Introduction to Perl

or, *Learn Perl in Two Hours*

MU Information & Access Technology Services Short Course

Information & Access Technology Services, University of Missouri - Columbia
Document update: 2 April 1999
Location: http://www.cclabs.missouri.edu/things/instruction/perl/

Contents

1. What Is Perl?

Perl is a *"Practical Extraction and Report Language"* freely available for Unix, MVS, VMS, MS/DOS, Macintosh, OS/2, Amiga, and other operating systems. Perl has powerful text-manipulation functions. It eclectically combines features and purposes of many command languages. Perl has enjoyed recent popularity for programming World Wide Web electronic forms and generally as glue and gateway between systems, databases, and users.

2. Course Requisites and Goals

This course presumes participants have elementary programming experience in a procedural programming language such as C, Pascal, or Basic; and access to a system with Perl 4 or Perl 5 installed, such as MU Information & Access Technology Services's SHOWME and SGI/NeXT systems.

By completing this course and its homework, you should be able to:

1. Locate reference materials and other resources related to Perl;
2. Express fundamental programming constructs such as variables, arrays, loops, subroutines and input/output in Perl;
3. Understand several concepts relatively idiosyncratic to Perl, such as associative arrays, Perl regular expressions, and system interfaces;
4. Program in Perl for data manipulation, file maintenance, packaging or interfacing system facilities, and for "Common Gateway Interface" Web applications. CGI resources will be examined in more depth in a subsequent MU Information & Access Technology Services Short Course.

To keep this a *short* course, we won't explain object-oriented concepts and some other facilities appropriate for large projects. Perl, perhaps more than any other computer language, is full of alternative ways to do the same thing; we tend to show only one or two. We will try to stimulate by examples of useful bits of code, results, and questions. Turn to the reference materials for further explanation.

3. Perl References and Resources

MU Perl and Perl CGI Materials:, `http://www.cclabs.missouri.edu/things/instruction/perl/`

MU On-line Perl Manual: `http://www.cclabs.missouri.edu/cgi-bin/perlman.cgi`

One of many sites for Perl Software: `http://www.cis.ufl.edu/perl/`

Usenet Newsgroups: `news:comp.lang.perl.announce, news:comp.lang.perl.misc.`

Reference Manual:

> Larry Wall, Tom Christiansen & Randall L. Schwartz, *Programming Perl*, 2nd Edition September 1996, 670 pages, O'Reilly and Associates, Inc., ISBN 1-56592-149-6, $39.95. The "Camel Book". A thorough Perl 5 reference with plenty of examples. See `http://perl.oreilly.com`.

Tutorials:

> Randal L. Schwartz, *Learning Perl*, 1993, 274 pages, O'Reilly and Associates, Inc., ISBN 1-56592-042-2, $24.95 The "Llama Book". 1st edition introduces programming concepts with Perl 4.
>
> Tom's Object-Oriented Perl Tutorial: `http://language.perl.com/all_about/perltoot.html`
>
> Randy's Column on OO Perl: `http://www.stonehenge.com/merlyn/UnixReview/col13.html`

Homework!

Learn how to search the Perl 5 online manual using the above URL with Netscape, Mosaic, or another World Wide Web browser! Find an alternative expansion of the word "Perl" as an acronym involving the word "eclectic". contents

4. State of Perl

Two types of programmers use Perl. System administrators like it for the way it glues together system commands to manipulate data and processes, and for its pattern-matching functions aids in system searches and reporting. People developing electronic forms for Unix Web servers find Perl easier to learn and use than C, and for their purposes Perl offers more built-in or publicly available functions such as easy data validation and simple databases.

The Perl code in this document works under both Perl 4.036 (the *last* version of Perl 4) and Perl 5. Perl 5 adds object-oriented capabilities and several other conveniences. The Reference Guide identifies with a double-dagger (as in ++) new features of Perl 5. For major general-purpose Perl applications, particularly CGI scripts and client or server applications, first check the 'Net for Perl modules that you can exploit! See the FAQ "Perl 5 Module List" regularly posted to the Usenet group *comp.lang.perl.announce*.

5. Taste of Perl

Quite useful Perl programs can be short. Suppose we want to change the same text in many files. Instead of editing each possible file or constructing some cryptic find, awk, or sed commands, you could issue a single command:

Example: Amazing Perl One-Liner That Substitutes Text In Multiple Files

```
perl  -e 's/gopher/World Wide Web/gi'  -p  -i.bak  *.html
```

This command, issued at the Unix prompt, executes the short Perl program specified in single quotes. This program consists of one perl operation; it substitutes for original word "gopher" the phrase "World Wide Web", (globally, ignoring case). The re mainder of the Unix command indicates that the perl program should run for each file ending in ".html" in the current directory. If any file "blah.html" needs changing, a backup of the original is made as file "blah.html.bak". *Programming Perl* lists additional handy one-liners.

