Logic in Programming

Logic is extremely important in both the hardware and software of computing. Here we will begin with the software aspects of logic which are involved in programming. Later we will briefly show some hardware aspects of gates involved in computing architecture. Logic involves conditions in virtually all Choice and Loop constructs (If, and While forms). Logic also occurs in assertions, preconditions, post conditions, invariants and even comments.

The boolean (or logical) type is binary; it has only two values, true and false in angular boxes:

```
true  false
```

Logical boxes (variables) include such examples as:

```
male, female, tall, old, done, isIsosceles
```

which often must be clarified, perhaps by comments. For example, "old" could be defined as as any age over 21 (or 12, 30, 100, whatever).

Declarations of logical types specify boxes having only the two truth values. They have a form which begins with "boolean" followed by any number of variables separated by commas and end with a semicolon. Examples are:

```
boolean male, female;
boolean old;  // age over 21
boolean tall; // height over or equal to 72 inches
```

Assignment of truth values to boolean boxes (variables) can be done simply as:

```
male = true;
old  = false;
tall = old ;
```
Arithmetic Relations

Arithmetic relations often occur in logical conditions; the relations compare two quantities of the same type (such as ints here).

(a < b) which reads "a is less than b"
(c > d) which reads "c is greater than d" or "c is more than d"

(e <= f) which reads "e is less than or equal to f"
(g >= h) which reads "g is greater than or equal to h"

(i == j) which reads "i is equal to j"
(j != k) which reads "j is not equal to k"

Alternate or equivalent ways are possible to express conditions:

\[(p < q)\] is equivalent to \[(q > p)\]

for example

(age < 12) is equivalent to \((12 > \text{age})\)

since they both are true for values 11, 10, 9, 8, .. etc
and both are false for values 12 and 13, 14, 15, …, etc

Similarly, other equivalent conditions are:

\[(x <= y)\] is equivalent to \[(y >= x)\]
as in the example:

\[(7 <= \text{sum})\] is equivalent to \[(\text{sum} >= 7)\]

Notice especially that \((a < b)\) is not the opposite or complement of \((a > b)\). Complements are treated shortly.
Assignment may also involve comparison with relations such as:

```java
over21 = (age > 21);
```

where over21 is true for age 22, 23, 24 .. etc and is false for ages 21,20, 19 .. etc.

Other examples of logical assignment are:

```java
tied   = (visitorScore == homeScore);
error  = (age < 0);
proper = (percent <= 100);
tall   = (height >= 72);    // inches
tall2  = (denominator == 0);
```

Use of logical terms can be done in a number of ways. For example,

```java
legal = (over21 == true);
```

can be done simpler and shorter with

```java
legal = over21;
```

Equivalent Control Forms: (optional at first)

Logical assignments can also be done as imperative or control-oriented forms. For example the assignment

```java
over21 = (age > 21);
```

can be written as the Choice form:

```
--- PseudoCode
If age > 21
   Set over21 = true
Else
   Set over21 = false
EndIf

// Java Code
if (age > 21) {
   over21 = true;
} else {
   over21 = false;
}//end if
```
**Complements (Opposites, negatives, inverses)**
Complements are logical opposites; when one is true the complement is false. Examples of complements include male, female also tall, short and old, young.

Complements are expressed with the logical operator "!" which reads "not". The complement, or Not, is unary, it acts on the one condition that follows it. If the boolean box $b$ is true then the complement $!b$ is false; if $b$ is false then $!b$ is true, as shown in the following truth table,

<table>
<thead>
<tr>
<th>$b$</th>
<th>$!b$</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
</tr>
</tbody>
</table>

and a shorter version

<table>
<thead>
<tr>
<th>$b$</th>
<th>$!b$</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
</tr>
</tbody>
</table>

For example, consider the assignment:

```
small = ! tall;
```

If tall is true then small is false, and if tall is false then small is true.

Other example assignments follow:

```
female = ! male;
young = ! (age > 12);
```

**Complements** can involve arithmetic relations:

- $(a < b)$ is the complement of $(a >= b)$
- $(a > b)$ is the complement of $(a <= b)$
- $(a == b)$ is the complement of $(a != b)$

The above condition for young can be written without the not operator as:

```
!(age > 12) is (age <= 12)
```

Similarly:

```
!(age <= 21) is (age > 21)
```
Logical Binary Operations (And and Or)

Operations on logical or boolean boxes include two binary operators (And and Or); The first letter of And and Or may be capitalized to avoid confusion, as in the line above.

- **And** (also called "andAlso" or conjunction, symbolized by &)
- **Or** (also called "eitherOr" or disjunction, symbolized by |)

Binary operations (Or, And) operate on two operands; the operator is between the two operands (called infix).

**And** is a logical operation often called the conjunction. The And denoted (p && q) is true when p is true and q is true. Examples of the use of this And follow.

```
increasing = (x < y) & (y < z);
equilateral = (s1 == s2) & (s1 == s3);
isInRange = (percent >= 0) & (percent <= 100);
isEligible = over21 & isEmployed;
```

Truth table of the logical And operator is shown below, including a shorter form; The And of two operands is true only in one case, when both p and q are true.

<table>
<thead>
<tr>
<th>first</th>
<th>second</th>
<th>first &amp; second</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>p</th>
<th>q</th>
<th>p &amp; q</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>
**Or** is a logical operation often called the disjunction, **eitherOr**, or inclusive-Or; The Or denoted \((p \lor q)\) is true when either \(p\) or \(q\) or **both** are true.

Examples of the use of the **Or** follow, in Java.

```java
winpoint = (sum == 7) | (sum = 11);
error     = (percent < 0) | (percent > 100);
playBall  = (inning <= 9) | (score1 == score2);
isosceles = (a == b) | (b == c) | (c == a);
```

Truth table of the logical **Or** operator is shown below.
The **Or** of two operands is true in 3 cases, when either \(p\) or \(q\) are true.
The **Or** is false only in one case, when both \(p\) and \(q\) are false (or neither is true).

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(p)</td>
<td>(q)</td>
<td>(p \lor q)</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(p)</td>
<td>(q)</td>
<td>(p \lor q)</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>F</td>
<td></td>
</tr>
</tbody>
</table>

**Exclusive-Or** (or **xor**, or **OrElse**) is similar to this common **Or**, but it is not often used. The exclusive-Or is true if only one or the other, but not both are true; The Exclusive-Or differs in the last row of the truth table as shown.
Logical Views:

Control Flow View

Behavior of logical operators can be shown by control flow charts as given below.

And \((a \& b)\)

Or \((p \mid q)\)

Not \(x\)
Another View: Lazy Logic (called short circuit): Optional at first

Another view of logic is shown by the following control flow diagrams.

Note that if the first condition "a" of the And is false, then the second one "b" is not tested. Note also that if the first condition of the Or is true then the second one is not tested.

