Introduction to Unix & Perl Programming 1

Bioinformatics: Issues and Algorithms
CSE 308-408 • Fall 2007 • Lecture 4
Administrative notes

- Homework #1 has been posted on Blackboard and is due on Tuesday, Sept. 11 at 5:00 pm. Submit your work online using the Blackboard Assignment function.

_CSE Department Distinguished Seminar Series_

**Topic:** “Architecture of Product Lines”
**Speaker:** Dr. David M. Weiss, Avaya Laboratories
**Location:** Packard Lab 466
**Date:** Thurs., Sept. 6, 4:00 pm – 5:00 pm
**Reception @ 3:30 pm in Packard Lobby**

_CSE Department Ice Cream Social (yum!)
Location:** Packard Lab 360
**Date:** Tues., Sept. 11, 4:10 pm – 5:00 pm
Programming environment

For CSE 308-408, you may develop the programs for your programming assignments using Perl on any platform. Most Linux installations generally include Perl (see discussion on page 17 of your BBP book).

In particular, you can use machines in the CSE Dept. Sun Lab. These are accessible remotely from all over campus (and beyond) using standard secure shell (\textit{ssh}) client software. See: \url{http://www.lehigh.edu/helpdesk/ssh32/ssh32.shtml}

Wherever you develop your code, however, at the end of the day, you must make sure it runs on our systems.

In addition, I will provide specific naming conventions to facilitate the testing of your programs.
Secure shell (ssh) client on the Windows desktop
Remote login procedure for the CSE Dept. Sun Lab

Initiate a secure shell (ssh) session to sunlab.cse.lehigh.edu, which provides a connection to a generally random machine among all of the Suns.

Since all of the Suns have unique ssh keys, you may find that, after your first connection, your client complains about changed keys on the destination machine. You can either ignore this complaint, or go directly to one of the Suns using its One True Name (as displayed in its login prompt). The full list of Suns is displayed in the login motd on the machine named gateway.

Our sys admins have confirmed that all students registered for CSE 308-408 have active accounts. If you have forgotten your password, send email to help@cse.lehigh.edu.
Logging in

You have new mail.

Sun Microsystems Inc.  SunOS 5.6  Generic Patch  October 2001

-------------

You have new mail.

proteus+4

Connected to sunlab.cse.lehigh.edu

SSP-4 - am120-hz - hazz-mud5 - name

00:24
### Basic Unix commands you should know

<table>
<thead>
<tr>
<th>Category</th>
<th>Commands</th>
</tr>
</thead>
<tbody>
<tr>
<td>Files</td>
<td><code>ls</code>, <code>more</code>, <code>cp</code>, <code>mv</code>, <code>rm</code>, <code>tail</code>, <code>cat</code></td>
</tr>
<tr>
<td>Directories</td>
<td><code>pwd</code>, <code>cd</code>, <code>mkdir</code>, <code>rmdir</code></td>
</tr>
<tr>
<td>Processes</td>
<td><code>ps</code>, <code>kill</code></td>
</tr>
<tr>
<td>Running Perl</td>
<td><code>perl</code> (or <code>/usr/bin/perl</code>)</td>
</tr>
<tr>
<td>Other</td>
<td><code>w</code>, <code>finger</code>, <code>time</code>, <code>spell</code></td>
</tr>
</tbody>
</table>

But the **single most important** command is `man`. Typing:

```
% man command
```

will display a manual page for any Unix `command` (or function).

Close behind in importance is `apropos`. Typing:

```
% apropos keyword
```

will list all Unix commands that mention `keyword`. 
Typing “man man” at the Unix shell prompt

User Commands

man (1)

NAME

man - find and display reference manual pages

SYNOPSIS

man [ - ] [ -adFlrt ] [ -M path ] [ -T macro-package ] [ -s section ] name ...
man [ -M path ] -k keyword ...
man [ -M path ] -f file ...

DESCRIPTION

The man command displays information from the reference manuals. It displays complete manual pages that you select by name, or one-line summaries selected either by keyword (-k), or by the name of an associated file (-f). If no manual page is located, man prints an error message.
User Commands

    apropos(1)

NAME

    apropos - locate commands by keyword lookup

SYNOPSIS

    apropos keyword ...

