LOGIC: A COMPUTER APPROACH

Morton L. Schagrin
State University of New York at Fredonia

William J. Rapaport
State University of New York at Buffalo

Randall R. Dipert
State University of New York at Fredonia

Ch. 6: Algorithms for Truth Tables & Determining Validity

Wang's Algorithm: pp. 114ff

McGraw-Hill Book Company
New York St. Louis San Francisco Auckland Bogotá Hamburg Johannesburg London Madrid Mexico Montreal New Delhi Panama Paris São Paulo Singapore Sydney Tokyo Toronto

© 1985

CHAPTER 6

SENTENTIAL LOGIC: Algorithms for Truth Tables and Determining Validity

In Chapters 3 and 4, we frequently used truth tables: extensional characterizations of truth functions. These were two-dimensional displays showing the functional relationship between the truth value of a molecular sentence and the truth values of its parts. In mathematics, such a table would be called a "matrix"; in computer science, it would be called a two-dimensional "array."

For disjunction, we wrote:

V(P)	V(Q)	V(P v Q)
0	0	0
0	1	1
1	0	1
1	1	1

The two columns on the left display the possible combinations of truth (1) and falsity (0) that the sentences **P** and **Q** may take in various "situations." We might think of each row as describing a situation. Up until now, the right-most column has displayed the truth values of a single molecular sentence in the situations described by the truth values of the sentences that are its parts. For example, in the above table, the first row describes the situation in which both **P** and **Q** are FALSE; in that situation, the disjunction (**P** v **Q**) is also FALSE.

The second row describes the situation in which the disjunction (**P** v **Q**) is TRUE; and so on for the other rows.

Since we have already fully described in Chapter 5 how to calculate the truth value of a molecular sentence, no matter how complex, given the truth values of its atomic parts, we can now produce truth tables of more complexity, such as:

V(P)	V(Q)	V(P → (P → (P & Q)))
0	0	1
0	1	1
1	0	0
1	1	1

We calculate each of the values in the last column by using the algorithm from Chapter 5. To do so for the first row, we replace each occurrence of **P** with '0' and each occurrence of **Q** with '0' and then calculate the value of the resulting hybrid:

(0 → (0 → (0 & 0)))

which will turn out to be 1. We could also have seen that this is the case by observing that V(0 → *n*) = 1 no matter what the value of *n* is.

We can also use a truth table to show how *several* molecular sentences are dependent on the truth values of their atomic parts. This kind of truth table will turn out to be very useful in determining the validity or invalidity of arguments in the logic of sentences, as we shall soon see.

Consider this truth table:

V(P)	V(Q)	V(R)	V(P → Q)	V(Q → R)	V(P → R)
0	0	0	1	1	1
0	0	1	1	1	1
0	1	0	1	0	1
0	1	1	1	1	1
1	0	0	0	1	0
1	0	1	0	1	1
1	1	0	1	0	0
1	1	1	1	1	1

The first row shows that whenever **P** is FALSE, **Q** is FALSE, and **R** is FALSE, then the truth value of (**P** → **Q**) is TRUE, the truth value of (**Q** → **R**) is TRUE, and the truth value of (**P** → **R**) is TRUE. In the whole truth table, we have a display of the truth values of (**P** → **Q**), (**Q** → **R**), and (**P** → **R**) as functions of all the combinations of truth values of their atomic parts.

General Truth Tables

The previous large table suggests the following generalization of the format of a truth table.

Truth Values of Atomic Sentences	**Truth Values of Sentences Dependent on These Atomic Sentences**
Situations: All Possible Combinations of Truth Values for the Atomic Sentences	

There are some general observations that can be made about truth tables:

1. The atomic sentences in which we are interested—displayed in the *left* columns—are just those that are contained in the dependent sentences. Thus if the sentences we are considering are (**P** → **Q**), (**Q** → **R**), and (**R** → **S**), then the atomic sentences to be displayed are **P, Q, R,** and **S.**
2. If there are *n* atomic sentences in which we are interested, then there are 2 ** *n* possible combinations of truth values for these atomic sentences. The reason for this is that there are only *two* possible values that *each* atomic sentence can have. So for two atomic sentences, there are 2 ** 2 = 2 × 2 = 4 combinations (00, 01, 10, 11); for three, there are 2 ** 3 = 2 × 2 × 2 = 8; and so on. (We use '*a* ** *n*' for *a* raised to the *n*th power.)
3. An orderly way to place truth values in the rows under each atomic sentence, so as to exhaust all 2 ** *n* combinations, is as follows. In row 1, place the binary numeral for 0, beginning at the far right. In row 2, place the binary numeral for 1, also at the far right. In general, in row *k*, place the binary numeral for *k* − 1 at the far right. The last row will be a string of *n* '1's. Finally, fill in all remaining blanks under the atomic sentences with '0's.

As an example, suppose we have three atomic sentences, **P, Q,** and **R.** We thus need 2 ** 3 = 8 rows for our truth table. The first row has '0' squeezed as far right as possible, the second has '1', the third has '10', the fourth has '11', and so on. The result is:

V (P)	V (Q)	V (R)	
		0	
		1	
	1	0	
	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

We then complete the procedure by filling in the blanks with '0's, obtaining:

V (**P**)	V (**Q**)	V (**R**)
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

We finish the truth table by displaying the sentences whose truth values we are interested in at the top of other columns to the right. We then fill in the rows under these sentences with the results of our calculations of their truth values (using TRUTH-VALUE CALCULATOR), using the truth values of the atomic sentences indicated at the beginning of each row.

