True is not False: Evaluating Logical Expressions

Ronald J. Fehd, Stakana Analytics

Abstract

Description: The SAS® software language provides methods to evaluate logical expressions which then allow conditional execution of parts of programs. In cases where logical expressions contain combinations of intersection (and), negation (not), and union (or), later readers doing maintenance may question whether the expression is correct.

Purpose: The purpose of this paper is to provide a truth table of Boole’s rules, De Morgan’s laws, and sql joins for anyone writing complex conditional statements in data steps, macros, or procedures with a where clause.

Audience: programmers, intermediate to advanced users

Keywords: Boolean algebra, Boolean logic, De Morgan’s law, evaluation, logical operators, sql joins

In this paper

Introduction 1
Venn Diagrams of Logical Expressions 4
Truth Tables of Logical Expressions 5
Programs 6
Truth Table . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6
True is not False . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7
Summary 8
References 9

Introduction

Overview

This article combines the ideas of three logicians, Boole, De Morgan and Venn with the language of set theory and sql in order to assemble a table of logical expressions which describe each of the four permutations of pairs of true and false values.

The intent of this exercise is to provide a thesaurus for programmers who have specifications written by non-programmers.
The introduction contains these topics.

- natural language
- set theory
- sql
- comparison operators
- combinations, permutations
- four sets

**natural language**

Each natural language has a set of grammar rules about conjunctions that are used to describe pairs of ideas.

This is a list of common phrases; logical operators are in **text font**.

<table>
<thead>
<tr>
<th>phrase</th>
<th>operator</th>
<th>logic</th>
<th>join</th>
<th>union</th>
</tr>
</thead>
<tbody>
<tr>
<td>both ... and</td>
<td>and</td>
<td>disjunction</td>
<td>inner</td>
<td>intersection</td>
</tr>
<tr>
<td>either ... or</td>
<td>xor, exclusive</td>
<td></td>
<td>left, right</td>
<td>except</td>
</tr>
<tr>
<td>but not both</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>either ... or</td>
<td>or, inclusive</td>
<td></td>
<td>conjunction</td>
<td>full</td>
</tr>
<tr>
<td>neither ... nor</td>
<td>nor</td>
<td></td>
<td>union</td>
<td></td>
</tr>
</tbody>
</table>

**Note:** The words *also* and *only* are used in oral and written descriptions.

**set theory**

Set theory has four descriptions: union, intersection, set difference and symmetric difference.

<table>
<thead>
<tr>
<th>phrase</th>
<th>Boolean</th>
<th>written</th>
<th>spoken</th>
</tr>
</thead>
<tbody>
<tr>
<td>intersection</td>
<td>and</td>
<td>$A \cap B$</td>
<td>A cap B</td>
</tr>
<tr>
<td>set difference</td>
<td>xor(T,F)</td>
<td>$A \setminus B$</td>
<td>A and not B</td>
</tr>
<tr>
<td>symmetric difference</td>
<td>xor</td>
<td>$A \Delta B$</td>
<td>A xor B</td>
</tr>
<tr>
<td>union</td>
<td>or</td>
<td>$A \cup B$</td>
<td>A cup B</td>
</tr>
</tbody>
</table>

**sql**

Structured Query Language (sql) has two groups of operators, joins and unions.

**Note:** Some dialects of sql insert the word *outer* between the keywords *left, right, full* and *join*; e.g.: full outer join is equivalent to full join.

<table>
<thead>
<tr>
<th>type</th>
<th>operator</th>
<th>Boolean</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>joins</td>
<td>inner</td>
<td>and</td>
<td>only in both tables</td>
</tr>
<tr>
<td></td>
<td>left</td>
<td>xor(T,F)</td>
<td>only in left</td>
</tr>
<tr>
<td></td>
<td>right</td>
<td>xor(F,T)</td>
<td>only in right</td>
</tr>
<tr>
<td></td>
<td>full</td>
<td>or</td>
<td>all from both tables</td>
</tr>
<tr>
<td>unions</td>
<td>except</td>
<td>xor(T,F)</td>
<td>compare to left join</td>
</tr>
<tr>
<td></td>
<td>intersect</td>
<td>and</td>
<td>compare to inner join</td>
</tr>
<tr>
<td></td>
<td>union</td>
<td>or</td>
<td>compare to full join</td>
</tr>
</tbody>
</table>
**comparison operators**

A logical expression is evaluated according to the rules of Boolean algebra. Expressions with comparisons, or relations, are reduced to the Boolean set of values \((0, 1)\) with this set of operators.

