Introduction to Perl

Texas A&M High Performance Research Computing (HPRC)

Keith Jackson
Acknowledgements

• Title page clip art taken from O’Reilly Publishers books *Programming Perl*.
• A few code examples taken from *Programming Perl* and *Beginning Perl for Bioinformatics*, as well as on-line references.
  (See hprc.tamu.edu)
• `perlconsole` was written by Alexis Sukrieh
  (See http://sukria.net/perlconsole.html)
Who Should Attend This Class?

HPRC users who need the power and simplicity of Perl to do:

- Text and pattern analysis
- File administration
- Database and network operations
- Quick, easy Unix tasks
Upcoming HPRC Short Courses

https://hprc.tamu.edu/training/

Thu, Oct 5  Viz Portal: Abaqus  **NEW**
Tue, Oct 10 Intro to MATLAB Parallel Toolbox
Wed, Oct 11 Intro to Next Generation Sequencing
Thu, Oct 12 Viz Portal: Paraview  **NEW**
Tue, Oct 17 Introduction to Code Parallelization using OpenMP
Suggested Prerequisites

• HPRC account
  (See https://hprc.tamu.edu/apply/)
• Intro to Unix shortcourse
  (https://hprc.tamu.edu/training/intro_unix.html)
• Experience with programming at least one language (C, C++, FORTRAN, Java, or similar)
Agenda

• What kind of language is Perl?
• Executing your program
• Finding documentation
• Statement syntax
• Variables, constants, expressions
• Control Flow
• Error messages
• I/O
What is Perl?

1. Interpreted, dynamic programming language
2. High-level language
   • Functional
   • Procedural
   • Object-oriented
3. Extensible library modules
What Perl Does Well

1. Pattern matching with regular expressions
2. String processing
3. Hash tables
4. File I/O
5. Network and database access
6. CGI (website)
Limitations of Perl

1. Compiled on each run
2. Large FP calculations not as fast or as easy as FORTRAN or C/C++
3. No contiguous multi-dimensional arrays. Complex data structures have memory management overhead
How to Run a Perl Program

Usually, program is in a file with “.pl” suffix. You can run it from the command line with the perl command:

```
$ perl sum.pl
```
Run Perl with \texttt{--e}

You can run short programs with \texttt{--e} option:

\begin{verbatim}
$ perl -e 'printf("%f\n", exp(10 * log(2)))'
\end{verbatim}

Be sure to put quotes around your statements so they are passed unchanged to \texttt{perl}.
Run with **eval perl**

You can use **eval perl** to enter statements without creating a file:

```
$ eval perl
$x = 3;
$y = 8;
printf("The sum is %d\n", $x + $y);
```

Press ctrl-d (^D) on a new line to complete input.
Testing with `eval perl`

Using `eval perl` allows you to quickly test Perl syntax without a program file.

```perl
$ eval perl
@a = ('red', 'green', 'blue');
print @a, "\n";
print @a . "\n";
print "@a\n";
```
Testing with **perlconsole**

**perlconsole** allows you to run Perl statements interactively for quick testing.

- Not a standard utility.

```
ada$ /scratch/training/Perl/bin/perlconsole
Perl Console 0.4
Perl> printf "sum is %d\n", 8 + 9;
```
Configuring perlconsole

• For informative output, create $HOME/.perlconsolederc:

$ echo ":set output=dumper" > ~/.perlconsolederc
On-Line Documentation

1. Unix man pages:
   - `$ man perl`
   - `$ man per1func`

2. Websites, such as:
   - [http://perldoc.perl.org](http://perldoc.perl.org)
Perl Books
The Perl Programming Language

http://www.perl.org
Variable Names

Names in Perl:
  • Start with a letter
  • Contain letters, numbers, and underscores “_”
  • Case sensitive

Two major types:
  $  Scalars (single value)
  @  Lists
Scalars

• Start with a dollar sign “$”
• Can be of type:
  • Integer
  • Floating point
  • String
  • Binary data
  • Reference (like a pointer)
• But Perl is not a strongly typed language
Lists

• Start with an at symbol “@”
• Also known as arrays
• Always one-dimensional
• Index starts at 0
• Can contain mixture of scalar types
  (not strongly typed)
Hash Tables

• Start with percent sign “%”
• Implemented as a list with special properties
• Key-Value pairs
• Keys are unique
• Keys can be any scalar value
• Values can be any scalar value
List Elements

