Algorithm 1.3.4 There is a simple algorithm for verifying that a given string is a P-wff. Given a string $S = s_1 s_2 \dots s_n$ compute

$$w(s_n),\ w(s_n) + w(s_{n-1}),\ w(s_n) + w(s_{n-1}) + w(s_{n-2}), \dots,$$
$$w(s_n) + w(s_{n-1}) + \cdots + w(s_1) = w(S).$$

Then S is a P-wff exactly when all these sums are ≥ 1 and $w(S) = 1$.

The correctness of this algorithm is a homework problem (Assignment #1).

1.3.B Reverse Polish notation

We can also define, in an obvious way, the *Reverse Polish notation* by defining a new formal language in which rules (b), (c) in the definition of P-wff are changed to: $A \to A\neg$ and $A, B \to AB*$. We refer to grammatically correct formulas in this formal language as *Reverse Polish wff*, *RP-wff*, or *RP-formulas*.

Example 1.3.5 $st\neg \Rightarrow pq\neg s \wedge \vee\neg \Rightarrow$ is a RP-wff with parsing sequence

s, t, $t\neg$, $st\neg \Rightarrow$, p, q, $q\neg$, $q\neg s\wedge$, $pq\neg s \wedge \vee$, $pq\neg s \wedge \vee\neg$, $st\neg \Rightarrow pq\neg s \wedge \vee\neg \Rightarrow$.

(This corresponds to example 1.3.1.)

Exactly as before, we can prove a unique readability theorem and devise a recognition algorithm, simply by reversing everything. We will not do this explicitly. Instead, we will discuss algorithms for translating formulas from Reverse Polish notation to ordinary notation and vice versa.

Algorithm 1.3.6 (Translating a RP-wff to an ordinary formula) We will describe it by an example, from which it is not hard to formulate the general algorithm:

Consider

$$st\neg \Rightarrow pq\neg s \wedge \vee\neg \Rightarrow$$

Scan from left to right until the first connective is met. Here it is $\neg$. Replace $t\neg$ by $\neg t$:

$$s\neg t \Rightarrow pq\neg s \wedge \vee\neg \Rightarrow$$

Again scan from left to right until the first unused connective is met. Here it is $\Rightarrow$. Replace $s\neg t \Rightarrow$ by $(s \Rightarrow \neg t)$.

$$(s \Rightarrow \neg t)pq\neg s \wedge \vee\neg \Rightarrow$$

Scan from left to right until the first unused connective is met. Here it is $\neg$. Replace $q\neg$ by $\neg q$:

$$(s \Rightarrow \neg t)p\neg qs \wedge \vee\neg \Rightarrow$$

Scan from left to right until the first unused connective is met. Here it is $\wedge$. Replace $\neg qs\wedge$ by $(\neg q \wedge s)$:

$$(s \Rightarrow \neg t)p(\neg q \wedge s) \vee \neg \Rightarrow$$

Scan from left to right until the first unused connective is met. Here it is $\vee$. Replace $p(\neg q \wedge s)\vee$ by $(p \vee (\neg q \wedge s))$:

$$(s \Rightarrow \neg t)(p \vee (\neg q \wedge s))\neg \Rightarrow$$

Scan from left to right until the first unused connective is met. Here it is $\neg$. Replace $(p \vee (\neg q \wedge s))\neg$ by $\neg(p \vee (\neg q \wedge s))$:

$$(s \Rightarrow \neg t)\neg(p \vee (\neg q \wedge s)) \Rightarrow$$

Scan from left to right until the first unused connective is met. Here it is $\Rightarrow$. Replace $(s \Rightarrow \neg t)\neg(p \vee (\neg q \wedge s)) \Rightarrow$ by

$$((s \Rightarrow \neg t) \Rightarrow \neg(p \vee (\neg q \wedge s))).$$

This is the final result.

If A is a RP-wff, denote by $O(A)$ the wff obtained in this fashion. Then it is not hard to see that $O(A)$ satisfies the following recursive definition:

(i) $O(p) = p$

(ii) $O(A\neg) = \neg O(A)$

(iii) $O(AB*) = (O(A) * O(B))$.

Algorithm 1.3.7 (Translating a wff to a RP-wff) The following analogous recursive definition will translate a wff A to a RP-wff $RP(A)$:

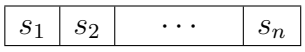
(i) $RP(p) = p$

(ii) $RP(\neg A) = RP(A)\neg$

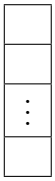
(iii) $RP((A * B)) = RP(A)RP(B)*$.

It is not hard to see by induction that these processes are inverses of each other: i.e., for each RP-wff A, $RP(O(A)) = A$ and for each wff B $O(RP(B)) = B$.

