Perl and OpenVMS – a Powerful Match

Although OpenVMS has a powerful command language with its DCL interpreter, there are everyday tasks which can be solved way more easily using another interpreted language like Perl which is readily available for OpenVMS on all three architectures. The following article is the result of my experiences with Perl on this platform as well as some talks covering this topic I delivered at DECUS and HP in 2008 and 2009. All examples in the following were chosen from my own daily work on a large OpenVMS system and range from simple code snippets, programs for one time usage to larger Perl programs running as batch jobs and performing crucial tasks like fetching mail from a POP3-server and the like.

To make one point clear at the beginning: Perl is not a replacement for DCL but it makes life much more easier when it comes to parsing and modifying files, socket communications, data base access etc.

Some Basics about Perl

Let us start with summing up some facts about Perl – although Perl is way too powerful to be described exhaustingly in a few pages like this, at least an impression of some of its basic features may be given for those who have never seen or written any actual Perl code.

First of all there is one main source of information and enlightenment for the Perl programmer, the so called *Camel Book*, "Programming Perl", published by O'Reilly. This book is a joy to read and contains a wealth of Perl know how with lots of practical examples and should not be missing on every desk of a Perl programmer, even an experienced one.

But now to the promised basic facts about Perl: First of all, Perl was developed initially by Larry Wall and its name is not an acronym but a retronym with a variety of interpretations (some people claim that it stands for "Pathological Eclectic Rubbish Lister" but "Practical Extraction and Report Language" fits better). Perl is a modern interpreter language – in fact, the interpreter does a terrific job so Perl programs tend to run surprisingly fast, even on slow machines like a VAX (compare that with Java which does not run on a VAX at all). Furthermore Perl runs on a wide variety of architectures and operating systems (maybe even more than Java is available for) making it a language for really portable applications. Perl is extremely powerful and concise and there is a plethora of modules readily available for nearly all purposes. Perl's basic philosophy is "*There is more than one way to do it*" (compared with other languages like Python which try to force the programmer to only one way to solve a problem which is good for the bondage and discipline fraction but not for those of us who like to write elegant code).

Most programmers who are not Perl-aware tend to call Perl an unreadable language, which is only true at first sight at best – at a first glimpse Perl code can really look like line noise, but for the initiated Perl programmer code like this is very readable (compare it with APL, eg.).

First of all, Perl does not care about the type of its variables – in fact it is basically a type free language. Instead, Perl cares about the structure of variables – in essence there are three such basic structures:

1. Scalars: A scalar variable can hold a single value at a time. The name of such a variable is always prefixed by a dollar sign like in "`$pi = 3.14159265;`".
2. Arrays: An array is an indexed list consisting of scalars as its elements. The name of an array variable is prefixed by a `@` like in "`my @entries;`".
3. Hashes: These are similar to arrays since their elements are scalars, too, but instead of numerical indices strings are used to address elements within a hash. The name of a hash variable is preceded by a `%` as in "`my %data;`" for example.

Some examples for these basic structures are shown in the following:

```
my $pi = 'three_point_one_four';
my @array;
$array[0] = 'Something';
$array[1] = 'Something else';
$array[2] = 2.718;
my %data;
$data{'name'} = 'Bernd';
$data{'occupation'} = 'VMS enthusiast';
```

Perl supports all of the common control structures like "`if...else`", "`while`", "`do...while`", "`for`" etc. Every such statement controls a block surrounded by braces which can not be left out like in C if only a single statement is to be controlled, so a typical `if`-statement has this form:

```
if (some_condition)
{
  do_something;
  ...
}
```

If only a single statement is to be controlled, a second form of control can be used, the so called *statement modifier*:

```
do_something if condition;
```

This is quite similar to spoken English and helps to keep program sources short. In the following some simple examples of loops are shown:

```
for (my $i = 0; $i < 10; $i++) # This is not very common in Perl
{
  print "$i\n";
}
for my $i (0..9) # This looks more like Perl
{
  print "$i\n";
}
print "$_\n" for (0..9); # A for loop as a statement modifier
```

As powerful as these control structures provided by Perl are, much of its power results from its embedded regular expression parser. Regular expressions really tend to look a bit like line noise so no reasonable example will be given now as some may be found in the examples section.

