Chapter 2
Propositional Logic: Formulas, Models, Tableaux

Propositional logic is a simple logical system that is the basis for all others. Propositions are claims like 'one plus one equals two' and 'one plus two equals two' that cannot be further decomposed and that can be assigned a truth value of *true* or *false*. From these *atomic propositions*, we will build complex *formulas* using *Boolean operators*:

'one plus one equals two' *and* 'Earth is farther from the sun than Venus'.

Logical systems formalize reasoning and are similar to programming languages that formalize computations. In both cases, we need to define the syntax and the semantics. The *syntax* defines what strings of symbols constitute legal formulas (legal programs, in the case of languages), while the *semantics* defines what legal formulas mean (what legal programs compute). Once the syntax and semantics of propositional logic have been defined, we will show how to construct *semantic tableaux*, which provide an efficient *decision procedure* for checking when a formula is true.

2.1 Propositional Formulas

In computer science, an *expression* denoted the computation of a value from other values; for example, $2 * 9 + 5$. In propositional logic, the term *formula* is used instead. The formal definition will be in terms of trees, because our the main proof technique called structural induction is easy to understand when applied to trees. Optional subsections will expand on different approaches to syntax.

M. Ben-Ari, *Mathematical Logic for Computer Science*,
DOI 10.1007/978-1-4471-4129-7_2, © Springer-Verlag London 2012

2.1.1 Formulas as Trees

Definition 2.1 The symbols used to construct formulas in propositional logic are:

- An unbounded set of symbols $\mathcal{P}$ called *atomic propositions* (often shortened to *atoms*). Atoms will be denoted by lower case letters in the set $\{p, q, r, \ldots\}$, possibly with subscripts.
- *Boolean operators*. Their names and the symbols used to denote them are:

negation	$\neg$
disjunction	$\vee$
conjunction	$\wedge$
implication	$\rightarrow$
equivalence	$\leftrightarrow$
exclusive or	$\oplus$
nor	$\downarrow$
nand	$\uparrow$

The negation operator is a *unary operator* that takes one operand, while the other operators are *binary operators* taking two operands. ■

Definition 2.2 A *formula* in propositional logic is a tree defined recursively:

- A formula is a leaf labeled by an atomic proposition.
- A formula is a node labeled by $\neg$ with a single child that is a formula.
- A formula is a node labeled by one of the binary operators with two children both of which are formulas. ■

Example 2.3 Figure 2.1 shows two formulas. ■

2.1.2 Formulas as Strings

Just as we write expressions as strings (linear sequences of symbols), we can write formulas as strings. The string associated with a formula is obtained by an *inorder traversal* of the tree:

Algorithm 2.4 (Represent a formula by a string)
Input: A formula A of propositional logic.
Output: A string representation of A.

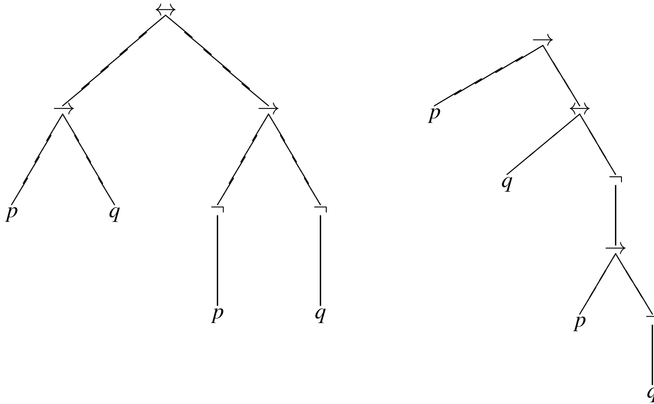


Fig. 2.1 Two formulas

Call the recursive procedure `Inorder(A)`:

```
Inorder(F)
  if F is a leaf
    write its label
    return
  let F1 and F2 be the left and right subtrees of F
  Inorder(F1)
  write the label of the root of F
  Inorder(F2)
```

If the root of `F` is labeled by negation, the left subtree is considered to be empty and the step `Inorder(F1)` is skipped. ∎

Definition 2.5 The term *formula* will also be used for the string with the understanding that it refers to the underlying tree. ∎

Example 2.6 Consider the left formula in Fig. 2.1. The inorder traversal gives: write the leftmost leaf labeled p, followed by its root labeled $\rightarrow$, followed by the right leaf of the implication labeled q, followed by the root of the tree labeled $\leftrightarrow$, and so on. The result is the string:

$$p \rightarrow q \leftrightarrow \neg\, p \rightarrow \neg\, q.$$

Consider now the right formula in Fig. 2.1. Performing the traversal results in the string:

$$p \rightarrow q \leftrightarrow \neg\, p \rightarrow \neg\, q,$$

which is precisely the same as that associated with the left formula. ∎

Although the formulas are *not* ambiguous—the trees have entirely different structures—their representations as strings are ambiguous. Since we prefer to deal with strings, we need some way to resolve such ambiguities. There are three ways of doing this.

2.1.3 Resolving Ambiguity in the String Representation

Parentheses

The simplest way to avoid ambiguity is to use parentheses to maintain the structure of the tree when the string is constructed.

Algorithm 2.7 (Represent a formula by a string with parentheses)
Input: A formula A of propositional logic.
Output: A string representation of A.
Call the recursive procedure `Inorder(A)`:

```
Inorder(F)
  if F is a leaf
    write its label
    return
  let F1 and F2 be the left and right subtrees of F
  write a left parenthesis '('
  Inorder(F1)
  write the label of the root of F
  Inorder(F2)
  write a right parenthesis ')'
```

If the root of `F` is labeled by negation, the left subtree is considered to be empty and the step `Inorder(F1)` is skipped. ∎

The two formulas in Fig. 2.1 are now associated with two different strings and there is no ambiguity:

$$((p \rightarrow q) \leftrightarrow ((\neg\, q) \rightarrow (\neg\, p))),$$

$$(p \rightarrow (q \leftrightarrow (\neg\, (p \rightarrow (\neg\, q))))).$$

The problem with parentheses is that they make formulas verbose and hard to read and write.

Precedence

The second way of resolving ambiguous formulas is to define *precedence* and *associativity* conventions among the operators as is done in arithmetic, so that we

immediately recognize $a*b*c+d*e$ as $(((a*b)*c)+(d*e))$. For formulas the order of precedence from high to low is as follows:

$$\begin{array}{l}\neg\\ \wedge, \uparrow\\ \vee, \downarrow\\ \rightarrow\\ \leftrightarrow, \oplus\end{array}$$

Operators are assumed to associate to the right, that is, $a \vee b \vee c$ means $(a \vee (b \vee c))$.

Parentheses are used only if needed to indicate an order different from that imposed by the rules for precedence and associativity, as in arithmetic where $a*(b+c)$ needs parentheses to denote that the addition is done before the multiplication. With minimal use of parentheses, the formulas above can be written:

$$p \rightarrow q \leftrightarrow \neg\, q \rightarrow \neg\, p,$$

$$p \rightarrow (q \leftrightarrow \neg\,(p \rightarrow \neg\, q)).$$

Additional parentheses may always be used to clarify a formula: $(p \vee q) \wedge (q \vee r)$.

The Boolean operators $\wedge$, $\vee$, $\leftrightarrow$, $\oplus$ are associative so we will often omit parentheses in formulas that have repeated occurrences of these operators: $p \vee q \vee r \vee s$. Note that $\rightarrow$, $\downarrow$, $\uparrow$ are *not* associative, so parentheses must be used to avoid confusion. Although the implication operator is assumed to be right associative, so that $p \rightarrow q \rightarrow r$ unambiguously means $p \rightarrow (q \rightarrow r)$, we will write the formula with parentheses to avoid confusion with $(p \rightarrow q) \rightarrow r$.

Polish Notation *

There will be no ambiguity if the string representing a formula is created by a *preorder traversal* of the tree:

Algorithm 2.8 (Represent a formula by a string in Polish notation)
Input: A formula A of propositional logic.
Output: A string representation of A.

Call the recursive procedure `Preorder(A)`:

```
Preorder(F)
  write the label of the root of F
  if F is a leaf
    return
  let F1 and F2 be the left and right subtrees of F
  Preorder(F1)
  Preorder(F2)
```

If the root of `F` is labeled by negation, the left subtree is considered to be empty and the step `Preorder(F1)` is skipped. ■

Example 2.9 The strings associated with the two formulas in Fig. 2.1 are:

$$\leftrightarrow\ \rightarrow p\, q \rightarrow \neg\, p \neg\, q,$$

$$\rightarrow p \leftrightarrow q \neg\ \rightarrow\ p \neg\, q$$

and there is no longer any ambiguity. ■

The formulas are said to be in *Polish notation*, named after a group of Polish logicians led by Jan Łukasiewicz.

We find infix notation easier to read because it is familiar from arithmetic, so Polish notation is normally used only in the internal representation of arithmetic and logical expressions in a computer. The advantage of Polish notation is that the expression can be evaluated in the linear order that the symbols appear using a stack. If we rewrite the first formula backwards (*reverse Polish notation*):

$$q \neg\, p \neg\ \rightarrow\ q p \rightarrow\ \leftrightarrow,$$

it can be directly compiled to the following sequence of instructions of an assembly language:

```
Push q
Negate
Push p
Negate
Imply
Push q
Push p
Imply
Equiv
```

The operators are applied to the top operands on the stack which are then popped and the result pushed.

2.1.4 Structural Induction

Given an arithmetic expression like $a * b + b * c$, it is immediately clear that the expression is composed of two terms that are added together. In turn, each term is composed of two factors that are multiplied together. In the same way, any propositional formula can be classified by its top-level operator.

Definition 2.10 Let $A \in \mathcal{F}$. If A is not an atom, the operator labeling the root of the formula A is the *principal operator* of the A. ■

Example 2.11 The principal operator of the left formula in Fig. 2.1 is $\leftrightarrow$, while the principal operator of the right formulas is $\rightarrow$. ■

Structural induction is used to prove that a property holds for *all* formulas. This form of induction is similar to the familiar numerical induction that is used to prove that a property holds for all natural numbers (Appendix A.6). In numerical induction, the *base case* is to prove the property for 0 and then to prove the *inductive step*: assume that the property holds for arbitrary n and then show that it holds for $n+1$. By Definition 2.10, a formula is either a leaf labeled by an atom or it is a tree with a principal operator and one or two subtrees. The base case of structural induction is to prove the property for a leaf and the inductive step is to prove the property for the formula obtained by applying the principal operator to the subtrees, assuming that the property holds for the subtrees.

Theorem 2.12 (Structural induction) *To show that a property holds for all formulas* $A \in \mathscr{F}$:

1. *Prove that the property holds all atoms* p.
2. *Assume that the property holds for a formula* A *and prove that the property holds for* $\neg A$.
3. *Assume that the property holds for formulas* A_1 *and* A_2 *and prove that the property holds for* A_1 *op* A_2, *for each of the binary operators.*

Proof Let A be an arbitrary formula and suppose that (1), (2), (3) have been shown for some property. We show that the property holds for A by numerical induction on n, the height of the tree for A. For $n = 0$, the tree is a leaf and A is an atom p, so the property holds by (1). Let $n > 0$. The subtrees A are of height $n - 1$, so by numerical induction, the property holds for these formulas. The principal operator of A is either negation or one of the binary operators, so by (2) or (3), the property holds for A. ■

We will later show that all the binary operators can be defined in terms negation and either disjunction or conjunction, so a proof that a property holds for all formulas can be done using structural induction with the base case and only two inductive steps.

2.1.5 Notation

Unfortunately, books on mathematical logic use widely varying notation for the Boolean operators; furthermore, the operators appear in programming languages with a different notation from that used in mathematics textbooks. The following table shows some of these alternate notations.

Operator	Alternates	Java language
$\neg$	$\sim$	`!`
$\wedge$	&	`&, &&`
$\vee$		`|, ||`
$\rightarrow$	$\supset, \Rightarrow$	
$\leftrightarrow$	$\equiv, \Leftrightarrow$	
$\oplus$	$\not\equiv$	`^`
$\uparrow$	$\mid$	

2.1.6 A Formal Grammar for Formulas *

This subsection assumes familiarity with formal grammars.

