Teaching guide: Boolean operations in programming languages

This resource will help with understanding Boolean operations in programming languages. It supports Section 3.2.5 of our current GCSE Computer Science specification (8520). The resource is designed to address the following learning outcomes:

- Use AND, OR and NOT in Boolean expressions
- Combine Boolean operators and relational operators to build complex Boolean expressions.

The AND operator

Let's start with two expressions that on their own are either True or False:

1. I am 14 years old
2. I am under 1.5 metres tall

You know whether expressions 1 and 2 are correct and so you can answer this expression that joins them with an AND:

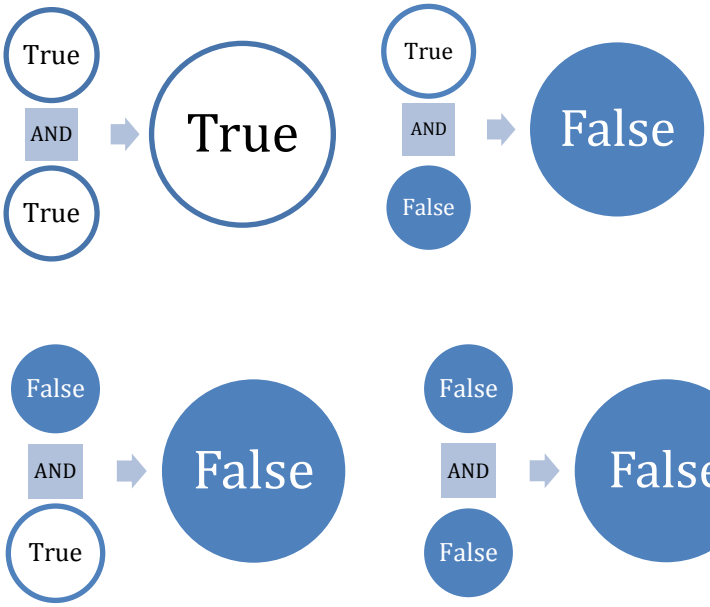
3. I am 14 years old AND I am under 1.5 metres tall

We can show the result of this expression with a table that has all of the four permutations, that the True/False values for sentences 1 and 2 can be put together:

I am 14 years old	I am under 1.5 metres tall	I am 14 years old AND I am under 1.5 metres tall
False	False	False
False	True	False
True	False	False
True	True	True

Sentence 3 is only True if both sentences 1 and 2 are also True. We can use AND with sentences 1 and 2 because they are both Boolean expressions. In fact

the AND operator takes any two Boolean expressions and evaluates to True only if both expressions themselves are True, otherwise it evaluates to False.



To evaluate the overall value of a Boolean expression involving an AND you have to first evaluate the Boolean expressions to the left and the right side of the AND and then use the rules above to work out if it is True or False, for example:

```
(4 ≠ 3) AND (1 < 5)
True AND True
True
```

```
((1 + 3) = 4) AND ((3 – 1) > 2)
True AND False
False
```

```
('a' = 'b') AND ('b' < 'c')
```

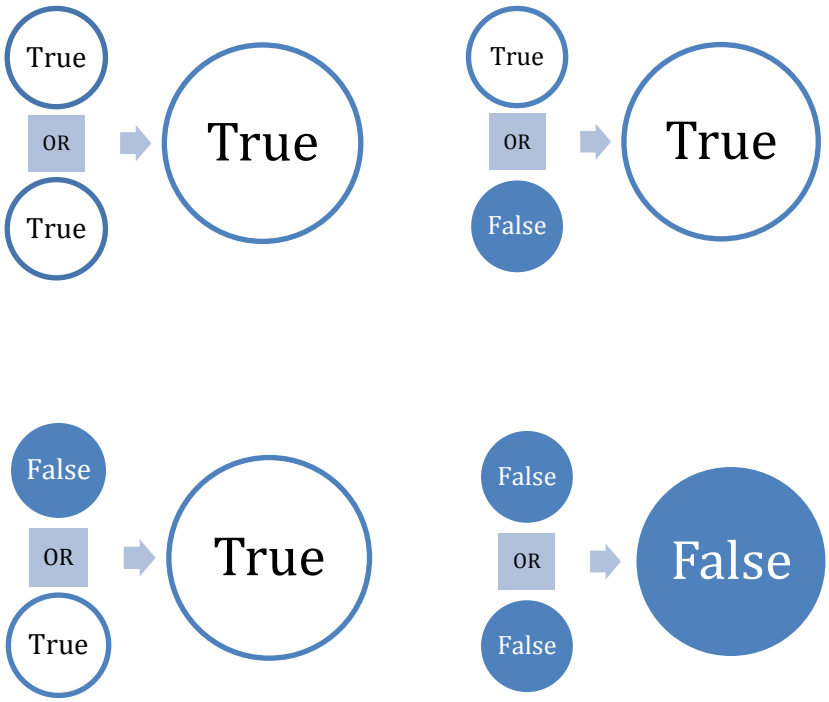
The left hand expression evaluates to `False`
There is no need to evaluate the right hand side, regardless of whether it is True or False the whole expression must be False.

