Essence numbers and the shape of G15 PMN

By Aristo Tacoma, May 7, 2019

A good deal of years ago, with the understanding of essence numbers in place--in principle, but without much of the far more eloquent expressions and finely tuned concepts now at my disposal--I was faced with the question: what now as for computer programming languages? Where do these come in?

Firstly, they seem to have a winning card on their hand: they are clearly finite, and they don't mess around with infinity proofs.

Secondly, they provide a sense of 'clean start', for one who has been led into doubting most of the foundations of mathematics--because of the complicated entanglements with unclear ideas about infinity that characterises most of mathematics.

Thirdly, a programming language can be useful, --such as in controlling robots--and such usefulness adds to the interest in having such a language.

I first launched a programming language called Firth, borrowing something from Forth, and developed this in several stages over some years. What with one thing and another, I grew dissatisfied with how much reliance it had on a computer environment which contained vast amounts of programming--such as the file systems--and, worse, which seemed to have a construction that seemed to be aimed at hiding the limits, the finiteness. (Take, for instance, the long lengths of the file names,

the multiple folders--indefinitely many--
that a folder of files can contain and the
often rather indefinite max sizes of files;
and the system for ordering these file
names and folders and for cleaning up unused
spaces--all presuming a big programming
language even before the first thing is done
in the programming language one is making.)

I could begin with a language that works
directly on chips, but the same thing is at
work there: the chips contain commercially
patented, undocumented circuits, in vast
quantities, even doing such things as
'simulating' what goes on in the program
that is supposedly 'running on' these
chips. The integrated circuits of early
21st century are as far as one can get from
a simple open finite idea of a core for a
programming language.

Then what? Make it of transistors? Or make
even the transistors? Looking into it
practically, it would take the capital of
a billionaire to raise the money to make
a practical working computer of the kind
that can run all normal business applications
and also control robots out of nothing but
chunks of silicone and iron and so on.
Nevertheless, I sketched the steps, and
came up with the 'interplates' concept in
so doing, and that's still on the planning
board but it will be done some day.

At some point, I took a decision: I will
accept one level of simulation--one level
only--in which the simplest chip possible
that can do the job is simulated by means
of the present chips we have. There will
be no 'file system', no vast bundles of
premade code--just some hundreds of very
simple 'instructions' that can be handled
with a moderate amount of the types of
components out of which one can construct
a CPU, a central processing unit, whether
by chips or something larger like intra-
plates (and preferrably that).

Taking a step back, eventually, after a

great deal of work on this, I brought pen and paper to a beach and watched the waves and wrote a little program in a language that didn't yet exist--but which was to be made on top of that CPU of mine. The PMN on top of the G15.

And though I didn't put it in these words, this is about how the language was constructed:

We have looked at the set 1, 2 and 3, and we have noticed the importance of such whole numbers, and of arithmetic such as addition. Well then, let us put these numbers first in our language. Let us use letters for the simple things we want to do with them. One letter? Well, that gives us only 26 or so alternative things. Add a letter, we have 26x26 alternatives. Let the second letter also be a number, and perhaps even a sign like ?, and we have still more alternatives. That seemed to me to be much. Let us build bigger words out of these simple words, three letters and more.

Let the numbers, as the words, be put vertically, in stacks--just as we had found useful in our triangle argument which led to the essence numbers. The point here would be: we want to be able to see, clearly, what is going on at every point. So one number or two-letter thing or three-letter thing at each line. Okay, two columns then, so we make some more use of the computer screen, and also can see some relationship between the two vertical columns. Eight lines of two colums. That's what? A page? A little too short to be called 'page'. Let us call it 'card'. So that's it. Cards. No files, just cards. No long lines, just one thing on each line, as in classical assembly language, but here it is really simple stuff. How simple?

We want it to be practical, so not so simple it is just small numbers up to a

thousand or a million. The natural levels are given by 8 bits (hundreds), 16 bits (tens of thousands), 32 bits (billions). So, 32 bit it will be.

If we need data, we store it on cards; when we need programs, we store them, as said, on cards. So same kind of thing. But a card read as text has more spaces and such, a data card can be a more cryptic thing, granted, as long as the program easily can decode it and wrap new content into it.

Every number is between 0 and about two billion, the full 32-bit range is a bit higher than two billion, and it can go into the signed direction as well. Every number has a reserved bit for the sign. Keep it simple.