Instead of defining formulas as trees, they can be defined as strings generated by a context-free formal grammar.

Definition 2.13 Formula in propositional logic are derived from the context-free grammar whose terminals are:

- An unbounded set of symbols $\mathcal{P}$ called *atomic propositions*.
- The *Boolean operators* given in Definition 2.1.

The productions of the grammar are:

$$\begin{array}{lcll} fml & ::= & p & \text{for any } p \in \mathcal{P} \\ fml & ::= & \neg\, fml & \\ fml & ::= & fml\ op\ fml & \\ op & ::= & \vee \mid \wedge \mid \rightarrow \mid \leftrightarrow \mid \oplus \mid \uparrow \mid \downarrow & \end{array}$$

A formula is a word that can be derived from the nonterminal *fml*. The set of all formulas that can be derived from the grammar is denoted $\mathcal{F}$. ■

Derivations of strings (words) in a formal grammar can be represented as trees (Hopcroft et al., 2006, Sect. 4.3). The word generated by a derivation can be read off the leaves from left to right.

Example 2.14 Here is a derivation of the formula $p \rightarrow q \leftrightarrow \neg\, p \rightarrow \neg\, q$ in propositional logic; the tree representing its derivation is shown in Fig. 2.2.

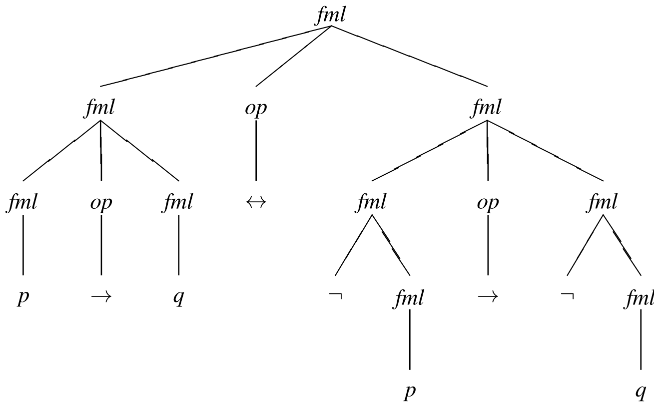


Fig. 2.2 Derivation tree for $p \rightarrow q \leftrightarrow \neg\, p \rightarrow \neg\, q$

1. fml
2. $\mathit{fml}\ \mathit{op}\ \mathit{fml}$
3. $\mathit{fml} \leftrightarrow \mathit{fml}$
4. $\mathit{fml}\ \mathit{op}\ \mathit{fml} \leftrightarrow \mathit{fml}$
5. $\mathit{fml} \rightarrow \mathit{fml} \leftrightarrow \mathit{fml}$
6. $p \rightarrow \mathit{fml} \leftrightarrow \mathit{fml}$
7. $p \rightarrow q \leftrightarrow \mathit{fml}$
8. $p \rightarrow q \leftrightarrow \mathit{fml}\ \mathit{op}\ \mathit{fml}$
9. $p \rightarrow q \leftrightarrow \mathit{fml} \rightarrow \mathit{fml}$
10. $p \rightarrow q \leftrightarrow \neg\, \mathit{fml} \rightarrow \mathit{fml}$
11. $p \rightarrow q \leftrightarrow \neg\, p \rightarrow \mathit{fml}$
12. $p \rightarrow q \leftrightarrow \neg\, p \rightarrow \neg\, \mathit{fml}$
13. $p \rightarrow q \leftrightarrow \neg\, p \rightarrow \neg\, q$

The methods discussed in Sect. 2.1.2 can be used to resolve ambiguity. We can change the grammar to introduce parentheses:

$$\begin{array}{lcl} \mathit{fml} & ::= & (\neg\, \mathit{fml}) \\ \mathit{fml} & ::= & (\mathit{fml}\ \ \mathit{op}\ \ \mathit{fml}) \end{array}$$

and then use precedence to reduce their number.

$v_{\mathscr{I}}(A) = \mathscr{I}_A(A)$	if A is an atom
$v_{\mathscr{I}}(\neg A) = T$	if $v_{\mathscr{I}}(A) = F$
$v_{\mathscr{I}}(\neg A) = F$	if $v_{\mathscr{I}}(A) = T$
$v_{\mathscr{I}}(A_1 \vee A_2) = F$	if $v_{\mathscr{I}}(A_1) = F$ and $v_{\mathscr{I}}(A_2) = F$
$v_{\mathscr{I}}(A_1 \vee A_2) = T$	otherwise
$v_{\mathscr{I}}(A_1 \wedge A_2) = T$	if $v_{\mathscr{I}}(A_1) = T$ and $v_{\mathscr{I}}(A_2) = T$
$v_{\mathscr{I}}(A_1 \wedge A_2) = F$	otherwise
$v_{\mathscr{I}}(A_1 \rightarrow A_2) = F$	if $v_{\mathscr{I}}(A_1) = T$ and $v_{\mathscr{I}}(A_2) = F$
$v_{\mathscr{I}}(A_1 \rightarrow A_2) = T$	otherwise
$v_{\mathscr{I}}(A_1 \uparrow A_2) = F$	if $v_{\mathscr{I}}(A_1) = T$ and $v_{\mathscr{I}}(A_2) = T$
$v_{\mathscr{I}}(A_1 \uparrow A_2) = T$	otherwise
$v_{\mathscr{I}}(A_1 \downarrow A_2) = T$	if $v_{\mathscr{I}}(A_1) = F$ and $v_{\mathscr{I}}(A_2) = F$
$v_{\mathscr{I}}(A_1 \downarrow A_2) = F$	otherwise
$v_{\mathscr{I}}(A_1 \leftrightarrow A_2) = T$	if $v_{\mathscr{I}}(A_1) = v_{\mathscr{I}}(A_2)$
$v_{\mathscr{I}}(A_1 \leftrightarrow A_2) = F$	if $v_{\mathscr{I}}(A_1) \neq v_{\mathscr{I}}(A_2)$
$v_{\mathscr{I}}(A_1 \oplus A_2) = T$	if $v_{\mathscr{I}}(A_1) \neq v_{\mathscr{I}}(A_2)$
$v_{\mathscr{I}}(A_1 \oplus A_2) = F$	if $v_{\mathscr{I}}(A_1) = v_{\mathscr{I}}(A_2)$

Fig. 2.3 Truth values of formulas

2.2 Interpretations

We now define the semantics—the meaning—of formulas. Consider again arithmetic expressions. Given an expression E such as $a * b + 2$, we can *assign* values to a and b and then *evaluate* the expression. For example, if $a = 2$ and $b = 3$ then E evaluates to 8. In propositional logic, truth values are assigned to the atoms of a formula in order to evaluate the truth value of the formula.

2.2.1 The Definition of an Interpretation

Definition 2.15 Let $A \in \mathscr{F}$ be a formula and let $\mathscr{P}_A$ be the set of atoms appearing in A. An *interpretation* for A is a total function $\mathscr{I}_A : \mathscr{P}_A \mapsto \{T, F\}$ that assigns one of the *truth values* T or F to *every* atom in $\mathscr{P}_A$. ■

Definition 2.16 Let $\mathscr{I}_A$ be an interpretation for $A \in \mathscr{F}$. $v_{\mathscr{I}_A}(A)$, the *truth value of A under $\mathscr{I}_A$* is defined inductively on the structure of A as shown in Fig. 2.3. ■

In Fig. 2.3, we have abbreviated $v_{\mathscr{I}_A}(A)$ by $v_{\mathscr{I}}(A)$. The abbreviation $\mathscr{I}$ for $\mathscr{I}_A$ will be used whenever the formula is clear from the context.

Example 2.17 Let $A = (p \rightarrow q) \leftrightarrow (\neg q \rightarrow \neg p)$ and let $\mathscr{I}_A$ be the interpretation:

$$\mathscr{I}_A(p) = F, \qquad \mathscr{I}_A(q) = T.$$

The truth value of A can be evaluated inductively using Fig. 2.3:

$$\begin{array}{lcl}
v_{\mathscr{I}}(p) & = & \mathscr{I}_A(p) = F \\
v_{\mathscr{I}}(q) & = & \mathscr{I}_A(q) = T \\
v_{\mathscr{I}}(p \rightarrow q) & = & T \\
v_{\mathscr{I}}(\neg q) & = & F \\
v_{\mathscr{I}}(\neg p) & = & T \\
v_{\mathscr{I}}(\neg q \rightarrow \neg p) & = & T \\
v_{\mathscr{I}}((p \rightarrow q) \leftrightarrow (\neg q \rightarrow \neg p)) & = & T.
\end{array}$$

■

Partial Interpretations *

We will later need the following definition, but you can skip it for now:

Definition 2.18 Let $A \in \mathscr{F}$. A *partial interpretation* for A is a partial function $\mathscr{I}_A : \mathscr{P}_A \mapsto \{T, F\}$ that assigns one of the *truth values* T or F to *some* of the atoms in $\mathscr{P}_A$. ■

It is possible that the truth value of a formula can be determined in a partial interpretation.

Example 2.19 Consider the formula $A = p \wedge q$ and the partial interpretation that assigns F to p. Clearly, the truth value of A is F. If the partial interpretation assigned T to p, we cannot compute the truth value of A. ■

2.2.2 Truth Tables

A truth table is a convenient format for displaying the semantics of a formula by showing its truth value for every possible interpretation of the formula.

Definition 2.20 Let $A \in \mathscr{F}$ and supposed that there are n atoms in $\mathscr{P}_A$. A *truth table* is a table with $n + 1$ columns and 2^n rows. There is a column for each atom in $\mathscr{P}_A$, plus a column for the formula A. The first n columns specify the interpretation $\mathscr{I}$ that maps atoms in $\mathscr{P}_A$ to $\{T, F\}$. The last column shows $v_{\mathscr{I}}(A)$, the truth value of A for the interpretation $\mathscr{I}$. ■

Since each of the n atoms can be assigned T or F independently, there are 2^n interpretations and thus 2^n rows in a truth table.

Example 2.21 Here is the truth table for the formula $p \rightarrow q$:

p	q	$p \rightarrow q$
T	T	T
T	F	F
F	T	T
F	F	T

■

When the formula A is complex, it is easier to build a truth table by adding columns that show the truth value for subformulas of A.

Example 2.22 Here is a truth table for the formula $(p \rightarrow q) \leftrightarrow (\neg q \rightarrow \neg p)$ from Example 2.17:

p	q	$p \rightarrow q$	$\neg p$	$\neg q$	$\neg q \rightarrow \neg p$	$(p \rightarrow q) \leftrightarrow (\neg q \rightarrow \neg p)$
T	T	T	F	F	T	T
T	F	F	F	T	F	T
F	T	T	T	F	T	T
F	F	T	T	T	T	T

■

A convenient way of computing the truth value of a formula for a specific interpretation $\mathscr{I}$ is to write the value T or F of $\mathscr{I}(p_i)$ under each atom p_i and then to write down the truth values incrementally under each operator as you perform the computation. Each step of the computation consists of choosing an innermost subformula and evaluating it.

Example 2.23 The computation of the truth value of $(p \rightarrow q) \leftrightarrow (\neg q \rightarrow \neg p)$ for the interpretation $\mathscr{I}(p) = T$ and $\mathscr{I}(q) = F$ is:

$(p$	$\rightarrow$	$q)$	$\leftrightarrow$	$(\neg$	q	$\rightarrow$	$\neg$	$p)$
T		F			F			T
T		F		T	F			T
T		F		T	F		F	T
T		F		T	F	F	F	T
T	F	F		T	F	F	F	T
T	F	F	T	T	F	F	F	T

If the computations for all subformulas are written on the same line, the truth table from Example 2.22 can be written as follows:

p	q	$(p$	$\rightarrow$	$q)$	$\leftrightarrow$	$(\neg$	q	$\rightarrow$	$\neg$	$p)$
T	T	T	T	T	T	F	T	T	F	T
T	F	T	F	F	T	T	F	F	F	T
F	T	F	T	T	T	F	T	T	T	F
F	F	F	T	F	T	T	F	T	T	F

2.2.3 *Understanding the Boolean Operators*

The natural reading of the Boolean operators $\neg$ and $\wedge$ correspond with their formal semantics as defined in Fig. 2.3. The operators $\uparrow$ and $\downarrow$ are simply negations of $\wedge$ and $\vee$. Here we comment on the operators $\vee$, $\oplus$ and $\rightarrow$, whose formal semantics can be the source of confusion.

