8 Truth Trees for Sentence Logic

Fundamentals

8–1. PROVING VALIDITY WITH TRUTH TREES

You know, from the exercises of chapter 4, that you can use truth tables to check the validity of any argument of sentence logic. But such truth table checks for validity are extremely tedious. If you read chapters 5, 6, and 7, you also know how to establish validity by using derivations. Many logicians like this natural deduction technique because (for those with a taste for logic) derivations are challenging and fun, and because derivations express and make precise informal modes of deductive reasoning.

In this and the next chapter I will present a third means for determining the validity of sentence logic arguments—the truth tree method. This method is more nearly mechanical than is natural deduction. This fact makes truth trees less fun, because they provide less of a challenge, but also less aggravating, because they are easier to do. Truth trees also have the advantage of making the content of sentence logic sentences clear, in a way which helps in proving general facts about systems of logic, as you will see if you study part II of Volume II.

As a basis for the truth tree method we need to remember two fundamental facts from sections 4–1 and 4–2. We know that an argument is valid if and only if every possible case which makes all its premises true makes its conclusion true. And we know that this comes to the same thing as an argument having no counterexamples, that is, no cases which make the premises true and the conclusion false.

The truth tree method proceeds by looking for counterexamples in·an organized way. The method has been cleverly designed so that it is guaranteed to turn up at least one counterexample to an argument if there are any counterexamples. If the method finds a counterexample, we know the argument is invalid. If the method runs to completion without turning up a counterexample, we know there are no counterexamples, so we know that the argument is valid. Finally, the method considers whole blocks of possible cases together, so in most cases it works much more efficiently than truth tables.

We will introduce the truth tree method with a specific example:

$$\begin{array}{r} \text{AvB} \\ \sim\text{BvC} \\ \hline \text{AvC} \end{array}$$

We are trying to find a counterexample to this argument. That is, we are looking for an assignment of truth values to sentence letters which makes the premises true and the conclusion false. Now, if we try to make some sentences true and another sentence false, things are going to get very confusing. It would be much more straightforward if we could follow a procedure which tried to make all of the sentences considered true.

Can we do that and still be looking for a counterexample to our argument? Yes—if we replace the conclusion of the argument with its negation. So we begin the truth tree method for looking for a counterexample by listing the premises of the argument and the **negation** of the conclusion:

1	AvB	*P* (Premise)
2	~BvC	*P* (Premise)
3	~(AvC)	~C (Negation of the Conclusion)

Our method now proceeds by trying to make lines 1, 2, and 3 true. "Make true" just means finding an assignment of truth values to sentence letters for which the sentences are true. If we succeed, we will have a case in which the argument's premises, lines 1 and 2, are true. And this same case will be one in which the conclusion is false, because the negation of the conclusion, line 3, will be true. So if we can make lines 1, 2, and 3 true, we will have our counterexample.

·Note that I have numbered the lines, and written some information off to the right of each line which tells you why the line got put there. You should always number and annotate your lines in this way so that we can talk about them easily.

Now to work. Let's begin by making line 1 true. We see that there are two alternative ways of making 'AvB' true. First, we can make it true just by making 'A' true. If we make 'A' true, we have made 'AvB' true, whatever eventually happens to the truth value of 'B'. But the same is true of 'B'. If we make 'B' true, 'AvB' will be true whatever eventually happens to the truth value of 'A'. So making 'A' true is a first and making 'B' true is a second, alternative way of making 'AvB' true. We need the truth tree method to be guaranteed to find a way of making all initial lines true if there is a way. So we must be sure to write down **all** ways in which a line might be true, and if there are alternative ways, we must list them independently.

We mark this fact by extending the tree as follows:

√1	AvB	*P*
2	~BvC	*P*
3	~(AvC)	~C

$$\begin{tikzcd} & \sim(\text{AvC}) \arrow[dl, no head] \arrow[dr, no head] & \\ 4 \quad \text{A} & & \text{B} \quad 1, \text{v} \end{tikzcd}$$

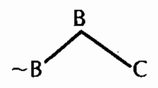
We have split the original path into two paths or branches. Each branch will represent one way of making true all the sentences which appear along it. The left branch will make line 1 true by making 'A' true. The right branch will make line 1 true by making 'B' true. Since the paths have branched, they represent alternative ways of making line 1 true. What happens along one path will have no effect on what happens along the other path below the point at which they branch.

I have added some notation to the tree. The '1, v' at the right of line 4 means that I got line 4 from line 1 by working on a disjunction. I have checked line 1 to remind myself that I have now done everything which needs doing to make it true. I won't have to work on that line anymore.

