Introduction to Perl

Science and Technology Support Group
High Performance Computing

Ohio Supercomputer Center
1224 Kinnear Road
Columbus, OH 43212-1163

OSC

Introduction to Perl

Setting the Stage

- What is Perl?
- How to get Perl
- Basic Concepts

What is Perl?

- Practical Extraction and Report Language
  - Or: Pathologically Eclectic Rubbish Lister
- Created by Larry Wall
- A compiled/interpreted programming language
- Combines popular features of the shell, `sed`, `awk` and C
- Useful for manipulating files, text and processes
  - Also for sysadmins and CGI
- Latest official version is 5.005
  - Perl 4.036 still in widespread use
  - Use `perl -v` to see the version
- Perl is portable
- Perl is free!

How to Get Perl

- Perl's most natural "habitat" is UNIX
- Has been ported to most other systems as well
  - MS Windows, Macintosh, VMS, OS/2, Amiga…
- Available for free under the GNU Public License
  - Basically says you can only distribute binaries of Perl if you also make the source code available, including the source to any modifications you may have made
- Can download source code (C) and compile yourself (unlikely to be necessary)
- Pre-compiled binaries also available for most systems
- Support available via
  - Perl man page
  - `perldoc` command
  - The Usenet group `comp.lang.perl`

Basic Concepts

- A *shell script* is just a text file containing a sequence of shell commands

```
$ cat testscript
echo Here is a long listing of the current directory
ls -l
```

- To run the script, first make it executable and then type its name:

```
$ chmod +x testscript
$ testscript
[output of echo and ls commands]
$
```

Basic Concepts

- Similarly, a Perl script is a text file containing Perl statements
- To indicate that it's a Perl program, include

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

  as the first line of the file
  - Note that the location of Perl on your system may vary!
- To run, make the script executable and then type its name at the shell prompt

```
$ chmod +x myscript.pl
$ myscript.pl
[output of script…]
$
```

Basic Concepts

```
$ cat welcome.pl
#!/usr/local/bin/perl

# Anything from # to EOL is a comment

print (”Enter your name: ”); # what it looks like...
$name = <STDIN>;             # Read from standard input
chop ($name);   # removes last character of $name (the newline)
print (”Hello, $name.  Welcome!\n”);

$ chmod +x welcome.pl
$ welcome.pl
Enter your name: Dave
Hello, Dave.  Welcome!
$
```

Basic Concepts

- Statements executed in “top-down” order; each is executed in succession
- Syntax – and hence style – similar to that of C
  - Free formatting
  - Case sensitive
  - Statements must end with a semicolon (`;`)
  - Groups of statements can be combined into blocks using curly braces (`{...}`)
  - Control structures generally analogous to those in C
  - But no `main()`
  - Also no variable declarations!
    - Introduce and use any type of variable at any time, including growing/shrinking arrays on the fly...
  - Lazy memory “management”

Basic Concepts

- Perl is both an interpreted and a compiled language
- When run, the program is first read and “compiled” in memory
  - Not a true compilation to native machine instructions
  - Similar to creation of Java “bytecode”
  - Some optimizations are performed, e.g.
    - eliminating unreachable code
    - reducing constant expressions
    - loading library definitions
- Second stage is execution via an interpreter (analogous to Java VM)
- Much faster than fully interpreted languages, such as the shell
- No object code

Data Types

- Scalar data
- Arrays
- Associative arrays, or “hashes”

Scalar Data

- A single number or string, depending on context
- References to scalars always begin with $
- Variable names may contain characters, numbers and underscores
- Assignment is done using the = operator
- Examples:

```
$pi = 3.14159;
$color = 'red';
$old_color = "was $color before";
$host = `hostname`; # command substitution
                    # (more on this later)
```

- In general if you refer to a variable before assigning it a value, it will contain the value undef
  - Auto-converts to the null string ("") or zero, depending on context

Numeric Values

- No distinct “types” – conversion handled automatically
- Internally, Perl treats all numbers as `doubles`
- Decimal:

  ```
  $pi = 3.14159;
  ```

- Hexadecimal:

  `$y = 0x1e;` (`$y` has the hex value `1e,` or decimal 30)