In other words, sometimes only one condition, the first one, is tested; this is called "short circuit" evaluation or "lazy evaluation". The lazy Or is similar. This “lazy” logic is only slightly faster, and will be treated later, when useful.
**Play Ball:** The condition for continuing to play a ball game can be done in many ways:

a. In English: play when the score is tied or inning less than or equal to 9 and it's not raining.

b. In Symbolic logic:

\[
\text{playBall} = ( (\text{score1} == \text{score2}) \lor (\text{inning} <= 9) ) \& ! \text{rain};
\]

c. As a control flow chart: two of them; more are possible.

There are many ways to do anything; there is a **tradeoff** between logic and program structure.
How many tests are needed to prove that this works properly? 3, 4, 5, 7, 8, 12, 16, more?
Non-Equality of Real Values

Real values (called floats and doubles) should not be compared for equality by the relation "==". This is because real values seem to be very precise, but actually may be approximate. For example, the real value 0.2 when converted to binary computer form is:

\[
0.2 = 0.001100110011 \ldots 0011 \ldots \text{ forever}
\]

which continue repeating forever, unending. In practice it is chopped off (often after 32 or 64 bits), and is sufficiently accurate for most needs, but may ultimately cause problems.

Comparison between two values, say a, b, can however be made by checking if the difference between them is less than a given small value, such as 0.1 as shown below.

\[
isClose = ((a-b) < 0.1) \; | \; ((b-a) < 0.1); \; // \text{within a tenth}
\]

For closer comparison we can decrease the size of the difference:

\[
isVeryClose = ((a-b) < 0.001) \; | \; ((b-a) < 0.001); \; //\text{within a thousandth}
\]

Also, we can use an absolute value function (from Math class) and specify a "tolerance" of error:

\[
\text{TOLERANCE} = 0.00001; \; // \text{Whatever small constant}
\]

\[
isWithinToleranace = (\text{Math.abs}(a-b) < \text{TOLERANCE});
\]
Another View of Logic: Data Flow View

Behavior of logical operators can be shown by data flow diagrams as given below.

For example, a true value flows out of an And only if a true value flows into both inputs. Similarly, a true value flows out of an Or if a true value flows into either or both inputs.

Two-dimensional tables are an equivalent way to define some logical operators as follows.

Viewed as an algebraic op: If F is 0 and T is 1, then And is multiplication, Or is like addition (except that $1 + 1 = 1$)
Illogic -- Looks good .. BUT is NOT

Illogical conditions or statements look and sound logical, and often have a reasonable meaning to human beings, but they are not of the proper form for computing.

For example, an "illogical" statement often written in mathematics is:

\[ x < y < z \]

which humans easily interpret as the two combined relations \( x < y \) and also \( y < z \). This is written "logically" as:

\[ (x < y) \land (y < z) \]

Conversion of other illogical, but meaningful conditions for humans follow. First, look at the illogic at the left and try to convert it before looking at the right. Assume that a, b, and c are boxes of integer type.

\[
\begin{align*}
\text{a and b < 7} & \quad \text{converts to} \quad (a < 7) \land (b < 7) \\
\text{a > b or c} & \quad \text{converts to} \quad (a > b) \lor (a > c) \\
\text{a <= b and c} & \quad \text{converts to} \quad (a <= c) \land (b <= c) \\
\text{a == b == c} & \quad \text{converts to} \quad (a == b) \land (b == c) \\
\text{a == b and c} & \quad \text{converts to} \quad (a == b) \land (a == c) \\
\text{a != b or c} & \quad \text{converts to} \quad (a != b) \land (a != c) \\
\text{and also to} & \quad !((a == b) \lor (a == c))
\end{align*}
\]
Compare many ways

A common relation which often occurs in describing an interval is:

\[ A < b < C \]

where A and C are constant values at the boundaries of an interval, and b is a box (variable) which is limited to be between these two.

For example, passing grades could depend on the percent score as:

\[ (50 < \text{score} \leq 100) \quad // \text{not proper} \]

This form is not acceptable in most programming languages; such relations should be rewritten as:

\[ (a < b) \& (b < c) \]
\[ \text{pass} = (50 < \text{score}) \& (\text{score} \leq 100) \]

or often rewritten using the box b at the left of any compare

\[ (b > a) \& (b \leq c) \]
\[ \text{pass} = (\text{score} > 50) \& (\text{score} \leq 100) \]

and very occasionally written at the right of a compare as

\[ (a < b) \& (c > b) \]
\[ \text{pass} = (50 > \text{score}) \& (100 \geq \text{score}) \]
Programming with Logic
The logical system, involving the boolean type is shown as a class diagram below. This system includes the boolean values (true and false,) three operations (&, |, !), assignment (=) and comparisons (== and !=). Notice that the other comparative relations (<, >, <=, >=) are not defined for this system; one truth value is not larger than another.
Other methods include input, output and conversion from boolean to string.

Logical System

Some Boolean Methods using JJS
JJS.inputBoolean()
JJS.outputBoolean(b);
JJS.outputlnBoolean(b);
JJS.booleanToString(b)

Other boolean operators include **Xor** and (but not in Java): **Nand, Nor, Inhibit!**, and others.
Logic Code
Code follows showing a simple check of the And operation in Java.

```java
// Does logical And operation
System.out.println(" And ");
System.out.println(false & false);
System.out.println(false & true );
System.out.println(true  & false);
System.out.println(true  & true);
```

(Output)
And
false
false
false
true

Code follows for using two boolean boxes T, F for simplicity. The two values T, F are declared as the logical type called boolean. These can also be declared as constants in the form:

```
public static fixed boolean T = true;
```

// Does logical Or operation
// Uses a shorter form of the truth values

```java
boolean T, F;
T = true;
F = false;

System.out.println(“ Or“);
System.out.println(F | F);
System.out.println(F | T);
System.out.println(T | F);
System.out.println(T | T);
```

(Output)
Or
false
ture
true
ture
Logic Code

Code follows showing a simple check of the exclusive Or Xor operation. The two logical values first and second are declared of type boolean. Notice again that the last case is the only different one from the usual Or.

```java
// Does logical XOr operation

boolean first, second;
System.out.println(" Xor ");
first = false; second = true;
System.out.println(first ^ second);
first = false; second = true;
System.out.println(first ^ second);
first = false; second = true;
System.out.println(first ^ second);
first = false; second = true;
System.out.println(first ^ second);
```