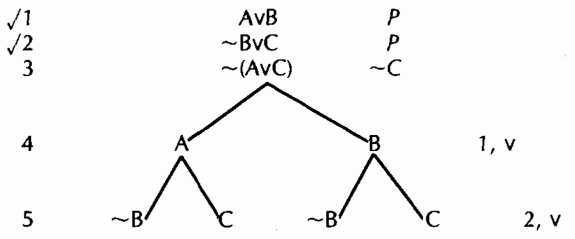
Now let's work on line 2. I have to make it true, and I will do this exactly as I did for line 1. I will make each of the two disjuncts true; that is, I will make '~B' true and I will independently make 'C' true along a separate branch. But I have to "add on" the ways of making line 2 true to each of the ways of making line 1 true. Remember, we are looking for some way of making **all** the original lines true **together.** So I must first write each of the alternative ways of making line 2 true as an extension of the first (left branch) way of making line 1 true. That is, I must extend the left branch with two new branches each of which represents one of the two ways of making line 2 true. So the left branch will itself split and look like this:

$$\begin{tikzcd} & \text{A} \arrow[dl, no head] \arrow[dr, no head] & \\ \sim\text{B} & & \text{C} \end{tikzcd}$$

The same reasoning obviously applies to the right branch. I must add on both of the ways of making line 2 true to the second (right branch) way of making line 1 true. We represent this by splitting the right branch so it looks like this:



Putting these pieces together, after working on line 2, the tree looks like this:



Each branch represents one of the ways of making line 1 true combined with one of the ways of making line 2 true. The leftmost, or first, branch makes line 1 true by making 'A' true and makes line 2 true by making '~B' true. The second branch makes line 1 true by making 'A' true and makes line 2 true by making 'C' true. (Do you see how we combine the alternative ways of making the first two lines true?)

The third branch makes line 1 true by making 'B' true and makes line 2 true by making '~B' true. **Whoops!** Surely something has gone wrong! Surely we can't make line 1 true by making 'B' true and **at the same time** make line 2 true by making '~B' true. If we were to make '~B' true, this would be to make 'B' false, and we just said that along the third branch 'B' would be made true to make line 1 true. We **can't** make 'B' both true and false. What is going on?

What has happened is that the third branch represents an inconsistent way of making lines 1 and 2 true. Focus on the fact that each branch represents one of the four ways of trying to make lines 1 and 2 true together. It turns out that the third of these ways won't work. One way of making line 1 true is by making 'B' true. One way of making line 2 true is by making '~B' true. But we can't combine these ways of making the two lines true into one way of making the lines true at the same time, because doing this would require making 'B' both true and false. We mark this fact by writing an '×' at the bottom of the branch on which both 'B' and '~B' appear. The '×' tells us that we cannot consistently

make true all of the sentences which appear along the branch. We say that the branch is *Closed.* We never have to work on a closed branch again:

> We say that a branch is *Closed* when a sentence, **X,** and its negation, ~**X,** both appear on the branch. We mark a branch as closed by writing an '×' at the bottom of the branch. Do not write anything further on a branch once it has been marked as closed.

The sentence **X** may be an atomic sentence, such as 'A', or a compound sentence, such as 'Bv~(C&~A)'. Also note that the sentence and its negation which cause a branch to close must both appear as entire sentences at points along a branch. If one or both appear as part of some larger compounds, that does not count. To illustrate this point, look at the tree drawn up to line 4, as presented on page 115. On the right-hand branch you see 'B' as the entire sentence at the end of the branch, and '~B' as part of a compound on line 2. The branch does not close at this point because there is no conflict between line 2 and the right branch at line 4. It is perfectly possible for 'B' and '~BvC' both to be true.

It is the fact that branches can close which gives the truth tree method its simplifying power. Here is how simplification occurs. We still have to make line 3 true, and we have to combine the ways of making line 3 true with the ways of making lines 1 and 2 true. But we can forget about one of these ways of (trying to) make lines 1 and 2 true because it turned out to be inconsistent. This corresponds to ruling out certain lines of the truth table before we have completely worked them out. Because a truth tree avoids working out what corresponds to some of the lines of the corresponding truth table, truth trees often save work.

Note how I annotated the tree in the last step: I put '2, v' to the right of line 5 to indicate that I got line 5 from line 2 by working on a disjunction. And I checked line 2 to remind myself that I have done everything needed to make line 2 true. I won't need to worry about line 2 anymore.

