

CS1Ah Lecture Note 18

Logical Expressions

This lecture note has two aims:

- to introduce *propositional logic*, for reasoning about *logical expressions*
- to describe the logical operators in the Java programming language.

The material in this lecture note is based closely on Chapter 12 of *Foundations of Computer Science* by Aho and Ullman.

18.1 Introduction

All programming languages have logical operators for building up logical (boolean valued) expressions. In Java, we might have the logical expression:

$a < b \mid (a \geq b \ \& \ c == d)$

Here, $\&$ is the ‘and’ operator and \mid is the ‘or’ operator, so this expression has the reading: *either (i) a is less than b, or (ii) a is greater or equal to b, and c equals d.*

Often logical expressions can be simplified, and the laws of propositional logic tell us how to do this. For example, the above expression can be simplified to:

$a < b \mid c == d$

18.2 Logical expressions in propositional logic

The components of a logical expression in propositional logic are:

- *logical constants*, t and f (denoting the *truth values* true and false respectively),
- *propositional variables* such as P , Q and others as needed, and
- the *logical operators* or *connectives* \neg (‘not’ or ‘negation’), \wedge (‘and’ or ‘conjunction’), \vee (‘or’ or ‘disjunction’), \Rightarrow (‘implies’ or ‘implication’) and \Leftrightarrow (‘if and only if’, ‘iff’, ‘equivalence’ or ‘bi-implication’).

A logical expression can be defined thus:

- The constants t and f are logical expressions.

- The propositional variables are logical expressions.
- If α is a logical expression then so are (α) and $\neg \alpha$.
- If α and β are logical expressions then so are $\alpha \wedge \beta$, $\alpha \vee \beta$, $\alpha \Rightarrow \beta$ and $\alpha \Leftrightarrow \beta$.

A common order of the operators for precedence purposes is, from high to low precedence, \neg , \wedge , \vee , \Rightarrow , \Leftrightarrow . The \wedge and \vee operators are usually assumed to be left-associative and the \Rightarrow operator to be right-associative.

Parentheses may be used to clarify the meaning of complex logical expressions even if the parentheses are strictly not needed. Sometimes \wedge and \vee are taken to have equal precedence, so it can be clearer to write $(A \wedge B) \vee C$ than $A \wedge B \vee C$.

18.3 Meaning of the logical operators

The meaning of the logical operators can be presented in a *truth table*. Each propositional variable in a logical expression can take on only two possible values, t and f . A truth table inspects a logical expression by examining all of the possible assignments of truth values to the propositional variables in the expression. For an expression which involves only two variables there are only four possible truth assignments so the table below has only four rows.

P	Q	$\neg P$	$P \wedge Q$	$P \vee Q$	$P \Rightarrow Q$	$P \Leftrightarrow Q$
f	f	t	f	f	t	t
f	t	t	f	t	t	f
t	f	f	f	t	f	f
t	t	f	t	t	t	t

Exercise: What is the value of the expression $f \Rightarrow f \Rightarrow f$?

The definitions above for the logical operators should capture the intuitive meaning of a combination of two propositions but there are a few cases which deserve closer attention:

Or is inclusive

The value of $t \vee t$ is t . That is to say, we consider \vee to be *inclusive or*, meaning “one or the other or both”. The alternative is *exclusive or*, meaning “one or the other but not both”. In everyday speech we use ‘or’ in an inclusive sense in some cases and in an exclusive sense in others. However, in propositional logic the \vee symbol is always inclusive. If we want a symbol for exclusive or, we often use \oplus .

False implies anything

We commonly read ‘ $P \Rightarrow Q$ ’ as ‘if P holds then Q holds’ or ‘ Q holds only if P holds’ (Saying ‘ P holds’ is the same as saying ‘ P is true’). The only way ‘ $P \Rightarrow Q$ ’ can be false is if P holds, but Q does not. In particular, if P is false, then ‘ $P \Rightarrow Q$ ’ is always true, no matter what the truth of Q is. Another angle on this is to note that, if we know ‘ $P \Rightarrow Q$ ’, but P is false, then no constraint is put on the value of Q .

Equivalence

The \Leftrightarrow operator is an equivalence operator. If ' $P \Leftrightarrow Q$ ' holds, then P and Q are either both true or both false. We prefer to use the symbol \Leftrightarrow rather than $=$ or \equiv for several reasons. One is that there are some logics in which ' $P \Leftrightarrow Q$ ' might be true even if ' $P \neq Q$ '. Another is that the low precedence of \Leftrightarrow is often convenient and can reduce the need for parentheses: $=$ and \equiv traditionally have precedences higher than all the logical operators.

