Chapter 11
Logic and Language

11.1 A brief Introduction

There are a number of *fundamentals* in existence, and here I mention just two. The first concerns *language*. You are reading the text that I wrote for you, and in writing I held you in mind in order to *persuade,* or *cajole* you in understanding some ideas. So language is more than *communication;* it can provide *explanation* and so lead to understanding.

More formally, language can be used to express an *argument* or a *proof*; you will find such language in a court of law where prosecution and defence lawyers produce argument and counter-arguments in order to establish the guilt or innocence of the one on trial. In fact, it was the dream of the mathematician Leibnitz[1] to produce a system which could automate the resolution of such courtroom arguments.

Let's think for a moment of the following argument:

> If there is a fault, then it will explode.
> There is no fault.
> Therefore it will not explode.

Do you think this argument is true? Sure, it is very appealing, but it turns out it is false, and we shall see why.

The second fundamental is the idea of *symbols* and things they *represent*. We are all familiar with this symbol 😊 which represents an emotion. Or the symbol **x** in the code statement **int x = 0;** where the symbol **x** *represents* an integer number. The crucial thing about symbols is that they can represent concepts, even language clauses, e.g., in the above proof we could represent the second clause by the symbol **P** and write **P = There is no fault**

1

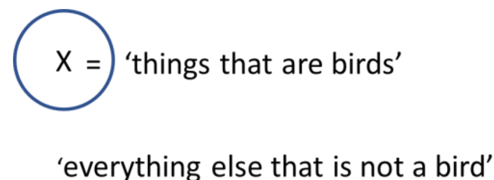
The third fundamental I wish to introduce here is the ubiquitous NAND gate, see Fig.1. It turns out that the electronics of an entire computer (both CPU and memory) can be made exclusively from NAND gates. So we don't actually need AND, OR and NOT gates and others too. Also the NAND gate is closely linked to language.

11.2 The Laws of Thought

Digital logic was given to us by the Englishman George Boole (1815 – 1864), though this was not then electronic digital logic (of course not!). It was the American Claude Shannon (1916 – 2001) who used Boolean Logic Algebra to construct telephone switching circuits.

Here we shall follow Boole's development of Logic from his treatise 'The Laws of Thought' (1854). Boole starts off by discussing the fact that symbols can represent concepts or language statements at great length.

Consider the following: **X = 'things that are birds'** where the symbol **X** is clearly representing birds. It's informative to express this as a diagram.

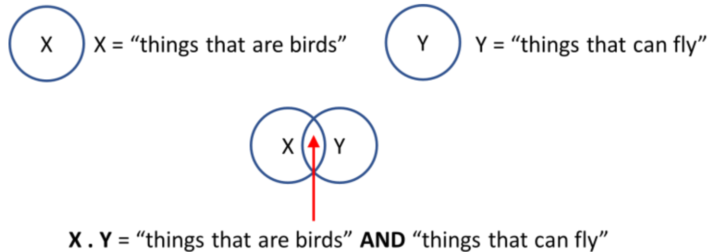


So all birds are inside the blue circle. Everything else that is not a bird is outside the circle. So we have divided reality into two sets (or classes) based on whether or not the thing we are looking at is a bird or not.

11.2.1 The And operation

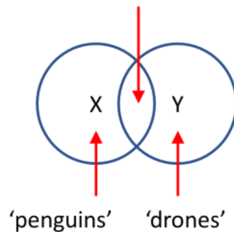
Things get interesting when we consider two sets and see how they interact. Consider the following: **X = 'things that are birds',** and **Y = 'things that can fly'**. We can look at the

intersection of these sets **X AND Y** which means **'things which are birds and which can fly'**. A diagram will help



You can clearly see the intersection of the two sets with the **AND** operation represented as the dot in **X . Y** (Boole didn't use dots). There are other regions in the diagram apart from the intersection, let's take a quick peek at these.