We will now describe a way of implementing the recursive definition $A \mapsto RP(A)$ by a "stack algorithm". Visualize the given wff A as being the *input string*:

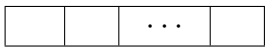


We also have a *stack string*:



This is initially empty and at various stages of the algorithm we will either put something on the top of the string or else remove something from the top of the string.

Finally we will have an *output string* which will eventually be $RP(A)$. In the beginning it is also empty:



Before we describe the algorithm, let us notice that if A is a wff, say $A = s_1 s_2 \ldots s_n$, and for some $1 \leq i < n$, $s_i = \neg$ but $s_{i-1} \neq \neg$, then there is a unique wff $E = s_i \ldots s_m$, $m \leq n$, of the form $E = \neg \ldots \neg p$ or $E = \neg\neg \ldots \neg(C * D)$. This is because any wff is of the form $p, \neg \ldots \neg p$, $\neg\neg \ldots \neg(C * D)$ or $(C * D)$. Notice that in case $E = \neg\neg \ldots \neg(C * D)$, we can find the end of E by starting from the beginning $\neg$ until left and right parentheses balance.

We now describe the algorithm:

First scan A from left to right and whenever a $\neg$ is met which is not preceded by $\neg$ and it starts a wff of the form $E = \underbrace{\neg \ldots \neg}_{n}(C * D)$ or $E = \underbrace{\neg\neg \ldots \neg}_{n} p$ as above, add to the right of it n copies of a new symbol, say $\sqsupset$,

to obtain $E\underbrace{\sqsupset\sqsupset\cdots\sqsupset}_{n}$. After this was done for all such occurrences of $\neg$ in A we obtain a new string, say A' (which might contain many occurrences of this new symbol $\sqsupset$).

We now start scanning A' from left to right. If we see (do nothing. If we see p, we add p to the right of the output string. If we see $\neg$ or $* \in \{\wedge, \vee, \Rightarrow, \Leftrightarrow\}$, we add this symbol to the top of the stack. If we see) or $\sqsupset$ we remove the top symbol in the stack and add it to the right of the output string. When we have scanned all of A' this process stops and the output string is $RP(A)$.

Example 1.3.8 Consider the wff

$$((s \Rightarrow \neg t) \Rightarrow \neg(p \vee \neg(\neg q \wedge s))) = A$$

for which the RP-wff is

$$st\neg \Rightarrow pq\neg s \wedge \neg \vee \neg \Rightarrow = \ RP(A)$$

.

First we form A':

$$((s \Rightarrow \neg t \sqsupset) \Rightarrow \neg(p \vee \neg(\neg q \sqsupset \wedge s) \sqsupset) \sqsupset)$$

In the stack below, we represent top to bottom as right to left. Therefore the top of the stack, where we push and pop symbols, is on the right.

input	stack	output
(	$-$	$-$
(	$-$	$-$
s	$-$	s
$\Rightarrow$	$\Rightarrow$	s
$\neg$	$\Rightarrow \neg$	s
t	$\Rightarrow \neg$	st
$\sqsupset$	$\Rightarrow$	$st\neg$
)	$-$	$st\neg \Rightarrow$
$\Rightarrow$	$\Rightarrow$	$st\neg \Rightarrow$
$\neg$	$\Rightarrow \neg$	$st\neg \Rightarrow$

input	stack	output
$($	$\Rightarrow \neg$	$st\neg \Rightarrow$
p	$\Rightarrow \neg$	$st\neg \Rightarrow p$
$\vee$	$\Rightarrow \neg\vee$	$st\neg \Rightarrow p$
$\neg$	$\Rightarrow \neg \vee \neg$	$st\neg \Rightarrow p$
$($	$\Rightarrow \neg \vee \neg$	$st\neg \Rightarrow p$
$\neg$	$\Rightarrow \neg \vee \neg\neg$	$st\neg \Rightarrow p$
q	$\Rightarrow \neg \vee \neg\neg$	$st\neg \Rightarrow pq$
$\sqsupset$	$\Rightarrow \neg \vee \neg$	$st\neg \Rightarrow pq\neg$
$\wedge$	$\Rightarrow \neg \vee \neg\wedge$	$st\neg \Rightarrow pq\neg$
s	$\Rightarrow \neg \vee \neg\wedge$	$st\neg \Rightarrow pq\neg s$
$)$	$\Rightarrow \neg \vee \neg$	$st\neg \Rightarrow pq\neg s\wedge$
$\sqsupset$	$\Rightarrow \neg\vee$	$st\neg \Rightarrow pq\neg s \wedge \neg$
$)$	$\Rightarrow \neg$	$st\neg \Rightarrow pq\neg s \wedge \neg\vee$
$\sqsupset$	$\Rightarrow$	$st\neg \Rightarrow pq\neg s \wedge \neg \vee \neg$
$)$	$-$	$st\neg \Rightarrow pq\neg s \wedge \neg \vee \neg \Rightarrow$

We will now verify the correctness of this algorithm, by induction on the construction of formulas. We will show that if we start with A and then construct A' and apply this stack algorithm but with the stack containing some string S (not necessarily empty) and the output some string T (not necessarily empty), then we finish with the stack string S and output string $T\hat{}RP(A)$, where $\hat{}$ means concatenation.

