c. P is T, Q and R are F

d. Q is T, P and R are F

It is false in the other four cases; hence, it is a contingency.

Section 3.5. Equivalences

In the above basic truth tables of the last section, note particularly the column for '$\leftrightarrow$'. This table says that p and q have the same truth value: if p is true then so is q, and if p is false then so is q. Hence if we wish to say that two formulas are *equivalent* we can do it in one of two ways: (1) we could say that they have the same truth table, or (2) we could say that the result of placing a $\leftrightarrow$ between them is a tautology. Either of these ways can be tested by truth tables. (By the way, we can justify our earlier claims in this manner. Write a truth table for each of $(p\wedge(q\wedge r))$ and $((p\wedge q)\wedge r)$. You will discover them to be the same, so we are justified in our practice of dropping the internal parentheses -- it wouldn't matter which way you added them back on. Now write a truth table for $\sim(p\vee q)\leftrightarrow(\sim p\wedge\sim q)$. You will discover that there are all T's in its final column; so it's a tautology. This means that $\sim(p\vee q)$ and $(\sim p\wedge\sim q)$ are equivalent ways of saying the same thing. Recall that we said that *neither p nor q* could be translated either way.)

Knowledge of certain of these equivalents, especially the "DeMorgan Laws" which relate '$\wedge$', '$\vee$', and '$\sim$' can make your programming life much easier. Most programming languages have "connectives" corresponding to these three. For example, Pascal has 'AND', 'OR', and 'NOT'. One type of "atomic expression" in Pascal is simple equality, greater than, and less than between variables. So 'X<Y', 'A=B', and the like are "atomic expressions" which can be either true or false, and which can be made into compound expressions by means of the connectives. Pascal (and other programming languages) use such expressions to control the action of a loop. Two loop structures in Pascal are

```
WHILE p DO
      <body>
```

and

```
REPEAT <body>
      UNTIL p
```

The "while loop" works as follows: the statement(s) in the "body" are continually performed so long as p is true. p is checked for truth or falsity, and if it is true then the "body" is performed and p is again checked. If it is true the process is repeated. When p is false the "body" is not performed, and control is passed to the next statement in line. The "until loop" performs the "body" until p becomes true -- that is, as long as p is false. (Actually, it performs the body, then checks for p's truth or falsity. If p is false, then it does it all again. If p is true, control is passed to the next statement.)

Suppose you wish to perform some action as long as some complex state of affairs is true. Let's say you want to perform "body" as long as either A<B or B=50. If you were to use the while loop you would say

```
WHILE ((A<B) OR (B=50)) DO
      <body>
```

If you were to use an until loop you would say

```
REPEAT <body>
      UNTIL NOT((A<B) OR (B=50))
```

Using the DeMorgan Laws you could alternatively put this last loop as

```
REPEAT <body>
      UNTIL (NOT(A<B) AND NOT(B=50))
```

But these are easy cases. Suppose instead you want a loop to stop when either A>B or A<C, and you were to use a while loop. You should say to yourself: "The loop is to stop when one or the other of these statements becomes true. That is, it continues so long as the disjunction is false." So you write

```
WHILE NOT((A>B) OR (A<C)) DO
      <body>
```

You might also appeal to DeMorgan's Laws and say to yourself: "So it stops when one or the other of the statements becomes true. So it must continue as long as they're both false." So you might write

```
WHILE (NOT(A>B) AND NOT(A<C)) DO
      <body>
```

Suppose the desire was instead to write a while loop to stop when both A=B and B=C. You say "So it is to stop when (A=B AND B=C); it must therefore continue as long as this compound statement is false." So you write

```
WHILE NOT((A=B) AND (B=C)) DO
      <body>
```

Or, using DeMorgan's Laws, you write

```
WHILE (NOT(A=B) OR NOT(B=C)) DO
      <body>
```

Finally suppose you want to loop to stop when neither A=B nor B=C. Again you say: "Stops when NOT(A=B OR B=C). Therefore continues so long as this compound statement is false." So you write

```
WHILE ((A=B) OR (B=C)) DO
      <body>
```

A good grasp of these simple equivalences helps programming a lot. If you don't know the answer off the top of your head, you can always write a truth table to figure it out.