For those accustomed to "classic" procedural programming, the "amazing one-liner" above can be expanded in Perl in a style more like C or Pascal:

Example: Global Substitution, The Scenic Route

```
#!/usr/local/bin/perl -w
# File:  go2www
# This Perl program in classic programming style changes
# the string "gopher" to "World Wide Web" in all files
# specified on the command line.
# 19950926 gkj
$original='gopher';
$replacement="World Wide Web";
$nchanges = 0;
# The input record separator is defined by Perl global
# variable $/.  It can be anything, including multiple
# characters.  Normally it is "\n", newline.  Here, we
# say there is no record separator, so the whole file
# is read as one long record, newlines included.
undefine $/;

# Suppose this program was invoked with the command
#     go2www ax.html  big.basket.html  candle.html
# Then builtin list @ARGV would contain three elments
# ('ax.html', 'big.basket.html', 'candle.html')
# These could be accessed as $ARGV[0] $ARGV[1] $ARGV[2]

foreach $file (@ARGV) {
    $file = $ARGV[$i];
    if (! open(INPUT,"<$file") ) {
        print STDERR "Can't open input file $bakfile\n";
        next;
    }

    # Read input file as one long record.
    $data=<INPUT;
    close INPUT;

    if ($data =~ s/$original/$replacement/gi) {
        $bakfile = "$file.bak";
        # Abort if can't backup original or output.
        if (! rename($file,$bakfile)) {
            die "Can't rename $file $!";
        }
        if (! open(OUTPUT,"$file") ) {
            die "Can't open output file $file\n";
        }
        print OUTPUT $data;
        close OUTPUT;
        print STDERR "$file changed\n";
        $nchanges++;
    }

    else {  print STDERR "$file not changed\n"; }
}
print STDERR "$nchanges files changed.\n";
exit(0);
```

Questions:

1. What do you guess that the "!" means, as in:

   ```
   if (! open(OUTPUT,"$file") ) {
       die "Can't open output file $file\n";
   }
   ```

2. What does the "" probably mean here? Compare with "open(INPUT ...)".

3. What does "die" do?

4. Some languages use "IF ... THEN DO ... END; ELSE IF ... THEN DO ... END". How is this notated in Perl?

5. What does $nchanges++ do?

The Perl Creed is, "There is more than one way!" This noble freedom of expression however results in the first of the four

Perl Paradoxes: *Perl programs are easy to write but not always easy to read.* For example, the following lines are equivalent!

```
if ($x == 0) {$y = 10;}  else {$y = 20;}
$y = $x==0 ? 10 : 20;
$y = 20;  $y = 10 if $x==0;
unless ($x == 0) {$y=20} else {$y=10}
if ($x)  {$y=20} else {$y=10}
$y = (10,20)[$x != 0];
```

6. Storing and Running Perl Programs

Homework: Hello

To test your ability to store and run a Perl program, enter and execute something like this classic code:

```
#!/usr/local/bin/perl -w
if ($#ARGV = 0) { $who = join(' ', @ARGV); }
else { $who = 'World'; }
print "Hello, $who!\n";
```

Here's How:

Let us assume that the above lines are stored in a Unix file `~/bin/hello`. (That's in your home directory, subdirectory bin, file hello.) You can then run the program by entering a command like:

```
perl ~/bin/hello
perl ~/bin/hello Citizens of Earth
perl hello          (If you're in the ~/bin directory.)
```

If you expect to use this program a lot and want to execute it as a command, then you need to do five things.

1. The first line of the program should after a "#!" specify the location of the perl command, typically as `#!/usr/local/bin/perl` or `#!/usr/bin/perl` , as illustrated in the preceding example program. This line can also give command options, like -w (warn of possible inconsistencies).

2. Set the execute permissions of the program file. To make the file executable (and readable and writable) by only yourself, use a Unix command like:

```
chmod 700 ~/bin/hello
```

To make it executable and readable by all enter a Unix command like the following:

```
chmod a+rx ~/bin/hello
```

You may also need to use chmod a+x on the directories ~ and ~/bin. See "man chmod" for details and the security implications.

3. Edit your file `~/.cshrc` or `~/.login` to make directory `~/bin` part of the path Unix searches for executables, with a line like this:

```
set path = ($path ~/bin)
```

4. This takes effect the next time you start a default tcsh or csh shell (.cshrc file) or login (.login file). If you want it to take effect immediately, enter the above set path command at the Unix prompt or enter execute your .cshrc or .login file with the "source" command. If you are using sh, bash, ksh or some other shell, alter ~/.profile or some other file to set the path at login.

If a program you want to execute has just been newly created, then issue the csh/tcsh command "rehash" to rescan the path.

If you perform (1)-(5), then you can execute your program via a command like this:

```
hello
```

7. The Elements

The rest of these notes will refer to the Perl 5 Reference Guide, highlighting and expanding on important points. So get your Reference Guide and turn to Section 2, *Literals.*

Perl's Three Data Structures

Scalars can be numeric or character as determined by context:

```
123  12.4  5E-10  0xff (hex)  0377 (octal)

'What you $see is (almost) what \n you get'   'Don\'t Walk'

"How are you?"  "Substitute values of $x and \n in \" quotes."

`date`   `uptime -u`   `du -sk $filespec | sort -n`

$x    $list_of_things[5]     $lookup{'key'}
```

Single-quotes '' allow no substitution except for \\ and \'. Double-quotes " " allow substitution of variables like $x and control codes like \n (newline). Back-quotes ` ` also allow substitution, then try to execute the result as a system command, returning as the final value whatever the system command outputs.