Let's complete the tree by working on line 3. What has to be done to make line 3 true? Line 3 is the negation of a disjunction. The negation of a disjunction is true just in case the disjunction itself is false. So, to make '~(AvC)' true, I have to make 'AvC' false. How do I make 'AvC' false? According to the truth table definition of 'v', I can do this **only** by making **both** 'A' false and 'C' false. So the result of making line 3 true will not be a branch of two alternative ways of making a sentence true. There is only one way to make line 3 true: a stack which first makes 'A' false and then makes 'C' false. But to keep things clearly organized I only write down true sentences. So what I have to write is a stack which makes '~A' true and makes '~C' true.:

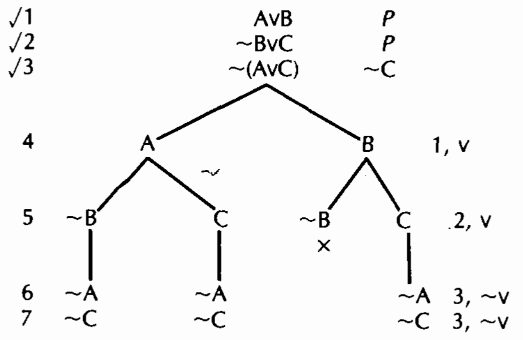
~A
~C

By making '~A' true followed by '~C' true, I have made 'A' and 'C' both false, which makes 'AvC' false, in the only way that this can be done. Finally, making 'AvC' false makes '~(AvC)' true, which was what we wanted to accomplish.

Where do I write this stack? I must combine the results of making line 3 true with all the ways I already have for making lines 1 and 2 true. At first thought, this would seem to indicate that I should write my '~A', '~C' stack at the bottom of every branch. This is almost right. But I won't have to write the results of working on line 3 at the bottom of the closed (third) branch. Remember that the closed branch represents a way of trying to make previous sentences true that can't work because it is inconsistent. So I only have to write my stack at the bottom of every *Open Branch*, that is, every branch which does not have an '×' at its bottom:

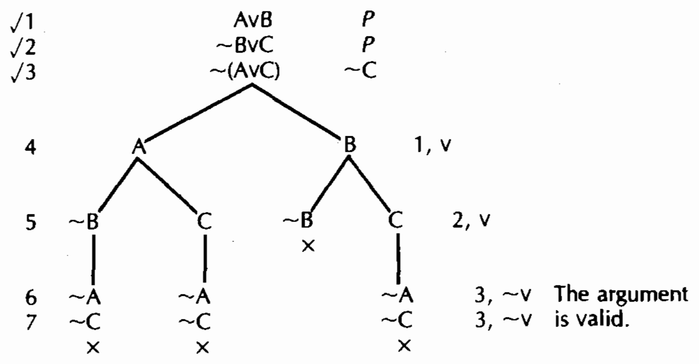
A branch which is not closed is *Open*.

Adding the stack to the open branches gives a tree that looks like this:



The '3, ~v' means that I got the lines from line 3 by working on a negated disjunction. I also checked line 3.

We have one more thing to do to this tree. Note that now the first, second, and fourth branches all have the problem that the third branch had. To make all the sentences along the first branch true, I would have to make 'A' true and '~A' true. That is, I would have to make 'A' both true and false. I can't do that, so I put an '×' at the bottom of the first branch. I do exactly the same to the second branch for exactly the same reason. The fourth branch has both 'C' and '~C', so I must close it with an '×' also. The final tree looks like this:



Here is the final result: All branches have closed. That means that every possible way of trying to make all the initial sentences true results in a conflict. Each way of trying to assign truth values to sentence letters requires that some sentence be made both true and false, and the rules of the game do not permit that. We agreed in chapter 1 that in a given problem a sentence letter would be true or false, but not both. Since there is no way of making all initial sentences true, there is no way of making the argument's premises true and its conclusion false. That is, there is no counterexample to the argument. So the argument is valid.

8–2. THE GENERAL STRATEGY FOR THE RULES

The example we have just completed contains all the ideas of the truth tree method. But explaining two more points will help in understanding the rest of the rules.

First, why did I write a **stack** when I worked on line 3 and a two-legged **branch** when I worked on lines 1 and 2? Here is the guiding idea: If there are two **alternative** ways of making a sentence true I must list the two ways **separately.** Only in this way will the method nose out all the possible **different** combinations of ways of making the original sentences true. I can make line 1 true by making 'A' true and, separately, by making 'B' true. I just list these two alternative ways separately, that is, on separate branches. This will enable me to combine the alternatives separately with all possible combinations of ways of making the other lines true.

But when I tried to make line 3 true there was only one way to do it. I can make '~(AvC)' true **only** by making **both** '~A' and '~C' true. Because there is only one way of making line 3 true, line 3 does not generate two branches. It generates only one thing: the extension of all existing open branches with the stack composed of '~A' followed by '~C'.

I have just explained how I decide whether to write a branch or a stack when working on a sentence. (I'll call the sentence we are working on the "target" sentence.) But how do I decide what new sentence to write on the resulting branches or the resulting stack? Since each path represents a way of making all the sentences along it true, I should put enough sentences to ensure the truth of the target sentence along each branched or stacked path. But I also want to be sure that I don't miss any of the possible ways of making the target sentence true.