Inclusive or *vs.* Exclusive or

Disjunction $\vee$ is *inclusive or* and is a distinct operator from $\oplus$ which is *exclusive or*. Consider the compound statement:

At eight o'clock 'I will go to the movies' *or* 'I will go to the theater'.

The intended meaning is 'movies' $\oplus$ 'theater', because I can't be in both places at the same time. This contrasts with the disjunctive operator $\vee$ which evaluates to true when either or both of the statements are true:

Do you want 'popcorn' or 'candy'?

This can be denoted by 'popcorn' $\vee$ 'candy', because it is possible to want both of them at the same time.

For $\vee$, it is sufficient for one statement to be true for the compound statement to be true. Thus, the following strange statement is true because the truth of the first statement by itself is sufficient to ensure the truth of the compound statement:

'Earth is farther from the sun than Venus' $\vee$ '$1+1=3$'.

The difference between $\vee$ and $\oplus$ is seen when both subformulas are true:

'Earth is farther from the sun than Venus' $\vee$ '$1+1=2$'.
'Earth is farther from the sun than Venus' $\oplus$ '$1+1=2$'.

The first statement is true but the second is false.

Inclusive or *vs.* Exclusive or in Programming Languages

When *or* is used in the context of programming languages, the intention is usually inclusive or:

```
if (index < min || index > max) /* There is an error */
```

The truth of one of the two subexpressions causes the following statements to be executed. The operator `||` is not really a Boolean operator because it uses *short-circuit evaluation*: if the first subexpression is true, the second subexpression is not evaluated, because its truth value cannot change the decision to execute the following statements. There is an operator `|` that performs true Boolean evaluation; it is usually used when the operands are bit vectors:

```
mask1 = 0xA0;
mask2 = 0x0A;
mask  = mask1 | mask2;
```

Exclusive or `^` is used to implement encoding and decoding in error-correction and cryptography. The reason is that when used twice, the original value can be recovered. Suppose that we encode bit of data with a secret key:

```
codedMessage = data ^ key;
```

The recipient of the message can decode it by computing:

```
clearMessage = codedMessage ^ key;
```

as shown by the following computation:

```
clearMessage == codedMessage ^ key
             == (data ^ key) ^ key
             == data ^ (key ^ key)
             == data ^ false
             == data
```

Implication

The operator of $p \rightarrow q$ is called *material implication*; p is the *antecedent* and q is the *consequent*. Material implication does not claim causation; that is, it does not assert there the antecedent *causes* the consequent (or is even related to the consequent in any way). A material implication merely states that if the antecedent is true the consequent must be true (see Fig. 2.3), so it can be falsified only if the antecedent is true and the consequent is false. Consider the following two compound statements:

‘Earth is farther from the sun than Venus’ $\rightarrow$ ‘$1+1=3$’.

is false since the antecedent is true and the consequent is false, but:

'Earth is farther from the sun than Mars' $\rightarrow$ '$1+1=3$'.

is true! The falsity of the antecedent by itself is sufficient to ensure the truth of the implication.

2.2.4 *An Interpretation for a Set of Formulas*

Definition 2.24 Let $S = \{A_1, \ldots\}$ be a set of formulas and let $\mathscr{P}_S = \bigcup_i \mathscr{P}_{A_i}$, that is, $\mathscr{P}_S$ is the set of all the atoms that appear in the formulas of S. An *interpretation* for S is a function $\mathscr{I}_S : \mathscr{P}_S \mapsto \{T, F\}$. For any $A_i \in S$, $v_{\mathscr{I}_S}(A_i)$, the *truth value of* A_i *under* $\mathscr{I}_S$, is defined as in Definition 2.16. ■

The definition of $\mathscr{P}_S$ as the union of the sets of atoms in the formulas of S ensures that each atom is assigned exactly one truth value.

Example 2.25 Let $S = \{p \rightarrow q,\ p,\ q \wedge r,\ p \vee s \leftrightarrow s \wedge q\}$ and let $\mathscr{I}_S$ be the interpretation:

$$\mathscr{I}_S(p) = T, \qquad \mathscr{I}_S(q) = F, \qquad \mathscr{I}_S(r) = T, \qquad \mathscr{I}_S(s) = T.$$

The truth values of the elements of S can be evaluated as:

$$\begin{aligned}
v_{\mathscr{I}}(p \rightarrow q) &= F \\
v_{\mathscr{I}}(p) &= \mathscr{I}_S(p) = T \\
v_{\mathscr{I}}(q \wedge r) &= F \\
v_{\mathscr{I}}(p \vee s) &= T \\
v_{\mathscr{I}}(s \wedge q) &= F \\
v_{\mathscr{I}}(p \vee s \leftrightarrow s \wedge q) &= F.
\end{aligned}$$

■

2.3 Logical Equivalence

Definition 2.26 Let $A_1, A_2 \in \mathscr{F}$. If $v_{\mathscr{I}}(A_1) = v_{\mathscr{I}}(A_2)$ for all interpretations $\mathscr{I}$, then A_1 is *logically equivalent* to A_2, denoted $A_1 \equiv A_2$. ■

Example 2.27 Is the formula $p \vee q$ logically equivalent to $q \vee p$? There are four distinct interpretations that assign to the atoms p and q:

$\mathscr{I}(p)$	$\mathscr{I}(q)$	$v_{\mathscr{I}}(p \vee q)$	$v_{\mathscr{I}}(q \vee p)$
T	T	T	T
T	F	T	T
F	T	T	T
F	F	F	F

Since $p \vee q$ and $q \vee p$ agree on all the interpretations, $p \vee q \equiv q \vee p$. ∎

This example can be generalized to arbitrary *formulas*:

Theorem 2.28 *Let $A_1, A_2 \in \mathscr{F}$. Then $A_1 \vee A_2 \equiv A_2 \vee A_1$.*

Proof Let $\mathscr{I}$ be an arbitrary interpretation for $A_1 \vee A_2$. Obviously, $\mathscr{I}$ is also an interpretation for $A_2 \vee A_1$ since $\mathscr{P}_{A_1} \cup \mathscr{P}_{A_2} = \mathscr{P}_{A_2} \cup \mathscr{P}_{A_1}$.

Since $\mathscr{P}_{A_1} \subseteq \mathscr{P}_{A_1} \cup \mathscr{P}_{A_2}$, $\mathscr{I}$ assigns truth values to all atoms in A_1 and can be considered to be an interpretation for A_1. Similarly, $\mathscr{I}$ can be considered to be an interpretation for A_2.

Now $v_{\mathscr{I}}(A_1 \vee A_2) = T$ if and only if either $v_{\mathscr{I}}(A_1) = T$ or $v_{\mathscr{I}}(A_2) = T$, and $v_{\mathscr{I}}(A_2 \vee A_1) = T$ if and only if either $v_{\mathscr{I}}(A_2) = T$ or $v_{\mathscr{I}}(A_1) = T$. If $v_{\mathscr{I}}(A_1) = T$, then:

$$v_{\mathscr{I}}(A_1 \vee A_2) = T = v_{\mathscr{I}}(A_2 \vee A_1),$$

and similarly if $v_{\mathscr{I}}(A_2) = T$. Since $\mathscr{I}$ was arbitrary, $A_1 \vee A_2 \equiv A_2 \vee A_1$. ∎

This type of argument will be used frequently. In order to prove that something is true of *all* interpretations, we let $\mathscr{I}$ be an *arbitrary* interpretation and then write a proof without using any property that distinguishes one interpretation from another.

2.3.1 The Relationship Between $\leftrightarrow$ and $\equiv$

Equivalence, $\leftrightarrow$, is a Boolean operator in propositional logic and can appear in formulas of the logic. Logical equivalence, $\equiv$, is not a Boolean operator; instead, is a notation for a *property* of pairs of formulas in propositional logic. There is potential for confusion because we are using a similar vocabulary both for the *object language*, in this case the language of propositional logic, and for the *metalanguage* that we use reason about the object language.

Equivalence and logical equivalence are, nevertheless, closely related as shown by the following theorem:

Theorem 2.29 *$A_1 \equiv A_2$ if and only if $A_1 \leftrightarrow A_2$ is true in every interpretation.*

Proof Suppose that $A_1 \equiv A_2$ and let $\mathscr{I}$ be an arbitrary interpretation; then $v_{\mathscr{I}}(A_1) = v_{\mathscr{I}}(A_2)$ by definition of logical equivalence. From Fig. 2.3, $v_{\mathscr{I}}(A_1 \leftrightarrow A_2) = T$. Since $\mathscr{I}$ was arbitrary, $v_{\mathscr{I}}(A_1 \leftrightarrow A_2) = T$ in all interpretations. The proof of the converse is similar. ∎

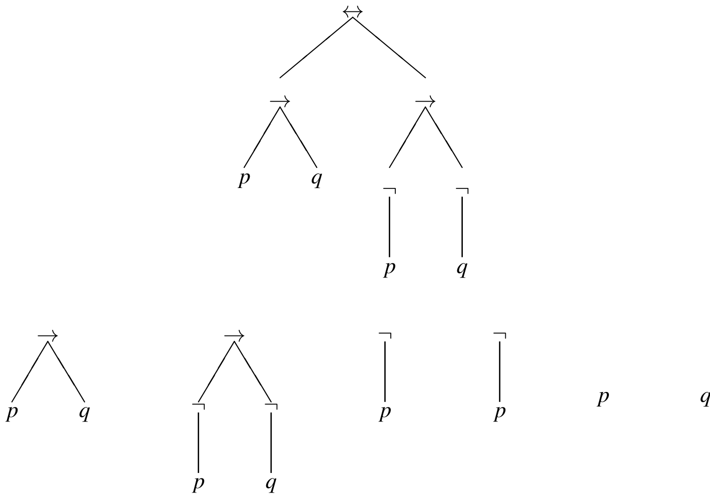


Fig. 2.4 Subformulas

2.3.2 Substitution

Logical equivalence justifies substitution of one formula for another.

Definition 2.30 A is a *subformula* of B if A is a subtree of B. If A is not the same as B, it is a *proper subformula* of B. ■

Example 2.31 Figure 2.4 shows a formula (the left formula from Fig. 2.1) and its proper subformulas. Represented as strings, $(p \rightarrow q) \leftrightarrow (\neg\, p \rightarrow \neg\, q)$ contains the proper subformulas: $p \rightarrow q$, $\neg\, p \rightarrow \neg\, q$, $\neg\, p$, $\neg\, q$, p, q. ■

Definition 2.32 Let A be a subformula of B and let A' be any formula. $B\{A \leftarrow A'\}$, the *substitution of A' for A in B*, is the formula obtained by replacing all occurrences of the subtree for A in B by A'. ■

Example 2.33 Let $B = (p \rightarrow q) \leftrightarrow (\neg\, p \rightarrow \neg\, q)$, $\quad A = p \rightarrow q$ and $A' = \neg\, p \vee q$.

$$B\{A \leftarrow A'\} = (\neg\, p \vee q) \leftrightarrow (\neg\, q \rightarrow \neg\, p).$$

■

Given a formula A, substitution of a logically equivalent formula for a subformula of A does not change its truth value under any interpretation.

Theorem 2.34 *Let A be a subformula of B and let A' be a formula such that $A \equiv A'$. Then $B \equiv B\{A \leftarrow A'\}$.*

Proof Let $\mathscr{I}$ be an arbitrary interpretation. Then $v_{\mathscr{I}}(A) = v_{\mathscr{I}}(A')$ and we must show that $v_{\mathscr{I}}(B) = v_{\mathscr{I}}(B')$. The proof is by induction on the depth d of the highest occurrence of the subtree A in B.

