Chapter 2: Propositional Logic: Syntax, Semantics

This chapter concerns syntax and semantics of propositional logic. In principle, these subjects have been seen in ECE 108. However, the chapter will cover some topics that have yet to be seen. It will also serve as a prototype for the extension to predicate logic and temporal logic.

2.1.1 Formulas as trees

Definition 2.1 The symbols used to construct propositional formulas are:

- A countably infinite set of symbols $\mathcal{P}$ of *atomic propositions* (often called *atoms*), typically denoted by lower-case letters $p, q, r, \ldots$, possibly with subscripts.
- Boolean operators:

  negation $\neg$

  disjunction $\vee$

  conjunction $\wedge$

  implication $\Longrightarrow$

  equivalence $\Longleftrightarrow$

  nor $\downarrow$

  nand $\uparrow$.

  Negation is a *unary* operator, while the others are *binary*.

□

In ECE 108, formulas were defined as strings of symbols. But it's often convenient to display the grammatical structure of a formula in the form of a so-called 'parse tree.' In this course, we will actually define formulas as such trees, and then derive string representations from the trees.

Definition 2.2 The *well-formed formulas*, or *wffs*, or simply, *formulas* of propositional logic can be defined recursively as trees:

- a leaf labelled by an atomic proposition is a formula;
- a node labelled by $\neg$ with a single child that is a formula is itself a formula;
- a node labelled by one of the binary operators with two children, both of which are formulas, is itself a formula;
- and the set $\mathscr{F}$ of all formulas is the smallest that satisfies the above properties.

□

The above is a minor variation on the definition in the text. In particular, the last bullet is often omitted.

Question: Is an "infinite conjunction," $p \wedge (q \wedge (r \wedge \ldots$ a formula?

Here are two examples of formulas:

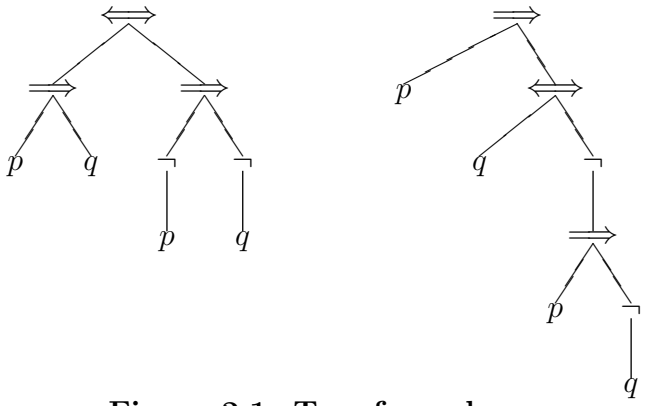


Figure 2.1: Two formulas.

Formulas as strings

A string representation of a formula can be obtained by inorder traversal of the tree:

Algorithm 2.4: Represent a formula A as a string.

Input: A formula A.

Output: A string representation of A.

Call the recursive procedure `Inorder(A)` recursively:

```
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 labelled by the unary operator $\neg$, the left subtree is considered to be the empty tree, and the step `Inorder(F1)` is skipped.)

As it stands, the above algorithm introduces some ambiguity, as the following example shows.

Example 2.6 Consider the string representations of the two formulas of Figure 2.1.

During an inorder traversal of the left-hand tree, we first write down p, then $\Longrightarrow$, then q, then $\Longleftrightarrow$, and so on, obtaining

$$p \Longrightarrow q \Longleftrightarrow \neg p \Longrightarrow \neg q \, .$$

Traversing the right-hand formula, we write p, then $\Longrightarrow$, then q, then $\Longleftrightarrow$, and so on, getting

$$p \Longrightarrow q \Longleftrightarrow \neg p \Longrightarrow \neg q \, .$$

□

2.1.3 Resolving ambiguity in the string representation

There are (at least) three means of resolving this ambiguity: parentheses, precedence, and Polish notation.

Parentheses

Algorithm 2.7: Represent a formula as a string containing parentheses.

Input: A formula A.

Output: A string representation of A.

Call the recursive procedure `Inorder(A)` recursively:

```
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 labelled by the unary operator $\neg$, the left subtree is considered to be the empty tree, and the step `Inorder(F1)` is skipped.)