DESCRIPTION

    The apropos utility displays the man page name, section number, and a short description for each man page whose NAME line contains keyword. ... Each word is considered separately and the case of letters is ignored. Words which are part of other words are considered; for example, when looking for `compile', apropos finds all instances of `compiler' also.
## Basic UNIX file commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ls</code></td>
<td>list contents of current directory</td>
</tr>
<tr>
<td><code>ls -l</code></td>
<td>generate long listing with more details</td>
</tr>
<tr>
<td><code>more file</code></td>
<td>page through file one screen at a time</td>
</tr>
<tr>
<td><code>cp file1 file2</code></td>
<td>copy file1 to file2 (i.e., duplicate)</td>
</tr>
<tr>
<td><code>mv file1 file2</code></td>
<td>move file1 to file2 (i.e., rename)</td>
</tr>
<tr>
<td><code>rm file</code></td>
<td>remove file (i.e., delete, erase)</td>
</tr>
<tr>
<td><code>tail file</code></td>
<td>display last lines in file</td>
</tr>
<tr>
<td><code>cat file</code></td>
<td>dump file to display</td>
</tr>
<tr>
<td><code>cat file1 file2 &gt; file3</code></td>
<td>concatenate file1 and file2 into file3</td>
</tr>
</tbody>
</table>
Basic UNIX directory commands

\textit{pwd} \quad \text{print working (current) directory}

\textit{cd \ dir} \quad \text{change directory (down one level) to \textit{dir}}
\textit{cd ..} \quad \text{change directory up one level}

\textit{mkdir \ dir} \quad \text{create a new directory named \textit{dir}}

\textit{rmdir \ dir} \quad \text{remove directory \textit{dir} (must be empty first)}
Basic UNIX process commands

- `ps` prints information about active processes
- `ps -f` generates full listing with more details
- `ps -a` prints information about all of your processes
- `ps -A` prints information about all processes
- `ps -A | more` pages through processes one screen at a time

**Unix pipe**

- `kill PID` kills active process with process ID `PID`
- `kill -9 PID` kills active process `PID` “with extreme prejudice”
Other basic UNIX commands

- `w` list who is currently using machine
- `finger userID` print details on a specific user
- `time command` report time used to execute `command`
- `spell file` run Unix spell checker on `file`

Wait ... aren't we missing one that's kind of important?

- `perl file` run Perl interpreter on source code `file`
Let's get started writing code ...

You'll need a way to input and edit your program. Possible choices include \textit{vi} and \textit{emacs}. These have some advantages because you can edit directly in a terminal window.

The following online tutorials seem reasonable:

- http://www.eng.hawaii.edu/Tutor/vi.html
- http://www2.lib.uchicago.edu/~keith/tcl-course/emacs-tutorial.html
Let's get started writing code ...

A strategy to follow (but you'll develop your own preferences):

One terminal window open for editing code ...  

... another terminal window open for running code.
Let's get started writing code ...

You can also use an application like *NotePad* or *WordPad* running on a PC for editing, but you'll have to upload your Perl code to a Unix machine to run it.

*There are implementations of Perl available for Windows PC's, but remember that ultimately your code will have to run on the CSE Dept. Unix system.*
print "Welcome to the Wonderful World of Bioinformatics!\n";

“\n” character sequence tells Perl to display a new line

What will this do?
Let's save it in a file named welcome and run it ...

umbria:~/CSE308/Chapter3% perl welcome
Welcome to the Wonderful World of Bioinformatics!
umbria:~/CSE308/Chapter3%
A less-simple Perl program

```perl
print "Welcome ";
print "to ";
print "the ";
print "Wonderful ";
print "World ";
print "of ";
print "Bioinformatics!\n";
```

Let's save it in a file named `welcome2` and run it ...

```
umbria:~/CSE308/Chapter3% perl welcome2
Welcome to the Wonderful World of Bioinformatics!
umbria:~/CSE308/Chapter3%
```

