PERL

PRACTICAL EXTRACTION AND REPORT LANGUAGE

Matt Smith (UAHuntsville / ITSC)
msmith@itsc.uah.edu
Kevin McGrath (Jacobs ESTS)
kevin.m.mcgrath@nasa.gov

1 November 2010

OUTLINE

- Intro
- Syntax
- Running Perl programs
- Pragmas
- Variables
- Modules
- String manipulation
- Control structures
- File manipulation
- I/O
- Regular expressions
- Subroutines/Functions
- System commands
- Date/Time manipulation
- Csh examples
- External apps

INTRODUCTION

- A general-purpose programming language originally developed for text manipulation and now used for a wide range of tasks including system administration, web development, network programming, GUI development, and more.
- Intended to be practical (easy to use, efficient, complete) rather than beautiful (tiny, elegant, minimal).
- Major features
  - Easy to use
  - Supports both procedural and object-oriented (OO) programming
  - Has powerful built-in support for text processing
  - Has one of the world's most impressive collections of third-party modules

WHY PERL?

- It is a *complete programming language*!
- Still – it's just another wrench in your toolbox
- Some simple tasks are still QED in c/b/ba/k-shell
- You don't have to compile and create object files and then execute
- Ability to perform floating-point arithmetic
- Can still do easy file manipulation, like a shell
- There is a MS Windows version, if interested
- Fewer external commands required (use internal perl functions)
- A plethora of libraries (modules) are available

SYNTAX

- Leading blank spaces and tabs are ignored
- End a command line with a semicolon ';'
- Indenting is good practice, and Smart!
- Variables are case sensitive
- # Comments, but no multi-line comment syntax
- Escape character backslash '\'
- Variables *always* have prefix character ($, @, %)
- Kevin McGrath has a suggested documentation template
  - At the SPoRT web site

RUNNING PERL SCRIPTS

- The "Hello world" script:

```
#!/usr/bin/perl -w          # hash-bang
print "Hello world!\n";     # Speak
```

- The first line (starting with the "shebang" '#!' interpreter directive) tells the kernel that this is a perl script and where to find the perl interpreter
- The -w switch tells perl to produce extra warning messages about potentially dangerous constructs
- Next, we use 'print' to print a string (in quotes) consisting of printable characters and a newline or linefeed '\n' (0x0A)
- Note the comments – everything to the right of a '#' on a line
- Usually placed in a file with a "pl" extension (e.g., test.pl)
- An `exit` is implied, though you can return a status code (`exit #;`)

PRAGMAS

- They turn on/off compiler directives, or *pragmas*
- E.g.,

```
use integer;       # perform integer math
use strict;        # restrict unsafe constructs. Use it!
use strict 'vars'; # strict only for variables
use strict 'subs'; # strict only for subroutines
use warnings;      # as opposed to …
no warnings;
use constant PI=>4*atan2(1,1); # argumentless inline func.
```

- There are many pragmas

VARIABLES

- $ for scalars (*a single value or string*)
  - E.g., `$a, $Joe54, $my_name`
- @ for arrays (*list of scalars*)
  - E.g., `@List, @files_to_read, @months`
- % for hashes (AKA *associative arrays* or *'key/value' arrays*)
  - E.g., `%mcidas_res, %Coins`

DEFINING AND USING SCALARS

```
$name="Bubba Gump";                # a string
$number=12;                        # an integer
$avg=3.254;                        # a float
$total=${amps}**2+$volts;          # an expression
$name=${first}.${last};            # '.' concatenates strings
$Email="msmith\@itsc.uah.edu"; # escape the '@'
```

- All calculations are performed internally using floats
- Scalars are handled as strings in a string context, and as numbers in a numeric context

```
$a=42;
$answer="The ultimate answer is " . $answer;
```

DEFINING AND USING ARRAYS (LISTS)

```
@list=("Gary", "Steve", "Bill"); # strings
@ListOfNumbers=(1..100);         # index generator
@odd=(1, 2, 12.34, "Bob");       # odd but fine
@empty=();                       # array w/ 0 elements
```

Accessing arrays is C-like – using brackets [ ] – and they're 0-based.
Use `$` when dealing with one element of an array.