X . Y = 'things that are birds' AND 'things that can fly'

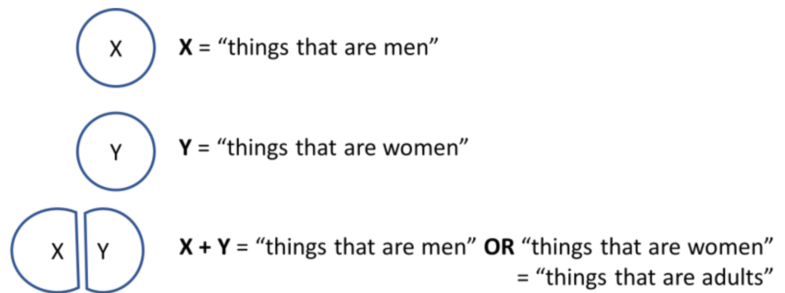


To the left of the intersection we have **Things which are birds which cannot fly** (they lie outside the circle **Y**) and penguins are an example. To the right of the intersection we have region in **Y** but outside **X**. These are things that can fly but which are not birds, such as aeroplanes and drones.

11.2.2 The Or operation

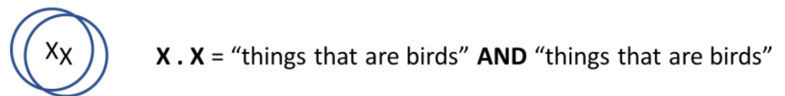
Boole introduced the symbol + in discussing the *union* of two sets. Consider the sets: **X = things that are men** and **Y = things that are women**. Then the union is

X + Y = things that are men OR women = adults. Here's a diagram to illustrate this.



11.2.3 A Fundamental Result

Boole goes on to do something a little unusual, but he recognizes the great importance of what he finds; this is the birth of the Binary system on which our machines are built. He considers the intersection of a set with itself like this

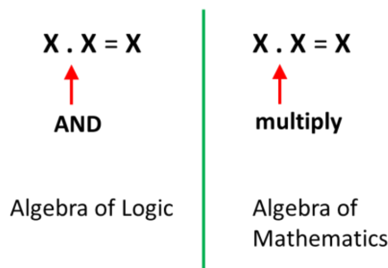


He concludes that the intersection can only be "things that are birds" (there's nothing else around) so he writes

X . X = X

Remember this is a *logical* expression expressed as the intersection of two sets and the dot symbol represents logical **AND**. It's at this point Boole sees a magical connection. He looks at the above expression and sees it as a *mathematical* expression where the dot represents **multiplication.** So the above expression is tantamount to asking the question "what number multiplied by itself gives itself". Boole deduces there are only two numbers that work **0** and **1** so he's discovered Binary numbers! Wow!

The nature of this connection between logic and mathematics is worthy of thought and we shall develop this later. For the moment let's summarise it in a little diagram.



Now that Boole has discovered the Binary numbers **1** and **0** in the realm of mathematics, he jumps back to the realm of logic and sets, and asks the question 'What do the symbols **0** and **1** mean in terms of sets, He soon reaches the conclusion

> **1** represents Everything
> **0** represents Nothing

This makes sense, since if **X** = 'things that are birds' then **1** – **X** means 'Everything minus things that are birds', or 'things that are not birds. So Boole establishes the **NOT** operation as **1** – **X** which we shall write as **~X.**

11.2.4 Truth Tables

We can use Boole's reasoning to understand those truth tables that may have been presented to us without explanation. Consider the truth table for logical **AND** shown below with the **AND** gate. On the right we have the truth

A
B
A . B

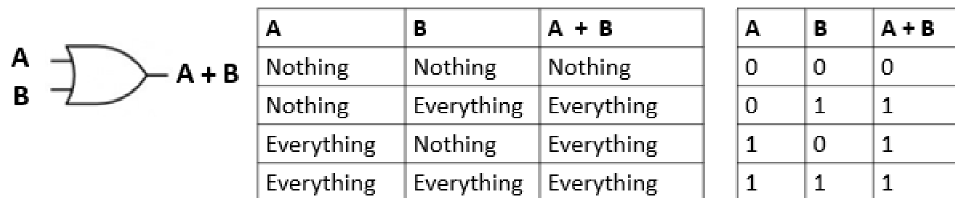
A	B	A . B
Nothing	Nothing	Nothing
Nothing	Everything	Nothing
Everything	Nothing	Nothing
Everything	Everything	Everything

A	B	A . B
0	0	0
0	1	0
1	0	0
1	1	1

table which we know. In the middle is the table constructed using Boole's interpretation of **1** as Everything and **0** as Nothing. So in the first row we have Nothing intersecting

with Nothing which of course gives Nothing. And the last row has Everything intersecting with Everything which of course gives Everything. Now, rows 2 and 3 are interesting since we have Everything overlapping with Nothing and this gives Nothing, which suggests that Nothing obliterates Everything. Boole returns to the realm of mathematics to offer some support for this, he notes that **0 . 1 = 0** (where the dot is now being interpreted as multiplication) and it is clear that if you multiply any number by **0** you will get **0**[2].