Every two-letter function has a number, this number is what the two letters are changed into. That number refers again to the level underneath it, the G15 level. That is a 'direct action' number, let's call it--as done in the previous language experiments--for 'warp'. In contrast to Forth, let this warp be a 32-bit number as a standard, transparent thing. In Forth, which dealt with stacks of numbers but without the estetics and design of vertical columns, there was no assertion about any standard of the size of what they there called 'pointers'--it was 16-bit, 32-bit or even more, and they sought to generalize it,--to make it deliberately unspecified. That is exactly what we want to avoid. We want the finite-ness of the language specified in as many ways as possible, so as not to get into any of the complications of assuming 'et cetera'.

Some might say: some applications may demand more than 32-bit, if they are going to do such as simulate a big portion of the world. Then we respond: we want the programs to be simple and transparent and open and human-readable and meaningful

and we want to keep all sorts of simulations to a minimum. 32-bit it shall be. No "artificial intelligence." Real intelligence at work in the programmers, and First-Hand work on Computerizing whatever make sense of their Mentality--FCM (see all my writings about FCM).

So whole numbers run the scheme. Each two-letter function (or one-letter) is converted to a whole number, a warp, that is what the underlaying level, G15, requires as active information to perform that operation--such as AD for addition. Then three-or-longer (not very much longer, just some thirteen or fourteen letters or so) words are made out of numbers and two-letter words and such simple constructs. These too get a number, a warp. So a three-letter function can, by having a place to store a warp, run any function not just a pre-programmed function--any function, that is, which exists within the active part of the maximum 32-bit memory or RAM area.

So, for example,

```
COUNTUP=
LL:3
I1
NN
LO.
```

(Written with lowercase on the computer.)
When typing in COUNTUP, it will show
1
2
3
Now let's look at this program. The = sign says: the word just before is a new one, in this context. Then comes the loop control, LL, with the number 3--that could be much higher, within the two billion range--and inside the LL is I1, get the number of the count, and NN, put it to the screen, then LO is the lower part of the loop, ie, the loop is done at this point. Then dot . to

tell: this is the entire definition.

So, in writing 1, 2, and 3, with the idea of adding 1, we are doing something like an algorithm. In then writing 1, 2, 3, ... we are saying: let's make this into a percept and just look at it and through it and see what we perceive. And, lo and behold, we perceive essence numbers. Or an essence number. (Look at article written just before this introducing the idea of 'percept'.)

Let's look at

```
COUNTUP=
LL:3
I1
2
MM
NN
LO.
```

Here we put in '2' and then 'mm' which means 'multiply'. So we get 2, 4, 6. (We could get this in another way eg by shifting a bit to the right, but clarity is important here.) As we argued in some previous articles, there is a different sense what we perceive through this percept. We have a different essence number.

So, yes, let us proceed to this point: we have whole numbers, very simple operations on them, and ways to group these operations on them, and some more such. And by this, shaped with as little as possible in terms of preconceived ideas about any surrounding technological environment, we get a possibility of running finite programs that works with finite things while at the same time having a calm perception of what is going on. We can look at the program, enjoy its elegance, the lack of presumption in its simple signs, the freedom from pompous statements about infinity in the context, and see something which is not simply the same as looking at individual whole numbers. We are looking at algorithms, we can say: but algorithms that are also leading us--when written as

elegantly as can be, in this maximum elegant language G15 PMN--to have percepts in our minds, percepts with which we can perceive. What we perceive refers to this world of abstract forms we talked about in connection to semantic mathematics. We noted that this is a 'nonstatic' set of forms. In some cases we spoke of them in musical terms.

The level underneath this elegant simple level must be an even more simple level in terms of technological requirements for how to put it together. It is not easy to put together something that can handle any set of numbers like that. It is easier to put logical gates or transistors or what we call them together so as to handle just a handful of numbers at a time. So I spent time shaping some short words, each corresponding to an instruction number from 1 and upwards to what eventually became two-three hundred--whose only function was to do something with a handful of numbers, such as arithmetic, or putting it to a screen or getting them from a keyboard or a mouse or from an external storage device like a disk or from the fast internal storage called RAM. Each number 32 bit, each signed, no simulation, no big deal of variation.

G15 having been made, instead of making PMN once and for all, PMN was made into a G15 program that compiles during start-up, so that it can easily be extended with a few more two-letter words dedicated to a particular task when speed or connection to a new physical device demanded so extra G15 work. In compiling it each time during start-up, it would be easy to be certain, in each case, of exactly what it is that is being performed, and how, rather than having concealed and closed layers that do something mysterious with a program that is in other senses open.