Arrays of scalars (also called **lists**) are sequentially-arranged scalars:

```
('Sunday', 'Monday', 'Tuesday', 'Wednesday',
    'Thursday', 'Friday', 'Saturday')

(13,14,15,16,17,18,19)   equivalent to  (13..19)

(13,14,15,16,17,18,19)[2..4]  equivalent to  (15,16,17)

@whole_list
```

Associative arrays (also called **hashes**) help you remember things:

```
$DaysInMonth{'January'} = 31;   $enrolled{'Joe College'} = 1;

$StudentName{654321} = 'Joe College';

$score{$studentno,$examno} = 89;

%whole_hash
```

Perl 5 allows combinations of these, such as lists of lists and associative arrays of lists.

Name Conventions

Scalar variables start with '$', even when referring to an array element. The variable name reference for a whole list starts with '@', and the variable name reference for a whole associative array starts with '%'.

Lists are indexed with square brackets enclosing a number, normally starting with [0]. In Perl 5, negative subscripts count from the end. Thus, `$things[5]` is the 6th element of array `@things`, and

```
('Sun','Mon','Tue','Wed','Thu','Fri','Sat')[1]
```

equals `'Mon'`.

Associative arrays are indexed with curly brackets enclosing a string. $whatever, @whatever, and %whatever are three different variables.

```
@days = (31,28,31,30,31,30,31,31,30,31,30,31);
               # A list with 12 elements.

$#days         # Last index of @days; 11 for above list

$#days = 7;    # shortens or lengthens list @days to 8 elements

@days          # ($days[0], $days[1],... )
```

```
@days[3,4,5]   # = (30,31,30)

@days{'a','c'} # same as ($days{'a'},$days{'c'})

%days          # (key1, value1, key2, value2, ...)
```

Case *is* significant--"$FOO", "$Foo" and "$foo" are all different variables. If a letter or underscore is the first character after the $, @, or %, the rest of the name may also contain digits and underscores. If this character is a digit, the rest must be digits. Perl has several dozen special variables whose second character is non-alphanumeric. For example, $/ is the input record separator, newline "\n" by default. An uninitialized variable has a special "undefined" value which can be detected by the function defined(). Undefined values convert depending on context to 0, null, or false.

The variable "**$_**" Perl presumes when needed variables are not specified. Thus:

```
<STDIN;          assigns a record from filehandle STDIN to $_
print;            prints the curent value of $_
chop;             removes the last character from $_
@things = split;  parses $_ into white-space delimited
                  words, which become successive
                  elements of list @things.
```

$_, $1, $2, $3, and other implicit variables contribute to **Perl Paradox Number Two:** *What you don't see can help you or hurt you.* See *Quick Reference Guide* Section 25, Special Variables.

Subroutines and **functions** are referenced with an initial '&', which is optional if reference is obviously a subroutine or function such as following the sub, do, and sort directives:

```
sub square { return $_[0] ** 2; }

print "5 squared is ", &square(5);
```

Filehandles don't start with a special character, and so as to not conflict with reserved words are most reliably specified as uppercase names: INPUT, OUTPUT, STDIN, STDOUT, STDERR, etc.

8. Literals and Operators

Example: Numbers and Characters

```
#!/usr/local/bin/perl
print '007,' has been portrayed by at least ', 004, ' actors. ';
print 7+3, ' ', 7*3, ' ', 7/3, ' ', 7%3, ' ', 7**3, ' ';
$x = 7;
print $x;
print '   Doesn\'t resolve variables like $x and backslashes \n. ';
print "Does resolve $x and backslash\n";
$y = "A line containing $x and ending with line feed.\n";
print $y;
$y = "Con" . "cat" . "enation!\n";
print $y;
```

This produces:

```
007 has been portrayed by at least 4 actors. 10 21
2.3333333333333335 1 343 7
Doesn't resolve variables like $x and backslashes \n. Does
resolve 7 and backslash
A line containing 7 and ending with line feed.
Concatenation!
```

Questions:

1. Why does the output from the first few print statements run together?
2. Is it necessary to declare variables in Perl? Is it possible?

Example: Comparisons

```
# The following "<<" variation of
# data input simplifies CGI forms.
$x = 'operator';
print <<THATSALL;
A common mistake:  Confusing the assignment $x =
and the numeric comparison $x ==, and the character
comparison $x eq.
THATSALL
$x = 7;
if ($x == 7) { print "x is $x\n"; }
if ($x = 5)  {
    print "x is now $x,",
    "the assignment is successful.\n";
}
$x = 'stuff';
if ($x eq 'stuff') {
    print "Use eq, ne, lt, gt, etc for strings.\n";
}
```

This produces:

```
A common mistake:  Confusing the assignment operator =
and the numeric comparison operator ==, and the character
comparison operator eq.
x is 7
x is now 5, the assignment is successful.
Use eq, ne, lt, gt, etc for strings.
```

Example: Ordinary Arrays

```
@stuff = ('This', 'is', 'a', 'list.');
print "Lists and strings are indexed from 0.\n";
print "So \$stuff[1] = $stuff[1], ",
     "and \$#stuff = $#stuff.\n";
print @stuff,"\n";
print join('...',@stuff),"\n";
splice(@stuff, 3, 0, ('fine', 'little'));
print join('...',@stuff),"\n";
```

This produces:

```
Lists and strings are indexed from 0.
So $stuff[1] = is, and $#stuff = 3.
Thisisalist.
This...is...a...list.
This...is...a...fine...little...list.
```

Homework: Validate a date.

```
#!/usr/local/bin/perl
print "Enter numeric:  month  day  year\n";
$_ = <STDIN;
($month,$day,$year) = split;
```

Complete this program. Print an error message if the month is not valid. Print an error message if the day is not valid for the given month (31 is ok for January but not for February). See if you can avoid using conditionals (if, unless, ?,...) statements but instead use data structures.