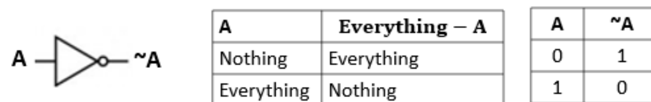
A similar justification for the **OR** gate can be made as in this diagram



A	B	A + B
Nothing	Nothing	Nothing
Nothing	Everything	Everything
Everything	Nothing	Everything
Everything	Everything	Everything

A	B	A + B
0	0	0
0	1	1
1	0	1
1	1	1

Here when the union of Nothing and Everything is taken (the **OR** operation) then clearly Everything results. Finally we have the **NOT** gate



A	Everything – A
Nothing	Everything
Everything	Nothing

A	~A
0	1
1	0

Here we are using Boole's interpretation of **1 – A** meaning 'things that are not A'.

---

[2] While you might agree with Boole, I feel he has made a conceptual error here, since mathematical addition does not correspond to logical OR for the or gate where 1 + 1 certainly does not equal 1.

11.3 From Logic to Language

Now that we understand how Logic emerged from language and sets, and how three important gates work, we can move onto using Logic to solve language problems. But first there is one more step we must take.

Consider the language of computer programming where we use the logical **AND** (e.g., **count && time**) the logical **OR** (e.g. **count || time**) and the logical **NOT** (e.g. **!count**). Well there is one more logical construct we use

```
if (condition) {

} else {

}
```

This presents us with an interesting question. Is it possible to create an **IF-**gate? The answer is Yes! To see this, we have to revisit Boole's fundamental result.

Let us rewrite this result as follows

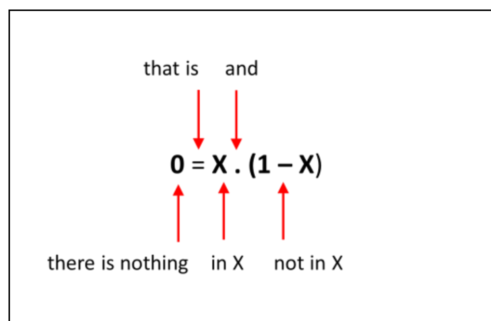
$$X . X = X$$

$$X.X - X.X = X - X.X \text{ (taking } X.X \text{ from both sides)}$$

$$0 = X - X.X \quad \text{(simplifying the left)}$$

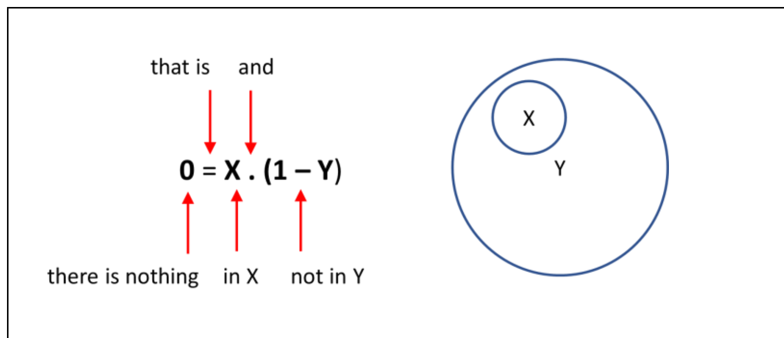
$$0 = X.(1 - X) \quad \text{(factoring out)}$$

and look in detail at the meaning of the last line



So the **0** on the left means ‘there is nothing’, the = sign means ‘that is’ , **X** means ‘in X’ and **1-X** means ‘not in X’. So in simple English this is telling us ‘There is nothing that is in X and is not in X’ which makes sense and you may think actually is obvious. You can’t be a man and not a man. You can’t be a student and not a student (though some students seem to try to defy this logic).

