PhUSE 2006

Paper TU 04

GKNIT your own - Creative ways to use Unix text tools for everyday problems

David J. Garbutt, BSI AG, Baden-Dättwil, Switzerland

ABSTRACT

Unix's text tools are perhaps partly responsible for its reputation as a system only for experts. I aim to show you how a little knowledge can go a long way to solving your everyday testing and validation problems. These are not tools for developing big user-friendly systems; they are for getting answers for specific questions that will most likely never recur. The Unix one-liner is the epitome of a fast and light programming system and it is a perfect fit as a second skill for SAS programmers. This tutorial will work up from simple beginnings to show you how to Gknit your own, but also give you some canned examples that will be useful today. Incidentally, Windows users should still come along because these tools are now available for Windows XP free from Microsoft.

INTRODUCTION

The hardest thing learning a new system is that lost feeling when you are faced with a problem you have no idea how to tackle. The goal of this talk is to provide a map to use when that lost feeling hits you. I am trying to write the paper I wish I had had when I started with Unix.

This paper gives you a fast overview of

1. *What* commands are available, and
2. Which *problems* they solve, and to give
3. Relevant *examples* of
4. How you can put them together to *solve* problems. And lastly, to introduce you to
5. The power of regular expressions (RE).

I have created tables of the text tools classified by function, which, I hope, will serve as a reference point for your future problem solving. Unix text tools are well documented, but there is not much material around that looks at tasks and how you can use and combine text tools to solve them.

None of the awk or Unix programming here is new; but I hope the actual examples, which all come from the SAS-in-Pharma programming area, will convince you there is much here that can be of daily use, and therefore inspire you to learn how to use these tools.

Many Unix texts use examples based on Unix Administration tasks, which perhaps gives the impression that that is their domain. Their domain is all text files: and SAS programs, logs, listings and RTF files *are* all text files and are therefore available for processing and analysis.

DATAFLOW PLUMBING WITH PIPES

Unix commands can all[1] be fitted together in long sequences separated by the vertical bar character. These sequences are known as pipes. All they really do is connect the output channels of one program to the input channel of the next in the sequence. But with this possibility of combining simple tools it is becomes easy to break complex tasks into sub-problems you know how to solve[2].

There may be one step you cannot solve, and this alone is useful because it helps you focus on where the problem really is. When you get stuck you can revisit the tables here to get clues about how to tackle the problem.

SAS as a programming language can also function like this: if you use simple Data and Proc steps that only output one dataset, then the passing of data from one step to another happens behind the scenes. And you end with a proc print (or ODS these days), or by creating a permanent dataset.

---

[1] All the programs we mention here. Strictly, the ones that work in pipes are called filters. Check the man page.

[2] This strategy, or heuristic, is one of those recommended by Polya in 'How to solve it', one of the first books ever published about general problem solving methods and strategies.

EXAMPLES

I have written the following examples during my everyday work as a SAS programmer. They have already helped me. I hope they will help you too.

I recommend experimentation, find an example that tackles a problem you have (or near to it at least) and copy it to your system. With command pipelines it is very easy to take them apart and see what each step does. Then modify them and see the effect.

It is best to cut and paste from the web version of this document. Word likes to make smart quotes and you may also get extra line-ends. I have added a > character at the start of lines that are commands, this is to better separate the commands from their output. Do not paste this character!

IN SHORT...

Quick solutions for problems where writing a SAS program is not worthwhile.[3]

Further examples not included in the paper are available from my website. The web site is hyperlinked and is recommended for browsing and for copy & paste to a terminal session.

UNIX TEXT COMMANDS

more about awk, nawk about more

The Unix text processing commands are a group of programs (provided with the system) concentrated on reading, writing and selecting text.

- They provide portable functionality because they are found on pretty well all Unix systems (including Interix and Cygwin).
- Although they fit together well, they are not totally integrated, but they are fast and easy to use after you have some familiarity.

WHAT OPERATIONS DO THEY DO?

They do what you can do with SAS: selecting variables and or observations, transforming them, and printing or saving them.

The names are completely different and to some extent the way the tasks are divided up are different. Nevertheless it is perhaps surprising that there are tools that can do all these things to text files:

- transform (edit) the values,
- calculate totals and other simple statistics,
- reformat the output,
- and even do sorting and merging.

I am not advocating these tools as replacements for SAS, but I am trying to give this new landscape a familiar shape, so you can adapt to the local customs, get to know the locals, and, perhaps, even marry , settle down, and have kids.

Classifying the operations to be done on files

Text files have no inherent variable structure within an observation (=line). This is very different from a SAS data set, but it is powerful too. Because there is no inherent structure you can select the method that is easiest for the situation: using column numbers or using fields defined by separators. And because you can add another command to a pipe you can actually have it both ways in the same command sequence.

These three tables have group the text processing commands by what parts of the input they keep, drop or modify. Some commands will work only on one file (or pipe) others work or two files, and many on any number of files. See Appendix for details.

Commands that work on the whole file:

Change content, by character	*Send content to pipe or commands*	*Rearrange rows*	*Split contiguous rows to files*
tr, iconv, expand, unexpand (convert spaces to tabs or vice versa)	print, echo, cat, tee, $(<file) $(command pipeline)	sort, fold, paste sort -k -m (interleave)	split (by line number), csplit (by regular expression)

[3] The ability to process text files before reading them into SAS by using the pipe facility of Filename can also save you a lot of SAS programming.

Commands that select observations

By observation number	*By content of whole line*	*By RE*	*By content of key fields*
head, tail, split	comm, uniq sort –u	grep (keep), grep –v (drop) csplit	join, comm, sort –u -k

Commands selecting only columns

Keep by column number	*Drop by column number*	*Keep by field*
cut -c	colrm	cut –f

Commands able to select columns or observations and make edits to fields

1. Pipes constructed using the previous commands
2. Commands that are also scripting languages:
    - awk
    - sed
    - Korn Shell
    - Perl
    - Ruby

These commands are all programming languages in their own right. All are the subject of several books each and if you want to read more then you are spoiled for choice. I will not use perl or ruby here because I want to focus on one line programs that are fast to develop and exploit the specificity of a problem. For example you have a filename fixed in the command. It is OK because that is the place you are working today, but if you were writing a general program you have to handle that as a parameter, check the file exists and so on, all of which makes one line a stretch.

See also the more comprehensive table in the Appendix.

HOW DO THEY FIT TOGETHER?