`$odd[2]` contains 12.34

They will grow to accommodate new elements. So,

```
$stations[99]=1000; # generates a 100 element array
```

When used in a scalar context, an array evaluates to its length. So,

```
$length=@stations;  # length is now 100
```

DEFINING AND USING HASHES

```
%coins=(Quarter=>25, Dime=>10, Nickel=>5);
```

```
my $name="Dime";
my $total = $coin{$name} + $coin{Nickel};
```

`$total` now contains 15

- Multidimensional hashes

```
my %goes = (
   vis => {band => 1, res => 1, loc=>"GHCC_GE/VIS"},
   wv  => {band => 3, res => 4, loc=>"GHCC_GE/IR3"});
my $channel = $goes{vis}->{band};
print "GOES-East WV has a res of $goes{wv}->{res} km\n";
```

SCOPE

- Normally, every variable has a global scope. Once defined, every part of your program can access a variable.
- When variables are declared with `my()`, they are only visible inside the code block. Any variable which has the same name outside the block is ignored.

```
$name = "Rover";
$pet = "dog";
print "The $pet is named $name\n";    # The dog is named Rover
{
   my $name = "Spot";          # local instance of $name
   $pet = "cat";               # overwrites $pet defined above
   print "The $pet is named $name\n"; # The cat is named Spot
}
print "The $pet is named $name\n";    # The cat is named Rover
```

SCOPE (CONT'D)

- When the `use strict` pragma is used (highly recommended)…
  - Each variable *must* be declared with either `my` or `our`
  - Declaring variables using `our` expands their scope beyond the block in which they are defined
  - Variables must be declared with `our` if you wish to make them visible to subroutines. The variables then must be "imported" into your subroutines (more on that later).

MODULES

- Modules expand the number of available functions (in addition to those “built-in”)
- Near top of code, list modules to `use`, using this syntax:

```
use module::name;
```

- The Comprehensive Perl Archive Network (CPAN) http://www.cpan.org/ has a huge list of documented modules that are publicly available.
- Example:

```
use File::Copy;
copy file1, file2;
move file1, file2;
```

- Many functions that serve as wrappers for syscalls return ***true*** on success, and ***undef*** on failure.

MORE MODULES

- Some modules are not already installed and require installation by SysAdmin
- Example modules:

```
use Math::Trig; # tan, cos, sin, acos, asin, pi, deg2rad, etc.
use Statistics::Basic; # median, mean, variance, stddev, etc.
use File::Basename; # basename, dirname
use Image::Magick;  # read, crop, contrast, draw, etc.
use GD; # rectangle,transparent, colorAllocate, Font, etc.
use NetCDF;  # open, varget, varput, close, etc.
use Net::FTP;# FTP functions
use PDL::IO:? # various Perl Data Language I/O modules
      FITS, GD, Grib, HDF, HDF5, IDL
```

STRING MANIPULATION

- Strings can be stored in any variable type (scalar, array, and hash).
- Enclosed in “quotes” ($ or @ variables are evaluated at run-time)
- Enclosed in ‘apostrophes’ ($ or @ variables are NOT evaluated)
- Dot operator “.” concatenates strings
- Repetition operator ‘x’ repeats
- Use `eq/ne/lt/le/gt/ge` for string comparisons (not ==/!=/</<=/>/>=)
- Special operators =~ and !~ (later)

```
$name = $first . " " . $last;
$fourSixes = "6"x4; # gives "6666"
@fours = ("4")x4;   # gives a list (array): ("4", "4", "4", "4")
if ($name eq "Smith") { $match = 1; }
```

STRING MANIPULATION (CONT'D)

- Concatenating and adding strings and numbers

```
$a=1; $b="hello";
$c=$a.$a;   # $c=11          (treats 1 as "1")
$c=$a+$b;   # $c=1           (*hello isn't numeric*)
$c=$a.$b;   # $c="1hello"    (treats 1 as "1")
$c=$b+$b;   # $c=0           (*hello isn't numeric*)
```

- Leading blanks and trailing non-numerics are ignored
- " 123.45tom" becomes 123.45
- Functions that expect a numeric will interpret strings as 0
- *undef* is interpreted as 0

STRING MANIPULATION (CONT'D)

- `lc("Hello")` returns "hello" (lowercase)
- `uc("Hello")` returns "HELLO" (uppercase)
- Most string input from STDIN (standard input) and other read functions end with a newline. This WILL bite you! To remove it, use chomp:

```
print "What is your name?\n";
$name=<STDIN>;
chomp($name);
```

