you see that sometimes they speak out and this may happen in some cases. Not a single branch has remained open. In comparison, what is an open tree? The open tree is the tree with at least one completed open branch. Ok?

So, this is our first of all the details about this. So, let's take a look into them all together. We have just learnt what is a closed branch, what is an open branch, what is a completed open branch, what a completed tree is, what a closed tree is, and what is an open tree. And we will try to take this knowledge further in our construction of a tree. Now we are going to learn how to construct a tree, how to read the tree and so on and so forth, but using these kinds of technical ideas.

(Refer Slide Time: 22:33)



MOOC Course on Symbolic logic

Chhanda Chakraborti

IIT Kharagpur

- Decomposed components of a compound must be listed at the bottom of every open branch below the compound.
- Every line generated by branching must be justified by reference to the rule, and to the line number.
- Always check ✓ the decomposed statement after decomposition.

Example:

1. B v C✓
2. K ≡ L✓
Root
3. B C 1, vD
4. K ~K 2, ≡D
5. L ~L 2, ≡D

Reminding you, and we are soon going to start doing this together, but reminding you that when you are doing the decomposition, the results of that decomposition must be listed at the bottom of every open branch. The compound that you are decomposing, the result of the decomposition must be listed at the bottom of, or *under every open branch that runs through that compound*. It's like parental property, you know. So, if you think about the compound as the parental property then who gets it? All the inheritors, all the legitimate progeny, or the sons and the daughters, should get it. Alright? So, whatever branch is still open *under this compound* should get the result of the decomposition. Remember that. Other branches which are not running through this compound, need not get the result of the decomposition. We'll show you how.

So, you… because branches are going to open in different phases and different places, but whenever you are doing decomposition, remember this rule that result must be listed under every open branch that opens up under that compound that we are decomposing. Ok? Not necessarily has to be replicated for everything.

This we are going to see, but it is important that we mention it right now that whenever you are generating a line, remember the root is given. Root does not require any justification, but every line that you are generating by decomposition that is branching must be justified. Because it is a new line that you are adding. How do you justify? By reference to the rule of decomposition that you have used, and to the line number to which line are you decomposing on. We'll show you examples.

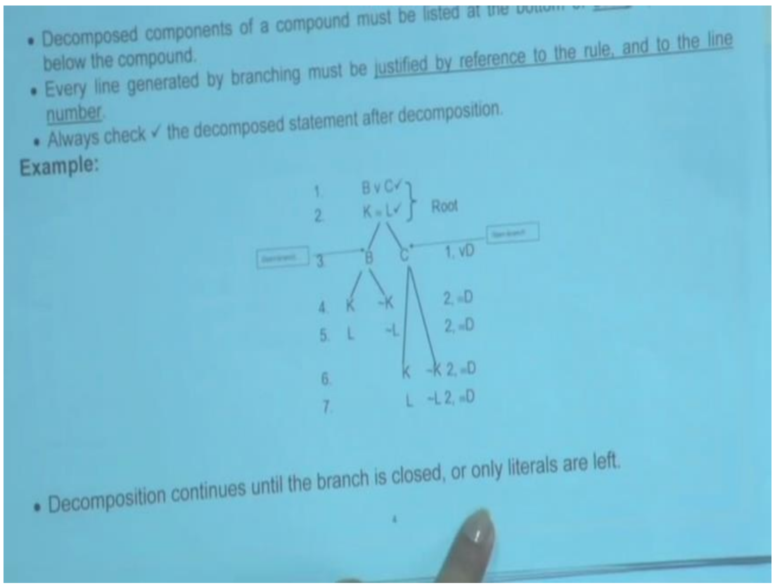
This is called checking. Checking as in ticking. So, whenever you are decomposing a compound statement, remember to put a tick. Why? As a reminder that this has been already decomposed. Don't forget to do that, because you will soon see that's going to be a helpful reminder for you. Because a root may have multiple candidates for decomposition, right? And, as you are growing the tree, as you are doing decomposition in the tree, you may not remember whether there is still something left in the root to be decomposed or not. If you leave something that is still not decomposed, what will happen is that the results may not show the true situation and so on so forth. So, you need to… that is a reminder, that's a reminder, that is the practical tip from my side, that remember to check with a tick against every decomposed statement whenever you have done the decomposition.