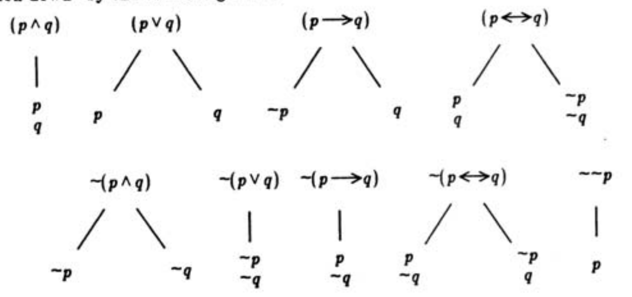
Section 3.6. Truth Table Shortcuts and Related Methods

3.6.1. Some Shortcuts

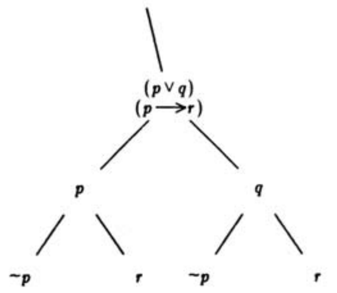
As mentioned above, to consider every possible assignment of T or F to each of the n sentence letters of a given complex sentence would require a truth table with 2^n rows in it. When there are more than three different sentence letters in a sentence, the construction of an entire truth table becomes extremely long and tedious. Therefore various shortcuts have been developed to aid in evaluating such sentences. Using the shortcuts depends on what you wish to show about the sentence in question. For example, if you wonder whether the formula is a tautology, you might check only those rows which you don't already know that it's true. Consider, for example, a sentence like $(p\wedge q)\rightarrow p$. We know from the '$\rightarrow$' truth table that the only time an '$\rightarrow$' statement can be false is when its antecedent is true. And since the antecedent is $(p \wedge q)$ and the only time a conjunction is true is when both conjuncts are true, this is the only case we have to look at. In other words, we need only look at the first row of the truth table. As we discover when looking at this row, the consequent, p, is also true. (And therefore, according to the '$\longrightarrow$' truth table, the entire sentence is true in this row). But as we said, we already knew that this was the only possible case where the formula might have been false, and so it must be a tautology because even here it is true. Alternatively, we might have said to ourselves that the only time this sentence could have been false was when the consequent was false. Therefore the only lines of the truth table we need look at are the last two (where p is false). But then we notice that in these two rows, $(p \wedge q)$ is false, so that the entire conditional must be true. So, you say to yourself, even in these rows the sentence is true, so it must be a tautology.

3.6.2. Truth Trees

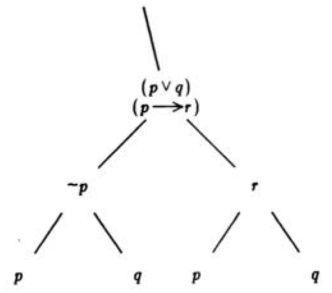
Another method often used in evaluating sentences for tautologousness (and, as we shall see later, for evaluating the validity of arguments) is called "truth trees". In a wide variety of cases the truth tree method (which we will state as an algorithm) can quickly yield an answer to whether a formula is a tautology, and if not, what rows of the truth table create the F's which prevent the formula from being a tautology. Before giving the algorithm, let us introduce "branching rules". There are nine such rules: one for each binary connective and one for the negation of any connective. The idea is that we shall construct a "tree" (don't worry about what precisely a tree is -- the idea will be clear enough) to test the formula we're interested in. Depending on just how the formula is constructed by the connectives, we shall "break it down" using the branching rules, thereby constructing a diagram (a "tree"). The nine branching rules tell us how to break down any complex formula (except simple negations) into its parts by making the tree bigger. Of course, when you "break down" a formula you might discover that one of its parts is itself complex, and so you have to "break it down" by using the branching rule appropriate to it. It is the continual "breakup" of complex formulas which constructs the truth tree. Every complex formula can be "broken down" by the branching rules.



As you can see, some of the rules introduce more than one formula on a node of the tree. Each of these formulas is eligible for being broken up, although you operate on just one at a time. This means we can be led to such sequences of "breakdowns" as the following. We might have a node of the tree with two formulas on it, and "break up" one of them (the order of breakdown never matters, except for efficiency), yielding two different branches. Now, to break up the other formula, we have to break it up on *both* branches, like this



Here we branched the $(p \vee q)$ to yield two branches, and then branched the $(p \longrightarrow r)$ to yield two branches under each of the former branches. We could have done it the other way: branch $(p \longrightarrow r)$ first to yield two branches and then branch $(p \vee q)$. But here too we must branch the second formula under both of the branches we got from $(p \longrightarrow r)$.



You should note -- we will mention why it is important later -- that the branching rules all have the property that: the formula being branched is true if and only if every formula on at least one branch is true. For instance, $(A \vee B)$ yields an A branch and a B branch. The $(A \vee B)$ is true just in case at least one of A and B is true. $\sim(A \longrightarrow B)$ yields only one branch, which contains both A and $\sim B$. $\sim(A \longrightarrow B)$ is true (that is, $(A \longrightarrow B)$ is false) exactly in case A and $\sim B$ are both true (that is, A is true and B is false). The same is true for all our branching rules.