Now we shall make a jump and replace the **X** on the right with a **Y** so we are looking at **0** = **X . (1 – Y)** and here’s the interpretation



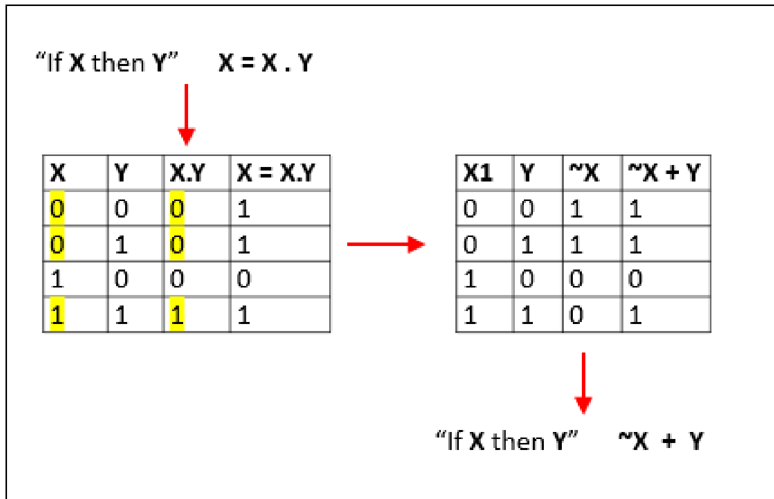
So the meaning is:

> There’s nothing in X that is not in Y.
> i.e., **IF** something is in X **then** it is in **Y.**

The set diagram clearly shows this fact. The set of **X** is completely contained in the set **Y**. To take this further, lets unwrap the above expression

> **0 = X . (1 – Y)**
> **0 = X – XY**
> **X = X . Y**

We can use this to create our **IF** gate. Simply we have, ‘If **X** then **Y’** is the same as the logical relation ‘**X = X.Y’.** So now let’s create the truth table for this and see how to implement it using standard logic gates. See below.



X	Y	X.Y	X = X.Y
0	0	0	1
0	1	0	1
1	0	0	0
1	1	1	1

X1	Y	~X	~X + Y
0	0	1	1
0	1	1	1
1	0	0	0
1	1	0	1

Starting at top left we introduce the expression. Then we create a truth table with the 4 values for **X** and **Y**. Then in the third column we calculate **X.Y** and in the fourth column we write a **1** when the third column is the same as the first, i.e., when **X** = **X.Y**. These values are highlighted.

Now we move to the right and get creative. Here we reproduce the fourth column of the first table using standard logic expressions. The result is that ~**X** + **Y** gives the identical fourth column. Therefore we have found the logic for our **IF** gate.

> if **X** then **Y** is realized by the logical expression ~**X** + **Y** or in simple English **'not X or Y'**

Let's take another peek at our prototype **IF** gate from the point of view of language, in particular the statement:

> If **Fred is at home** then **he is asleep**

We can divide this sentence into its two 'atomic' clauses:

> **X = Fred is at home**
> **Y = Fred is asleep**

So here comes the truth table where we have inserted the sentence values in the X-Y columns instead of (0,0), (0,1), (1,0) and (1,1) to make it clearer.

'If **Fred is at home** then **he is asleep**

X	**Y**		**~X + Y**
Fred is not home	Fred is not asleep	?	1
Fred is not home	Fred is asleep	?	1
Fred is home	Fred is not asleep	**0**	0
Fred is home	Fred is asleep	**1**	1

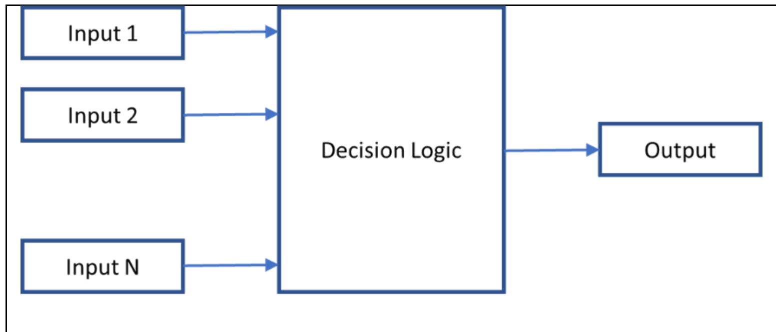
"If **X** then **Y**" **~X + Y**

The bottom two lines are straightforward. The line Fred is home, Fred is asleep is clearly true and follows from the starting sentence. The line Fred is home, Fred is not asleep is clearly false. So these unambiguous lines are highlighted.