Ok. Let's take an example and I think I suggest that you take out your own piece of paper and then to start doing this. Suppose I give you this root. This root which has two statements or propositions. So, $B \vee C$ and $K \equiv L$ right? Now we have to do the tree. This is up to here is root. Then you see, first thing if you, if you go which one to decompose you do not know, so you started with the line number one. If you do that , notice how the branches grow, right from the root, like so. And you write the result of decomposition as B, C and this is the justification. Remember line three was not given, so you need to justify. How do you justify? Line number one, you got it by wedge decomposition.

This is not closed, not closed. These are still open branches. I will show you one more and more, and then please remember that at this point you are not done. Why? Because number two still to be decomposed, right? So, we need to have decomposition done here and we need to do it decomposition here also. These are two branches. So, only decomposing it once will not help. Why? Because both the branches are under this compound. We just learnt that whatever has to be decomposed has to be done under every open branch. So, this is an open branch, this an open branch, so decomposition results should be repeated under each. How do you do that? And that is the point. So, this is my result. 4 and 5 we generated by decomposing this triple bar ($\equiv$), line two triple bar decomposition, line two triple bar decomposition. Do we need to write this twice?

The answer is yes. Every line has to be justified. So, here is K, L and there is ~K , ~L and this is where you put a tick. Similarly when you are doing this decomposition, remember to hold the other side. So, decomposition when one side of the branch is happening, keep the other branch on hold. Why? Because, otherwise it is very confusing for you. Ok? So, procedurally I am saying you need to keep this out.

(Refer Slide Time: 27:54)



- Decomposed components of a compound must be listed at the
  below the compound.
- Every line generated by branching must be justified by reference to the rule, and to the line number.
- Always check ✓ the decomposed statement after decomposition.

Example:

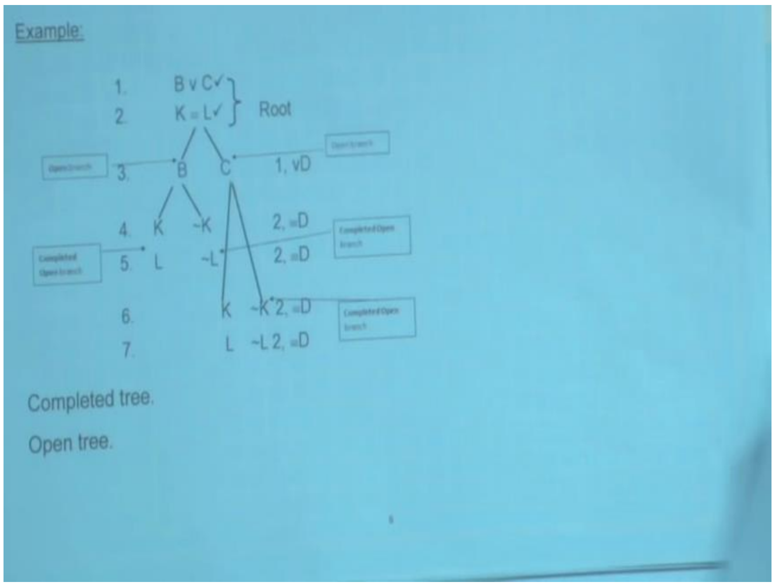
1. B v C✓
2. K ≡ L✓
Root
3. B C 1, vD
4. K ~K 2, ≡D
5. L ~L 2, ≡D
6. K ~K 2, ≡D
7. L ~L 2, ≡D

- Decomposition continues until the branch is closed, or only literals are left.

So, this is when you are doing it on the right hand side. Two, line two still triple bar decomposition, line two triple bar decomposition, right ? And here is the result. Ok?

So are we done with the decomposition? Take a look. You have reduced it all into the literals level and all decompositions have been done; only literals are left.

(Refer Slide Time: 28:18)



Completed tree.

Open tree.

So, let’s take a look into this and this would be our last slide for this module: That this is the situation with this tree. You did the tree and you found that this is an open branch, this is an open branch. So, the results have to be entered under every open branch. Fine, but is it completed at this stage? No. When is it completed at this stage? That is a completed open branch, everything, all decompositions are finished. That is a completed open branch. This is a completed open branch. Which makes this a completed tree. There is at least one completed open branch and what this makes it also an open tree. Get me? So, this is where we are going to close it for this module, and go though it a little bit because we have more exercises in the next module.

Thank you very much.