5-2009

Zeros and ones

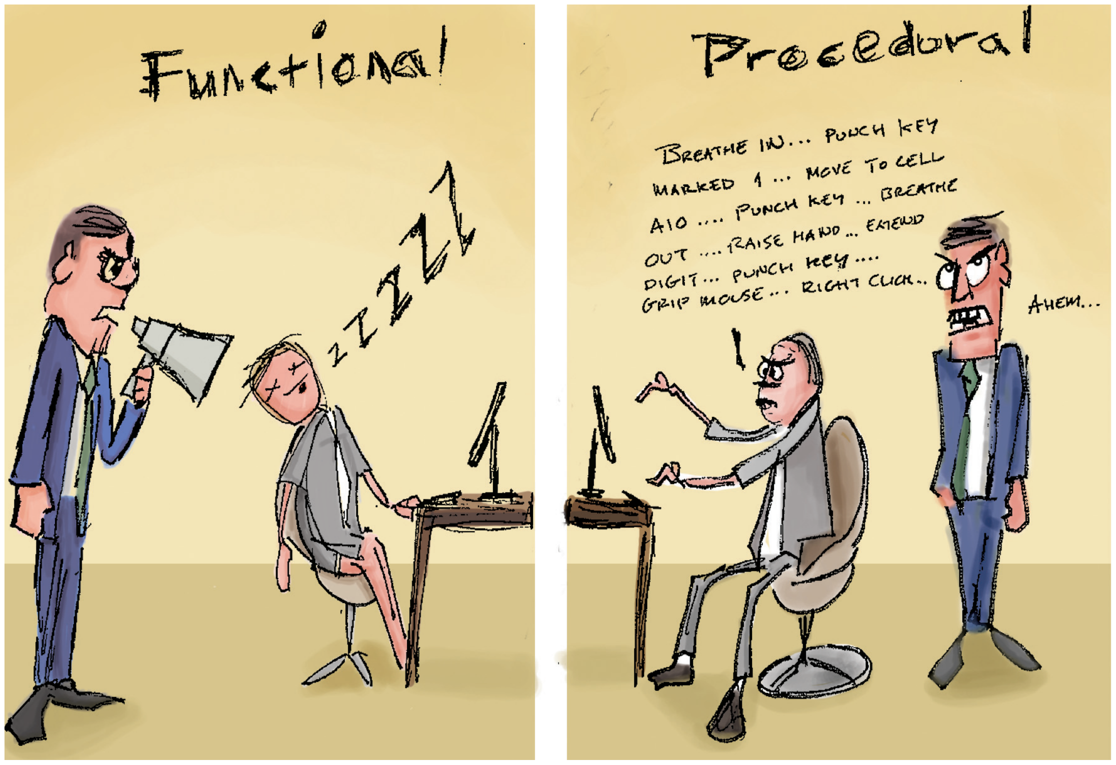
M. THULASIDAS

Singapore Management University, manojt@smu.edu.sg

Manoj Thulasidas

Zeros and Ones

Spare a thought for the way your glorified adding machine makes sense of things ...


Functional
ZZZZZZ
Procedural
BREATHE IN... PUNCH KEY MARKED 1 ... MOVE TO CELL A10 .... PUNCH KEY ... BREATHE OUT .... RAISE HAND ... EXTEND DIGIT... PUNCH KEY .... GRIP MOUSE... RIGHT CLICK..
!
AHEM...

Workplace paradigms

Computers are notorious for their infuriatingly literal obedience. I am sure anyone who has ever worked with a computer has come across the lack of empathy on its part – it follows our instructions to the dot, yet ends up accomplishing something altogether different from what we intend. We have all been bitten in the rear end by this literal adherence to logic at the expense of commonsense. We can attribute at least some of the blame to our lack of understanding (yes, literal and complete understanding) of the paradigms used in computing.

