PERL

Practical Extraction and Report Language
or
Pathologically Eclectic Rubbish Lister

CS/EE Systems Group
Workshop Lecture

Trent A. Fisher

Based on a lecture by
Tom Christiansen
CONVEX Computer Corporation

Overview

- What is *Perl*: features, advantages, disadvantages, preview
- Data Types: scalars and various sorts of arrays and their associated builtin functions.
- Regular Expressions
- Operators
- Flow Control
- I/O: regular I/O, system functions, directory access, formatted I/O
- Functions and Subroutines
- Advanced topics: eval, dbm file access, dynamic scoping suid scripts, debugging, packages, library packages and command line options
- Examples

What is Perl?

- An interpreted language that looks a lot like C with built-in *sed*, *awk*, and *sh*, as well as bits of *csh*, *Pascal*, FORTRAN and BASIC-PLUS thrown in.

- Highly optimized for manipulating printable text, but also able to handle binary data.

- Especially suitable for system management tasks due to interfaces to most common system calls.

- Rich enough for most general programming tasks.

- *“A shell for C programmers.” [Larry Wall]*

Features

- Easy to learn because much derives from existing tools.
- More rapid program development because it's an interpreter
- Faster execution than shell script equivalents.
- More powerful than *sed*, *awk* or *sh*.
- Translators available for old *find*, *sed* and *awk* scripts
- Comes with a symbolic debugger (itself written in Perl).
- Portable across many different architectures.
- Absence of arbitrary limits like string length.
- It's free!

Disadvantages

- Difficult to learn because much derives from existing tools.
- No complex data types.
- Slower than C.
- Tends to use large amounts of memory.
- Powerful rather than elegant.
- Feeping Creaturism.

Where to get it

- Any *comp.sources.unix* archive

Famous archive servers

- uunet.uu.net 192.48.96.2
- tut.cis.ohio-state.edu 128.146.8.60

Its author, Larry Wall <lwall@jpl-devvax.jpl.nasa.gov>

- jpl-devvax.jpl.nasa.gov 128.149.1.143

Resources

- The man "page."
- *Programming Perl* by Wall and Schwartz.
- Perl reference guide (in postscript form) also available from Ohio State, along with some sample scripts.
- A Texinfo Reference Guide (available via *info*)
- Also, convex.com has many sample scripts plus *comp.lang.perl* archives.
- USENET newsgroup *comp.lang.perl* good source for questions, comments, examples.
- Locally, a collection of scripts can be found in /home/rigel/diner/banzai/perl, and also in ˜trent/src/perl.

Preview

It's not for nothing that *perl* is sometimes called the *pathologically eclectic rubbish lister.*

Before you drown in a deluge of features, here's a simple example to whet your appetites that demonstrates the principal features of the language, all of which have been present since version 1.

```
while (<>) {
    next if /^#/;
    ($x, $y, $z) = /(\S+)\s+(\d\d\d)\s+(foo|bar)/;
    $x =~ tr/a-z/A-Z/;
    $seen{$x}++;
    $z =~ s/foo/fear/ && $scared++;
    printf "%s %08x %-10s\n", $z, $y, $x
     if $seen{$x} > $y;
}
```

General Coding Info

- Executable scripts start with **#!/usr/local/bin/perl**.
- Use standard # for comments.
- No need to use backslash for continuation lines.
- White space almost never matters.
- Strings may contain newlines.
- Statements end in semi-colons.
- Blocks use braces like C.

Data Types

Perl has one basic data type: *scalars*.

- Scalars are either string, numeric, or boolean, **depending on context**.
- Boolean is much like C: values of 0 (zero) and “” (null string) are false; all else is true.
- Numbers are stored, internally, as double precision.
- Variables are not declared, except for locals.
- Variables automagically initialized to the null string or 0 depending on the context.
- Precede the identifier with “$” to indicate a scalar value.

```
$foo = 3.14159;

$foo = $foo + 1;
```

Quoting

- Quoting is (almost) like the shell: single, double, backtick and *here* documents.

```
$foo = 'red';
$foo = 'I paid $4275000.32';

$foo = "was $foo before";
$foo = "was ${foo}ish before";

$host = `hostname`;

$foo = <<SPAM;
Spam, eggs, sausage and $spam.
SPAM
```

Special Scalar Variables

Special scalars are named with punctuation (except **$0**). Examples are

- **$0** name of the currently executing script
- **$_** default for pattern operators and implicit I/O
- **$$** the current pid
- **$!** the current system error message from *errno*
- **$?** status of last ‘backtick‘, pipe, or system
- **$|** whether output is buffered
- **$.** the current line number of last input
- **$[** array base, 0 by default; *awk* uses 1
- **$<** the real uid of the process
- **$(** the real gid of the process
- **$>** the effective uid of the process
- **$)** the effective gid of the process

Many others (one for every symbol on the keyboard, plus one).