The first and second lines are a little problematical, so I've put question marks here. The issue is that they refer to Fred not being at home, and the starting sentence contains no information about this. So we must make a choice, and we argue that in the case that Fred is not at home, then he may be asleep or he may be awake. So we agree that both of these cases must be true.

11.4 Sum-of-Products Circuits

Now let's turn to some actual problems encountered in class. The first set of problems follows the electronic engineering 'sum of products' solution strategy. For each problem we have several inputs, a single output and in the middle is a collection of logical gates which we call the 'decision logic'.



Let's look at this problem where we have two switches **A** and **B** as inputs and an output light **L**. The problem statement is:

> The light comes on in either of two cases:
> When A is pressed **and** B is pressed
> **or**
> When A is **not** pressed **and** B is pressed

(a) Complete the truth table. We only need to write the **1**s.

A	**B**	
0	0	
0	1	1
1	0	
1	1	1

(b) Add mini-terms to the truth table.

A	**B**		mini-terms
0	0		
0	1	1	**~A . B**
1	0		
1	1	1	**A . B**

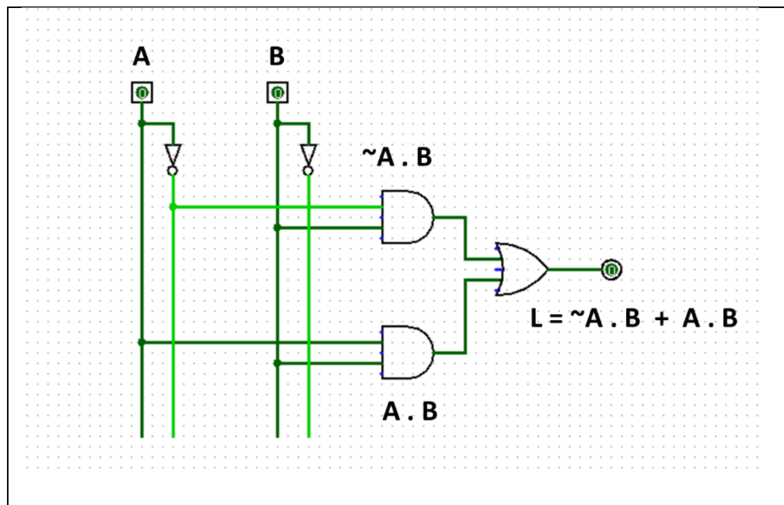
You create the mini-terms by looking at the values of **A** and **B** for that row, and writing an expression that is true. So for the second row **A = 0** and **B = 1** so the expression **~A.B** is just **1.1** which evaluates to true. In the last row both **A** and **B** are true so **A.B** evaluates to true.

(c) Now we deduce a logical expression for **L** the state of the light. The truth table shows that the light is on for rows 2 and 4, so we **or** the mini-terms for these two rows

L = ~A . B + A . B

You can see this captures correctly the initial problem statement.

(d) Next we build a digital simulated circuit using this expression (but you could keep an eye on the truth table too)



This way of laying things out conforms to standard electronic engineering practice. First we draw vertical wires for **A** and **B,** then we invert these and draw vertical wires for **~A** and **~B**. Then we use AND gates to form the two mini-terms, then we use an OR gate to combine the mini-terms to produce the output **L.** All sum-of-products problems will look like this, invertors on the left, ANDs in the middle, and a single OR at the right. This circuit can then be simulated and the output verified against the truth table.

It's interesting to look at the expression for **L.** Using the rules of Boolean algebra, we can attempt to simplify this expression. Here's the steps in the simplification:

L = ~A . B + A . B
= (~A + A) . B
= 1.B
= B

We have greatly simplified the solution. Check it is correct by inspecting the truth table; you will see the light is on when switch B is pressed.

In terms of language, we have greatly reduced the problem from the initial statement of :

The light comes on in either of two cases:
When A is pressed **and** B is pressed
or
When A is **not** pressed **and** B is pressed

to the much simpler one

The light is on when B is pressed

This language simplification is very important, why? Well if we are electronic engineers designing safety-critical circuits for the automotive or aerospace industries, then we need to describe simply and unambiguously the behaviour of our circuits, not only for engineers, but for salespersons and managers. The same is true for software engineers where we must be able to explain in language the logic behind our computer code.