Approach this incrementally. On the first draft, assume that the user enters 3 numbers separated by spaces and that February has 28 days. Subsequent refinements should account for bad input and leap year. Finally, find a Perl builtin function that converts a date to system time, and see how to use that to validate time data generally.

Homework: Play with associative arrays.

Start with a few assignments like:

```
$name{12345} = 'John Doe';
$name{24680} = 'Jane Smith';
```

Print these scalars. What is the value of an associative array element that has never been assigned? What happens if you assign an associative array to a scalar? What happens if you assign an associative array to a normal array?

```
$blunk = %name;
@persons = %name;
print '$blunk=',$blunk,', @persons=',
    join(', ',@persons),"\n";
```

What happens if you assign a normal array to an associative array?

9. Loops and I/O

Example: Command Line Values and Iterative Loops

```
print "$#ARGV is the subscript of the ",
    "last command argument.\n";
# Iterate on numeric subscript 0 to $#ARGV:
for ($i=0; $i <= $#ARGV; $i++) {
    print "Argument $i is $ARGV[$i].\n";
}
# A variation on the preceding loop:
foreach $item (@ARGV) {
    print "The word is:  $item.\n";
}
# A similar variation, using the
# "Default Scalar Variable" $_ :
foreach (@ARGV) {
    print "Say:  $_.\n";
}
```

Demonstration:

```
 perl example5.pl Gooood morning, Columbia!
2 is the subscript of the last command argument.
Argument 0 is Gooood.
Argument 1 is morning,.
Argument 2 is Columbia!.
The word is:  Gooood.
The word is:  morning,.
The word is:  Columbia!.
Say:  Gooood.
Say:  morning,.
Say:  Columbia!.
```

Example: Standard I/O

```
print STDOUT "Tell me something: ";
while ($input = <STDIN) {
        print STDOUT "You said, quote: $input endquote\n";
        chop $input;
        print STDOUT "Without the newline:  $input endquote\n";
        if ($input eq '') { print STDERR "Null input!\n"; }
        print STDOUT "Tell me more, or ^D to end:\n";
}
print STDOUT "That's all!\n";
```

Note 1: The `while` statement's condition is an assignment statement: assign the next record from standard input to the variable $input. On end of file, this will assign not a null value but an "undefined" value. An undefined value in the context of a condition evaluates to "false". So the "`while ($input = <STDIN)`" does three things: gets a record, assigns it to $input, and tests whether $input is undefined. In other contexts, Perl treats an undefined variable as null or zero. Thus, if `$i` is not initialized, `$i++` sets `$i` to 1. Perl Paradox Number Three: *Side effects can yield an elegant face or a pain in the rear.*

Note 2: Data records are by default terminated by a newline character "\n" which in the above example is included as the last character of variable $input. The "chop" function removes the last character of its argument. Perl 5 introduces a "chomp" function that removes the last characters of a variable only if they are the currently defined end-of-record sequence, which is defined in the special variable $/.

Demonstration:

```
 perl example6.pl
Tell me something: I'm warm.
You said, quote: I'm warm.
 endquote
Without the newline:  I'm warm. endquote
```

```
Tell me more, or ^D to end:
Can I have some water?
You said, quote: Can I have some water?
 endquote
Without the newline:  Can I have some water? endquote
Tell me more, or ^D to end:

You said, quote:
 endquote
Without the newline:   endquote
Null input!
Tell me more, or ^D to end:
^D
That's all!
```

Example: Perls, A Perl Shell, Calculator, & Learning Tool

```
#!/usr/local/bin/perl
for (;;) {
    print '(',join(', ',@ReSuLt),') ?';
    last unless $InPuT = <STDIN;
    $? = ''; $@ = ''; $! = '';
    @ReSuLt = eval $InPuT;
    if ($?) { print 'status=',$?,' ' }
    if ($@) { print 'errmsg=',$@,' ' }
    if ($!) { print 'errno=',$!+0,': ',$!,' ' }
}
```

This reads a line from the terminal and executes it as a Perl program. The "for (;;) {...}" makes an endless loop. The "last unless" line might be equivalently specified:

```
$InPuT = <STDIN;               # Get line from standard input.
if (! defined($InPuT)) {last;}  # If no line, leave the loop.
```

The "eval" function in Perl evaluates a string as a Perl program. "$@" is the Perl error message from the last "eval" or "do".

Demonstration:

```
perls
() ?Howdy
(Howdy) ?2+5
(7) ?sqrt(2)
(1.4142135623730951) ?$x
() ?$x = sqrt(2)
(1.4142135623730951) ?$x + 5
(6.4142135623730949) ?1/0
errmsg=Illegal division by constant zero in file (eval) at line 2, next 2 tokens "0;"
() ?system 'date'
Fri Sep 27 10:02:43 CDT 1996
(0) ?$x = `date`
(Fri Sep 27 10:02:52 CDT 1996
) ?chop $x
(
) ?$x
(Fri Sep 27 10:02:52 CDT 1996) ?@y = split(/ /,$x)
(Fri, Sep, 27, 10:02:52, CDT, 1996) ?@y[1,2,5]
(Sep, 27, 1996) ?localtime()
(37, 4, 10, 27, 8, 96, 5, 270, 1) ?
() ?foreach (1..10) {print sqrt(),' '}
1 1.4142135623730951 1.7320508075688772 2 2.2360679774997898
2.4494897427831779 2.6457513110645907 2.8284271247461903 3 3.1622776601683795
() ?exit
```