Note:

STDIN = Standard Input

STDOUT = Standard Output,

STDERR = Standard Error

STRING MANIPULATION (CONT'D)

- How to tell if a variable is a number?
- Use an external function in a *module* ("looks_like_number")

```
use Scalar::Util 'looks_like_number';
print "Enter a number: ";
while (! looks_like_number(<STDIN>)) {
   print "Not numeric, try again: ";
}
```

SPLIT/JOIN

- `@array_variable = split(/separator/, string);`

```
my $data = "Becky Windham,25,female,Madison";
my @values = split(/,/, $data);
values[0] contains "Becky Windham"
values[1] contains 25
values[2] contains "female"
values[3] contains "Madison"
```

- If no separator is given, / / (space) is assumed
- If no string is given, $_ is assumed
- Regex example

```
my @pieces = split(/\d+/, $data); # split on one or more digits
```

CONTROL STRUCTURES: IF

- Very similar to csh and C. `elsif` and `else` are optional. Note the missing "e" in `elsif`.

```
$month = `date +%m`;
chomp($month);
if ($month == 1) {
   print "The month is January.\n";
} elsif ($month == 2 || $month == 3) {
   print "It's February or March.\n";
} else {
   print "It's after March.\n";
}
```

- `and` and `&&` are interchangeable, as are `or` and `||`

CONTROL STRUCTURES: WHILE & DO

```
print "How old are you? ";
$a = <STDIN>;  chomp($a);
while ($a > 0) {          # note optional use of parentheses
   print "At one time, you were $a years old.\n";
   $a--;
}
```

- The opposite of `while` is `until`
- `do` is similar to `while`, except that the expression is evaluated at the end of the block. The contents of the `do` block will be executed at least once.

```
$day = 0;
do {
   $day++;
   print "Processing data for day: $day.\n";
} while $day < 10;     # note optional lack of parentheses
```

CONTROL STRUCTURES: FOR/FOREACH

```
for ($i = 1; $i <= 10; $i++) {
   print "$i\n"; }

for ($j = 0; $j <= 100; $j+=5) {
   print "$j\n"; }

@a = (1..4);
foreach (@a) {
   $square = $_** 2;            # $_ default loop variable
   print "The square of $_ is $square\n"; }

@a = (1,2,3,4);
foreach $number (@a) {
   $square = $number * $number;
   print "The square of $number is $square\n";}
```

CONTROL STRUCTURES

- `last` is similar to break statement of C.
  - Whenever you want to quit from a loop.
- To skip the current loop use the `next` statement.
  - It immediately jumps to the next iteration of the loop.
- The `redo` statement is used to repeat the same iteration again.

DIE/WARN

- `die` throws an exception – printing a message to STDERR

```
open($file, ">tempfile") or die "error opening tempfile\n";
```

- `warn` doesn't throw an exception, but still prints a message
- This code:

```
if ($T_ob > $limit-2) {
    print "Temp $T_ob near limit\n";
}
```

…can be written as:

```
($T_ob <= $limit-2) or warn "Temp $T_ob near limit\n";
#   Note: 'if' implied in usage with warn or die
```

OPEN/CLOSE FILES

- `open FILEHANDLE, MODE, "filename"`

Mode	Operand	Create	Truncate
Read	<		
Write	>	x	x
Append	>>	x	
Read/write	+<		
Read/write	+>	x	x
Read/append	+>>	x	

```
open LOGFILE, ">>log.txt" || die "Cannot open log.txt!";
```

- To print to a file, use `print FILEHANDLE " ";`
- Use `close(FILEHANDLE)` to close a file

FILE MANIPULATION

- Use `unlink` to remove files. Returns 1 if successful, 0 if unsuccessful.

```
unlink("sample.txt", $filename, "$dir/$user/tempfile");
unlink glob("2010_11*");
unlink <*.gif>;          # quotes optional with < >
foreach (<*.gif>) {
   unlink || warn "I'm having trouble deleting $_";
}
rename "file23", $new_file;      # if you only want to rename
```

- To copy or move a file…

```
use File::Copy;
copy "log.txt", $newFile;
move $file, "${SPoRT_ADAS_DIR}/$newfile";
```