11.5 Consistency Checks for Language

Now we shall turn to a deeper look at logic in language, and in particular look for consistency in a set of sentences. That means, is there any ‘solution’ to a set of sentences which is correct. This is not as clean as the sum-of-products but is a worthwhile study.

Consider the following set of sentences:

(1) Smiley is an English spy. (2) Smiley is not both a Russian spy and an English spy. (3) If Smiley is a Cad then he is a Russian spy.

(a) The first step is to extract the 'atomic clauses' from this set. This gives us:

A = Smiley is an English spy **B** = Smiley is a Russian spy **C =** Smiley is a Cad

Note we have not used any 'negations' in these.

(b) Now we use these to create Boolean expressions (mini-terms) for the set of sentences:

(1)	**A**
(2)	**~(A . B)**
(3)	**~C + B**

For (2) 'both a Russian spy **and** an English spy' would have been **A . B** so we've just negated that. For (3) we have used the **IF** gate.

Now we can use these expressions to build up a truth table and look for any possible mini-terms expressing truth.

A	B	C	A	~(A.B)	~C + B	1s	mini
0	0	0	0	1	1		
0	0	1	0	1	0		
0	1	0	0	1	1		
0	1	1	0	1	1		
1	0	0	1	1	1	1	**A. ~B . ~C**
1	0	1	1	1	0		
1	1	0	1	0	1		
1	1	1	1	0	1		

So let's see what we have. The columns labelled green correspond to our mini-terms (1), (2) and (3). The binary values filled in here come from the values for **A, B, C** on each row. In the column labelled **1s** we have put a **1** when all three mini-terms evaluate to true. This represents any consistent solution. Here we have a single consistent solution A . ~B . ~C which in simple English is:

Smiley is an English spy, not a Russian spy and not a Cad

11.6 Validation of Arguments

Let us return to the argument presented in the chapter introduction.

(1) If there is a fault, then it will explode.
(2) There is no fault.
(3) Therefore it will not explode.

To validate this argument, we proceed in a similar way to consistency checking. Except we use a trick. **We negate the conclusion** and do a consistency check on the new set of sentences. The idea is, if we find a consistent solution then this provides us with a *counter-example* where the original proof is invalid.

(a) First get the atomic sentences:

A = There is a fault
B = It will explode

(b) Now create mini-terms for the original set of sentences **negating the conclusion.**

(1) ~A + B
(2) ~A
(3) B

(c) Now build up the truth table.

A	B	~A + B	~A	B	1s	mini
0	0	1	1			
0	1	1	1	1	1	~A . B
1	0	0				
1	1	1		1		

So we have found one consistent solution. Therefore the original argument is **invalid** and the mini-term provides the counter example which is 'There is no fault but it will blow up'. This makes sense since it could blow up for another reason other than a fault, for example a bomb would blow up without a fault, since this is what it is designed to do.

11.7 Simplification of Boolean expressions

11.7.1 Logic Identities

There exist a number of logical identities, that is expressions where both sides of the = sign are identical. Here's a list

~ (~X) = X	Involution	
X + 0 = X		X . 1 = X
X + 1 = 1	Identity	X . 0 = 0
X + X = X	Idempotent	X . X = X
X + ~X = 1	Complement	X . ~X = 0
X + Y = Y + X	Commutativity	X . Y = Y . X
X + (Y + Z) = (X + Y) + Z	Associativity	X . (Y . Z) = (X . Y) . Z
X . (Y + Z) = X . Y + X . Z	Distributivity	X + Y.Z = (X + Y) . (X + Z)
(X + Y) . (W + Z) = X . W + X . Z + Y . W + Y . Z	Distributivity	
X + X . Y = X	Absorption	X . (X + Y) = X
~(X + Y) = ~X . ~Y	De Morgan	~(X . Y) = ~X + ~Y

The first thing to note is that there are two columns, the left is based on the or + conjunction and the right is based on the and **.** conjunction. These are called *duals* and we'll return to discuss this concept later. So what use are these identities?

