Table of Contents

Introduction

About this course

This manual was designed with the goal of assisting instructors in their efforts of teaching students to be able to create Perl programs.

Typographical syntax

Examples in this text of commands will appear in **bold** text and the output of the commands will appear in *italic* text. The commands and the output of the commands will be placed in a box to separate them from other text. Example:

```
[student@linux1 student]$ pwd
/home/student
```

Note: "[student@linux1 student]$" is a **prompt**, a method the shell uses to say "I'm ready for a new command".

Bold text within a sentence will indicate an important term or a command. Files and directories are highlighted by being placed in `courier` font.

Using this manual while in class

In many ways, class manuals are different from textbooks. Textbooks are often filled with lengthy paragraphs that explain a topic in detail. Unfortunately, this style doesn't work well in a classroom environment.

Class manuals often are much more concise than textbooks. Its difficult to follow the instructor's example and read lengthy paragraphs in a book at the same time. For this purpose, class manuals are often more terse.

Lab Exercises

The lab exercises provided in this class are intended to provide practical, hands on experience with a programming Perl. Students are strongly encouraged to perform the labs provided at the end of each module to reinforce the knowledge provided in class.

Floppy contents

The floppy disk that accompanies this course contains the following:

- All of the examples provided in the manual
- All of the answers to the labs provided in the manual

Unit One
Advanced Regular Expressions

Module topics:

Notes:

1.1 Review: Basic Regular Expressions

Basic Regular Expressions are discussed in the Beginning Perl class and are only mentioned here for a brief review. Depending on the students' experience level, the instructor may or may not cover this section.

1.1.1 Basic operations

The following are the basic operations:

op	Meaning
m	Pattern Matching
s	Substituting
tr	Translating

Examples of basic operations:

```
DB<1> $line = "Today is a good day to learn Perl"
DB<2> if ($line =~ m/good/) {print "yes"}
yes
DB<3> $line =~ s/good/great/
DB<4> print $line
Today is a great day to learn Perl
DB<5> $line =~ tr/a-z/A-Z/
DB<6> print $line
TODAY IS A GREAT DAY TO LEARN PERL
```

Notes:

Notes about the basic operators:

- Since matching is the most common operation, the "m" can be dropped in most cases:

```
DB<2> if ($line =~ /good/) {print "yes"}
```

- If you perform matching, substitution, or translation on the default variable ($_), you can drop the "$var =~" portion of the command:

```
DB<1> $_ = "Today is a good day to learn Perl"
DB<2> if (/good/) {print "yes"}
yes
DB<3> s/good/great/
DB<4> print $_
Today is a great day to learn Perl
DB<5> tr/a-z/A-Z/
DB<6> print $_
TODAY IS A GREAT DAY TO LEARN PERL
```

- The "y" operator is the same as the "tr" operator:

```
DB<4> print $line
Today is a great day to learn Perl
DB<5> $line =~ y/a-z/A-Z/
DB<6> print $line
TODAY IS A GREAT DAY TO LEARN PERL
```

Notes:

1.1.2 Basic modifiers

The following basic modifiers were discussed in the Beginning Perl class:

Mod	Meaning
g	Global match or substitution
i	Case Insensitive match

"g" modifier example:

```
DB<1> $_="The dog ate the dog food"
DB<2> s/dog/cat/
DB<3> print
The cat ate the dog food
DB<4> $_="The dog ate the dog food"
DB<5> s/dog/cat/g
DB<6> print
The cat ate the cat food
```

"i" modifier example:

```
DB<1> $_="This is a good day to learn Perl"
DB<2> if (/perl/) {print "yes"}
DB<3> if (/perl/i) {print "yes"}
yes
```

Notes:

1.1.3 Regular Expressions Classes

Class	Matches
\w	Alphanumeric and underscore character
\d	Numeric
\s	White space (space, tab, newline, formfeed, return)
\b	Word boundary (includes "white space", end/beginning of line, punctuation, etc.)
\W	Non-alphanumeric and underscore character
\D	Non-numeric characters
\S	Non-white space
\B	Non-word boundary

Examples:

"\w" and "\d" example:

```
DB<1> $_="The code is A127Z"
DB<2> s/\d\d\d/---/
DB<3> print
The code is A---Z
DB<4> s/\w---\w/ZZZZZ/
DB<5> print
The code is ZZZZZ
```

Notes:

"\s" and "\b" example:

```
DB<1> $_="This is fun"
DB<2> s/\sis\s/was/
DB<3> print
Thiswasfun
DB<4> $_="This is fun"
DB<5> s/\bis\b/was/
DB<6> print
This was fun
DB<7> $_="This is"
DB<8> if (/\sis\s/) {print "yes"}
DB<9> if (/\bis\b/) {print "yes"}
yes
```