This is obvious if $A = p$. Assume it holds for B and consider $A = \neg B$. Then notice that $A' = \neg B' \sqsupset$. Starting the algorithm from A' and stack string S and output string T, we first add $\neg$ to the top of S. Then we perform the algorithm on B' and stack string $S\neg$, output T. This finishes by producing stack string $S\neg$ and output $T\hat{}RP(B)$. Then since the last symbol of A' is $\sqsupset$, we get stack string S and output string $T\hat{}RP(B)\neg = T\hat{}RP(A)$.

Finally assume it holds for B, C and consider $A = (B * C)$. Then $A' = (B' * C')$. Starting the algorithm from A' and stack string S, output T, we first have $($, which does nothing. Then we have the algorithm on B' on stack string S, output T which produces $T\hat{}RP(B)$ and stack string S. Then $*$ is put on top of the stack and the algorithm is applied to C' with stack string $S*$ and output $T\hat{}RP(B)$ to produce stack string $S*$ and output

$T\hat{}RP(B)\hat{}RP(C)$. Finally we have) which produces stack string S and output $T\hat{}RP(B)\hat{}RP(C)* = T\hat{}RP(A)$. $\dashv$

1.4 Abbreviations

Sometimes in practice, in order to avoid complicated notation, we adopt various abbreviations in writing wff, if they don't cause any confusion.

For example, we usually omit outside parentheses: We often write $A \wedge B$ instead of $(A \wedge B)$ or $(A \wedge B) \vee C$ instead of $((A \wedge B) \vee C)$, etc.

Also we adopt an order of preference between the binary connectives, namely

$$\wedge, \vee, \Rightarrow, \Leftrightarrow$$

where connectives from left to right bind closer, i.e., $\wedge$ binds closer than $\vee, \Rightarrow, \Leftrightarrow$; $\vee$ binds closer than $\Rightarrow, \Leftrightarrow$; and $\Rightarrow$ binds closer than $\Leftrightarrow$. So if we write $p \wedge q \vee r$ we really mean $((p \wedge q) \vee r)$ and if we write $p \Rightarrow q \vee r$ we really mean $(p \Rightarrow (q \vee r))$, etc.

Finally, when we have repeated connectives, we always associate parentheses to the left, so that if we write $A \wedge B \wedge C$ we really mean $((A \wedge B) \wedge C)$, and if we write $A \Rightarrow B \Rightarrow C \Rightarrow D$ we really mean $(((A \Rightarrow B) \Rightarrow C) \Rightarrow D)$.

1.5 Semantics of Propositional Logic

We now want to ask the question, "When is a proposition true or false?" Specifically, if we know whether each atomic proposition is true or false, what is the truth value of a compound proposition? We will consider two truth values: T or 1 (true), and F or 0 (false).

1.5.A Valuations

Definition 1.5.1 A *truth assignment* or *valuation* consists of a map

$$\nu : \{p_1, p_2, \dots\} \to \{0, 1\}$$

assigning to each propositional variable a truth value.

Given such a ν we can extend it in a unique way to assign an associated truth value $\nu^*(A)$ to every wff A, by the following recursive definition:

(i) $\nu^*(p) = \nu(p)$.

(ii) $\nu^*(\neg A) = 1 - \nu^*(A)$.

(iii) For the binary connectives, the rules are:

$$\begin{aligned}
\nu^*((A \wedge B)) &= \begin{cases} 1 & \text{if } \nu^*(A) = \nu^*(B) = 1 \\ 0 & \text{otherwise} \end{cases} \\
&= \nu^*(A) \cdot \nu^*(B) \\
\nu^*((A \vee B)) &= \begin{cases} 1 & \text{if } \nu^*(A) = 1 \text{ or } \nu^*(B) = 1 \\ 0 & \text{if } \nu^*(A) = \nu^*(B) = 0 \end{cases} \\
&= 1 - (1 - \nu^*(A)) \cdot (1 - \nu^*(B)) \\
\nu^*((A \Rightarrow B)) &= \nu^*((\neg A \vee B)) \\
&= \begin{cases} 0 & \text{if } \nu^*(A) = 1,\ \nu^*(B) = 0, \\ 1 & \text{otherwise} \end{cases} \\
&= 1 - \nu^*(A) \cdot (1 - \nu^*(B)) \\
\nu^*((A \Leftrightarrow B)) &= \begin{cases} 1 & \text{if } \nu^*(A) = \nu^*(B) \\ 0 & \text{otherwise} \end{cases} \\
&= \nu^*(A) \cdot \nu^*(B) + (1 - \nu^*(A)) \cdot (1 - \nu^*(B))
\end{aligned}$$

For simplicity, we will write $\nu(A)$ instead of $\nu^*(A)$, if there is no danger of confusion.