It turns out that I can achieve this objective most efficiently by writing the **least** amount on a branch which gives one way of making the target sentence true. I must then write, along separate branches, all the **different** ways in which I can thus make my target sentence true with as few components as possible. I will express this idea by saying that, when working on a sentence, I must write, along separate branches, each **minimally sufficient** sentence or stack of sentences which ensures the truth of my target sentence. This will make sure that no avoidable inconsistency will arise. And in this way the method will be sure to find a way of making all the initial sentences true if there is a way.

In sum, in working on a target sentence, I do the following. I figure out all the **minimally sufficient** ways of making my target sentence true. If there is only one such way, I write it at the bottom of every open path on which the target sentence appears. If there is more than one, I write each separate minimally sufficient way on a separate branch at the bottom of every open path on which the target sentence appears.

This reasoning works to explain all the further rules.

EXERCISES

8–1. Use the truth tree method to show that the following arguments are valid. Show your trees, following the models in the text, complete with annotations to the right indicating at each stage which rule you have used and what line you have worked on.

a) D
J
———
DvJ

b) ~(Mv~N)
———
N

c) ~(FvP)
———
~Fv~P

d) HvI
~IvJ
~JvK
———
HvK

8–2. If you ran into a conjunction on a tree, what would you do? If you don't see right away, try to figure this out on the basis of the description in this section of how the rules work and the two rules you have seen. If you succeed in stating the rule for a conjunction, try to state the rules for negated conjunctions, conditionals, and negated conditionals. If you succeed again, try stating the rules for biconditionals and negated biconditionals, which are a little harder.

8–3. PROVING INVALIDITY WITH TRUTH TREES

Let us now look at an example of an invalid argument:

AvB
———
A&B

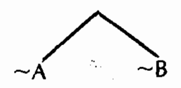
Keep in mind that this argument is invalid just in case there is a counterexample to it, and that the truth tree method works by searching for such counterexamples.

We begin by listing the premise and the negation of the conclusion.

1	AvB	*P*
2	~(A&B)	*~C*

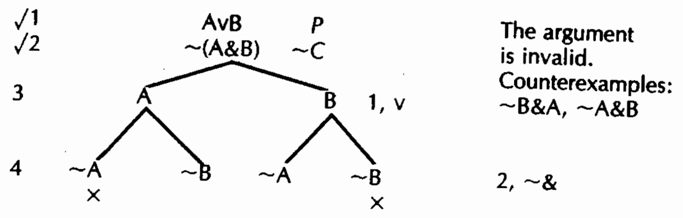
An assignment of truth values which makes both of these sentences true will constitute a counterexample, demonstrating the invalidity of the original argument.

We already know what to do with line 1. But when we get to line 2 we will have a new kind of sentence to work on, a negated conjunction. To make the negated conjunction '~(A&B)' true, I must make the conjunction '(A&B)' false. Making 'A' false is minimally sufficient for making '(A&B)' false, and so for making '~(A&B)' true. So for line 2 I must produce one branch which makes 'A' false, which I do by writing a branch which makes '~A' true. Making 'B' false is likewise minimally sufficient for making 'A&B' false. So for line 2 I also need a second, separate branch with '~B' on it. Altogether, the result of working on line 2 will look like this:



I will write this at the bottom of every open path on which line 2 appears. Note that this rule is very different from the rule for the negated disjunction '~(AvB)'. In working on the negated disjunction, '~(AvB)', I had to write a stack of the negated disjuncts ('~A' followed by '~B') at the bottom of every open branch. This is because only the stack, '~A' followed by '~B', is minimally sufficient for making the negated disjunction true. In working on the negated **conjunction,** '~(A&B)', I must write a branch with '~A' on one leg and '~B' on the other leg. This is because each of the negated conjuncts ('~A' or '~B') is by itself minimally sufficient for making the negated conjunction true.

We now know everything we need to test our new argument for validity:



I first worked on line 1, producing a branch with 'A' on one leg and 'B' on the other, representing the two minimally sufficient ways of making line 1 true. I checked line 1 to remind myself that I have done everything needed to make it true. I then worked on line 2. At the bottom of each open path I put a new branch, one leg representing one minimally sufficient way of making line 2 true, and the second leg representing the second minimally sufficient way of making line 2 true. The annotation on the right of line 4 indicates that I got line 4 by applying my rule for a negated conjunction to line 2. And I checked line 2 to remind myself that I have done all that is needed to make it true. Next I inspected each path to see if it contained both a sentence and the negation of the same sentence. The first path has both 'A' and '~A'. The fourth path has both 'B' and '~B'. So I marked both these paths closed. At this point I observed that there are no unchecked compound sentences which I could make true by making some simpler sentences true.

How does this tree show the argument to be invalid? You see that this completed tree has two open paths. What do they mean? Look, for example, at the first open path. Reading up from the bottom, suppose we make '~B' true and make 'A' true. This constitutes an assignment of truth values to sentence letters ('B' false and 'A' true) which will make the original sentences 1 and 2 true. This is because we made '~B' true as one way of making line 2 true. And we made 'A' true as one way of making line 1 true. So by assigning truth values f to 'B' and t to 'A', we make all sentences along the first open path true.