- Octal:

  `$y = 075;` (`$y` is now octal 75 = 61 decimal)

- Scientific notation:

  `$z = 3e+2;` (`$z` gets the value 300)

  `$z = 5e-3;` (`$z` gets the value 0.005)

Strings

- Sequences of characters
- No end of string character as in C
- Single-quoted (note: ' not `)
  - no variable interpolation or backslash escape handling, e.g.

```
$x = "dog";
print 'bob $x'; # displays bob $x
```

- Double-quoted
  - variable interpolation and escape handling are performed, so

```
$x = "dog";
print "bob $x"; # displays bob dog
```

Double-Quoted Backslash Escapes

Sequence	Description
`\n`	newline (form feed + carriage return)
`\r`	carriage return
`\t`	tab
`\f`	formfeed
`\b`	backspace
`\a`	bell
`\001`	octal ASCII value (here `ctrl-A`)
`\x20`	hex ASCII value (here space)
`\cD`	control character (here `ctrl-D`)
`\\`	backslash
`\"`	double quote

Some Specially Defined Variables

`$0`	Name of currently executing script
`$_`	default variable for many operations
`$$`	the current process ID
`$!`	the current system error message from `errno`
`$?`	exit status of last command substitution or pipe
`$|`	whether output is buffered
`$.`	the current line number of last input
`$]`	the current Perl version
`$<`	the real `uid` of the process
`$>`	the effective `uid` of the process

The Default Variable

- $_ is a generic default variable for many operations
- Many functions are assumed to act on $_ if no argument is specified

```
print;  # same as print ("$_");
chop;   # same as chop ($_);
```

- $_ is also a default for pattern matching, implicit I/O and other operations
  - Will see examples as we go...
- If you see Perl code that appears to be missing argument(s), chances are that $_ is involved

Conversion Between Numbers and Strings

- If a string is used as though it were a number, it is converted to a number automatically
- Examples:

```
" 3.145foo"  converts to  3.145
"foo"        converts to  0  (without warning!)
```

- If a number is used in a place where a string is expected (for example, if you concatenate a number and a string), it is automatically converted to the string that would have been printed for that number
- Examples:

```
27.123  converts to  "27.123"
1e+4    converts to  "10000"
```

Some Useful String Functions

- `chop ()` [see also `chomp ()`]
  - Takes a string as argument and removes the last character
  - Return value is the removed character
  - Most often used to strip newlines from strings
  - If passed a list of strings, `chop` removes the last character from each

```
print “Enter your name: “;
$name = <STDIN>;
chop ($name);    # remove the trailing newline
print “Hello $name, how are you?\n”;
```

- `length ()`
  - Returns the number of characters in the string

```
$a = “hello world\n”;
print length($a)   # prints 12
```

Arrays (aka “Lists”)

- Ordered lists of scalar data items, indexed by an integer
- Array variable names start with a `@`
- There is a separate namespace for scalar and array variables
  - `$foo` and `@foo` are unrelated
- Arrays are subscripted using square brackets
- Indexing begins at `0`
- The (scalar) variable `$#arry` is the highest assigned index of the array `@arry`
- Arrays need not be declared; they come into existence when used
  - Size can also change dynamically
- Individual array elements can be any mixture of numbers or strings

Arrays

Examples:

```
@nums = (2,4,6);                # initialize an array
$a = $nums[1];                  # $a = 4
$nums[3] = 4;                   # @nums grows automatically
$nums[5] = 12;                  # @nums is now (2,4,6,4,undef,12)
$x = $#nums;                    # $x is 5

@foo = (”one”, ”two”);
@bar = (3.14, @foo, 2.72); # @bar = (3.14, ”one”, ”two”, 2.718)
$foo = $bar[2];                 # $foo = ”two” (no relation to @foo)

@new_array = @old_array;    # copy entire array
@huh = 1;                       # becomes @huh = (1) automatically

@a = (1..5);                    # @a = (1,2,3,4,5);
@b = @a[1..3];                  # a “slice”; @b is (2,3,4)
@c = (A..D);                    # ranges operate using ASCII codes
                                # so @c = (A,B,C,D)
```