These rules can be also represented by the following *truth tables*:

A	B	$\neg A$	$A \wedge B$	$A \vee B$	$A \Rightarrow B$	$A \Leftrightarrow B$
0	0	1	0	0	1	1
0	1	1	0	1	1	0
1	0	0	0	1	0	0
1	1	0	1	1	1	1

Note the similarity with the truth tables presented in section 1.1. We are now formalizing our intuitive sense of what the propositional connectives should "mean."

Definition 1.5.2 For each wff A, its *support*, $\mathrm{supp}(A)$, is the set of propositional variables occurring in it.

Example 1.5.3

$$\mathrm{supp}(\underbrace{((p \wedge q) \Rightarrow (\neg r \vee p))}_{A}) = \{p, q, r\}. \qquad (*)$$

Thus we have, by way of recursive definition,

$$\begin{aligned} \mathrm{supp}(p) &= \{p\} \\ \mathrm{supp}(\neg A) &= \mathrm{supp}(A) \\ \mathrm{supp}((A * B)) &= \mathrm{supp}(A) \cup \mathrm{supp}(B). \end{aligned}$$

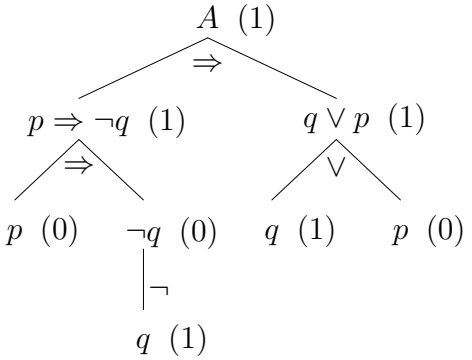
The following is easy to check:

Proposition 1.5.4 *If* $\mathrm{supp}(A) = \{q_1, \ldots, q_n\}$ *and* ν *is a valuation, then* $\nu(A)$ *depends only on* $\nu(q_1), \ldots, \nu(q_n)$*; i.e., if* μ *is a valuation, and* $\mu(q_i) = \nu(q_i)$ *for* $i = 1, \ldots, n$*, then* $\mu(A) = \nu(A)$*.*

For instance, in example 1.5.3 above, $\nu(A)$ depends only on $\nu(p)$, $\nu(q)$, and $\nu(r)$.

We can visualize the calculation of $\nu(A)$ in terms of the parse tree T_A as follows, where we illustrate the general idea by an example.

Example 1.5.5 Consider $A = ((p \Rightarrow \neg q) \Rightarrow (q \vee p))$ and its parse tree. Let $\nu(p) = 0,\ \nu(q) = 1$. Then we can find the truth value of each node of the tree working upwards. So $\nu(A) = 1$.



We also have the following algorithm for evaluating $\nu(A)$, given the valuation ν to all the propositional variables in the support of A.

Algorithm 1.5.6 First replace each propositional variable p in A by its truth value (according to ν), $\nu(p)$. Then scanning from left to right find the first occurence of a string of the form $\neg i,\ (i \wedge j),\ (i \vee j),\ (i \Rightarrow j),\ (i \Leftrightarrow j)$, where $i, j \in \{0, 1\}$, and replace it by its value according to the truth table (e.g., replace $\neg 0$ by 1, $(0 \wedge 1)$ by 0, etc.). Repeat the process until the value $\nu(A)$ is obtained.

Example 1.5.7 A, ν as in example 1.5.5. Then we first obtain $((0 \Rightarrow \neg 1) \Rightarrow (1 \vee 0))$. Next we have successively $((0 \Rightarrow 0) \Rightarrow (1 \vee 0)),\ (1 \Rightarrow (1 \vee 0)),\ (1 \Rightarrow 1),\ 1 = \nu(A)$.

It is not hard to prove by induction on A that this algorithm is correct, i.e., produces always $\nu(A)$.

Given a valuation ν we can of course define in a similar way the truth value $\nu(A)$ for any P-wff or RP-wff A. We will now describe a stack algorithm for calculating the truth value for RP-wff.

Algorithm 1.5.8 We start with a RP-wff A and a valuation ν to all the propositional variables in the support of A. The stack is empty in the beginning.

We scan A from left to right. If we see a propositional variable p, we add to the top of the stack $\nu(p)$. If we see $\neg$, we replace the top of the stack a (which is either 0 or 1) by $1 - a$. If we see $* \in \{\wedge, \vee, \Rightarrow, \Leftrightarrow\}$, we replace the top two elements of the stack b (= top), a (= next to top) by the value given by the truth table for $*$ applied to the truth values a, b (in that order). The truth value in the stack, when we finish scanning A, is $\nu(A)$.