(Output)

```
Xor
false
true
true
false
```
**Logic Code**

Code follows showing a simple check of the And operation. It uses the `inputBoolean` and the `outputlnBoolean` methods of the JJS. Three of the four possible cases are shown as output; Do the fourth one.

```java
// Does logical AND
// Shows boolean code

boolean first, second, both;

JJS.outputlnString("true or false? ");

JJS.outputlnString("Is first true? ");
first = JJS.inputBoolean();
JJS.outputlnBoolean(first);

JJS.outputlnString("Is second true? ");
second = JJS.inputBoolean();
JJS.outputlnBoolean(second);

JJS.outputlnString("Both are true is ");

\[ \text{both} = \text{first} \& \text{second}; \]

JJS.outputlnBoolean(both);
```

true or false?
Is first true?
true
Is second true?
true
Both are true is true

true or false?
Is first true?
false
Is second true?
true
Both are true is false

true or false?
Is first true?
false
Is second true?
false
Both are true is false
Logic DisProof: cannot factor out the Not operation

Code follows which disproves the factoring of the Not operation over the And operation.

```
// Does prove inequality
// Shows non linearity of And

boolean first, second, left, right, equal;

JJS.outputlnString ("true or false? ");

JJS.outputlnString ("Is first true? ");
first  = JJS.inputBoolean();
JJS.outputlnBoolean (first);

JJS.outputlnString ("Is second true? ");
second  = JJS.inputBoolean();
JJS.outputlnBoolean (second);

left  = (!first) & (!second); //ERR
right = ! (first & second);

equal = (left == right); //NOT TRUE

JJS.outputlnString ("Equality is ");
JJS.outputlnBoolean (equal);
```

Disproving a property such as this factoring:

```
(! a & !b) = !(a & b) //NO
```
can be done by showing just one counter-example (the second case above).
You try with the Or operation.
// Does Majority of three
// Shows boolean evaluation

boolean first, second, third, majority;

JJS.outputlnString ("true or false? ");

JJS.outputlnString ("Is first true? ");
first  = JJS.inputBoolean ();
JJS.outputlnBoolean (first);

JJS.outputlnString ("Is second true? ");
second = JJS.inputBoolean ();
JJS.outputlnBoolean (second);

JJS.outputlnString ("Is third true? ");
third  = JJS.inputBoolean ();
JJS.outputlnBoolean (third);

majority = first  & second  | first  & third  | second & third ;

JJS.outputString ("The majority is " );
JJS.outputlnBoolean (majority);

true or false?
Is first true?
true
Is second true?
true
Is third true?
true
The majority is true

true or false?
Is first true?
false
Is second true?
false
Is third true?
false
The majority is false

true or false?
Is first true?
true
Is second true?
true
Is third true?
false
The majority is true

There are 5 other cases; what are they?
DeMorgan's Law is a logical equality which relates the Not of a combined logical expression. It has the two following forms.

\[
!(p \ & \ q) = !p \mid !q \quad // \quad \text{Not (p And q) = (Not p) Or (Not q)}
\]

\[
!(p \mid q) = !p \& !q \quad // \quad \text{Not (p Or q) = (Not p) And (Not q)}
\]

Notice especially that the Not of the And of two truth values is equal to the Or of the Not of the two truth values; it is not equal to the And of the two Nots. In other words the Not is distributed over the And, but the And changes to an Or. Similarly the Not of Ands is the Or of the Nots.

These deMorgan's laws are often used to complement a condition; sometimes the complement is more convenient to use or read. Examples of complements follow.

\[
!(\text{old} \ & \ \text{rich}) = (! \ \text{Old}) \mid (! \ \text{Rich})
\]

\[
= \quad \text{young} \mid \text{poor}
\]

\[
\text{percentInRange} = ! \ \text{percentOutOfRange}
\]

\[
= ! \ ( (\text{percent} < 0) \mid (\text{percent} > 100) )
\]

\[
= \quad ! (\text{percent} < 0) \ & \ ! (\text{percent} > 100)
\]

\[
= \quad (\text{percent} >= 0) \ & \ (\text{percent} <= 100)
\]

Parentheses are not required around the inner conditions, but it is useful to include them for clarity. For example, the last above example could also be written (non preferably) as:

\[
\text{percentInRange} = \text{percent} >= 0 \ & \ \text{percent} <= 100
\]
**Proof using Truth Tables**

Logical proofs involve testing all possible cases; which are often a small and finite number.

For example to prove the deMorgan Rule:

\[
!(p \land q) = !p \lor !q
\]

involves only 4 cases of \(p, q\): FF, FT, TF, TT.

For the first case: \(p=F\) and \(q=F\) the left side becomes:

\[
!(p \land q) = !(F \land F) = !(F) = T
\]

and the right side evaluates to:

\[
!p \lor !q = !F \lor !F = T \lor T = T
\]

Similarly for the second case: \(p\) is F, \(q\) is T both sides evaluate to the same result, which is T.

**Row-by-row** evaluation of all 4 cases is summarized on the given truth table, which shows the same result for all cases, so proving this DeMorgan's law.
Column-by-Column proof is also possible, as shown in the given table:

\[
\begin{array}{ccccccc}
\text{p} & \text{q} & \text{!} & \text{(p & q)} & \text{==} & \text{!p} & \text{!q} \\
\hline
\text{F} & \text{F} & \text{T} & \text{F} & \text{F} & \text{T} & \text{T} \\
\text{F} & \text{T} & \text{T} & \text{F} & \text{F} & \text{T} & \text{T} \\
\text{T} & \text{F} & \text{T} & \text{F} & \text{F} & \text{T} & \text{T} \\
\text{T} & \text{T} & \text{F} & \text{F} & \text{F} & \text{T} & \text{T} \\
\end{array}
\]

1 2 4 3 8 5 7 6 is order of operation

Numbers under the columns show the order in which the columns are done. The last column 8 is most important, showing the equality in all the cases.

Another proof follows; it renames some interior points (r, u, v) and is more graphical rather than algebraic. It may seem simpler in some ways.

Proof of DeMorgan’s Law

\[
\begin{array}{cccccccc}
\text{p} & \text{q} & \text{r} & \text{s} & \text{u} & \text{v} & \text{t} \\
\hline
\text{F} & \text{F} & \text{F} & \text{T} & \text{T} & \text{T} & \text{T} \\
\text{F} & \text{T} & \text{F} & \text{T} & \text{T} & \text{T} & \text{T} \\
\text{T} & \text{F} & \text{F} & \text{T} & \text{F} & \text{T} & \text{T} \\
\text{T} & \text{T} & \text{T} & \text{F} & \text{F} & \text{F} & \text{T} \\
\end{array}
\]
Duals

The dual of a logical formula is created by exchanging the binary operations.
For example, the previous DeMorgan’s Law has a corresponding dual as follows.

\[ ! (a \land b) = !a \lor !b \]

is the dual of \[ ! (a \lor b) = !a \land !b \]

This dual version of DeMorgan’s law is proved below using a table with logical flows.
Distribution: logical factoring

Three logical conditions in a formula need 8 rows. For example, the property of distribution follows; it shows that the common boolean box "a" is factored out of two terms. It is similar to the ordinary arithmetic factoring formula:

\[ a \times b + a \times c = a \times (b + c) \]

\[ (a \land b) \lor (a \land c) = a \land (b \lor c) \]

The logical data flow diagrams show that these two equivalent formulas correspond to two very different structures; one can be substituted for another. The simplest one is better; that is the second one, on the right hand side. Notice the order of the 8 combinations of a,b,c: it is not random, but common.