- **parentheses**: evaluate expression inside
- **equality**: equal: \(eq, =\)
  - not equal: \(ne, \sim (\text{caret=}), \sim= (\text{tilde=})\)
- **quantity**: less than: \(lt, <\)
  - less than or equal: \(le, <=\)
  - greater than: \(gt, >\)
  - greater than or equal: \(ge, >=\)

**Note**: Other programming languages refer to this concept as *relational operators*.

---

**combinations, permutations**

The difference between a combination and a permutation is a very important idea in deconstructing logical expressions. The *permutations* of two values \(T\) and \(F\) are two sets: \((T, F)\) and \((F, T)\); but these two sets are different examples of the single *combination* \((T, F)\).

---

**four sets**

This table lists the four permutations of two expressions, Left \(L\), and Right \(R\), each with two values, true \(T\), and false \(F\).

<table>
<thead>
<tr>
<th>L</th>
<th>(T)</th>
<th>(T)</th>
<th>(F)</th>
<th>(F)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>(T)</td>
<td>(F)</td>
<td>(T)</td>
<td>(F)</td>
</tr>
</tbody>
</table>

The task is to use Boole’s operators and De Morgan’s Laws to uniquely identify each permutation.

---

**Overlapping Operators**

This set diagram shows which permutations return true for each operator. Notice that the operators *or* and *nor* define only one element as *True*, whereas *xor* and *nand* have two or more elements defined as *True*.

```
  or
 _xor
  (T,T) (T,F) (F,T) (F,F)
  and
  nor
  nand
```

This list provides the operators with common natural language constructs and explanations.

- **or**: inclusive or: one, or more, **True**
- **nor**: not or: neither **True**;
  - both values **False**
- **xor**: exclusive or: only one **True**, but not both (and)
- **and**: both **True**;
  - neither **False**
- **nand**: not and: one, or more, **False**
Venn Diagrams of Logical Expressions

Overview

John Venn was an English logician known for the visual representations of set theory known as Venn diagrams. The diagrams shown below illustrate the three operators and, xor and or with these three permutations of true and false values.

\[
\begin{array}{ccc}
(T, T) & (T, F) & (F, T) \\
\text{and} & \text{xor} & \text{or}
\end{array}
\]

name : and
phrase : both ... and
expression : L and R
logic : disjunction
join : inner
union : intersect

name : xor-left
phrase : only one
expression : L and not R

name : xor-right
phrase : only one
expression : not L and R

name : exclusive or
phrase : either ... or
expression : bxor(L, R)
union : except
union : (L union R)

name : inclusive or
phrase : either ... or
expression : L or R
logic : conjunction
join : full
Truth Tables of Logical Expressions

Overview

This section contains the following topics.

- overlapping sets
- expressions
- De Morgan’s Laws

overlapping sets

This table shows the four permutations of sets of pairs of values, $and(T,T)$, xor-left($T,F$), xor-right($F,T$), nor($F,F$), and the logical operators xor, or, and nand which include two or more of the basic four.

<table>
<thead>
<tr>
<th>name</th>
<th>values</th>
<th>or</th>
</tr>
</thead>
<tbody>
<tr>
<td>and</td>
<td>$T,T$</td>
<td></td>
</tr>
<tr>
<td>xor</td>
<td>$T,F$</td>
<td>$F,T$</td>
</tr>
<tr>
<td>nor</td>
<td>$F,F$</td>
<td></td>
</tr>
</tbody>
</table>

expressions

This table shows the logical expressions that are used to describe each of the four permutations of pairs of $(T,F)$ values.

<table>
<thead>
<tr>
<th>values</th>
<th>L and R</th>
<th>xor</th>
<th>or</th>
<th>nor</th>
<th>or</th>
</tr>
</thead>
<tbody>
<tr>
<td>L and R</td>
<td>$T,T$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>not(L and R)</td>
<td>not L or not R</td>
<td>$T$</td>
<td>$F$</td>
<td>$T$</td>
<td>$F$</td>
</tr>
<tr>
<td>not(L and R)</td>
<td>not L or not R</td>
<td>$F$</td>
<td>$T$</td>
<td>$T$</td>
<td>$F$</td>
</tr>
<tr>
<td>not(L and R)</td>
<td>not L or not R</td>
<td>$F$</td>
<td>$F$</td>
<td>$T$</td>
<td>$F$</td>
</tr>
</tbody>
</table>

De Morgan’s Laws

Augustus De Morgan was a contemporary of Boole. These rules are stated in formal logic. Conjunction means and; disjunction means or.