The Algorithm: Truth-Table Generator

Having just informally described the construction of a truth table, we can now give an algorithm for generating one.

```
ALGORITHM TRUTH-TABLE GENERATOR

1. INPUT the sentences whose truth-functional relationships we are investi-
   gating.
2. Let m = the number of these sentences. (For an argument, m will be the
   number of premises plus one (for the conclusion).)
3. Let n = the number of distinct atomic sentences occurring in these sen-
   tences.
4. Create a table with n + m columns and 2 ** n rows.
5. FOR each atomic sentence S
     (a) Write 'V(S)' at the top of the columns, beginning at the left.
6. FOR each sentence S whose truth-functional relationships we are investi-
   gating
     (a) Write 'V(S)' at the top of the remaining columns, beginning with column
         n + 1.
7. (a) Count in the binary system from 0 to (2 ** n - 1).
   (b) Place these binary numerals, one to each row, squeezed right under
       the first n columns, one digit to a box. (That is, in the first row, the
       nth column will have '0'. In the second row, the nth column will have
       '1'. In the third row, the (n - 1)th column will have '1' and the nth
       column will have '0'—and so on.)
   (c) When the last row has been filled, return to earlier rows, placing '0'
       in any of the spaces in the first n columns that were left blank.
8. FOR each column c from n + 1 to n + m
     (a) FOR each row r from 1 to 2 ** n
           i. OUTPUT the truth value produced by TRUTH-VALUE CAL-
              CULATOR, using as input the sentence at c and the truth values
              at r.
9. STOP.
```

For example, if our three initial sentences are (**P** → **Q**), (**Q** → **R**), and (**P** → **R**), this algorithm will output the following truth table:

V (**P**)	V (**Q**)	V (**R**)	V (**P** → **Q**)	V (**Q** → **R**)	V (**P** → **R**)
0	0	0	1	1	1
0	0	1	1	1	1
0	1	0	1	0	1
0	1	1	1	1	1
1	0	0	0	1	0
1	0	1	0	1	1
1	1	0	1	0	0
1	1	1	1	1	1

Determining Validity

Truth tables can be used in the logic of sentences for determining the validity or invalidity of an argument. To see this, you should recall what an argument is and what a *valid* argument is. An argument is a set of sentences, one sentence of which is identified as the conclusion—which is claimed to follow from the other sentences (the premises). An argument is valid iff

In every situation where the premises are all TRUE, the conclusion is also TRUE.

or, equivalently,

There is no situation where the premises are all TRUE and the conclusion is FALSE.

An argument is *in*valid iff

There *is* some situation where the premises are all TRUE and the conclusion FALSE.

The "situations" in the logic of sentences are just different combinations of the truth and falsity of *atomic* sentences (as described by the first *n* columns). Our truth tables have been constructed to exhaust all possible situations: Each row represents a different situation, and, all together, the different rows exhaust all possible situations.

If we had reserved the right-most columns except for the last—that is, the (*n* + 1)th through the (*n* + *m* − 1)th columns—for the *premises* of an argument and the *last*, or (*n* + *m*)th, column for the *conclusion*, then we could use this truth table to decide whether that argument is valid or invalid. Our reasoning would be as follows:

If there is any row containing a '1' (TRUE) in every premise-column and a '0' (FALSE) in the conclusion-column, then the argument is invalid.
If there are no such rows, then the argument is valid.

Consider the following argument:

1. If the Soviet Union rejects the proposed treaty, then a final agreement will be postponed.
2. The Soviet Union rejects the proposed treaty.
   So,
3. A final agreement will be postponed.

We could symbolize this argument as follows:

1. (A → B)
2. A
∴ 3. B

We use ∴ as a sign for 'So' or 'Therefore', standard conclusion indicators.

Is this argument valid? Using the truth-table method we have just proposed, we answer this question as follows. We construct a truth table, using some of the procedures described in TRUTH-TABLE GENERATOR.

Step 1: There are three sentences in the argument: two premises and a conclusion. So, *m* = 3.
Step 2: These sentences contain just two atomic sentences (A, B). So, *n* = 2.
Step 3: Thus our table will have *n* + *m* = 2 + 3 = 5 columns, and 2 ** *n* = 2 ** 2 = 4 rows.
Steps 4–7: Labeling the columns and filling in the rows, we have:

V (A)	V (B)	V (A → B)	V (A)	V (B)
0	0	1	0	0
0	1	1	0	1
1	0	0	1	0
1	1	1	1	1

This display contains a great deal of information. The first row tells us that in the situation where the sentences 'The Soviet Union rejects the proposed treaty' and 'A final agreement will be postponed' are both FALSE, the sentence 'If the Soviet Union rejects the proposed treaty, then a final agreement will be postponed' is TRUE. In this situation, the first premise is TRUE, the second premise is FALSE, and the conclusion is FALSE. Each of the last three rows describes a different situation and indicates whether the premises and conclusion are TRUE or FALSE in that situation.

We can use this information to determine whether the argument is valid or invalid. If there is a row where both premises are TRUE and the conclusion is FALSE, then the argument is invalid; if there is no such row, then the argument is valid. Inspection of the above truth table will show that there are *no* rows where the premises are TRUE and the conclusion is FALSE. Hence, the argument is valid.