Now the two formulas of Figure 2.1 yield different string representations, respectively:

$$((p \implies q) \iff ((\neg p) \implies (\neg q))) \text{ ; and}$$
$$(p \implies (q \iff (\neg(p \implies (\neg q))))) .$$

The use of parentheses resolves the ambiguity of the string representation, but it can clutter the notation.

Precedence

A second way of resolving the ambiguity is to define *precedence* and *associativity* conventions.

Suppose that the order of precedence of operations (from high to low) is as follows:

$\neg$

$\wedge, \uparrow$

$\vee, \downarrow$

$\Longrightarrow$

$\Longleftrightarrow, \oplus$

Suppose also that operators 'associate to the right': for example, $a \downarrow b \downarrow c$ means $a \downarrow (b \downarrow c)$.

If a different order of application of the operators is intended, then parentheses can be used. (In the previous example, note that, because nor is not associative, $(a \downarrow b) \downarrow c$ generally has a different truth value from $a \downarrow (b \downarrow c)$.)

Polish notation

So called because it was developed by Polish logicians,[1] this string representation is based an a preorder traversal of the tree:

[1]Actually, it was developed by one Polish logician, Jan Łukasiewicz. But rather than attempt to pronounce his name, people simply called it Polish notation.

Algorithm 2.8: Represent a formula A as a string in Polish notation.

Input: A formula A.

Output: A string representation of A.

```
Preorder(F)
        Write the label of the root of F
        If F is a leaf
             write its label
             return
        let F1 and F2 be the left and right subtrees of F
        Preorder(F1)
        Preorder{F2}
```

(If the root of `F` is labelled by the unary operator $\neg$, the left subtree is considered to be the empty tree, and the step `Preorder(F1)` is skipped.)

Example 2.9: In Polish notation, the two formulas of Figure 2.1 are respectively:

$$\iff \implies p\ q \implies \neg\ p \neg\ q \text{ , and}$$
$$\implies p \iff q \neg \implies p \neg\ q\ .$$

□

These representations are unambiguous.

In fact, this notation allows for efficient transcription of a formula by means of a stack. Rewrite the second formula in reverse order – this is called *reverse Polish notation*:

$$q \neg\ p \implies \neg\ q \iff p \implies \quad .$$

To find the tree representation of the formula, write, for instance,

`push p` to mean, "push onto the stack the tree consisting of a leaf labelled by p," and

`Implies` to mean,

> "pop the two subtrees `F1` and `F2` from the top of the stack, and then
>
> push onto the stack the tree whose root is labelled $\Longrightarrow$, and whose left subtree is `F1` and whose right subtree is `F2`."

Then the tree for the whole formula is found by performing similar operations in the linear order of the reverse Polish notation:

```
push q
Negate
push p
Implies
Negate
push q
Equiv
push p
Implies
```

For example, after the first three operations, the stack contains the tree representation of p at the top, and that of $\neg q$ below. After the following operation, the tree representation of $p \implies \neg q$ (in conventional, infix notation) is at the top of the stack, and so forth.

In fact, Polish notation, or reverse Polish notation (RPN), gives an unambiguous representation of any expressions made up of operators and operands, provided the operators have fixed *arity* (a fixed number of arguments).

RPN is used in programming languages such as Forth and PostScript, and in Hewlett-Packard calculators (for arithmetic expressions).

The algorithm for evaluating expressions in RPN is always the same: when you encounter an operand, push it onto the stack; when you encounter an operator, pop the requisite sequence of operands from the stack, apply the operator to them, and push the result.

Formulas as strings (conclusion)

Under any of the above conventions, the process of converting a formula to a string can be unambiguously reversed, yielding the 'underlying' tree representation:

Definition 2.5 (really terminology): An unambiguous string representation of a formula will also be referred to as the formula.

□

2.1.4 Structural induction

- We performed mathematical induction on the natural numbers in ECE 108, and also saw examples of other kinds of induction.
- Generally, induction is a means of extending an argument from 'simple' cases to more complex ones; or from special cases to more general ones.
- In this course we'll perform induction on the structure of formulas: starting from the base case of leaves, we'll generalize the proof by means of an induction step that considers formulas obtained from simpler ones by applying logical operators.
- This can be thought of as induction on the 'height' of trees.