Piper at the gates of dawn…

The pipe is a Unix method of combining programs (often called filters) together so they run in sequence with the output from one program being the input to the next program.

This method of running program sequences has many advantages:

- easy to set up, and add or delete components, which, typically, occupy just one command line
- no need to mess around with temporary files
- the pipe components actually run in parallel as communicating processes
- because the buffers are connected to each other directly, data passing is fast

REGULAR EXPRESSIONS – WILD, WILD CARDS

Many Unix text-processing commands use regular expressions to define text for matching. They take some effort and practice to learn but they are very expressive and now they are found in SAS V9 your knowledge is transferable. The command line is a good place to learn how to write them.

Regular expressions are a formalized and developed version of the wild cards used by many systems for filename searching. Actually they were formalized before systems we use now were even thought of, see http://en.wikipedia.org/wiki/Regular_expression for an introduction and useful links.

The concept is not hard: use some characters (meta characters) to define sets of characters that can match **others**

Example: *.sas matches all filenames ending in sas. The '*' means any sequence of characters (allowed in a filename ;-)

The regular expressions we use will get more complicated than that, but not much more.
The concept may be simple but why are regular expressions hard to learn? I believe it is because we are not used to single characters having different meanings, so getting used to this takes practice and a few judicious stickies on your screen.

RE are also called patterns in some commands, and also confusingly there are different dialects of RE in different commands and terms like extended regular expressions.

REs are now implemented in many programming languages and editors, they have really moved on from being a piece of Unix exotica to mainstream computing (for programmers I mean)

Build up regular expressions piece by piece testing as you go.

Testing your regular expressions carefully is vital because if you get no matches grep will print nothing. Missing does not equal zero! Getting no lines back only means 'there are no lines like that matching the RE you typed'. Typos or incorrect

matching definitions will also give no matches. Test with a file you know will give you a hit.

A common source of confusion is that the use of * is different in an RE than it is in filename matching. In an RE * means 'zero or more matches' so labl*.sas does not match 'labl_1.sas' (it does match lablllzsas) ; what you need is 'labl.*\.sas'. The fact that the first works for filenames does not help.

Another common problem is having a meta-character in a string without realizing it. Try putting non-alphanumeric characters in [ ] – fewer meta-characters are allowed there. I often put [ ] around spaces, although it is not necessary, it much easier to read.

Some more details of RE syntax are in Appendix 2.

Example: Search a SAS log for the string ERROR:

Looking in the functional table we find grep will check for matching rows (see also hgrep in the section Further Reading). This could also be a job for awk, but with a simple string and to print the whole row grep is simpler.

```
> grep ERROR mysas.log
```

But what if the programmer doesn't know the put 'ERR''OR' trick? Then you find all the statements that print an error too.

```
> grep -E '^ERROR:' mysas.log
```

The hat (^) character means at the beginning of the line and is a simple regular expression. The -E means the string given is an extended regular expression. It is needed.

What if a wily programmer has added a space before the colon on his messages?

```
> grep -E '^ERROR[ ]*:' mysas.log
```

The [ ]* means – optional space (or spaces): [ ]+ would mean at least one space.

Oh, that wily programmer, he has used lower case for the really serious errors!

```
> grep -E '^[Ee][Rr]+[Oo][Rr][ ]*:' mysas.log
> grep -i -E '^ERROR[ ]*:' mysas.log
```

Gotcha! The first says upper or lower case E, followed by (at least one) upper or lower case R, etc. The second uses the -i option to ignore case.

So, it is 17:15, time to tidy up & go home. Phone wily programmer's boss to say saslog is OK.

> Oh great. You really checked them all? That fast?

(keys clattering)

```
> grep -i -E '^ERROR[ ]*:' *.log
```

No output.

"Yes, that fast ☺". 17:18 – out of the door.

FPW (FREQUENTLY POSED WORRIES)

Do I have to learn all these commands before I can get anything done?

> No. I have analyzed my history file and the top 5 commands in command lines and starting pipes account for about 80% of the commands lines recorded. This means you can get a long way with a grep, awk, sort, uniq. As long as you know the others exist, you can find then when you need them.

Are your tips only for one kind of Unix, or one kind of shell?

> No, the versions of these commands are pretty stable and consistent across Unix versions.
>
> Unfortunately, it is not true to say that all their options are the same (or mean the same across versions). So, if you see examples from this paper failing first check your systems man pages:
> man man
> to ask about the man (manual) command, for example.
>
> An obvious place where you may see this is in commands using cut because it is dependent on output being in fixed columns and the exact layout of output (from ls -l and ps, for example) is system dependent.
>
> The few shell examples included are run on Korn shell (http://en.wikipedia.org/wiki/Korn_shell) which is becoming more widely available since the source was opened by Lucent. The Bash shell (http://en.wikipedia.org/wiki/Bash_shell) is widely available and is pretty compatible with Korn. Both of these shells descend from the Bourne Shell (sh), and can run Bourne shell scripts without change.
>
> They have significant advantages over it for programming and interactive use. Very few people use the C-shell because of its faults.

What about Windows?

> Since 2004 Microsoft provides a 'migration support' environment free for XP intended for those moving to Windows from Unix. But you can also use the scripting tools there. See 'Further Reading' for URL. There are other commercial ports.

Won't this take a long time, (my log files are taking up about 500KB) to run on my 200 listings?

> No, I have a log scanning program that can process 500,000 lines of SAS log in about a minute, and I didn't put all the re-scanning into one awk program yet.

It looks like Chinese, how can I ever understand that gobbledygook?

> Leaving aside remarks about how common Mandarin is as a global language...
>
> Unix *is* somewhat technical and it was designed with the minimization of typing as a design goal.
>
> Users of teletypes and JCL on MVS, SINTRAN, or DCL on VMS might remember why this was seen as important...
>
> Put together this does lead to a certain lack of redundancy and significance for "every" character you type. This can be disconcerting at first.
>
> BUT, on the positive side the syntax is designed to be simple and easily parsed and this makes it simpler than human languages to understand once you 'get it'.
>
> Do not try to learn from the manuals and man[ual] pages (called help on more modern systems), these are reference materials.
>
> "DO" study the examples in this tutorial, try them out with copy&paste on your own data and modify them for your needs.

EXAMPLE PROBLEMS - ILLUSTRATING THE BUILD IT BIT-BY-BIT METHOD:

Can I see who has access to my studies?