Example 1.5.9 Let A be the RP-wff

$$pqp\neg\vee \Rightarrow rs\neg u \Rightarrow \vee \Rightarrow .$$

(in ordinary notation this is $(p \Rightarrow (q \vee \neg p)) \Rightarrow (r \vee (\neg s \Rightarrow u))$ and $\nu(p) = 1,\ \nu(q) = 1,\ \nu(r) = 0,\ \nu(s) = 1,\ \nu(u) = 1$. Then by applying the algorithm we have (once again, top to bottom on the stack is represented as right to left):

input	stack
p	1
q	11
p	111
$\neg$	110
$\vee$	11
$\Rightarrow$	1
r	10
s	101
$\neg$	100
u	1001
$\Rightarrow$	101
$\vee$	11
$\Rightarrow$	1

Again it is easy to show by induction that this algorithm is correct, i.e., it always produces $\nu(A)$. The appropriate statement we must prove, by induction on the construction of A, is that if we start this algorithm with input a RP-wff A and stack S, we end up by having stack $S\nu(A)$. (Here S is a string of 0's and 1's.)

1.5.B Models and Tautologies

Definition 1.5.10 If ν is a valuation and A a wff, we say that ν *satisfies* or *models* A if $\nu(A) = 1$.

That is, ν models A if A is true under the truth values that ν assigns to the propositional variables. We use the notation

$$\nu \models A$$

to denote that ν models A. We also write

$$\nu \not\models A$$

if ν does not satisfy or model A, i.e., if $\nu(A) = 0$. Notice that

$$\begin{aligned}
\nu \models \neg A &\text{ iff } \nu \not\models A \\
\nu \models (A \wedge B) &\text{ iff } \nu \models A \text{ and } \nu \models B \\
\nu \models (A \vee B) &\text{ iff } \nu \models A \text{ or } \nu \models B \\
\nu \models (A \Rightarrow B) &\text{ iff either } \nu \not\models A \text{ or else } \nu \models B \\
\nu \models (A \Leftrightarrow B) &\text{ iff } (\nu \models A \text{ and } \nu \models B) \text{ or } (\nu \not\models A \text{ and } \nu \not\models B).
\end{aligned}$$

Definition 1.5.11 A wff A is a *tautology* (or *valid*) iff for *every* valuation ν we have $\nu \models A$. So A is a tautology if it is true independently of the truth assignments to its propositional variables.

Examples 1.5.12

(i) $\neg(A \wedge B) \Leftrightarrow (\neg A \vee \neg B)$ is a tautology. This is an expression of one of De Morgan's laws, which we first saw in section 1.1. The other "equivalences" we discussed there can also be expressed as tautological formulas: $A \Leftrightarrow \neg\neg A$ for the law of the double negative, and so on.

(ii) $A \vee (B \wedge C) \Leftrightarrow (A \vee B) \wedge (A \vee C)$ is a tautology. This is an equivalence we did not discuss: it says that $\vee$ *distributes over* $\wedge$. It is also true that $\wedge$ distributes over $\vee$, i.e. $A \wedge (B \vee C) \Leftrightarrow (A \wedge B) \vee (A \wedge C)$ is also a tautology.

(iii) $(p \Rightarrow q) \Rightarrow (q \Rightarrow p)$ is *not* a tautology, since it is falsified by the valuation $\nu(p) = 0$, $\nu(q) = 1$. We saw in section 1.1 that the truth of an implication is generally unrelated to the truth of its converse, but this formula claims that the converse is true whenever the original formula is true.

Definition 1.5.13 A wff A is *satisfiable* iff there is *some* valuation ν such that $\nu \models A$, i.e. A has at least one model. Otherwise, we say that A is *unsatisfiable* or *contradictory*.

Clearly A is contradictory iff $\neg A$ is a tautology, and vice versa.

Example 1.5.14 $(p \Rightarrow q)$ is satisfiable (e.g. by the valuation $\nu_1(p) = \nu_1(q) = 1$) and its negation is also satisfiable (by the valuation $\nu_2(p) = 1$, $\nu_2(q) = 0$).

To check whether a given formula is a tautology or not, we can write down its truth table and check that it gives always 1's, for any value of the propositional variables.

Example 1.5.15 Consider $(p \Rightarrow q) \Leftrightarrow (\neg q \Rightarrow \neg p)$.

p	q	$\neg p$	$\neg q$	$p \Rightarrow q$	$\neg q \Rightarrow \neg p$	$(p \Rightarrow q) \Leftrightarrow (\neg q \Rightarrow \neg p)$
0	1	1	0	1	1	1
0	0	1	1	1	1	1
1	1	0	0	1	1	1
1	0	0	1	0	0	1

This procedure requires 2^n many entries if the wff A contains n propositional variables, so it has "exponential complexity." No substantially more efficient algorithm is known. The possibility of improving the "exponential" to "polynomial" complexity is a famous open problem in theoretical computer science known as the "$\mathcal{P} = \mathcal{NP}$ problem", that we will discuss in Chapter 3.