The OR operator

Using OR is a very commonplace logical device whenever people are offered a choice: do you take the road on the left or the road on the right? Do you have custard or cream?

Each of these choices will (probably) involve choosing one or the other, but not both. This is where the use of OR in computer science differs from its use in natural language - logical OR evaluates to `True` if one or other of the choices is True, as well as if they are both `True`. An alternative way to view this is that OR only evaluates to `False` if the Boolean expressions to the left and the right are both `False`. There is a further logical operator called XOR (or exclusive-OR) that

requires either the left or the right side to be True but not both, however, this operator is beyond the requirements of the specification.



In exactly the same way as when evaluating expressions with AND, to evaluate OR you need to evaluate the Boolean expressions to either side of it.

```
(1 = 2) OR (2 = 3)
False OR False
False
```

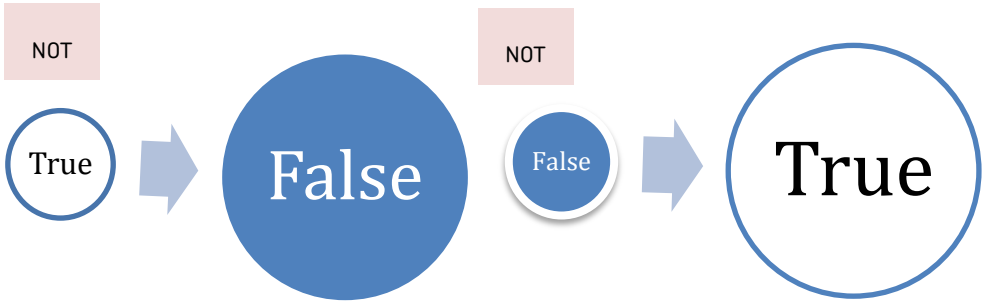
```
(3.4 < 2.9) OR (2.9 < 3.4)
False OR True
True
```

```
(1 ≠ 2) OR (1 > 2)
```

The left hand expression evaluates to `True` so it doesn't matter if the right hand expression evaluates to `True` or `False` as the overall value will evaluate to `True`.

The NOT operator

`AND` and `OR` are both binary Boolean operators which means they take two Boolean expressions and then give the result. There is another very useful operator that takes only one input (a unary operator) called `NOT`. `NOT` is the easiest of the Boolean operators to understand – whatever a Boolean expression evaluates to, if you put a `NOT` in front of it then it evaluates to the opposite. This is exactly how it is used in natural language: 'it is raining' is the opposite to 'it is not raining'.



NOT is the easiest to evaluate as it inverts whatever Boolean value it precedes as these examples show:

`NOT (4 > 3)`
`NOT True`
`False`

`NOT (4 ≠ 4)`
`NOT False`
`True`

Combining Boolean operators

All three Boolean operators (AND, OR, NOT) can be combined to create complex Boolean expressions. No matter how complicated they become, and how many operators they use, they will always evaluate to either `True` or `False`. They can be evaluated by working 'inside out'. The following examples explain this:

Example 1

`NOT ((4 = 5) OR (3 > 4))`

The only expressions we can evaluate are `(4 = 5)` and `(3 > 4)` which both evaluate to `False`. Our expression is now simplified to

`NOT (False OR False)`

We know from the diagram that `False OR False` evaluates to `False` and so our expression is now

`NOT False`

And we know this evaluates to `True`.

Example 2

`(3 ≠ 4) OR (NOT (5 > 3))`

We start by evaluating `(3 ≠ 3)` and `(5 > 3)` to `False` and `True` respectively. The expression can now be written

`False OR (NOT True)`

`NOT True` evaluates to `False`, and so we now have

`False OR False`

Which evaluates to `False`.

Common pitfall

It makes sense to say '3 and 5 are both less than 7', but if we were to write this directly as a program we would have:

```
(3 AND 5) < 7
```

Programming languages have formal 'grammars' that require expressions and statements to be written in specific ways and most programming languages have a rule that the left and right side of an AND have to themselves be Boolean expressions and here they are both integers. The correct way to write this expression follows the alternative way of saying the same thing in English, '3 is less than 7 and 5 is less than 7':

```
(3 < 7) AND (5 < 7)
```

This is a common mistake when creating complex Boolean expressions in programs. To further complicate things many programming languages have Boolean interpretations of numbers (typically 0 means `False` and any non-zero number is `True`) so the expression `(3 AND 5) < 7` is logically incorrect but may still be a valid expression in your language. When code is syntactically correct (it does not break any of the rules of the language such as misspelling a variable name or using one = sign instead of ==) but the code does not do what it is intended to, it contains logical errors. They are generally far harder to find than syntax errors.