Some Useful Array Functions

- `push/pop`
  - Add/remove an element to/from the end of an array
  - Either a scalar or a list can be added

```
push (@a, $b);  # same as @a = (@a, $b);
@x = (1,2);
push (@a, @x);  # same as @a = (@a, 1, 2);
$c = pop (@a);  # returns and removes last element of @a
```

- `shift/unshift`
  - Add/remove element(s) at the beginning of a list
  - Either a scalar or a list can be aded

```
unshift (@a, $b); # same as @a = ($b, @a);
$c = shift (@a);  # returns and removes first element
                  # of @a
```

Some Useful Array Functions

- `reverse`
  - Reverses the elements of a list

```
@a = (7,8,"foo");
@b = reverse (@a);          # @b is now ("foo",8,7)
@b = reverse (7,8,"foo"); # same thing
@x = reverse (@x);
```

- `sort`
  - Returns a list sorted according to ASCII string value

```
@x = sort ("joe","betty","dave"); # @x gets
                                  # ("betty","dave","joe")
@y = (1,2,4,8,16,32,64);
@y = sort (@y);                   # @y is now
                                  # (1,16,2,32,4,64,8)
```

Some Useful Array Functions

- `split ()`
  - Given a delimiter and a string, splits the string into pieces:

```
$file = ”/etc/hosts”;
@tmp = split (”/”, $file); # split using / as delimiter:
                           # @tmp is (“”, “etc”, “hosts”)
```

  - The delimiter can also be a regular expression as well as a literal string

- `join ()`
  - Given a delimiter and an array, joins the array elements together into a single string:

```
$file = ”/usr/local/bin/prog”;
@tmp = split (”/”, $file); # @tmp = (“”,”usr”,”local”,”bin”,”prog”)
pop (@tmp);
$dir = join (”\\”, @tmp);  # $dir is now ”\usr\local\bin”
```

Associative Arrays (aka “Hashes”)

- An array indexed by arbitrary scalars (not necessarily integers)
  - Index values are called *keys*
- Associative array variable names begin with a `%`
- Separate namespace from scalars and ordinary arrays
  - `$foo`, `@foo` and `%foo` are all different
- Subscripted using curly braces `{}`
- Elements have no particular order

Associative Arrays

Examples:

```
$lastname{”John”} = ”Doe”;
$ssn{”John”} = 1234567890;

%ranking = (”UCLA”,1,”OSU”,2);
$x = $ranking{”UCLA”};          # $x is 1

%ranking = (”UCLA” => 1,        # another way to initialize;
            ”OSU” => 2);        # equivalent to the above

@y = %ranking;                  # @y is (”UCLA”,1,”OSU”,2)
%z = @y;                        # %z is the same as %ranking
%z = %ranking;                  # faster way to do the same
```

Some Useful Hash Functions

- `keys`
  - Returns a list (array) of the keys in an associative array

```
$ranks{"UCLA"} = 1;
$ranks{"OSU"} = 2;
@teams = keys (%ranks);  # @teams is ("UCLA","OSU")
                         # or possibly ("OSU","UCLA")
```

  - Note that the order of the elements in a hash is undefined (it is under the internal control of Perl)
- `values`
  - Returns a list of the values in an associative array
  - Order matches that returned by `keys`

Some Useful Hash Functions

- `each`
  - Returns a key-value pair
  - Subsequent calls return additional pairs, stepping through the entire array

```
$ranks{"UCLA"} = 1;
$ranks{"OSU"} = 2;
while (($team, $rank) = each (%ranks)) {
        print ("Ranking for $team is $rank\n");
}
```

- `delete`
  - Removes a key-value pair from a hash, returning the value of the deleted element

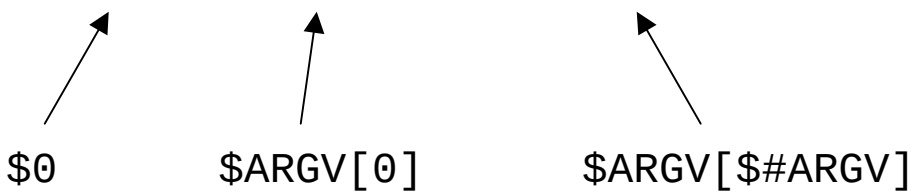
```
$x = delete $ranks{"UCLA"}; # %ranks is now just one
                            # key-value pair, and
                            # $x is 1
```

Some Special Arrays and Hashes

`@ARGV`	command line arguments
`@INC`	search path for files called with `do`, `require`, or `use`
`@_`	default for `split` and subroutine parameters
`%ENV`	the current environment (e.g., `$ENV{"HOME"}`)
`%SIG`	used to set signal handlers

Example:

```
$ script.pl arg1 arg2 … argn
```



A Note on Context

- Much of Perl's behavior is determined by the context in which objects (including variables and function calls) appear
- For example, if a list appears where a scalar is expected, Perl automatically inserts the length of the list

```
$len = @foo;                          # $len is equal to the number
                                      # of elements in @foo

print "Length = ", @foo;              # legal, but wrong
print "Length = ", scalar(@foo); # forces scalar context
```

- In general, `$#arry + 1 == @arry`
- If a scalar appears where an array is expected, it is promoted to a one-element array
- We'll see other context rules later...

Operators

- Numeric operators
- String operators
- Comparisons
- Assignments

Numeric Operators

- These are mostly the same as in C

```
5 + 3
4.4 - 0.3
3*9
27 / 3
11.2 / 6.1
10/3          # floating point, so 3.3333…
2**3          # Fortran-style exponentiation
10 % 3
10.6 % 3.2    # same as above
3 > 2         # returns TRUE (not empty or “0”)
5 != 5        # returns FALSE
```

- Also `&&, ||, ?:, ...`
- Operator associativity and precedence is identical to that in C

Incrementation

- Like C, Perl provides a shorthand for incrementing or decrementing variables
- "Postfix" form:

```
$j++;  # the same as $j = $j + 1
$j--;  # the same as $j = $j - 1
```

  - In an expression, `$j` is first used, and afterwards incremented
- "Prefix" form:

```
++$j;  # the same as $j = $j + 1
--$j;  # the same as $j = $j - 1
```

  - In an expression, `$j` is first incremented, and then used in the expression
- Examples:

```
$j = 10;
$x = $j++;  # $x is 10, $j is 11
$y = ++$j;  # $y and $j both 12
```

String Operators

- Concatenation: `.`

```
"hello" . "world" # the same as "helloworld"
```

- Repetition: `x`

```
"fred" x 3      # same as "fredfredfred"
"Bob" x (1+1)   # same as "BobBob"
(3+2) x 4       # same as "5" x 4 or 5555
                # (note auto-conversion
                # of 5 to "5")
```

Comparisons

- Separate operators for numeric and string comparisons:

Comparison	*Numeric*	*String*
equal	==	eq
not equal	!=	ne
less than	<	lt
greater than	>	gt
less than or equal to	<=	le
greater than or equal to	>=	ge

- Examples:

```
"foo" == "bar"          # true (both are converted to 0!)
"foo" eq "bar"          # false
"zebra" gt "aardvark"   # true (ASCII comparison)
"10" == " 10"           # true
"10" eq " 10"           # false
```

Assignments

- A scalar assignment itself has a value, which is equal to the value assigned:

```
$b = 4 + ($a = 3);
```

  This assigns 3 to $a, then assigns 4 + (3) or 7 to $b

- As in C, binary operators can be turned into assignment operators:

```
$a += 5;      # same as $a = $a + 5;
$x *= 42;     # same as $x = $x * 42
$str .= " ";  # same as $str = $str . " ";
```

- Like all assignments, these have a value as well:

```
$a = 3;
$b = ($a += 4); # $a and $b are both 7 now
```

Exercises 1

1. Write a program that computes the area of a circle of radius 12.5.
2. Modify the above program so that it prompts for and accepts a radius from the user, then prints the area.
3. Write a program that reads a string and a number, and prints the string the number of times indicated by the number on separate lines. (Hint: use the x operator.)
4. Write a program that reads a list of strings and prints out the list in reverse order.
5. Write a program that reads a list of strings and a number, and prints the string that is selected by the number.