On my home system a clinical project has a single group. So if we check the group membership we will get the answer.

```
> lsgroup MYPROJ
MYPROJ id=8996 admin=false users=scott,ccadmin,flopsy,mopsy,cottontailo ,peter
registry=files
```

But it isn't exactly easy to read. To just get the list of users:

```
> lsgroup –a users MYPROJ
MYPROJ users=scott,ccadmin,flopsy,mopsy,cottontailo[4] ,peter
```

Let us drop scott and sort by user id, but first we have to get every user on a separate line. The octal for line-feed is '\012', using tr we can translate commas to line ends. We delete the first line with the sed program, 1d means for line 1, do a delete.

```
> lsgroup –a users MYPROJ |  tr ',' '\012' |sed '1d' | sort
```

gives:

```
ccadmin
cottontailo
flopsy
mopsy
peter
```

A file I copied from Windows gives me strange messages

All text files are not equal. Windows uses two characters to mark the end-of-line and Unix only one. This extra ^M (control+M, ASCII 15 (octal) character is a legal character in a file name so it causes allsorts of issues.

There are various ways to fix it perhaps the simplest is
tr –d '\015' <inputfile >cleaned_file ; mv –f cleaned_file inputfile

I am programming the flagging of lab values, how can I see just the lines with flags?

The file is 12,000 lines long!

We want to select rows that have at least once the pattern 'number<space>flagchar'; But we do not want to see lines with only number<space>number. We avoid using just the flag characters because they occur on all the footnotes. Checking the table we see that grep can do it. What about the RE for the number? We can put in ranges in an RE so [0-9.+-] will cover

[4] Everyone can tell this is a fictional example, because we all know that Unix user ids are limited to 8 characters.

a number (beware it will not match exponential notation!). The flag characters are any mix of #$^+[5] with n intervening spaces and spaces after the number [ ]+. Putting that together we get:

```
> grep -E '[0-9]+[ ]*[#$^+]+'  labl*.lst
```

Which, when I tried it gave me

```
labl_23.lst: Week  64    31MAY2006/352    09:03  385#    434     417     786     95      146
labl_23.lst: Week  64    22MAY2006/378    09:12  404#    465#    443#    755     88      137
labl_23.lst: Week  64    10APR2006/369    09:48  407#    443     431     841     96      162
labl_23.lst: Week  24    31MAR2005/84     09:34  393#    442#    425#    790     79      216
labl_23.lst: Week  64    05JAN2006/364    10:06  403#    436#    425#    858     76      216
labl_23.lst: Week  24    25APR2005/91     08:05  346     428#    398     653     86      120
labl_23.lst: Week  64    30MAR2006/395    10:54  382     454#    429     710     93      130
labl_23.lst: Week  64    13MAR2006/369    16:31  378#    432     413     769     83      139
labl_23.lst: Week  24    13JUN2005/84     09:01  368     440#    415     698     88      144
labl_23.lst: Week  16    04JUL2005/32     09:52  407# +  422     417     928     81      135
labl_23.lst: Week  24    23JUN2005/79     07:08  327     411#    381     634     84      132
labl_23.lst: Week  64    29MAR2006/358    06:06  344     394#    376     766     93      133
labl_23.lst: Week  24    12MAY2005/86     09:54  395#    422     413     877     84      151
labl_23.lst: Week  24    12MAY2005/86     09:25  424#    430     428     973     88      153
labl_23.lst: Week  24    26MAY2005/86     08:37  394     413#    406#    912     88      137
labl_23.lst: Week  64    01MAR2006/365    15:22  389     425#    412#    838     96      142
labl_23.lst: Week  24    14JUL2005/71     09:07  382#    418#    406#    836     82      146
labl_23.lst: Week  64    03MAY2006/364    13:34  346     420#    394     679     93      144
labl_23.lst: Week  64    07SEP2005/372    07:53  366#    430     407     724     91      157
labl_23.lst: Week  64    02AUG2005/363    09:51  394     463#    439#    723     76      138
labl_23.lst: Week  16    22MAR2005/28     10:22  393#    392     392     1005    95      175
labl_23.lst: Week  24    29AUG2005/104    10:08  360#    407     391     780     79      188
labl_23.lst: Week  64    16MAY2006/364    11:32  357#    410     392     758     91      177
labl_23.lst: Week  64    21MAR2006/365    10:08  384     425#    411     819     95      191
labl_23.lst: Week  64    16MAY2006/364    07:40  354     436#    407     660     103     154
labl_23.lst: Week  24    16JUN2005/84     11:03  390#    434#    419#    805     90      148
labl_23.lst: Week -2     09NOV2004/-105   08:42  467     551 $   521 $   717     130     154
```

Helpfully grep always gives the file name if more than one file was searched. Checking the listing I notice there are flags with intervening spaces, (# +) but the flagged lines are all here. Why?

Using grep -n will give you line numbers to plug into your editor if it doesn't support REs.

Was the format catalog updated?

A simple approach without complex wildcards is

```
> ls ../../report/pgm_a/format*  ../../report/pgm_s/format*  ../../report/data_a/format*
../../report/data_s/format*
```

It assumes the current directory is one of the target directories and it lists the possible directories the files might be in. It relies on the feature of Unix wild cards that distinguishes them from the DOS/ Windows version: "." is treated the same as any other character- so formats* will match programs as well as catalogs. So,

```
-r--r-----   1 CCadmin this-study     507904 Feb  4 09:04 /vob/myproj/this-study/report/data_a/formats.sas
-r--r-----   1 CCadmin this-study     507904 Feb  1 07:34 /vob/myproj/this-study/report/data_s/formats.sas7bcat
```

- The program is newer!

This is not all, you can use flexible wildcards in filenames and "in directory names" too. So we can shorten the above example to:

```
> ls ../../report/*/format*
```

But in some cases format* or 'any directory' might list many files you are not interested in. The solution is to use other file-name expansions to limit the directories or file names you are interested in.

```
> ls -l /vob/this-study/report/@(data|pgm)_[sa]/formats.@(sas|sas7bcat)
-r--r-----   1 CCadmin this-study     507904 Feb  4 09:04 /vob/myproj/this-study/report/data_a/formats.sas
-r--r-----   1 CCadmin this-study     507904 Feb  1 07:34 /vob/myproj/this-study/report/data_s/formats.sas7bcat
[/vob/myproj/this-study/report/pgm_saf]
```

Notes:

- @(data|pgm) means data or pgm,

[5] Sharp-eyed readers will notice some of these flag characters are also RE meta-characters. The order they appear in the […] is carefully selected.