- **nand**: The negation of a conjunction is the disjunction of the negations.
- **nor**: The negation of a disjunction is the conjunction of the negations.

<table>
<thead>
<tr>
<th>name</th>
<th>expression</th>
<th>parentheses</th>
</tr>
</thead>
<tbody>
<tr>
<td>nand</td>
<td>not and not L or not R</td>
<td>no required</td>
</tr>
<tr>
<td>nor</td>
<td>not or not L and not R</td>
<td>no required</td>
</tr>
</tbody>
</table>
This program shows a truth table of the logical expressions defined above and their resolution.

### Programs

#### Truth Table

<table>
<thead>
<tr>
<th>nand.and</th>
<th>nand.or</th>
<th>and.L.R</th>
<th>and.L_not_R</th>
<th>xor =</th>
<th>or =</th>
<th>nor.and</th>
<th>nor.or</th>
</tr>
</thead>
<tbody>
<tr>
<td>not(L and R)</td>
<td>not L or not R</td>
<td>L and R</td>
<td>L and not R</td>
<td>xor(L,R)</td>
<td>L or R</td>
<td>not L and R</td>
<td>not L or R</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</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>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>

The truth table shows the outcomes of various logical expressions for different combinations of L and R. The table includes columns for nand-and, nand-or, xor-left, xor-right, nor-and, nor-or, and or, with the last two columns marked as duplicate due to redundancy.
This section contains programs for the following topics.

- data step function ifc
- implicit evaluation in macro expressions
- refactoring macro values with %eval

**data step function ifc**

The ifc function has four parameters:

1. a logical expression
2. character value returned when true
3. value returned when false
4. value returned when missing, which is optional

This example shows the function in a data step.

```
%let false = false;  
DATA test_ifc_integers;  
  attrib integer length = 4  
  text_if length = $ %length( false)  
  text_ifc length = $ %length(&false);  
  do integer = ., -1 to 2;  
    if not integer then text_if = 'false';  
    else text_ifc = ifc(integer, 'true', "&false", 'missing');  
    output;  
  end;  
run;
```

Note zero and missing are false, negative and positive values are true!

---

**Implicit Evaluation in Macro Expressions**

This program shows that the macro language performs an evaluation of an integer, similar to the data step function ifc.

```
%macro test_tf;  
%do value = -1 %to 2;  
  %if &value %then  
    %put &value is true;  
  %else  
    %put &value is false;  
%end;  
%mend;  
%test_tf;
```
Refactoring Macro Values With %eval

Many programmers provide a macro variable to use while debugging or testing. This macro variable may be initialized to any number of values representing false, such as (no, off,). The problem of checking for the correctly spelled value such as YES, Yes, yes, Y, y, ON, On, on, can be eliminated by recoding the value to boolean with this comparison expression,

! → %eval(0 ne &testing)
which acknowledges any value other than zero as true.

 %macro testing
  (testing=0 /* default: false, off */
   %*recode: any value turns testing on;
   %let testing = %eval(0 ne &testing);
   %if &testing %then %do;
     %put &=testing is true;
   %end;
   %else %do;
     %put &=testing is false;
   %end;
%mend;

Summary

Suggested Reading
people: [3], George Boole [2], Augustus De Morgan [4], John Venn
predecessors: [6] shows three logical and expressions to choose output data sets
[7] provides examples of checking command-line options during testing to add additional code to programs [8], using %sysfunc with ifc [9], macro design ideas
sets: [5] set theory

Conclusion

Evaluating logical expressions has two aspects, conversion of comparisons to boolean values and logical algebra using the operators not, and or. This paper provides the following benefits. The names of each of the permutations are and, xor-left, xor-right, and nor. Each permutation is identified using not with and. The names of sets of permutations are xor, or, and nand. Venn diagrams are provided for each of the permutations; these visual representations are helpful in understanding the sql concepts of addition and subtraction of the permutations. With this vocabulary and conceptual representations a programmer may be confident of understanding requirements and specifications no matter what language, discipline or scientific dialect they are written in.
Evaluating Logical Expressions

Truth Table

Writing Testing Aware Programs

Acknowledgements

Kirk Lafler and Søren Lassen reviewed a draft of this paper; each provided clarification on SQL concepts. Lassen noted his SAS-L post with reference to Celko’s SQL explanation.

Trademarks

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. In the USA and other countries © indicates USA registration. Other brand and product names are trademarks of their respective companies.

References