Example: File I/O

```
#!/usr/local/bin/perl
# Function:  Reverse each line of a file

# 1: Get command line values:
if ($#ARGV !=1) {
    die "Usage:  $0 inputfile outputfile\n";
}
($infile,$outfile) = @ARGV;
if (! -r $infile) {
    die "Can't read input $infile\n";
}
if (! -f $infile) {
    die "Input $infile is not a plain file\n";
}
# 2: Validate files
# Or statements "||" short-circuit, so that if an early part
# evaluates as true, Perl doesn't bother to evaluate the rest.
# Here, if the file opens successfully, we don't abort:
open(INPUT,"<$infile") ||
    die "Can't input $infile $!";
if ( -e $outfile) {
    print STDERR "Output file $outfile exists!\n";
    until ($ans eq 'r' || $ans eq 'a' || $ans eq 'e' ) {
        print STDERR "replace, append, or exit? ";
        $ans = getc(STDIN);
    }
    if ($ans eq 'e') {exit}
}
if ($ans eq 'a') {$mode=''}
else {$mode=''}
open(OUTPUT,"$mode$outfile") ||
    die "Can't output $outfile $!";

# 3: Read input, reverse each line, output it.
while (<INPUT) {
    chop $_;
    $_ = reverse $_;
    print OUTPUT $_,"\n";
}

# 4: Done!
close INPUT,OUTPUT;
exit;
```

10. Data-Processing: Grade Book Example

This example produces a score summary report by combining data from a simple file of student info and a file of their scores.

Input file "stufile" is delimited with colons. Fields are Student ID, Name, Year:

```
123456:Washington,George:SR
246802:Lincoln,Abraham "Abe":SO
357913:Jefferson,Thomas:JR
212121:Roosevelt,Theodore "Teddy":SO
```

Input file "scorefile" is delimited with blanks. Fields are Student ID, Exam number, Score on exam. Note that Abe is missing exam 2:

```
123456 1 98
212121 1 86
246802 1 89
357913 1 90
123456 2 96
212121 2 88
357913 2 92
123456 3 97
212121 3 96
246802 3 95
357913 3 94
```

The desired report:

```
Stu-ID Name...                       1   2   3   Totals:

357913 Jefferson,Thomas             90  92  94       276
246802 Lincoln,Abraham "Abe"        89      95       184
212121 Roosevelt,Theodore "Teddy"   86  88  96       270
123456 Washington,George            98  96  97       291

                           Totals:  363 276 382
```

The program that made this report:

```
#!/usr/local/bin/perl
# Gradebook - demonstrates I/O, associative
# arrays, sorting, and report formatting.
# This accommodates any number of exams and students
# and missing data.  Input files are:
$stufile='stufile';
$scorefile='scorefile';

# If file opens successfully, this evaluates as "true", and Perl
# does not evaluate rest of the "or" "||"
open (NAMES,"<$stufile")
    || die "Can't open $stufile $!";
open (SCORES,"<$scorefile")
    || die "Can't open $scorefile $!";

# Build an associative array of student info
# keyed by student number

while (<NAMES) {
        ($stuid,$name,$year) = split(':',$_);
        $name{$stuid}=$name;
        if (length($name)$maxnamelength) {
                $maxnamelength=length($name);
        }
}
close NAMES;

# Build a table from the test scores:

while (<SCORES) {
   ($stuid,$examno,$score) = split;
   $score{$stuid,$examno} = $score;
   if ($examno  $maxexamno) {
      $maxexamno = $examno;
   }
}
close SCORES;

# Print the report from accumulated data!

printf "%6s %-${maxnamelength}s ",
        'Stu-ID','Name...';
foreach $examno (1..$maxexamno) {
        printf "%4d",$examno;
}
printf "%10s\n\n",'Totals:';

# Subroutine "byname" is used to sort the %name array.
# The "sort" function gives variables $a and $b to
# subroutines it calls.
# "x cmp y" function returns -1 if x<y, 0 if x=y,
# +1 if xy.  See the Perl documentation for details.
```

```
sub byname { $name{$a} cmp $name{$b} }

    # Order student IDs so the names appear alphabetically:
    foreach $stuid ( sort byname keys(%name) ) {
       # Print scores for a student, and a total:
       printf "%6d %-${maxnamelength}s ",
              $stuid,$name{$stuid};
       $total = 0;
          foreach $examno (1..$maxexamno) {
             printf "%4s",$score{$stuid,$examno};
             $total += $score{$stuid,$examno};
             $examtot{$examno} += $score{$stuid,$examno};
          }
          printf "%10d\n",$total;
    }

    printf "\n%6s %${maxnamelength}s ",'','Totals: ";
    foreach $examno (1..$maxexamno) {
       printf "%4d",$examtot{$examno};
    }
    print "\n";
    exit(0);
```

Perl allows an associative array to be "tied" to a genuine database, such that expressions like $record = $student{$key} use the database. See the "dbm" and "tie" functions.