Theorem 2.12 (Structural induction): Given a property of formulas and a formula $A \in \mathscr{F}$, suppose that,

1. the property holds for all atoms p in A;
2. for any subformula of A of the form $\neg F$, if the property holds for F, then it also holds for $\neg F$; and
3. for any subformula of A of the form F_1 *op* F_2, if the property holds for F_1 and F_2, then it also holds for F_1 *op* F_2;

then the property holds for A.

Before proving the theorem, recursively define the *height* of a tree: If the tree consists only of a leaf, define its height to be zero. If the root of the tree is labelled by a logical operator, define its height to be one greater than that of the highest subtree rooted at one of its children.

Proof: Assume that the three assumptions are satisfied by A: note that this means that they are also satisfied by all subformulas of A. We shall show by induction on the height of the tree representation of A that the Theorem holds for A.

Base case: if the height of A is zero, then A is a leaf, so A satisfies the property, by assumption 1. Therefore, the Theorem holds for A.

Induction step: Suppose that height of A is $n > 0$, and that the Theorem holds for formulas of lesser height. Then A is either of the form $\neg F$ or of the form $F_1 \ op \ F_2$. In the first case, the height of F is $n - 1$, so the Theorem holds for F by inductive assumption. Given that its three assumptions hold for F, F must satisfy the property; but then, by assumption 2, $\neg F = A$ also satisfies the property, In the second case, the subformulas F_1 and F_2 are of height less than n, so the Theorem holds, by inductive hypothesis, for F_1 and F_2. Given that its three assumptions also hold for F_1 and F_2, both F_1 and F_2 satisfy the property; but then, by assumption 3, $F_1 op F_2 = A$ also satisfies the property. The Theorem therefore holds for A.

This completes the induction. □

The above is a stronger version of the induction principle given in the text: the latter only lets one prove properties that hold for all formulas.

Example: Consider A to be the left-hand formula of Figure 2.1, and use structural induction to show that it has the following property: an inorder traversal, without the addition of parentheses, gives a string representation of A that is correct under the standard rules of precedence.

To prove that the formula has the property, we show that the three assumptions of the theorem are satisfied. For this, denote by $SR(F)$ the string representation (obtained through inorder traversal, without the addition of parentheses) of a formula $F \in \mathcal{F}$.

1. For any atom p, $SR(p) = p$ is (trivially) a correct string representation.

2. If $\neg F$ is a subformula of A, then the subformula F must (in this particular case) be an atomic proposition; therefore $SN(\neg F) = \neg F$ is a correct string representation of $\neg F$.

3. If $F_1 op F_2$ is a subformula of A, then note, owing to the structure of A, that op is of strictly lower precedence than any operators in F_1 or F_2. It follows that, if $SR(F_1)$ and $SR(F_2)$ are correct string representations of F_1 and F_2, respectively, then $SR(F_1 op F_2) = SR(F_1) op SR(F_2)$ is a correct string representation of $F_1 op F_2$.

This completes the induction. □

2.1.5 and 2.16 – skipped.

2.2 Interpretations

The semantics – or meaning – of formulas will be formally defined in terms of *interpretations*.

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 $\mathscr{I}_A : \mathscr{P}_A \to \{T, F\}$ that assigns a truth value T or F to each atom in A.

Definition 2.16 Let $\mathscr{I}_A$ be an interpretation for $A \in \mathscr{F}$. Then $v_{\mathscr{I}_A}(A')$, the *truth value of subformula A' under $\mathscr{I}_A$* is defined inductively on the structure of A as shown in Figure 2.3 (where $v_{\mathscr{I}_A}(A')$ is abbreviated by $v(A')$):

A	$v(A_1)$	$v(A_2)$	$v(A)$
$\neg A_1$	T		F
$\neg A_1$	F		T
$A_1 \vee A_2$	F	F	F
$A_1 \vee A_2$	otherwise		T
$A_1 \wedge A_2$	T	T	T
$A_1 \wedge A_2$	otherwise		F
$A_1 \to A_2$	T	F	F
$A_1 \to A_2$	otherwise		T