In deciding to put trigonometric functions of a whole-number kind into this, a set of

precalculated whole numbers was put into the RAM so that PMN functions could, with a definite limit to the resolution, shape circles and calculate angles back and forth and also square roots. This matched the sense of this approach being first-hand and not oriented towards simulation nor incorporating such routines as imply 'indefinite' resolution towards the 'infinitesimal error' with all the lack of clear ideas in the foundations of those concepts. True, it would put a limit to how much simulation can be done without going to great lengths in making new algorithms giving extra decimal digits, but after all that was part of the plan. We make that which is first-hand easy, and other things can be implemented when one has to for practical reasons, later.

To control robots, the FCM concept is then not about making a network of programs 'self-organizing' in any way: rather, it is about making several levels of a network pre-programmed so as to implement something of what the programmer has as his or her mentality relative to a certain type of task. This can also include some degree of what can be called 'entrainment'--without using the organic word 'learning' or any such pompous, nonsensical phrase like 'artificial intelligence' in any context. This network is defined by means of a matrix of a concrete, limited set of optional variables, to which some algorithms can be attached so that the network can be described as 'doing' something: but the robot is never 'on its own', but rather is in the control of a certain type of first-hand PMN program, even if it has some fuzziness in how something is to be done without definite boundaries, in definite contexts, and with definite criterions for what and when it is right for the robot to do anything

at all (which the programmer can think of as 'ethical priorities', but which are given an operational description in terms of pattern matching and scoring and such).

After groundwork for robot programming was made, but even more when the core group of simple 2D-without-simulation 'space invader shooting'-like games were made--to wrap up the bundle of applications which included text processing and image editing and such--the PMN had 'come of age'. Finally it was, as it now is, possible to lean back and ask the big philosophical questions and also the concrete mathematical questions. Each PMN function--we can now say, using our mature, sophisticated language--when written as elegantly as can be, doing something concrete, and lovely, is also possible to read as a percept. And through this percept we can have a perception into what it refers to--what essence it refers to (or essence number, but perhaps it is more clear to drop the word 'number' at this point).

Let us go again briefly to the idea of platonic abstract forms, even if we allow them to have movement, as we earlier said. Where do they come from? Are they just there?

This takes us into the worldview discussion. It is hard to say anything at all as answer to this without presuming much about worldview including metaphysics.

But if we admit to the notion, as this author does, that human consciousness is a light that can connect to some light beyond itself, and beyond even matter, then the perception is of something real, and when we have good percepts--and I believe the G15 PMN are as elegant and as good for this as can be--we will

perceive readily.

By analogy to 1, 2, 3 to which we add "et cetera"--the first part being as it were an 'algorithm', the second part being the perception of what it points to, in the subtle world of forms--every clear, elegant, simple program can be perceived to have something associated with itself, something perhaps beautiful, perhaps very beautiful. Let us not ask at the moment 'where' this subtle world is, or what 'dimension' it exists on, or 'who' underlies it, or 'what' or 'who' helps maintaining it, but let's rather try and get hold of an idea, even if entirely a poetic and metaphysical idea, of how such a subtle world arises in the first place.

I have asked this of myself and the answer that feels right to say is this: by means of a playful analogue sort of computer that looks at the digital algorithm (what I a very long time ago called 'the Uncomputer', but it is a concept I didn't feel like using because it felt too omnious). A playful analogue computer, without the types of limits that a digital computer has, --playful signifying that it is going beyond all known mechanical constraints altogether, in a lively and musical way--LOOKS at an algorithm and it is the very process of looking, by this playful analogue computer, that gives rise to the subtle form we can perceive by means of the percept of the elegantly written PMN program.

We are far out into the realm of that which to an atheist is pure nonsense, or worse, insane nonsense, and so we might as well go a step further in this article, just one step, and we have completed the

metaphysical excursions into the deeper meaning of G15 PMN algorithms.

I submit, as an intuition, that some of the most beautiful, simple programs, involving, for instance, the Fibonacci numbers, are percepts of a kind through which we perceive quintessential human beauty, but of a kind that goes beyond what a human can have. In a word, these percepts allow us to perceive something beyond the human realm altogether, something, someone--and my intuition is to bring in here the word 'muses' (and see my writings at other places as for why I feel that the modern word 'muse' in the English language indeed is an appropriate one for such a lofty use).

And this, my dear reader, explains how these articles can be written in between bouts of intense work also as a fashion and beauty photographer and fashion and beauty magazine editor: it is one and the same theme, speaking in terms of essences.

In lighter language, something of this is re-presented in the second volume of the Art of Thinking series, G15 PMN Programming for Teens, which will also treat the theme sex. I have already written the first volume, G15 PMN Programming for Kids. The third volume will discuss the core of the robotic programming approach in the sense of an overview. The fourth and fifth volume are both more philosophical and more scientific, but also, in a psychological sense, rather practical.