11. Pipe I/O and System Calls

```
    #!/usr/local/bin/perl
    # Report on disk usage under specified files
    # The Unix command "du -sk ..." (on BSD Unix, "du -s ...")
    # produces a series of lines:
    #   1942    bin
    #   2981    etc
    #   ...
    # listing the K bytes used under each file or directory.
    # It doesn't show other information, such as the
    # modification date or owner.
    # This program gets du's kbytes and filename, and merges
    # this info with other useful information for each file.
    #
    $files = join(' ',@ARGV);
    # The trailing pipe "|" directs command output
    # into our program:
    if (! open (DUPIPE,"du -sk $files | sort -nr |"))  {
        die "Can't run du! $!\n";
    }
    printf "%8s %-8s %-16s %8s %s\n",
        'K-bytes','Login','Name','Modified','File';
    while (<DUPIPE) {
        # parse the du info:
        ($kbytes, $filename) = split;

        # Call system to look up file info like "ls" does:
        ($dev,$ino,$mode,$nlink,$uid,$gid,$rdev,
            $size,$atime,$mtime,$ctime)
            = stat($filename);

        # Call system to associate login & name with uid:
        if ($uid != $previous_uid) {
            ($login,$passwd,$uid,$gid,$quota,$comment,
                $realname,$dir,$shell) = getpwuid($uid);
            ($realname) = split(',',substr($realname,0,20));
            $previous_uid = $uid;
        }
        # Convert the modification-time to readable form:
        ($sec,$min,$hour,$mday,$mon,$myear) = localtime($mtime);
        $mmonth = $mon+1;
        printf "%8s %-8s %-16s %02s-%02d-%02d %s\n",
            $kbytes, $login, $realname,
            $myear, $mmonth, $mday, $filename;
    }
```

Demonstration Output

```
K-bytes Login    Name             Modified File
  40788 c527100  Fred Flintstone  95-10-05 c527100
  32685 c565060  Peter Parker     95-10-05 c565060
  24932 c579818  Clark Kent       95-10-06 c579818
  15388 c576657  Lois Lane        95-10-06 c576657
   9462 c572038  Bruce Wayne      95-10-06 c572038
   8381 c517401  Eric McGregor    95-10-05 c517401
   7022 c594912  Asterisk de Gaul 95-10-05 c594912
```

12. Matching

Matching involves use of patterns called "regular expressions". This, as you will see, leads to Perl Paradox Number Four: *Regular expressions aren't.* See sections 13 and 14 of the *Quick Reference.*

The =~ operator performs pattern matching and substitution. For example, if:

```
$s = 'One if by land and two if by sea';
```

then:

```
if ($s =~ /if by la/) {print "YES"}
else {print "NO"}
```

prints "YES", because the string $s matches the simple constant pattern "if by la".

```
if ($s =~ /one/) {print "YES"}
else {print "NO"}
```

prints "NO", because the string does not match the pattern. However, by adding the "i" option to ignore case, we would get a "YES" from the following:

```
if ($s =~ /one/i) {print "YES"}
else {print "NO"}
```

Patterns can contain a mind-boggling variety of special directions that facilitate very general matching. See *Perl Reference Guide* section 13, Regular Expressions. For example, a period matches any character (except the "newline" \n character).

```
if ($x =~ /l.mp/) {print "YES"}
```

would print "YES" for $x = "lamp", "lump", "slumped", but not for $x = "lmp" or "less amperes".

Parentheses () group pattern elements. An asterisk * means that the preceding character, element, or group of elements may occur zero times, one time, or many times. Similarly, a plus + means that the preceding element or group of elements must occur at least once. A question mark ? matches zero or one times. So:

```
/fr.*nd/  matches "frnd", "friend", "front and back"
/fr.+nd/  matches "frond", "friend", "front and back"
            but not "frnd".
/10*1/    matches "11", "101", "1001", "100000001".
/b(an)*a/ matches "ba", "bana", "banana", "bananananana"
/flo?at/  matches "flat" and "float"
            but not "flooat"
```

Square brackets [ ] match a class of single characters.

```
[0123456789] matches any single digit
[0-9]        matches any single digit
[0-9]+       matches any sequence of one or more digits
[a-z]+       matches any lowercase word
[A-Z]+       matches any uppercase word
[ab n]*      matches the null string "", "b",
                any number of blanks, "nab a banana"
```

[^...] matches characters that are *not* "...":

```
[^0-9]       matches any non-digit character.
```

Curly braces allow more precise specification of repeated fields. For example `[0-9]{6}` matches any sequence of 6 digits, and `[0-9]{6,10}` matches any sequence of 6 to 10 digits.

Patterns float, unless anchored. The caret ^ (outside [ ]) anchors a pattern to the beginning, and dollar-sign $ anchors a pattern

at the end, so:

```
/at/         matches "at", "attention", "flat", & "flatter"
/^at/        matches "at" & "attention" but not "flat"
/at$/        matches "at" & "flat", but not "attention"
/^at$/       matches "at" and nothing else.
/^at$/i      matches "at", "At", "aT", and "AT".
/^[ \t]*$/   matches a "blank line", one that contains nothing
                     or any combination of blanks and tabs.
```

The Backslash. Other characters simply match themselves, but the characters `+?.*^$()[]{}|\` and usually `/` must be escaped with a backslash `\` to be taken literally. Thus:

```
/10.2/       matches "10Q2", "1052", and "10.2"
/10\.2/      matches "10.2" but not "10Q2" or "1052"
/\*+/        matches one or more asterisks
/A:\\DIR/    matches "A:\DIR"
/\/usr\/bin/ matches "/usr/bin"
```

If a backslash preceeds an alphanumeric character, this sequence takes a special meaning, typically a short form of a [ ] character class. For example, \d is the same as the `[0-9]` digits character class.

```
/[-+]?\d*\.?\d*/      is the same as
/[-+]?[0-9]*\.?\d*/
```

Either of the above matches decimal numbers: "-150", "-4.13", "3.1415", "+0000.00", etc.