We are now ready to state the tautology-testing algorithm using truth trees. The algorithm for evaluating a formula says to do the following.

(1) start by writing *the negation* of the formula to be tested as the root of the tree. (The negation of the *entire* formula).

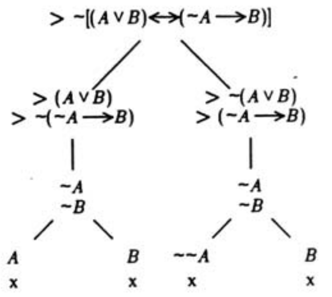
(2) Apply a branching rule to any unchecked formula in the tree, writing the result of the branching under every "active" (to be defined soon) path beneath the formula,

(3) check off (with a '>') the formula you just used the branching rule on,

(4) if any path from the root of the tree to a leaf contains a formula and also the negation of that formula, put an 'x' under that path. Any path with an 'x' under it is not active and need be considered no further. All other paths are active.

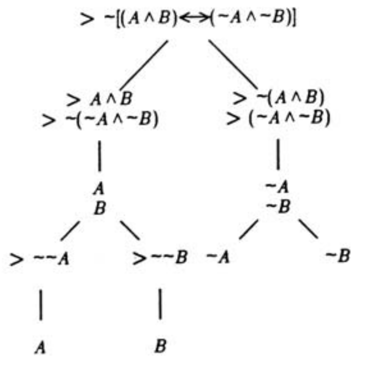
(5) Repeat steps (2)-(4) until either (a) there are no more active paths, or (b) there is an active path but there are no more unchecked complex formulas on that path.

If you reach step (5a), then the original *unnegated* formula was a tautology. If you reach step (5b) then the original *unnegated* formula was not a tautology and every open path describes some row (or rows) of the truth table where it was false. Let us consider two examples, the first of a tautology and the second of a contingency. The first formula is $(A \vee B) \leftrightarrow (\sim A \rightarrow B)$



Since no path is active, the original formula, $(A \vee B) \leftrightarrow (\sim A \rightarrow B)$ is a tautology.

The second formula, which is not a tautology, is $(A \wedge B) \leftrightarrow (\sim A \wedge \sim B)$



Here every path is active. (The fact that *all* paths are active is irrelevant, all that's important is that *some* path is still active.) Each open path describes some row of the truth table according to which $[(A \wedge B) \leftrightarrow (\sim A \wedge \sim B)]$ is false. We can describe the relevant rows by looking at a path and seeing which atomic sentences occur in it unnegated and which occur negated. In the leftmost branch, for example, both A and B occur. The relevant row is where both A and B are true. The second-to-left path also has both A and B in it and therefore describes the same row of the truth table. The third and fourth paths both have $\sim A$ and $\sim B$ in them, and the relevant row of the truth table is where both A and B are false. (When checking an active path, it may happen that some sentence letter does not occur in it. In this case, we have described two rows of the truth table, one where the missing letter is T and one where it is F. If two letters are missing from a path, then we have described four rows of the truth table, etc.) These rows of the truth table which show that the formula is not a tautology are called ***counterexamples*** (to the formula).

This tree method gives a rather quick way of checking truth tables, especially in those cases where there are a lot of sentence letters. The method works because of the interplay of two things. First, the branching rules have the properties that, if the branched sentence is true then so is at least one of the branches, and conversely. This means that every open path describes a way that all the sentences on that path might be true, including the sentence at the root. And since these are *all* the ways a sentence might be true, it follows that there are no rows of truth tables which have been ignored. Second, we have placed the negation of the sentence at the root. So if all paths are non-active, there is no way this negation can be true; and hence the unnegated sentence must be a tautology. If some path is still active, then it *does* describe a way the root (negated sentence) can be true, and hence a way the unnegated sentence can be false.

Section 3.7. Arguments

An *argument* is a set of sentences in which one of the sentences (called the *conclusion*) is claimed to "follow from" the others (called the *premises*). For example, the following collection of sentences forms an argument.

> Either it is not the case that Leslie pays attention and does not lose track of the argument, or it is not the case that she does not take notes and does not do well in the course. Leslie neither does well in the course nor loses track of the argument. If Leslie studies logic, then she does not do well in the course only if she does not take notes and pays attention. Therefore Leslie does not study logic.