This sort of thinking works generally for open paths. Keep in mind how we constructed the paths. Each time we found a sentence with an '&' or a 'v' in it, we wrote a shorter sentence farther down the path. We did this in a way which guaranteed that if we made the shorter sentence true, the longer sentence from which it came would be true also. So if we start at the bottom of a path and work up, making each sentence letter and each negated sentence letter true, that will be enough to make all sentences along the path true. Any sentence longer than a sentence letter or

negated sentence letter will be made true by one or more sentences farther down the path.

Even if we start with very complicated sentences at the top, each sentence is made true by shorter sentences below, and the shorter sentences are made true by still shorter sentences farther down, until we finally get to the shortest sentences possible, sentence letters and negated sentence letters. Then we can work backward. Making the sentence letters and negated sentence letters along the path true will, step by step, also make true all longer sentences higher up on the path.

We have seen that the assignment of f to 'B' and t to 'A' makes all sentences along the first open path true. In particular, it makes the original first two sentences true. These sentences were our argument's premise and the negation of our argument's conclusion. So making the initial two sentences true makes our argument's premise true and its conclusion false. In short, 'B' false and 'A' true constitutes a counterexample to our argument. Thus the argument is invalid. One counterexample is enough to show an argument to be invalid, but you should note that the second open path gives us a second counterexample. Reasoning in exactly the same way as we did for the first open path, we see that making '~A' true and 'B' true makes every sentence along the second open path true. So making 'A' false and 'B' true constitutes a second counterexample to the argument.

Our last step is to record all this information next to the completed tree. We write 'invalid' next to the tree and write the counterexamples which show the argument to be invalid. We can do this most easily with the sentences of sentence logic which describe the counterexamples. The sentence '~B&A' describes the counterexample given by the first open path and the sentence '~A&B' describes the counterexample given by the second open path.

A last detail will conclude this section: The order of the cases in the description of the counterexample obviously does not matter. I described the first counterexample with the sentence '~B&A' because I read the counterexample off the path by reading from the bottom up. As a matter of practice, I recommend that you do the same. But in principle the equivalent description 'A&~B' describes the same counterexample just as well.

EXERCISES

8–3. Use the truth tree method to show that the following arguments are invalid. Show your trees, being careful to show which branches are closed. In each problem give any counterexamples which show the argument being tested to be invalid.

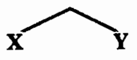
a) F / F&K　b) ~(~S&T) / S　c) KvH, ~K / H&D　d) ~(I&P), PvF / IvF

8–4. THE COMPLETE RULES FOR THE CONNECTIVES

All we have left to do is to state the rules for the remaining connectives. And this further development involves nothing very different from what we have done already. The reasoning that enables us to figure out the new rules is just like the reasoning we went over in explaining the rules for disjunction, negated disjunction, and negated conjunction.

To see how to state the rules generally, let us start with one we have already seen in action:

Rule v: If a disjunction of the form **XvY** appears as the entire sentence at a point on a tree, write the branch



at the bottom of every open path on which **XvY** appears.

Notice that I have stated the general rule using bold face '**X**' and '**Y**' instead of sentence letters 'A' and 'B'. This is because I want to be able to apply the rule to more complicated sentences, which result if we substitute compound sentences for the '**X**' and the '**Y**'. Don't worry about this for now. We will work on this complication in the next chapter.

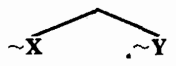
We continue with a statement of the other two rules which we have already seen in action:

Rule ~v: If a negated disjunction of the form ~(**XvY**) appears as the entire sentence at a point on a tree, write the stack

~**X**
~**Y**

at the bottom of every open path on which ~(**XvY**) appears.

Rule ~&: If a negated conjunction of the form ~(**X&Y**) appears as the entire sentence at a point on a tree, write the branch



at the bottom of every open path on which ~(**X&Y**) appears.

Did you try your hand at exercise 8–2? If not, or if you did and had trouble, try again, now that you have seen how I have given a general statement of the rules we have already worked with. Try to figure out the new rules for the other connectives by thinking through what, for example, you would need to do to a tree to make true a conjunction which appears on a tree, or a conditional, or a negated conditional.

In order to make a conjunction true on a path we have to make both conjuncts true. This is the only minimally sufficient way of making the conjunction true. So the rule for conjunction is

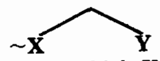
Rule &: If a conjunction of the form **X&Y** appears as the entire sentence at a point on a tree, write the stack

X
Y

at the bottom of every open path on which **X&Y** occurs.