In fact, not only is *this* argument (involving a treaty and the Soviet Union) valid, but any argument of the same *form* is also valid. That is, any argument of the form

If <Sentence 1> then <Sentence 2>
<Sentence 1>
∴ <Sentence 2>

is valid. Arguments of this form are said, in traditional Latin terminology going back to the Middle Ages, to have used the inference pattern *modus ponens*. Any instance of *modus ponens* is valid.

Let us consider another argument:

1. If Harry uses heroin, then Harry once smoked marijuana.
2. Harry once smoked marijuana.
   Therefore,
3. Harry uses heroin.

Symbolized, the argument becomes:

1. (H → M)
2. M
∴ 3. H

Before continuing, you should notice the difference between the pattern in this example and the pattern in the previous example. In this example, the second premise is the *consequent* of the conditional (the first premise). And the conclusion is the *antecedent* of that conditional. That is, the pattern of this new argument is:

If <Sentence 1> then <Sentence 2>
<Sentence 2>
∴ <Sentence 1>

Fallacies are patterns of invalid arguments that unfortunately are often used in everyday life. This particular fallacy is called "affirming the consequent." In the earlier example, the second premise was the *antecedent* of the conditional, and the conclusion was the *consequent* of the conditional.

Our truth table for this new argument would be:

V (H)	V (M)	V (H → M)	V (M)	V (H)
0	0	1	0	0
0	1	1	1	0
1	0	0	0	1
1	1	1	1	1

Here, we can see that there *is* a row in which the premises are both TRUE but the conclusion FALSE: the second row. The second row describes the situation in which Harry does smoke marijuana but does not use heroin. In this situation, both premises are TRUE, but the conclusion is FALSE. Since there is such a row, the argument is *invalid*. In a valid argument, the truth of the premises "guarantees" the truth of the conclusion. However, as we have just seen, the truth of the premises of this argument does not guarantee the truth of the conclusion.

The Algorithm: Validity/Invalidity Determiner

The truth-table method we have just been exploring provides the basis for an algorithm for determining whether or not an already symbolized argument in the logic of sentences is valid. Applied properly, it will always yield the correct answer in a finite amount of time, and applying it requires no guesswork or special imagination.

Described in full, our method for evaluating the validity of an argument in the logic of sentences would consist of the following steps:

```
ALGORITHM VALIDITY/INVALIDITY DETERMINER

1. INPUT a symbolized argument.
2. Apply TRUTH-TABLE GENERATOR.
   (Note: In step 6 of TRUTH-TABLE GENERATOR, the $(n + m)$th column
   should contain the truth values of the conclusion.)
```

```
3. IF there is a row in which each box in a premise column (that is, the $(n + 1)$th
   through $(n + m - 1)$th columns) contains '1' and the box in the last column
   contains '0'
     THEN OUTPUT "Argument invalid" and STOP.
4. IF there is no such row
     THEN OUTPUT "Argument valid" and STOP.
```

Step 3 is, of course, the critical step, since it is here that we actually obtain the result of whether the argument is, or is not, valid. It can be refined as follows:

```
3.1. FOR each row
     (a) Let COLUMN = $n + 1$.
     (b) WHILE COLUMN $\leq n + m - 1$ and '1' occurs in this row and column
           Let COLUMN be COLUMN + 1.
     (c) IF COLUMN = $n + m$
           THEN IF '0' occurs in this row and column
           THEN OUTPUT "Argument invalid" and STOP.
```

This procedure begins by scanning across the first row (step 3.1), starting in the first premise-column, $n + 1$ (step 3.1(a)). As long as there are still premises and each one is TRUE, the scanning continues (step 3.1(b)). If a FALSE premise is found, then the next row is scanned from the first premise. If all premises are TRUE, then the conclusion (in column $n + m$) is examined (step 3.1.(c)). If the conclusion is FALSE, then the argument is invalid (step 3.1(c)i) and the algorithm stops; otherwise, the *next* row is scanned from the first premise. If all rows are examined and a row with all TRUE premises and a FALSE conclusion is not found, then the argument is valid (step 4).

Alternative Methods

If the number of atomic sentences involved in our premises and conclusion is relatively large, the resulting truth table may approach awesome proportions. For the following straightforward—and invalid—argument:

1. $(A \rightarrow B)$
2. $(B \rightarrow C)$
3. $(C \rightarrow D)$
4. $(D \rightarrow E)$
5. E

∴ 6. A

the truth table has 2 ** 5, or 32, rows, and 11 columns are required. Since the truth table has 32 × 11 boxes, there are 352 truth values to be written into them. In other cases, some molecular sentences may be quite complex, such as:

$$((A \rightarrow (\sim B \lor (C \;\&\; \sim D))) \lor \sim(\sim D \rightarrow A))$$

The truth value of each such sentence must be recalculated for every row, using TRUTH-VALUE CALCULATOR. In short, the truth-table method may consume large amounts of paper, ink, and our valuable time.

If we program a computer to perform this procedure, then TRUTH-VALUE CALCULATOR presents no serious difficulties. The computer will not use paper and ink in its calculations; instead it will use electrical states in memory locations as its "truth table." And even arguments containing many atomic sentences or having long, complicated sentences would not consume *our* time—although they would consume *computer* time.

