/usr/bin/3sOteric repOrt frOm the 4lternative universe Of c0mput3r prOgr4mming

I have been programming since I was 8. Long before picking up any book on magick or arcane occult arts, I started with BASIC, moving onto Turbo Pascal, then emerging into the UNIX hacker world of C, Perl and assembler language – using the internet before internet was available as a consumer product. In those days, everything online was meaningful, as only professors, geeks and hackers had access to the hidden online world. It became an escape from reality. To me, making computer programs has always been all about striving for perfection, a way to abstract the process of creation and manifestation within a limited universe, yet one that resembles the large one that surrounds us. In theory, if I understood one of them, the other would be understood too.

There is a vast difference between knowledge and experience. Knowledge lets you deduce the right thing to do, expertise makes the right thing a reflex, hardly requiring conscious thought at all. When one thinks about a computer operating system, one thought that first strikes most is the knowledge required for specialists to be working in this field. But the esoteric side of the Unix culture reveals a philosophy, tradition, and design that transcend technical boundaries. Unix was created by humans to solve problems. It thus emanated from humans, thus it is a whole little universe that has been engineered, that has its creation, myths, history, wars and arts and all the things that belong in a universe but scaled down, in a sense, to allow it to resemble a reflection of the higher.

How does this relate to the magickian? Well, if you consider an operating system the universe, then, the programming language becomes the logos, i.e. the words that create matter, substance, logic and meaning and dimensions. Here, both philosophy, symbols, logic and rules come into play. Even though a programming language has been perfected, it is only as good as who is using it.

Many books have been written on Unix, such titles as "The Timeless Art of Building" and "The Art of Computer Programming". Every branch of engineering and design has technical cultures. In most kinds of engineering, the unwritten traditions of the field are parts of a working practitioner's education as important as (and, as experience grows, often more important than) the official

handbooks and textbooks. Senior engineers develop huge bodies of implicit knowledge, which they pass to their juniors by (as Zen Buddhists put it) "a special transmission, outside the scriptures".

Genesis: Unix was born in 1969 (UNIX year zero is midnight, January 1st, 1970, and time is measured in the number of seconds passed since then) and has been in continuous production ever since. UNIX has evolved from big mainframes onto normal PC and laptop computers, with the advent of Linux and Mac OS X. It sort of represents an anti-culture from Microsoft Windows.

The Unix philosophy is not a formal design method. It is pragmatic and grounded in experience. It encourages a sense of proportion and skepticism – and shows both by having a sense of (often subversive) humour.

Looking at the whole, some rules emanated. Here is a few examples:

1: Rule of modularity: Write simple parts connected by clean interfaces.
2: Rule of Clarity: Clarity is better than cleverness.
For those of you who like Gematria, It is interesting that the rule of Har-Par-Kraat naturally falls on number eleven, the number of Magick:
11: Rule of Silence: When a program has nothing surprising to say, it should say nothing.
16: Rule of Diversity: Distrust all claims for the "One True Way".

17: Rule of Extensibility: "Design for the future, because it will be here sooner than you think".

If you are new to Unix, these principles are worth some meditation. If you are a mystic or magickian, these parables sound familiar. I find it interesting to see how once a totally new universe is manifested from void, a familiar pattern appears...

One interesting explanation of the "Rule of Silence" is explained by the fact that Unix predates video displays. On the old printing terminals of 1969, each line of unneccessary output was a serious drain on the user's time. That constraint is gone, but the practice has been kept. The terseness of Unix programs has lead to Unix's success, in many ways. When your program's output becomes another's input, it should be easy to pick up the needed bits. Important information should not be mixed in with verbosity about internal program behaviour.

To achieve enlightenment and surcease from suffering, Zen teaches detachment. The Unix tradition teaches the value of detachment from the particular, accidental conditions under which a design problem was posed. Abstract. Simplify. Generalize. Because we write software to solve problems, we cannot completely detach from the problems – but it is well worth the mental effort to see how many preconceptions you can throw away, and whether the design becomes more compact

and orthogonal as you do that. Possibilities for code reuse often result, hence the popular "Open Source" culture, why rewrite code and reinvent the wheel when you can recycle 80-90% of what has been made before?

Jokes about the relationship between Unix and Zen are a live part of the Unix tradition as well. This is not an accident. One example is the poem:

"Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it." –Brian Kernighan, co-creator of Unix.