The following more unusual distribution property also holds:

\[ (a \lor b) \land (a \lor c) = a \lor (b \land c) \]
Coding Larger Truth Tables

Code follows showing a simple check of the exclusive Or Xor operation, involving all 8 combinations of 3 boolean variables.

Similarly 4 boolean variables would involve 16 combinations, And 5 boolean variables would involve 32 rows.

// Does show the truth table of 3 boxes
// Defines 8 cases of Xor (a ^ b ^ c)

boolean F = false;
boolean T = true;

System.out.println (a^b^c);
System.out.println (-----); //--------------------
System.out.println (F ^ F ^ F);
System.out.println (F ^ F ^ T);
System.out.println (F ^ T ^ F);
System.out.println (F ^ T ^ T);
System.out.println (T ^ F ^ F);
System.out.println (T ^ F ^ T);
System.out.println (T ^ T ^ F);
System.out.println (T ^ T ^ T);
Coding Truth Tables with Loops (Optional)

Code follows showing a simple check of the exclusive Or Xor operation. This is done using two loops, one nested within another. This could be extended to larger logical formulas by more nests. You do this for three nested loops checking some distributive property.

```java
// Does truth table defining Xor
// Using a loop: while-true-break
boolean a, b;
System.out.print("  a   b");
System.out.println("  a ^ b");
a = false;
while (true) {
    b = false;
    while (true) {
        System.out.print(a + "\t");
        System.out.print(b + "\t");
        System.out.println(a ^ b);
        if (b) break;
        b = ! b;
    }//end while b
    if (a) break;
    a = ! a;
}//end while a
```

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>a ^ b</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
<td>false</td>
</tr>
</tbody>
</table>
PlayBall is a condition involving 3 boolean boxes, and 8 possible combinations of the 3 conditions shown. One condition (F,T,T) is emphasized; the others are equally important.

Play = (early | tied) & ! rain

PlayBall can be specified in more detail, by including the numeric relational operators as shown in the data flow diagram at the right.
**Boolean Algebra**

Logic can be viewed as a boolean algebra; for example, the complement (or Not) of playBall, which is stopPlay, can be created using DeMorgan’s law algebraically as follows.

\[
\text{stopPlay} = \neg \text{playBall} \\
= \neg ( (\text{inning} \leq 9) \mid (\text{score1} = \text{score2}) ) \& \neg \text{rain} \\
= \neg ( (\text{inning} \leq 9) \mid (\text{score1} = \text{score2}) ) \mid \neg (\neg \text{rain}) \\
= \neg (\text{inning} \leq 9) \& \neg(\text{score1} = \text{score2}) \mid \text{rain} \\
= (\text{inning} > 9) \& (\text{score1} \neq \text{score2}) \mid \text{rain}
\]

The data flow diagram showing this stopPlay logic follows, at the left along with the previous playBall logic, at the right.
// Does playBall algorithm
// Shows boolean evaluation

boolean tied, early, rain, play;

JJS.outputlnString ("true or false? ");

JJS.outputlnString ("Is it raining? ");
rain  = JJS.inputBoolean ();
JJS.outputlnBoolean (rain);

JJS.outputlnString ("Is score tied? ");
tied  = JJS.inputBoolean ();
JJS.outputlnBoolean (tied);

JJS.outputlnString ("Inning <= 9 ? ");
early  = JJS.inputBoolean ();
JJS.outputlnBoolean (early);

play = (tied | early) & ! rain;

JJS.outputString ("Play is ");
JJS.outputlnBoolean (play);

true or false?
Is it raining?
true
Is score tied?
true
Inning <= 9?
true
Play is false

true or false?
Is it raining?
true
Is score tied?
false
Inning <= 9?
true
Play is false

true or false?
Is it raining?
false
Is score tied?
false
Inning <= 9?
true
Play is true

There are 5 other cases; what are they?
Larger Truth Tables

Truth tables involving four boolean variables would require 16 rows as shown below. In this case the output is true when either the first two are the same value, or the last two are the same value (either both true or both false).

\[ e = (a == b) \ | \ (c == d) \]

The three last columns can be done one at a time as shown.

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>a=b</th>
<th>c=d</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>

Note the form of the values:

\begin{align*}
F & F & F & F \\
F & F & F & T \\
F & F & T & F \\
F & F & T & T \\
F & T & F & F \\
F & T & F & F \\
F & T & T & F \\
F & T & T & T \\
T & F & F & F \\
T & F & F & T \\
T & F & T & F \\
T & F & T & T \\
T & T & F & F \\
T & T & F & T \\
T & T & T & T \\
T & T & T & F
\end{align*}
Complementary Ranges
A range of values on the number line can be described logically in two ways, by the "inner" values within the range, and by the "outer" values outside of it.

Ranges can also be described by logic, as shown below each of the figures.

Truth tables at the right show how the two ranges are complementary.

They also show that some combinations (on the last line) are not possible. The question marks indicate that a value Cannot be less than or equal to 2 and Also be greater than 4.
Bigger Complements

Prove or disprove the complementarity of the following two conditions:

\[
\text{top} = (x \leq 30) \& \ ( (x > 20) \ | \ (x \leq 10) );
\]
\[
\text{bot} = (x > 10 ) \& \ ( (x > 30) \ | \ (x \leq 20) );
\]

There are many ways to do this; some ways are better than others.

1. Truth table (with test cases)
   Not all cases are possible;

\[
\begin{array}{ccc|cc|cc}
(x \leq 10) & (x \leq 20) & (x \leq 30) & x & \text{top} & \text{bot} \\
F & F & F & 35 & F & T \\
F & F & T & 25 & T & F \\
F & T & F & ? & ? & ? \\
F & T & T & 15 & F & T \\
T & F & T & ? & ? & ? \\
T & T & F & ? & ? & ? \\
T & T & T & 5 & T & F \\
p & q & r & a & b & \\
\end{array}
\]

2. Graphic way
   provides great insight!

3. Finally a Boolean algebra proof is not possible!

   If \( p = (x \leq 10) \), \( q = (x \leq 20) \), \( r = (x \leq 30) \) then
   \[
   r \& (\lnot q \mid p) \neq \lnot p \& (\lnot r \mid q)
   \]
   This can be "disproved" with a truth table or by finding one case that does not hold.
Complement Code
Evaluating numeric relations can be done by code as the following.

```java
// Does evaluate complements
// Shows complementary relations

int x; // could also be double
boolean top, bot;

JJS.outputlnString("Enter a number ");
x = JJS.inputInt();

top = (x <= 30) & ((x > 20) | (x <= 10));
bot = (x > 10) & ((x > 30) | (x <= 20));

JJS.outputString("Top is ");
JJS.outputlnBoolean(top);

JJS.outputString("Bot is ");
JJS.outputlnBoolean(bot);
```

Note that only the given four tests out of a total possible eight combinations are required. all the other four combinations correspond to situations that cannot occur.
Simplifying Logic

Some logic can be simplified if more information is known.