You can tell this is an argument in part by the context -- here someone is trying to convince you of something (that Leslie does not study logic) by adducing some reasons. Also the fact that the word *therefore* occurs here gives the information that you are supposed to become convinced. Such words as *therefore, hence, so, it follows that,* and the like, function as conclusion-indicators. They tell you that the sentence following is the conclusion of the argument. (Not all arguments in English will necessarily have the conclusion at the end). Words like *because, for the reason that, on account of,* and the like, are premise-indicators telling you that the associated sentences function as premises to the argument.

Virtue, in an argument, is called *validity*. An argument is valid if and only if the conclusion really *does* "follow from" its premises. Conversely, a bad argument is called *invalid*. So the question arises: how can we determine whether an argument is good or bad, virtuous or unvirtuous? There are, generally speaking, two ways that can be used. One way is to construct an explicit formal proof of the translation into symbolic form of the argument. This method will be discussed later in this chapter. The other way is to use one of the truth table methods. This way rests upon understanding what "follows from" means in terms of the truth or falsity of the parts of an argument.

Definition 3.7.1: valid argument

> An argument is (truth functionally) *valid* if and only if: In any row of a truth table in which all the premises are true, so is the conclusion.

To determine whether an argument is valid according to this definition, you need to write a truth table (or use some other truth method). If you wrote a truth table it would contain columns for *every* atomic sentence letter that occurred anywhere in the argument. This typically means that the truth table would be very large. If this method is employed, you write a column for each premise and for the conclusion. Once their truth tables have been constructed it is a simple matter to see whether there is any row in which all the premises are true and the conclusion false. If there is such a row, then the argument is invalid, otherwise it is valid. If it is invalid, then the row found describes (one) *counterexample:* the T's and F's assigned in that row to the atomic sentences tells you what state of affairs could happen in the world that would make all the premises be true and the conclusion false. That is, it shows you why the argument is invalid.

Of course not every argument in ordinary language is so carefully stated as the one above. Often, certain premises are left out especially when the speaker views them as "obvious". According to the definition above, such enthymatic arguments (as they are called) are invalid. But in another sense they are good arguments and could be stated validly merely by making explicit these unstated, "obvious" premises. Finding the literal counterexample can often help in uncovering precisely what the "hidden" premises are.

Let us symbolize the argument above. We use the following scheme of abbreviation.

P: Leslie pays attention

L: Leslie loses track of the argument

N: Leslie takes notes

W: Leslie does well in the course

S: Leslie studies logic

And the argument is symbolized as (can you get the same answer?):

$$\sim(P \wedge \sim L) \vee \sim(\sim N \wedge \sim W)$$
$$\sim(W \vee L)$$
$$S \longrightarrow (\sim W \longrightarrow (\sim N \wedge P))$$
$$\therefore\ \sim S$$

We note that there are five atomic sentences here, so a complete truth table would have 2^5, i.e., 32, lines in it. It could be written, but it would be tedious. We could also try one of the shortcut methods. Since we wish to know whether it is valid, we are interested in whether it is possible that all premises are true and the conclusion false. (If this *is* possible, then the argument is invalid.) To try to find this out, we might start by making $\sim S$ be false, that is we start by assigning

S: T

The second premise is supposed to be true (as are all premises), so, since it is a negation, the internal disjunction should be false, therefore

S: T
W: F
L: F

If the third premise is to be true, then since S is true, $\sim W \longrightarrow (\sim N \wedge P)$ must also be true. But since $\sim W$ is already required to be true, it follows that $(\sim N \wedge P)$ must be true, i.e.,

S: T
W: F
L: F
N: F
P: T

Given all this, let's see if we have succeeded in making the first premise true. If we have, the argument is invalid. If we haven't, then since we were forced to use these assignments to make premises two and three true and the conclusion false, the argument must be valid. (Because there is no possible way to make all premises true and the conclusion false). We could write a truth table for premise one to see whether it is true or false in the row described by the already-given assignment of truth values to the atomic sentences. But we could also do it in a quicker manner. Since L is false, $\sim L$ is true; and since P is true, it follows that $(P \wedge \sim L)$ is true. So, the first part of premise 1, $\sim(P \wedge \sim L)$, is false. Therefore, if premise one is to be true, the second disjunct must be true. But since both W and N are false, $(\sim N \wedge \sim W)$ must be true and so $\sim(\sim N \wedge \sim W)$ must be false. Therefore, we cannot make all premises true and the conclusion false, and so this is a valid argument.