Here is an example:

```
while ($programming and not $thinking)
{
      $enlightenment++;
}
```

This program will actually run on a computer. Translated into human language, the program reads thus: "While the condition of the value of programming is true and value of thinking is not true, the value of enlightenment increases by the factor of one for each time continuously in an infinite loop into eternity".

Another example where programming languages can break out of the box of normal time-space continuum is the legendary "camel" code, made in Perl:

```
#!/usr/bin/perl -w # camel code use strict;

                                        $_='ev
                                      al("seek\040D
          ATA,0,                     0;");foreach(1..3)
       {<DATA>;}my                  @camel1hump;my$camel;
  my$Camel  ;while(                <DATA>){$_=sprintf("%-6
9s",$_);my@dromedary              1=split(//);if(defined($
_=<DATA>)){@camel1hum            p=split(//);}while(@dromeda
 ry1){my$camel1hump=0           ;my$CAMEL=3;if(defined($_=shif
        t(@dromedary1    ))&&/\S/){$camel1hump+=1<<$CAMEL;}
       $CAMEL--;if(d   efined($_=shift(@dromedary1))&&/\S/){
      $camel1hump+=1  <<$CAMEL;}$CAMEL--;if(defined($_=shift(
     @camel1hump))&&/\S/){$camel1hump+=1<<$CAMEL;}$CAMEL--;if(
     defined($_=shift(@camel1hump))&&/\S/){$camel1hump+=1<<$CAME
     L;;}$camel.=(split(//,"\040..m`{/J\047\134}L^7FX"))[$camel1h
      ump];}$camel.="\n";}@camel1hump=split(/\n/,$camel);foreach(@
      camel1hump){chomp;$Camel=$_;y/LJF7\173\175`\047/\061\062\063\
      064\065\066\067\070/;y/12345678/JL7F\175\173\047`/;$_=reverse;
       print"$_\040$Camel\n";}foreach(@camel1hump){chomp;$Camel=$_;y
        /LJF7\173\175`\047/12345678/;y/12345678/JL7F\175\173\0 47`/;
         $_=reverse;print"\040$_$Camel\n";}';;s/\s*//g;;eval;   eval
           ("seek\040DATA,0,0;");undef$/;$_=<DATA>;s/\s*//g;(   );;s
             ;^.*_;;;map{eval"print\"$_\"";}/.{4}/g; __DATA__   \124
               \1   50\145\040\165\163\145\040\157\1 46\040\1  41\0
                    40\143\141  \155\145\1 54\040\1   51\155\  141
                    \147\145\0  40\151\156 \040\141    \163\16 3\
                     157\143\   151\141\16  4\151\1     57\156
                     \040\167  \151\164\1   50\040\      120\1
                     45\162\   154\040\15    1\163\      040\14
                     1\040\1   64\162\1      41\144       \145\
                     155\14    1\162\       153\04        0\157
                      \146\     040\11     7\047\         122\1
                      45\15      1\154\1  54\171           \040
                      \046\         012\101\16            3\16
                      3\15           7\143\15             1\14
                      1\16            4\145\163           \054
                     \040            \111\156\14         3\056
                    \040\          125\163\145\14         4\040\
                    167\1          51\164\1  50\0         40\160\
                  145\162                                \155\151
                \163\163                  - 14 -         \151\1
               57\156\056
```

Of course this code actually runs, too. What does it do? You guessed it. It prints out an identical camel.

It reminds me about the last line of one section of the loginataka, a discourse between the student and the hacker on how to become a hacker, much resembling a dialogue with a novise monk and the zen master where the student is advised to "travel in the way of the camel", meaning of course the book about the "Perl" programming language, which has the image of a camel on it. (All the O'Reilly books have pictures of animals on it, they can be considered the bibles of Unix culture).

Speak, O Guru: What books should I study? Are the O'Reilly "Nutshell" guides a good place to start?

O, Nobly Born: know that the Nutshell Guides are but the outermost Portal of the True Enlightenment. Worthy are they (and praise to the Name of O'Reilly, whose books show forth the Hacker Spirit in numerous pleasing ways), but the Nutshell Guides are only the Beginning of the Road.

If thou desirest with True Desire to tread the Path of Wizardly Wisdom, first learn the elementary Postures of Kernighan & Pike's The Unix Programming Environment; then, absorb the mantic puissance of March Rochkind's Advanced Unix Programming and W. Richard Stevens's Advanced Programming In The Unix Environment.
Immerse thyself, then, in the Pure Light of Maurice J. Bach's The Design Of The Unix Operating System. Neglect not the Berkelian Way; study also The Design and Implementation Of The 4.4BSD Operating System by Kirk McKusick, Keith Bostic et. al.
For useful hints, tips, and tricks, see Unix Power Tools, Tim O'Reilly, ed. Consider also the dark Wisdom to be gained from contemplation of the dread Portable C And Unix Systems Programming, e'en though it hath flowed from the keyboard of the mad and doomed Malvernite whom the world of unknowing Man misnames "J. E. Lapin".
These tomes shall instruct thy Left Brain in the Nature of the Unix System; to Feed the other half of thy Head, O Nobly Born, embrace also the Lore of its Nurture. Don Libes's and Sandy Ressler's Life With Unix will set thy Feet unerringly upon that Path; take as thy Travelling Companion the erratic but illuminating compendium called The New Hacker's Dictionary (Eric S. Raymond, ed., with Guy L. Steele Jr.).

In this wise shalt thou travel the Way of the Camel."

--- From Eric S. Raymonds "Loginataka". http://catb.org/~esr/faqs/loginataka.html

Unix uses Gematria too, in files, where metadata in the form of numbers are included. One way to incorporate such metadata, often associated with Unix and its derivatives, is just to store a "magic number" inside the file itself. Originally, this term was used for a specific set of 2-byte

identifiers at the beginning of a file, but since any undecoded binary sequence can be regarded as a number, any feature of a file format which uniquely distinguishes it can be used for identification. GIF images, for instance, always begin with the ASCII representation of either GIF87a or GIF89a, depending upon the standard to which they adhere. Many file types, most especially plain-text files, are harder to spot by this method. HTML files, for example, might begin with the string <html> (which is not case sensitive), or an appropriate document type definition that starts with <!DOCTYPE, or, for XHTML, the XML identifier, which begins with <?xml. The files can also begin with HTML comments, random text, or several empty lines, but still be usable HTML.

The magic number approach offers better guarantees that the format will be identified correctly, and can often determine more precise information about the file. Since reasonably reliable "magic number" tests can be fairly complex, and each file must effectively be tested against every possibility in the magic database, this approach is relatively inefficient, especially for displaying large lists of files (in contrast, filename and metadata-based methods need check only one piece of data, and match it against a sorted index). Also, data must be read from the file itself, increasing latency as opposed to metadata stored in the directory. Where filetypes don't lend themselves to recognition in this way, the system must fall back to metadata. It is, however, the best way for a program to check if a file it has been told to process is of the correct format: while the file's name or metadata may be altered independently of its content, failing a well-designed magic number test is a pretty sure sign that the file is either corrupt or of the wrong type. On the other hand a valid magic number does not guarantee that the file is not corrupt or of a wrong type.

So-called "shebang" lines in script files are a special case of magic numbers. Here, the magic number is human-readable text that identifies a specific command interpreter and options to be passed to the command interpreter. It is written thus:

```
#!
```

And often used to tell the system that a certain programming language should be used to interpret the file, for instance the bash script language or perl:

```
#!/usr/bin/perl

#!/bin/bash
```

So much for magick/occult symbols in Unix.

Now, it would be intersting to comment on Gnostic ideas and the universe of Unix.

As a program is made through

the programming code (logos) the intention of the idea (will) emerges.

Even the best software tools tend to be limited by the imaginations of their designers. Nobody is smart enough to optimize for everything, nor to anticipate all the uses to which their software might be put. Designing rigid, closed software that won't talk to the rest of the world is an unhealthy form of arrogance.

Therefore, the Unix tradition includes a healthy mistrust of "one true way" approaches to software design or implementation. It embraces multiple languages, open extensible systems, and customization hooks everywhere.

One can say that a potent manifestation of Gnostic thought and the polarity of systems can be best exampled by Microsoft Windows, the most prevalent operating system among personal computers still, which is a proprietary, licenced, monopolised closed-off system where the code is hidden and you have to be a hacker to "break out" of the shell that imprison the user, contrast this to Linux, an incarnation of Unix that is free, open, and where the user can be instant in control and see the source code of any piece of the system. It puts Adam back in the garden of Eden, where time began in year zero (1st of January 1970), where tapes of free and open software were circulated among hackers in Berkeley. To this day, the utility for compressing folders in Unix is simply named "tar", although many has forgotten it once was used mainly on Tape Archives..

by Frater E.A.S