Now let's try the conditional. How can we make a conditional true? If we make the antecedent false, that does it right there. Making the antecedent false is minimally sufficient for making the conditional true. Similarly, making just the consequent true is minimally sufficient for making the conditional true. Clearly, we need a branch, each fork of which has one of the minimally sufficient ways of making the conditional true. One fork of the branch will have the negation of the antecedent, and the other fork will have the consequent:

Rule ⊃: If a sentence of the form **X⊃Y** appears as the entire sentence at a point on a tree, write the branch



at the bottom of every open path on which **X⊃Y** occurs.

What about negated conditionals? A negated conditional produces a stack as do conjunctions and negated disjunctions. A negated conditional can be made true only by making its antecedent true and its consequent false at the same time. So our rule for negated conditionals reads

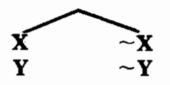
Rule ~⊃: If a sentence of the form ~(**X⊃Y**) appears as the entire sentence at a point on a tree, write the stack

X
~**Y**

at the bottom of every open path on which ~(**X⊃Y**) appears.

The rule for the biconditional is just a bit more complicated. You can't make the biconditional **X≡Y** true just by making one component true or false. You have to assign a truth value to two components to make it true. So far, the rule for **X≡Y** is looking like that for **X&Y.** But there are two independent ways of making **X≡Y** true. We can make it true by making both components true, and we can make it true by making both components false. Finally, these are all the minimally sufficient ways of making **X≡Y** true. So we will have to make a branch, each fork of which will have two sentences:

Rule ≡: If a biconditional of the form **X≡Y** appears as the entire sentence at a point on a tree, write the branch

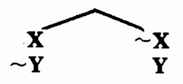


at the bottom of every open path on which **X≡Y** appears.

Note that the rule ≡ looks different from all the previous rules. Each previous rule instructed you to branch or stack, but not both. The rule ≡ requires both branching and stacking. We need a branch with two forks, and each fork has a stack of two sentences. Only thus do we get all the minimally sufficient ways of making a biconditional true.

The reasoning behind the rule for the negated biconditional looks very similar to that for the conditional. A negated biconditional is true just in case the biconditional itself is false. Under what conditions is a biconditional false? It's false when the components have different truth values. This can happen in two ways: The biconditional is false when the first component is true and the second component is false, and it is false when the first component is false and the second component is true. This gives us the rule

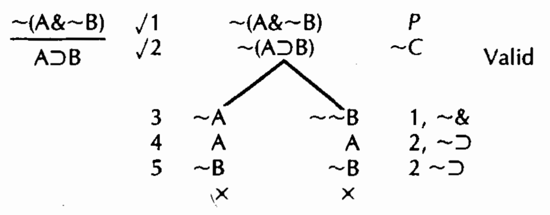
Rule ~≡: If a negated biconditional of the form ~(**X≡Y**) appears as the entire sentence at a point on a tree, write the branch



at the bottom of every open path on which ~(**X≡Y**) appears.

As with the conditional, we need two branches, each with a stack of two sentences. Only in this way will we get all the minimally sufficient ways of making a negated biconditional true.

We need one last truth tree rule. Consider the following argument and the tree for testing its validity:



How did I get the branch on the right to close? We can look at this in two equally correct ways. We can note that '~~B' is the negation of '~B'. Thus we have an inconsistent pair of sentences on the same branch. This is what I hope occurred to you when you met double negations in the previous exercises in this chapter. I can no more make '~B' and '~~B' both true than I can make 'B' and '~B' both true. So the branch closes. Also, we can observe that '~~B' is logically equivalent to 'B'. Clearly, the one will be true if and only if the other is true. So we can make '~~B' true by making 'B' true. We could rewrite the right branch on the above tree as follows:

```
~~B    1, ~&
  B    3, ~~
  A    2, ~⊃
 ~B    2, ~⊃
  ×
```

We will formalize this move in a further rule:

Rule ~~: If the sentence ~~**X** appears as the entire sentence at a point on a tree, write **X** at the bottom of every open path on which ~~**X** appears.

These nine rules for the connectives tell you what to do when working on a sentence which appears as the entire sentence at a point on a tree. The examples should give you a pretty good idea of how to go about applying these rules. But let's review the basic ideas:

A truth tree works by looking for an assignment of truth values to sentence letters which makes all of the tree's original sentences true.

We will see in the next chapter that a truth tree can be used for other things in addition to checking arguments for validity. But so far we have studied only validity checking:

To use the truth tree method to test an argument for validity, list, as the initial sentences of a tree, the argument's premises and the negation of its conclusion.

You then apply the rules in this way:

1) Starting with a tree's initial sentences, you may apply the rules in any order.
2) After applying a rule to a sentence, check the sentence, to remind yourself that you do not need to work on that sentence again.
3) After working on a sentence, look to see if any paths have closed. Mark any closed paths with an '×'.
4) Continue working on unchecked sentences until the tree has been completed, which happens when either

   a) All paths have closed,

   or b) No unchecked sentences are left to which you can apply a rule.