A simple `\s` specifies "white space", the same as the character class `[ \t\n\r\f]` (blank, tab, newline, carriage return,form-feed). A character may be specified in hexadecimal as a `\x` followed by two hexadecimal digits; \x1b is the ESC character.

A vertical bar | specifies "or".

```
if ($answer =~ /^y|^yes|^yeah/i ) {
     print "Affirmative!";
}
```

prints "Affirmative!" for $answer equal to "y" or "yes" or "yeah" (or "Y", "YeS", or "yessireebob, that's right").

13. Parsing

See the *Perl Reference Guide* section 14, Search and replace functions. When you include parenthesis ( ) in a matched string, the matching text in the parenthesis may subsequently be referenced via variables $1, $2, $3, ... for each left parenthesis encountered. These matches can also be assigned as sequential values of an array.

```
#!/usr/local/bin/perl -w
$s = 'There is 1 date 10/25/95 in here somewhere.';
print "\$s=$s\n";
$s =~ /(\d{1,2})\/(\d{1,2})\/(\d{2,4})/;
print "Trick 1: \$1=$1, \$2=$2, \$3=$3,\n",
      "         \$\'=",$',"  \$\'=",$',"\n";

($mo, $day, $year) =
    ( $s =~ /(\d{1,2})\/(\d{1,2})\/(\d{2,4})/ );
print "Trick 2: \$mo=$mo, \$day=$day, \$year=$year.\n";

($wholedate,$mo, $day, $year) =
    ( $s =~ /((\d{1,2})\/(\d{1,2})\/(\d{2,4}))/ );
print "Trick 3: \$wholedate=$wholedate, \$mo=$mo, ",
    "\$day=$day, \$year=$year.\n";
```

Results of above:

```
$s=There is 1 date 10/25/95 in here somewhere.
Trick 1: $1=10, $2=25, $3=95,
         $'=There is 1 date  $'= in here somewhere.
Trick 2: $mo=10, $day=25, $year=95.
Trick 3: $wholedate=10/25/95, $mo=10, $day=25, $year=95.
```

Note that when patterns are matched in an array context as in Tricks 2 and 3, $1, $2, ..., and $', $', and $& are not set.

Regular expressions are greedy. In the following example we try to match whatever is between "<" and "" :

```
#!/usr/local/bin/perl -w
$s = 'Beware of <STRONGgreedy</strong regular expressions.';
print "\$s=$s\n";
($m) = ( $s =~ /<(.*)/ );
print "Try 1: \$m=$m\n";
($m) = ( $s =~ /<([^]*)/ );
print "Try 2: \$m=$m\n";
```

This results in:

```
$s=Beware of <STRONGgreedy</strong regular expressions.
Try 1: $m=STRONGgreedy</strong
Try 2: $m=STRONG
```

Homework: Parsing and Reporting

1. See preceding "Grade Book" example. Using the same "stufile" input, print a list of students ordered by family name, with any quoted nickname listed in place of the given name, and family name last. Produce output like this:

```
Student-ID  Year  Name
357913      JR    Thomas Jefferson
246802      SO    Abe Lincoln
212121      SO    Teddy Roosevelt
123456      SR    George Washington
```

14. Simple CGI

For an introduction to Common Gateway Interface, see http://hoohoo.ncsa.uiuc.edu/cgi/ and http://www.ncsa.uiuc.edu/SDG/Software/Mosaic/Docs/fill-out-forms/overview.html .

Example: Server Status Report

Let's start with a CGI program that takes no input but produces output. The following Perl program reports the load on the Web server, using the standard Unix commands "hostname", "uptime", and "w". The output would look something like this:

What's Happening at sgi1

7:09pm up 1 day, 18:38, 4 users, load average: 0.08, 0.21, 0.53

```
c676828  q0  128.206.58.85       3  pico
c676828  ftp UNKNOWN@128.20         -
ccgreg   q1  monad.missouri         tcsh
c552997  q5  mizzou-ts2.mis         telnet
```

Here's the program that produced it:

```
#!/usr/local/bin/perl
# Send error messages to the user, not system log
open(STDERR,'<&STDOUT');  $| = 1;

# Headers terminate by a null line:
print "Content-type: text/html\n\n";
$host = 'hostname';
chop $host;
$uptime = 'uptime';
$w = 'w -s -h';
print <<BUNCHASTUFF;
<HTML<HEAD
<TITLE$host Status</TITLE
</HEAD<BODY
<H1What's Happening at $host</H1
$uptime
```

```
<PRE$w</PRE
<HR
</BODY</HTML
BUNCHASTUFF
exit;
```

Suppose user "bitman" wants to store this program on MU's SHOWME and SGI webservers, which use the "Apache" server software, configured for "CGI anywhere". If bitman doesn't already have a web directory, he should create one with these Unix commands:

```
mkdir ~/www; chmod a+x ~ ~/www
```

Then the above program could be put in a file `~/www/status.cgi`, and that file made readable and executable:

```
chmod a+rx ~/www/status.cgi
```

On the SHOWME server, the program would then be referenced as: `http://www.missouri.edu/~bitman/status.cgi`

Example: Web Form

Here is the image of a World Wide Web Electronic Form:

Web Form Example

(From http://some.webserver.missouri.edu/~user/myform.cgi)

Enter your ID Number: [ ]

Enter your Name: [ ]

Select favorite Color: [▼]

To submit the query, press this button: [Submit Request]

When something is entered and the submit button pressed, here is a resulting screen:

Results of Form

(From http://some.webserver.missouri.edu/cgi-bin/myform)

Your ID Number is 196965, your name is G. K. Johnson, and your favorite color is green.