Well, they enable us to *simplify* a Boolean expression and therefore express it in simpler language. So if we find something like **A + ~A** then we use the *complement* identity to simplify this to **1**. This makes sense since the sentence 'the switch is on or it is off' is always true. Some expressions are hard to 'see' but we can establish their truth by constructing a truth table for each side of the identity and showing that they are identical. For this example we have

A	~A	A + ~A
0	1	1
1	0	1

=

1
1
1

11.7.2 The Absorption Identity

This is a rather special critter and deserves some investigation. First let's establish its truth by creating the truth tables for both sides, which clearly works

X	Y	X.Y	X + X.Y
0	0	0	0
0	1	0	0
1	0	0	1
1	1	1	1

=

X
0
0
1
1

The question is why does this work? The answer lies in the column **X.Y**. The only case this is true is when both X and Y are true, and moreover when X is true (last row), so when we **or** this with **X** in this row, X is already **1** so nothing changes.

Now let's see how to apply this rule. Often you may come across an expression like this where you cannot apply the distributive rule

L = A + ~A.B

Here's how we proceed: From the original expression we can add an extra term. Since **X + X.Y = X**, this extra term must start with **A** in this example, and we have free choice for **Y.**

L = A + A.? + ~A.B

How to choose **?** ? Well, it can be **A, ~A, B, ~B**. To decide which, we look at the last term on the right and see how to craft the extra term so it can combine with this and hopefully simplify things. Here we choose **?** = **B** and then we can apply the distributive rule as the next step. Here's the complete simplification:

L = A + A.B + ~A.B

= A + B.(A + ~A)

= A + B.1

= A + B

11.7.3 De Morgan's Laws

These can be verified as usual by constructing truth tables for both sides of the expression which shows they are identical. Let's take **~X + ~Y = ~(X.Y)**

X	Y	~X + ~Y
0	0	1
0	1	1
1	0	1
1	1	0

=

X.Y	~(X.Y)
0	1
0	1
0	1
1	0

So if **X** = 'The door is closed' and **Y** = 'The window is closed' then ~X + ~Y means 'Either the door is open or the window

is open’, while **~(X.Y)** means ‘The door and the window are not both closed’. You may choose which you find simpler.

In the language of programming, we may need to do a compound test, e.g., to see if a user has entered both their ID and their password PW and throw an exception if this is not true. We may write this

if(!(userID && userPW)) ***throw exception***

which reads ‘If both the userID and userPW have not been entered, then throw an exception’ which we could using De Morgan write like this

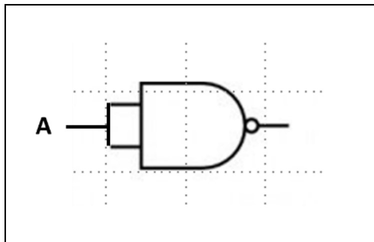
if(~userID || ~userPW) ***throw exception***

which means ‘If either the userID or userPW has not been entered, then throw an exception’. Perhaps you find one way easier to understand.

11.7.4 The Ubiquitous AND gate

Here we shall prove using De Morgan that all gates can be made from **NAND** gates.

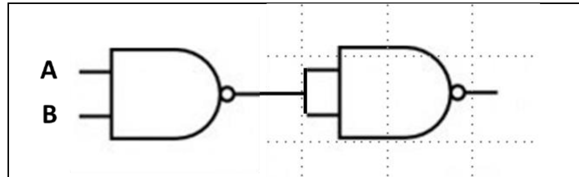
First, we establish the **NOT** gate. Consider this circuit where both inputs of the **NAND** gate are tied together so they get the same input **A**.



Now using De Morgan we find

$$\sim(A.B) = \sim A + \sim B$$
$$\sim(A.A) = \sim A + \sim A$$
$$\sim(A.A) = \sim A$$

so that the above circuit behaves like a **NOT** gate. Now onto the **AND** gate. This is straightforward since an **AND** is just a **NAND** followed by a **NOT** so here's an **AND** gate made from **NAND**s.



Now let's turn to the **OR** gate; this needs a little more work. Starting with De Morgan

~(A + B) = ~A . ~B

we negate both sides, so we get the OR gate

~ (~(A + B)) = ~ (~A . ~B)
i.e., **A + B = ~ (~A . ~B)**

Now before we move on, look at the meaning of the term on the right. This is of the form **~(X.Y)** which is 'not (X and Y)' which is of course a **NAND** gate. So the term on the right is just a **NAND** gate fed with **~A** and **~B** as its inputs. And we can create these two **NOT**s from **NAND**s. Wow! So, here's the circuit for an **OR** gate which is equivalent to three **NAND** gates.