Data Types

- Scalars can be aggregated into indexed arrays (lists) or and associative arrays.

- Type of variable determined by leading special character.

$	scalar
@	indexed array (lists)
%	associative array
&	function (call)
*	all of the above

- All data types have their own separate namespaces, as do labels, functions, and file and directory handles.

Gotcha

- No nested data structures or C style structs. However, associative arrays often can fake it well enough.

Data types (lists)

- Indexed arrays a use **$** for one scalar element, @ for all or a portion

```
$foo[$i+2] = 3;       # set one element to 3
@foo = ( 1, 3, 5 );   # init whole array
@foo = ( );           # initialize empty array
@foo = @bar;          # copy whole @array
@foo = @bar[$i..$i+5]; # copy slice of @array
@foo[3..5] = @bar;    # first 3 elements of bar
```

- Lists can be represented by a series of values enclosed on parenthesis.

```
($foo, $bar, $glarch) = ('eggs', 'sausage', 'spam');
($foo, $bar) = ($bar, $foo);       # exchange
$fullname = (getpwnam("trent"))[6];
print (('a'..'z')[(15, 4, 17, 11)];
```

- **$#ARRAY** is index of highest subscript, and evaluating **@ARRAY** in a scalar context will return the number of elements.

- The builtin variable **$[** defines the starting index of arrays. 0 by default.

List Weirdness

- Interpolating a list in double quotes will produce elements separated by spaces. Without quotes, all elements are concatenated.

- If a backquote evaluation is done into an array, each line of output is put into a separate element (w/ a newline).

```
@yppasswd = `ypcat passwd`;
chop(@yppasswd);
```

- There are two contexts: scalar and array. You can use the **scalar** function to force an array returning function to supply a scalar. For example:

```
$foo = <FOO>;          # get next line
@foo = <FOO>;          # get whole file
@foo = scalar(<FOO>);  # get next line as $foo[0]
```

- Attempts to combine lists will result in one flat list.

```
@foo = ( 1, 3, 5 );
@bar = ( 2, 4, 6 );
@new = ( @foo, 10, @bar );
print "@new\n";
☞  1 3 5 10 2 4 6
```

Special Array Variables

@ARGV

Command line arguments. So the script's name is **$0** and its arguments run from **$ARGV[0]** through **$ARGV[$#ARGV]**, inclusive. For example, a simple-minded **echo** could be written as:

```
print "@ARGV\n";
```

@INC

search path for files called with **do**

@_

default for **split** and subroutine parameters.

Built-in Array Functions

- Indexed arrays function as lists; you can add items to or remove them from either end using these functions:

pop	remove last value from end of array
push	add values to end of array
shift	remove first value from front of array
unshift	add values to front of array

For example:

```
push(@list, $bar);

push(@list, @rest);

$tos = pop(@list);

while ( $arg = shift(@ARGV) )  { }

unshift( @ARGV, 'zeroth arg', 'first arg');
```

Built-in Array Functions (split and join)

- **split** breaks up a string into an array of new strings. You can **split** on arbitrary regular expressions, limit the number of fields you **split** into, and save the delimiters if you want.

- By default, uses **$_** for input, @_ for output, and **/\s+/** for the delimiter.

- In a scalar context, it returns the number of matches.

```
@list = split(/[, \t]+/, $expr);

while (<PASSWD>)
{
    ($login, $passwd, $uid, $gid, $gcos,
        $home, $shell) = split(/:/);
}
```

- The inverse of **split** is **join**.

```
$line = join(':', $login, $passwd, $uid,
            $gid, $gcos, $home, $shell);
```

Built-in Array Functions (sort, grep, reverse)

- **reverse** inverts a list. (if given a scalar it reverses it).

```
foreach $tick (reverse 0..10) { }
$is_palindrome = reverse($foo) eq $foo;
```

- **sort** returns a new array with the elements ordered according to their ASCII values. Use your own routine for different collating.

```
print sort @list;

sub numerically { $a - $b; }
print sort numerically @list;
```

- **grep** returns a new list consisting of all the elements for which a given expression is true. For example, this will delete all lines with leading pound signs

```
@lines = grep(!/^#/, @lines);
```

- In a scalar context, it returns the number of lines in the result.