Exactly same output as before!
Simpler is better, however.
Checking Perl syntax

Everyone makes mistakes from time-to-time. Perl has a way of checking whether or not your program contains certain simple mistakes called “syntax errors”:

```perl
print "Welcome to the Wonderful World of Bioinformatics!\n";
```

umbria:~/CSE308/Chapter3% perl -c welcome
welcome syntax OK
umbria:~/CSE308/Chapter3%

“-c” stands for “check”

Note: correct syntax does not mean no bugs!
Checking Perl syntax

Now consider:

```perl
pint "Welcome to the Wonderful World of Bioinformatics!\n";
```

umbria:~/CSE308/Chapter3% perl -c welcome
String found where operator expected at welcome line 1, near "pint "Welcome to the Wonderful World of Bioinformatics!\n"
(Do you need to predeclare pint?)
syntax error at welcome line 1, near "pint "Welcome to the Wonderful World of Bioinformatics!\n"
welcome had compilation errors.

Useful information, once you learn to decipher it!
More detailed source code checking

Consider program **whoops**:

```
print "Welcome to the Wonderful World of Bioinformatics!\n";
```

```
umbria:/CSE308/Chapter3% perl whoops
Whoops indeed – no output!
```

```
umbria:/CSE308/Chapter3% perl -c whoops
whoops syntax OK
```

```
umbria:/CSE308/Chapter3% perl -c -w whoops
Useless use of a constant in void context at whoops line 1.
```

```
umbria:/CSE308/Chapter3% perl -c -w whoops
```

Cryptic, but better! 
“-w” asks for warnings.
More detailed source code checking

Consider program *whoops*:

```
print ; "Welcome to the Wonderful World of Bioinformatics!\n";
```

Really important note: even if your program passes both of these tests, it might still have bugs!

It is considered a fundamental impossibility for static testing – no matter how rigorous – to catch all bugs.

Even so, it's a good idea to employ the "-c" and "-w" options to check your code.
Making code self-executing

Recall:

```
umbria:~/CSE308/Chapter3% perl welcome
Welcome to the Wonderful World of Bioinformatics!
umbria:~/CSE308/Chapter3%
```

Nicer if we could just type program name – call this `welcome3`:

```
#! /usr/bin/perl -w
print "Welcome to the Wonderful World of Bioinformatics!\n";
```

```
umbria:~/CSE308/Chapter3% chmod u+x welcome3
umbria:~/CSE308/Chapter3% welcome3
Welcome to the Wonderful World of Bioinformatics!
umbria:~/CSE308/Chapter3%
```
Iteration is a basic programming construct. We repeat the same sequence of statements over and over until a pre-specified condition is either satisfied or violated.

while ( some condition is true )
{
    do something
}

Presumably something happens here to make condition false eventually
**Iteration** is a basic programming construct. We repeat the same sequence of statements over and over until a pre-specified condition is either satisfied or violated.

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

# The 'forever' program - a (Perl) program,
# which does not stop until someone presses Ctrl-C.

use constant TRUE       => 1;
use constant FALSE      => 0;

while ( TRUE ) {
    print "Welcome to the Wonderful World of Bioinformatics!\n";
    sleep 1;
}
```

- Any line starting with "#" is a comment
- Constants provide useful English names
- Executes so long as condition is true
- Program “sleeps” here for 1 second
Getting caught in a loop

Save previous program as \textit{forever}, then:

\begin{verbatim}
umbria:~/CSE308/Chapter3% chmod u+x forever
umbria:~/CSE308/Chapter3% forever
Welcome to the Wonderful World of Bioinformatics!
Welcome to the Wonderful World of Bioinformatics!
Welcome to the Wonderful World of Bioinformatics!
Welcome to the Wonderful World of Bioinformatics!
Welcome to the Wonderful World of Bioinformatics!
Welcome to the Wonderful World of Bioinformatics!
Welcome to the Wonderful World of Bioinformatics!
Welcome to the Wonderful World of Bioinformatics!
^C
umbria:~/CSE308/Chapter3%
\end{verbatim}

Type “Ctrl-C” (^C) to kill process and end infinite loop
Variables are “containers” whose contents (values) can change over lifetime of a program.

In Perl, simplest variable is a scalar, which holds single value.