We can extend the definition of satisfiability to sets containing more than one formula. Clearly the only interesting notion is that of *simultaneous* satisfiability; i.e. whether all formulas are satisfied by the same valuation.

Definition 1.5.16 If S is any set of wff (finite or infinite) and ν is a valuation, we say that ν *satisfies* or *models* S, in symbols

$$\nu \models S$$

if $\nu \models A$ for *every* $A \in S$, i.e., ν models *all* wff in S. If there is some valuation satisfying S we say that S is *satisfiable* (or has a *model*).

Examples 1.5.17

(i) $S = \{(p_1 \vee p_2),\ (\neg p_2 \vee \neg p_3),\ (p_3 \vee p_4),\ (\neg p_4 \vee \neg p_5), \dots\}$ is satisfiable, with model the valuation:

$$\nu(p_i) = \begin{cases} 1 & \text{if } i \text{ is odd;} \\ 0 & \text{if } i \text{ is even.} \end{cases}$$

(ii) $S = \{q \Rightarrow r,\ r \Rightarrow p,\ \neg p, q\}$ is *not* satisfiable.

(iii) If $S = \{A_1, \dots, A_n\}$ is finite, then S is satisfiable iff $A_1 \wedge \cdots \wedge A_n$ is satisfiable.

1.5.C Logical Implication and Equivalence

Definition 1.5.18 Let now S be any set of wff and A a wff. (View S as a set of hypotheses and A as a conclusion.) We say that S *(tauto)logically implies* A if every valuation that satisfies S satisfies also A (i.e., any model of S is also a model of A).

If S logically implies A, we write

$$S \models A$$

(and if it does not, $S \not\models A$). We use the same symbol as for models of a formula, because the notions are compatible. If $S = \emptyset$, we just write $\models A$. This simply means that A is a tautology (it is true in all valuations).

Examples 1.5.19

(i) (Modus Ponens) $\{A, A \Rightarrow B\} \models B$.

(ii) $\{A, A \Rightarrow (B \vee C),\ B \Rightarrow D,\ C \Rightarrow D\} \models D$.

(iii) $p \Rightarrow q \not\models q \Rightarrow p$

(iv) $\{p_1 \Rightarrow p_2,\ p_2 \Rightarrow p_3,\ p_3 \Rightarrow p_4, \dots\} \models p_1 \Rightarrow p_n$ (for any n)

(v) If $S = \{A_1, \dots, A_n\}$ is finite, then $S = \{A_1, \dots, A_n\} \models A$ iff $A_1 \wedge \cdots \wedge A_n \models A$ iff $\models (A_1 \wedge \cdots \wedge A_n) \Rightarrow A$.

(vi) $S \models (A \Rightarrow B)$ iff $S \cup \{A\} \models B$.

(vii) $S \models A$ iff $S \cup \{\neg A\}$ is *not* satisfiable.

Example 1.5.20 Consider the following argument (from Mendelson's book "Introduction to Mathematical Logic")

> If capital investment remains constant, then government spending will increase or unemployment will result. If government spending will not increase, taxes can be reduced. If taxes can be reduced and capital investment remains constant, then unemployment will not result. Hence, government spending will increase.

To see whether this conclusion logically follows from the premises, represent:

p : "capital investment remains constant"
q : "government spending will increase"
r : "unemployment will result"
s : "taxes can be reduced"

Then our premises are

$$S = \{p \Rightarrow (q \vee r),\ \neg q \Rightarrow s,\ s \wedge p \Rightarrow \neg r\}$$

Our conclusion is

$$A = q.$$

So we are asking if

$$S \models q.$$

This is false since the truth assignment $\nu(p) = 0, \nu(q) = 0, \nu(r) = 1, \nu(s) = 1$ makes S true but q false. So the argument above is not valid.

Remark. Note that if $S \subseteq \tilde{S}$ and $S \models A$, then $\tilde{S} \models A$. Also, if $S \models A$ for all $A \in S'$, and $S' \models B$, then $S \models B$.

Definition 1.5.21 Two wff A, B are called *(logically) equivalent*, in symbols

$$A \equiv B,$$

if $A \models B$ and $B \models A$, i.e. $A \Leftrightarrow B$ is a tautology.

Notice that logical equivalence is an equivalence relation, i.e.,

$$A \equiv A$$
$$A \equiv B \text{ implies } B \equiv A$$
$$A \equiv B,\ B \equiv C \text{ imply } A \equiv C.$$

We consider equivalent wff as semantically indistinguishable. However, in general equivalent formulas are different syntactically. By that we mean that they are distinct strings of symbols, e.g. $(p \Rightarrow q)$ and $(\neg q \Rightarrow \neg p)$.