Paradigms all the way

Paradigms permeate almost all aspects of computing. Some of these paradigms are natural. For instance, it is natural to talk about an image or a song when we actually mean a JPEG or an MP3 file. "File" is already an abstraction evolved in the file-folder paradigm popularized in Windows systems. The underlying objects or streams are again abstractions for patterns of ones and zeros, which represent voltage levels in transistors, or spin states on a magnetic disk. There is an endless hierarchy of paradigms. Like the proverbial turtles that confounded Bertrand Russell (or was it Samuel Johnson?), it is paradigms all the way down.

Some paradigms have faded into the background, although the terminology evolved from them lingers. The original paradigm for computer networks (and of the Internet) was a mesh of interconnections residing in the sky above. This view is more or less replaced by the World Wide Web residing on the ground at our level. But we still use the original paradigm whenever we say "download" or "upload." The World Wide Web, by the way, is represented by the acronym www, which figures in the name of all Web sites. It is an acronym with the dubious distinction of being about the only one that takes us longer to say than what it stands for. But, getting back to our topic, paradigms are powerful and useful means to guide our interactions with unfamiliar systems and environments, especially in computers, which are strange and complicated beasts to begin with.

A basic computer processor is deceptively simple. It is a string of gates. A gate is a switch (more or less) made up of a small group of transistors. A 32-bit processor has 32 switches in an array. Each switch can be either off, representing a zero, or on (one). And a processor can do only one function – add the contents of another array of gates (called a register) to itself. In other words, it can only "accumulate."

In writing this last sentence, I have already started a process of abstraction. I wrote "contents," thinking of the register as a container holding numbers. It is the power of multiple levels of abstraction, each of which is simple and obvious, but building on whatever comes before it, that makes a computer enormously powerful.

We can see abstractions, followed by the modularization of the abstracted concept, in every aspect of computing, both hardware and software. Groups of transistors become arrays of gates, and then processors, registers, cache, or memory. Accumulations (additions) become all arithmetic operations, string manipulations, user interfaces, image and video editing, and so on.

Another feature of computing that aids in the seemingly endless march of the Moore's Law (which states that computers will double in their power every 18 months) is that each advance seems to fuel further advances, generating an explosive growth. The first compiler, for instance, was written in the primitive assembler level language. The second one was written using the first one, and so on. Even in hardware development, one generation of computers become the tools in designing the next generation, stoking a seemingly inexorable cycle of development.

While this positive feedback in hardware and software is a good thing, the explosive nature of growth may take us in wrong directions, much like the strong growth in the credit market led to the banking collapses of 2008. Many computing experts now wonder whether the object-oriented technology has been overplayed.

Magic of object-oriented languages

Nowhere is the dominance of paradigms more obvious than in object-oriented languages. Just take a look at the words that we use to describe some their features: polymorphism, inheritance, virtual, abstract, overloading – all of them normal (or near-normal) everyday words, but signifying notions and concepts quite far from their literal meaning. Yet, and here is the rub, their meaning in the computing context seems exquisitely appropriate. Is it a sign that we have taken these paradigms too far? Perhaps. After all, the "object" in object-oriented programming is already an abstract paradigm, having nothing to do with "*That Obscure Object of Desire*," for instance.

We do see the abstraction process running a bit wild in design patterns. When a pattern calls itself a visitor or a factory, it takes a geekily forgiving heart to grant the poetic license silently usurped. Design patterns, despite the liberties they take with our sensitivities, add enormous power to object-oriented programming, which is already very powerful, with all the built-in features like polymorphism, inheritance, overloading, and so on.

To someone with an exclusive background in sequential programming, all these features of object-oriented languages may seem like pure magic. But most of the features are really extensions or variations on their sequential programming equivalents. A class is merely a structure, and can even be declared as such in C++. When you add a method in a class, you can imagine that the compiler is secretly adding a global function with an extra argument (the reference to the object) and a unique identifier (say, a hash value of the class name). Polymorphic functions also can be implemented by adding a hash value of the function signature to the function names, and putting them in the global scope.