```perl
#! /usr/bin/perl -w
# The 'tentimes' program - a (Perl) program, # which stops after ten iterations.
use constant HOWMANY => 10;
$count = 0;
while ( $count < HOWMANY )
{
    print "Welcome to the Wonderful World of Bioinformatics!\n";
    $count++;
}
```

Variable $count set equal to 0

False when $count equals HOWMANY

Variable $count incremented
A well-behaved loop

Save previous program as `tentimes`, then:

```
umbria:~/.CSE308/Chapter3% chmod u+x tentimes
umbria:~/.CSE308/Chapter3% tentimes
Welcome to the Wonderful World of Bioinformatics!
Welcome to the Wonderful World of Bioinformatics!
Welcome to the Wonderful World of Bioinformatics!
Welcome to the Wonderful World of Bioinformatics!
Welcome to the Wonderful World of Bioinformatics!
Welcome to the Wonderful World of Bioinformatics!
Welcome to the Wonderful World of Bioinformatics!
Welcome to the Wonderful World of Bioinformatics!
umbria:~/.CSE308/Chapter3%
```
Selection is another basic programming construct that allows a program to take one path or another depending on a condition.

```plaintext
if ( some condition is true )
{
    do something
}
else
{
    do something else
}
```

The "else" clause is optional.
A loop with selection

#!/usr/bin/perl -w

# The 'fivetimes' program - a (Perl) program, # which stops after five iterations.

use constant TRUE       => 1;
use constant FALSE      => 0;
use constant HOWMANY    => 5;

$count = 0;

while ( TRUE )
{
    $count++;
    print "Welcome to the Wonderful World of Bioinformatics!\n";
    if ( $count == HOWMANY )
    {
        last;
    }
}

Test for equality: note "==" and not "="!

Terminates current loop immediately
Save previous program as *fivetimes*, then:

```
umbria:~/CSE308/Chapter3% chmod u+x fivetimes
umbria:~/CSE308/Chapter3% fivetimes
Welcome to the Wonderful World of Bioinformatics!
Welcome to the Wonderful World of Bioinformatics!
Welcome to the Wonderful World of Bioinformatics!
Welcome to the Wonderful World of Bioinformatics!
umbria:~/CSE308/Chapter3%
```

Which style of loop is better, *tentimes* or *fivetimes*?

Your BBP textbook offers the helpful maxim:

*There's more than one way to do it.*
Checking parity of an integer ('odd' vs. 'even')

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

use constant HOWMANY => 4;
$count = 0;

while ( $count < HOWMANY )
{
    $count++;
    if ( $count == 1 )
    {
        print "odd\n";
    }
    elsif ( $count == 2 )
    {
        print "even\n";
    }
    elsif ( $count == 3 )
    {
        print "odd\n";
    }
    else    # at this point $count is four.
    {
        print "even\n";
    }
}
```

While this works, it seems kind of awkward and not very general. (What happens if we want to do this for a large number of values?)
More than one way to do it ...

While it's true “there's more than one way to do it,” some ways are definitely better than other ways. Case in point:

```perl
#!/usr/bin/perl -w
# The 'terrible' program - a poorly formatted 'oddeven'.
use constant HOWMANY => 4; $count = 0;
while ( $count < HOWMANY ) { $count++;
if ( $count == 1) { print "odd\n"; } elsif ( $count == 2 )
{ print "even\n"; } elsif ( $count == 3 ) { print "odd\n"; }
else  # at this point $count is four.
{ print "even\n"; } }
```

This program works, but it is very hard to understand.

```
metis:~:/CSE308/Chapter3% terrible
odd
even
odd
even
metis:~:/CSE308/Chapter3%
```

Moral: make your programs easy to read by using whitespace and good formatting style.
Both *oddeven* and *terrible* reflect poor style. This is better:

```perl
#!/usr/bin/perl -w
use constant HOWMANY => 4;
$count = 0;
while ( $count < HOWMANY ) {
    $count++;
    if ( $count % 2 == 0 ) {
        print "even\n";
    } else  # count % 2 is not zero.
    {   # count % 2 is not zero.
        print "odd\n";
    }
}
```

Moral: in general, aim for generality. :-)

Modulo arithmetic operator (%) returns remainder of division. When divisor is 2, remainder is 0 if integer is even and 1 if odd.
A (slightly) better parity checking program ...