FILE AND DIRECTORY TESTS

- To test if a file or directory exists, use `if (-e $filename)`. Returns a true-false condition.
- Other useful tests:

File Test	Meaning	File Test	Meaning
-e	File or directory is exists	-l	Entry is a symlink
-r, -w	File or directory is readable/writable	-T	File is “text”
-z	File exists and has zero size	-B	File is “binary”
-s	File exists and has nonzero size	-M	Modification age in days
-d	Entry is a directory	-A	Access age in days

FILE STATUS

- To get detailed information about a file, call the `stat` function. Time is in seconds since the epoch and size is in bytes.

```
($dev,$ino,$mode,$nlink,$uid,$gid,$rdev,$size,$atime,$mtime,
   $ctime,$blsize,$blocks) = stat($fileName);
```

or

```
($size,$mtime) = stat($fileName)[7,9];
```

- The File::Stat module is a by-name interface to the stat function:

```
use File::stat;
$status1 = stat($fileName1); $status2 = stat($fileName2);
$ageDiff = $status2->mtime - $status1->mtime;
print "$fileName2 is $ageDiff seconds older than fileName1";
```

READING TEXT FILES

- To read a text file line-by-line, you can use:

```
my @lines = <FILEHANDLE>;
```

- Alternatively, you could process the file line by line using…

```
while (<FILEHANDLE>)) {
    print "Processing $_";
}
```

or

```
while (my $line = <FILEHANDLE>) {...}
```

- Remember you may want to chomp each line!

TERMINAL INPUT

- <STDIN> can be abbreviated by using simple <>. By declaring a scalar variable and setting it equal to <STDIN> we set the variable equal to whatever will be typed by a user at the command prompt.

```
print "What is the radius of the circle? ";
$r=<>;       # chomp not required in numeric context
$diameter = (2 * $r);
$area = (3.14 * ($r ** 2));
$cir = $diameter * 3.14;
print " Radius: $r\n Diameter: $diameter\n Circumference:
   $cir\n Area: $area";
```

COMMAND LINE ARGUMENTS

- Command line arguments are stored in the `@ARGV` array
- Access the elements as you would any other array (`$ARGV[0]`)
- `$#ARGV` to examine the size of the array
- Example code:

```
my $channel = $ARGV[0] || die "No argument passed!\n";
print "Processing GOES $channel data…\n"; }
```

- Executing the code:

```
cmd> goesImager.pl IR
```

- Returns:

```
Processing GOES IR data…
```

BINARY I/O

```
my $val;                          # scalar for storing data
my @r, @g, @b;                    # arrays for r, g, b
open(OUTP, ">output.fil");        # open output file
binmode OUTP;                     # place OUTP in binary mode
. . .                             # fill RGB arrays with 256 values
$val=pack('L', 0x80808080);       # pack McIDAS missing data value
print OUTP $val;                  # write to OUTP
$val=pack('N256', @r);            # pack values into Red array
print OUTP $val;                  # write Red array to OUTP
. . .                             # pack & write G & B arrays
. . .
$val=pack('N48', 0);              # pack 48 Reserved words - empty
print OUTP $val;                  # write to OUTP
```

PRINT/PRINTF

```
$number = "5";
$string = "Hello, PERL!";
$float = 12.39;
$ddd = 9;
$nothing = undef;   # assign an empty (undefined) value
print "$number\n";                          # 5
print "$string\n";                          # Hello, PERL!
print "$float\n";                           # 12.39
printf "Value:%8.4f\n", $float;             # Value: 12.3900
$doy = sprintf ("%03d", $ddd);
print "Day of Year = $doy\n";               # Day of Year=009
print "There is nothing: $nothing\n";       # There is nothing:
```

REGULAR EXPRESSION EXAMPLES

- Complex string comparisons

```
if ($string =~ m/sought_text/)  # m is the "match" operator.
```

- Complex string selections

```
if ($string =~ m/whatever(sought_text)whatever2/)
$soughtText = $1;
```

- Complex string replacements

```
$string =~ s/originaltext/newtext/; # s is the "substitute"
   operator.
```

- Parsing based on the above abilities

```
if ("20100501_T_212.grib" =~ m/^20100501_(.)_212.grib$/)  #true
```