Now we need to review how you should interpret a tree once it has been completed:

> An open path provides an assignment of truth values to sentence letters as follows: Assign the truth value t to each sentence letter which occurs as the entire sentence at some point along the open path. Assign the truth value f to each sentence letter whose negation occurs as the entire sentence along the open path.
>
> In a completed tree the assignment of truth values to sentence letters provided by an open path makes all sentences along the path true. If all paths have closed, there is no assignment of truth values to sentence letters which makes all the initial sentences in the tree true.

When we use a tree to test an argument for validity, we apply this last general fact about trees, as follows:

> Suppose we have a completed tree whose initial sentences are an argument's premises and the negation of the argument's conclusion. An open path in this tree provides an assignment of truth values to sentence letters which makes all of the initial sentences true, and so makes the argument's premises true and its conclusion false. Such an assignment is a counterexample to the argument, showing the argument to be invalid. If all of the tree's paths have closed, there is no such assignment, hence no counterexample, and the argument is valid.

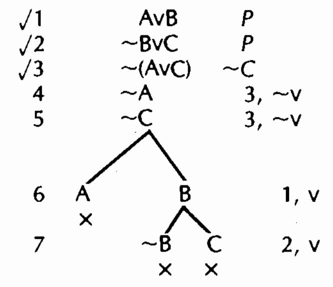
We will also always use the following practical procedure:

> When using the truth tree method to test an argument for validity, write next to your completed tree the word 'valid' or 'invalid' to record what the tree shows about the argument. Also, if the tree shows the argument to be invalid, write down all the counterexamples which the tree provides to the argument.

Of course, one counterexample is enough to show that an argument is invalid. But you should practice your skill at constructing trees, and as part of this practice it is useful to read off all the counterexamples a tree provides.

8–5. IN WHICH ORDER SHOULD YOU WORK ON THE SENTENCES IN A TREE?

The summary statement tells you that you may apply the rules in any order you like. Indeed, the order does not matter in getting the right answer. But the order can make a practical difference as to how quickly you get the right answer. In all the examples you have seen so far, I started with the first sentence on the tree and worked downward. To show you that order can make a practical difference I am going to redo the first example and work on line 3 first:



Compare this tree with the first way I worked the problem, and you will see that this one is a good bit shorter. I worked out the problem the longer way the first time because I wanted to illustrate how branches get stacked on top of several other branches. And I wanted to show you how working on the sentences in a different order can make a difference as to how quickly you can do the problem.

But how can you tell what the shortest way will be? I have no surefire formula which will work for you all the time. In general, you have to try to "look ahead" and try to see how the problem will work out to decide on an order in which to work the lines. Your objective is to get as many branches to close as quickly as possible. If you don't make the best choice, the worst that will happen is that your problem will take a little longer to do.

There are several practical rules of thumb to help you out. First,

> Practical guide: Work on lines that produce stacks before lines that produce branches.

Branches make trees messy. Stacks line up more sentences along a path and improve your chances of getting a path to close. In the problem which I redid I knew I was likely to get a shorter tree by working on line 3 before working on lines 1 and 2. I knew this because line 3 produces a stack while both 1 and 2 produce branches.

A second suggestion:

> Practical guide: When you have as sentences on which to work only ones which are going to produce branches, work first on one which will produce at least some closed paths.

If the first two suggestions don't apply, here is a final piece of advice:

> Practical guide: If the first two guides don't apply, then work on the longest sentence.

If you put off working on a long sentence, you may have to copy the results of working on it at the bottom of many open branches. By working on a long sentence early on, you may get away with its long pieces at the bottom of relatively few branches, making your tree a little less complicated.

Now you should sharpen your understanding of the rules by working the following problems. The easiest way to remember the rules is to understand how they work. Once you understand the reasoning which led us to the rules, you will very quickly be able to reconstruct any rule you forget. But you may also refer to the rule summary on the inside back cover. You should understand the boxes and circles in this way: Whenever you find a sentence with the form of what you see in a box occurring as the entire sentence at a point along a tree, you should write what you see in the circle at the bottom of every open path on which the first sentence occurred. Then check the first sentence. I have written the abbreviated name of the rule above each rule.