So, in principle, we could use a computer to apply the truth-table method of determining validity to arguments of normally intimidating size or complexity. Of course, we may not always have a computer available, or we might not have the skill or patience to program it properly. Furthermore, the full truth-table method, with its maze of truth values, seldom resembles the way we actually reason. It is unlikely that a person proposing an argument used the truth-table method to devise it. And it is very unlikely that most people hearing the argument—even logicians—would think it to be valid (or invalid) because of a full truth table they were doing in their heads. Are there, we might ask, quicker, easier, or more natural ways of determining whether an argument is valid or invalid?

There are indeed such ways, but they are not as powerful as we might desire. We can, in what is called a "formal deduction system," construct a "proof" of a valid argument. If we can find a correct proof, we then know that the argument is valid. Unfortunately, if we cannot find a proof, we cannot conclude that the argument is invalid. Our failure to find a proof might have been due to our own inability, or perhaps we didn't try hard enough. In this case, the argument might still be valid; we could then fall back on the full truth-table method to determine if it is valid or invalid. In Chapters 8 and 9, we shall discuss the method of showing an argument to be valid by giving a proof.

WANG'S ALGORITHM

Another algorithm to determine whether or not a symbolized argument is valid is WANG'S ALGORITHM, named after the logician Hao Wang (1921–). WANG'S ALGORITHM, like the truth-table algorithm VALIDITY/INVALIDITY DETERMINER, will always determine whether a symbolized argument in the logic of sentences is valid or invalid. But unlike the truth-table algorithm, it does not require us to create large tables.

The idea behind WANG'S ALGORITHM is very simple. In constructing a truth table to determine the validity of an argument, we are searching for a situation in which the premises are all TRUE but the conclusion is FALSE. If there is such a situation, then the argument is invalid. If there is no such situation, then the argument is valid. In terms

of the truth table, we look for the *row* where the premises are all TRUE and the conclusion FALSE. WANG'S ALGORITHM is designed to find this situation (row) *without* having to examine all situations. The algorithm will also tell us if there exists no such situation.

Before we introduce WANG'S ALGORITHM in full detail, we shall look at some examples showing the principles that lie behind it. Consider this argument:

1. A
2. B

∴ 3. A

Let us consider an attempt to make the premises TRUE and the conclusion FALSE. To do so, we shall write the sentences of this argument in a new way. On the left, we shall write a list of the sentences we are trying to make TRUE, and on the right, we shall write a list of the sentences we are trying to make FALSE. We shall draw a vertical line to separate the two columns. In the case of this argument, we write:

$$\begin{array}{c|c} A & A \\ B & \end{array}$$

In other words, we are trying to make the sentences A and B TRUE and, in the same situation, make A FALSE. But it should be easy to see that we cannot make the same sentence both TRUE and FALSE in the same situation. Since sentence A appears on both the list of sentences we are trying to make TRUE and the list of sentences we are trying to make FALSE, it should be clear that this attempt will fail. This result leads us to our first observation about these lists:

> If the same sentence occurs on both the left and right lists of sentences, then we shall fail in the attempt to make the sentences in the left list TRUE and those in the right list FALSE. When this occurs, we shall say that this attempt has *failed.*

Let us consider another, more unusual argument:

1. C
2. ~C

∴ 3. D

Using the technique of left and right lists of sentences, we would now write:

$$\begin{array}{c|c} C & D \\ \sim C & \end{array}$$

meaning that we are trying to make C and ~C TRUE and D FALSE. But trying to make ~C TRUE is the same as trying to make C FALSE. A situation in which ~C is TRUE is the same situation as one in which C is FALSE. So, we could transform the lists in the following way. We could move ~C from the left to the right list, removing its outer negation sign as we do so. The result would be:

C	D
	C

But this is a case we have seen before: The same sentence occurs on *both* lists. We have already seen that such an attempt fails. Seeing what occurred to the negation sign, we can make a second observation:

> If a negated sentence occurs on a list, move it to the other list and drop its outer negation sign.

Now consider this argument:

1. (A & B)
∴ 2. B

Transformed into our pair of lists, the argument becomes:

(A & B) | B

But how could '(A & B)', the first sentence on the left list, be made TRUE? It is a conjunction, and there is only one way. The two conjuncts, A and B, must both be TRUE. In other words, the above lists can be transformed into:

A	B
B	

We then see that this attempt, too, fails, because the same sentence occurs on both lists. What we just saw happen to a conjunction leads us to our third observation:

> If a conjunction appears on the *left* list, then remove that conjunction and add its two conjuncts separately to the left list.

There is a similar operation that we may perform on a sentence on the right list:

> If a disjunction appears on the *right* list, then remove that disjunction and add its two disjuncts separately to the right list.

The justification for this operation is not difficult to find. The right list contains the sentences we are trying to make FALSE. But how can we make a *disjunction* FALSE? There is only one way: both of its disjuncts must be made FALSE.

We are accumulating rules for manipulating our pair of lists. We have three rules so far for creating new list-pairs from old ones. They are:

1. If a negated sentence occurs on a list, then move it from that list to the other list and drop its outer negation sign.
2. If a conjunction occurs on the *left* list, then replace that conjunction with its two conjuncts, listed separately.

3. If a disjunction occurs on the *right* list, then replace that disjunction with its two disjuncts, listed separately.

We also have made a generalization that "passes judgment" on a pair of lists:

> If there is a sentence that occurs on both lists, then the attempt has failed.