For example, given three sides of a triangle, labelled \( s_1, s_2, s_3 \), we can determine if it is an equilateral triangle logically as:

\[
equilateral = (s_1 == s_2) \& (s_2 == s_3) \& (s_3 == s_1)
\]

or more simply as:

\[
equilateral = (s_1 == s_2) \& (s_1 == s_3)
\]

or alternately as:

\[
equilateral = (s_1 == s_2) \& (s_2 == s_3)
\]

However, if it is known that the sides are in increasing order, 
\[(s_1 <= s_2) \text{ and also } (s_2 <= s_3)\]

then this can be more simply written as:

\[
equilateral = (s_1 == s_3);
\]

since if the first equals the last, then the middle value must equal both.

Alternately, if a triangle is given by three angles \( a_1, a_2, a_3 \) then

\[
equilateral = (a_1 == a_2) \& (a_1 == a_3) \& (a_1 == 60)
\]

and also

\[
equilateral = (a_1 == a_3) \& (a_1 + s_2 + a_3 == 180)
\]

as well as

\[
equilateral = (a_1 == 60) \& (a_2 == 60) \& (a_3 == 60)
\]
**Finite Quantified Logic (optional)**

**Quantifiers** involving the words "all" or "some" are often used in logic. For example, an acute triangle has all angles less than 90 degrees; an obtuse quadrilateral has some angles larger than 90 degrees. When the number of entities (angles) is finite (such as the 3, or 4 here) then such logic simply involves a "chain" of Ands or Ors, as follows:

\[
\text{allAnglesLessThan90} = (a_1 < 90) \land (a_2 < 90) \land (a_3 < 90);
\]

\[
\text{someAngleMoreThan90} = (a_1 > 90) \lor (a_2 > 90) \lor (a_3 > 90) \lor (a_4 > 90);
\]

or alternately as:

\[
\text{isAcuteTriangle} = \text{allAnglesLessThan90};
\]

\[
\text{isAnObtuseQuad} = \text{someAngleMoreThan90};
\]

**Universal** quantifies (involve "all") are sometimes written as an inverted A symbol,

\[
\forall i \ (i < 90) \ // \text{For all angles } i, i \text{ is less than } 90
\]

**Existential** quantifiers (involve "some") are written as an inverted E symbol.

\[
\exists j \ (j > 90) \ // \text{For some angle } j, j \text{ is more than } 90
\]
Many Logical Operations

Binary operators (or binOps) involve two variables; the most common logical ones are the And and the Or. There are many more operators which are not as common. The given truth table describing a general binary op, showing that 16 such operators are possible, since there are $2^4$ ways to fill in this table.

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>

All possible binary operators are listed at the right, where the truth table is drawn on its side.

Nand and Nor operators are used in engineering, Implies is an operator of use in philosophy, ExclusiveOr is of interest in mathematics, Inhibit is not too often used anywhere. Others (such as 0,3,5,15) are trivial.
More Logic: Nand and Nor and More (optional)

Basic Logical Operations include And, Or and Not; Java also includes Xor. However there are more logical operations including Nand, Nor, and even more. Nand and Nor are using in Engineering; they are given in the next section: Logic 2. Implication is an operator (connective) which is used in Philosophy.

Nor is an operation which is simply the Not of an Or; Nor is true when neither of the two values is true as shown below.

<table>
<thead>
<tr>
<th>first</th>
<th>second</th>
<th>nor</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
</tbody>
</table>

Nand is an operation which is simply the Not of an And.

Implies is used in philosophy to mean “if - then”; Note that first implies second is very different from second implies first.

<table>
<thead>
<tr>
<th>first</th>
<th>second</th>
<th>first implies second</th>
<th>second implies first</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td>false</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
</tbody>
</table>
Implication (Optional)

Implication (biconditional, if..then, etc) is a very common logical operation in symbolic logic which is related to human language. This operator is not very common in computing, but is still useful to know.

The notation "p -> q" reads "if p then q", and has the following truth table at the left. It can be expressed as "!p | q" as proven at the right.

<table>
<thead>
<tr>
<th>p</th>
<th>q</th>
<th>p -&gt; q</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>p</th>
<th>q</th>
<th>!p</th>
<th>q</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>

Deduction is a process of beginning with axioms (truths) and concluding other truths based on these. It makes use of properties of such a logical rule as:

Modus Ponens

\[(p \&\& (p \Rightarrow q)) \Rightarrow q\]

Ponens is sometimes written as an argument in three lines; the first two are called premises, the last line is called the conclusion.

\[p \Rightarrow q\]

-- if p is true and
\[\neg p \Rightarrow q\]

-- if p implies q

---

\[q\]

-- q is true
Arguments, such as the following fit the form of Modus Ponens

\( \text{p is "Fire is lit"} \)
\( \text{q is "There is light"} \)

\( (\text{p} \rightarrow \text{q}) \) -- if the fire is lit then there is light
\( \text{p} \) -- and the fire is lit

---------- -- then (therefore)
\( \text{q} \) -- there is light

Proof of Ponens can be done by a truth table as shown, which is ultimately true for all 4 truth combinations (and is called a tautology, logically true in all cases).

<table>
<thead>
<tr>
<th>p</th>
<th>q</th>
<th>(p &amp; (p -&gt; q))</th>
<th>-&gt;</th>
<th>q</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>

| 1 | 2 | 3 | 5 | 4 | 7 | 6 |

**Problem**: Show by means of a truth table, that this operator \( \rightarrow \) can be written in terms of Not and And.
Fallacies, or improper arguments can similarly be proven false. For example consider the false truth, which may look like a proper form of argument:

\[(p \rightarrow q) \quad \text{-- if } p \text{ implies } q\]
\[! p \quad \text{-- and not } p\]
\[--------- \quad \text{-- therefore}\]
\[! q \quad \text{-- } q \text{ is not true}\]

Using the previous symbols for p, q shows the wrong reasoning:

- If the fire is lit then there is light
- And the fire is not lit
- Therefore there is no light

But the light need not come only from the fire (perhaps from a bulb, sun, etc)

This fallacy can be written as a single expression:

\[
( (p \rightarrow q) \& ! p ) \rightarrow ! q
\]

It can be disproved by a truth table. Actually the entire table is not required; only one case is sufficient, and this is the second case (when p is false and q is true).
**Dilemmas**, describe conflicts, which are logically very interesting. The general form of dilemmas is as follows, where c and d are usually inescapable, unavoidable consequences.

\[
\begin{align*}
a & \rightarrow c \\
b & \rightarrow d \\
a \text{ or } b & \\
\hline 
c \text{ or } d
\end{align*}
\]

The following example illustrates the concept of a dilemma.