The real value of the object-oriented methodology is that it encourages good design. But good programming discipline goes beyond mere adaptation of an object-oriented language, which is why my first C++ teacher said: "You can write bad Fortran in C++ if you really want. Just that you have to work a little harder to do it."

For all their magical powers, the object-oriented programming languages all suffer from some common weaknesses. One of their major disadvantages is, in fact, one of the basic design features of object-oriented programming. Objects are memory locations containing data, as laid down by the programmer (and the computer). Memory locations remember the state of the object – by design. What state an object is in determines what it does when a method is invoked. So, the object oriented-approach is inherently stateful, if we can agree on what "state" means in the object-oriented context.

But in a user interface, where we do not have much control over the sequence in which various steps are executed, we might get erroneous results in stateful programming, depending on what step gets executed and at what point in time. Such considerations are especially important when we work with parallel computers in complex situations. One desirable property in such cases is that the functions return a number solely based on their arguments. This property, termed "purity," is the basic design goal of most functional languages, although their architects will concede that most of them are not strictly "pure."

Functional programming

Functional programming is the programming methodology that puts great emphasis on statelessness and religiously avoids side effects of one function in the evaluation of any other function. Functions in this methodology are like mathematical functions. The conventional programming style, on the other hand, is considered "imperative" and uses states and their changes for accomplishing computing tasks.

Adapting this notion of functional programming may sound like regressing back to the preobject-oriented age, and sacrificing all the advantages thereof. But there are practitioners, both in academia and in the industry, who strongly believe that functional languages are the only approach that ensures stability and robustness in financial and number crunching applications.

Functional languages, by definition, are stateless. They do everything through functions, which return results that are, well, functions of their arguments. This statelessness immediately makes the functions behave like their mathematical counterparts. Similarly, in a functional language, variables behave like mathematical variables rather than labels for memory locations. And a statement like $x = x + 1$ would make no sense. After all, it makes no sense in real life either.

This strong mathematical underpinning makes functional programming the darling of mathematicians. A piece of code written in a functional programming language is a set of declarations quite unlike a standard computer language such as C or C++, where the code represents a series of instructions for the computer. In other words, a functional language is declarative – its statements are mathematical declarations of facts and relationships, which is another reason why a statement like $x = x + 1$ would be illegal.

The declarative nature of the language makes it "lazy," meaning that it computes a result only when we ask for it. (At least, that is the principle. In real life, full computational laziness may be difficult to achieve.)

Computational laziness makes a functional programming language capable of handling many situations that would be impossible or exceedingly difficult for procedural languages. Users of Mathematica, which is a functional language for symbolic manipulation of mathematical equations, would immediately appreciate the advantages of computational laziness and other functional features such as its declarative nature. In Mathematica, we can carry out an operation like solving an equation, for instance. Once that is done, we can add a few more constraints at the bottom of our notebook, scroll up to the command to solve the original equation, and re-execute it, fully expecting the later constraints to be respected. They will be, because a statement appearing at a later part in the program listing is not some instruction to be carried out at a later point in a sequence. It is merely a mathematical declaration of truism, no matter where it appears.

This affinity of functional languages toward mathematics may appeal to quants as well, who are, after all, mathematicians of the applied kind. To see where the appeal stems from, let us consider a simple example of computing the factorial of an integer. In C or C++, we can write a factorial function either using a loop or making use of recursion. In a functional language, on the other hand, we merely restate the mathematical definition, using the syntax of the language we are working with. In mathematics, we define factorial as:

$$n! = \begin{cases} 1 & n = 1 \\ n \times (n-1)! & \text{otherwise} \end{cases} \tag{1}$$

And in Haskell (a well known functional programming language), we can write:

$$\begin{aligned} &\text{bang } 1 = 1 \\ &\text{bang } n = n * \text{bang } (n-1) \end{aligned} \tag{2}$$