18.4 Generating Truth Tables

Truth tables are used to calculate and show the truth values of logical expressions for every possible assignment of truth values to their propositional variables. For example, here is a truth table for: $\neg P \wedge Q \Rightarrow P \vee Q$:

P	Q	$\neg P$	$\neg P \wedge Q$	$P \vee Q$	$\neg P \wedge Q \Rightarrow P \vee Q$
f	f	t	f	f	t
t	f	f	f	t	t
f	t	t	t	t	t
t	t	f	f	t	t

On the left we give all possible combinations of truth values for the propositional variables in the expression. With 2 variables, there are 4 possibilities. With n variables, 2^n possibilities. We then add a column for each intermediate sub-expression of the expression. The values for each intermediate column are based on the values in appropriate previous columns and on the truth table for the sub-expression's outermost logical connective. For example, the values in the column for $\neg P \wedge Q$ are based on the truth table for \wedge and the columns for $\neg P$ and Q . Finally on the right we can calculate the truth values of the whole expression.

Note how each row shows how the value of the expression is calculated working from the leaves of the expression's syntax tree up through the internal nodes of the tree to the tree's root. Most programming languages compute values of logical expressions (and other kinds of expressions too) by working bottom up from the leaves of an expression to the root. In addition, Java always evaluates the left subtree of any binary operator before it moves on to evaluating the right subtree.

18.5 Tautologies, Satisfiability and Contradictions

A logical expression is

- a *tautology* if it always has the value t for all of the possible assignments of truth values to its propositional variables,
- *satisfiable* if it has the value t for at least one of the possible assignments of truth values to its propositional variables,
- a *contradiction* if it always has the value f for all of the possible assignments of truth values to its propositional variables.

Exercise: Check that the following are tautologies:

1. $P \vee \neg P$
2. $(P \Rightarrow Q) \vee (Q \Rightarrow P)$
3. $((P \Rightarrow Q) \Rightarrow P) \Rightarrow P$

18.6 Algebraic laws for logical expressions

A variety of tautologies have \Leftrightarrow as the outermost logical operator and so express some logical equivalence property. Here is a set Some useful tautologies that can be checked by completing truth tables are:

- | | | |
|-----|--|--|
| 1. | $P \Leftrightarrow P$ | \Leftrightarrow is reflexive |
| 2. | $P \wedge Q \Leftrightarrow Q \wedge P$ | \wedge is commutative |
| 3. | $P \vee Q \Leftrightarrow Q \vee P$ | \vee is commutative |
| 4. | $P \wedge (Q \wedge R) \Leftrightarrow (P \wedge Q) \wedge R$ | \wedge is associative |
| 5. | $P \vee (Q \vee R) \Leftrightarrow (P \vee Q) \vee R$ | \vee is associative |
| 6. | $P \wedge P \Leftrightarrow P$ | \wedge is idempotent |
| 7. | $P \vee P \Leftrightarrow P$ | \vee is idempotent |
| 8. | $P \wedge (Q \vee R) \Leftrightarrow (P \wedge Q) \vee (P \wedge R)$ | \wedge distributes over \vee |
| 9. | $P \vee (Q \wedge R) \Leftrightarrow (P \vee Q) \wedge (P \vee R)$ | \vee distributes over \wedge |
| 10. | $\neg \neg P \Leftrightarrow P$ | double negation elimination |
| 11. | $\neg(P \wedge Q) \Leftrightarrow \neg P \vee \neg Q$ | DeMorgan's Law for \neg over \wedge |
| 12. | $\neg(P \vee Q) \Leftrightarrow \neg P \wedge \neg Q$ | DeMorgan's Law for \neg over \vee |
| 13. | $P \wedge t \Leftrightarrow P$ | t is the identity for \wedge |
| 14. | $P \vee f \Leftrightarrow P$ | f is the identity for \vee |
| 15. | $P \wedge f \Leftrightarrow f$ | f is the annihilator for \wedge |
| 16. | $t \vee P \Leftrightarrow t$ | t is the annihilator for \vee |
| 17. | $P \wedge \neg P \Leftrightarrow f$ | |
| 18. | $P \vee \neg P \Leftrightarrow t$ | |
| 19. | $P \vee (P \wedge Q) \Leftrightarrow P$ | |
| 20. | $P \wedge (P \vee Q) \Leftrightarrow P$ | |
| 21. | $P \wedge (\neg P \vee Q) \Leftrightarrow P \wedge Q$ | |
| 22. | $P \vee (\neg P \wedge Q) \Leftrightarrow P \vee Q$ | |
| 23. | $(P \Rightarrow Q) \Leftrightarrow \neg P \vee Q$ | \Rightarrow characterisation |
| 24. | $\neg(P \Rightarrow Q) \Leftrightarrow P \wedge \neg Q$ | \neg of \Rightarrow characterisation |
| 25. | $(P \Rightarrow Q) \Leftrightarrow (\neg Q \Rightarrow \neg P)$ | |
| 26. | $(P \Leftrightarrow Q) \Leftrightarrow (P \wedge Q) \vee (\neg P \wedge \neg Q)$ | \Leftrightarrow characterisation |
| 27. | $\neg(P \Leftrightarrow Q) \Leftrightarrow (P \wedge \neg Q) \vee (\neg P \wedge Q)$ | \neg of \Leftrightarrow characterisation |
| 28. | $\neg(P \Leftrightarrow Q) \Leftrightarrow (\neg P \Leftrightarrow Q)$ | |
| 29. | $(P \Leftrightarrow Q) \Leftrightarrow (Q \Leftrightarrow P)$ | \Leftrightarrow is symmetric |
| 30. | $(P \Leftrightarrow Q) \Leftrightarrow (\neg P \Leftrightarrow \neg Q)$ | |
| 31. | $(P \Leftrightarrow Q) \Leftrightarrow (P \Rightarrow Q) \wedge (Q \Rightarrow P)$ | \Leftrightarrow is bi-implication . |