If you accept certain advise then you feel dominated.
If you reject the advise then you feel guilty.
But you must either accept the advice or reject it.
So therefore you feel dominated or guilty.

A truth table describing dilemmas follows;
Notice the form of the Or which is used.
Try this with both the inclusive and exclusive Or, which leads to three other forms of dilemma.
One form of the dilemma views the first Or as an exclusive Or, and the second Or as in inclusive Or as follows:

\[ a \rightarrow c \]
\[ b \rightarrow d \]
\[ a \lor b \]
\[ c \mid d \]

The premises \( p \) are:
\[ p = (a \rightarrow c) \land (b \rightarrow d) \land (a \lor b) \]

The consequences or conclusions are:
\[ q = (c \mid d) \]

So the final is
\[ t = (p \rightarrow q) \]

which is true in all possible cases
(and is called a tautology)

You try this for the 3 other combinations of inclusive Or and exclusive Or.
**Logic as Functions (later)**
Logical operations can be written as boolean functions. For example, the And operator can be written as \texttt{and(a,b)}, the Not operator as \texttt{not(c)}, and the inclusive Or as \texttt{ior(d,e)}.

The logical assignment
\[ r = p \land \neg q \]
can be written in this function notation as
\[ r = \texttt{and(p, not(q))} \]

Similarly other logical functions can be defined, such as
\texttt{xor(f,g)} is the exclusive Or
\texttt{nor(h,i)} is the Nor operator
\texttt{nand(j,k)} is the Nand

Equality of two logical states \( m, n \) can be defined as
\texttt{equ(m,n)} is true if \( m \) is equal to \( n \)

DeMorgan’s Law, which is written as
\[ \neg(p \lor q) = (\neg p) \land (\neg q) \]
can be written in the function notation as:
\[ \texttt{equ(not(ior(p,q)), and(not(p), not(q)))} \]
Logic as Functions in Java
Logical operations can be written as boolean functions in java as follows:

```java
public static boolean and (boolean x, boolean y) {
    return (x & y);
};//end method and

public static boolean ior (boolean x, boolean y) {
    return (x | y);
};//end method inclusive or

public static boolean not (boolean x) {
    return (! x);
};//end method not

public static boolean imp (boolean x, boolean y) {
    return (!x | y);
};//end method implication

public static boolean equ (boolean x, boolean y) {
    return (x == y);
};//end method equal

// similarly you do the xor, nor, and nand
```
Using Logic Functions
The previously defined functions (and others such as xor, nor, nand, etc)
can be used in a program such as the following one,
which simply shows the definition of implication.

```java
public static void main (String[] args) {
    // Does define implication
    System.out.println ( imp (false, false) );
    System.out.println ( imp (false, true ) );
    System.out.println ( imp (true , false) );
    System.out.println ( imp (true , true ) );
}
```

Modus Ponens can be shown to hold in the following.

```java
public static void main (String[] args) {
    // Does show Modus Ponens: ((p -> q) & p) -> q
    p = false; q = false;
    System.out.println ( imp ( and (imp(p,q), p), q ) );
    p = false; q = true;
    System.out.println ( imp ( and (imp(p,q), p), q ) );
    p = true; q = false;
    System.out.println ( imp ( and (imp(p,q), p), q ) );
    p = true; q = true;
    System.out.println ( imp ( and (imp(p,q), p), q ) );
}
```

Output:
```
true
true
false
true
```
**Logical Notation: Infix, Prefix and Postfix Forms (Optional)**

Binary logical operations, such as And and Or, are normally written in the infix form where the binary operation is between the two operands (such as \( p \& q \), or \( r \mid s \)).

The unary operator of Not, is called prefix form, since it occurs before the operand (such as \(!t\), or \(!(u \& v)\).

Postfix form is yet another notation in which the operation follows after the operands. For example in postfix form

\[
\begin{align*}
(\!a) & \quad \text{is written as} \quad (a!)
\end{align*}
\]

\[
\begin{align*}
b \& c & \quad \text{is written as} \quad b \ c \ & \\
d \mid e & \quad \text{is written as} \quad d \ e \ |
\end{align*}
\]

More complex combinations are similar:

\[
\begin{align*}
f & \& !g & \quad \text{is written as} \quad f \ g \ ! \ & \\
(h & \ i) & \mid !j & \quad \text{is written as} \quad h \ i \ & \ j \ ! \mid \\
(x & \mid y) & \& !(x & y) & \quad \text{is written as} \quad x \ y \mid x \ y \ & \ ! \ & \ \\
\end{align*}
\]

Also going the other way, from postfix to infix is:

\[
\begin{align*}
x \ y \ ! \ & \ & \ x \ ! \ y \ & \mid & \quad \text{is written as} \quad (x \ & \ !y) \mid (x! \ & \ y)
\end{align*}
\]

Postfix forms are interesting in two ways;
First, no parentheses are ever needed;
the order of evaluation does not depend on such grouping.
Evaluation of the postfix form proceeds left to right in order. For example, truth tables for infix and postfix shown below. The infix one shows jumping (2,1,4,3); the postfix has order (1,2,3,4).

| x | y | ! (x & y) & (x | y) |
|---|---|---------------------|
| F | F | T                   |
| F | T | T                   |
| T | F | T                   |
| T | T | F                   |

<table>
<thead>
<tr>
<th>2</th>
<th>1</th>
<th>4</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>4</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>x</th>
<th>y &amp; ! (x &amp; y)</th>
<th>&amp;</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
</tbody>
</table>

Arithmetic operations, such as add '+' or multiply '*'
can also be done in postfix form, in calculators.
Postfix form is also called Reverse Polish notation, or RPN.
Many-Valued Logic
Binary logic with its two values of true and false is rather limiting. Implication in two values, repeated below, is especially bothersome to many. A third value can be introduced and indicated by the symbol I, for intermediate. This leads to larger tables as shown below; but the binary ones are subsets of the larger.

Three different versions of implication follow (out of how many possible?)

Lukasiewicz
Bochvar
Kleene
Sobocinski
**Logical Calculus: Boolean Algebra**

The logic here can be viewed as an algebra, with variables operations and constants; The two constants false and true can be represented as 0 and 1 respectively. Many properties are summarized below; note the duality between Ands and Ors. The properties of Xor are rather different.