If $d = 0$, there is only one occurrence of A, namely B itself. Obviously, $v_{\mathscr{I}}(B) = v_{\mathscr{I}}(A) = v_{\mathscr{I}}(A') = v_{\mathscr{I}}(B')$.

If $d \neq 0$, then B is $\neg B_1$ or $B_1 \, op \, B_2$ for some formulas B_1, B_2 and operator op. In B_1, the depth of A is less than d. By the inductive hypothesis, $v_{\mathscr{I}}(B_1) = v_{\mathscr{I}}(B_1') = v_{\mathscr{I}}(B_1\{A \leftarrow A'\})$, and similarly $v_{\mathscr{I}}(B_2) = v_{\mathscr{I}}(B_2') = v_{\mathscr{I}}(B_2\{A \leftarrow A'\})$. By the definition of v on the Boolean operators, $v_{\mathscr{I}}(B) = v_{\mathscr{I}}(B')$. ■

2.3.3 Logically Equivalent Formulas

Substitution of logically equivalence formulas is frequently done, for example, to simplify a formula, and it is essential to become familiar with the common equivalences that are listed in this subsection. Their proofs are elementary from the definitions and are left as exercises.

Absorption of Constants

Let us extend the syntax of Boolean formulas to include the two constant atomic propositions *true* and *false*. (Another notation is $\top$ for *true* and $\bot$ for *false*.) Their semantics are defined by $\mathscr{I}(true) = T$ and $\mathscr{I}(false) = F$ for any interpretation. Do not confuse these symbols in the object language of propositional logic with the truth values T and F used to define interpretations. Alternatively, it is possible to regard *true* and *false* as abbreviations for the formulas $p \vee \neg p$ and $p \wedge \neg p$, respectively.

The appearance of a constant in a formula can collapse the formula so that the binary operator is no longer needed; it can even make a formula become a constant whose truth value no longer depends on the non-constant subformula.

$$
\begin{array}{lll@{\qquad}lll}
A \vee true & \equiv & true & A \wedge true & \equiv & A \\
A \vee false & \equiv & A & A \wedge false & \equiv & false \\
A \rightarrow true & \equiv & true & true \rightarrow A & \equiv & A \\
A \rightarrow false & \equiv & \neg A & false \rightarrow A & \equiv & true \\
A \leftrightarrow true & \equiv & A & A \oplus true & \equiv & \neg A \\
A \leftrightarrow false & \equiv & \neg A & A \oplus false & \equiv & A
\end{array}
$$

Identical Operands

Collapsing can also occur when both operands of an operator are the same or one is the negation of another.

$$
\begin{array}{lcllcl}
A & \equiv & \neg\neg A & & & \\
A & \equiv & A \wedge A & A & \equiv & A \vee A \\
A \vee \neg A & \equiv & true & A \wedge \neg A & \equiv & false \\
A \rightarrow A & \equiv & true & & & \\
A \leftrightarrow A & \equiv & true & A \oplus A & \equiv & false \\
\neg A & \equiv & A \uparrow A \neg A & \equiv & A \downarrow A &
\end{array}
$$

Commutativity, Associativity and Distributivity

The binary Boolean operators are commutative, except for implication.

$$
\begin{array}{lcllcl}
A \vee B & \equiv & B \vee A & A \wedge B & \equiv & B \wedge A \\
A \leftrightarrow B & \equiv & B \leftrightarrow A & A \oplus B & \equiv & B \oplus A \\
A \uparrow B & \equiv & B \uparrow A & A \downarrow B & \equiv & B \downarrow A
\end{array}
$$

If negations are added, the direction of an implication can be reversed:

$$A \rightarrow B \equiv \neg B \rightarrow \neg A$$

The formula $\neg B \rightarrow \neg A$ is the *contrapositive* of $A \rightarrow B$.

Disjunction, conjunction, equivalence and non-equivalence are associative.

$$
\begin{array}{lcllcl}
A \vee (B \vee C) & \equiv & (A \vee B) \vee C & A \wedge (B \wedge C) & \equiv & (A \wedge B) \wedge C \\
A \leftrightarrow (B \leftrightarrow C) & \equiv & (A \leftrightarrow B) \leftrightarrow C & A \oplus (B \oplus C) & \equiv & (A \oplus B) \oplus C
\end{array}
$$

Implication, *nor* and *nand* are not associative.

Disjunction and conjunction distribute over each other.

$$
\begin{array}{lcl}
A \vee (B \wedge C) & \equiv & (A \vee B) \wedge (A \vee C) \\
A \wedge (B \vee C) & \equiv & (A \wedge B) \vee (A \wedge C)
\end{array}
$$

Defining One Operator in Terms of Another

When proving theorems *about* propositional logic using structural induction, we have to prove the inductive step for each of the binary operators. It will simplify proofs if we can eliminate some of the operators by replacing subformulas with formulas that use another operator. For example, equivalence can be eliminated be-

cause it can be defined in terms of conjunction and implication. Another reason for eliminating operators is that many algorithms on propositional formulas require that the formulas be in a *normal form*, using a specified subset of the Boolean operators. Here is a list of logical equivalences that can be used to eliminate operators.

$$
\begin{array}{lcllcl}
A \leftrightarrow B & \equiv & (A \rightarrow B) \wedge (B \rightarrow A) & A \oplus B & \equiv & \neg (A \rightarrow B) \vee \neg (B \rightarrow A) \\
A \rightarrow B & \equiv & \neg A \vee B & A \rightarrow B & \equiv & \neg (A \wedge \neg B) \\
A \vee B & \equiv & \neg (\neg A \wedge \neg B) & A \wedge B & \equiv & \neg (\neg A \vee \neg B) \\
A \vee B & \equiv & \neg A \rightarrow B & A \wedge B & \equiv & \neg (A \rightarrow \neg B)
\end{array}
$$

The definition of conjunction in terms of disjunction and negation, and the definition of disjunction in terms of conjunction and negation are called *De Morgan's laws*.

2.4 Sets of Boolean Operators *

From our earliest days in school, we are taught that there are four basic operators in arithmetic: addition, subtraction, multiplication and division. Later on, we learn about additional operators like modulo and absolute value. On the other hand, multiplication and division are theoretically redundant because they can be defined in terms of addition and subtraction.

In this section, we will look at two issues: What Boolean operators are there? What sets of operators are adequate, meaning that all other operators can be defined using just the operators in the set?

2.4.1 Unary and Binary Boolean Operators

Since there are only two Boolean values T and F, the number of possible n-place operators is 2^{2^n}, because for each of the n arguments we can choose either of the two values T and F and for each of these 2^n n-tuples of arguments we can choose the value of the operator to be either T or F. We will restrict ourselves to one- and two-place operators.

The following table shows the $2^{2^1} = 4$ possible one-place operators, where the first column gives the value of the operand x and the other columns give the value of the nth operator $\circ_n(x)$:

x	$\circ_1$	$\circ_2$	$\circ_3$	$\circ_4$
T	T	T	F	F
F	T	F	T	F

x_1	x_2	$\circ_1$	$\circ_2$	$\circ_3$	$\circ_4$	$\circ_5$	$\circ_6$	$\circ_7$	$\circ_8$
T	T	T	T	T	T	T	T	T	T
T	F	T	T	T	T	F	F	F	F
F	T	T	T	F	F	T	T	F	F
F	F	T	F	T	F	T	F	T	F

x_1	x_2	$\circ_9$	$\circ_{10}$	$\circ_{11}$	$\circ_{12}$	$\circ_{13}$	$\circ_{14}$	$\circ_{15}$	$\circ_{16}$
T	T	F	F	F	F	F	F	F	F
T	F	T	T	T	T	F	F	F	F
F	T	T	T	F	F	T	T	F	F
F	F	T	F	T	F	T	F	T	F

Fig. 2.5 Two-place Boolean operators

Of the four one-place operators, three are trivial: $\circ_1$ and $\circ_4$ are the constant operators, and $\circ_2$ is the identity operator which simply maps the operand to itself. The only non-trivial one-place operator is $\circ_3$ which is negation.

There are $2^{2^2} = 16$ two-place operators (Fig. 2.5). Several of the operators are trivial: $\circ_1$ and $\circ_{16}$ are constant; $\circ_4$ and $\circ_6$ are projection operators, that is, their value is determined by the value of only one of operands; $\circ_{11}$ and $\circ_{13}$ are the negations of the projection operators.

The correspondence between the operators in the table and those we defined in Definition 2.1 are shown in the following table, where the operators in the right-hand column are the negations of those in the left-hand column.

op	name	symbol	op	name	symbol
$\circ_2$	disjunction	$\vee$	$\circ_{15}$	nor	$\downarrow$
$\circ_8$	conjunction	$\wedge$	$\circ_9$	nand	$\uparrow$
$\circ_5$	implication	$\rightarrow$			
$\circ_7$	equivalence	$\leftrightarrow$	$\circ_{10}$	exclusive or	$\oplus$

The operator $\circ_{12}$ is the negation of implication and is not used. Reverse implication, $\circ_3$, is used in logic programming (Chap. 11); its negation, $\circ_{14}$, is not used.

2.4.2 Adequate Sets of Operators

Definition 2.35 A binary operator $\circ$ *is defined from* a set of operators $\{\circ_1, \ldots, \circ_n\}$ iff there is a logical equivalence $A_1 \circ A_2 \equiv A$, where A is a formula constructed from occurrences of A_1 and A_2 using the operators $\{\circ_1, \ldots, \circ_n\}$. The unary operator $\neg$

is defined by a formula $\neg A_1 \equiv A$, where A is constructed from occurrences of A_1 and the operators in the set. ■

Theorem 2.36 *The Boolean operators* $\vee, \wedge, \rightarrow, \leftrightarrow, \oplus, \uparrow, \downarrow$ *can be defined from negation and* one *of* $\vee, \wedge, \rightarrow$.

Proof The theorem follows by using the logical equivalences in Sect. 2.3.3. The *nand* and *nor* operators are the negations of conjunction and disjunction, respectively. Equivalence can be defined from implication and conjunction and non-equivalence can be defined using these operators and negation. Therefore, we need only $\rightarrow, \vee, \wedge$, but each of these operators can be defined by one of the others and negation as shown by the equivalences on page 26. ■

It may come as a surprise that it is possible to define all Boolean operators from either *nand* or *nor* alone. The equivalence $\neg A \equiv A \uparrow A$ is used to define negation from *nand* and the following sequence of equivalences shows how conjunction can be defined:

$$\begin{array}{lll}
(A \uparrow B) \uparrow (A \uparrow B) & \equiv & \text{by the definition of } \uparrow \\
\neg((A \uparrow B) \wedge (A \uparrow B)) & \equiv & \text{by idempotence} \\
\neg(A \uparrow B) & \equiv & \text{by the definition of } \uparrow \\
\neg\neg(A \wedge B) & \equiv & \text{by double negation} \\
A \wedge B. & &
\end{array}$$

From the formulas for negation and conjunction, all other operators can be defined. Similarly definitions are possible using *nor*.

In fact it can be proved that only *nand* and *nor* have this property.

Theorem 2.37 *Let* $\circ$ *be a binary operator that can define negation and all other binary operators by itself. Then* $\circ$ *is either nand or nor.*

Proof We give an outline of the proof and leave the details as an exercise.

Suppose that $\circ$ is an operator that can define all the other operators. Negation must be defined by an equivalence of the form:

$$\neg A \equiv A \circ \cdots \circ A.$$

Any binary operator *op* must be defined by an equivalence:

$$A_1 \; op \; A_2 \equiv B_1 \circ \cdots \circ B_n,$$

where each B_i is either A_1 or A_2. (If $\circ$ is not associative, add parentheses as necessary.) We will show that these requirements impose restrictions on $\circ$ so that it must be *nand* or *nor*.