Once again, we see the same equivalences that we mentioned in section 1.1, such as $A \equiv \neg\neg A$ and $\neg(A \wedge B) \equiv (\neg A \vee \neg B)$. Here are a couple more:

Examples 1.5.22
(i) $(A \Leftrightarrow B) \equiv ((A \wedge B) \vee (\neg A \wedge \neg B))$

(ii) $((A \wedge B) \Rightarrow C) \equiv ((A \Rightarrow C) \vee (B \Rightarrow C))$

Recommendation. Read Appendix A, which lists various useful tautologies and equivalences. Verify that they are indeed tautologies and equivalences. You will often need to use some of them in homework assignments.

Here are some useful facts about logical equivalence.

(i) Let A be a wff containing the propositional variables $q_1, \ldots, q_n$, and let $A_1, \ldots, A_n$ be arbitrary wff. If A is a tautology, so is

$$A[q_1/A_1, q_2/A_2, \ldots, q_n/A_n],$$

the wff obtained by substituting q_i by A_i $(1 \leq i \leq n)$ in A.

(ii) If A is a wff, B a subformula of A and B' a wff such that $B \equiv B'$, and we obtain A' from A by substituting B by B', then $A \equiv A'$. This can be easily proved by induction on the construction of A. Thus we can freely substitute equivalent formulas (as parts of bigger formulas) without affecting the truth value.

Example. $A = (\neg(p \wedge q) \Rightarrow (q \vee r)), B = \neg(p \wedge q), B' = (\neg p \vee \neg q), A' = ((\neg p \vee \neg q) \Rightarrow (q \vee r))$.

The *De Morgan Laws*, which we have seen before, are the equivalences

$$\neg(p \wedge q) \equiv (\neg p \vee \neg q)$$
$$\neg(p \vee q) \equiv (\neg p \wedge \neg q)$$

By the first fact above, we can conclude that $\neg(A \wedge B) \equiv (\neg A \vee \neg B)$, $\neg(A \vee B) \equiv (\neg A \wedge \neg B)$ for any wff A, B. The De Morgan laws also have the following generalization:

Proposition 1.5.23 (General De Morgan Laws) *Let A be a wff containing only the connectives $\neg, \wedge, \vee$. Let A^* be obtained from A by substituting each propositional variable p occurring in A by $\neg p$ and replacing $\wedge$ by $\vee$ and $\vee$ by $\wedge$. Then*

$$\neg A \equiv A^*.$$

Example 1.5.24

$$\begin{aligned}\neg((p \wedge q) \vee (\neg r \wedge p)) &\equiv ((\neg p \vee \neg q) \wedge (\neg\neg r \vee \neg p)) \\ &\equiv ((\neg p \vee \neg q) \wedge (r \vee \neg p)), \text{ since } \neg\neg r \equiv r.\end{aligned}$$

Proof. The proof of the General De Morgan's Laws will be left for Homework Assignment #2.

Augustus de Morgan was born in 1806 in Madura, India, and died in 1871 in London. De Morgan lost the sight on his right eye as a boy. His family moved to England when he was 10. At 1823, he refused to take a theological test, and as a result Trinity College (Cambridge) only gave him a BA instead of an MA. In 1828 he was appointed Professor at the new University College (London); he was the first mathematics professor of the college. De Morgan was first to formally define *mathematical induction* (1838). He was a friend of Charles Babbage, the creator of the first "difference engine" (1819–22), a predecessor of the modern computer. This led to his interest in logic as an *algebra* of identities and to his introduction of what are now known as De Morgan's laws.

"A dry dogmatic pedant I fear is Mr. De Morgan, notwithstanding his unquestioned ability." Thomas Hirst.

1.6 Truth Functions

We can consider a formula (or even an equivalence class of logically equivalent formulas) to define a function. Its inputs are the truth values of the propositional variables in its support (that is, a valuation, or at least the relevant part of a valuation), and its output is the truth value of the formula (that is, whether the valuation models the formula). We can also go the other way: starting from any such function, we will construct a propositional formula which describes it.

1.6.A Truth Functions

Definition 1.6.1 For $n \geq 1$, an *n-ary truth function* (also called a *Boolean function*) is any map

$$f : \{0,1\}^n \to \{0,1\}.$$

By convention we also have two "0-ary" truth functions, namely the constants 0 and 1.