A	$v(A_1)$	$v(A_2)$	$v(A)$
$A_1 \uparrow A_2$	T	T	F
$A_1 \uparrow A_2$	otherwise		T
$A_1 \downarrow A_2$	F	F	T
$A_1 \downarrow A_2$	otherwise		F
$A_1 \leftrightarrow A_2$	$v(A_1) = v(A_2)$		T
$A_1 \leftrightarrow A_2$	$v(A_1) \neq v(A_2)$		F
$A_1 \oplus A_2$	$v(A_1) \neq v(A_2)$		T
$A_1 \oplus A_2$	$v(A_1) = v(A_2)$		F

Figure 2.3: Truth values of formulas

Partial interpretations

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

Example 2.19 Consider the formula $A = p \wedge q$ and the partial interpretation that assigns F to p. The truth value of A is determined by this partial interpretation. □

2.2.2 Truth tables – skipped

This material should be familiar from ECE 108 and ECE 124.

2.2.3 Understanding the Boolean operators – skipped

This should also be familiar.

2.2.4 An interpretation for a set of formulas

Definition 2.2.4 Let $S = \{A_1,\ A_2.\ A_3,\ \dots\}$ be a set of formulas and let $\mathscr{P}_S = \bigcup_i \mathscr{P}_{A_i}$. An *interpretation* for S is a function $\mathscr{I}_S : \mathscr{P}_S \longrightarrow \{T, F\}$. 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.

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

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

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

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

Here, the subscript on the $\mathscr{I}$ has been dropped, as we may do whenever it won't lead to confusion.

□

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 $\mathscr{I}$ of $\{A_1, A_2\}$, then A_1 is said to be *logically equivalent* to A_2, denoted $A_1 \equiv A_2$.

Example 2.27

p	q	$v(p \vee q)$	$v(q \vee p)$
T	T	T	T
T	F	T	T
F	T	T	T
F	F	F	F

□

The above example generalizes 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. If $\mathscr{I}$ is an interpretation for $A_1 \vee A_2$, then it is an interpretation for $A_2 \vee A_1$, and vice versa, because the two formulas contain exactly the same atomic propositions. Either formula has the value T under $\mathscr{I}$ if and only if either $v_{\mathscr{I}}(A_1) = T$ or $v_{\mathscr{I}}(A_2) = T$. Because $\mathscr{I}$ is an arbitrary interpretation of the disjunctions, the result follows. □

Recall from ECE 108 that we say that $\mathscr{I}$ is an *arbitrary* interpretation of the disjunctions because nothing else has been assumed about it. That means that the above argument applies to *any* such interpretation $\mathscr{I}$, and therefore to *all* such interpretations.

2.3.1 The relationship between $\iff$ and $\equiv$

We have called the Boolean operator $\iff$ 'equivalence,' but it is not exactly the same thing as logical equivalence, as denoted by $\equiv$. The latter is not a Boolean operator, but rather a property of pairs of propositional formulas.

The former is part of the language of propositional logic; the latter a part of the *metalanguage* that we use to talk about that language.

But obviously, there is a close relationship between the two:

Theorem 2.29 $A_1 \equiv A_2$ if and only if $A_1 \iff A_2$ is true in every interpretation (of $\{A_1, A_2\}$).

Proof: Let $\mathscr{I}$ be an arbitrary interpretation of $\{A_1, A_2\}$. Then $v_{\mathscr{I}}(A_1) = v_{\mathscr{I}}(A_2)$ if and only if $v_{\mathscr{I}}(A_1 \iff A_2) = T$. Hence A_1 and A_2 are logically equivalent if and only if $A_1 \iff A_2$ is true in every interpretation of $\{A_1, A_2\}$. □

2.3.2 Substitution

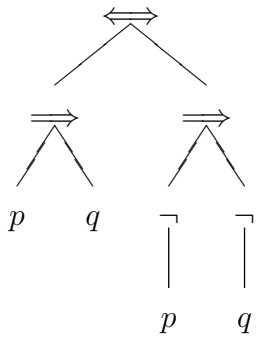
Logical equivalence justifies, on semantic grounds, the substitution of one *subformula* for another.

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

Example 2.31 Figure 2.4 shows the left-hand formula from Figure 2.1 and its proper subformulas. Represented as strings, the formula

$$(p \implies q) \iff (\neg p \implies \neg q)$$

contains the proper subformulas $p \implies q$, $\neg p \implies \neg q$, $\neg p$, $\neg q$, p, and q.



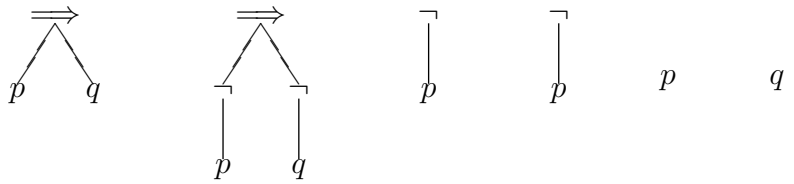


Figure 2.4 Subformulas.

□

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

Example 2.33 Let B be $(p \implies q) \iff (\neg p \implies \neg q)$, A be $p \implies q$, and A' be $\neg p \vee q$. Then $B\{A \leftarrow A'\}$ is

$$(\neg p \vee q) \iff (\neg p \implies \neg q) \ .$$

□

Note: A could be a subformula of A', as it would have been in the above example if A' had been $(p \vee \neg p) \implies (p \implies q)$. In such cases, we do not substitute for A repeatedly; that is why we used the word *simultaneously* in Definition 2.32 (though it's not included in the textbook).

In particular, if B were, say, $(p \implies q) \iff (\neg p \implies \neg(p \implies q))$, and you initially substituted only for the first occurrence of A, you would get

$$((p \vee \neg p) \implies (p \implies q)) \iff (\neg p \implies \neg(p \implies q))) \ .$$

You would then have to treat this formula with care, and substitute only for the *second* occurrence of A. This is why we stress that the substitution should be 'simultaneous.'

Substituting a logically equivalent formula for a subformula does not alter the truth value of a formula under any interpretation:

Theorem 2.34 (Substitution of a logically equivalent subformula) Let A be a subformula of B and let A' be a formula that is logically equivalent to A. Then $B \equiv B\{A \leftarrow A'\}$.

Before proving the theorem, recursively define the *depth* of a node in a tree: if the node is the root, its depth is zero; otherwise, the node appears in a specific subtree of the root, and its depth is defined to be one greater than its depth in that subtree.

Proof: Let $\mathscr{I}$ be an arbitrary interpretation for $\{B, B\{A \leftarrow A'\}\}$. Then the restriction of $\mathscr{I}$ to the set of atomic propositions occurring in A and A' is an interpretation of $\{A, A'\}$. Because A and A' are logically equivalent, they must both have the same truth values under the restriction, and therefore under $\mathscr{I}$.

But then it can be shown that, for any subformula $\tilde{B}$ of B, $\tilde{B}\{A \leftarrow A'\}$ has the same truth value as $\tilde{B}$ under $\mathscr{I}$.

The proof is by induction on the greatest depth d in $\tilde{B}$ of any occurrence of A. (If there is no such occurrence, the result holds vacuously.)

Base case: If $d = 0$, then $\tilde{B} = A$, and $\tilde{B}\{A \leftarrow A'\} = A'$, so the result holds.

Induction step: Now suppose that $d > 0$, and that the result holds for any subformula of B in which the greatest depth of any occurrence of A is less than d. The greatest depth of any occurrence of A in any proper subformula of $\tilde{B}$ is strictly less than d, so by inductive hypothesis, all such subformulas have the same truth values under $\mathscr{I}$ before and after substitution. Therefore, $\tilde{B}$ and $\tilde{B}\{A \leftarrow A'\}$ also have the same truth value under $\mathscr{I}$. This completes the induction.

Because $\mathscr{I}$ is an arbitrary interpretation for $\{B, B\{A \leftarrow A'\}\}$, it follows that $B \equiv B\{A \leftarrow A'\}$. □

2.3.3 (Common examples of) logically equivalent formulas

Substitution of logically equivalent formulas is done so frequently, that it is important to be familiar with common examples of it. This subsection lists those common examples; proofs are left as exercises.