Exercises 1

6. Write a program that reads and prints a string and its mapped value according to the mapping

Input	*Output*
red	apple
green	leaves
blue	ocean

7. Write a program that reads a series of words with one word per line, until end-of-file (ctrl-D), then prints a summary of how many times each word was seen.

Control Structures

- `if/unless`
- `while/until`
- `for`
- `foreach`
- `do`
- Simple constructs

if/unless

- Basic decision making:

```
if (expression) {
        true_statement1;
        true_statement2;
        …
} else {
        false_statement1;
        false_statement2;
}
```

- Curly braces are required around each block (unlike in C)
- *expression* is evaluated for a *string* value to determine its truth or falsehood

Truth and Falsehood

- When testing an *`expression`* for truth, it is first converted to a string
- Basic rules are then:

String Value	***True or False?***
Empty (`""`) or `"0"`	False
Anything else	True

- Examples:

```
0       # converts to “0”, so false
1-1     # converts to 0, then “0” so false
1       # converts to “1” so true
“00”    # not “” or “0”, so true
“0.000” # true for the same reason
undef   # converts to “”, so false
```

if/unless

- Can also include any number of `elsif` clauses:

```
if (expr1) {
        expr1_true1;
        expr1_true2;
        …
} elsif (expr2) {
        expr2_true1;
        expr2_true2;
        …
} …
} else {
        all_false1;
        all_false2;
}
```

- Note: the `else` block is optional
- `unless` is just `if` negated

while/until

- Basic iteration:

```
while (expression) {
        statement1;
        statement2;
        …
}
```

- *expression* is tested and, if true, the following block of statements is executed
- At the end of the block, *expression* is tested again
- *expression* must become false at some point, or the loop will be infinite!

while/until

- Example:

```
$j = 1;
while ($j <= 10) {
        print “The square of $j is “;
        print $j * $j, “\n”;
        $j++;
}
```

- `until` is just the negation of `while`:

```
$j = 1;
until ($j > 10) { # same loop as above
        print “The square of $j is “;
        print $j * $j, “\n”;
        $j++;
}
```

for

- Another looping construct:

```
for (init_expr; test_expr; incr_expr) {
        statement1;
        statement2;
        ...
}
```

- Example:

```
for ($j=1; $j<=10; $j++) {
        print “The square of $j is “;
        print $j * $j, “\n”;
}
```

- Note that `$j = 11` when the loop exits

foreach

- Allows convenient cycling through the elements in a list:

```
foreach $var (@list) {
        statement1;
        statement2;
        ...
}
```

- The scalar variable `$var` takes on the value of each item in `@list` in turn
- `$var` is local to the construct; it becomes `undef` when the loop is finished
- If you omit the scalar variable `$var`, Perl assumes you specified `$_` instead

```
foreach (@a)    # same as foreach $_ (@a)
{
      print;    # same as print "$_";
}
```

foreach

- Example:

```
foreach $j (1..10) {
        print “The square of $j is “;
        print $j * $j, “\n”;
}
```

- If `@list` is a single array variable, then `$var` is actually a *reference* to the items in `@list`
- This means that if you modify `$var` in the loop, you are actually changing that element in `@list`:

```
@a = (3,5,7,9);
foreach $tmp (@a) {
        $tmp *= 3;
}
# @a is now (9,15,21,27)!
```

do while/until

- Similar to C:

```
do {
        …
} while ($something_is_true);

do {
        …
} until ($something_is_false);
```

- Loop control statements (`next`, `last`) do not work here, though

Simple Constructs

- To loop on or branch around a single statement, you can use

```
print “Odd\n” if $n % 2 == 1;
print “Even\n” unless ($n % 2 == 1);
print $num--,”\n” while $num > 0;
```

- Note position of the semicolon
- `else` is not allowed with this construct

Simple Constructs

- Logical constructs can be built from && (“and”) and || (“or”)

```
expr1 && expr2; # equivalent to:
                # if (expr1) { expr2 };

expr1 || expr2; # equivalent to:
                # unless (expr1) { expr2 };
```

- Example:

```
# Try to open file; print error msg and exit on failure

open (FH,”/etc/llamas”) || die ”cannot open llamas!”;
```

Basic I/O

- Basics
- Reading from stdin
- The diamond operator
- Writing to stdout and stderr
- Error message shortcuts

I/O Basics

- I/O in Perl proceeds through *filehandles*, which are used to refer to input or output streams
- We will see later how to attach these to files (or devices) for reading and writing
- There are three pre-defined filehandles:

  `STDIN` – refers to the keyboard

  `STDOUT` – connected to the screen

  `STDERR` – connected to the screen

Reading from the Standard Input

- To read from the keyboard, use `<STDIN>`
- Behavior depends on context:

```
$line = <STDIN>;  # In a scalar context, reads the next line
                  # of input, placing it in the scalar
                  # variable $line.

@lines = <STDIN>; # In an array context, reads all remaining
                  # lines.  Each line becomes an element of
                  # the array @lines.
```

- Note that newlines remain intact
  - Can use `chop ()` to remove these

Reading from the Standard Input

- Example: Read and echo a series of input lines

```
while ($line = <STDIN>) {
        print $line; # echo each line as it is entered
}
```

- Shortcut: whenever a loop test consists solely of an input operator, Perl copies the input line into the variable `$_`:

```
while (<STDIN>) {
        print $_;    # the same thing!
}
```

- `$_` is a default for many Perl functions and operations

The Diamond Operator

- If the input operator is used without a filehandle, as `<>`, data are read from the files specified on the command line

```
$ cat cat.pl
#!/usr/local/bin/perl

while (<>) { print $_; }
$ cat.pl file1 file2 … filen
```

- In fact, Perl looks at `@ARGV` for the list of input files
- Can set or modify this from within the script:

```
@ARGV = ("file1", "file2");
while (<>) { print; } # $_ is the default for print, too!
```

- If no files are specified, `<>` reads from `STDIN` instead

Printing to the Standard Output

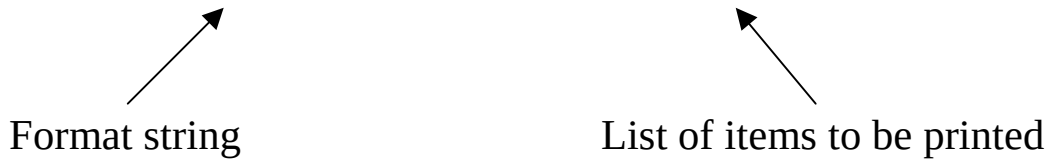
- `print`
  - Takes a list of strings as argument, and sends each to `STDOUT` in turn
  - No additional characters are added
  - Returns a true/false value indicating whether the print succeeded
  - Examples:

```
print (”Hello”,”world”,”\n”); # prints “Helloworld” with newline
print (2+3), ”foo”;           # prints 5, “foo” ignored!
print ((2+3),”bar”);          # prints 5bar
print 2+3,”bar”;              # also prints 5bar
```

Printing to the Standard Output

- For formatted output, use `printf`
  - Works just like the C function of the same name
  - Example:

```
printf "%15s %5d %10.2f\n", $a, $b, $c;
```



  - See `printf` man page for further details

Printing to the Standard Error

- Just include `STDERR` as the first argument to `print` or `printf`

```
print STDERR ”Whoops, an error has occurred!\n”;
```

- This also appears on the terminal screen by default, but can be redirected by the shell separately from the standard output

```
$ script.pl > output 2> err.log
```

Error Message Shortcuts

- `die ()`
  - Takes a list as its argument
  - Prints the list (like `print`) to `STDERR` and ends the Perl process
  - If no `\n` appears at the end of the printed string, die also prints the script name and line number

```
die "Oops, an error occurred\n"; # prints msg and exits

die "Error occurred at "; # prints msg, then script name
                          # and line number of die
```

- `warn ()`
  - Same as `die`, except Perl does not exit

```
warn "Debug enabled" if $debug;
```

Exercises 2

8. Write a program that accepts the name and age of the user and prints something like “Bob is 26 years old.” Insure that if the age is 1, “year” is not plural. Also, an error should result if a negative age is specified.
9. Write a program that reads a list of numbers on separate lines until 999 is read, and then prints the sum of the entered numbers (not counting the 999). Thus if you enter 1, 2, 3 and 999 the program should print 6.
10. Write a program that reads a list of strings and prints out the list in reverse order, but without using the `reverse` operator.
11. Write a program that prints a table of numbers and their squares from 0 to 32. Try to find a way where you don’t need all the numbers from 0 to 32 in a list, then try one where you do.
12. Build a program that computes the intersection of two arrays. The intersection should be stored in a third array. For example, if `@a = (1, 2, 3, 4)` and `@b = (3, 2, 5)`, then `@inter = (2, 3).`

Exercises 2

13. Write a program that generates the first 50 prime numbers. (Hint: start with a short list of "known" primes, say 2 and 3. Then check if 4 is divisible by any of these numbers. It is, so you now go on to 5. It isn't, so push it onto the list of primes and continue…)
14. Build a program that displays a simple menu. Each of the items can be specified either by their number or by the first letter of the selection (e.g., P for Print, E for Exit, etc.). Have the code simply print the choice selected.
15. Write a program that asks for the temperature outside and prints "too hot" if the temperature is above 75, "too cold" if it is below 68, and "just right" if it between 68 and 75.
16. Write a program that acts like `cat` but reverses the order of the lines.

Regular Expressions

- Overview
- Metacharacters
- Substitutions
- Translations
- Modifiers
- Memory
- Anchoring patterns
- Miscellaneous

Overview

- A regular expression defines a pattern of characters
- Typical uses involve pattern matching and substitution
- Used by many UNIX programs (`grep`, `sed`, `awk`, `vi`, `emacs`, …), but not always with exactly the same rules!
- Also appears similar to shell wildcarding (“globbing”), but the rules are *much* different

RE Basics

- A regular expression (RE) in Perl is indicated by enclosing it in forward slashes:

```
/abc/
```

  This represents a pattern consisting of these three characters
- When compared against a string, the result is “true” if the pattern “abc” occurs anywhere in that string
- Comparison operator: =~

```
# print a string if it contains “abc”

if ($line =~ /abc/) {
        print $line;
}
```

- Negated comparison: !~

RE Basics

- Consider the regular expression

  ```
  Abc
  ```

  - Each character is itself a RE which matches only that single character
  - Case sensitivity: “A” does not match “a”
- REs are composed of two types of characters
  - “literals”, or ordinary characters
  - “Metacharacters,” which have a special meaning

Metacharacters

- These characters have a special meaning inside a regular expression

Character	*Meaning*
.	Any single character *except* newline (\n)
*	zero or more of the preceding RE
?	Zero or one of the preceding RE
+	One or more of the preceding RE
{}	Some number of the preceding RE
()	grouping and sub-expressions
|	'or'
[]	a character class

- To remove their special meaning, you can backslash-escape them
- Note that backslash-escapes (`\n, \t, \r, \f,` …) retain their special meaning inside a RE

Examples

- Any single character matches itself, so

  ```
  bob
  ```

  will match the word "bob"
- What if we want to match bob or bobby, or anything containing an "o"?

  `b.b` matches:

  ```
  bob, bib, bbb, ...
  ```

  `bob*` matches:

  ```
  bob, bobbb, bobcc, ...
  ```

  `bob.*` matches:

  ```
  bob, bobby, bob barker, ...
  ```

Character Classes

- Represented by `[]` enclosing a list of characters
- This matches any *one* of the characters in the list
- Examples:
  - What if we want `b.b` but only with a vowel in between?

    ```
    b[aeiou]b
    ```

  - To ignore capitalization?