• Individual array elements:
  • Scalar values
  • Start with “$”
  • Indexed in square brackets: “[ ]”

```
@a = (2, 3, 5, 7, 11);
print $a[3];
@a[5] = 13;
```

prints “7”
extends @a by one
List Size

- Assign array to scalar to get list length
- The top index is given by "#$"
Hash Elements

- Individual array elements:
  - Scalar values
  - Start with "$"
  - Indexed in curly brackets: "{ }"

$h = (name => "Sam", phone => "555-1212");
print $h{name};
$h{age} = 27;

prints "Sam"
extends %h by one
Same Name, Different Variables

• The same name can be reused for scalars, arrays, hashes, subroutine names, file handles, etc..
• Each of the following refer to a completely different thing:
Perl Statement Syntax

• Statements separated by semicolons: “;”
• Statement blocks surrounded by curly braces: “{ }”
• Comments preceded by pound sign: “#”
Sample Syntax

```perl
# see if we need to run work()
$remaining = queue_size();
if ($remaining > 0)
{
    work($x);       # does the main task
    print "did work\n";
}
```
if ($remaining > 0)
    work($x);

for ($i = 1; $i < $n; $i++)
    print a[$i], "\n";

while ($c = 1)
{
    $c = do_thing($m, $q);
}
Fixing Errors

if ($remaining > 0)
{
    work($x);
}

for ($i = 1; $i < $n; $i++)
{
    print $a[$i], "\n";
}

while ($c == 1)
{
    $c = do_thing($m, $q);
}
Perl Control Statements

• Conditionals:
  o \texttt{if/elsif/else}
  o \texttt{unless} (inverse of “if”)
  o Experimental statements \texttt{given/when} (like “\texttt{switch/case}”)

• Loops:

\begin{verbatim}
for(;;)   foreach ()   while()
until()   do()
\end{verbatim}
Common Perl Statements

• Assignment (use equal sign: “=”)
• Jumping
  • next/last/redo/goto
• Subroutine calls
  • Functional
  • Procedural
  • Object method
• Print statements
Assignments

• Use a single equal sign: “=”
• Put the value on the right into the place specified on the left

• Left-hand side of assignment called the “L-value”, most often a variable
Assignment Examples

```perl
$a = 2.75;          # scalar, floating point
$color = 'yellow';  # scalar, string

# array of four strings
@ary = ( "Perl", "C++", "Java", "FORTRAN" );

# hash, two keys (strings), numeric values
%ht = ( "AAPL" => 282.52, "MSFT" => 24.38 );
```
Assigning List Elements

@ary = ( "Perl", "C", "Java", "FORTRAN" );

$ary[1] = "C++";  # overwrite "C"

%ht = ( "AAPL" => 282.52, "MSFT" => 24.38 );

$ht{IBM} = 135.64;  # add 1 item to hash
Operator-Assignment

```php
$a += 10;          # add 10 to $a
$b *= 2.5;         # multiply $b by 2.5
$name .= ', Jr.';  # append to $name
$mode &amp;= 0722;  # apply bitwise mask to $mode
```
Scalar Values

Scalar expression can be numeric or string, made from any of:

<table>
<thead>
<tr>
<th>Variable</th>
<th>$a$</th>
<th>$ht{MSFT}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constant</td>
<td>2.6</td>
<td>'blue'</td>
</tr>
<tr>
<td>Function</td>
<td>sin($angle$)</td>
<td>length($name$)</td>
</tr>
<tr>
<td>Operators</td>
<td>$a/sin($ang)+ 2.6</td>
<td>$col \ &quot;_&quot; . $sz</td>
</tr>
</tbody>
</table>
Numeric Constants

10       # decimal integer
0722     # octal integer
0xF3E9   # hexadecimal integer
-2.532   # floating point
6.022E23 # floating point (scientific)
String Constants

'simple'       # single quotes

"one\ttwo"      # double quotes

<<"END_TEXT";  # here document (dbl quotes)
1\t15kg\t3:25
2\t9kg\t0:22
END_TEXT
List Forms

("one", "two", 3)  # list of mixed constants
qw(one two 3)     # same thing

# hash table
(
    "Mustang" => "Ford",
    "Civic"   => "Honda",
)
Operators

• Numeric operators are mostly the same as C/C++, also the “**” (exponent) operator
• Also has string and regular expression operators
• Documentation:
  0 [http://perldoc.perl.org/perlop.html](http://perldoc.perl.org/perlop.html)
  0 “man perlop”
Numeric Operators