Before we go further, let's apply these principles to another argument:

1. ~~(A & (B v C))
∴ 2. (A v D)

The first list-pair we form is:

~~(A & (B v C)) | (A v D)

Since '~~(A & (B v C))' on the left list is a negation, we move it to the right side, being careful to drop only its outer negation sign. The result is:

	(A v D)
	~(A & (B v C))

The left list is temporarily empty. But '~(A & (B v C))' on the right list is also a negation, so applying rule 1 again, we have:

(A & (B v C)) | (A v D)

The sentence now on the left list is a conjunction, so we can apply rule 2, which deals with conjunctions on the left list. The result is:

A	(A v D)
(B v C)	

The sentence on the right list, '(A v D)', is a disjunction. According to rule 3, we obtain:

A	A
(B v C)	D

At this point, we should see that one sentence, A, occurs on both lists. Employing our generalization that passes judgment on list-pairs, we see that our attempt to make the sentences on the left list TRUE and on the right list FALSE has failed. Because we began by trying to make the premises of the argument TRUE and the conclusion FALSE, and because making these successive list-pairs is the only way to do so, we can conclude that this argument must be valid. That is, there is no situation where the premises are each TRUE and the conclusion is FALSE.

So far, we have looked only at valid arguments. Consider this argument:

1. (A & ~B)
∴ 2. (B v C)

Our first list-pair is:

(A & ~B) | (B v C)

Because '(A & ~B)' is a conjunction on the left list, we get:

A	(B v C)
~B	

And because '(B v C)' on the right list is a disjunction, we obtain:

A	B
~B	C

Finally, since '~B' on the left list is a negation, applying rule 1 produces:

A	B
	C
	B

No sentence occurs on both lists. Furthermore, all the sentences on both lists are *atomic* sentences.

When all the sentences on each list are atomic, and when no atomic sentence occurs on both lists, we shall say that we have *succeeded* in our attempt to make the sentences on the left list TRUE and on the right FALSE. Whenever this occurs, we know that the original argument is invalid. When each list is composed of only atomic sentences, and when no sentence occurs on both lists, we have found the situation in which the premises are TRUE but the conclusion FALSE. Looking at the last pair of lists for the preceding argument, we note that that situation is:

V(A) = TRUE
V(B) = FALSE
V(C) = FALSE

since A is on the "Make TRUE" list and B and C are on the "Make FALSE" list.

Let us consider another argument:

1. ~A
2. (A v B)
∴ 3. B

The first list-pair based on the argument is:

~A	B
(A v B)	

Applying rule 1, we obtain these lists:

(A v B)	B
	A

But now what do we do? Rule 2 applies only to conjunctions on the left list, and rule 3 applies only to disjunctions on the right list. Here we have a disjunction on the left list.

The occurrence of a sentence on the left list means that we are trying to make it TRUE. In order to make a disjunction TRUE, at least one of the disjuncts must be TRUE. But we have a choice of which one. In the above example, '(A v B)' must be made TRUE, and this can be accomplished in either of two ways: either one disjunct, 'A', is made TRUE, or the other disjunct, 'B', is made TRUE.

In other words, there are at least two ways to make '(A v B)' TRUE. This means that we should now create two new pairs of lists. To do this, we duplicate the original pair and then replace each disjunction with a disjunct—a different one for each duplicate:

```
1.                          (A v B) | B
                                    | A

2(a).  (A v B) | B                               (A v B)    B
               | A                                          A

2(b).      A     B                                   B  |   B
                 A                                      |   A
```

Our attempt to make '(A v B)' TRUE "branches" into two new attempts. One of these new attempts makes '(A v B)' TRUE by making A TRUE, the other by making B TRUE.

But notice that both of these two attempts have a sentence occurring on both lists. On the new left branch, the sentence letter 'A' occurs on both lists. On the new right branch, the sentence letter 'B' occurs on both lists. So *both* of these attempts fail, and the argument is therefore valid.

We now add four new rules for transforming lists:

4. If a disjunction occurs on the *left* list, create two new list-pairs by duplicating the original pair. On one of the new pairs, replace the disjunction with its first disjunct. On the other new pair, replace the disjunction with its second disjunct.
5. If a conjunction occurs on the *right* list, create two new list-pairs by duplicating the original pair. On one of the new pairs, replace the conjunction with its first conjunct. On the other new pair, replace the conjunction with its second conjunct.
6. If a conditional occurs on the *left* list, create two new list-pairs by duplicating the original pair. On one of the new pairs, replace the conditional with the negation of its antecedent. On the other new pair, replace the conditional with its consequent.
7. If a conditional occurs on the *right* list, replace it with two sentences: the negation of its antecedent and also its consequent.

Rule 6 is based on the idea that for a conditional to be made TRUE, either the antecedent must be made FALSE or the consequent made TRUE. On the other hand, if a conditional appears on the *right* list (rule 7), for it to be made FALSE, its antecedent must be made

TRUE and its consequent FALSE. To make its antecedent TRUE, we make the negation of that antecedent FALSE.

Whenever we encounter a rule that requires us to "branch" into two attempts, we shall always continue working on the first attempt. We shall put the second attempt "on hold" and turn to it later. In the terminology of computer science, we place these other list-pairs resulting from branching onto a "stack." WANG'S ALGORITHM is now virtually complete. We need only to add two generalizations for evaluating these lists. These two generalizations are:

If every attempt to make the sentences on the left lists TRUE and those on the right lists FALSE *fails*—including all branched attempts—then the argument is valid.
If a single attempt succeeds, then the argument is invalid.