Notes:

1.1.4 Basic metacharacters

Char	**Meaning**
*	Represents the previous character repeated zero or more times
+	Represents the previous character repeated one or more times
{x,y}	Represents the previous character repeated *x* to *y* times
.	Represents exactly one character (any one character)
[ ]	Represents any single character listed within the bracket.
?	Represents an optional character. The char. prior to the "?" is optional.
^	Represents the beginning of the line when it is the first character in the RE
$	Represents the end of the line when it is the last character in the RE
( )	Used to group an expression.
|	Represents an "or" operator
\	Used to "escape" the special meaning of the above characters.

"*" Examples:

Example	**Meaning**
abc*	"ab" followed by zero or more c's
c*enter	zero of more c's followed by "enter"
a*	Anything. Warning: This expression ALWAYS will find a match and will most likely match "nothing". Look for examples of this later in this Module.

Notes:

"+" Examples:

Example	**Meaning**
abc+	"ab" followed by at least one (or more) c
c+enter	At least one c (or more) followed by "enter"
a+	Match one or more "a"

"{ }" Examples:

Example	**Meaning**
abc{3,5}	"ab" followed by three to five "c's"
abc{3,}	"ab" followed by three or more "c's"
abc{3}	"ab" followed by exactly three "c's"

"." Examples:

Example	**Meaning**
a.c	An "a" followed by any single character followed by a "c"
abc.	A "abc" followed by any single character
ab.*	A "ab" followed by zero or more of any character

Notes:

"[ ]" Examples:

Example	**Meaning**
[abc]xyz	Either an a, b or c followed by "xyz"
[bca]xyz	Same as previous
[a-c]xyz	Same as previous
[c-a]xyz	An improper range
[a-z]xyz	Any lower case character followed by "xyz"
[A-Z]xyz	Any upper case character followed by "xyz"
[A-z]xyz	Any lower case or upper case character **or** any of these characters: "[ ] ^ _ ' " followed by "xyz"
[A-Za-z]xyz	Any upper case or lower case character followed by "xyz"
[A-Z][a-z]	A upper case character followed by a lower case character
gr[ae]y	Either "gray" or "grey"
[^A-Z]xyz	Any non-upper case character followed by "xyz"
[abc^]xyz	First character is either "a", "b", "c" or "^" followed by "xyz"

"?" Examples:

Example	**Meaning**
abc?	Either "ab" or "abc"
colou?r	Either "color" or "colour"

Notes:

"^" and "$" Examples:

Example	Meaning
^abc	"abc" found at the beginning of the line
abc$	"abc" found at the end of the line
^abc$	A line that just contains "abc"
^$	A blank line
^\^[^^]*$	A line that starts with a "^" and has no other "^" characters on it

"( )" Examples:

Example	Meaning
(abc)*xyz	"abc" zero or more times followed by xyz
(abc)+xyz	"abc" one or more times followed by xyz
^(abc)+$	A line that contains one or more groups of "abc"

"|" Examples:

Example	Meaning
a|bxyz	Either an “a” or "bxyz"
(ab)|(xyz)	Either "ab" or "xyz"

"\" Examples:

Example	Meaning
abc\$	The string "abc$"
\^*\.\$	The string "^*.$"

Notes:

1.1.5 Backreferencing

Grouping can also be used to "back reference" patterns that have been matched. When Perl makes a match of characters within parentheses, what was matched can be referred back to:

```
$var =~ s/^(…)abc\1/;
```

The \1 means "match what was matched in the first group". A \2 means "match what was matched in the second group".

In addition to being able to back reference within the regular expression, Perl assigns what was matched within the grouping to special variables. The first group match is assigned to $1, the second group matched is assigned to $2, etc.

```
$var =~ m/(abc..)/;
print $1;
```

The above will match the string "abc" followed by the next two characters and assign all five characters to the string $1.

Note: Future successful matches will cause these variables ($1, $2, etc.) to be overwritten.

Notes:

Example #1:

The variables $1, $2, etc. can be used immediately after a successful pattern match. In this example, the user enters their first and last name. Then pattern matching is used to extract the first and last name and print them out in a different format (last name, first name):

```
#!/usr/local/bin/perl
#1_back1.pl

print "Please enter your first and last name";
$_=<STDIN>;

if (m/(.*) (.*)/)  #ex: "Bob Smith"
{
   print "$2, $1\n";
}
```

Notes:

Example #2:

In this example, the UNIX file /etc/group will be read into the script one line at a time and “parsed”. Each line contains four fields of data that are separated by colons. This script will add the third field of each line and print the total:

```
#!/usr/local/bin/perl
#1_back2.pl

open (GROUP, "</etc/group");

while (<GROUP>) {
   m/(.*):(.*):(.*):(.*)/;
   $total += $3;
}

print "Total: $total\n";
```

Notes:

Example #3

When you need to refer back to what was matched within the pattern itself, you need to use \1, \2, etc. instead of $1, $2, etc.:

```
#!/usr/local/bin/perl
#1_back3.pl

print "Please enter a line: ";
$_=<STDIN>;
chomp $_;

if (/^(...).*\1$/) {print "$1\n";}

$junk="whatever";

if ($junk =~ /what/) {print "yes\n";}

print "$1\n";
```

Also note that when another pattern match is attempted, Perl will overwrite $1, $2, etc. even if you don't use parentheses.

Notes:

<u>Module #1 Mini Lab</u>

Write a program that takes a valid date and converts into this format:

January 01, 2001

The format of the valid date should be "01/01/2001". The first number should be between 01 and 12. The second number should be between 01 and 31. The last number should be a four-digit number.

Don't worry about "errors" such as "02/31/2001".

If an incorrect date is given, display an error message and ask for the input again.

Notes:

1.2 Modifiers

In addition to the **g** and **i** modifiers discussed in the Beginning Perl class, there are other modifiers that change the behavior of a regular expression match. Modifiers for matching and substitution are different than the modifiers for translation.

1.2.1 Matching and substitution modifiers:

Mod	**Details found**	**Meaning**
e	Section 1.2.3	Right-hand side of substitution is the code to evaluate
ee	N/A	Right-hand side of substitution is a string to evaluate and run as code. After completion, the return value is to be evaluated.
g	Section 1.1.2 & 1.3	Global match or substitution
gc	N/A	Doesn't reset the search position after a failed match
i	Section 1.1.2	Case Insensitive match
m	Section 1.9	Allows ^ and $ to match embed \n characters
o	N/A	Only compile the pattern once
s	Section 1.9	Allows the "." metacharacter to match newlines
x	Section 1.10	Ignores white space in pattern and allows comments

Notes:

1.2.2 Translation modifiers:

Mod	**Details found**	**Meaning**
c	N/A	Complement the search list
d	Section 1.2.4	Delete characters that are not replaced
s	Section 1.2.5	Delete replaced characters that are duplicates

1.2.3 The e modifier

When the **e** modifier is used, the right-hand (replacement) side of the substitution is evaluated as a Perl statement. The result of the statement is used as the replacement value:

```
DB<1> $var="123456789"
DB<2> $code="ABCDEFGHIJ"
DB<3> $code =~ s/J/chop $var/e
DB<4> print $code
ABCDEFGHI9
```

Note: The **e** modifier can only be used for substitution, not matching.

Notes:

1.2.4 The d modifier

Normally when you have too many characters on the left side of a translation operation, you get "weird" results:

```
DB<1> $var = "This can become very odd"
DB<2> $var =~ tr/abcdefghij/ABC/
DB<3> print $var
TCCs CAn BCComC vCry oCC
```

In the above example, the **tr** operator replaced "a" with "A", "b" with "B" and all of the other characters ("c-j") with "C".

The **d** modifier means, "if something is matched and we don't specify what to replace it with, then remove it":

```
DB<1> $var="Lets cap this and remove all numbers: 1234567890"
DB<2> $var =~  tr/[a-z][0-9]/[A-Z]/d
DB<3> print $var
LETS CAP THIS AND REMOVE ALL NUMBERS:
```

Notes:

1.2.5 The s modifier

When the **s** modifier is used with the **tr** operator. It tells **tr** to delete duplicated characters that are replaced:

```
DB<1> $var="Exxtra chars are removed"
DB<2> $var =~ tr/xyz/XYZ/s
DB<3> print $var
EXtra chars are removed
```

Note: There is also a **s** modifier for matching and substitution that works differently than the **s** modifier for translation

1.2.6 Other modifiers

Not all of the modifiers listed on the preceding page are discussed in detail in this course. The **g** and **i** modifiers were covered in the Beginning Perl class and are reviewed in section 1.1. Other modifiers will be introduced in future sections.