Absorption of constants

Extend the language of propositional logic to include two 'constant' atomic propositions *true* and *false*. (Alternatively, one may use the respective symbols $\top$ and $\bot$.) We call them 'constant' atomic propositions because their

semantics are the same for all interpretations:

$$\mathscr{I}(\mathit{true}) = T \;\&\; \mathscr{I}(\mathit{false}) = F \ ,$$

for any interpretation $\mathscr{I}$.

These constant propositions could alternatively be considered to be abbreviations for the formulas $p \vee \neg p$ and $p \wedge \neg p$.

The appearance of a constant in a formula may mean that the formula 'collapses' in such a way that a binary operator is no longer needed; it may even mean that the truth value of the formula itself is a constant:

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

Collapsing can also occur if both operands of a binary operator are the same, or if one is the negation of the other:

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

Commutativity, associativity and distributivity

With the exception of implication, the Boolean operators are **commutative**:

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

Moreover, as we know from ECE 108, the direction of an implication can be reversed, if negations are added to yield the **contrapositive**:

$$A \implies B \equiv \neg B \implies \neg A \ .$$

Disjunction, conjunction, and exclusive or are **associative**:

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

However, implication, nor, and nand are not associative.

Disjunction and conjunction **distribute** over each other:

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

It will simplify some of our proofs if we can consider formulas to contain only a small, but 'adequate' set of operators. For this reason, it is convenient to express some operators in terms of others; another reason is that some algorithms require formulas to be written in a *normal form* – a particular form in which all formulas can be expressed:

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

Recall from ECE 108 and ECE 124 that the fifth and sixth of the above equivalences are called *De Morgan's laws*.

2.4 Sets of Boolean operators

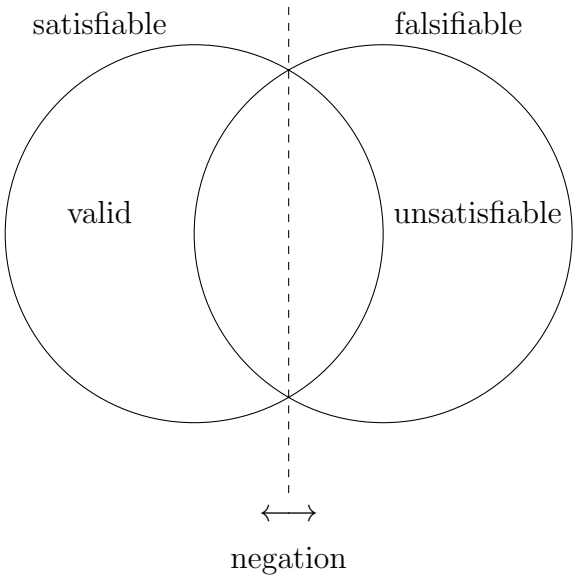
This material should be familiar from ECE 108 and ECE 124.

2.5 Satisfiability, Validity, and Consequence

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

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

The interrelationships among these concepts are displayed below:



As the dashed 'axis of reflection' is meant to suggest, the four semantical concepts are also related via negation:

Theorem 2.39 Let $A \in \mathscr{F}$. Then

- A is valid if and only if $\neg A$ is unsatisfiable; and
- A is satisfiable if and only if $\neg A$ is falsifiable.

Proof: Let $\mathscr{I}$ be an arbitrary interpretation for A (and therefore, for $\neg A$). Then

$$v_{\mathscr{I}}(A) = T \text{ if and only if } v_{\mathscr{I}}(\neg A) = F \ . \tag{1}$$

The formula A is valid if and only if the truth values in (1) hold for *all* $\mathscr{I}$, which is the case if and only if $\neg A$ is unsatisfiable; and A is satisfiable if and only if the truth values in (1) hold for *some* $\mathscr{I}$,which is the case if and only if $\neg A$ is falsifiable. □

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 *no* if $A \notin \mathscr{U}$.

The problem of finding a decision procedure is called a *decision problem*. Though Definition 2.40 is formulated specifically in terms of logic, decision problems arise in any area where computing is an important issue.
The study of the existence or nonexistence of decision procedures is *decidability theory*, and the computational complexity of decision procedures is one of the main topics of *computational complexity theory*.