Associative Arrays

- Instead of being indexed by just integers, these may be indexed by any scalar.

- Use **$** for one scalar element, **%** for all. Use curly braces for indexing.

```
$frogs{'green'} += 23;    # 23 more green frogs

%bar = ( 1, 'one', 2, 'two' );

%foo = %bar;          # copy whole %array

@frogs{'green', 'blue', 'yellow'} = (3, 6, 9);
```

- Multi-dimensional arrays can be faked. The real index will be joined with **$;** as a separator.

```
$location{$x, $y, $z} = 'troll'; # multi-dim array
```

- Use **defined** operator to check for existence of an element.

```
if ($foo{'bar'})      # might be 0
if (defined $foo{'bar'})
```

- An element can be removed with **delete**.

```
delete $table{$killed};
```

Special Associative Array Variables

%ENV

the current environment; e.g. **"$ENV{'HOME'}"**

%SIG

The list of signal handling subroutines.

```
sub trapped {
    print STDERR "Interrupted\007\n";
    exit 1;
}
$SIG{'INT'} = 'trapped';
```

Built-in Array Functions (%arrays)

For manipulating associative arrays, the **keys** and **values** functions return indexed arrays of the indices and data values respectively. **each** is used to iterate through an associative array to retrieve one **($key,$value)** pair at a time.

```
while (($key,$value) = each %array)
{
    printf "%s is %s\n", $key, $value;
}

foreach $key (keys %array)
{
    printf "%s is %s\n", $key, $array{$key};
}

print reverse sort values %array;
```

Regular Expressions

- Understands *egrep* regexps, plus

- Meta-characters
  - **\w**, **\W** alphanumerics plus _ (and negation)
  - **\d**, **\D** digits (and negation)
  - **\s**, **\S** white space (and negation)
  - **\b**, **\B** word boundaries (and negation)

- C-style escapes recognized, like **\t**, **\n**, **\034**

- Don't escape these characters for their special meaning: **( ) | { } +**

- Character classes may contain metas, e.g. **[\w.$]**

- Special variables: **$&** means all text matched, **$‘** is text before match, **$’** is text after match.

Regular Expressions (continued)

- Matches are done against $_ by enclosing the regular expression in slashes. It returns 1 if it matches, 0 otherwise.

```
print if /^Subject: /;
```

- A trailing **i** means the matching will be case-insensitive.

- Matches within parenthesis are put into **\1** .. **\9** within regular expressions; **$1** .. **$9** outside.

```
if (/^this (red|blue|green) (bat|ball) is \1/)
    { ($color, $object) = ($1, $2); }
if (/(\d\d):(\d\d):(\d\d)/) {
    $hours = $1;    $minutes = $2;    $seconds = $3;
}
```

- In an array context, the match operator returns the matches within parens.

```
($color, $object) =
    /^this (red|blue|green) (bat|ball) is \1/;
```

Regular Expressions (continued)

- Substitute and translation operators are like *sed*'s **s** and **y**.

```
s/alpha/beta/;
s/(.)\1/$1/g;
y/A-Z/a-z/;
```

- Use **=~** and **!~** to match against variables

```
if ($foo !~ /^\w+$/) { exit 1; }
$foo =~ s/\boregon\b/OR/i;
```

- The **/e** modifier means evaluate the RHS as an expression.

```
$_ = "7abc123xyz\n";
s/\d+/3 * $&/eg; # "21abc369xyz\n";
```

Operators

Perl has a rich set of operators, many which are stolen from other languages (like C).