Notes:

1.3 Getting the Nth occurrence of a match

In some cases you will want to find the Nth occurrence of a match. In these cases, use pattern matching with the **g** modifier in a **while** loop:

```
#!/usr/local/bin/perl
#1_nth.pl

$line="Code: A127Z Code: B999E Code: G678T Code: T765J";

while ($line =~ /(Code: [A-Z][0-9]{3}[A-Z])/g) {
   $count++;
   print "The $count match is $1\n";
}
```

Notes:

1.4 Greedy vs. Non-Greedy matches

By default, Perl patterns are "greedy". This means that when matching a pattern, Perl will attempt to "grab" as many characters that will possibly match:

```
DB<1> $line="It was the best of times; it was the worst of times"
DB<2> $line =~ s/the.*times/a very bad year/
DB<3> print $line
It was a very bad year
```

The ".*" matched the string "the best of times; it was the worst of " because that was the most it could possibly match. To make your patterns Non-Greedy (match the minimal amount), use the "?" after the metacharacter:

```
DB<1> $line="It was the best of times; it was the worst of times"
DB<2> $line =~ s/the.*?times/a very bad year/
DB<2> print $line
It was a very bad year; it was the worst of times
```

You can use the following Non-Greedy patterns:

*?	{n}?
+?	{n,}?
??	{n,m}?

Notes:

1.5 Regular Expression variables

There are many variables that are set as the result of a pattern match:

Variable	Meaning
$`	String preceding what was last matched
$'	String following what was last matched
$+	Last parens match of last pattern match
$&	Last pattern match
$1..$9	Subpattern matches of last pattern match

1.5.1 What was matched

You can "look back" to what was matched during the last pattern match by looking at the **$&** variable:

```
#!/usr/local/bin/perl
#1_match1.pl

print "Enter a line of text and I will find the first 1 digit number: ";
$line=<STDIN>;

$line =~ m/[0-9]/;

print "The number was $&\n";
```

Notes:

1.5.2 Before and after what was matched

You can see what was in the string before and after the match by looking at the **$`** and **$'** variables:

```
#!/usr/local/bin/perl
#1_match2.pl

print "Enter a line of text and I will find the first 1 digit number: ";
$line=<STDIN>;

$line =~ m/[0-9]/;

print "The number was $&\n";
print "Before that number was $`\n";
print "After that number was $'\n";
```

Notes:

1.5.3 The last paren match

If you use the or ("|") operation with backreferencing, it's difficult to determine what holds the actual match:

```
#!/usr/local/bin/perl
#1_last1.pl

$_="Code: B999Z";

m/Code: (A127Z)|(B999Z)/;

print "$1 $2  \n";
```

In the above example, $2 is set while $1 is not. A better variable to use in this case is $+ which holds the last paren match:

```
#!/usr/local/bin/perl
#1_last2.pl

$_="Code: B999Z";

m/Code: (A127Z)|(B999Z)/;

print "last match is $+ \n";
```

Notes:

1.6 Special characters in regular expressions

In addition to the classes mentioned previously, there are other special characters allowed within regular expressions:

Spec. char	**Meaning**
\077	Octal character
\a	Bell character
\c	Control character
\E	End case change
\e	Escape character
\f	Form feed character
\l	Makes the next character lower case
\L	Makes following characters lower case until \E
\n	Newline Character
\Q	Disable metacharacters until \E
\r	Return character
\t	Tab character
\u	Makes the next character upper case
\U	Makes following characters upper case until \E
\x1	Match hex character

Notes:

1.7 Assertions

Some assertions (such as the **^** and **$** characters) have already been introduced. Assertions are used to match certain conditions within a string (such as beginning and end of a line):

Assertion	**Meaning**
^	Match beginning of line
$	Match end of line
\b	Match a word boundary
\B	Match a non-word boundary
\A	Match only at the beginning of the string (note: This is the same as ^ except when using the **m** modifier)
\Z	Match only at the end of the string or before a newline character at end of the string (note: This is the same as $ except when using the **m** modifier)
\z	Match only at the end of the string
\G	Match only where previous m//g left off (this works only with matching, not substitution or translation)
(?=EXPR)	Look ahead match (positive)
(?!EXPR)	Look ahead match (negative)
(?<=EXPR)	Look behind match (positive)
(?<!EXPR)	Look behind match (negative)

Notes:

1.7.1 Looking forward and back

The "look forward" and "look back" assertions are useful when you want to be certain that a pattern is found but you only want to "work with" a portion of the pattern. For example, you want to replace the word "great" with "bad", but only if it isn't the last word in the string. The following will allow this to occur:

```
DB<1> $_="This is a good time to learn Perl"
DB<2> s/good(?=.)/great/
DB<3> print
This is a great time to learn Perl
DB<4> $_="This is good"
DB<5> s/good(?=.)/great/
DB<6> print
This is good
```

Notes:

Or, suppose we want to replace "A127Z" with "-----" if the string "Code: " does not appear at the prior to "A127Z":

```
DB<1>  $_="Code: A127Z"
DB<2> s/(?<!Code: )A127Z/---/
DB<3> print
Code: A127Z
DB<4> $_="Answer: A127Z"
DB<5> s/(?<!Code: )A127Z/---/
DB<6>  print
Answer: ---
```

Notes:

1.8 Reading from filehandles using split

In the Beginning Perl class the **split** command was introduced. It was used in that class to break up a string using regular expressions and store the resulting items into an array:

```
DB<1> $line="Bob:Sue:Steve:Nick:Trevor"
DB<2> @names=split(/:/, $line)
DB<3> print $names[0]
Bob
```

The **split** command can also be used to read from a filehandle:

```
#!/usr/local/bin/perl
#1_split.pl

undef $/;  #undefine the input separator variable
@words=split (/\s+/, <STDIN>);

print "First word: $words[0]\n";
print "Last word: $words[$#words]\n";
print "Number of words ", $#words+1, "\n";
```

Notes:

1.9 Multiple line matching

In cases in which a string contains multiple lines (text separated with newline characters), the behavior of Perl's RE may not be what you want. The default behavior of Perl is to "ignore" new line characters when it comes to matching the end of a string:

```
DB<1> $_="Today is the day\n"
DB<2> if (/day$/) {print "yes"}
yes
```

You can look for a newline character if you want to:

```
DB<1> $_="Today is the day\n"
DB<2> if (/day\n$/) {print "yes"}
yes
```

Notes:

But what if you want to look for something that appears at the "end of a line"? The following will only look for something at the "end of the string":

```
DB<1> $_="This is a good day\nto learn Perl"
DB<2> if (/Perl$/) {print "yes"}
yes
DB<3> if (/day$/) {print "yes"}
```

You could say "match something followed by a newline character", but that won't match the last line the string unless there is a newline character:

```
DB<1> $_="This is a good day\nto learn Perl"
DB<2> if (/Perl\n/) {print "yes"}
DB<3> if (/day\n/) {print "yes"}
yes
```

To match the end of a line or the end of the string, use the **m** modifier:

```
DB<1> $_="This is a good day\nto learn Perl"
DB<2> if (/Perl$/m) {print "yes"}
yes
DB<3> if (/day$/m) {print "yes"}
yes
```

The meaning of "**$**" changes with the **m** modifier. Instead of meaning "end of the string" it means "end of the string or prior to a newline character".

Notes:

1.9.1 Using the s modifier

Another method that you can use is the **s** modifier. With this modifier, Perl treats newlines just like normal characters. This means that the "." metacharacter will match a newline character:

```
DB<1> $_="This is a good day\nto learn Perl"
DB<2> if (/day.to/) {print "yes"}
DB<3> if (/day.to/s) {print "yes"}
yes
```

Notes:

1.10 Commenting Regular Expressions

While you can place comments before and after your regular expressions, sometimes it would be nice to place comments within your regular expressions to help explain what the expression does. With the **x** modifier you can place comments and whitespace within your regular expressions.

When the **x** modifier is used, comments (# to end of line) and whitespace (Tabs, spaces, newlines, etc.) are completely ignored. This means if you want to "look for" one of these characters, you need to escape them with a backslash.

An example of commenting within a pattern:

```
#!/usr/local/bin/perl
#1_comm.pl

$_='Code: 127 -- \State=99\ ?UNSET?';

m/
  (?<=Code:)       #Look back for "Code:" but don't replace
  (\ \d{3})        #match and group " " followed by three numbers
  \ --\            #match " -- "
  \\State=         #match "\State="
  (\d+)            #match and group one or more digits
/x;

print "First number: $1\n";
print "Second number: $2\n";
```

Notes:

1.11 Alternative Delimiters

Consider the following code:

```
#!/usr/local/bin/perl
#1_alt1.pl

$_="Path: /usr/local/bin/perl";

m/\/([a-z]+)\/([a-z]+)/;

print "$1\n$2\n";
```

While it works just fine, the RE can be difficult to read. The problem is that in order to match a "/" you need to escape it.

Notes:

While most programmers use "/" by default as a delimiter, you can choose any character you wish. If you use a different character, then you don't have to escape the "/" character:

```
#!/usr/local/bin/perl
#1_alt2.pl

$_="Path: /usr/local/bin/perl";

m,/([a-z]+)/([a-z]+),;

print "$1\n$2\n";
```

Note: Be careful of what character you choose for the alternative delimiter. Avoid using metacharacters as you won't be able to use that character as a metacharacter within the RE.

1.12 Additional Resources

Books

Mastering Regular Expressions
by Jeffrey Friedl
O'Reilly & Associates
ISBN: 0596002890

Perl Cookbook
Chapter #6
by Tom Christiansen, Nathan Torkington
O'Reilly & Associates
ISBN: 0596003137

On line

http://search.cpan.org/dist/perl/pod/perlre.pod

Note: On line documentation may change.

1.13 Lab Exercises - Suggested lab time: 45 minutes

Throughout this class you will be creating one script. The script will take the output of the UNIX command 'ps -fe' and parse the data. In this module you will do the following (call the file parse1.pl):

When your script begins, open a file handle to read the output of the 'ps -fe' command. Read the data, perform regular expression substitution listed below and assign this data to an array:

- Remove all leading white space in each element
- Compress all multiple spaces into a single space

Create a main menu that has the following options (we will add more options as the class progresses:

1. Remove newline characters from each element
2. Convert dates into 01/31 format
3. Remove PPID field
4. Print the array
5. Exit

Write the code for each of these options.

Notes and hints:

- If you don't remember how to open up a file handle that reads the output of a UNIX command, ask the instructor.
- When printing the array, consider sending the data to the UNIX command **more**
- If the user runs option #3 more than one time, nothing should happen after the first time.
- Use subroutines to logically break up your program.

Unit Five
Pragmas

Module topics:

Notes:

5.1 Perl pragmas

The intent behind pragmas is to modify the behavior of your Perl script. Pragmas are invoked with the **use** statement:

```
use strict;
```

To disable the use of a pragma, use the **no** statement:

```
no strict;
```

Note: Some pragmas cannot be turned off.

The purpose of this section is to review some of the useful pragmas. Some pragmas will not be discussed either because they are covered in another class or because they are beyond the scope of this class.

Notes:

5.2 Pragma chart

The following chart lists some of the pragmas available in Perl (note: different versions of Perl have different pragmas available):

Pragma	**Details found**	**Meaning**
autouse	N/A	Delays the operation of a **require** statement until one of the specified subroutines is called.
blib	N/A	Modifies the **@INC** variable at compile time to use MakeMaker.
constant	Adv. Perl	Defined constants during compile time.
diagnostics	Section 7.4	Issues verbose error messages.
integer	N/A	Performs integer arithmetic instead of double.
less	N/A	This pragma is currently unimplemented
lib	Section 6.1	Modifies the **@INC** variable at compile time.
locale	N/A	States to either use or ignore the current locale for builtin operations.
ops	N/A	Restricts opcodes
overload	OOP Perl	Overloads the basic Perl operations
re	N/A	Modifies the default behavior of regular expressions
sigtrap	N/A	Allows you to handle signals
strict	Section 5.3	Prevents unwise statements
subs	Section 5.4	Allows you to predeclare subroutines
vmsish	N/A	Only used on VMS systems. Allows for VMS-style operations.
vars	Section 5.5	Allows you to predeclare global variables.

Notes:

5.3 The use strict pragma

There are three things you can tell Perl to be strict about: reference usage, subroutine usage and variable usage:

5.3.1 use strict 'ref'

This will cause your program to exit if a symbolic reference is used. Symbolic references are a method of referring to variable and are typically discussed in advanced classes.

5.3.2 use strict 'subs'

Creates an error message for "barewords" (strings without quotes around them that appear to be subrouting calls) that don't call a valid subroutine. Example:

```
#!/usr/bin/perl
#5_subs.pl

use strict 'subs';
sub hello {
print "hello\n";
}

hello;     #Calls a valid subroutine, no problem
justatest; #Bareword that isn't a subroutine.
```

Notes:

5.3.3 use strict 'vars'

This pragma will generate an error if a variable is used that:

- has not been declared as a **my** variable or
- isn't a fully qualified variable name or
- has not been declared as an **our** variable or
- has not been declared with a **use vars** statement

A fully qualified variable is one that includes its package namespace in the variable name. Packages are discussed in detail in another class. The following is just a brief introduction to using fully qualified variable names from the Main part of your script:

While it is sometimes useful to have global variables, **use strict vars** doesn’t allow this. If you want to use or modify variables from the "main" part (AKA "main" package) of your program, use the following syntax:

```
$main::var
```

See example on following pages.

Notes:

The following program is an example of using global variables (Perl's default behavior):

```
#!/usr/local/bin/perl
#5_usevars1.pl

use strict 'vars';

sub test {
   print "$total\n";
}

$total=100;
&test;
```

Notes:

In this example, we are implementing **use strict vars**, which would cause compile errors if we didn't use fully qualified variable names:

```
#!/usr/local/bin/perl
#5_usevars2.pl

use strict 'vars';

sub test {
   print "$main::total\n";
}

$main::total=100;
&test;
```

Notes regarding **use strict**:

- The statement "**use strict**" will enforce all restrictions (refs, subs, and vars).
- Perl built-in variables are not affected by "use strict vars".

Notes:

5.4 Predeclaring subroutines

Typically you need to create a subroutine prior to using it. For examples, the following code won't produce any output since the subroutine isn't declared until after it is called:

```
#!/usr/local/bin/perl
#5_sub1.pl

hello;

sub hello {
  print "hi there\n";
}
```

This can cause problems, especially if you are using "use strict":

```
#!/usr/local/bin/perl
#5_sub2.pl

use strict subs;

hello;

sub hello {
  print "hi there\n";
}
```

Notes:

Using "**use subs**" you can "predefine" subroutines:

```
#!/usr/local/bin/perl
#5_sub3.pl

use subs qw(hello);
use strict subs;

hello;

sub hello {
   print "hi there\n";
}
```

Notes:

- Once invoked, you cannot use "**no subs**" to undo a "**use subs**" statement.
- If you use the ampersand character before the function name, you do not have to pre-declare subroutines that are placed after they are called.
- The **use subs qw(hello)** statement is the same as invoking **sub hello {}**

Notes:

5.5 Predeclaring global variables

When you refer to an undeclared variable, Perl either returns a 0 or a null string:

```
#!/usr/local/bin/perl
#5_var1.pl

print "The total is $total\n";
print "The result is ", $total+5, "\n";
```

However, if you invoke "**use strict vars**", you will receive an error message:

```
#!/usr/local/bin/perl
#5_var2.pl

use strict vars;

print "The total is $total\n";
print "The result is ", $total+5, "\n";
```

Notes:

To be able to use a variable prior to having it set, you can use the statement "**use vars**":

```
#!/usr/local/bin/perl
#5_var3.pl

use strict vars;
use vars qw($total);

print "The total is $total\n";
print "The result is ", $total+5, "\n";
```

Notes:

This is very useful when you are not sure if a variable is set or not. When you use "**strict vars**" even a **defined** statement will fail:

```
#!/usr/local/bin/perl
#5_var4.pl

use strict vars;

if (defined ($total))           #This will result in an error
  {print "hey, it's here!\n";}
else
  {print "It's not there!\n";}
```

When you use "**use vars**", the **defined** statement will not fail:

```
#!/usr/local/bin/perl
#5_var5.pl

use strict vars;
use vars qw($total);

if (defined ($total))
  {print "hey, it's here!\n";}
else
  {print "It's not there!\n";}
```

Note: once invoked, you cannot use "**no vars**" to undo a "**use vars**" statement.

Notes:

use vars is obsolete

As of Perl 5.6, **use vars** is considered to be obsolete. It is covered in this course for the following reasons:

- #1. You may wish to write code that is backward compatiable to older versions of Perl. If so, you may want to continue to use the **use vars** statement.
- #2. While **use vars** is considered to be obsolete, it still performs the same function that is always has. As a result, you will still see it being used in other programmer's code as well as in older scripts.

Instead of using **use vars**, you should use the **our** statement to "globally declare" a variable. Much like **use vars**, specifying the **our** statement will allow you to use a variable without its fully qualified name while your code has **use strict** implemented:

```
#!/usr/local/bin/perl
#5_var6.pl

use strict vars;
our $total;

if (defined ($total))
  {print "hey, it's here!\n";}
else
  {print "It's not there!\n";}
```

The **our** statement is covered in greated detail in the Advanced Perl class.

5.6 Additional Resources

Books

Programming Perl
Chapter #4
By Larry Wall, Tom Christiansen, Jon Orwant
O'Reilly & Associates
ISBN: 0-596-00027-8

On line

http://perldoc.perl.org/index-pragmas.html

Note: On line documentation may change.

5.7 Lab Exercises - Suggested lab time: 25 minutes

Important note: If you did not finish the previous lab, either finish it before starting this lab or use the completed parse4.pl provided on your floppy disk.

Modify parse4.pl to include the following changes:

- Implement "use strict"
- Move your subroutines to the bottom of your script

Save these changes into a file called parse5.pl

Unit Eight
Perl TK Basics

Module topics:

Notes:

8.1 Working with Windows

The idea behind Tk is to create an easy to use interface between Perl and Windows. In order to do this, Tk builds on top of the X Window System (or Window XP/2000/2003) to create "sub-windows" that contain buttons, menu bars, scroll bars and other windows components. These components are called "*widgets"*.

Widgets are controls that are built into Motif (the heart of the X Window System on UNIX platforms).   In fact, you can think of Tk as the process of putting widgets together in an application until you have the graphic interface you need.

8.1.1 The TK module

TK isn't part of Perl by default. It needs to be installed on your system and imported into your program with the **use** statement.

To determine if TK is installed on your system, run the following command:

```
[student@linux1 student]$ perl -e "use Tk;"
```

If you don't get any error messages, Tk is installed. If you do get an error message, Tk is probably not installed.

Important Note: Tk is a huge topic. While this section will show you how to create and use basic widgets, a complete discussion of Tk is belong the scope of this class.

Notes:

8.2 Types of Widgets

The following are the primary widgets available to Tk:

Widget	**Purpose**
Frames	Used to group other widgets together.
Toplevels	Toplevels are special frames that create a "separate" window (not a sub-window like normal frames do).
Labels	Similar to frames but also allow text and bitmap graphics to be displayed.
Buttons	Buttons can be used to "*bind*" an action to a graphic.
Checkbuttons	Used to select options.
Radiobuttons	Used to select one option only.
List Boxes	Lists lines of text and allows user to select one or more line.
Scroll bars	Allows the user to control the display with a scroll bar.
Scales	Allows the user to control the setting of an item with a slider bar.
Entries	Allows the user to type in text.
Menus	Give the user menu options.

Each of these widgets will be discussed in greater detail in the next section.

A note regarding options: There are many options for widgets that affect size, position, affects and additional widget features. Many of these options will be discussed as the widgets are explored.

Notes:

8.3 Geometry Managers

While you can modify the look and feel of widgets with different options, geometry managers control the location and size of widgets. Consider these managers as functions that can see the "big picture" while the widgets only can see themselves.

The primary geometry manager in Tk is **pack**. This manager can place a series of widgets within a frame. The **pack** geometry manager is useful for simple Tk applications.

The **grid** geometry manager is designed to allow you to place widgets into rows and columns. The **place** geometry manager is designed to place widgets using an x/y coordinate.

The **pack** geometry manager is probably the most commonly used of the three and the easiest to initially learn. As a result, this course will focus on the **pack** geometry manager.

Notes:

8.4 Creating Widgets

To get started, we are going to create a very simple Tk script. The following will just create a window:

```
#!/usr/local/bin/perl
#8_basic.pl

use Tk;

$main = MainWindow -> new;
$main -> title ("First Tk program!");
MainLoop;
```

Notes about the program:

- The "use Tk;" statement imports the Tk module
- The line "$main = MainWindow -> new;" tells Tk that you want to create a window. The window isn't created until you run the "MainLoop" statement.
- The line "$main -> title ("First Tk program!");" tells Tk that you want to put the string "First Tk program!" in the title bar of the window
- The line "MainLoop;" creates the Window. This statement is referred to as an "event loop".

Notes:

8.5 The OO nature of the Tk module

One aspect of Perl/Tk that "throws" people is that it is an Object Orientated module. If you don't know how OO works in Perl, don't let this aspect of the module throw you off. The good thing about Object Oriented Programming in Perl is that you don't have to understand how to write or read OO Perl code in order to use an OO module.

If you understand the concept of OO from other languages (such as C++ or Java), then the following might be useful information:

```
$main = MainWindow -> new;
```

This command calls the "new" method from the "MainWindow" class and returns an object that is assigned to the $main variable.

With that said, understand that OOP is a concept, not a standard, therefore how OOP "works" in C++ or Java can be quite a bit different than how it works in Perl…

Once again, since you don't know how to write or read OO code in order to use an OO module, covering more detail regarding OOP in Perl is deferred to another class.

8.6 Additional resources

Books

Mastering Perl/TK
by Steve Lidie Nancy Walsh
O'Reilly & Associates
ISBN: 1565927168

Perl/Tk Reference
by Stephen Lidie
O'Reilly & Associates
ISBN: 1565925173

On line

www.perltk.org
w4.lns.cornell.edu/~pvhp/ptk/ptkFAQ.html
www.lehigh.edu/~sol0/ptk/

8.7 Lab Exercises - Suggested lab time: 10 minutes

Taking an existing command-line based script and converting it into a GUI based script can be challenging. Typically the best course of action is to create a separate GUI based script and incorporate the code from the command-line based script.

To start this process, create a program that will generate a window that has the title of "Process Data". At this point the program shouldn't do anything except provide a window.

Save this program as parse8.pl.