Unit 181
Boolean variables and logical operations

In the next cipher machines that we will see, characters are encoded with electric signals that can be on (1) or off (0), and signals pass from one part of a machine to another with similar values. The basic thing that can have a value of 0 or 1 is called a *bit*. When we are talking about manipulating bits with logic, we may label them with letters and call them *boolean variables* (named after George Boole), and we often use *T* or TRUE to represent 1 and *F* or FALSE to represent 0.

The first thing we might want to do to a boolean variable is apply an unary operation to it. An *unary operation* (*unop*) is an operation that acts on a single variable; in arithmetic the negation operation − is an unary operation: $-x$ has the value that is just as far from zero as x but on the other side of it. Suppose we have a boolean variable A and want to see what sorts of unary operations can be applied to it. We can tabulate all possible values of A in a *truth table* and add a column for the output of every possible unary operation. There are four possibilities:

A	UNOP_0	UNOP_1	UNOP_2	UNOP_3
0	0	0	1	1
1	0	1	0	1

Two of these are trivial: UNOP_0 always gives FALSE and UNOP_3 always gives TRUE. Operator UNOP_1 gives the same value that it receives; electrical engineers call this operator a *buffer*, but for us it is useless. Operator UNOP_2 is the interesting one: it flips the bit of A. This operation we call NOT, and we write "NOT A" or "$\neg A$" (or some people write "$\overline{A}$" or "$\sim A$"). At the risk of being overly pedantic, its truth table is in Table 181.1. We can forget about the other three unops.

A	$\neg A$
0	1
1	0

Table 181.1: The NOT *operation.*

Next we look at binary operations. A *binary operation* (*binop*) is one that acts on two variables. You are already familiar with this concept, since addition (+), subtraction (−), multiplication (·), and division (/) are binary operations on real numbers. Since we are dealing with two boolean variables, each of which can have one of two values, there are $2^4 = 16$ possibilities:

A	B	BINOP_0	BINOP_1	BINOP_2	BINOP_3	BINOP_4	BINOP_5	BINOP_6	BINOP_7
0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1

A	B	BINOP_8	BINOP_9	BINOP_{10}	BINOP_{11}	BINOP_{12}	BINOP_{13}	BINOP_{14}	BINOP_{15}
0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1

Again we have some trivial and useless operations: BINOP_0 and BINOP_{15} are constants, and BINOP_3, BINOP_5, BINOP_{12}, and BINOP_{10} are A, B, $\neg A$, and $\neg B$. That leaves ten interesting binary operations.

The AND *operation*

BINOP_1 is TRUE when and only when its operands A and B are both TRUE, so we can rename it the AND operation. It is equivalent to multiplication (modulo 2) (see the unit on modular arithmetic), and it has an identity element (1), and all non-zero elements (just 1) have inverses (still 1). Symbols for this operation are AND, $\wedge$, and $\cdot$.

A	B	$A \wedge B$
0	0	0
0	1	0
1	0	0
1	1	1

Table 181.2: The AND *operation.*

The OR *operation*

BINOP_7 is TRUE when A is TRUE or B is TRUE, and FALSE otherwise, so we call it the OR operation. Symbols for this operation are OR and $\vee$.

A	B	$A \vee B$
0	0	0
0	1	1
1	0	1
1	1	1

Table 181.3: The OR *operation.*

Exclusive or

BINOP_6 is TRUE if and only if exactly one of its operands is TRUE. Because it excludes the case when both operands are TRUE, we call rename it the *exclusive or* operation, or XOR. It is equivalent to addition modulo 2 (see the unit on modular arithmetic), and it has an identity element (0), and all elements have inverses ($-0 = 0$ and $-1 = 1$). It is also the same as inequality, since it is TRUE only when the operands are different. It is also the negation of a bi-implication (see below), meaning that the truth of one operand implies the falsehood of the other. Symbols for this operation include XOR, $\oplus$, $\veebar$, $\nleftrightarrow$, and $\not\equiv$, but we will usually use XOR and $\oplus$.

A	B	$A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

Table 181.4: The XOR *operation.*

Nor

BINOP_8 is TRUE when A OR B is FALSE, and FALSE when A OR B is TRUE, so we call it the NOR operation. Symbols for it include NOR, $\downarrow$, and $\bar{\vee}$.

A	B	$A \downarrow B$
0	0	1
0	1	0
1	0	0
1	1	0

Table 181.5: The NOR *operation.*

Not and

BINOP_{14} is the opposite of AND, so we call it the NAND operation. Symbols for it include NAND, $\uparrow$, and $\bar{\wedge}$.