Let $\mathscr{I}$ be any interpretation such that $v_{\mathscr{I}}(A) = T$; then

$$F = v_{\mathscr{I}}(\neg A) = v_{\mathscr{I}}(A \circ \cdots \circ A).$$

Prove by induction on the number of occurrences of $\circ$ that $v_{\mathscr{I}}(A_1 \circ A_2) = F$ when $v_{\mathscr{I}}(A_1) = T$ and $v_{\mathscr{I}}(A_2) = T$. Similarly, if $\mathscr{I}$ is an interpretation such that $v_{\mathscr{I}}(A) = F$, prove that $v_{\mathscr{I}}(A_1 \circ A_2) = T$.

Thus the only freedom we have in defining $\circ$ is in the case where the two operands are assigned different truth values:

A_1	A_2	$A_1 \circ A_2$
T	T	F
T	F	T or F
F	T	T or F
F	F	T

If $\circ$ is defined to give the same truth value T for these two lines then $\circ$ is *nand*, and if $\circ$ is defined to give the same truth value F then $\circ$ is *nor*.

The remaining possibility is that $\circ$ is defined to give different truth values for these two lines. Prove by induction that only projection and negated projection are definable in the sense that:

$$B_1 \circ \cdots \circ B_n \equiv \neg \cdots \neg B_i$$

for some i and zero or more negations. ■

2.5 Satisfiability, Validity and Consequence

We now define the fundamental concepts of the semantics of formulas:

Definition 2.38 Let $A \in \mathscr{F}$.

- A is *satisfiable* iff $v_{\mathscr{I}}(A) = T$ for *some* interpretation $\mathscr{I}$.
  A satisfying interpretation is a *model* for A.
- A is *valid*, denoted $\models A$, iff $v_{\mathscr{I}}(A) = T$ for *all* interpretations $\mathscr{I}$.
  A valid propositional formula is also called a *tautology*.
- A is *unsatisfiable* iff it is not satisfiable, that is, if $v_{\mathscr{I}}(A) = F$ for *all* interpretations $\mathscr{I}$.
- A is *falsifiable*, denoted $\not\models A$, iff it is not valid, that is, if $v_{\mathscr{I}}(A) = F$ for *some* interpretation v. ■

These concepts are illustrated in Fig. 2.6.

The four semantical concepts are closely related.

Theorem 2.39 *Let $A \in \mathscr{F}$. A is valid if and only if $\neg A$ is unsatisfiable. A is satisfiable if and only if $\neg A$ is falsifiable.*

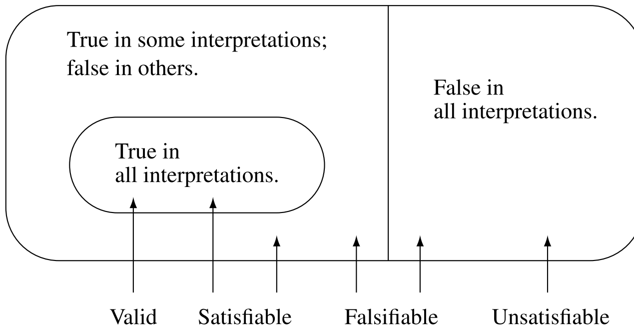


Fig. 2.6 Satisfiability and validity of formulas

Proof Let $\mathscr{I}$ be an arbitrary interpretation. $v_{\mathscr{I}}(A) = T$ if and only if $v_{\mathscr{I}}(\neg A) = F$ by the definition of the truth value of a negation. Since $\mathscr{I}$ was arbitrary, A is true in all interpretations if and only if $\neg A$ is false in all interpretations, that is, iff $\neg A$ is unsatisfiable.

If A is satisfiable then for *some* interpretation $\mathscr{I}$, $v_{\mathscr{I}}(A) = T$. By definition of the truth value of a negation, $v_{\mathscr{I}}(\neg A) = F$ so that $\neg A$ is falsifiable. Conversely, if $v_{\mathscr{I}}(\neg A) = F$ then $v_{\mathscr{I}}(A) = T$. ■

2.5.1 Decision Procedures in Propositional Logic

Definition 2.40 Let $\mathscr{U} \subseteq \mathscr{F}$ be a set of formulas. An algorithm is a *decision procedure* for $\mathscr{U}$ if given an arbitrary formula $A \in \mathscr{F}$, it terminates and returns the answer *yes* if $A \in \mathscr{U}$ and the answer *no* if $A \notin \mathscr{U}$. ■

If $\mathscr{U}$ is the set of satisfiable formulas, a decision procedure for $\mathscr{U}$ is called a decision procedure for satisfiability, and similarly for validity.

By Theorem 2.39, a decision procedure for satisfiability can be used as a decision procedure for validity. To decide if A is valid, apply the decision procedure for satisfiability to $\neg A$. If it reports that $\neg A$ is satisfiable, then A is not valid; if it reports that $\neg A$ is not satisfiable, then A is valid. Such an decision procedure is called a *refutation procedure*, because we prove the validity of a formula by refuting its negation. Refutation procedures can be efficient algorithms for deciding validity, because instead of checking that the formula is always true, we need only search for a falsifying counterexample.

The existence of a decision procedure for satisfiability in propositional logic is trivial, because we can build a truth table for any formula. The truth table in Example 2.21 shows that $p \rightarrow q$ is satisfiable, but not valid; Example 2.22 shows that $(p \rightarrow q) \leftrightarrow (\neg q \rightarrow \neg p)$ is valid. The following example shows an unsatisfiable formula.

Example 2.41 The formula $(p \vee q) \wedge \neg\, p \wedge \neg\, q$ is unsatisfiable because all lines of its truth table evaluate to F.

p	q	$p \vee q$	$\neg\, p$	$\neg\, q$	$(p \vee q) \wedge \neg\, p \wedge \neg\, q$
T	T	T	F	F	F
T	F	T	F	T	F
F	T	T	T	F	F
F	F	F	T	T	F

■

The method of truth tables is a very inefficient decision procedure because we need to evaluate a formula for each of 2^n possible interpretations, where n is the number of distinct atoms in the formula. In later chapters we will discuss more efficient decision procedures for satisfiability, though it is extremely unlikely that there is a decision procedure that is efficient for all formulas (see Sect. 6.7).

2.5.2 Satisfiability of a Set of Formulas

The concept of satisfiability can be extended to a set of formulas.

Definition 2.42 A set of formulas $U = \{A_1, \ldots\}$ is *(simultaneously) satisfiable* iff there exists an interpretation $\mathscr{I}$ such that $v_{\mathscr{I}}(A_i) = T$ for all i. The satisfying interpretation is a *model* of U. U is *unsatisfiable* iff for every interpretation $\mathscr{I}$, there exists an i such that $v_{\mathscr{I}}(A_i) = F$. ■

Example 2.43 The set $U_1 = \{p,\ \neg\, p \vee q,\ q \wedge r\}$ is simultaneously satisfiable by the interpretation which assigns T to each atom, while the set $U_2 = \{p,\ \neg\, p \vee q,\ \neg\, p\}$ is unsatisfiable. Each formula in U_2 is satisfiable by itself, but the set is not simultaneously satisfiable. ■

The proofs of the following elementary theorems are left as exercises.

Theorem 2.44 *If U is satisfiable, then so is $U - \{A_i\}$ for all i.*

Theorem 2.45 *If U is satisfiable and B is valid, then $U \cup \{B\}$ is satisfiable.*

Theorem 2.46 *If U is unsatisfiable, then for any formula B, $U \cup \{B\}$ is unsatisfiable.*

Theorem 2.47 *If U is unsatisfiable and for some i, A_i is valid, then $U - \{A_i\}$ is unsatisfiable.*

2.5.3 Logical Consequence

Definition 2.48 Let U be a set of formulas and A a formula. A is a *logical consequence* of U, denoted $U \models A$, iff every model of U is a model of A. ■

The formula A need not be true in every possible interpretation, only in those interpretations which satisfy U, that is, those interpretations which satisfy every formula in U. If U is empty, logical consequence is the same as validity.

Example 2.49 Let $A = (p \vee r) \wedge (\neg q \vee \neg r)$. Then A is a logical consequence of $\{p, \neg q\}$, denoted $\{p, \neg q\} \models A$, since A is true in all interpretations $\mathscr{I}$ such that $\mathscr{I}(p) = T$ and $\mathscr{I}(q) = F$. However, A is not valid, since it is not true in the interpretation $\mathscr{I}'$ where $\mathscr{I}'(p) = F$, $\mathscr{I}'(q) = T$, $\mathscr{I}'(r) = T$. ■

The caveat concerning $\leftrightarrow$ and $\equiv$ also applies to $\rightarrow$ and $\models$. Implication, $\rightarrow$, is an operator in the object language, while $\models$ is a symbol for a concept in the metalanguage. However, as with equivalence, the two concepts are related:

Theorem 2.50 *$U \models A$ if and only if $\models \bigwedge_i A_i \rightarrow A$.*

Definition 2.51 $\bigwedge_{i=1}^{i=n} A_i$ is an abbreviation for $A_1 \wedge \cdots \wedge A_n$. The notation $\bigwedge_i$ is used if the bounds are obvious from the context or if the set of formulas is infinite. A similar notation $\bigvee$ is used for disjunction. ■

Example 2.52 From Example 2.49, $\{p, \neg q\} \models (p \vee r) \wedge (\neg q \vee \neg r)$, so by Theorem 2.50, $\models (p \wedge \neg q) \rightarrow (p \vee r) \wedge (\neg q \vee \neg r)$. ■

The proof of Theorem 2.50, as well as the proofs of the following two theorems are left as exercises.

Theorem 2.53 *If $U \models A$ then $U \cup \{B\} \models A$ for any formula B.*

Theorem 2.54 *If $U \models A$ and B is valid then $U - \{B\} \models A$.*

2.5.4 Theories *

Logical consequence is the central concept in the foundations of mathematics. Valid logical formulas such as $p \vee q \leftrightarrow q \vee p$ are of little mathematical interest. It is much more interesting to assume that a set of formulas is true and then to investigate the consequences of these assumptions. For example, Euclid assumed five formulas about geometry and deduced an extensive set of logical consequences. The formal definition of a mathematical theory is as follows.

Definition 2.55 Let $\mathscr{T}$ be a set of formulas. $\mathscr{T}$ is *closed under logical consequence* iff for all formulas A, if $\mathscr{T} \models A$ then $A \in \mathscr{T}$. A set of formulas that is closed under logical consequence is a *theory*. The elements of $\mathscr{T}$ are *theorems*. ■

Theories are constructed by selecting a set of formulas called *axioms* and deducing their logical consequences.

Definition 2.56 Let $\mathscr{T}$ be a theory. $\mathscr{T}$ is said to be *axiomatizable* iff there exists a set of formulas U such that $\mathscr{T} = \{A \mid U \models A\}$. The set of formulas U are the *axioms* of $\mathscr{T}$. If U is finite, $\mathscr{T}$ is said to be *finitely axiomatizable*. ■

Arithmetic is axiomatizable: There is a set of axioms developed by Peano whose logical consequences are theorems of arithmetic. Arithmetic is not finitely axiomatizable, because the induction axiom is not by a single axiom but an axiom scheme with an instance for each property in arithmetic.

2.6 Semantic Tableaux

The method of *semantic tableaux* is an efficient decision procedure for satisfiability (and by duality validity) in propositional logic. We will use semantic tableaux extensively in the next chapter to prove important theorems about deductive systems. The principle behind semantic tableaux is very simple: search for a model (satisfying interpretation) by decomposing the formula into sets of atoms and negations of atoms. It is easy to check if there is an interpretation for each set: a set of atoms and negations of atoms is satisfiable iff the set does not contain an atom p and its negation $\neg p$. The formula is satisfiable iff one of these sets is satisfiable.

We begin with some definitions and then analyze the satisfiability of two formulas to motivate the construction of semantic tableaux.

2.6.1 Decomposing Formulas into Sets of Literals

Definition 2.57 A *literal* is an atom or the negation of an atom. An atom is a *positive literal* and the negation of an atom is a *negative literal*. For any atom p, $\{p, \neg p\}$ is a *complementary pair of literals*.