We have already described what it is for an attempt to "fail" or "succeed." An attempt has failed when a sentence occurs on both lists. An attempt has succeeded when only atomic sentences occur on each list and no sentence occurs on both lists.

We are now in a position to give the full algorithm.

```
WANG'S ALGORITHM

1. INPUT a symbolized argument.
2. Create two lists: Place the premises of the argument on the left list and the
   conclusion on the right list.
3. FOR every sentence on both lists:
     (i)  IF the sentence is a negation
            THEN
            (a) Move it from that list to the other list, and drop its outer negation
                sign.
            (b) TEST the lists. (TEST is a procedure described below.)
     (ii) IF the sentence is a conjunction on the left list
            THEN
            (a) Replace that conjunction with its two conjuncts, listed sepa-
                rately.
            (b) TEST the lists.
     (iii) IF the sentence is a disjunction on the right list
            THEN
            (a) Replace that disjunction with its two disjuncts, listed separately.
            (b) TEST the lists.
     (iv) IF the sentence is a disjunction on the left list
            THEN
```

```
            (a) Create two new list-pairs by duplicating the original pair.
            (b) On one of the new pairs, which will remain the current attempt,
                replace the disjunction with its first disjunct.
            (c) TEST the lists.
            (d) On the other new pair, to be put on the "stack" for later manip-
                ulation, replace the disjunction with its second disjunct.
            (e) TEST the lists.
     (v)  IF the sentence is a conjunction on the right list
            THEN
            (a) Create two new list-pairs by duplicating the original pair.
            (b) On one of the new pairs, which will remain the current attempt,
                replace the conjunction with its first conjunct.
            (c) TEST the lists.
            (d) On the other new pair, to be "stacked," replace the conjunction
                with its second conjunct.
            (e) TEST the lists.
     (vi) IF the sentence is a conditional on the left list
            THEN
            (a) Create two new list-pairs by duplicating the original pair.
            (b) On one of the new pairs, which will remain the current attempt,
                replace the conditional with the negation of its antecedent.
            (c) TEST the lists.
            (d) On the other new pair, to be "stacked," replace the conditional
                with its consequent.
            (e) TEST the lists.
     (vii) IF the sentence is a conditional on the right list
            THEN
            (a) Replace it with two sentences: the negation of its antecedent
                and also its consequent.
            (b) TEST the lists.
4. STOP.
```

The output and the use of the stack are handled by the procedure TEST:

```
PROCEDURE TEST

   1. IF the current attempt has a sentence on both lists
        THEN
        (a) Mark that attempt "failed."
        (b) IF there is a list-pair remaining in the stack
        THEN
        (i) Consider the last such pair that was added to be the current
            attempt.
        (ii) Delete it from the stack.
        (iii) Proceed to the next step in the main procedure.
    (c) IF there is not such a list-pair remaining in the stack
        THEN OUTPUT "Argument valid" and STOP.
2. IF an attempt does not have a sentence occurring on both lists
    THEN
    (a) IF the sentences on both lists are all atomic
        THEN OUTPUT "Argument invalid" and STOP.
    (b) IF the sentences on both lists are not all atomic
        THEN proceed with the next step in the main procedure.
```

Main Connectives

One idea used in step 3 of WANG'S ALGORITHM is the notion of a sentence being a negation, a disjunction, a conjunction, or a conditional. How can this be determined in a mechanical way? A sentence is a conditional if its "main connective" is → (and so on for the other connectives).

The *main connective* of a sentence is the connective that is surrounded by the *fewest* parentheses. This makes the notion of a main connective almost the opposite of the notion of an "innermost subformula" (see Chapter 5).

The following procedure can be used to find what the main connective of a sentence is:

```
PROCEDURE MAIN-CONNECTIVE

1. INPUT a sentence.
2. IF the sentence consists of only one character
    THEN
    (a) The sentence is atomic and has no main connective.
    (b) STOP.
3. IF the first character is ~
    THEN
    (a) The main connective is ~.
    (b) STOP.
4. Let $n = 0$.
```

```
5. FOR every character in the sentence:
    (a) IF that character is '('
        THEN let $n$ be $n + 1$.
    (b) IF that character is ')'
        THEN let $n$ be $n - 1$.
    (c) IF that character is v and $n = 1$
        THEN
        (i) The main connective is v.
        (ii) STOP.
    (d) IF that character is & and $n = 1$
        THEN
        (i) The main connective is &.
        (ii) STOP.
    (e) IF that character is → and $n = 1$
        THEN
        (i) The main connective is →.
        (ii) STOP.
```

Observe that steps 5(a) and 5(b) of this procedure are just the "first counter" of TRUTH-VALUE CALCULATOR from Chapter 5. The "first counter" measures how deeply a character is buried in parentheses. If the sentence is neither atomic nor a negation, then the *least* deeply a character can be buried is 1. So the '$n = 1$' clauses in step 5 tell us when a character is surrounded by the fewest parentheses.

Some Examples

Let's apply WANG'S ALGORITHM to several arguments.