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>x &amp; y</th>
<th>x</th>
<th>y</th>
<th>x ^ y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

- **Idempotence**
  \[(x \lor x) = x\] \[(x \land x) = x\]

- **Complementary**
  \[(x \lor !x) = 1\] \[(x \land !x) = 0\]

- **Operation of 0**
  \[(x \lor 0) = x\] \[(x \land 0) = 0\]

- **Operation of 1**
  \[(x \lor 1) = 1\] \[(x \land 1) = x\]

- **Commutativity**
  \[(x \lor y) = (y \lor x)\] \[(x \land y) = (y \land x)\]

- **Dualization**
  \[(!x \lor !y) = !(x \land y)\] \[(!x \land !y) = !(x \lor y)\]

- **Absorption**
  \[x \land (x \lor y) = x\] \[x \lor (x \land y) = x\]

- **Associativity**
  \[x \lor (y \lor z) = (x \lor y) \lor z\] \[x \land (y \land z) = (x \land y) \land z\]

- **Distributivity**
  \[x \land (y \lor z) = x \land y \lor x \land z\] \[x \lor (y \land z) = (x \lor y) \land (x \lor z)\]
Problems:

1. Prove the other DeMorgan result, using truth tables:
   \((\neg (p \lor q)) = \neg p \land \neg q\)

2. Prove or disprove the following "Big" DeMorgan Laws
   \((\neg ((p \lor q) \lor r)) = (\neg p \land \neg q) \land \neg r\)
   \((\neg (p \land q \land r)) = \neg p \lor \neg q \lor \neg r\)

3. Draw truth tables of the two following expressions, one of which is equivalent to the "Exclusive-Or" and then convert the other to the Exclusive-Or
   \((p \land \neg q) \lor (\neg p \land q)\)
   \((p \lor q) \land \neg (p \land q)\)

4. Prove, or disprove the "dual" logical property of distribution (or factoring) corresponding to the arithmetic property, and be prepared for a surprise:
   \((a + b) \times (a + c) = a + b \times c\)
Problems:

A. Count truth tables
A truth table of 2 logical variables has 4 combinations listed as below. The order of the combinations is usually given as shown: FF, FT, TF, TT and sometimes is the reverse of that order, as TT, TF, FT, FF. How many other ways are there to order these 4 combinations of two variables?

<table>
<thead>
<tr>
<th>p</th>
<th>q</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>

B. Count bigger tables
Truth tables of 3 logical variables have 8 combinations as listed below. How many ways are there to order these 8 combinations? Why is the given order, one of the most common ones? How many different logical functions are there of 3 variables?

F F F
F F T
F T F
F T T
T F F
T F T
T T F
T T T
Exclusive Problems:

Prove or disprove, by truth tables, the following about the exclusive-Or, \( p \ ^\lor \ q \).
Assume that the And has higher precedent that the Xor
(I.e the And is done before the Xor, unless there are brackets)
Draw data flow diagrams of the proven ones.

a. \( p \ ^\lor \ q = q \ ^\lor \ p \)
b. \( !(p \ ^\lor \ q) = p \ ^\lor \ q \)
c. \( !p \ ^\lor \ !q = p \ ^\lor \ q \)
d. \( p \ ^\lor \ p \ & \ q = p \ & \ !q \)
e. \( p \ | \ q = p \ ^\lor \ !p \ & \ q \)
f. \( p \ | \ q = p \ ^\lor \ q \ ^\lor \ p \ & \ q \)
g. \( p \ ^\lor \ (q \ ^\lor \ r) = (p \ ^\lor \ q) \ ^\lor \ r \)
1. **TriAngle Problem:**
Write logical assignments to determine if three angles (say ints) form:
   a. a triangle (where all angles sum to 180)
   b. a right triangle (where one angle equals 90)
   c. an isosceles triangle (where two angles are equal)
   d. an acute triangle (where all angles are less than 90)
   e. an obtuse triangle (where one angle is more than 90)
   f. a scalene triangle (where all the angles differ in size)
   g. an equilateral triangle (where all angles are same size)
Write a program that classifies 3 given integers; for example
the 3 values 60, 60, 60 result in an output of :
   Triangle is true
   Right triangle is false
   Isosceles is true
   etc...

2. **Quadrilateral Problem:**
Given 4 sides (ints in clockwise order) of a quadrilateral
(4 sided polygon) write logical assignments to determine if it is:
   a. quadrilateral (4 sided polygon)
   b. rectangle
   c. square
   d. parallelogram
   e. rhombus
   f. a trapezium
   g. kit
Logical Equivalence

Prove whether or not the following diagrams are equivalent. Write the not (negative or inverse) of each, using DeMorgan. Create another simpler diagram equivalent to the second one.
**Time Compare**

Given two “military” times in the same day, 
the first specified by two integers hour1, and minute 1, and 
the second specified by integers hour2 and minute 2 
(where hours go from 0 to 24, and minutes from 0 to 59), 
write the following three logical statements. 
1. earlier, which is true when first time is less than the second time.

2. later, which is true when the second time is larger

3. isSame, which is true only when both times are identical.

**Date Compare**

Given two dates in the same year, each specified by 3 integers 
(indicating the year, the month, and the day), 
write three logical statements for the three conditions, 
precedes, succeeds, and same.
Problems: Poker Dice
The game of Dice Poker involves the throwing of 5 dice, each yielding a value from 1 to 6. Some of these combinations (poker hands) occur less probably than others. For example, fiveOfAKind, where all values are the same, is very rare; This condition can be coded as:

\[
fiveOfAKind = (d1 == d2) \&\& (d2 == d3) \&\& (d3 == d4) \&\& (d4 == d5);
\]

However, if the five dice values are in order:

\[
d1 <= d2 <= d3 <= d4 <= d5  // dice in non-decreasing order
\]

then when the first equals the last, all the rest are equal also; The condition becomes:

\[
fiveOfAKind = (d1 == d5);  // only if dice are in order!!
\]

Write the code for the following conditions 
(which happen to be in increasing order of probability) Hint: picture it.

a. fourOfAKind, where 4 of the dice have the same value  
(such as 1,1,1,1,2, or 3,5,5,5,5 or 4,4,4,4,6 etc)

b. fullHouse, where 3 dice have one value and 2 have another value  
(such as 11133 or 22555)

c. straight, where there are 5 values in consecutive order (12345 or 23456)

d. threeOfAKind, where only three dice have the same value

e. twoPairs, where two dice are of one value, and two are of another value

f. onePair, where only two dice have the same value
1. **Play game of 42**
Consider a game between two players which is played when both scores are less than 42, or when at least one score is at or beyond 42, but there is less than a 3 point difference in scores. Write this logical expression, draw its data flow diagram, select some good test cases. Do also the complement (reverse, or negative) of this condition.