- *Perl* has an operator precedence scheme as complex as C.
- Standard operators: **+**, **-**, *****, **/**, **%**
- Bitwise operators: **|**, **&**, **^**, **<<** and **>>**
- Assignment: All of the above operators can be combined with **=**.

```
$foo += 42;
```

- Auto-increment. Both pre- and post- increment and decrement.

```
$foo++;
$foo = --$bar;
```

- exponentiation: **, ****=**
- range operator: **..** In a scalar context, it indicates a line range; in an array context it generates a list of the given range.

```
$inheader = 1 if /^From / .. /^$/;
if (1..10) { do foo(); }
for $i (60..75) { do foo($i); }
@new = @old[30..50];
```

- string concatenation: **.**, **.=**

```
$x = $y . &frob(@list) . $z;
$x .= "\n";
```

Operators (continued)

- string repetition: **x**, **x=**

```
$bar = '-' x 72; # row of 72 dashes
```

- Numeric comparison: **==**, **!=**, **<**, **>**, **<=**, **>=** and **<=>**

- String tests: **eq, ne, lt, gt, le, ge** and **comp**

```
if ($x eq 'foo') { }
if ($x ge 'red' ) { }
```

- Ternary operator: **?:**

- Comma operator.

- Search or substitution operators: **=~**, **!~**

- file test operators like augmented */bin/test* tests work on strings or filehandles

```
if (-e $file)  { }    # file exists
if (-z $file) { }     # zero length
if (-O LOG) { }       # LOG owned by real uid
die "$file not a text file" unless -T $file;
```

Scalar Built-ins

- Besides the powerful regular expression features, several well-known C string manipulation functions are provided, including **crypt**, **index**, **rindex**, **length**, **substr**, and **sprintf**.

- The **chop** function efficiently removes the last character from a string. It's usually used to delete the trailing newline on input lines. Like many *perl* operators, it works on **$_** if no operand is given.

```
chop($line);

chop ($host = `hostname`);

while (<STDIN>)
{
    chop; ...
}
```

- The **substr** function can be assigned to:

```
$phrase = "in big words";
substr($phrase, 3, 3) = "other";
print $phrase, "\n";
☞  in other words
```

More Scalar Built-ins

- *Perl* has the usual set of mathematical functions: **atan2**, **cos**, **exp**, **int**, **log**, **sin**, **sqrt**, **rand**, **srand**.

- Conversion functions: **hex**, **oct** and **ord**.

- Time functions: **time**, **times**, **localtime** and **gmtime**.

- Bitwise manipulation: **vec**.

Flow Control

- Unlike C, blocks always require enclosing braces { }

- **unless** and **until** are just **if** and **while** negated

  - if (EXPR) BLOCK else BLOCK
  - if (EXPR) BLOCK elsif (EXPR) BLOCK else BLOCK
  - while (EXPR) BLOCK
  - do BLOCK while EXPR
  - for (EXPR; EXPR; EXPR) BLOCK
  - foreach $VAR (LIST) BLOCK

- For readability, **if**, **unless**, **while**, and **until** may be used as trailing statement modifiers as in BASIC-PLUS

```
return -1 unless $x > 0;
```

- **do** takes 3 forms

  - execute a block
    ```
    do { $x += $a[$i++] } until $i > $j;
    ```
  - execute a subroutine
    ```
    do foo($x, $y);
    ```
  - execute a file in current context
    ```
    do 'subroutines.pl';
    ```

Flow Control (continued)

- Use **next** and **last** rather than C's **continue** and **break**

- **redo** restarts the current iteration, ignoring the loop test

- A block for a loop can have a "continue" part, which will be executed for each iteration and if **next** is used.

```
foreach $i (@list)
{
    next if $i =~ /spam/;
    ...
}
continue
{
    $last = $i;
}
```

- Blocks (and **next**, **last**, and **redo**) take optional labels for clearer loop control, avoiding the use of **goto** to exit nested loops.

- **||** and **&&** work as in the shell. They evaluate left to right, using short-circuit logic, and return the last expression evaluated.

```
open(FILE, $file) || die "$file: $!";
unlink($bar) && print "ok\n";
```

I/O

- Filehandles have their own distinct namespaces, but are typically all upper case for clarity. Pre-defined filehandles are **STDIN**, **STDOUT**, **STDERR** and **DATA**.

- The **DATA** filehandle refers to everything after the **__END__** in the script.

- Mentioning a filehandle in angle brackets reads next line in scalar context, all lines in an array context; newlines are left intact.

```
$line = <TEMP>;
@lines = <TEMP>;
```

- **<>** means all files supplied on command line (or **STDIN** if none). When used this way, **$ARGV** is the current filename.