Consider this argument:

1. A
2. (A → (B & C))

∴ 3. C

Applying WANG'S ALGORITHM, after steps 1 and 2, we have:

A (A → (B & C))	C

Note that step 3 does not modify atomic sentences, such as A and C, so the next step to take action is step 3(vi). The result is that we "branch" to two attempts. Here, and in later examples, we duplicate list pairs and replace in one step.

A	C
~A	

A	C
(B & C)	

We continue work on the first pair of lists, applying now step 3(i):

A	C
	A

FAILED

That is, applying TEST, we see that a sentence occurs on both lists. We then return to the other branch, which we had temporarily put aside:

A	C
(B & C)	

Applying step 3(ii):

A	C
B	
C	

FAILED
ARGUMENT VALID

That is, the test of this list-pair shows that the attempt has failed. Since there are no more attempts to consider (that is, no more list-pairs in the stack), the argument is shown to be valid.

Consider another argument:

1. ~A
2. (A → B)

∴ 3. ~B

We first have:

~A	~B
(A → B)	

Then:

	~B
(A → B)	A

But now we branch to:

~A	~B
	A

B	~B
	A

Continuing work on the first branch:

	~B
	A
	A

Then:

B	A
	A

But since only atomic sentences occur on each list, and no sentence occurs on both lists, TEST tells us the argument is invalid.

Summary

This chapter began with a review of truth tables, showing how to construct them for any molecular sentence, no matter how complex. We gave an algorithm for doing this—TRUTH-TABLE GENERATOR (which calls on TRUTH-VALUE CALCULATOR as one of its procedures). We then showed how to use a truth table to determine whether or not an argument is valid. The key ideas here are (1) to generate a truth table whose left-most columns are for the atomic sentences occurring in the argument and whose right-most columns are for the premises and conclusion, and (2) that the rows of this truth table provide the information needed to determine the situations, if any, in which the premises are TRUE and the conclusion FALSE. The algorithm VALIDITY/INVALIDITY DETERMINER (which calls on TRUTH-TABLE GENERATOR as one of *its* procedures) does this. While efficient enough for computers, this latter algorithm can rapidly get out of hand (by growing quite large). WANG'S ALGORITHM is an elegant and more efficient procedure for determining validity.

Exercises

A. For each of the arguments given below:
   (a) State how many rows are required for a truth table for the argument.
   (b) State the truth values in the columns for each of the atomic sentences.

Example:

(A → (B & ~C))
(~B & ~C)
∴ ~A

Answer:

(a) Eight rows are necessary.
[There are three distinct atomic sentences, A, B, and C, and 2 ** 3 = 8.]

(b)

V(A)	V(B)	V(C)	
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

1. ~(A → B)
~B
∴ A

2. (C v D)
(~C & (D → B))
∴ (D & B)

3. (A & ~A)
B
∴ ~B

4. A
B
∴ (C v ~C)

5. (A → (B & C))
(B → D)
A
∴ D

6. (A → B)
(~B v C)
~A
∴ ~C

7. (~D & (B v ~C))
(~D → B)
∴ C

8. (B → ~(C & D))
(C & D)
∴ ~~~B

9. (A → B)
(B → C)
∴ (A → C)

10. (A → B)
∴ (~B → ~A)

11. (A → B)
(A → C)
∴ (B → C)

12. (A v B)
~A
∴ B

B. Examine the following right-hand fragments of truth tables. Answer these questions: (1) Is the argument valid or invalid? (2) If the argument is invalid, which row(s) show it to be so?

1.

Prem. 1	Prem. 2	Conclusion
0	1	1
1	1	1
0	0	0
1	1	0
1	0	0

2.

Prem. 1	Prem. 2	Prem. 3	Conclusion
1	0	1	1
0	0	0	0
1	1	1	1
0	1	1	1

3.

Prem. 1	Prem. 2	Conclusion
0	1	1
1	1	0
0	1	1
0	0	0
1	1	1
0	1	0
1	1	0
0	1	0

4.

Prem. 1	Prem. 2	Prem. 3	Prem. 4	Conclusion
0	1	1	1	0
1	1	1	0	0
1	1	1	1	0

C. Apply TRUTH-TABLE GENERATOR to the arguments given in Exercise A. Indicate whether each argument is valid or invalid.

D. Steps 2 and 3 of TRUTH-TABLE GENERATOR tell us that the size of the truth table needed to test an argument is a function of (1) the number of premises and (2) the number of distinct atomic sentences in the premises and conclusion. Calculate the size of the truth table, measured in rows and columns, needed for arguments that have:
   1. 3 premises, 2 atomic sentences
   2. 2 premises, 3 atomic sentences
   3. 4 premises, 3 atomic sentences
   4. 3 premises, 5 atomic sentences

E. In the algorithm VALIDITY/INVALIDITY DETERMINER, we first filled in *all* the blanks in a truth table (as part of step 2) and only then examined each row to see if it showed the argument to be invalid (step 3). We might, however, consider an alternative that first fills in only the first row of truth values and then immediately examines this row to see if it shows the argument to be invalid. If it does, the procedure outputs "Invalid" and stops. If it doesn't, the procedure goes to the next row and repeats the process.

   What are the advantages and disadvantages to this method?