EXERCISES

8–4. Use the truth tree method to determine whether or not the following arguments are valid. In each case show your tree, indicating which paths are closed. Say whether the argument is valid or invalid, and if invalid give all the counterexamples provided by your tree.

$$\text{a)}\ \begin{array}{c} A\\ B\\ \hline A\&B \end{array}\qquad \text{b)}\ \begin{array}{c} \sim C\supset H\\ \hline \sim H\supset C \end{array}\qquad \text{c)}\ \begin{array}{c} K\supset F\\ F\\ \hline K \end{array}\qquad \text{d)}\ \begin{array}{c} P\supset M\\ M\supset B\\ \hline P\supset B \end{array}\qquad \text{e)}\ \begin{array}{c} F\lor G\\ \sim G\lor H\\ \hline F\&H \end{array}\qquad \text{f)}\ \begin{array}{c} J\supset D\\ K\supset D\\ J\&K\\ \hline D \end{array}$$

$$\text{g)}\ \begin{array}{c} (K\lor S)\\ \sim(K\&H)\\ \hline K\supset H \end{array}\qquad \text{h)}\ \begin{array}{c} J\supset D\\ K\supset D\\ J\lor K\\ \hline D \end{array}\qquad \text{i)}\ \begin{array}{c} M\equiv N\\ \hline M\equiv\sim N \end{array}\qquad \text{j)}\ \begin{array}{c} T\equiv I\\ I\&A\\ \hline \sim T\lor\sim A \end{array}\qquad \text{k)}\ \begin{array}{c} \sim(F\&\sim L)\\ \sim(L\&\sim C)\\ \hline F\equiv L \end{array}$$

$$\text{l)}\ \begin{array}{c} F\equiv G\\ G\equiv H\\ \hline F\equiv H \end{array}\qquad \text{m)}\ \begin{array}{c} C\supset N\\ \sim(I\supset C)\\ \sim(N\lor\sim I)\\ \hline \sim N\supset\sim I \end{array}\qquad \text{n)}\ \begin{array}{c} \sim(Q\supset D)\\ \sim(Q\&\sim B)\\ \sim(B\&\sim R)\\ \hline D\lor\sim Q \end{array}\qquad \text{o)}\ \begin{array}{c} R\equiv\sim S\\ \sim(R\equiv T)\\ \hline R\&\sim S \end{array}\qquad \text{p)}\ \begin{array}{c} O\equiv\sim F\\ \sim(F\equiv K)\\ \hline O\supset K \end{array}$$

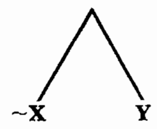
8–5. The rules, as I have stated them, specify an order for the branches and an order for sentences in a stack. For example, the rule for a negated conditional, $\sim(\mathbf{X}\supset\mathbf{Y})$ instructs you to write a stack

$$\begin{array}{c} \mathbf{X}\\ \sim\mathbf{Y} \end{array}$$

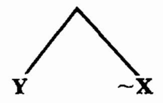
But could you just as well write the stack

$$\begin{array}{c} \sim\mathbf{Y}\\ \mathbf{X} \end{array}$$

at the bottom of every open path on which $\sim(\mathbf{X}\supset\mathbf{Y})$ appears? If so, why? If not, why not? Likewise, the rule for the conditional, $\mathbf{X}\supset\mathbf{Y}$, instructs you to write the branches



But could you just as well write the branches in the other order, writing



at the bottom of every open path on which $\mathbf{X}\supset\mathbf{Y}$ appears? If so, why? If not, why not? Comment on the order of branches and the order within stacks in the other rules as well.

8–6. If we allow ourselves to use certain logical equivalences the truth tree method needs fewer rules. For example, we know from chapter 4 that, for any sentences **X** and **Y**, $\mathbf{X}\supset\mathbf{Y}$ is logically equivalent to ~XvY. Now suppose we find a sentence of the form X⊃Y on a tree. We reason as follows: Our objective is to make this sentence true by making other (in general shorter) sentences true. But since ~XvY is logically equivalent to X⊃Y, we can make X⊃Y true by making ~XvY true. So I will write ~XvY at the bottom of every open branch on which X⊃Y appears, check X⊃Y, and then apply the rule for disjunctions to ~XvY. In this way we can avoid the need for a special rule for conditional sentences.

Apply this kind of reasoning to show that, by appealing to de Morgan's rules, we can do without the rules for negated conjunctions and negated disjunctions, using the rules for disjunctions and conjunctions in their place. Also show that we could equally well do without the rules for conjunctions and disjunctions, using the rules for negated disjunctions and negated conjunctions in their place.

8–7. In chapter 3 I extended the definition of conjunctions and disjunctions to include sentences with three or more conjuncts and sentences with three or more disjuncts. But we have not yet stated truth tree rules for such sentences.

a) State truth tree rules for conjunctions of the form X&Y&Z and for disjunctions of the form XvYvZ.
b) State truth tree rules for conjunctions and disjunctions of arbitrary length.

8–8. Write a truth tree rule for the Sheffer stroke, defined in section 3–5.

CHAPTER SUMMARY EXERCISES

Here are this chapter's important new ideas. Write a short explanation for each in your notebook.

a) Truth Tree
b) Counterexample
c) Branching Rule
d) Nonbranching Rule
e) Closed Branch (or Path)
f) Open Branch (or Path)
g) Rule ~~
h) Rule &
i) Rule ~&
j) Rule v

k) Rule ~v
l) Rule ⊃
m) Rule ~⊃
n) Rule ≡
o) Rule ~≡