Non-Associativity of Addition

There an infinite number of integers. In the mid 1800's it was noticed by Georg Cantor and others that there are significantly more real numbers. The integers **Z** are infinite; whereas, the real numbers **R** are uncountably infinite. This means there is no possible way to put the elements of **R** in a one-to-one correspondence with elements of the **Z**.

A modern digital computer, no matter how large the memory is, can only store a finite number of digits, and hence only a finite number of integers. Almost all real numbers must approximated. Storing and working with arbitrary functions $f : \mathbf{R} \to \mathbf{R}$ presents an even more difficult problem. In the future, quantum computers or some other device may be able to calculate with arbitrary real-valued functions as primitive data types, or not.

There are two common ways to store approximations of real numbers on a digital computer: fixed point and floating point. In the early days of computing, floating point was considered a theoretical abstraction that was too slow to be of practical use in real programs. It was nice, because it provided a uniform bound on the relative error of the approximation over a wide range of real numbers, but all computation was done using carefully scaled fixed point arithmetic.

With the creation of efficient floating point accelerators followed by modern CPUs that contain multiple FPUs, floating point computation has become practical. Floating point is still slower than fixed point, and there are some computations where the faster speed of fixed point arithmetic is necessary, notably real-time digital signal processing, computer graphics and the compression of digital video. However, all general purpose programming languages today possess at least one primitive floating point datatype, and floating point arithmetic has become the standard for scientific computation.

A fixed point number is stored in fixed number of memory locations with a decimal point given in a fixed place. Let `D` stand for an arbitrary decimal digit, then the 5 digit decimal fixed point representation ±`DDD.DD` could be used to approximate any real number between -999.995 and 999.995 accurately to the nearest cent. Note that the number 0.001 would be approximated by $+000.00$ in this representation and although this is accurate to the nearest cent, the relative error of this approximation is

$$\frac{0 - 0.001}{0.001} = -1$$

or, in other words, 100% relative error.

A floating point number is stored as scientific notation: with a mantissa and an exponent. Thus, a possible decimal floating point representation is

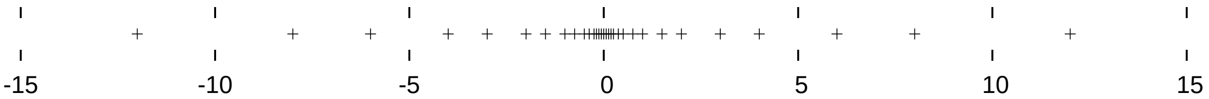
$$\pm\texttt{D.DDDDDDD} \times 10^{\pm\texttt{DD}}.$$

Of course, most real computers use a binary, or base 2, representation. Consider the the following base 2 example with

$$\pm\texttt{B.B} \times 2^{\pm\texttt{BB}}.$$

where each `B` stands for an single arbitrary binary digit. This representation can express the following (decimal) numbers exactly: -12, -8, -6, -4, -3, -2, -1.5, -1, -0.75, -0.5, -0.375, -0.25, -0.1875, -0.125, -0.0625, 0, 0.0625, 0.125, 0.1875, 0.25, 0.375, 0.5, 0.75, 1, 1.5, 2, 3, 4, 6, 8, 12. Notice how the real numbers which are exactly representable are spaced farther apart the further from zero they are. Plotting these values on the number line gives a clearer picture of what is happening.

Number Line



One disappointing consequence of the fact that floating point numbers are not uniformly spaced on the number line is that the associativity property of real numbers is not preserved by floating point arithmetic. Given $x \in \mathbf{R}$, let x^* denote the best floating point approximation to x. Then, the floating point addition operator $+^*$ is given by

$$x +^* y = (x + y)^*.$$

For illustration we consider the decimal floating point representation given by

$$\pm\texttt{D.D} \times 10^{\pm\texttt{DD}}.$$

These are the decimal numbers between -9.95×10^{99} and 9.95×10^{99} that have been rounded to two significant digits. Since

$$2.3 \times 10^1 + 4.4 \times 10^0 = 23 + 4.4 = 27.4,$$

it follows that

$$2.3 \times 10^1 +^* 4.4 \times 10^0 = 2.7 \times 10^1.$$

Suppose $a = 2.3 \times 10^1$, $b = 4.4 \times 10^0$ and $c = 4.1 \times 10^{-1}$. Then

$$(a +^* b) +^* c = 27 +^* 0.41 = 27$$

whereas

$$a +^* (b +^* c) = 23 +^* 4.8 = 28.$$

Similar examples can be constructed for binary floating point representations with more significant bits.

When the C programming language was first designed, it was not explicitly stated in what order a mathematically associative expression such as $a + b + c$ should be evaluated. In fact, the language definition given by Kernighan and Ritchie 1978 explicitly stated

> ... *expressions involving one of the associative and commutative operators* (`*`, `+`, `&`, `^`, `|`) *can be rearranged even when parenthesized.*

Therefore writing

Code Example 3a

```
r=(a+b)+c;
```

in an early C compiler did not guarantee that the operation $a + b$ would be performed first and not $b + c$ or even $a + c$ instead. Allowing a compiler the freedom to evaluate a given expression using any mathematically equivalent form allows for certain optimizations affecting register allocation and reuse. However, this freedom was restricted in the ANSI C standard. Thus, in ANSI mode, a compliant C compiler is guaranteed to evaluate the parenthesized subexpression $a + b$ first in Code Example 3a.