TP5

The course homepage is here:

`http://www.lsv.ens-cachan.fr/~schwoon/enseignement/systemes/ws1415/`.

You will find the slides from the course and some other files for the exercise there.

Details of shell commands and C functions can be obtained by using the `man` command.

The goal of this exercise is to work with the data representations for floating-point numbers and characters. For this, we need to manipulate data at the bit level.

1 Operations on bits

The C language has operators for bitwise operations on words. E.g., let x, y be variables of some integer type (`char`, `short int`, ...) and $\otimes$ some logical operator. Denote by x_i, y_i the i-th bit of x and y, respectively. Then result of $x \otimes y$ is the word z such that $z_i = x_i \otimes y_i$.

The operators are `&` (and), `|` (or), `^` (xor), `~` (not). Attention, these are not to be confused with the *logical* operators `&&`, `||`, etc, which merely test whether the operands are non-zero. Thus, `4&2` equals 0 but `4&&2` equals 1.

There exist shorthands for the binary operators, e.g. `x|=2` means `x=x|2`, and `x^=y` means `x=x^y`. Also, operators for *shifting* are available (`<<`, `>>`), e.g. `4<<2` equals 16.

1. Consider the following small fragment of a C program, where x, y are integer types.

   ```
   x ^= y;  y ^= x;  x ^= y;
   ```

   What does this fragment do?

2. Consider the following fragment, where `c,n` are integers. What value does `c` take as a function of `n`?

   ```
   for (c = 0; n != 0; n &= (n-1)) c++;
   ```

3. Following (b), when does the following expression yield non-zero?

   ```
   n & (n-1)
   ```

4. Write functions `getBit`, `setBit`, `toggleBit`, and `clearBit` that each take two arguments of type `int`; these are n and p such that p indicates the position of a bit in n (where 0 is the least significant bit). Function `getBit` should return whether the value of the p-th bit in n is 1. The other functions should return the value resulting from changing the p-th bit in n as the name of the function suggests.

2 Floating-point numbers

Recall that in C, the type `float` represents the 32-bit variant of the IEEE 754 standard, with 1 bit for the sign, 8 bits for the exponent, and 23 bits for the mantissa. Consider the following data type for storing the three components separately:

```
typedef struct { int sign; int exponent; int mantissa; } fc;
```

1. Write a function that decomposes a given `float` value into its three components (simply seen as integers, without further modifications). In other words, the input for the function is a `float`, its output of type `fc`. For instance, the bit pattern for $2.5 = 2^1 \cdot 1.25$ is

   ```
   0 . 1000 0000 . 010 0000 0000 0000 0000 0000
   ```

   (points added for better visualization). In this case, we represent the sign as 0, the exponent as 128 (hex 80), and the mantissa as 4194304 (hex 400000).

2. Write a function that does the inverse, i.e. take a structure of type `fc` and return the `float` value it represents.

3. Implement floating-point addition by writing a function that takes two structures of type `fc` and returns an `fc` structure representing their sum. To keep matters simple, we implement only a part of the addition procedure, making the simplifying assumptions that (i) both operands are positive, (ii) no overflow can happen, (iii) we do not deal with NaN/Inf etc.

   The addition takes part in three steps:

   (a) Bring both numbers to the same exponent, by shifting the mantissa of the smaller exponent accordingly.

   (b) Add the two resulting mantissas, keeping in mind the "implicit" 1 before their most significant bits.

   (c) Re-normalize the mantissa of the sum, adjusting the exponent accordingly.

3 Character encodings

As we saw in the course, there exist different ways to represent characters. A *character set* is a mapping of integers (also called *code points*) to characters (letters, digits, punctuation marks etc). The most important character sets that one encounters in a Western European context are:

- ASCII, whose domain is 0..127;
- the so-called Latin-1 (ISO 8859-1) extension of ASCII, covering the domain 128..255;
- Unicode, compatible with ASCII/Latin-1, but defining a much larger code space (hex 0..1FFFF).

A *character encoding* describes how to describe a code point (or more generally, a sequence of them). For ASCII/Latin-1, the encoding is trivial, each byte describes one code point. For Unicode, one uses a variable-length encoding called UTF-8 (which was discussed in the course). In this encoding, a code point is represented by 1 to 4 bytes.

1. On the course homepage, you find a link to a HTML page that does not display correctly because its author did not understand about character encodings. Save the file on your home page and correct it.

2. Write a function that takes a Unicode code point and outputs its UTF-8 representation. Use this function to correctly print the following names of cities on your console:

   L'Haÿ-les-Roses (France)
   Kroměříž (Czechia)
   Gödöllő (Hungary)