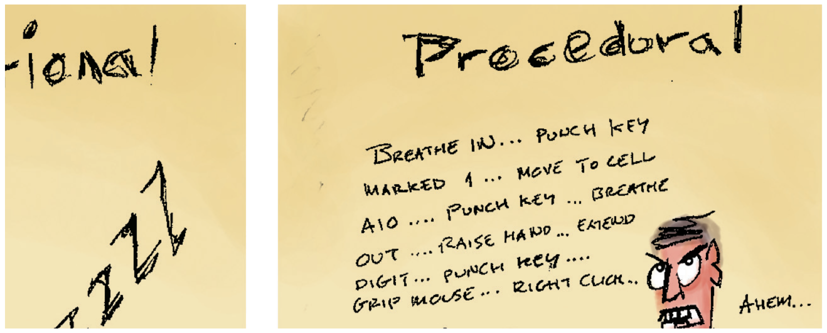
And expect to make the call **bang 12** to get the factorial of 12. This example may look artificially simple. But we can port even more complicated problems from mathematics directly to a functional language. For an example closer to home, let us consider a binomial pricing model, illustrating that the ease and elegance with which Haskell handles factorials do indeed extend to real-life quantitative finance problems as well.

A new kind of binomial tree

The binomial tree pricing model works by assuming that the price of an underlying asset can only move up or down by constant factors u and d during a small time interval δt. Stringing together many such time intervals, we make up the expiration time of the derivative instrument we are trying to price. The derivative is defined as a function of the price of the underlying asset at any point in time.

We can visualize the binomial tree as shown in Figure 1. At time $t = 0$, we have the asset price $S(0) = S_0$. At $t = \delta t$ (with the maturity $T = N\delta t$), we have two possible asset values, $S(t) = S_0 u$ and $S(t) = S_0 d = S_0/u$, where we have chosen $d = 1/u$. In general, at time $i\delta t$, at the asset price node level j, we have

$$S_{ij} = S_0 u^j \tag{3}$$



-ional
zzz
Procedural
BREATHE IN... PUNCH KEY
MARKED 1 ... MOVE TO CELL
A10 .... PUNCH KEY ... BREATHE
OUT .... RAISE HAND ... EXTEND
DIGIT... PUNCH KEY....
GRIP MOUSE... RIGHT CLICK..
AHEM...

Figure 1: The binomial tree. On the *x*-axis, labeled *i*, we have the time steps. The *y*-axis represents the price of the underlying, labeled *j*. The only difference from the standard binomial tree is that we have let *j* be both positive and negative, which is mathematically natural, and hence simplifies the notation in a functional language.

By choosing the same size of up-and-down price movements, we have created a recombinant binomial tree, which is why we have only $2i + 1$ price nodes at any time step idt. In order to price the derivative, we have to assign risk-neutral probabilities to the up-and-down price movements. The risk-neutral probability for an upward movement of u is denoted by p. With these notations, we can write down the fair value of an American call option (of expiry T, underlying asset price $S0$, strike price K, risk-free interest rate r, asset price volatility d, and number of time steps in the binomial tree N), using the binomial tree pricing model as follows:

$$\text{OptionPrice}(T, S_0, K, r, \sigma, N) = f_{00} \tag{4}$$

where f_{ij} denotes the fair value of the option at any the node i in time and j in price (referring to Figure 1).

$$f_{ij} = \begin{cases} \text{Max}(S_{ij} - K, 0) & \text{if } i = N \\ \text{Max}(S_{ij} - 0, e^{-\delta tr}(pf_{i+1j+1} + (1+p)f_{i+1j-1}) & \text{otherwise} \end{cases} \tag{5}$$

At maturity, $i = N$ and $i\delta t = T$, where we exercise the option if it is in the money, which is what the first Max function denotes. The last term in the expression above represents the risk-neutral backward propagation of the option price from the time layer at $(i+1)j i\delta t$ to $i\delta t$. At each node, if the option price is less than the intrinsic value, we exercise the option, which is the second Max function.