These tautologies can be used as equivalence rules in 'chain of equivalences' arguments, just as one uses equality in algebra. For example, to show that

$$(P \Rightarrow Q) \Leftrightarrow (\neg Q \Rightarrow \neg P)$$

one can reason as follows:

$$\begin{aligned}
 (P \Rightarrow Q) &\Leftrightarrow (\neg P \vee Q) && \text{characterisation of } \Rightarrow \\
 &\Leftrightarrow (Q \vee \neg P) && \text{commutativity of } \vee \\
 &\Leftrightarrow (\neg\neg Q \vee \neg P) && \text{double negation rule} \\
 &\Leftrightarrow (\neg Q \Rightarrow \neg P) && \text{characterisation of } \Rightarrow .
 \end{aligned}$$

To reason effectively with propositional equivalences, most of the above rules need to be committed to memory, or readily figured out when needed. It helps to note the many patterns in the rules. For example, those involving \wedge and \vee come in pairs where the \wedge s are swapped with \vee s and the ‘t’s are swapped with ‘f’s.

18.7 Logical expressions in Java

A subset of the logical expressions in Java can be defined thus:

- The constants `true` and `false` are logical expressions.
- The boolean variables are logical expressions.
- Equality tests and comparison expressions are logical expressions.
- If P is a logical expression then so are (P) and $!P$.
- If P and Q are logical expressions then so are $P \ \& \ Q$, $P \ \&\& \ Q$, $P \ | \ Q$, $P \ || \ Q$.

Strict and Conditional evaluation

Java has two versions of the logical ‘and’ and ‘or’ operators. The operator `&` is the *strict* version of logical ‘and’ and the operator `|` is the strict version of logical ‘or’. These always evaluate both their left and their right operand.

In contrast, `&&` is the *conditional* version of logical ‘and’ and `||` is the conditional version of logical ‘or’. These evaluate their left operand and, if this determines the result overall, they do not evaluate their right operand. Conditional operators are sometimes called *non-strict* or *short-circuiting* operators.

A typical use of conditional operators is in testing if an operation to follow is safe. Here, we want to test whether or not y divided by x is 2. Before testing this we check that x is not zero, because division by zero is undefined.

```
if(x != 0 && y/x == 2) { ... }
```

If we used the strict version of ‘and’ in this case, and x was zero, the division would still be performed and would throw an arithmetic exception.

Differences between Java and Propositional Logic

Logical expressions in Java and propositional logic are *not* the same. For example:

- The laws of propositional logic do not apply to boolean-valued Java expressions with side effects. $(i++ == 0) \ \& \ (i++ == 0)$ is not equivalent to $i++ == 0$.

- The conditional logical operators of Java are *not* commutative. $(x \neq 0) \ \&\& \ (y/x == 2)$ is not equivalent to $(y/x == 2) \ \&\& \ (x \neq 0)$.

*Authored by Stephen Gilmore, David Aspinall and Paul Jackson.
Paul Jackson, 2002/11/18 15:33:58.*