Even better, perhaps, because it is more succinct:

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

# The 'oddeven3' program - yet another version of 'oddeven'.
use constant HOWMANY => 4;
$count = 0;
while ( $count < HOWMANY )
{
    $count++;
    print "even\n" if ( $count % 2 == 0 );
    print "odd\n" if ( $count % 2 != 0 );
}
```

This form of conditional statement is rarely found in other languages, but is very handy! (Perl calls it a “statement qualifier.”)
We can't do much unless we can get data into our programs.

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

# The 'getlines' program which processes lines.

while ( $line = <> )
{
    print $line;
}
```

What I typed on keyboard

- Hello, Lehigh!
- Hello, Lehigh!
- Are we having fun yet?
- Are we having fun yet?
- Hey, stop doing that!!!
- Hey, stop doing that!!!

Type “Ctrl-D” (^D) for “end-of-file”

Output from program
Taking input from a file

Under Unix, input can easily come from file or from keyboard:

```perl
#! /usr/bin/perl -w
while ( ( $line = <> ) ) {
    print $line;
}
```

Directs `getlines` to take its input from the file `terrible`

```
metis:~/CSE308/Chapter3% getlines terrible
#! /usr/bin/perl -w
# The 'terrible' program - a poorly formatted 'oddeven'.
use constant HOWMANY => 4; $count = 0;
while ( $count < HOWMANY ) { $count++;
if ( $count == 1) { print "odd\n"; } elsif ( $count == 2 ) { print "even\n"; } elsif ( $count == 3 ) { print "odd\n"; }
else    # at this point $count is four.
{ print "even\n"; } }
metis:~/CSE308/Chapter3%
```
As noted earlier, pattern matching is a very important feature of Perl for bioinformatics applications. We'll consider this in more detail later, but for now, let's look at a simple example.

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

# The 'patterns' program - introducing regular expressions.

while ( $line = <> )
{
  print $line if $line =~ /even/;
}
```

“Print the contents of the scalar variable $line if and only if it contains the pattern 'even'”

“=~” is known as the “binding operator” in Perl
Pattern matching

```perl
#! /usr/bin/perl -w
while ( $line = <> )
{
    print $line if $line =~ /even/;
}
```

Note: patterns tell perl what to look for, not how to do it.

```
metis:~/CSE308/Chapter3% patterns terrible
# The 'terrible' program - a poorly formatted 'oddeven'.
{ print "even\n"; } elsif ( $count == 3 ) { print "odd\n"; }
{ print "even\n"; }
metis:~/CSE308/Chapter3%
```

Only lines containing the string “even” get printed.

```
metis:~/CSE308/Chapter3% patterns oddeven
# The 'oddeven' program - a (Perl) program,
# is an odd number, and 'even' when $count is an even
    print "even\n";
    print "even\n";
metis:~/CSE308/Chapter3%
```
Maxims from BBP Chapter 3

Your book provides short summaries of key points ("maxims"):

- Programs execute in sequential order.
- Less is better (in general).
- There's more than one way to do it.
- Add comments to facilitate future code maintenance.
- Use human-friendly names for constants and variables.
- A condition results in a "true" or "false" value.
- For readability, use plenty of whitespace and good formatting.
- Patterns tell perl what to look for, not how to do it.
Readings for next time:
• BB&P Chapters 4-5 (more Perl programming).

Remember:
• Come to class having done the readings.
• Check Blackboard regularly for updates.
• If enrolled in CSE 408, let me know which lecture topic you wish to scribe by Friday, Sept. 7. (Send me several choices.)