$x + 3 \quad -4.3 / z \quad 2 \ ** \ 10$

$i++ \quad -j \quad f \ % \ mod$
Bitwise Operators

$mode \ll 8$  $t \mid 0x3F$

$v \wedge mask$  $\neg q$
Comparison Operators

# numeric
$x == 3      $a >= -4.3       $m != $n

# stringwise
$y eq "AT"    $title lt 'm'    $q ne 'B'
Sort Comparison Operators

-1  Left is less than right
0   Left equals right
+1  Left is greater than right

# numeric  stringwise
$x <=> y$  $s$ cmp $t$
Logical Operators

# C-style
$ready && ($y > 2) !$done $e || $r

# lower precedence
$ready and $y > 2 not $done $e or $r

# ternary conditional
($d != 0) ? ($n / $d) : "Inf"
Comparison Operators

# numeric
$x == 3       $a >= -4.3       $m != $n

# stringwise
$y eq "AT"    $title lt 'm'    $q ne 'B'
# List separators
2, 4, 6    "R" => 255, "G" => 0, "B" => 127

# string concatenation
"First " . $item

# string repetition
"AB" x 10

# range operator
1..10      0..$#ary
# choose the size of x
if ($x > 5) {
    print "big x\n";
} elsif ($x > 3) {
    print "medium x\n";
} else {
    print "small x\n";
}
Conditional After Statement

```perl
print "f is even\n" if ($f % 2 == 0);

print "not capitalized\n" unless ($name =~ /^[A-Z]/);
```

- Special feature of Perl
- Avoids need for braces
- Can be confusing and no “else” branch
Logical Operator as Conditional

\[(y \neq 0) \land \# \text{ C-style}\]
\[\text{$ratio = \frac{x}{y}$;}\]

\[\text{-f myfile) or} \# \text{ word style}\]
\[\text{die "File ``$myfile'' does not exist!";}\]

- Avoids need for braces
- Can be confusing and no “else”
- Word style better rather than C-style
While Loops

```perl
$x = 1;  # initialize
while ($x != 7) {  # test
    print "x = $x\n";
    $x += 2;
    last if ($x > 12);  # escape clause
}
```
for ($x = 1; $x != 7; $x += 2) {
    print "x = $x\n";
    last if ($x > 12); # escape clause
}

For Loops
Foreach Loops

```perl
foreach $index (0..$#myarray) {
    $item = $myarray[$index];
    printf "cost of $item is $price{$item}\n",
          $item, $price{$item};
    print "TOO MUCH!\n"
    if ($price{$item} > 1200);
}
```
Errors and Warnings

• Warning is “non-fatal”, can still keep going
• Error can be at:
  o Compile time, e.g., syntax errors
  o Run time:
    • Numeric, e.g., division by zero
    • Reference type, e.g., hash vs. List
    • Object method
Warnings

Using -w option turns on warning messages

$ perl -w bounds.pl
Use of uninitialized value in addition (+) at bounds.pl line 8.
b = 3
Put "use warnings" pragma at top to turn on warning messages.

```perl
use warnings;
my @a = (1, 2);
my $b = $a[0] + $a[1] + $a[2];
print "b = $b\n";
```
my @a = (1, 2);
my $b = $a[0] + $a[1] + $a[2];

$ ./bounds.pl
Use of uninitialized value in addition (+) at ./bounds.pl
line 8.
"should be =="

```perl
print "c is big\n" if ($c = 100);
```

```
$ ./twoeq.pl
Found = in conditional, should be == at ./twoeq.pl line 8.
```
while a < 10 {

$ ./synerr.pl
syntax error at ./synerr.pl line 4, near "while a "
syntax error at ./synerr.pl line 8, near "}" Execution of ./synerr.pl aborted due to compilation errors.
Runaway Strings

$ ./closeq.pl
Scalar found where operator expected at ./closeq.pl line 8, near "print "$a"
   (Might be a runaway multi-line "" string starting on line 3)
   (Do you need to predeclare print?)
Backslash found where operator expected at ./closeq.pl line 8, near "$a\"
   (Missing operator before \\?)
String found where operator expected at ./closeq.pl line 8, at end of line
   (Missing semicolon on previous line?)
syntax error at ./closeq.pl line 8, near "print "$a"
Can't find string terminator '"' anywhere before EOF at ./closeq.pl line 8.
Checking for Errors