- [sa] means an s or an a, but no other character at this position,
- @(sas|sas7bcat) means one of 'sas' and 'sas7bcat' (@() is a Korn shell extension to file name expansion)

The directories searched are

> report/data_a
> report/data_s
> report/pgm_a
> report/pgm_s

Another approach would be to use compound (shell) commands.

```
[[/vob/myproj/this-study/report/data_a/formats.sas -nt /vob/myproj/this-
study/report/data_s/formats.sas7bcat ]] &&
print "run the program"
```

-**nt** is true if the left hand file is *newer than* the right hand file, && means that if the command is true, then the second command is run. The surrounding [ ] brackets means the command is evaluated as a true/false expression.

Have I run all the SAS programs in this directory?

We could check if there is a log for every program:-

```
> ls *.sas | wc -l
  200
> ls *.log | wc -l
  200
```

That's all very well, but what if there are some old listings left?

Let's just look at the dates of the listings. Then, if they are not all today we obviously have a problem.

We can use ls to get lists of files and information about them. (but do not type > that is the prompt character).

```
> ls -l *.sas
120 -rwxr-xr-x  1 dave  unknown   55380  6 Dec  1999 A55219.sas
 40 -rwxr-xr-x  1 dave  unknown   18798 27 Mar  2001 A55991_clin_stats.sas
144 -rwxr-xr-x  1 dave  unknown   71352 18 Apr  2001 basic_ods.sas
 32 -rwxr-xr-x  1 dave  unknown   13581  6 Dec  1999 dataest.sas
  8 -rwxr-xr-x  1 dave  unknown    1284  6 Dec  1999 dups.sas
 16 -rwxr-xr-x  1 dave  unknown     615 17 Apr  2001 guide.sas
 16 -rwxr-xr-x  1 dave  unknown     647 19 Mar  2001 median using SQL.sas
  0 -rwxr-xr-x  1 dave  unknown       0 27 Mar  2001 rounding.sas
168 -rwxr-xr-x  1 dave  unknown   83529 27 Mar  2001 sql_v8_examples.sas
224 -rwxr-xr-x  1 dave  unknown  111339 28 Mar  2001 ts404_distance_measures.sas
 24 -rwxr-xr-x  1 dave  unknown    8346 28 Mar  2001 varlist.sas
```

But that is too much information to check quickly for suspicious dates. We need a quick assessment of how many are modified today and how many are older.

A histogram would be nice! To turn the above listing into a histogram we have to:

1. Extract the date for each file,
2. Count the occurrences of each date,
3. Print the resulting table neatly.

How to get the dates? The date is in a fixed column, but it may or may not have a space[6]. This makes it hard to use awk. (There are ways, of course). We just need those columns, how about cut?

```
> ls -l -H -t *.sas | cut -c 38-49
8 Apr  2001
7 Apr  2001
8 Mar  2001
8 Mar  2001
7 Mar  2001
7 Mar  2001
7 Mar  2001
9 Mar  2001
6 Dec  1999
6 Dec  1999
6 Dec  1999
```

[6] Not obvious in these listings, but true. Dates within the last 24 hours are shown as times by ls.

Better! But we have duplicates…

So let us sort,

```
> ls -l -H -t *.sas | cut -c 38-49 | sort
6 Dec  1999
6 Dec  1999
6 Dec  1999
7 Apr  2001
7 Mar  2001
7 Mar  2001
7 Mar  2001
8 Apr  2001
8 Mar  2001
8 Mar  2001
9 Mar  2001
> ls -l -H -t *.sas | cut -c 38-49 | sort -u
6 Dec  1999
7 Apr  2001
7 Mar  2001
8 Apr  2001
8 Mar  2001
9 Mar  2001
```

Well, no duplicates with the sort –u but they are just deleted. So we do not have a histogram exactly. The other command that can delete duplicates is uniq, (sound of manual pages flipping) it has a count option –c!

```
> ls -l -H -t *.sas | cut -c 38-49 | sort | uniq -c
3 6 Dec  1999
1 7 Apr  2001
3 7 Mar  2001
1 8 Apr  2001
2 8 Mar  2001
1 9 Mar  2001
```

The count is in the first column, let's sort it by the first column, and by date in the second column:

```
> ls -l -H -t *.sas | cut -c 38-49 | sort | uniq -c | sort  -g -r -k 1,2 -k 3,4 -Mr -k
3,4 -gr -k 2,3
3 7 Mar  2001
3 6 Dec  1999
2 8 Mar  2001
1 9 Mar  2001
1 8 Apr  2001
1 7 Apr  2001
```

By adjusting the order of the sort you can have a histogram within year, month or day. Study the man page for sort very carefully to understand what it can do. Note: although you select the fields for sorting very flexibly you keep the whole line for printing.

How do I run every SAS program in a directory?

The best way is to use a make file, or failing that a batch script. Sometimes we want to do this and the order of the programs doesn't matter. Then we can use this strategy:

- Generate a list of all the sas files
- Pass this to a command to execute the passed names

The commands that executes its list are xargs and apply.

```
apply 'sas -noterminal -noovp %1 &'  *.sas
apply 'sas -noterminal -noovp %1 '  *.sas &
```

Note where the & character is! The first command runs each program in the background, all at once. The second runs them one at a time with the whole process (i.e., the apply command) going into the background.

The xargs version needs a pipe:

```
printf "%s\n" *.sas | xargs sas –noovp –noterminal
```

If the programs are independent (ie do not share work dataset names) then

```
sas –noovp –noterminal *.sas &
```

Should be equivalent, although only one log and listing file would be given by default.

How many patients in each lab listing? Is the count consistent with other listings and patient lists?