- When used in a **while** construct, input lines are automatically assigned to the **$_** variable.

- It is common to iterate over file a line at a time, assigning to **$_** (implicitly) each time and using that as the default operand.

```
while ( <> )
{
    chop;
    next if /^#/;     # skip comments
    s/left/right/g;   # global substitute
    print;            # print $_
}
```

I/O (continued)

- If not using the pseudo-file **<>**, use **open** to create new ones.

```
open (PWD,      "/etc/passwd");
open (TMP,      ">/tmp/foobar.$$");
open (LOG,      ">>logfile");
open (TOPIPE,   "| lpr");
open (FROMPIPE, "/usr/etc/netstat -a |");
```

- **open** returns 1 on success (or the child's pid for a pipe), 0 otherwise.

```
open (PWD, "/etc/passwd") || die "ACK!";
```

- Both **print** and **printf** take a filehandle as an optional first argument.

```
printf LOG "%-8s %s: weird bits: %08x\n",
    $program, &ctime, $bits;
```

- May also use **getc** for character I/O and **read** for raw I/O.
- Access to **eof**, **seek**, **close**, **flock**, **ioctl**, **fcntl**, and **select** calls for use with filehandles.
- Access to **mkdir**, **rmdir**, **chmod**, **chown**, **link**, **symlink** (if supported), **stat**, **rename**, **unlink** calls for use with filenames.

Gotcha

- Don't forget to close pipes and files.

Formatted I/O

- Besides **printf**, formatted I/O can be done with **format** and **write** statements.

- Automatic pagination and printing of headers.

- Picture description facilitates lining up multi-line output

- Fields in picture may be left or right-justified or centered

- Multi-line text-block filling is provided, something like having a **%s** format string with a built-in pipe to **fmt**)

- These special scalar variables are useful:

  $% for current page number,

  $= for current page length (default 60)

  $- for lines left on page

Formatted I/O (example)

```
# a report from a bug report form; taken from perl man page
format top =
                        Bug Reports
@<<<<<<<<<<<<<<<<<<<<<<<     @|||         @>>>>>>>>>>>>>>>>>>>>>>>
$system,                      $%,         $date
------------------------------------------------------------------
.


format STDOUT =
Subject: @<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
         $subject
Index: @<<<<<<<<<<<<<<<<<<<<<<<<<<<< ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<
       $index,                       $description
Priority: @<<<<<<<<<< Date: @<<<<<<< ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<
          $priority,        $date,   $description
From: @<<<<<<<<<<<<<<<<<<<<<<<<<<<<< ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<
      $from,                         $description
Assigned to: @<<<<<<<<<<<<<<<<<<<<<< ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<
             $programmer,            $description
~                                    ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<
                                     $description
~                                    ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<
                                     $description
~                                    ^<<<<<<<<<<<<<<<<<<<<<<<...
                                     $description
.
```

Directory Access

Three methods of accessing directories are provided.

- You may open a pipe from */bin/ls* like this:

```
open(FILES,"/bin/ls *.c |");
while ($file = <FILES>) { chop($file); ... }
```

- The directory-reading routines are provided as built-ins and operate on directory handles. Supported routines are **opendir**, **readdir**, **closedir**, **seekdir**, **telldir**, and **rewinddir**. For example:

```
opendir(DIR, ".");
@files = readdir(DIR);
print "@files";
```

- The easiest way is to use *perl*'s file globbing notation. A string enclosed in angle brackets containing shell meta-characters evaluates to a list of matching filenames.

```
foreach $x ( <*.[ch]> ) { rename($x, "$x.old"); }
chmod 0644, <*.c>;
```

Database Access

- Associative arrays bay be bound to *dbm* files with **dbmopen()**.

```
dbmopen(ALIAS,'/usr/lib/aliases',undef)
    || die "can't dbmopen aliases: $!";
```

- Use **each** not **keys** for big *dbm* files.

```
$alias = $alias["postmaster\0"];
chop $alias;
print "postmaster -> $alias0;
```

Gotcha

- Many *dbm* files have null bytes on the end of the keys and values.

