Background Theory

A bitwise number is a number in base 2. In base 2, place values are either a 1 or 0, depending on whether or not that power of two is in the sum to the number: For example, the decimal number 12 is written:

$$12_{10} = 8_{10} + 4_{10} = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 = 1100_2$$

See §**25.5.1: Bits and bit operations** for a great description of bitwise operations and their precedence. Also, `http://www.cprogramming.com/tutorial/bitwise_operators.html`.
To be sure, the precedence is also listed at `http://en.cppreference.com/w/cpp/language/operator_precedence`

precedence	operator	Description
1	`~`	`bitwise NOT`
2	`&`	`Bitwise AND`
3	`^`	`Bitwise XOR (exclusive or)`
4	`|`	`Bitwise OR (inclusive or)`

The interesting difference with the four function `(*,/,+,-)` calculator developed in chapter 6 and the bitwise calculator we develop here is that there are no right-associative operations: no "left to right" operations as in "+ and -, left to right." It's a strict hierarchy with only one operation at each level. Also, there's one more level.

Here's a first attempt at a grammar defining the syntax of our input before writing a program that implements the rules of that grammar.

```
AndExpression:
     Primary
     AndExpression "&" primary

XOrExpression:
     AndExpression
     XOrExpression "^" AndExpression

IOrExpression:
     XOrExpression
     IOrExpression "|" XOrExpression

Primary:
     Number
     "(" AndExpression ")"
     ~Number

Number:
     Integer
```

Let's look at how to adapt Stroustrup's calculator to evaluate bitwise expressions like this one, which only uses bytes: 8|6^10& ~ 6
= 00001000|00000110^00001010& ~ 00000110
= 00001000|00000110^00001010&11111001
= 00001000|00000110^00001000 = 00001000|00001110 = 00001110

1. Change the `Token` to have `int` values (2 bytes) instead of `double` values...but limit the output to 4 bits using `bitset<4>(value)`
2. Change all the types of the various functions from `double` to `int`, accordingly.
3. Change the operators to handle bitwise operations. For this you will need all new functions. Keep `primary()` but scrap `expression()` and `term()` and replace these with `iOrExp()`, `xOrExp()` and `andExp()`.

Here's some starter code:

```
#include "std_lib_facilities.h"
#include <bitset>

//------------------------------------------------------------------------------

const char number = '8';
const char print = ';';
const char quit = 'q';

//------------------------------------------------------------------------------


int iOrExp();
int xOrExp();
int andExp();
int primary();

class Token {
public:
    char kind;        // what kind of token
    int value;     // for numbers: a value
    Token(char ch)    // make a Token from a char
        :kind(ch), value(0) { }
    Token(char ch, int val)     // make a Token from a char and a double
        :kind(ch), value(val) { }
};

//------------------------------------------------------------------------------

class Token_stream {
public:
    Token_stream();   // make a Token_stream that reads from cin
    Token get();      // get a Token (get() is defined elsewhere)
    void putback(Token t);    // put a Token back
private:
    bool full;        // is there a Token in the buffer?
    Token buffer;     // here is where we keep a Token put back using putback()
};

//------------------------------------------------------------------------------

// The constructor just sets full to indicate that the buffer is empty:
Token_stream::Token_stream()
    :full(false), buffer(0)    // no Token in buffer
{ }

//------------------------------------------------------------------------------

// The putback() member function puts its argument back into the Token_stream's buffer:
void Token_stream::putback(Token t)
{
    if (full) error("putback() into a full buffer");
    buffer = t;       // copy t to buffer
    full = true;      // buffer is now full
```

```
55 }
57 //------------------------------------------------------------------------------
59 Token Token_stream::get()
   {
61     if (full) {       // do we already have a Token ready?
           // remove token from buffer
63         full=false;
           return buffer;
65     }
67     char ch;
       cin >> ch;    // note that >> skips whitespace (space, newline, tab, etc.)
69
       switch (ch) {
71     case ';':    // for "print"
       case 'q':    // for "quit"
73     case '(': case ')': case '|': case '^': case '&': case '~':
           return Token(ch);        // let each operation character represent itself
75     case '0': case '1': case '2': case '3': case '4':
       case '5': case '6': case '7': case '8': case '9':
77         {
               cin.putback(ch);         // put digit back into the input stream
79             int val;
               cin >> val;              // read a floating-point number
81             return Token(number,val);   // number is the const char '8'
           }
83     default:
           error("Bad token");
85     }
   }
87
   //------------------------------------------------------------------------------
89
   Token_stream ts;        // provides get() and putback()
91
   //------------------------------------------------------------------------------
93
   // deal with numbers, parentheses and 2's complement
95 int primary()
   {
97     Token t = ts.get();
       switch (t.kind) {
99     case '(':    // handle '(' expression ')'
       {
101        int d = iOrExp();
           t = ts.get();
103        if (t.kind != ')') error("')' expected");
           return d;
105    }
       case number:         // number is the const char '8'
107        return t.value;  // return the number's value
       case '~':  // handle negation
109        return ~primary();
       default:
111        error("primary expected");
       }
113 }
115 //------------------------------------------------------------------------------
117 // deal with |
    int iOrExp() //'inclusive or' expression
119 {
```

```
  // assign left the output of xOrExp();
  // create a Token, t, and assign it the output of ts.get()
    switch (t.kind) {
    case '|':
       // return the inclusive or of left with the output of iOrExp();
    default:
       // put t back into the token stream
       // return left;
    }
}

//------------------------------------------------------------------------------

// deal with ^
int xOrExp()
{
    // assign left the output of andExp()
    // create a Token, t, and assign it the output of ts.get()
    switch(t.kind) {
    case '^':
        // return the exclusive or of left with the output of xOrExp()
    default:
       // put t back into the token stream
       // return left;
    }
}

//------------------------------------------------------------------------------

int andExp() {
    // assign left the output of primary();
    // create a Token, t, and assign it the output of ts.get()
    switch(t.kind) {
    case '&':
        // return the and of left with the output of andExp();
    default:
       // put t back into the token stream
       // return left;
    }
}

//------------------------------------------------------------------------------

int main()
try
{
    int val = 0;
    while (cin) {
        Token t = ts.get();

        if (t.kind == quit) break;
        if (t.kind == print)
            cout << "=" << bitset<4>(val) << '\n';
        else
            ts.putback(t);
        val = iOrExp();
    }
}
catch (exception& e) {
    cerr << "error: " << e.what() << '\n';
    return 1;
}
catch (...) {
    cerr << "Oops: unknown exception!\n";
    return 2;
```

```
185 }
```

4. Do a check/expect to see that you get these sorts of correct output:

```
~8;
=0111
~(3|8);
=0100
~3&~8;
=0100
3|8^11;
=0011
3|8^11&15;
=0011
3|8^11&15|8;
=1011
3|8^11&15|8&9;
=1011
```

5. Use the bitwise calculator to test deMorgan's laws:

$$\sim [p|q] \equiv \sim p\& \sim q, \tag{1}$$

$$\sim [p\&q] \equiv \sim p| \sim q. \tag{2}$$