For any formula A, $\{A, \neg A\}$ is a *complementary pair of formulas*. A is the *complement* of $\neg A$ and $\neg A$ is the *complement* of A. ■

Example 2.58 In the set of literals $\{\neg p, q, r, \neg r\}$, q and r are positive literals, while $\neg p$ and $\neg r$ are negative literals. The set contains the complementary pair of literals $\{r, \neg r\}$. ■

Example 2.59 Let us analyze the satisfiability of the formula:

$$A = p \wedge (\neg q \vee \neg p)$$

in an arbitrary interpretation $\mathscr{I}$, using the inductive rules for the evaluation of the truth value of a formula.

- The principal operator of A is conjunction, so $v_{\mathscr{I}}(A) = T$ if and only if both $v_{\mathscr{I}}(p) = T$ and $v_{\mathscr{I}}(\neg q \vee \neg p) = T$.
- The principal operator of $\neg q \vee \neg p$ is disjunction, so $v_{\mathscr{I}}(\neg q \vee \neg p) = T$ if and only if either $v_{\mathscr{I}}(\neg q) = T$ or $v_{\mathscr{I}}(\neg p) = T$.
- Integrating the information we have obtained from this analysis, we conclude that $v_{\mathscr{I}}(A) = T$ if and only if either:

1. $v_{\mathscr{I}}(p) = T$ and $v_{\mathscr{I}}(\neg q) = T$, or
2. $v_{\mathscr{I}}(p) = T$ and $v_{\mathscr{I}}(\neg p) = T$.

A is satisfiable if and only if there is an interpretation such that (1) holds or an interpretation such that (2) holds. ■

We have reduced the question of the satisfiability of A to a question about the satisfiability of sets of literals.

Theorem 2.60 *A set of literals is satisfiable if and only if it does* not *contain a complementary pair of literals.*

Proof Let L be a set of literals that does not contain a complementary pair. Define the interpretation $\mathscr{I}$ by:

$$\begin{aligned} \mathscr{I}(p) = T \quad & \text{if } p \in L, \\ \mathscr{I}(p) = F \quad & \text{if } \neg p \in L. \end{aligned}$$

The interpretation is well-defined—there is only one value assigned to each atom in L—since there is no complementary pair of literals in L. Each literal in L evaluates to T so L is satisfiable.

Conversely, if $\{p, \neg p\} \subseteq L$, then for any interpretation $\mathscr{I}$ for the atoms in L, either $v_{\mathscr{I}}(p) = F$ or $v_{\mathscr{I}}(\neg p) = F$, so L is not satisfiable. ■

Example 2.61 Continuing the analysis of the formula $A = p \wedge (\neg q \vee \neg p)$ from Example 2.59, A is satisfiable if and only at least one of the sets $\{p, \neg p\}$ and $\{p, \neg q\}$ does not contain a complementary pair of literals. Clearly, only the second set does not contain a complementary pair of literals. Using the method described in Theorem 2.60, we obtain the interpretation:

$$\mathscr{I}(p) = T, \qquad \mathscr{I}(q) = F.$$

We leave it to the reader to check that for this interpretation, $v_{\mathscr{I}}(A) = T$. ■

The following example shows what happens if a formula is unsatisfiable.

Example 2.62 Consider the formula:

$$B = (p \vee q) \wedge (\neg p \wedge \neg q).$$

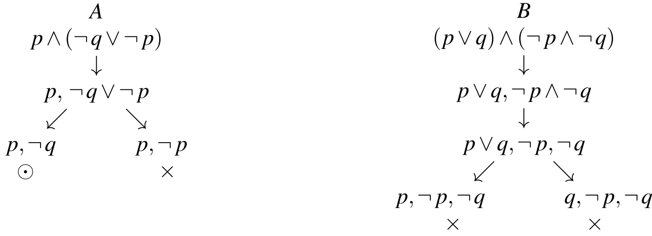


Fig. 2.7 Semantic tableaux

The analysis of the formula proceeds as follows:

- $v_{\mathscr{I}}(B) = T$ if and only if $v_{\mathscr{I}}(p \vee q) = T$ and $v_{\mathscr{I}}(\neg p \wedge \neg q) = T$.
- Decomposing the conjunction, $v_{\mathscr{I}}(B)$=T if and only if $v_{\mathscr{I}}(p \vee q) = T$ and $v_{\mathscr{I}}(\neg p) = v_{\mathscr{I}}(\neg q) = T$.
- Decomposing the disjunction, $v_{\mathscr{I}}(B) = T$ if and only if either:

1. $v_{\mathscr{I}}(p) = v_{\mathscr{I}}(\neg p) = v_{\mathscr{I}}(\neg q) = T$, or
2. $v_{\mathscr{I}}(q) = v_{\mathscr{I}}(\neg p) = v_{\mathscr{I}}(\neg q) = T$.

Both sets of literals $\{p, \neg p, \neg q\}$ and $\{q, \neg p, \neg q\}$ contain complementary pairs, so by Theorem 2.60, both set of literals are unsatisfiable. We conclude that it is impossible to find a model for B; in other words, B is unsatisfiable. ■

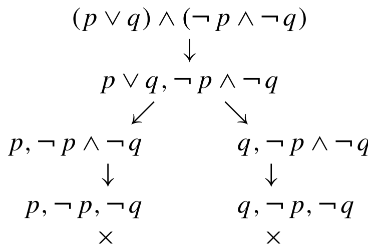
2.6.2 Construction of Semantic Tableaux

The decomposition of a formula into sets of literals is rather difficult to follow when expressed textually, as we did in Examples 2.59 and 2.62. In the method of semantic tableaux, sets of formulas label nodes of a tree, where each path in the tree represents the formulas that must be satisfied in one possible interpretation.

The initial formula labels the root of the tree; each node has one or two child nodes depending on how a formula labeling the node is decomposed. The leaves are labeled by the sets of literals. A leaf labeled by a set of literals containing a complementary pair of literals is marked ×, while a leaf labeled by a set not containing a complementary pair is marked ⊙.

Figure 2.7 shows semantic tableaux for the formulas from the examples.

The tableau construction is not unique; here is another tableau for B:



α	α_1	α_2	β	β_1	β_2
$\neg\neg A_1$	A_1				
$A_1 \wedge A_2$	A_1	A_2	$\neg(B_1 \wedge B_2)$	$\neg B_1$	$\neg B_2$
$\neg(A_1 \vee A_2)$	$\neg A_1$	$\neg A_2$	$B_1 \vee B_2$	B_1	B_2
$\neg(A_1 \rightarrow A_2)$	A_1	$\neg A_2$	$B_1 \rightarrow B_2$	$\neg B_1$	B_2
$\neg(A_1 \uparrow A_2)$	A_1	A_2	$B_1 \uparrow B_2$	$\neg B_1$	$\neg B_2$
$A_1 \downarrow A_2$	$\neg A_1$	$\neg A_2$	$\neg(B_1 \downarrow B_2)$	B_1	B_2
$A_1 \leftrightarrow A_2$	$A_1 \rightarrow A_2$	$A_2 \rightarrow A_1$	$\neg(B_1 \leftrightarrow B_2)$	$\neg(B_1 \rightarrow B_2)$	$\neg(B_2 \rightarrow B_1)$
$\neg(A_1 \oplus A_2)$	$A_1 \rightarrow A_2$	$A_2 \rightarrow A_1$	$B_1 \oplus B_2$	$\neg(B_1 \rightarrow B_2)$	$\neg(B_2 \rightarrow B_1)$

Fig. 2.8 Classification of α- and β-formulas

It is constructed by branching to search for a satisfying interpretation for $p \vee q$ before searching for one for $\neg p \wedge \neg q$. The first tableau contains fewer nodes, showing that it is preferable to decompose conjunctions before disjunctions.

A concise presentation of the rules for creating a semantic tableau can be given if formulas are classified according to their principal operator (Fig. 2.8). If the formula is a negation, the classification takes into account both the negation and the principal operator. α-formulas are conjunctive and are satisfiable only if both subformulas α_1 and α_2 are satisfied, while β-formulas are disjunctive and are satisfied even if only one of the subformulas β_1 or β_2 is satisfiable.

Example 2.63 The formula $p \wedge q$ is classified as an α-formula because it is true if and only if both p and q are true. The formula $\neg(p \wedge q)$ is classified as a β-formula. It is logically equivalent to $\neg p \vee \neg q$ and is true if and only if either $\neg p$ is true or $\neg q$ is true. ■

We now give the algorithm for the construction of a semantic tableau for a formula in propositional logic.

Algorithm 2.64 (Construction of a semantic tableau)
Input: A formula ϕ of propositional logic.
Output: A semantic tableau $\mathscr{T}$ for ϕ all of whose leaves are marked.

Initially, $\mathscr{T}$ is a tree consisting of a single root node labeled with the singleton set $\{\phi\}$. This node is not marked.

Repeat the following step as long as possible: *Choose* an unmarked leaf l labeled with a set of formulas $U(l)$ and apply one of the following rules.

- $U(l)$ is a set of literals. Mark the leaf *closed* $\times$ if it contains a complementary pair of literals. If not, mark the leaf *open* $\odot$.
- $U(l)$ is not a set of literals. *Choose* a formula in $U(l)$ which is not a literal. Classify the formula as an α-formula A or as a β-formula B and perform one of the following steps according to the classification:

- A is an α-formula. Create a new node l' as a child of l and label l' with:

$$U(l') = (U(l) - \{A\}) \cup \{A_1, A_2\}.$$

  (In the case that A is $\neg\neg A_1$, there is no A_2.)
- B is a β-formula. Create two new nodes l' and l'' as children of l. Label l' with:

$$U(l') = (U(l) - \{B\}) \cup \{B_1\},$$

  and label l'' with:

$$U(l'') = (U(l) - \{B\}) \cup \{B_2\}.$$

■

Definition 2.65 A tableau whose construction has terminated is a *completed tableau*. A completed tableau is *closed* if all its leaves are marked closed. Otherwise (if some leaf is marked open), it is *open*. ■

2.6.3 Termination of the Tableau Construction

Since each step of the algorithm decomposes one formula into one or two simpler formulas, it is clear that the construction of the tableau for any formula terminates, but it is worth proving this claim.

Theorem 2.66 *The construction of a tableau for any formula ϕ terminates. When the construction terminates, all the leaves are marked $\times$ or $\odot$.*

Proof Let us assume that $\leftrightarrow$ and $\oplus$ do not occur in the formula ϕ; the extension of the proof for these cases is left as an exercise.

Consider an unmarked leaf l that is chosen to be expanded during the construction of the tableau. Let $b(l)$ be the total number of binary operators in all formulas in $U(l)$ and let $n(l)$ be the total number of negations in $U(l)$. Define:

$$W(l) = 3 \cdot b(l) + n(l).$$

For example, if $U(l) = \{p \vee q, \neg p \wedge \neg q\}$, then $W(l) = 3 \cdot 2 + 2 = 8$.

Each step of the algorithm adds either a new node l' or a pair of new nodes l', l'' as children of l. We claim that $W(l') < W(l)$ and, if there is a second node, $W(l'') < W(l)$.

Suppose that $A = \neg(A_1 \vee A_2)$ and that the rule for this α-formula is applied at l to obtain a new leaf l' labeled:

$$U(l') = (U(l) - \{\neg(A_1 \vee A_2)\}) \cup \{\neg A_1, \neg A_2\}.$$

Then:

$$W(l') = W(l) - (3 \cdot 1 + 1) + 2 = W(l) - 2 < W(l),$$

because one binary operator and one negation are removed, while two negations are added.

Suppose now that $B = B_1 \vee B_2$ and that the rule for this β-formula is applied at l to obtain two new leaves l', l'' labeled:

$$\begin{aligned} U(l') &= (U(l) - \{B_1 \vee B_2\}) \cup \{B_1\}, \\ U(l'') &= (U(l) - \{B_1 \vee B_2\}) \cup \{B_2\}. \end{aligned}$$

Then:

$$W(l') \leq W(l) - (3 \cdot 1) < W(l), \qquad W(l'') \leq W(l) - (3 \cdot 1) < W(l).$$

We leave it to the reader to prove that $W(l)$ decreases for the other α- and β-formulas.