System Functions

A plethora of functions from the C library are provided as built-ins, including most system calls. These include:

- **chdir**, **chroot**, **exec**, **exit**, **fork**, **getlogin**, **getpgrp**, **getppid**, **kill**, **setpgrp**, **setpriority**, **sleep**, **syscall**, **system**, **times**, **umask**, **wait**.
- If your system has Berkeley-style networking, **bind**, **connect**, **send**, **getsockname**, **getsockopt**, **getpeername**, **recv**, **listen**, **socket**, **socketpair**.
- **getpw***, **getgr***, **gethost***, **getnet***, **getserv***, and **getproto***.

Handling Binary Data

- *Perl* strings can hold arbitrary binary data.
- The functions **pack** and **unpack** can break data up into strings.
- Each takes a “template” describe the conversion.

```
$var = pack(TEMPLATE, LIST);
@ary = unpack(TEMPLATE, EXPR);
```

- A template is a string which specifies a type (i.e. byte, short, float) and a quantity, or * for the rest of the data.
- Spaces are ignored in a template (for readability).
- Examples:

`S4`	Four unsigned shorts
`i f c8`	An int, a float, and 8 chars
`A20 A*`	A 20 byte string and the rest as a string
`c3 x i`	3 chars, skip one byte, and an int

- For example, to read */etc/utmp*:

```
while ( read(utmp,$utmp,36)) {
    ($line,$name,$host,$time)
     = unpack('A8 A8 A16 l',$utmp);
    ...
```

- **unpack** is more efficient than **substr** at breaking up static data

Bit Strings

- To help in manipulating binary data, and to support the BSD **select** call, there is a special kind of string called a ***bit-vector***.

- The bitwise operators **|**, **&**, **^** and **~** assume bit-vector mode if given strings.

- Like **substr**, **vec** may be used as an lvalue.

```
$rin = 0;
vec($rin, fileno(SOCKET), 1) = 1;

print unpack("b*", $rin);
```

Gotcha

If you evaluate a bit-vector in a numeric context, they lose their magic.

Subroutines

- Order of definition doesn't matter.
- Functions may be called recursively or indirectly.
- Subroutines called either with '**do**' operator or with '**&**'.
- Scalars, lists and arrays may be passed as parameters. However, they all become one list.

```
do foo(1.43);

do foo(@list)

$x = &foo('red', 3, @others);

@list = &foo(@olist);

%foo = &foo($foo, @foo);
```

Subroutines (continued)

- Parameters are received by the subroutine in the special array @_.
- Parameters are passed by reference, so they should be copied to local variables.

```
$result = &simple($alpha, $beta, @tutti);

sub simple {
    local($x, $y, @rest) = @_;
    local($sum, %seen);
    return $sum;
}
```

- Subroutines may also be called indirectly

```
$foo = 'some_routine';
do $foo(@list)
($x, $y, $z) = do $foo(%maps);
```

- Since @_ supplies references to the variables given as parameters, these variables may be modified. For example:

```
sub invflag
{
    if ((@_[0] != 0) && (@_[0] != 1)) { @_[0] = 0; }
    else { @_[0] ^= 1; }
}
```

Dynamic scoping

- The scope of **local()** is through the end of its block and is visible in subroutines called in that block.

```
sub wisconsin {
    local ($_bar_) = 18;  &drink();
}

sub illinois {
    local ($_bar_) = 21;  &drink();
}

sub drink {
    if ($AGE < $_bar) { &bouncer(); }
    else              { &imbibe(); }
}

$state = "wisconsin";
&$state();
```

Eval

- The **eval** operator lets you execute dynamically generated code.

- Any syntax or runtime errors (including **die** messages) will be put in **$@** and the **eval** will return the undefined value.

- An **eval** counts as a block, so you can declare locals within that will last until it's done. but other variables and subroutines will persist.

- For example, to process any command line arguments of the form **variable=value**, place this at the top of your script:

```
eval '$'.$1."'$2';"
    while $ARGV[0] =~ /^([A-Za-z_]+=)(.*)/ && shift;
```

Eval Applications

- The **eval** operator is also useful for run-time testing of system-dependent features which would otherwise trigger fatal errors. For example, not all systems support the **symlink** or **dbmopen**.

```
eval "symlink(\$old, \$new)"
   || warn ("oops: " .
     ($@ ? "no symlinks"
         : "symlink ($old $new): $!"));
```

- Another use of **eval** and **die** is to return control from a deeply nested subroutine, much as **setjmp()** and **longjmp()** do in C.

```
eval '&func1';
die $@ unless $@ =~ /exception:/i;

sub func1 { &func2; }
sub func2 { &func3; }
sub func3 { die "EXCEPTION: returning"; }
```