Let's look at an example lab listing:

```
^Lmeproj-mestudy FINAL

                                    Appendix 9.1 Listing 1-123 (Page 7 of 226)
                                                 Urinalysis data
                                                  by Treatment
                                               Safety population

Treatment: MeDrug 50mg qd

         Country/        Age/
         Center/         Sex/         Date/day of
         Patient         Race          Analysis      Test                              Result
         ------------------------------------------------------------------------------------

         ROW/0004/00010  53/M/Ca      13JUN2005/-1    Urine Bilirubin dipstick test     Negative
                                                     Urine Blood dipstick test         Negative
                                                     Urine Glucose dipstick test       Negative
                                                     Urine Ketone dipstick test        Negative
                                                     Urine Leukocytes dipstick test    Negative
                                                     pH                                6.5
                                                     Urine Protein dipstick test       Negative
                                                     Specific Gravity                  1.015

                                    31MAY2006/352    Urine Bilirubin dipstick test     Negative
                                                     Urine Blood dipstick test         Negative
                                                     Urine Glucose dipstick test       Negative
                                                     Urine Ketone dipstick test        Negative
                                                     Urine Leukocytes dipstick test    Negative
                                                     pH                                6.5
                                                     Urine Protein dipstick test       Negative
                                                     Specific Gravity                  1.015

         ROW/0004/00016  47/F/Ca      16MAR2005/-97   Urine Bilirubin dipstick test     Negative
                                                     Urine Blood dipstick test         Negative
                                                     Urine Glucose dipstick test       ++
                                                     Urine Ketone dipstick test        Negative
                                                     Urine Leukocytes dipstick test    Negative
                                                     pH                                5

Race categories: Caucasian = Ca, Black = Bl, Asian = A,  Hispanic or Latino = H/L,
                 Japanese = Jp, Native American = Na, Pacific islander = Pi, Other = Ot
Day is relative to the day of baseline (Week 5, Visit 3).

report/pgm_saf/app9l1_221.sas@@/main/1 - 03AUG2005:8:03
```

The patient label (in bold) has a fixed form (AAA/NNN/NNN (country, centre no, pat no), and is always first on a line:

Let's start by writing a regular expression to match the patient id. We can then plug that into grep.

I have added head just show the top 10 lines. The string to match is: three letters, a slash (/), three digits, a slash(/), five digits followed by one or more spaces. The RE does express that exactly but it is near enough to only get the ones we want with these listings. We could add ^[ ]* if the id appeared elsewhere on a line.

```
>  grep -E '[A-Za-z]+/[0-9]+/.+/' l1ab_21.lst | head
ROW/0001/00001  49/F/Ca    08NOV2004/-102   Urine Bilirubin dipstick test  Negative
ROW/0001/00001  49/F/Ca     17MAR2005/28    Urine Protein dipstick test    Negative
ROW/0002/00010  55/M/Ca     03MAR2005/-84   Urine Bilirubin dipstick test  Negative
ROW/0002/00010  55/M/Ca     25MAY2006/365   Urine Ketone dipstick test     Negative
ROW/0003/00004  44/F/Ca     01DEC2004/-92   Urine Bilirubin dipstick test  Negative
ROW/0003/00004  44/F/Ca     15MAR2006/378   Urine Bilirubin dipstick test  Negative
ROW/0003/00012  57/M/Ot     20DEC2004/-94   Urine Bilirubin dipstick test  Negative
ROW/0003/00012  57/M/Ot     23MAR2005/-1    Urine Protein dipstick test    Negative
ROW/0003/00015  51/M/Ot     02FEB2005/-92   Urine Bilirubin dipstick test  Negative
```

```
ROW/0003/00015  51/M/Ot     04MAY2005/-1    Urine Ketone dipstick test     Negative
```

Great – but it gives the whole line, which is too much information, especially if we plan to sort and drop keep the unique keys. We just need the ID, we could use a column selection but that might give us issues if there are extra leading spaces, or if the last code can be 3 letters. We can write our first awk program. The form of an awk program is

```
lineSelector { program}
```

If the line selector matches then the program is executed.

```
lineSelector
           { program }
```

the default program is to print and the default lineSelector is all lines.

These clauses can be many and each is tested sequentially on every line of the input. Much like a data step is automatically looping through the dataset as it is read. It also is similar to the first.byvar And last.byvar mechanisms. These also work like triggers: the code is executed when the condition is fulfilled.

In our case we want to print all lines so we just omit the lineSelector. Fields are defined by the field separator, which is "␣" space, by default. Fields are denoted by $n and the first field is called $1. One-line awk programs are enclosed in (single) quotes on the command line - so let us add

```
awk '{print $1}'
```

at the end of the pipe to extract the patient id.

```
>  grep -E '[A-Za-z]+/[0-9]+/.+/' llab_21.lst |  awk '{print $1}' | head
ROW/0001/00001
ROW/0001/00001
ROW/0002/00010
ROW/0002/00010
ROW/0003/00004
ROW/0003/00004
ROW/0003/00012
ROW/0003/00012
ROW/0003/00015
ROW/0003/00015
```

Success. But we have duplicates…,

Check the Appendix table and find that add sort and uniq to the pipe will do it:

```
> grep -E '[A-Za-z]+/[0-9]+/.+/' llab_21.lst |  awk '{print $1}' | sort | uniq  | head
ROW/0001/00001
ROW/0001/00004
ROW/0002/00001
ROW/0002/00009
ROW/0002/00010
ROW/0003/00004
ROW/0003/00005
ROW/0003/00012
```

Now we have one line per id we can count it: using wc (word count). The -l option just counts lines. And here is the final version –all on one line…,
It says there are two hundred distinct patients:

```
> grep -E '[A-Za-z]+/[0-9]+/.+/' llab_21.lst | awk '{print $1}' | sort | uniq |wc -l
     200
```

We can simplify this because the awk and grep step can all be done together by awk, but amazingly the original version is faster[7].

Could we get a count for all the lab listing files?

Let's start by adding a wild card on the file names:

```
>   grep -E '[A-Za-z]+/[0-9]+/.+/' labl1_2?.lst  | head
llab_21.lst: ROW/0001/00001  49/F/Ca    08NOV2004/-102   Urine Bilirubin dipstick test
Negative
llab_21.lst: ROW/0001/00001  49/F/Ca     17MAR2005/28    Urine Protein dipstick test
Negative
llab_21.lst: ROW/0002/00010  55/M/Ca     03MAR2005/-84   Urine Bilirubin dipstick test
Negative
```

[7] I did my tests on a multi-processor machine and that could be why – each part of the pipe runs as a separate sub-process and could be running on different processors.

```
llab_21.lst: ROW/0002/00010  55/M/Ca     25MAY2006/365   Urine Ketone dipstick test
Negative
llab_21.lst: ROW/0003/00004  44/F/Ca     01DEC2004/-92   Urine Bilirubin dipstick test
Negative
llab_21.lst: ROW/0003/00004  44/F/Ca     15MAR2006/378   Urine Bilirubin dipstick test
Negative
llab_21.lst: ROW/0003/00012  57/M/Ot     20DEC2004/-94   Urine Bilirubin dipstick test
Negative
llab_21.lst: ROW/0003/00012  57/M/Ot     23MAR2005/-1    Urine Protein dipstick test
Negative
llab_21.lst: ROW/0003/00015  51/M/Ot     02FEB2005/-92   Urine Bilirubin dipstick test
Negative
llab_21.lst: ROW/0003/00015  51/M/Ot     04MAY2005/-1    Urine Ketone dipstick test
Negative
```

Now we have too much on the line again and grep prints the name of the file for us.
First take the one line for each combination of file and patid and we tack on the now standard sort and uniq. Nearly there: we have one record per pat in a file. If we drop the patid now:

```
> grep -E '[A-Za-z]+/[0-9]+/.+/' llab_2?.lst |  awk '{print $1, $2}' | sort | uniq |
 awk '{print $1}' | head
llab_21.lst:
llab_21.lst:
llab_21.lst:
llab_21.lst:
llab_21.lst:
llab_21.lst:
llab_21.lst:
llab_21.lst:
llab_21.lst:
llab_21.lst:
```

We can now count the file names we will have the number of patients per file. Count the lines with `uniq –c` to get:

```
> grep -E '[A-Za-z]+/[0-9]+/.+/' llab_2?.lst |  awk '{print $1, $2}' | sort | uniq |
  awk '{print $1}' | uniq -c
 200 llab_21.lst:
 200 llab_22.lst:
 200 llab_23.lst:
   8 llab_24.lst:
 155 llab_25.lst:
```

Looks good! I guess 24 and 25 have subsets of patients, and we can check that, but that was a lot of page turning saved.

More importantly, tests performed like this are measuring directly what is in the deliverable; they make no assumptions about the datasets, variable names etc. And they can be scripted and run automatically so you can keep an up to date summary table or document and perhaps compare it to a version printed from SAS.

Can I extract the population used from the listing?

```
>  grep  'population' llab_2[1-9].lst  | sort | uniq | head -20
llab_21.lst:                                              Extension Safety population
llab_22.lst:                                              Extension Safety population
llab_23.lst:                                              Extension Safety population
llab_24.lst:                                              Extension Safety population
llab_25.lst:                                              Extension Safety population
```

Hmm. Interesting, can I match it up with the patient counts?

If you check the table you will discover the join command, which does what it says, joins text files on key fields.

You first store the output from the above examples into files by adding a re-direction operator at the end of the command line (or pipe). Let us assume that bit is done and we have two files that look like this:

```
> cat file-count.txt labpops.txt
 200 llab_21.lst:
 200 llab_22.lst:
 200 llab_23.lst:
   8 llab_24.lst:
 155 llab_25.lst:
llab_21.lst: Extension Safety population
llab_22.lst: Extension Safety population
```

```
llab_23.lst: Extension Safety population
llab_24.lst: Extension Safety population
llab_25.lst: Extension Safety population
```

Now look up the syntax of the join command, because the files are already sorted we can call it immediately

```
> join  -1 2 -2 1 file-count.txt labpops.txt
llab_21.lst: 200 Extension Safety population
llab_22.lst: 200 Extension Safety population
llab_23.lst: 200 Extension Safety population
llab_24.lst: 8 Extension Safety population
llab_25.lst: 155 Extension Safety population
```

Done! There is another way (just as with a SAS merge statement, with or without a by): paste is the dangerous version of merge/join that assumes the files are the same length and in the same order. We also get a duplicated key column this way. Fortunately paste has other uses as well.

```
>  paste file-count.txt labpops.txt
 200 llab_21.lst:       llab_21.lst: Extension Safety population
 200 llab_22.lst:       llab_22.lst: Extension Safety population
 200 llab_23.lst:       llab_23.lst: Extension Safety population
   8 llab_24.lst:       llab_24.lst: Extension Safety population
 155 llab_25.lst:       llab_25.lst: Extension Safety population
```

Can we extend this version to count the patients per treatment, per file?

Here is a solution to this problem. It does a two level summary by using the sort | uniq sequence twice.

The strategy is:

1. get the lines from top of each page giving the Treatment: and save the treatment label in a variable.
2. get the lines with a patient id and print just the patient id from that line, but add the filename and treatment label first.
3. run uniq to one line per patients (per file and treatment). This is because patients are more than once in a listing if the program has split parameters into two sections
4. run uniq again with -c to count how many pats per file and treatment
5. add an awk to end of pipe to print filename once and tabs to lay output out "NICELY"

```
gawk '/^[tT]reatment:/ {sub("Treatment[ ]*:",""); trt = $0 } # store the treatment
      /[A-Za-z]+[/][0-9]+[/].+/ {print  "|"FILENAME"|"trt"|"$1  }
                                # print file & treat for every line with an id
     ' app9l1_12?.lst |           # check listings 120 - 129
   sort   |
   uniq |                       # could be more than one line per patient
   awk -F"|" '{ print $2": "$3} ' |    #  select just two fields, the treatment & pat id
   uniq -c |                     # now count patients per file & treatment
   awk '{ if ( oldfile != $2 ) {print "\nFile ",$2" :";
                                oldfile = $2}}  # print file name once when it changes
        {print "\t"$1"\t" $3,$4,$5,$6,$7,$8}
        # print for all lines:
        # add tabs for first two fields but not others (part of treatment name)
       '
```

might output:

```
File  app9l1_121.lst: :
        35      Bio 30mg qd (core) + combination
        87      Draino 50mg bid (core) + combination
        78      Draino 50mg qd (core) + combination

File  app9l1_122.lst: :
        35      Bio 30mg qd (core) + combination
        87      Draino 50mg bid (core) + combination
        78      Draino 50mg qd (core) + combination

File  app9l1_123.lst: :
        35      Bio 30mg qd (core) + combination
        87      Draino 50mg bid (core) + combination
        78      Draino 50mg qd (core) + combination
```

```
File  app9l1_124.lst: :
        2       Bio 30mg qd (core) + combination
        4       Draino 50mg bid (core) + combination
        2       Draino 50mg qd (core) + combination

File  app9l1_125.lst: :
        27      Bio 30mg qd (core) + combination
        67      Draino 50mg bid (core) + combination
        61      Draino 50mg qd (core) + combination
```

- paste-able version (change filename wild card!):

```
gawk '/^[tT]reatment:/ {sub("Treatment[ ]*:",""); trt = $0 } /[A-Za-z]+[/][0-9]+[/].+/
{print  "|"FILENAME"|"trt"|"$1  } ' app9?l1_12?.lst | sort   | uniq | awk -F"|" '{ print
$2": "$3} ' | uniq -c | awk '{ if ( oldfile != $2 ) {print "\nFile ",$2" :"; oldfile =
$2}} {print "\t"$1"\t" $3,$4,$5,$6,$7,$8} '
```

I need to make a last check of all the titles to ensure they match the specifications

Is it possible to list the titles, when there are different numbers of title lines in each output?

Some versions of grep have the –p option that prints the paragraph (set of lines delimited by a blank lines) around the matched text. If there is text that is on every first page title then

```
> grep –ip '(Page 1 of' *.lst
```

If you do not have this version of grep is there still a way?

Yes, this can be done using awk. The key to the trick is to set the field (FS) and record (RS) separators: set field separator to \n (line end) and the record separator to null (awk interprets this as a blank line).

The awk program has the string to be matched in /…/ followed by the program to be executed when a match is found enclosed in {…}.

FILENAME is an awk system variable (variables in awk are not preceded with $ like in shells), $0 is the whole record (i.e. a paragraph in this case).

```
> gawk 'BEGIN { FS= "\n" ; RS = ""} /Page 1 of/ { print FILENAME":\n" $0 } ' *.lst

labl_10.lst:
                                          Appendix 10 Listing 3-10 (Page 1 of 50)
            Patients with abnormal biochemistry values (in conventional US unit) by
treatment
                                                Extension Safety population

labl_11.lst:
                                           Appendix 10 Listing 3-11 (Page 1 of 8)
                    Laboratory normal and notable ranges and relevant percent change
criteria
                                    by laboratory identification number and laboratory group
```

Store this in a file for editing or printing by adding

```
>filename.txt
```

at the end of the pipe. Use >| if you will run it twice.

How about getting the first page?

If we set the RS to page feed and line end as field separator we get one (logical) record per page. Then we can extract all first pages to keep track of changes. "\014 is ^L, ctrl+L, or page feed.

```
> gawk 'BEGIN { FS= "\n" ; RS = "\014"} /Page 1 of/ { print "\014"FILENAME": " $0 }
' *.lst
```

to save the file in a date stamped filename, add: (include >, It is not the prompt here.)

```
>firstpage-$(date  +%Y-%m-%d:%H:).txt
```

How can I check all *my RTF files were re-run OK today?*

This example is very interesting for me. When I saw this problem (and not all the RTF files had listings so using the lst files was not an option) I thought there would be no way without learning a lot of RTF and writing a parser (this has already been done for Perl, incidentally). But, one day I looked at an RTF file and thought – all I need is the matching string out of the file. The markup doesn't matter.

So looked for a way to extract the matching text and there is an awk function to do it.

I wrote this gawk program:

- It assumes the date the file was written is in a line containing the program file path, which begins with /report/ (the path is report/key or report/tier1, or report/tier2, perhaps)
- On these lines there is a date - which is the file write date
- We need to find the lines with a date (why not match on lines containing any date?).
- Then extract the date from the line...
- This is done with the match function: it returns 1 if it found a matching string, and then
- It puts the location into awk system variables that we can plug in to the substring function to get the date into a variable.
- Then if we found a date on the line the print statement prints the file & date

```
> gawk  '/report\// { match($0,"report[/].*\sas");
       file=substr($0,RSTART,RLENGTH);
       ok=match($0,"[0-9]+[A-Z]+[0-9]+");
       date=substr($0,RSTART,RLENGTH);
       if (ok) print date, file } ' *.rtf |
 sort -t" "   |
 uniq
```

Note:

- This test does not catch RTF files that do not have a date or are empty (i.e. one blank page), because a blank page still has RTF mark-up in it.
- I added extra lines for readability beware if copying & pasting.
- We could speed the program up by only extracting the date from one page per RTF file. Gawk can do this using the nextfile statement.

CONVERTING ONE-LINERS TO SCRIPTS

When?

The simple answer is: “When one line is not enough”. A subtler one is “when you can simplify reusing it by adding parameters”.

How?

Start with a copy paste of the pipe into a text file. If you add line breaks add them *after* the pipe symbol. It is best to add a shriek-bang line or shebang as the first line

```
#! /bin/ksh
```

makes it explicit that the script will be run as with the Korn shell. This convention means that type extensions and conventions are not needed for Unix scripts.

CONCLUSION

I hope I have convinced you it is worth learning to use Unix text tools for helping in your daily work. It does take some effort: but the programs you write are light weight and once you have attained a certain level of familiarity you can use them for ad hoc problems that would not be worth writing a bigger SAS program for.

SUMMARY:

- Unix text tools occupy a space complimentary to SAS
- The rewards are worthwhile especially for short focussed, perhaps unique problems.

RECOMMENDED READING

There are many Unix tutorials and other resources on the web, but you will certainly need access to your systems manual. In addition I recommend:

Unix Power Tools (O’Reilly, http://www.oreilly.com/catalog/upt3/index.html), because it is task oriented and therefore gives lots of examples of how to fit Unix bits & pieces together. It is not a book to read from cover to cover, but is great to dip into for ideas and tricks. It covers general Unix issues as well (e.g. an excellent explanation of what the Unix permissions are and how they work).

How to solve it by G. Polya is summarized at http://www.math.utah.edu/~pa/math/polya.html and published by Princeton University Press, 1945, 2nd ed 1957, ISBN 0-691-08097-6.

The New Unix, article at O’Reilly on lamp: http://www.onlamp.com/lpt/a/2901

Unix philosophy: start from: http://www.faqs.org/docs/artu/ch01s06.html but don't miss The Power Way for those moments when a quiet small voice of calm is needed : http://mercury.ccil.org/~cowan/upc/ and if writing one-liners gets too easy there is always the *Unix Koans of Master Foo* at http://www.catb.org/~esr/writings/unix-koans/

Microsoft's free Unix tools: http://www.microsoft.com/technet/itsolutions/interop/sfu/pshscrpt.mspx

Regular expressions. For learning these get an editor that supports them and practice on some of your files, and a giant crib sheet is at http://regexlib.com/default.aspx there are probably others. The hgrep script (download it from here: http://www.shelldorado.com/scripts/cmds/hgrep.txt) will print matching lines and highlight the matched text, *very* useful while learning and testing regular expressions. Get it onto your system with wget, curl, or ftp.

Awk: there are various texts on awk programming, and an excellent free guide to the Gnu project awk, called gawk. If you have gawk available use it. The diagnostics are better than earlier awk versions. It also has a profiling and pretty printing tool.

Pipes and SAS: there is a useful article with SAS examples at http://support.sas.com/sassamples/quicktips/pipes_0702.html

Others: The web site for this tutorial is at: http://DaveG.tiddlyspot.com/

APPENDIX TABLE – FUNCTIONAL KEY TO UNIX TEXT TOOLS

UNIX TEXT COMMANDS:

This table classifies the commands that can process text (in files, or in pipes) by the type of operations they can do. It enables you to easily find a command to do what you need. The next step is to try the defaults to see if that is enough and then to refer to the man page otherwise.

If the column marked edit is YES it means there is a way to change the values either of one field, or the whole record. In awk and sed this is done with the sub(stitute) functions.

Although I have included Perl here it is not much used for one-liners, but is a good basis for developing general and portable programs.

The separator is given with the option used to set it. "\t" means the tab character. Many awk versions allow a regular expression to be the separator. The separator can also be a non-printable character (e.g. \015, or ^L, page feed).

Program	Type	Select Rows?	Select Cols?	Default field separator	Edit?	Multiple files?	Comments
cut	2	NO	YES, by col number or field	-d"\t"	NO	YES	cut -f (by field) cut -c (by character position)
join	2	NO	YES, by field content	-t"\t"	YES, select any non-key columns	TWO files only	Print the common lines. Files must be sorted on the fields used. -a prints unique lines. (Like SAS merge ; by ,)
head	1	YES	NO	-	NO	YES	First -n observations from top of file or pipe.
tail	1	YES	NO	-	NO	NO (some versions, YES)	Last -n observations from bottom of file or pipe. -f option keeps dynamic watch
sort	1	YES	NO	-t"\t"	NO	YES	Sorts on entire record by default, can drop records with duplicate keys (-u) and merge sorted files (-m) (like set ; by ;)
fold	0	NO	NO	col number		NO	Wraps long lines
tr iconv	0	NO	NO	-	YES	YES	Translates every occurrence of a character(s). iconv converts between encodings.
paste	2	NO	YES	-d"\t"	NO	YES	Merges content of files in to one column by column
wc	0	NO	NO	-	NO	YES	Counts characters, words and lines

Program	Type	Select Rows?	Select Cols?	Default field separator	Edit?	Multiple files?	Comments
diff sdiff	0	NO	NO	-	NO	TWO files	Prints differences between files
comm	3	YES	NO whole line is matched	-	NO	TWO files	List lines in common, or in only one of the files
col	2	NO	NO	-	NO	YES	Processes reverse line feeds, not for selecting columns.
colrm	2	NO	YES	startcol - endcol	NO	YES	Copy only the given columns to stdoutput
print echo printf	0	NO	NO		NO	YES	Print the input (KSH Shells)
sed	4	YES	YES	none	YES	YES	Editing, line selection and substitution
awk	4	YES	YES	-F" "	YES	YES	print matching records; select fields as $1,$2..., programmable
perl	4	YES	YES	YES		YES	Power string and data processing tool. OO programming
fmt	0	NO	NO	-	NO	YES	Fill lines to the same length
look	3	YES	NO	-	NO	YES	Returns lines where start of line matches the string given. File must be sorted
locate	3	YES	NO	-	NO	Finds file names	Not all Unix systems
diff	0	NO	NO	-	NO	TWO	Locates differences in the files
patch	4	NO	NO	-	YES	ONE	Applies difference file as an edit to a target file

APPENDIX 2 – MORE ABOUT REGULAR EXPRESSIONS

More on Regular Expressions

Regular expressions are often termed patterns.

Rules:[8]

- Patterns are applied line by line; therefore they cannot match across a line boundary.
- Matches to a pattern are found anywhere on the line, unless there is an anchor in the expression.

'ERROR' is more general than '^ERROR'

Components

Expressions to be matched have *characters*, *counts*, *ranges*, *alternatives* and *anchors*

- *Characters*. Match these kinds of characters:

1. a normal character matches itself,
2. . full stop, decimal point '.' matches any character
3. [abc] matches a, or b, or c
4. [a-z] is a *range* that matches any lower case letter (but not ö or é)
5. [^a] is a *range* that matches any character except 'a'
6. (abc|xyz) matches either of the two *alternative* sets of three characters abc *or* xyz

[8] These are informal statements with caveats and exceptions ignored (e.g. there *are* ways to match across line boundaries).

- Match these *repetitions*:

1. * any number of occurrences (including none)
2. + at least once
3. {m,n} match if there are at least m, but at most, n, occurrences of the previous RE (a single character, group or sequence. (NB. not all implementations.)

- In these *places*:

1. ^ the start of the line
2. $ at the end of a line

- We can express the line scope rule as saying that a pattern is treated as ^.*//pattern//.*$ unless ^ or $ are explicitly included in the pattern.

Sample patterns:

[_A-Za-z][_A-Za-z0-9]{0,7}	matches a SAS variable identifier
.*\.sas7b[cd]at	matches the file names of SAS catalogues and datasets
population	matches the string 'population', anywhere on the line

How much is a match?

The regular expression dialects implemented by different tools are not all the same, nor are they equal. In one respect the awk and grep engines have a limitation: they only allow so-called greedy matching. Most of the time this does not matter, but sometimes it will hit you. An example will explain. It uses the hgrep script which runs a grep then highlights in the matched string where the match is:

```
> ~/hgrep -E '.*/' paths
paths:lala/dipsy/po
paths:flopsy/mopsy/cottontailo/my.sas
```

The match is underlined and in orange. Why is it not the / after lala or flopsy? After all lala/ also matches '.*/'.

This is because the longest possible matching string is taken, hence the name greedy. Some RE engines (Perl, for example) also allow non-greedy matching to be specified.

ACKNOWLEDGEMENTS

I would like to thank BSI AG for allowing me time to write this tutorial. My thanks also to Mia and Asitta for tolerating my strange working hours and genuinely enjoying my company.

CONTACT INFORMATION

I would value your comments and questions on this paper. The tips and tricks will be gathered on my web-site, at http://DaveG.tiddlyspot.com/ . Please contact the author at:

David J Garbutt
BSI AG
Täfernstrasse 16A
CH – 5405 Baden-Dättwil
Work Phone: +41 56 484 1920
Fax: +41 56 484 1930
Email: d.garbutt@bsiag.com or =djg
Web: www.bsiag.com

I would be very happy to receive feedback on what helped you from these examples and interested in suggestions for additional examples, especially with solutions.