The common choice for the upward price movement depends on the volatility of the underlying asset. $u = \exp(\sigma\sqrt{\delta t})$ and the downward movement is chosen to be the same $d = 1/u$ to ensure that we have a recombinant tree. For risk neutrality, we have the probability defined as:

$$p = \frac{e^{r\delta t} - d}{u - d} \tag{6}$$

For the purpose of illustrating how our binomial tree translates to the functional programming language of Haskell, let us put all these equations together once more.

$$\text{OptionPrice}(T, S_0, K, r, \sigma, N) = f_{00}$$

where

$$\begin{aligned} f_{ij} &= \begin{cases} \text{Max}(S_{ij} - K, 0) & \text{if } i = N \\ \text{Max}(S_{ij} - 0, e^{-\delta tr}(pf_{i+1j+1} + \\ \quad (1+p)f_{i+1j-1}) & \text{otherwise} \end{cases} \\ S_{ij} &= S_0 u^j \\ u &= e^{\sigma\sqrt{\delta t}} \\ d &= 1/u \\ \delta t &= T/N \\ p &= \frac{e^{r\delta t} - d}{u - d} \end{aligned} \tag{7}$$

Now, let us look at the code in Haskell.

```
optionPrice t s0 k r sigma n = f 0 0
   where
     f i j =
         if i == n
         then max ((s i j) - k) 0
         else max ((s i j) - k)
                    (exp(-r*dt) * (p * f(i+1)(j+1) +
                    (1-p) * f(i+1)(j-1)))
     s i j = s0 * u**j
     u = exp(sigma * sqrt dt)
     d = 1 / u
     dt = t / n
     p = (exp(r*dt)-d) / (u-d)
```
(8)

As we can see, it is a near-verbatim rendition of the mathematical statements, nothing more. This code snippet actually runs as it is, and produces the result.

```
*Main> optionPrice 1 100 110 0.05 0.3 20
10.10369526959085
```
(9)

Looking at the remarkable similarity between the mathematical equations and the code in Haskell, we can understand why mathematicians love the idea of functional programming. This particular implementation of the binomial pricing model may not be the most computationally efficient, but it certainly is one of great elegance and brevity.

A functional programming language may not be appropriate for a full-fledged implementation of a financial program because of performance issues. However, many of its underlying principles, such as type abstractions and strict purity, may prove invaluable in programs we use in quantitative finance where heavy mathematics and number crunching are involved. The mathematical rigor enables us to employ complex functional manipulations at the program level. The religious adherence to the notion of statelessness in functional programming has another great benefit. It helps in parallel and grid computing, enabling the computations with almost no extra work.

Back to the basics

Rich in paradigms, the field of computing has a strong influence on the way we think and view the world. If you don't believe me, just look at the way we learn things these days. Do we learn anything now, or do we merely learn how to access information through browsing and searching? Even our arithmetic abilities have eroded, along with the advent of calculators and spreadsheets. I remember the legends of great minds like Enrico Fermi, who estimated the power output of the first nuclear blast by floating a few pieces of scrap paper, and like Richard Feynman, who beat an abacus expert by doing binomial expansion. I wonder if the Fermis and Feynmans of our age would be able to pull those stunts without pulling out their pocket calculators.

Procedural programming, through its unwarranted reuse of mathematical symbols and patterns, has shaped the way we interact with our computers. The paradigm that has evolved is distinctly unmathematical. Functional programming represents a counter attack, a campaign to win our minds back from the damaging influences of the mathematical monstrosities of procedural languages. The success of this battle may depend more on might and momentum than on truth and beauty. In our neck of the woods, this statement translates to a simple question: can we find enough developers who can do functional programming? Or is it cheaper and more efficient to stick to procedural and object-oriented methodologies?

ABOUT THE AUTHOR

The author is a scientist from the European Organization for Nuclear Research (CERN), who currently works as a senior quantitative professional at Standard Chartered in Singapore. The views expressed in this column are his own, which have not been influenced by considerations of his employer's business or client relationships. More information about the author and his forthcoming book (*Principles of Quantitative Development*, to be published by John Wiley & Sons Ltd.) can be found at his blog: http//www.Thulasidas.com.