Do your own error checking, to get diagnostics:

```perl
die "denominator zero " if ($d == 0);
$r = $n / $d;
```

- The “die” and “warn” functions output to stderr and can show line number
- The “Carp” module is more detailed
System Error String

If a system call fails, look at "$!" variable

```perl
open FH, $myfile, "r" or die "open $myfile: $! ";

$ ./nofile.pl
open nofile: No such file or directory at ./nofile.pl line 4.
```
Perl Debugger

You can use **perl -d** to debug your program:

```
$ perl -d debugme.pl
Loading DB routines from perl5db.pl version 1.28

Enter h or `h h' for help, or `man perldebug' for more help.

main::(debugme.pl:7): my @a = qw(TAGC CGTA ATTT GGCA);
DB<1>
```
Variable Scope

• Using the “my” declaration makes a variable local to the statement block or file.
• Don’t use “local” declaration unless you understand it—the “my” declaration is almost always what you want.
• The “our” declaration is used for declaring global variables within packages (modules).
Examples of Local Variables

- Surround multiple declared variables with parentheses.

```perl
my $a;
my @f;
my $x = "initial value";
my ($i, $j, $k);
foreach my $item (@ilist) {
    $sum += $item;
}
```
Example of Scope

```perl
my @numlist = (3, 4, 5);

foreach my $item (@numlist) {
    print "item = $item\n";
}
print "item = $item\n";
```

- item = 3
- item = 4
- item = 5
- item =
Strict Pragma

Put "use strict" pragma at top to require use of "my" declarations.

```perl
use strict;

my $x = 15;
$y = 19;

print "y = $y\n";
```
File Handles

- **STDIN, STDOUT, and STDERR** correspond to the **stdin**, **stdout**, and **stderr** of C/C++.
- A simple Perl filehandle is a name by itself, typically all caps.
- Filehandles can be scalar variables, too.
- Objects from **IO::File** and similar library modules have advantages.
Printing

- **print**
  - Prints a list of strings

- **printf**
  - C-style formatting

- **syswrite**
  - Low-level `write(2)` system call
  - Unbuffered and unformatted.
File Handle in Printing

- `print FH LIST`
- `printf FH FORMAT, LIST`
- `syswrite FH, DATA, ...`

For `print` and `printf` there is no comma separating file handle from arguments. Without a file handle, the output goes to `STDOUT`, by default.
Print Examples

```perl
print "Hello, world!\n";
print STDOUT "Hello, world!\n"; # same

print STDERR "File not found:\", $fname, "\n";

printf MYRPT "\%d items processed\n", $count;
printf MYRPT ("\%d items processed\n", $count); # same
```
Reading

• `< >`
  • Input operator for buffered input.
  • Uses `STDIN`, by default.

• `< FH >`
  • Input from filehandle `FH`.

• `sysread`
  • Low-level `read(2)` system call
  • Unbuffered and unformatted.
print "Enter name:";
$name = <>;

@listing = <STDIN>; # read all lines

# one line at a time
while ($line = <$myinfo>) {
    myprocess($line);
}
Opening a File

• The **open** function opens a file for reading, writing, appending, or more.
• Can specify a bareword file handle name or a scalar variable.

```perl
open(MYINFO, "<info.dat") or
die("open info.dat: $! ");

open $fh, ">logfile" or die $!;
```
# File Mode

<table>
<thead>
<tr>
<th>Mode</th>
<th>Symbol</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read</td>
<td><code>&lt;</code></td>
<td><code>open(MYINFO, &quot;&lt;info.dat&quot;) or</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>die(&quot;open info.dat: $! &quot;);</code></td>
</tr>
<tr>
<td>Create/truncate</td>
<td><code>&gt;</code></td>
<td><code>open $fh, $fname, &quot;&gt;&gt;&quot; or die $!;</code></td>
</tr>
<tr>
<td>Append</td>
<td><code>&gt;&gt;</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
open RAW, $rfile, "<";
open $ofh, ">>results";

while ($line = <RAW>) {
    @useful = myprocess($line);
    printf $ofh "%d,%6.2f,%s\n", @useful;
}
Command Substitution

• A shorthand way to run a command an capture the output is with the backwards single quotes, or "qx":

```bash
$hostname = `/bin/hostname`
$hostname = qx{/bin/hostname};
```

Same thing