F. Apply WANG'S ALGORITHM to the following arguments to determine which are valid and which are invalid.

```
1.     (A & (C v D))
       ~C
       (D → E)
     ∴ E
2.     (B → ~C)
       (A v (D & B))
     ∴ (D v A)
3.     (A & C)
       (D → A)
       (D → E)
     ∴ (D & E)
4.     ~(A → ~B)
       ~(C v E)
     ∴ (C → A)
5.     (~A & (C → D))
       (D → A)
       ((F & E) → C)
     ∴ (~F v ~E)
```

G. Determine a formula for calculating how large the left and right lists might become for a given argument. (*Hint:* Consider the number of atomic sentences.)

H. Determine a formula for calculating the maximum number of branches an application of WANG'S ALGORITHM might take.

I. Write a program to apply WANG'S ALGORITHM to a list-pair that contains & and ~ as main connectives on the left list and →, ~, and v on the right list. (You will not have to consider branching and the resulting stacks.)

J. Write a program that applies WANG'S ALGORITHM in its entirety to an input argument and prints out all the list-pairs, such that every branch of this "tree" ultimately ends with either "ATTEMPT FAILS" or "ATTEMPT SUCCEEDS, ARGUMENT INVALID."

Suggestions for Computer Implementation

The design of a program to implement WANG'S ALGORITHM requires the following considerations. First, something must be created to store the main list-pair ("the current attempt"). Provisions must also be made for the list-pairs that are put "on hold"—that is, for the stack. Second, a procedure, or function, must be created to input a sentence and determine whether the sentence is atomic, a negation, a conditional, a conjunction, or a disjunction. We might call this procedure MAIN-CONNECTIVE, since its principal task is to determine the main connective in a sentence. And third, a procedure must be designed to test the list-pairs: the procedure TEST.

In most programming languages, the list-pairs would be stored in string arrays. (In languages that accept genuine lists, such as LISP and LOGO, the list-pairs can be stored as lists.) LLIST and RLIST would be convenient names for the left and right lists currently under consideration. It might also prove convenient to have a variable that remembers how many sentences are stored in LLIST and RLIST at any given time: we might call these two variables LCOUNT and RCOUNT.

The procedure MAIN-CONNECTIVE inputs a string and might output:

A	If the input sentence is atomic.
~	If a negation.
v	If a disjunction.
&	If a conjunction.
>	If a conditional.

For the manipulation of strings, we would also need to construct some functions or procedures to perform the following tasks:

1. Remove the first character (usually a negation sign) from a string. We can call this 'NEGOFF' for "NEGation OFF."
2. Generate from an input sentence the sentence to the left of the main connective. We might call this the 'LEFTSEN' function.
3. Generate from an input sentence the sentence to the right of the main connective. We might call this the 'RIGHTSEN' function.

Here are some examples of the way these three functions work:

```
NEGOFF [~~(A & B)]              = ~(A & B)
LEFTSEN [((A & B) v (C v D))]   = (A & B)

RIGHTSEN [((A & B) v (C v D))]  = (C v D)
```

Using these procedures, and using our program design language in a very detailed way, the program might begin:

```
1. INPUT LLIST [the premises].
2. INPUT RLIST [the conclusion].
3. Let LCOUNT be the number of sentences on LLIST.
4. Let RCOUNT be 1 [since only the conclusion is initially on RLIST].
5. FOR every sentence on LLIST, from I = 1 to LCOUNT:
   (a) IF MAINCONNECTIVE(LLIST(I)) = ~
         THEN
          (i) Let RLIST(RCOUNT + 1) be NEGOFF(LLIST(I)).
         (ii) Let LLIST(I) be " " [a space, which serves as a placeholder].
        (iii) Let RCOUNT be RCOUNT + 1.
         (iv) TEST the list-pair.
   (b) IF MAINCONNECTIVE(LLIST(I)) = &
         THEN
          (i) Let LLIST(LCOUNT + 1) be LEFTSEN(LLIST(I)).
         (ii) Let LLIST(I) be RIGHTSEN(LLIST(I)).
        (iii) Let LCOUNT be LCOUNT + 1.
         (iv) TEST the list-pair.
```

and so on for the RLIST as well.

The branching that results from disjunctions or conditionals on LLIST and conjunctions on RLIST requires some special considerations. One list-pair at a time is put on a stack. Two stacks are needed; we can call them 'LSTACK' and 'RSTACK'. LSTACK will store the stacked left lists, and RSTACK will store the stacked right lists. It is extremely handy if both LSTACK and RSTACK are two-dimensional arrays whose first index indicates *when* a list was stacked (first, second, and so on) and whose second index indicates the sentence on the list. For example, LSTACK (2,3) would refer to the third sentence on the second left list that was previously stacked. We shall put the lists onto the stacks in the order of when we branch, and we shall take them off and make them the "main attempt" in reverse order. That is, we shall first take the last list-pair that was stacked. This is called the "last in, first out" method. A good reason for using stacks rather than, say, a "first in, first out" organization (a queue) is that the last list-pair put on the stack is probably more highly "digested"—that is, broken into simpler strings—than the first and so might allow us to reach a "failed" attempt with minimal manipulation. It is also useful to have counters to keep track of how many sentences are in each "level" of each stack.

One advantage of WANG'S ALGORITHM is that its implementation does not require the sometimes huge arrays that VALIDITY/INVALIDITY DETERMINER does. If there are not too many premises (no more than, say, 6) and not too many atomic sentences (no more than, say, 10), then the arrays LLIST and RLIST need not be larger than 10, or LSTACK and RSTACK larger than 10 × 10.