[Try again]

Perl Program to Generate and Process Form

Here is the Perl program that generates both the form and the response. It uses an external module called "cgi-lib.pl" that is available from network Perl archives such as http://cgi-lib.stanford.edu or a local copy.

```
#!/usr/local/bin/perl
# Send error messages to the user, not system log
open(STDERR,'<&STDOUT');  $| = 1
print &PrintHeader;
require "cgi-lib.pl";  # Get external subroutines

$script = $ENV{'SCRIPT_NAME'};
$webserver = $ENV{'SERVER_NAME'};

if (! &ReadParse(*input)) { &showform }
else { &readentries }
exit;

sub showform {
# If there is no input data, show the blank form

print <<EOF;
<HTML<HEAD
<TITLEForm Example, Part 1</TITLE
</HEAD<BODY
<H1Web Form Example</H1
<P(From http://$webserver$script)
<FORM METHOD="POST"
    ACTION=$script
<PRE
Enter your ID Number:  <INPUT NAME=idnum
Enter your Name:       <INPUT NAME=name
Select favorite Color: <SELECT NAME=color
<OPTIONred<OPTIONgreen<OPTIONblue
</SELECT
</PRE
To submit the query, press this button:
<INPUT TYPE=submit VALUE="Submit Request"
</FORM
</BODY</HTML
EOF
} # End of sub showform #

sub readentries {
# Input data was detected.  Echo back to form user.

print <<EOF;
<HTML<HEAD
<TITLEForm Example, Part 2</TITLE
</HEAD<BODY
<H1Results of Form</H1
<P(From http://$webserver$script)
<PYour ID Number is $input{'idnum'}, your name is $input{'name'},
and your favorite color is $input{'color'}.
<HR
[<A HREF=$scriptTry again</A]
EOF
} # end of sub readentries #
```

15. Testing Perl Programs

Use the compiler `-w` switch to warn about identifiers that are referenced only once, uninitialized scalars, redefined subroutines, undefined file handles, probable confusion of "==" and "eq", and other things. This can be coded in the magic cookie first line:

```
#!/usr/local/bin/perl -w
```

As you write your program, put in `print` statements to display variables as you proceed. Comment "#" them out when you feel you don't need to see their output.

CGI scripts require some special attention in testing.

MU's "showme" and "SGI" Web servers (www.missouri.edu and www.cclabs.missouri.edu) use the Apache "sucgi" facility. This causes CGI programs stored under your directory ~/www/ with file name ending ".cgi" to execute as your own ID. On

some other Web server the script does not execute under your login ID! It executes under the ID of the Web server, typically as user "nobody".

Thus a script that works at the command line may fail under the Web server because: The path for executables or Perl library might be inappropriate. Print $ENV{'PATH'} and @INC to see if there's a difference. Fix it with Perl statements like:

```
$ENV{'PATH'} .= ':~myid/bin:~myid/cgibin';
push @INC,'~myid/lib/perl';
```

File permissions give your ID access don't give the Web server appropriate access.

16. Common Goofs for Novices

Adapted from *Programming Perl*, page 361.

1. Testing "all-at-once" instead of incrementally, either bottom-up or top-down.
2. Optimistically skipping `print` scaffolding to dump values and show progress.
3. Not running with the `perl -w` switch to catch obvious typographical errors.
4. Leaving off `$` or `@` or `%` from the front of a variable, or omitting & when invoking a subroutine in Perl 4.
5. Forgetting the trailing semicolon.
6. Forgetting curly braces around a block.
7. Unbalanced (), {}, [], "", ", ‘‘, and sometimes <.
8. Mixing " and "", or / and \.
9. Using == instead of `eq`, `!=` instead of `ne`, = instead of ==, etc. `('White' == 'Black')` and `($x = 5)` evaluate as `(0 == 0)` and `(5)` and thus are true!
10. Using "else if" instead of "elsif".
11. Not chopping the output of backquotes `‘date‘` or not chopping input:

```
print "Enter y to proceed: ";
$ans = <STDIN;
chop $ans;
if ($ans eq 'y') { print "You said y\n";}
else { print "You did not say 'y'\n";}
```

12. Putting a comma after the file handle in a print statement.
13. Forgetting that Perl array subscripts and string indexes normally start at 0, not 1.
14. Using $_, $1, or other side-effect variables, then modifying the code in a way that unknowingly affects or is affected by these.
15. Forgetting that regular expressions are greedy, seeking the longest match not the shortest match.

Special thanks to Hugo van der Sanden for corrections.
- Greg Johnson - JohnsonG@missouri.edu

Published by: MU Information & Access Technology Services - Webmaster <webmaster@www.missouri.edu>