A	B	$A \uparrow B$
0	0	1
0	1	1
1	0	1
1	1	0

Table 181.6: The NAND *operation.*

Implications

A statement like "if A then B" means that whenever A is TRUE then B is also TRUE. It also means that when A is FALSE, then B can have any value. We say that A *implies* B, and write $A \rightarrow B$. From the table of binary operations, we see that BINOP_{13} fits this description: it is TRUE if A and B are both TRUE and FALSE if A is TRUE but B is FALSE; if A is FALSE then the implication is vacuously satisfied, and BINOP_{13} is TRUE. There is no reason that we cannot have implication in the other direction. This is called the *converse* of $A \rightarrow B$, and it is the operation BINOP_{11}. If the operands imply each other, then they must be equal to get a TRUE result. We call this *bi-implication* or *equivalence* or *exclusive nor*. Symbols for it

include $\leftrightarrow$ and $\equiv$, and XNOR. This operation is the opposite (the NOT) of XOR, and is the one we used to call BINOP_9. Finally, BINOP_2 and BINOP_4 are the NOTs of $\rightarrow$ and $\leftarrow$.

A	B	$A \rightarrow B$	$A \leftarrow B$	$A \leftrightarrow B$	$A \nrightarrow B$	$A \nleftarrow B$
0	0	1	1	1	0	0
0	1	1	0	0	0	1
1	0	0	1	0	1	0
1	1	1	1	1	0	0

Table 181.7: Implication operations.

Python tips

Python has a data type called `bool`. A variable of this type can have value `True` or `False`, which are defined as constants in the Python language. However, in a logical expression, any non-zero value is treated as `True`, and any zero value or empty item, such as an empty string or list or dictionary, is treated as `False`. Logical operators `and`, `or`, and `not` do what you expect. The (bitwise) operator `^` can be used as XOR between two `bool` variables.

Reading and references

Any book on symbolic logic.

Python reference: https://docs.python.org/3/reference/expressions.html#boolean-operations

George Boole, *The Mathematical Analysis of Logic: Being an Essay towards a Calculus of Deductive Reasoning*, 1847, https://www.gutenberg.org/ebooks/36884

George Boole, *An Investigation of the Laws of Thought*, 1854.

Wikipedia has many pages about boolean arithmetic and operations. Start with https://en.wikipedia.org/wiki/Boolean_algebra

Exercises

1. Use truth tables to show these facts for any boolean variables A, B, C. Tabulate all possible values for the inputs and check that the column for the left-hand side matches the column for the right-hand side for each equation. This is called *proof by exhaustion*, not only because it is tiring, but also because it exhausts all possibilities.

   a. $A \rightarrow B \;=\; (A \wedge B) \vee (\neg A)$

   b. $(A \wedge B) \wedge C \;=\; A \wedge (B \wedge C)$ (associativity of AND)

   c. $(A \vee B) \vee C \;=\; A \vee (B \vee C)$ (associativity of OR)

   d. $A \wedge (B \vee C) \;=\; (A \wedge B) \vee (A \wedge C)$ (distributivity of AND over OR)

   e. $A \vee (B \wedge C) \;=\; (A \vee B) \wedge (A \vee C)$ (distributivity of OR over AND)

   f. $A \vee (A \wedge B) \;=\; A$ (absorption)

   g. $A \wedge (A \vee B) \;=\; A$ (absorption)

   h. $(A \wedge B) \vee (B \wedge C) \vee (C \wedge A) \;=\; (A \vee B) \wedge (B \vee C) \wedge (C \vee A)$

2. Show that De Morgan's laws hold for any boolean variables A and B:

   a. $\neg (A \wedge B) = (\neg A) \vee (\neg B)$

   b. $\neg (A \vee B) = (\neg A) \wedge (\neg B)$

3. Convince yourself that any binary operation can be written as an expression containing only the operations AND, OR, and NOT (and parentheses). Now convince me.

4. Write a convincing argument that any n-ary logical operation can be written as an expression containing only the operations AND, OR, and NOT (and parentheses). An n-ary operation is one that acts on n operands. We can think of such things as *binary functions* with n arguments (a function that takes n boolean values and returns one boolean value). If your argument is sound, then you have proven that {AND, OR, NOT} is a *complete* set of operators and that any binary function can be written in terms of these three operations.

5. Show that {AND, NOT} is a complete set of operators. Hint: use Exercise 4 and De Morgan's law(s).

6. Show that {OR, NOT} is a complete set of operators.

7. Show that {NAND} is a complete set of operators. That's right—everything can be written in terms of NAND.

8. There is another binary operation that forms a complete set by itself. What is it? Prove it.