SUBROUTINES/FUNCTIONS

- `sub NAME BLOCK`
- Use all lower case names (suggestion)
- BLOCK is code within braces { }
- Arguments may be passed

```
print_greeting("The year is", 2010);

sub print_greeting {
   $string = $_[0];                  # Grab passed arguments
   $year = $_[1];
   print "$string $year\n";          # Prints "The year is 2010"
}
```

SUBROUTINES/FUNCTIONS

- An `our` declaration declares a global variable that will be visible across its entire lexical scope, even across package boundaries. To use global variables in a subroutine while using `strict`, you must "import" them.

```
#!/usr/bin/perl –w
use strict;
{  our $name = "Kevin";
   our $office = 3031;
   printinfo();              # Call subroutine printinfo
}
sub printinfo {
   # Use the following variables defined in the main block
   use vars qw($name $office);  # Or use vars ("$name", "$office");
   print " Name: $name\n Office: $office\n";
}
```

SYSTEM COMMANDS

There's more than one way to skin a cat.

- System command (returns command status)

```
$stat = system("mv out.dat /tmp/junk");
```

- Backticks (returns command's output)

```
$output = `imglist.k GHCC_GE/IR4 | grep "18:45"`;
```

- Many Perl methods can replace system commands

```
@list = <*.dat>;          # Use instead of @list = `ls *.dat`
```

- Some security risks exist with non-Perl system calls

DATE / TIME MANIPULATION

- use Date::Manip; # full set of functions (quicker subsets exist)

```
$ddd = Date_DayOfYear($mm, $dd, $yyyy);
```

- Default date format is ***yyyymmddhh:mm:ss***

```
$date = ParseDate("2nd Sunday in 2011"); # returns yyyymmddhh:mm:ss
$date = ParseDate("39 minutes ago");     # returns yyyymmddhh:mm:ss
$future = DateCalc($date, "12 hours later");# 12 hours from $date
$past = DateCalc($date, -30:00:00);          # 30 hours before $date
```

(Did you know that the Unix *date* command can act similarly?)

- Determine delta between two dates/times

```
$diff=DateCalc($date1, $date2); # Returns "y:m:d:h:m:s"
```

- Increment a date/time
- Use/convert/compare almost any date/time format

CSH EXAMPLES

- Environment variables
  - Getting

```
$path = $ENV{"PATH"};
```

  - Setting

```
$ENV{"PATH"} = $path . ":/home/kmcgrath/mcidas/data";
```

- Running external Perl scripts

```
do "/data/user/setEnv.pl" || die "Error\n";
```

- Change working directory

```
chdir($dataDir);
```

FTP EXAMPLE

The Net:FTP module implements a simple FTP client in Perl. Methods return true or false to indicate operation success.

```
use Net::FTP;
$ftp = Net::FTP->new("ssd.nesdis.noaa.gov", Passive => 1);
$ftp->login($ftpUser, $ftpPassword);
$ftp->binary();
$ftp->get($remoteFile, $localDir/$file) || die "get failed " .
   $ftp->message;
$ftp->quit();
```

MCIDAS EXAMPLE

You can execute many McIDAS commands as system commands (e.g., imglist.k) and gather output returned, but screen manipulation (e.g., FRMSAVE) requires a McIDAS session – via *mcenv*.

Note: McIDAS commands need POSITIONAL PARAMETERS and KEYWORDS to be in all-caps, but not the executable name itself.)

```
$output = `mcenv -f 600x844 -e 10m -g 8 -i 240 << EOC
imgdisp.k GHCC_GE/IR4 MAG=-1 -2 LAT=$lat $lon
map.k VH
frmsave.k X $gifName FORM=GIF
exit
EOC`;
print $output;
```

EMAIL EXAMPLE

```
#!/usr/bin/perl -w
my $recipients = "msmith\@itsc.uah.edu kevin.m.mcgrath\@nasa.gov";
my $subject = "Perl test";
my $product = "LIS 12-Hour Forecast";

my $status = `/bin/mail -s "$subject" $recipients << EOF
This is the body of the email.
The $product isn't updating.
EOF`;

if ($status ne "") {
        print "error with mail\n";
        exit 1;
}
```

WEB SITES FOR HELP

www.perl.com

www.perl.org

perlmonks.org

perldoc.perl.org

www.tutorialspoint.com/perl

www.perltutorial.org

www.tizag.com/perlT