- While you can't make an array of arrays, you can make an array of array names.
- For example, you can use **@name** to contain the names of arrays (which actually contain the data).

```
$ary = $name[$i];
$val = eval "\$$ary[$j]";
```

Packages

- Using **package**s you can write modules with separate namespaces to avoid naming conflicts.
- Variable are accessed by the **package'name** notation.

```
$packname'variable++;

open (APACK'FILE, "<$somefile") || die;

for (keys %network'listeners)  { }
```

- Identifiers that are not fully-qualified are considered to be in the current package.
- The null package is the same as the main package: **$'foo** is the same as **$main'foo.**
- *Perl* programs begin in the **main** package.
- Use the **package** statement to switch to another package.
- The scope of a **package** statement is lexical, not dynamic, and lasts until the end of the current block.

Suid Scripts

- *Perl* programs can be made to run setuid, and can actually be more secure than the corresponding *C* program.

- Because interpreters have no guarantee that the filename they get as the first argument is the same file that was **exec**'ed, *perl* won't let your run a setuid script on a system where setuid scripts are not disabled.

- Using a dataflow tracing mechanism triggered by setuid execution, perl can tell what data is safe to use and what data comes from an external source and thus is "tainted."

- Tainted data may not be used directly or indirectly in any command that modifies files, directories or processes or else a fatal run-time error will result.

Perl Libraries

Perl comes with many useful library routines.

shellwords
Breaks a line up like the shell (quotes).

termcap
Accesses the termcap database

complete
Interactive completion of words

getopt, getopts
For parsing command line arguments, similar to it's C counterpart.

importenv
Turn the environment into perl variables, i.e. **$HOME**.

pwd
Sets up current directory tracking, i.e. **$CWD**.

ctime
For printing dates.

bigint, bigrat, bigfloat
Arbitrary precision math.

Require and Search Paths

- To use a library, use **require**:

```
require 'Getopts.pl'
```

- *Perl*'s library search path is in **@INC**.

- The **PERLLIB** environment variable is a colon delimited list of pathnames which will be prepended to **@INC** if set.

- Perl will only include a file once under the same name. It maintains a table in **%INC** of what's been included.

Debugging

- When invoked with the **-d** switch, *perl* runs your program under a symbolic debugger (written in *perl*). It can:
- Set breakpoints, conditional or unconditional
- Print variables, list source code
- Produce stack traces
- Single step through code
- Set arbitrary actions at certain lines

Because it uses **eval** on your code, you can execute any arbitrary perl code you want from the debugger.

- You can enter the debugger without a script via:

```
perl -de 0;
```

Command Line Options

The following are the more important command line switches recognized by *perl*:

- **–v** print out version string
- **–w** issue warnings about error-prone constructs
- **–d** run script under the debugger
- **–e** like *sed*: used to enter single command lines
- **–n** loop around input like *sed -n*
- **–p** as with **-n** but print out each line
- **–i** edit files in place
- **–a** turn on autosplit mode (like *awk*) into **@F** array
- **–P** call C pre-processor on script

Common mistakes

- Forgetting the semi-colon at the end of a statement.
- Leaving off the **$** in the left-hand side of an assignment, or in a **foreach** loop.
- Using **==** instead of **eq**.
- Putting a comma after a filehandle in a print statement.

```
print STDOUT, "hello, world\n";
```

- Forgetting the **&** in a function call.
- Using **else if** or **elif** instead of **elsif**.
- Forgetting to save **$1** .. **$9** before doing other matches or substitutions.
- Forgetting to enclose a *BLOCK* in curly braces.
- Not **chop**ing input lines, or results from a backtick'ed command.
- Saying @**foo[1]** instead of **$foo[1]**.
- Different behaviours in scalar and array context.
- Not using **-w** to find typos

Examples: Command Line

```
# output current version
perl -v

# simplest perl program
perl -e 'print "hello, world.\n";'

# useful at end of "find foo -print"
perl -n -e 'chop;unlink;'

# add first and last columns (filter)
perl -a -n -e 'print $F[0] + $F[$#F], "\n";'

# in-place edit of *.c files changing all foo to bar
perl -p -i -e 's/\bfoo\b/bar/g;' *.c

# run a script under the debugger
perl -d myscript
```