As already mentioned there is a wealth of modules which can be used with Perl readily and which are available at a central location called *CPAN*, short for "*C*omprehensive *P*erl *A*rchive *N*etwork", which can be accessed at `http://www.cpan.org` and `http://search.cpan.org`. Regardless what your initial problem is, you should always have a look at CPAN first – normally there already is a module which solves at least part of your problem if not the whole problem at all. Examples for the power of Perl's modules are countless as you will see in the examples section.

It is time to give some advices about what to do in general and what not:

Do:

- Be open for the Perl way of solving problems. Perl programs tend to be very short and

powerful, especially when compared with equivalent solutions in other programming languages.

- Use functions like `split`, `join`, `map`, `grep` and the like instead of unnecessary C-style loops.
- Use hashes when you need to lookup values.
- Always use regular expressions to parse, manipulate or split (complicated) strings.
- Be strict and use warnings all the time.
- Get the *Camel Book*.

Do not – do not even think about it:

- ...program in Perl like you program in C or Java or DCL or anything else.
- ...use arrays when you can use hashes – especially never ever loop over an array to find an element.
- ...write linear narrative code.

How does one get a running Perl installation on an OpenVMS system (or any other system as well)? Basically there are two ways to get a Perl interpreter up and running on a given system:

1. You can use a precompiled package – for OpenVMS there is a HP-supplied distribution kit, for other platforms there are equivalent such kits available.
2. Get the sources and compile, link and install the system yourself.

Personally, I always prefer the second method since I like to know what is really running on my system and sometimes I want to do things differently compared with a precompiled installation kit. (Compiling a Perl system on an Alpha or Itanium system is fast, but on a VAX this can take several hours, so be prepared!)

As all of you know (and love), OpenVMS is different from (and superior to :-)) other operating systems which has to be taken into account when porting or writing software. There are quite some modules to be found on CPAN which encapsulate OpenVMS specific tasks like interfacing the mail system etc. In addition to that there are modules to handle operating system specific tasks as transparently as possible which should be used whenever possible to yield operating system agnostic code.

In the following a short and definitely incomplete list of modules is shown which are especially useful in an OpenVMS environment:

- `VMS::Device` – interface to `$GETDVI` and the like.
- `VMS::Filespec` – converts between OpenVMS and UNIX filespecs.
- `VMS::FlatFile` – use hashes to work with index files.
- `VMS::ICC` – intra cluster communication services.
- `VMS::Mail` – interface to the OpenVMS mail system.
- `VMS::Process` – manage OpenVMS processes.
- `VMS::Queue` – work with queues and their entries.
- `VMS::Stdio` – file operations like binmode, flush, vmsopen etc.
- `VMS::System` – retrieve system information.
- `File::Basename` – system independent operations on filenames and the like.

Having all this said, it is time to show some real live examples of Perl in an OpenVMS environment.

Examples

Some of the following examples, especially the simpler ones, are accompanied with their source code which may be of interest, although more complex examples are only described textual with some code snippets. (If you are interested in the source code of one of these more complex examples, please feel free to contact the author directly by mail.)

Programs for one-time-usage

Many everyday tasks require that system administrators as well as programmers solve unexpected problems like clever pattern matching, parsing logfiles etc. which are not readily handled with standard DCL tools. Many of these problems can be solved on the fly using a couple of lines of Perl code.

Perl can be used as a command line tool like `awk` for example. This can be very useful when you have a puzzling problem which does not deserve a real program but nevertheless needs a clever data conversion on the fly or something like that. Since there are a variety of command line options for Perl which are useful in this context, only simple examples are given in the following (more information may be found elsewhere like the Camel Book).

Adapting configuration files to VMS

Once I inherited a configuration file `transfer.ini` which looked like this (but contained literally hundreds of such sections):

```
[logging]
  log    = log/transfer.log
  ticket = log/ticket.log
[templates]
  ticket = templates/ticket.tpl
  mail   = templates/mail.tpl
```

Of course, these pathnames are not very OpenVMS like and it would have been quite cumbersome to edit all of the manually. Now one could write a Perl program reading the file, performing the necessary changes using regular expressions and then writing the result back to disk. Since tasks like these are commonplace, Perl can be used as a mighty command line tool for performing in-place edit operations like transforming the pathnames in the example above into valid OpenVMS file names. In the example shown, this was accomplished with the following command line:

```
$ perl -i -pe “s/^(.*\s*)=(\s*)(.+)\/(.+)/$1=$2\[\.$3\]$4/” transfer.ini
```

It looks a bit like line noise, right? What does it do? First of all it loops over all lines in `transfer.ini` and matches lines which contain an equal sign with a string to its left and two strings separated by a slash to its right. These three strings are captured using parentheses and then the line which matched this expression will be substituted by a new line constructed out of the parts just captured. Applying this single line statement to the configuration file shown above yields a new version of this file with the following structure

```
[logging]
  log    = [.log]transfer.log
  ticket = [.log]ticket.log
[templated]
```

```
ticket = [.templates]ticket.tpl
mail   = [.templates]mail.tpl
```

which is exactly what was desired.

Repairing HTML files

Another problem I was faced with was that a user of a WASD web server insisted on creating her web pages using "modern" tools running on a MAC. Unfortunately these particular tools just refused to generate proper HTML encoding for special German characters like "ä" which should be coded as "`&auml;`" in HTML. Instead these tools just insert "ä" which results in a completely bogus display of the resulting web page. This problem can be corrected on the fly with a Perl call like this:

```
$ perl -i -pi "s/\xC3\xA4/\&auml\;/g; s/\xC3\xB6/\&ouml\;/g;
s/\xC3\xBC/\&uuml\;/g; s/\xC3\x84/\&Auml\;/g; s/\xC3\x96/\&Ouml\;/
g; s/\xC3\x9C/\&Uuml\;/g; s\xC3\x9F/\&szlig\;/g;" [...]*.html
```

OK – it really looks like line noise, but it can be easily put into a DCL routine which can be called after each upload of HTML files by this particular user and thus correcting the problem on the fly. This very special user (my beloved wife, to be exact) also wanted to include a background picture into her web pages which the tool used just does not support. Using Perl it was simple to extend the generated HTML code by a proper `background-image` directive, too.

Making sure that a large LaTeX document is consistent

Another problem came up when I wrote a really large book using LaTeX without using BibTeX (which is stupid, but the project grew from a pet project to a major project and when I realized that the simple bibliography of basic LaTeX was not really powerful enough to cope with the bibliography, it was literally too late to switch to BibTeX). Having a very long list of literature, I feared that some entries could have been rendered unused in the text body due to changes in its structure etc. Although LaTeX tells you when you cite something which is not defined, it does not tell you if you have bibliography entries which are not cited which is annoying.

A typical entry has the form

```
\bibitem{zachary} %book
G. Pascal Zachary, \emph{Endless Frontier – Vannevar Bush,
Engineer of the American Century},
The MIT Press, 1999
```

while a citation looks like

```
cf. \cite{zachary}[p.~142]
```

Having a document with more than 120000 lines of LaTeX code resulting in about 600 pages of text with more than 600 bibliography entries, a solution was necessary to make sure that no entry went uncited. This was accomplished with the following Perl program which reads in the complete LaTeX source code with a single statement and parses this for all citations in a first run while building a hash containing these citations. In a second run through this data all bibliography entries are processed and a message is printed for every bibliography entry without a corresponding citation:

```
use strict;
use warnings;

die "Usage bib.pl <filename.tex>\n" unless @ARGV + 0;
```

```
my $data;
open my $fh, '<', $ARGV[0] or die "Could not open $ARGV[0]: $!\n";
{
  local $/;
  $data = <$fh>;
}
close $fh;

my %cite;
$cite{$_}++ for $data =~ m/\\cite\{(.+?)\}/g;

$cite{$_} or print "$_\n" for $data =~ m/\\bibitem\{(.+?)\}/g;
```

Parsing a log file and generating some statistics

Some months ago I had to parse a log file containing entries like these:

```
[LOG|SYSTEM|2008 May 13, 14:15:26 (886)|ENGINE.batch]
Loaded 16 events in 497 milliSecs
[END]
[LOG|SYSTEM|2008 May 13, 14:15:55 (281)|Risk|BatchJob]
Time to execute Scenario 24902 ms
[END]
[LOG|SYSTEM|2008 May 13, 14:15.55 (283)|Risk|BatchJobThread]
Time to execute Scenario 13662 ms
[END]
```

I was asked to calculate the arithmetic mean and possibly other values of the time necessary to execute scenarios and thus wrote the following short Perl program:

```
use strict;
use warnings;
die “Usage: stat3 \”yyyy mm dd\” \”hh:mm:ss\”\n” if @ARGV != 3;

my ($file, $date, $min_time) = @ARGV;
my @values;
open my $fh, '<', $file or die “Could not open $file: $!\n”;
{
  local $/ = '[END]';
  while (my $entry = <$fh>)
  {
    my ($time, $duration) = $entry =~
      m/^.+\|.+\|$date,\s(\d\d:\d\d:\d\d\s).*execute
Scenario\s(\d+)\sms/s;
    push (@values, $duration) if $time and $time ge $min_time;
  }
}
close $fh;
```

```
print 'Average: ', in(eval(join('+'. @values)) / @values) / 1e3,
  ' s (', @values + 0, ")\n" if @values + 0;
```

Of course, the simple arithmetic mean could have been calculated in the main loop without the need of storing all values captured from the log file into an array but since it was not clear if more complex calculations might be necessary, it was decided to save all values first and use them for the calculations in the next step. This led to the funny way of computing the arithmetic mean by concatenating all array elements in a single long character string, joined together with '+'-characters. This string is then fed into an eval which is not efficient but shows what can be done using a dynamic programming language like Perl.

Are there any files with W:WD-rights on my system disk?

Another day I was asked "How can you be sure there are no files on your system disk which are writable by WORLD?" Good question – this calls for a short Perl program, too, which shows how external commands and functions can be called using backticks while capturing their output into program variables:

```
use strict;
use warnings;

my ($fc, $mc) = (0, 0);
for my $line (`dir/prot/width=(file=60) [...[`)
{
  my ($file, $w) = $line =~ m/(.+)\s+.+,(.*)\)/;
  next unless $file;
  $fc++;
  print "$file\n" and $mc++ if ($w =~ m/[WD]/);
}

print "$fc files processed, $mc are world writable/deletable!\n";
```

It turned out that no files were endangered by wrong protection settings and yes, the program was tested by creating a file with W:WD rights deliberately.

Migrating a MySQL database to Oracle/RDB

Another one time script which proved itself being very useful was written to migrate a MySQL database running on a LINUX machine to an Oracle/RDB system running on an OpenVMS system (cf. "Bringing Vegan Recipes to the Web with OpenVMS", OpenVMS Technical Journal, No. 8, June 2006). All out of the box attempts to solve this problem did not work directly due to the very different output/input formats regarding the generated files. A first attempt to transform a MySQL output file into a suitable load file for Oracle/RDB proved to be quite complicated and having much overhead, so it was decided to give up this approach and try an online approach using a simple Perl program to connect to both databases at once, reading data from MySQL and writing it directly into Oracle/RDB.

The resulting program turned out to be quite generic and only expects the necessary database connection parameters as well as a list of tables to be copied. The copy operation itself was faster than expected and even outperformed the first attempt using file based export/import with an external transformation routine implemented in Perl.

Larger Perl programs

Many problems which occur on a regular basis can be solved using Perl, too. Examples for such problems are:

- Generating simple web server statistics on a daily basis.
- Fetching mail from a POP3 server in regular time intervals and distributing these mails to the OpenVMS mail system.
- Sending outgoing mails to an SMTP server requiring authentication which is not currently supported by OpenVMS's TCPIP stack.
- Caching results from database queries to speed up execution time of programs requesting data from a database etc.

These four examples will be described briefly in the following showing the power of Perl in larger applications:

Simple web server statistics

After observing that the WASD web server running on an OpenVMS system was unexpectedly busy, a simple web server statistics was to be programmed to see which files were requested how often. All in all a result like this should be generated:

```
2734: my_machines/dornier/do80/chapter_1.pdf
 288: my_machines/bbc/tisch_analogrechner/anleitung.pdf
 117: publications/anhyb.pdf
  97: publications/handson.pdf
```

This was accomplished after only a couple of minutes with the following short Perl program:

```
use strict;
use warnings;

die "File name and account name expected!\n" unless @ARGV == 2;
my ($log_file, $account) = @ARGV;

open my $fh, '<', $log_file or die "Unable to open log file
$log_file, $!\n";

my %matches;
while (my $line = <$fh>)
{
  my ($ip, $key) = $line =~ m/^(\d+\.\d+\.\d+\.\d+).*"GET \/
$account\/(.+?)\s/;
  next if !$ip or $ip =~ '^192.168.31';
  $key =~ s/"//g;
  $key .= 'index.html' if $key =~ m:/$:;
  $matches{$key}++ if $key =~ m/(html|pdf|txt)$/;
}

close $fh;

printf "%5d: %s\n", $matches{$_}, $_
```

```
for (sort {$matches{$b} <=> $matches{$a}} keys(%matches));
```

A couple of months later my friend Michael Monscheuer wrote an equivalent web server statistics script in pure DCL which was much (very much, in fact) longer than the program shown above, although I have to admit that his solution seemed more easy to read at first sight, but due to the sheer amount of code this impression faded rather quickly.

Fetching mail from a POP3-server

Sometimes it would be desirable to fetch mails from a standard POP3-server and make these mails available in the OpenVMS mail system so the system's users can access their mails using MAIL or a suitable webinterface like `yahmail` or `soymail`. To make this possible, a Perl written batch job is required which polls in regular intervals a variety of POP3-servers and their associated mailboxes, fetches mails and distributes these mails to the various users of the OpenVMS system.

The overall Perl code for implementing this batch job consists of only 140 lines since most of the really complicated subtasks were already implemented in the following modules readily found on CPAN:

- `Net::POP3` – client interface to the POP3-protocol.
- `IO::File` – file creation and access methods.
- `POSIX qw(tmpnam)` – used to create temporary file names.
- `VMS::Mail` – interface to the OpenVMS mail system.

When it is possible to receive mails, it would be nice to be able to send mails, too, as the following example shows:

SMTP-Proxy

Almost every current mail provider requires that its clients authenticate prior to sending mail via their SMTP server(s). Unfortunately, authentication is not supported by the TCPIP package for OpenVMS. Since a requirement was to send output mail directly from the OpenVMS system, i.e. without an intermediate proxy system like a LINUX host or the like, it was decided to implement a small SMTP-proxy in Perl running on the OpenVMS system itself.

This proxy connects on the local machine to port 25 and listens for outgoing mail while another connection is maintained to port 25 of the provider. Every outgoing mail is parsed and enriched with the necessary authentication information before being sent to the provider which solved the initial problem quite easily.

This SMTP-proxy makes use of the following modules yielding an overall code size of only 68 lines of Perl code:

- `Net::ProxyMod` – this module allows easy TCPIP-packet modification.
- `MIME::Base64` – MIME-encoding and -decoding.
- `Tie::RefHash` – allows using references as hash keys.

Database-Proxy

Sometimes it is desirable to perform database accesses not directly but via a proxy which might either contain some business logic and/or caching mechanisms to reduce database load and to speed up the application at the cost of some additional memory consumption. Since a former article already described this Perl based proxy in detail (cf. “Bringing Vegan Recipes to the Web with OpenVMS”, OpenVMS Technical Journal No. 8, June 2006) only the obtained speedup of a factor

of 10 obtained with this proxy should be noted here.

Conclusion

Over the years it turned out that Perl is an invaluable tool for solving everyday problems as well as for writing large and complex programs running interactively as well as in batch mode. Especially in an OpenVMS environment which often poses very special needs when it comes to system connectivity, interfacing and the like, Perl can be employed with much benefit.

Perl does not consume too many resources and is really fast for an interpreted language so it even runs very well on smaller VAX systems (where not even a JVM is available).

It is important to realize that Perl is not just a “scripting language” as it is sometimes called. Instead it is mighty programming language and programming environment, thus Perl should be taken seriously. So, have fun with Perl and OpenVMS – a perfect team for all of us.

The author can be reached at ulmann@vaxman.de