- If $\mathscr{U}$ is the set of satisfiable formulas, the corresponding decision problem is referred to as Boolean satisfiability, or SAT. SAT is obviously a decidable problem: it suffices to construct a truth table to solve it. However, that takes exponential time in the number of atomic formulas.
- By Theorem 2.39, deciding validity can be reduced to deciding satisfiability: any $A \in \mathscr{F}$ is valid if and only if $\neg A$ is unsatisfiable. A procedure that checks unsatisfiability of $\neg A$ is called a *refutation procedure*.
- We'll shortly look at decision procedures for SAT that often work out to be more efficient than truth tables, but it is unlikely that any of them take less than exponential time in the worst case (if any deterministic polynomial-time algorithm exists, then $P = NP$).

2.5.2 Satisfiability of a set of formulas

Definition 2.42 A set of formulas $U = \{A_i : i \in I\}$ is *satisfiable* if there exists an interpretation $\mathscr{I}$ for U such that $v_{\mathscr{I}}(A_i) = T$ for all $i \in I$. Otherwise, U is *unsatisfiable*. A satisfying interpretation $\mathscr{I}$ is called a *model* of U.

Example 2.43 Let $U_1 = \{p, \neg\, p \vee q, q \wedge r\}$, and $U_2 = \{p, \neg p \vee q, \neg p\}$. Then U_1 is satisfiable. Every formula in U_2 is satisfiable, but the set itself is not. □

Theorem 2.44 If $U = \{A_i : i \in I\} \subseteq \mathscr{F}$ is satisfiable, then so is $U \setminus \{A_i\}$, for any $i \in I$.

Theorem 2.45 If $U \subseteq \mathscr{F}$ is satisfiable and $B \in \mathscr{F}$ is valid, then $U \cup \{B\}$ is satisfiable.

Theorem 2.46 If $U \subseteq \mathscr{F}$ is unsatisfiable, then, for any $B \in \mathscr{F}$, $U \cup \{B\}$ is unsatisfiable.

Theorem 2.47 If $U = \{A_i : i \in I\} \subseteq \mathscr{F}$ is unsatisfiable, and, for some $i \in I$, A_i is valid, then $U \setminus \{A_i\}$ is unsatisfiable.

The proofs of the above are left as exercises for the student.

2.5.3 Logical consequence

Definition 2.48 Let $U \subseteq \mathscr{F}$ and $A \in \mathscr{F}$. Then A is a *logical consquence* of U, denoted $U \models A$, if every model of U is a model of A.

If $U = \emptyset$, then every interpretation is vacuously a model of U. In that case, A is a logical consequence of U if and only if A is valid.

Theorem 2.53 If $U \models A$, then $U \cup U' \models A$, for any $U' \subseteq \mathscr{F}$.

Theorem 2.54 If $U \models A$, and $B \in \mathscr{F}$ is valid, then $U \setminus \{B\} \models A$.

2.5.4 Theories

– skipped.

2.6 Semantic tableaux

We'll skip this section.

Semantic tableaux represent an alternative means of deciding satisfiability (or validity) of propositional formulas. Tableaux can also be constructed for other logics (though not always yielding decision procedures). In the textbook, they are used to unify the treatments of different logics; and results on tableaux are leveraged to prove results about deductive systems. However, propositional resolution yields more efficient decision procedures for propositional logic, and we shall present stronger results on deductive systems than those that follow directly from tableau constructions. Moreover, Ben-Ari's use of tableaux leads him to define a deductive system for first-order logic that is unconventional, and lacks some desirable properties.

Chapter summary

- This treatment of propositional logic will serve as a prototype for the treatments of predicate and temporal logic.
- First, the syntax is given, with formulas defined unambiguously as trees.
- A method of induction on the structure of the syntax trees is defined ("structural induction").
- Then the semantics are defined: an interpretation assigns truth values to atoms, and the truth values of formulas are defined by induction on the structure of the formula.
- A formula is *satisfiable* if it is true in some interpretation and *valid* if it is true in all.
- Two formulas are *logically equivalent* if they have the same truth values in all interpretations; in that case they can be substituted for each other without changing the truth values of formulas.