The value of $W(l)$ decreases as each branch in the tableau is extended. Since, obviously, $W(l) \geq 0$, no branch can be extended indefinitely and the construction of the tableau must eventually terminate.

A branch can always be extended if its leaf is labeled with a set of formulas that is not a set of literals. Therefore, when the construction of the tableau terminates, all leaves are labeled with sets of literals and each is marked open or closed by the first rule of the algorithm. ■

*2.6.4 Improving the Efficiency of the Algorithm **

The algorithm for constructing a tableau is not deterministic: at most steps, there is a choice of which leaf to extend and if the leaf contains more than one formula which is not a literal, there is a choice of which formula to decompose. This opens the possibility of applying heuristics in order to cause the tableau to be completed quickly. We saw in Sect. 2.6.2 that it is better to decompose α-formulas before β-formulas to avoid duplication.

Tableaux can be shortened by closing a branch if it contains a formula and its negation and not just a pair of complementary literals. Clearly, there is no reason to continue expanding a node containing:

$$(p \wedge (q \vee r)), \quad \neg\, (p \wedge (q \vee r)).$$

We leave it as an exercise to prove that this modification preserves the correctness of the algorithm.

There is a lot of redundancy in copying formulas from one node to another:

$$U(l') = (U(l) - \{A\}) \cup \{A_1, A_2\}.$$

In a variant of semantic tableaux called *analytic tableaux* (Smullyan, 1968), when a new node is created, it is labeled only with the new formulas:

$$U(l') = \{A_1, A_2\}.$$

The algorithm is changed so that the formula to be decomposed is selected from the set of formulas labeling the nodes on the branch from the root to a leaf (provided, of course, that the formula has not already been selected). A leaf is marked closed if two complementary literals (or formulas) appear in the labels of one or two nodes on a branch, and a leaf is marked open if is not closed but there are no more formulas to decompose.

Here is an analytic tableau for the formula B from Example 2.62, where the formula $p \vee q$ is not copied from the second node to the third when $p \wedge q$ is decomposed:

$$\begin{array}{c}(p \vee q) \wedge (\neg p \wedge \neg q)\\ \downarrow \\ p \vee q, \neg p \wedge \neg q \\ \downarrow \\ \neg p, \neg q \\ \swarrow \qquad \searrow \\ p \qquad\qquad q \\ \times \qquad\qquad \times\end{array}$$

We prefer to use semantic tableaux because it is easy to see which formulas are candidates for decomposition and how to mark leaves.

2.7 Soundness and Completeness

The construction of a semantic tableau is a purely formal. The decomposition of a formula depends solely on its syntactical properties: its principal operator and—if it is a negation—the principal operator of the formula that is negated. We gave several examples to motivate semantic tableau, but we have not yet proven that the algorithm is correct. We have not connected the syntactical outcome of the algorithm (Is the tableau closed or not?) with the semantical concept of truth value. In this section, we prove that the algorithm is correct in the sense that it reports that a formula is satisfiable or unsatisfiable if and only if there exists or does not exist a model for the formula.

The proof techniques of this section should be studied carefully because they will be used again and again in other logical systems.

Theorem 2.67 Soundness and completeness *Let $\mathscr{T}$ be a completed tableau for a formula A. A is unsatisfiable if and only if $\mathscr{T}$ is closed.*

Here are some corollaries that follow from the theorem.

Corollary 2.68 *A is satisfiable if and only if $\mathscr{T}$ is open.*

Proof A is satisfiable iff (by definition) A is not unsatisfiable iff (by Theorem 2.67) $\mathscr{T}$ is not closed iff (by definition) $\mathscr{T}$ is open. ■

Corollary 2.69 *A is valid if and only if the tableau for $\neg A$ closes.*

Proof A is valid iff $\neg A$ is unsatisfiable iff the tableau for $\neg A$ closes. ■

Corollary 2.70 *The method of semantic tableaux is a decision procedure for validity in propositional logic.*

Proof Let A be a formula of propositional logic. By Theorem 2.66, the construction of the semantic tableau for $\neg A$ terminates in a completed tableau. By the previous corollary, A is valid if and only if the completed tableau is closed. ■

The forward direction of Corollary 2.69 is called *completeness*: if A is valid, we can discover this fact by constructing a tableau for $\neg A$ and the tableau will close. The converse direction is called *soundness*: any formula A that the tableau construction claims valid (because the tableau for $\neg A$ closes) actually is valid. Invariably in logic, soundness is easier to show than completeness. The reason is that while we only include in a formal system rules that are obviously sound, it is hard to be sure that we haven't forgotten some rule that may be needed for completeness. At the extreme, the following vacuous algorithm is sound but far from complete!

Algorithm 2.71 (Incomplete decision procedure for validity)
Input: A formula A of propositional logic.
Output: A is not valid. ■

Example 2.72 If the rule for $\neg(A_1 \vee A_2)$ is omitted, the construction of the tableau is still sound, but it is not complete, because it is impossible to construct a closed tableau for the obviously valid formula $A = \neg p \vee p$. Label the root of the tableau with the negation $\neg A = \neg(\neg p \vee p)$; there is now no rule that can be used to decompose the formula. ■

2.7.1 Proof of Soundness

The theorem to be proved is: if the tableau $\mathscr{T}$ for a formula A closes, then A is unsatisfiable. We will prove a more general theorem: if $\mathscr{T}_n$, the subtree rooted at node n of $\mathscr{T}$, closes then the set of formulas $U(n)$ labeling n is unsatisfiable. Soundness is the special case for the root.

To make the proof easier to follow, we will use $A_1 \wedge A_2$ and $B_1 \vee B_2$ as representatives of the classes of α- and β-formulas, respectively.

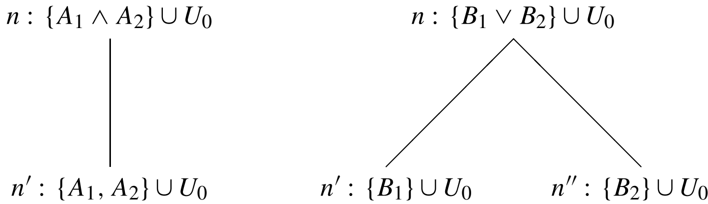
Proof of Soundness The proof is by induction on the height h_n of the node n in $\mathscr{T}_n$. Clearly, a closed leaf is labeled by an unsatisfiable set of formulas. Recall (Definition 2.42) that a set of formulas is unsatisfiable iff for any interpretation the truth value of at least one formula is false. In the inductive step, if the children of a node n

are labeled by an unsatisfiable set of formulas, then: (a) either the unsatisfiable formula also appears in the label of n, or (b) the unsatisfiable formulas in the labels of the children were used to construct an unsatisfiable formula in the label of n. Let us write out the formal proof.

For the base case, $h_n = 0$, assume that $\mathcal{T}_n$ closes. Since $h_n = 0$ means that n is a leaf, $U(n)$ must contain a complementary set of literals so it is unsatisfiable.

For the inductive step, let n be a node such that $h_n > 0$ in $\mathcal{T}_n$. We need to show that $\mathcal{T}_n$ is closed implies that $U(n)$ is unsatisfiable. By the inductive hypothesis, we can assume that for any node m of height $h_m < h_n$, if $\mathcal{T}_m$ closes, then $U(m)$ is unsatisfiable.

Since $h_n > 0$, the rule for some α- or β-formula was used to create the children of n:



Case 1: $U(n) = \{A_1 \wedge A_2\} \cup U_0$ and $U(n') = \{A_1, A_2\} \cup U_0$ for some (possibly empty) set of formulas U_0.

Clearly, $\mathcal{T}_{n'}$ is also a closed tableau and since $h_{n'} = h_n - 1$, by the inductive hypothesis $U(n')$ is unsatisfiable. Let $\mathscr{I}$ be an arbitrary interpretation. There are two possibilities:

- $v_{\mathscr{I}}(A_0) = F$ for some formula $A_0 \in U_0$. But $U_0 \subset U(n)$ so $U(n)$ is also unsatisfiable.
- Otherwise, $v_{\mathscr{I}}(A_0) = T$ for all $A_0 \in U_0$, so $v_{\mathscr{I}}(A_1) = F$ *or* $v_{\mathscr{I}}(A_2) = F$. Suppose that $v_{\mathscr{I}}(A_1) = F$. By the definition of the semantics of $\wedge$, this implies that $v_{\mathscr{I}}(A_1 \wedge A_2) = F$. Since $A_1 \wedge A_2 \in U(n)$, $U(n)$ is unsatisfiable. A similar argument holds if $v_{\mathscr{I}}(A_2) = F$.

Case 2: $U(n) = \{B_1 \vee B_2\} \cup U_0$, $U(n') = \{B_1\} \cup U_0$, and $U(n'') = \{B_2\} \cup U_0$ for some (possibly empty) set of formulas U_0.

Clearly, $\mathcal{T}_{n'}$ and $\mathcal{T}_{n''}$ are also closed tableaux and since $h_{n'} \le h_n - 1$ and $h_{n''} \le h_n - 1$, by the inductive hypothesis $U(n')$ and $U(n'')$ are both unsatisfiable. Let $\mathscr{I}$ be an arbitrary interpretation. There are two possibilities:

- $v_{\mathscr{I}}(B_0) = F$ for some formula $B_0 \in U_0$. But $U_0 \subset U(n)$ so $U(n)$ is also unsatisfiable.
- Otherwise, $v_{\mathscr{I}}(B_0) = T$ for all $B_0 \in U_0$, so $v_{\mathscr{I}}(B_1) = F$ (since $U(n')$ is unsatisfiable) *and* $v_{\mathscr{I}}(B_2) = F$ (since $U(n'')$ is unsatisfiable). By the definition of the semantics of $\vee$, this implies that $v_{\mathscr{I}}(B_1 \vee B_2) = F$. Since $B_1 \vee B_2 \in U(n)$, $U(n)$ is unsatisfiable. ■

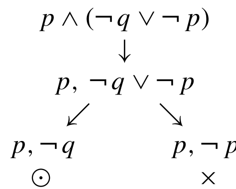
2.7.2 Proof of Completeness

The theorem to be proved is: if A is unsatisfiable then *every* tableau for A closes. Completeness is much more difficult to prove than soundness. For soundness, we had a single (though arbitrary) closed tableau for a formula A and we proved that A is unsatisfiable by induction on the structure of a tableau. Here we need to prove that no matter how the tableau for A is constructed, it must close.

Rather than prove that every tableau must close, we prove the contrapositive (Corollary 2.68): if some tableau for A is open (has an open branch), then A is satisfiable. Clearly, there is a model for the set of literals labeling the leaf of an open branch. We extend this to an interpretation for A and then prove by induction on the length of the branch that the interpretation is a model of the sets of formulas labeling the nodes on the branch, including the singleton set $\{A\}$ that labels the root.

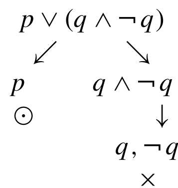
Let us look at some examples.

Example 2.73 Let $A = p \wedge (\neg q \vee \neg p)$. We have already constructed the tableau for A which is reproduced here:



The interpretation $\mathscr{I}(p) = T$, $\mathscr{I}(q) = F$ defined by assigning T to the literals labeling the leaf of the open branch is clearly a model for A. ■

Example 2.74 Now let $A = p \vee (q \wedge \neg q)$; here is a tableau for A:



The open branch of the tableau terminates in a leaf labeled with the singleton set of literals $\{p\}$. We can conclude that any model for A must define $\mathscr{I}(p) = T$. However, an interpretation for A must also define an assignment to q and the leaf gives us no guidance as to which value to choose for $\mathscr{I}(q)$. But it is obvious that it doesn't matter what value is assigned to q; in either case, the interpretation will be a model of A. ■

To prove completeness we need to show that the assignment of T to the literals labeling the leaf of an open branch can be extended to a model of the formula labeling the root. There are four steps in the proof:

1. Define a property of sets of formulas;
2. Show that the union of the formulas labeling nodes in an open branch has this property;
3. Prove that any set having this property is satisfiable;
4. Note that the formula labeling the root is in the set.

Definition 2.75 Let U be a set of formulas. U is a *Hintikka set* iff:

1. For all atoms p appearing in a formula of U, either $p \notin U$ or $\neg p \notin U$.
2. If $A \in U$ is an α-formula, then $A_1 \in U$ *and* $A_2 \in U$.
3. If $B \in U$ is a β-formula, then $B_1 \in U$ *or* $B_2 \in U$. ■

Example 2.76 U, the union of the set of formulas labeling the nodes in the open branch of Example 2.74, is $\{p,\ p \vee (q \wedge \neg q)\}$. We claim that U is a Hintikka set. Condition (1) obviously holds since there is only one literal p in U and $\neg p \notin U$. Condition (2) is vacuous. For Condition (3), $B = p \vee (q \wedge \neg q) \in U$ is a β-formula and $B_1 = p \in U$. ■

Condition (1) requires that a Hintikka set not contain a complementary pair of literals, which to be expected on an open branch of a tableau. Conditions (2) and (3) ensure that U is *downward saturated*, that is, U contains sufficient subformulas so that the decomposition of the formula to be satisfied will not take us out of U. In turn, this ensures that an interpretation defined by the set of literals in U will make all formulas in U true.

The second step of the proof of completeness is to show that the set of formulas labeling the nodes in an open branch is a Hintikka set.

Theorem 2.77 *Let l be an open leaf in a completed tableau $\mathscr{T}$. Let $U = \bigcup_i U(i)$, where i runs over the set of nodes on the branch from the root to l. Then U is a Hintikka set.*

Proof In the construction of the semantic tableau, there are no rules for decomposing a literal p or $\neg p$. Thus if a literal p or $\neg p$ appears for the first time in $U(n)$ for some n, the literal will be copied into $U(k)$ for all nodes k on the branch from n to l, in particular, $p \in U(l)$ or $\neg p \in U(l)$. This means that all literals in U appear in $U(l)$. Since the branch is open, no complementary pair of literals appears in $U(l)$, so Condition (1) holds for U.

Suppose that $A \in U$ is an α-formula. Since the tableau is completed, A was the formula selected for decomposing at some node n in the branch from the root to l. Then $\{A_1, A_2\} \subseteq U(n') \subseteq U$, so Condition (2) holds.

Suppose that $B \in U$ is an β-formula. Since the tableau is completed, B was the formula selected for decomposing at some node n in the branch from the root to l. Then either $B_1 \in U(n') \subseteq U$ or $B_2 \in U(n') \subseteq U$, so Condition (3) holds. ■

The third step of the proof is to show that a Hintikka set is satisfiable.

Theorem 2.78 (Hintikka's Lemma) *Let U be a Hintikka set. Then U is satisfiable.*

Proof We define an interpretation and then show that the interpretation is a model of U. Let $\mathscr{P}_U$ be set of all atoms appearing in all formulas of U. Define an interpretation $\mathscr{I} : \mathscr{P}_U \mapsto \{T, F\}$ as follows:

$$\begin{array}{ll} \mathscr{I}(p) = T & \text{if } p \in U, \\ \mathscr{I}(p) = F & \text{if } \neg p \in U, \\ \mathscr{I}(p) = T & \text{if } p \notin U \text{ and } \neg p \notin U. \end{array}$$

Since U is a Hintikka set, by Condition (1) $\mathscr{I}$ is well-defined, that is, every atom in $\mathscr{P}_U$ is given exactly one value. Example 2.74 demonstrates the third case: the atom q appears in a *formula* of U so $q \in \mathscr{P}_U$, but neither the literal q nor its complement $\neg q$ appear in U. The atom is arbitrarily mapped to the truth value T.

We show by structural induction that for any $A \in U$, $v_{\mathscr{I}}(A) = T$.

- If A is an atom p, then $v_{\mathscr{I}}(A) = v_{\mathscr{I}}(p) = \mathscr{I}(p) = T$ since $p \in U$.
- If A is a negated atom $\neg p$, then since $\neg p \in U$, $\mathscr{I}(p) = F$, so $v_{\mathscr{I}}(A) = v_{\mathscr{I}}(\neg p) = T$.
- If A is an α-formula, by Condition (2) $A_1 \in U$ and $A_2 \in U$. By the inductive hypothesis, $v_{\mathscr{I}}(A_1) = v_{\mathscr{I}}(A_2) = T$, so $v_{\mathscr{I}}(A) = T$ by definition of the conjunctive operators.
- If A is β-formula B, by Condition (3) $B_1 \in U$ or $B_2 \in U$. By the inductive hypothesis, either $v_{\mathscr{I}}(B_1) = T$ or $v_{\mathscr{I}}(B_2) = T$, so $v_{\mathscr{I}}(A) = v_{\mathscr{I}}(B) = T$ by definition of the disjunctive operators. ■

Proof of Completeness Let $\mathscr{T}$ be a completed open tableau for A. Then U, the union of the labels of the nodes on an open branch, is a Hintikka set by Theorem 2.77. Theorem 2.78 shows an interpretation $\mathscr{I}$ can be found such that U is simultaneously satisfiable in $\mathscr{I}$. A, the formula labeling the root, is an element of U so $\mathscr{I}$ is a model of A. ■

2.8 Summary

The presentation of propositional logic was carried out in a manner that we will use for all systems of logic. First, the syntax of formulas is given. The formulas are defined as trees, which avoids ambiguity and simplifies the description of structural induction.

The second step is to define the semantics of formulas. An interpretation is a mapping of atomic propositions to the values $\{T, F\}$. An interpretation is used to give a truth value to any formula by induction on the structure of the formula, starting from atoms and proceeding to more complex formulas using the definitions of the Boolean operators.

A formula is satisfiable iff it is true in *some* interpretation and it is valid iff is true in *all* interpretations. Two formulas whose values are the same in all interpretations are logically equivalent and can be substituted for each other. This can be used to show that for any formula, there exists a logically equivalent formula that uses only negation and either conjunction or disjunction.

While truth tables can be used as a decision procedure for the satisfiability or validity of formulas of propositional logic, semantic tableaux are usually much more efficient. In a semantic tableau, a tree is constructed during a search for a model of a formula; the construction is based upon the structure of the formula. A semantic tableau is closed if the formula is unsatisfiable and open if it is satisfiable.

We proved that the algorithm for semantic tableaux is sound and complete as a decision procedure for satisfiability. This theorem connects the syntactical aspect of a formula that guides the construction of the tableau with its meaning. The central concept in the proof is that of a Hintikka set, which gives conditions that ensure that a model can be found for a set of formulas.

2.9 Further Reading

The presentation of semantic tableaux follows that of Smullyan (1968) although he uses analytic tableaux. Advanced textbooks that also use tableaux are Nerode and Shore (1997) and Fitting (1996).

2.10 Exercises

2.1 Draw formation trees and construct truth tables for

$$(p \rightarrow (q \rightarrow r)) \rightarrow ((p \rightarrow q) \rightarrow (p \rightarrow r)),$$

$$(p \rightarrow q) \rightarrow p,$$

$$((p \rightarrow q) \rightarrow p) \rightarrow p.$$

2.2 Prove that there is a unique formation tree for every derivation tree.

2.3 Prove the following logical equivalences:

$$A \wedge (B \vee C) \equiv (A \wedge B) \vee (A \wedge C),$$

$$A \vee B \equiv \neg(\neg A \wedge \neg B),$$

$$A \wedge B \equiv \neg(\neg A \vee \neg B),$$

$$A \rightarrow B \equiv \neg A \vee B,$$

$$A \rightarrow B \equiv \neg(A \wedge \neg B).$$

2.4 Prove $((A \oplus B) \oplus B) \equiv A$ and $((A \leftrightarrow B) \leftrightarrow B) \equiv A$.

2.5 Simplify $A \wedge (A \vee B)$ and $A \vee (A \wedge B)$.

2.6 Prove the following logical equivalences using truth tables, semantic tableaux or Venn diagrams:

$$\begin{aligned} A \rightarrow B &\equiv A \leftrightarrow (A \wedge B), \\ A \rightarrow B &\equiv B \leftrightarrow (A \vee B), \\ A \wedge B &\equiv (A \leftrightarrow B) \leftrightarrow (A \vee B), \\ A \leftrightarrow B &\equiv (A \vee B) \rightarrow (A \wedge B). \end{aligned}$$

2.7 Prove $\models (A \rightarrow B) \vee (B \rightarrow C)$.

2.8 Prove or disprove:

$$\models ((A \rightarrow B) \rightarrow B) \rightarrow B,$$

$$\models (A \leftrightarrow B) \leftrightarrow (A \leftrightarrow (B \leftrightarrow A)).$$

2.9 Prove:

$$\models ((A \wedge B) \rightarrow C) \rightarrow ((A \rightarrow C) \vee (B \rightarrow C)).$$

This formula may seem strange since it could be misinterpreted as saying that if C follows from $A \wedge B$, then it follows from one or the other of A or B. To clarify this, show that:

$$\{A \wedge B \rightarrow C\} \models (A \rightarrow C) \vee (B \rightarrow C),$$

but:

$$\{A \wedge B \rightarrow C\} \not\models A \rightarrow C,$$

$$\{A \wedge B \rightarrow C\} \not\models B \rightarrow C.$$

2.10 Complete the proof that $\uparrow$ and $\downarrow$ can each define all unary and binary Boolean operators (Theorem 2.37).

2.11 Prove that $\wedge$ and $\vee$ cannot define all Boolean operators.

2.12 Prove that $\{\neg, \leftrightarrow\}$ *cannot* define all Boolean operators.

2.13 Prove that $\uparrow$ and $\downarrow$ are not associative.

2.14 Prove that if U is satisfiable then $U \cup \{B\}$ is not necessarily satisfiable.

2.15 Prove Theorems 2.44–2.47 on the satisfiability of sets of formulas.

2.16 Prove Theorems 2.50–2.54 on logical consequence.

2.17 Prove that for a set of axioms U, $\mathscr{T}(U)$ is closed under logical consequence (see Definition 2.55).

2.18 Complete the proof that the construction of a semantic tableau terminates (Theorem 2.66).

2.19 Prove that the method of semantic tableaux remains sound and complete if a tableau can be closed non-atomically.

2.20 Manna (1974) Let *ifte* be a *tertiary* (3-place) operator defined by:

A	B	C	$\mathit{ifte}(A, B, C)$
T	T	T	T
T	T	F	T
T	F	T	F
T	F	F	F
F	T	T	T
F	T	F	F
F	F	T	T
F	F	F	F

The operator can be defined using infix notation as:

$$\mathit{if}\ A\ \mathit{then}\ B\ \mathit{else}\ C.$$

1. Prove that *if then else* by itself forms an adequate sets of operators if the use of the constant formulas *true* and *false* is allowed.
2. Prove: $\models \mathit{if}\ A\ \mathit{then}\ B\ \mathit{else}\ C \equiv (A \rightarrow B) \wedge (\neg A \rightarrow C)$.
3. Add a rule for the operator *if then else* to the algorithm for semantic tableaux.

References

M. Fitting. *First-Order Logic and Automated Theorem Proving* (*Second Edition*). Springer, 1996.

J.E. Hopcroft, R. Motwani, and J.D. Ullman. *Introduction to Automata Theory, Languages and Computation* (*Third Edition*). Addison-Wesley, 2006.

Z. Manna. *Mathematical Theory of Computation*. McGraw-Hill, New York, NY, 1974. Reprinted by Dover, 2003.

A. Nerode and R.A. Shore. *Logic for Applications* (*Second Edition*). Springer, 1997.

R.M. Smullyan. *First-Order Logic*. Springer-Verlag, 1968. Reprinted by Dover, 1995.