Let's retest this argument by truth trees. Recall that the account of truth trees given earlier tested an individual sentence for being or not being a tautology: We negated the sentence and checked whether there were any active paths left in the tree. If not, the sentence was a tautology (since there was no possible way for the negation of the sentence to be false). If so, it wasn't and we could find at least one counterexample. With a minor modification of the algorithm given there (the only change is in step 1) we can apply that method to arguments.

1. At the root of the tree, list all premises and the *negation* of the conclusion.

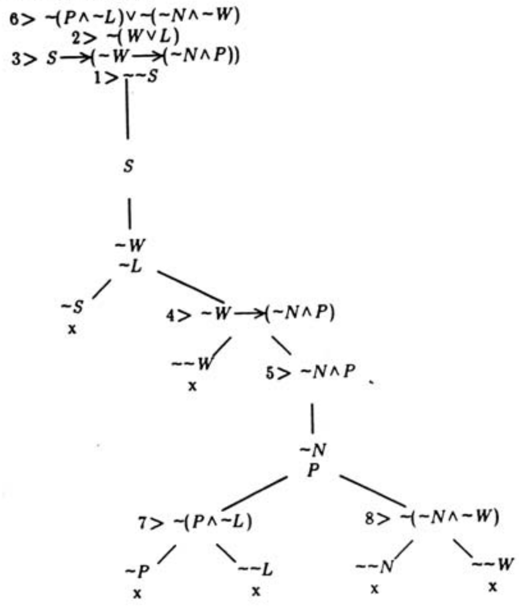
2. Apply a branching rule to any unchecked formula in the tree, writing the result under every active path beneath the formula.

3. Check off the formula you just branched.

4. If any path from the root of the tree to a leaf contains a formula and also the negation of that formula, put an 'x' under that leaf. Any path with an 'x' under it is not active.

5. Repeat steps (2)-(4) until either (a) there are no more active paths, or (b) there is an active path but no more unchecked complex formulas.

If you reach step (5a) then the argument is valid, because it is not possible to have all the premises and also the negation of the conclusion all be true. If you reach step (5b), then it *is* possible, and you can use the method described earlier to find the counterexample(s) to the argument. Here is the truth tree method applied to the above argument. (The numbers beside the "checks" merely give the order in which we applied the branching rules).



Since all paths close, the argument is valid.

Section 3.8. Normal Forms

A *normal form* of a formula is a (possibly different) formula which has the same truth table as the original formula and furthermore is written in a certain way. For example, we might want our formulas not to have any $\longleftrightarrow$ in them. We could replace any part of a formula that does have a $\longleftrightarrow$ in it by a truth table equivalent representation, for instance replace any part like $(p\longleftrightarrow q)$ with $((p\longrightarrow q)\wedge(q\longrightarrow p))$. This would give us a formula which we might call the biconditional-free normal form. This is not a particularly interesting type of normal form, but other types are more interesting.

In working with logical formulas, it is often convenient to restrict our attention to formulas of some particular simple form. In this section, we will define several types of "normal forms" for formulas, and we will show how to convert any formula into normal form.

Objective 3.8.1
Give reasons for studying normal forms.

Objective 3.8.2
Define disjunctive normal form and conjunctive normal form.

Objective 3.8.3
Show how to convert any propositional formula into disjunctive or conjunctive normal form.

Objective 3.8.4
Illustrate some applications of normal forms.

Before we can begin to talk about normal forms, we must specify precisely what it means for two formulas to be "equivalent". Once we have introduced the appropriate definitions and notation for equivalence, the definitions of normal forms will follow quite naturally.

Definition 3.8.1: logical equivalence
Two formulas are ***logically equivalent*** (or just ***equivalent***, for short) if they represent the same propositional function.

In other words, two formulas (or complex propositions) are equivalent if they have the same truth table. (Strictly speaking, this is not true. The formulas q and $((p\longrightarrow q)\wedge(\sim p\longrightarrow q))$ are equivalent, since they represent the same propositional function, but their truth tables have different numbers of rows and columns. Note that the truth tables are the same, however, in the sense that for every row with a T in the p column there is an otherwise identical row with an F in the p column. When the p column is deleted, and the identical rows are merged, the two truth tables become identical.)

Notation: $\equiv$
If p and q are equivalent formulas, we write $p\equiv q$.

In order to be able to talk about propositional formulas in general, we give a special name to the set of all formulas.

Definition 3.8.2: F
$\mathbf{F} = \{p \mid p \text{ is a propositional formula}\}$.

Then $\equiv$ is simply a relation on $\mathbf{F}$. In fact, $\equiv$ is an equivalence relation. (We go into this more in Chapter 6. You might look at this now and try to think of what you have to show in order to prove this fact?) This justifies our use of the term "equivalent" for $\equiv$.