An n-ary truth function can be represented by a *truth table* as follows:

x_1	x_2	$\cdots$	x_n	$f(x_1, \ldots, x_n)$
0	0	$\cdots$	0	$f(0, \ldots, 0)$
$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$
i_1	i_2	$\cdots$	i_n	$f(i_1, \ldots, i_n)$
$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$
1	1	$\cdots$	1	$f(1, \ldots, 1)$

This table has 2^n rows, allowing for all possibilities for the vector $(i_1, \ldots, i_n)$, where each i_k is 0 or 1. Since for each of these 2^n rows the values of f are arbitrary (can be either 0 or 1), altogether there are 2^{2^n} possible n-ary truth functions. We will look at some or all of these, for some small values of n.

n = 0. As mentioned above, there are two "0-ary" truth functions: the binary constants 0 and 1.

n = 1. There are $2^2 = 4$ *unary* truth functions:

x	$f_1(x)$	$f_2(x)$	$f_3(x)$	$f_4(x)$
0	0	0	1	1
1	0	1	0	1

So $f_1(x) = 0$ (constant), $f_4(x) = 1$ (constant), $f_2(x) = x$ (identity), $f_3(x) = 1 - x$ (corresponds to negation).

n = 2. There are $2^{2^2} = 16$ *binary* truth functions. For some examples, notice that each binary connective (via its truth table) gives rise to a corresponding binary truth function. For example, $\wedge$ gives rise to $f(x, y) = x \wedge y = x \cdot y$ (i.e. multiplication modulo two). Conversely, we can view each binary truth function as defining a binary connective, so we have altogether 16 binary connectives. (In a similar way, we can also view n-ary truth functions as *n-ary connectives*.) We have seen the standard binary connectives: $\wedge, \vee, \Rightarrow, \Leftrightarrow$. Here are a few others that play an important role in various ways.

x	y	$x+y$
0	1	1
0	0	0
1	1	0
1	0	1

This corresponds to *binary addition*, i.e., addition in $\mathbb{Z}_2$ (the integers modulo 2). We see that $x+y = (x \wedge \neg y) \vee (y \wedge \neg x)$. This connective is called *exclusive or* and sometimes also *symmetric difference*.

x	y	$x \downarrow y$
0	0	1
0	1	0
1	0	0
1	1	0

Notice that $x \downarrow y = \neg(x \vee y)$. $\downarrow$ is called *nor*.

x	y	$x\|y$
0	0	1
0	1	1
1	0	1
1	1	0

Notice that $x|y = \neg(x \wedge y)$. $|$ is called *nand* or *Sheffer stroke*.

Remark. Notice that some of the 16 binary connectives are degenerate, i.e., depend on only one (or none) of the arguments, e.g.,

x	y	$f(x,y) = \neg x$
0	1	1
0	0	1
1	1	0
1	0	0

In fact, every 0-ary connective (i.e. constant) gives rise to a (constant) binary connective, and every unary connective gives rise to two (degenerate) binary connectives ($f(x,y) = g(x)$ and $f(x,y) = g(y)$). (These are not all distinct, of course, since some unary connectives are themselves degenerate.) More generally, if $n > m$, then any m-ary connective gives rise to degenerate n-ary connectives.

n = 3. There are $2^{2^3} = 256$ ternary connectives. These are too many to list, but an interesting example is the "if x then y, else z" connective, whose value is y if $x = 1$ and z if $x = 0$. Another one is the majority connective:

$$\mathrm{maj}(x, y, z) = \begin{cases} 1 & \text{if the majority of } x, y, z \text{ is } 1 \\ 0 & \text{otherwise} \end{cases}$$

For each wff A we indicate the fact that A contains only propositional variables among $p_1, \ldots, p_n$ by writing $A(p_1, \ldots, p_n)$. This simply means that $\mathrm{supp}(A) \subseteq \{p_1, \ldots p_n\}$. It does not mean that A contains *all* of the propositional variables $p_1, \ldots, p_n$.

Definition 1.6.2 For each wff $A(p_1, \ldots, p_n)$, we define an n-ary truth function $f_A^n : \{0,1\}^n \to \{0,1\}$ by

$$f_A^n(x_1, \ldots, x_n) = \text{ (the truth value of } A \text{ given by the valuation } \nu(p_i) = x_i).$$

In other words, f_A is the truth function corresponding to the truth table of the wff A.

Examples 1.6.3

(i) For any binary connective $*$, if $A = (p_1 * p_2)$, then f_A is the truth function f_* corresponding to $*$. Similarly, if $A = \neg p_1$, then $f_A = f_\neg$, and so on.

(ii) If $A = ((p_1 \wedge p_2) \vee (\neg p_1 \wedge p_3))$, then f_A is the truth function of the "if... then... else..." connective.

Remark. Strictly speaking, each wff A gives rise to infinitely many truth functions f_A^n: one for each $n \geq n_A$, where n_A is the least number m for which $\mathrm{supp}(A) \subseteq \{p_1, \ldots, p_m\}$. This is the same situation that we face when we have a polynomial, like $x + y$, which we can view as defining a function of two variables x, y, but also as a function of three variables x, y, z, which only depends on x, y, etc. However when the n is understood or irrelevant we just write f_A.

Note that by definition

$$A \equiv B \text{ iff } f_A = f_B.$$

The main fact about truth functions is the following: