Boolean Operations

Truth Tables

Truth Tables are a tool we use to compute the values of logical expressions over each combination of input value. For example, a truth table for the logical AND of two logic variables, *p* and *q*, would look something like this:

p	**q**	**p ^ q**
true	true	true
true	false	false
false	true	false
false	false	false

not

the *NOT* operation (~) is the logical negation of *p*. ~p = false if p = true and ~p = true if f = false.

Truth Table for ~p:

p	**~p**
true	false
false	true

or

the *OR* operation (∨) holds true when either *p* or *q* are true.

Truth table for p ∨ q:

p	**q**	**p ∨ q**
true	true	true
true	false	true
false	true	true
false	false	false

and

the *AND* operation (∧) holds true only when both *p* and *q* are true.

Truth table for p ∧ q:

p	**q**	**p ∧ q**
true	true	true
true	false	false
false	true	false
false	false	false

xor

The *EXCLUSIVE-OR* operation (⊕) holds true when either *p* or *q* are true, but not both.

Truth table for p ⊕ q:

p	**q**	**p ⊕ q**
true	true	false
true	false	true
false	true	true
false	false	false

Boolean Algebra

Example: Evaluate the boolean function: F(p,q) = p ∧ (p ⊕ q)

Let us construct a truth table:

p	q	p ⊕ q	p ∧ (p ⊕ q)
false	false	false	false
false	true	true	false
true	false	true	true
true	true	false	false

F(0,0) = 0
F(0,1) = 0
F(1,0) = 1
F(1,1) = 0

Boolean Algebra Identities

Identity	AND form	OR form
Identity Laws	p ∧ 1 = p	p ∨ 0 = p
Dominance Laws	p ∧ 0 = 0	p ∨ 1 = 1
Idempotent Laws	p ∧ p = p	p ∨ p = p
Complement Laws	p ∧ ~p = 0	p ∨ ~p = 1
Double Complement Laws	~(~p) = p	
Commutative Laws	p ∧ q = q ∧ p	p ∨ q = q ∨ p
Associative Laws	p ∧ (q ∧ r) = (p ∧ q) ∧ r	p ∨ (q ∨ r) = (p ∨ q) ∨ r
Distributive Laws	p ∧ (q ∨ r) = (p ∧ q) ∨ (p ∧ r)	p ∨ (q ∧ r) = (p ∨ q) ∧ (p ∨ r)
Absorption Laws	p ∧ (p ∨ q) = p	p ∨ (p ∧ q) = p
DeMorgan's Laws	~(p ∧ q) = ~p ∨ ~q	~(p ∨ q) = ~p ∧ ~q

We can use the identiies to help us evaluate boolean functions. Each law can be proved by either a truth table or by using other laws.

Example:Use a truth table to prove the distributive law over the AND operation:

Proposition: p ∧ (q ∨ r) = (p ∧ q) ∨ (p ∧ r)
Proof by construction using a truth table:

p	q	r	q ∨ r	p ∧ q	p ∧ r	p ∧ (q ∨ r)	(p ∧ q) ∨ (p ∧ r)
false	false	false	false	false	false	false	false
false	false	true	true	false	false	false	false
false	true	false	true	false	false	false	false
false	true	true	true	false	false	false	false
true	false	false	false	false	false	false	false
true	false	true	true	false	true	true	true
true	true	false	true	true	false	true	true
true	true	true	true	true	true	true	true

Example: Use the laws of Boolean Algebra to prove the Absorption Law over the AND operation:

```
Proposition: p ∧ ( p ∨ q ) = p

        p ∧ ( p ∨ q ) = (p ∧ p ) ∨ (p ∧ q )     by Distributive Law
                  = p ∨ ( p ∧ q )               by Idempotent Law
                  = p ∧ ( 1 ∨ q )               by Distributive Law
```

```
                = p ∧ 1                     by Dominance Law
                = p                         by Identity Law
```

bit level operations in C

C provides us with a set of bit-level operations including and (&) , or (|), not (~) and XOR (^).

```
Let x=0100 1110 , Let y=1101 0101

   ~x = 1011 0001

   x | y =
        0100 1110
        1101 0101
        ----------------
        1101 1111

  x & y =
        0100 1110
        1101 0101
        ----------------
        0100 0100

 x^y =
        0100 1110
        1101 0101
        ----------------
        1001 1011
```

C also provides us with two *shift* operations that allow us to shift bit patterns to the left or to the right.

left shift (<<) moves all bits to the left, dropping the most significant bits and filling the least significant bits with zeros

right shift (>>) moves all bits to the right, dropping the least significant bits. How it treats the most significant bits depends on the data type and the system. A **logical right shift** fills the most significant bits with zeros. An **arithmetic right shift** fills the most significant bits with repretitions of the most significant bit. This is done in order to preserve the sign which you will see is important when we take a look at integer representations.

```
        x << k     // left shift - drops the leading k bits, shifts all bits of x left by k, sets the k least significant digits to 0.

        x >> k    // right shift - drops the k least significant bits, shifts all bits of x right by k. k most significant bits are
                   //set according to logical or arithmetic shift
```

Examples:

```
Let x = 1100 0011, Let y = 0101 1101

        x << 4 = 0011 0000
            y << 3 = 1110 1000
            x >> 4 (logical) = 0000 1100
            x >> 4 (arithmetic) = 1111 1100
            y >> 3 (logical) = 0000 1011
            y >> 3 (arithmetic) = 0000 1011
```

The type of right shift depends on the data type and on the system. C itself, does not guarantee if a logical or arithmetic right shift occurs. If the data type is an unsigned integer, then logical right shift is performed. If the data type is signed (the default behavior), then it is system dependent. arithmetic shift is most commonly used, but it is not guaranteed so BUYER BEWARE. if you want your code to work, consider using an unsigned data type if you will be performing right shifts!

Question: Given a data type consisting of *w* bits, what happens if you try to shift by some value *k>=w*??

In C, surprise, this is undefined.. so it will be system dependent. On many systems it is treated as k mode w, so x << 32 would become x << 0 and x << 40 would become x << 8. But remember BUYER BEWARE. this behavior is not gauranteed, so be sure to keep your shift amounts less than the word size of the data type.

Question: How is 1<<2 + 3<<4 evaluated in C?

due to addition having a higher precendence than shift and the left-to-right associativity rule, it would be evaulated as: (1<<(2+3))<<4. To correctly evaluarte, we will need to enclose our shifts in paranthesis: (1<<2)+(3<<4)

Observe

```
Let x = 34 (10 0010)

    x<<2 = 100 1000   =   68
    x<<2 = 1001 0000  =  136

 Let x = 47 10 1111
    x>>2(logical) = 01 0111 = 23
```

x>>3(logical) = 00 1011 = 11