2. **Complements**
Prove or disprove that the following conditions are opposite, (or complementary). Hint: there are many ways; find two.

\[(x \leq 30) \& ( (x > 20) \mid (x \leq 10) )\]

\[(x > 10) \& ( (x > 30) \mid (x \leq 20) )\]

3. **Equivalents**
Prove or disprove the equivalence of the following two forms:

\[\text{top} = (x \leq 2) \mid ( (x > 4) \& (x \leq 6) )\]

\[\text{bot} = (x \leq 6) \& ( (x > 4) \mid (x \leq 2) )\]
**Associativity**

The associative property of a binary operator has the form

\[(a + (b + c) == (a + b) + c]\]

Which indicates that the order of bunching the op is irrelevant.

Test this property for the following operators:

a. exclusive or
b. nand
c. nor
d. implication

**Transitivity**

The transitive property involving the comparison \(<\) has the form

\[( (p < q) & (q < r) ) -> (p < r)\]

Test this property for the implication operator \(\rightarrow\)
Logical Swap

Prove, or disprove, for all possible truth value combinations that the given series of logical commands do swap (interchange, exchange) the contents of the logic boxes r and s.

\[
\begin{align*}
    r &= r \land s; \\
    s &= r \land s; \\
    r &= r \land s; \\
\end{align*}
\]

Rules of Game

Consider a game where two people take turns; they play the game until both scores are over 7, and one score is double the other or until time runs out (after 17 turns).

Write some logic which when given two scores indicates
a. if the game is completed,
    b. which person (first or second) wins this game.

Show some test cases.
Many Operators (ops)

Do the following problems, showing the truth tables

1. Write the operator \(\rightarrow\) in terms of the Or and Not operators.

2. Write the exclusive-Or operator in terms of And, Or and Not operators.

3. Write the exclusive-Or operator in terms of one of the two comparison operators.

4. Write the exclusive-Or operator in terms of the Not and a comparison operator.

5. Write the Nand operator in terms of the Or and Not operators.

6. Write the Nor operator in terms of And and Not operators.
Laws from the Theory of inference

Write each of the following in the form of premises and conclusions (on many lines) and then prove (or disprove) them.

**Simplification:**

\[(p \& q) \rightarrow p\]

**Addition:**

\[P \rightarrow (P \mid q)\]

**Tollens: (Modus Tollendo Tollens)**

\[((p \rightarrow q) \& !Q) \rightarrow !p\]

**Contraposition**

\[(p \rightarrow q) \equiv (!q \rightarrow !p)\]

**Absurdity:**

\[((p \rightarrow (q \& !q)) \rightarrow !p\]

**Hypothetical Syllogism:**

\[((p \rightarrow q) \& (q \rightarrow r)) \rightarrow (p \rightarrow r)\]

**Schrodinger’s Argument: (on light: wave or particle)**

\[((p1 \rightarrow p3) \& (p2 \rightarrow p4) \& (p0 \rightarrow (p1 \lor p2)) \& !(p3 \mid p4)) \rightarrow !p0\]
Coding Larger Truth Tables with Loops

Modify the following code to extend it from 2 logical variables to 3 variables. Have it print out the result of (a ^ b ^ c). Then extend it further to 4 variables and print out (a ^ b ^ c ^ d). Finally try it for 5 variables.

```java
// Does truth table defining Xor
// Using a while-true-break loop

boolean a, b;
System.out.print("  a  b ");
System.out.println("      a ^ b ");
a = false;
while (true) {
  b = false;
  while (true) {
    System.out.print(a + "\t");
    System.out.print(b + "\t");
    System.out.println(a ^ b);
    if (b) break;
    b = ! b;
  } // end while b
  if (a) break;
  a = ! a;
} // end while a
```

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>a ^ b</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
<td>false</td>
</tr>
</tbody>
</table>
Alternative Coding Larger Truth Tables with Loops

Following is yet another code to loop through logical value. Modify it to extend it from 2 logical variables to 3 variables. Have it print out the result of \((a \land b \land c)\). Then extend it further to 4 variables and print out \((a \land b \land c \land d)\). Finally try it for 5 variables.

```
// Does truth table defining Xor
// Using a do while loop

boolean a, b;
System.out.print (" a    b ");
System.out.println (" a ^ b ");

a = false;
do {
    b = false;
do {
        System.out.print (a + "\t");
        System.out.print (b + "\t");
        System.out.println (a ^ b);
        b = ! b;
    }while (b);
a = ! a;
}while (a);
```

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>a ^ b</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
<td>false</td>
</tr>
</tbody>
</table>
Arithmetic Logical Project (later, after units on functions, MOP)

Write logical operations as functions, such as:
and(p,q), ior(p,q), not(p), nor (p,q), imp(p,q), xor(p,q), equ(p,q)
The logical values are the integers 0 and 1 only.
DeMorgan’s Law can be written in this functional form as:
   equ (ior( not(p), not(q) ) , not (and (p,q)) )

Some definitions of the logical functions in terms of arithmetic operations are:
   not( a ) = 1 - a
   and(b,c) = b * c
   ior(d,e) = d + e - d * e

Complete all the rest of the logical operations,
by defining some in terms of arithmetic operations and
by defining some in terms of the other function operations.

Show how Modus Ponens can be written in this function notation.

Use nested for loops to create truth tables, (for majority of 3 and
use them to prove various properties (such as DeMorgan’s Laws) and
use them to disprove various properties (such as fallacies).

Character Logic
Write logical operations as functions involving truth values as characters ‘T’ and ‘F’.
Then do some of the above project.
Character Logic: with values ‘T’ and ‘F’ can be defined as follows; Do something with it!

```java
class CharLogicFunctions {

    public static final char T = 'T';
    public static final char F = 'F';

    public static char and (char x, char y) {
        if ((x == 'T') & (y == 'T'))
            return 'T';
        else
            return 'F';
    }

    public static char or (char x, char y) {
        if (x == T | y == T)
            return T;
        return F;
    }

    public static char not (char x) {
        if (x == T)
            return F;
        return T;
    }

    public static char imp (char x, char y) {
        if (x == T & y == F)
            return T;
        else
            return F;
    }

    public static char equ (char x, char y) {
        if (x == y)
            return T;
        return F;
    }

    public static void main (String[] args) {
        System.out.println ( or (T,T) );
    }
}
```
Converting Logical Forms (Infix to Postfix and back)

Convert the following logical formulas from infix to postfix form.

\[ !\left( a \mid b \right) \& \left( c \mid \neg d \right) \]

\[ !\left( a \& b \right) \mid !\left( \neg c \& d \right) \]

Convert the following logical formulas from postfix to infix form.

\[ a \! b \! c \& \]

\[ a \ b \& \ ! c \ ! d \& \ ! \]