    ```
    [Bb]ob
    ```

- Ranges are also allowed, for example
  - `[b-d]ob` matches: `bob, cob,` or `dob`
- To get a literal dash (`-`) in a class, precede it with a backslash
  - `[0-9\-]` matches any single digit or a dash
- If `^` is the first character in the list, the class is *negated*
  - `[^0-9]` matches any single non-digit

Special Class Abbreviations

- Perl provides shorthand names for some common classes:

Construct	Equivalent Class	Negated Construct
`\d (digits)`	`[0-9]`	`\D`
`\w (words)`	`[a-zA-Z0-9_]`	`\W`
`\s (space)`	`[ \r\t\n\f]`	`\S`

(For example, `\D` is equivalent to `[^0-9]`)

More Examples

- `be+` matches:

  `be, bee, beeeeeeee, ...`

- To match `bob` or `bobby`, make use of parentheses for grouping:

  `bob(by)?`

- Using the "or" symbol to match `bob` or `dog`:

  `(bob|dog)`

- Combining metacharacters: `b[aeiou]*b` matches:

  `bob, bab, baaab, boab, beieb, bb, ...`

- `Bo?b` will only match:

  `Bob or Bb`

The General Multiplier

- Curly braces `{}` can be used to specify in detail how many occurrences of a RE are desired:

  `x{5,10}` matches five to ten occurrences of `x`

  `x{5,}` matches 5 or more occurrences of `x`

  `x{5}` matches exactly 5 occurrences of `x`

  `x{0,5}` matches five or fewer occurrences of `x` (must include the zero)

- Examples:
  - `a.{5}b` matches `a` followed by exactly 5 non-newline characters and a `b`
  - What about `\([0-9]{3}\) ?[0-9]{3}-[0-9]{4}` ?

Parentheses

- Used for grouping sub-expressions
- `(bob)+` matches:

  `bob, bobbob, bobbobbobbob, ...`
- Also causes the enclosed pattern to be memorized
- These patterns can then be recalled as `$1, $2, $3, …`
  - Within the pattern match, use `\1, \2, \3, …` instead
- Thus `Fred(.)Barney\1` matches:

  `FredxBarneyx` or `FredyBarneyy` but not `FredxBarneyy`
- `a(.)b(.)c\2d\1` matches:

  `a`, any one character (call it #1), `b`, any one character (call it #2), `c`, character #2, `d`, character #1 (For example: `aXbYcYdX`)
- Memory is also very useful when doing substitutions...

String Substitutions

- String modification is performed using the “substitute” operator:

```
$var =~ s/regexp/replacement-string/;
```

- The variable `$var` is matched against the RE *`regexp`*. If successful, the part that matches is replaced by *`replacement-string`*

```
$_ = ”foobar”;
$_ =~ s/bar/bear/; # $_ is now “foobear”
```

- If several parts of `$var` match *`regexp`*, only the first is substituted for by default

```
$_ = ”foobaring up the foobar road”;

$_ =~ s/bar/bear/; # $_ is now “foobearing
                   # up the foobar road”
```

Translations

- Similar to the `tr` program in UNIX, the `tr` operator translates characters in regular expressions:

```
$str = “Bob the Dog”;

$str =~ tr/a-z/A-Z/;    # $str now “BOB THE DOG”
$str =~ tr/BDO/xyz/;    # $str now “xzx THE yzG”

$str =~ tr [A-Z] [a-z]; # also valid syntax
                        # $str is now “xzx the yzg”
```

Modifiers

- REs can have optional modifying suffixes. These include “g” (global; substitute as many times as possible), “i” (case insensitivity), “m” (treat string as multiple lines), and “s” (treat string as a single line)

```
$_ = ”foobaring up the foobar road”;

$_ =~ s/bar/BEAR/g;     # $_ is now “fooBEARing
                        # up the fooBEAR road”
```

```
if ($str =~ /abc/i) { # same as /[Aa][Bb][Cc]/
        …
}

$_ = ”fooBARing up the foobar road”;

$_ =~ s/bar/bear/ig; # $_ is now “foobearing
                     # up the foobear road”
```

Memory and Substitutions

- Example:

```
$name = "John Wilbur Smith";

$name =~ s/(\w+)\s+(\w)?\w*\s(\w+)/\3, \1 \2\./;

# $name is now "Smith, John W."
```

- Note that matches are *greedy*; the longest string that matches is the one taken

```
$foo = "fred xxxxxxxxxx barney";

$foo =~ s/(x*)/boom/;  # $foo is now "fred boom barney"
                       # and $1 is "xxxxxxxxxx"
```

Anchoring Patterns

- Special notations that allow you to anchor the pattern to specific parts of the string:

Symbol	*Meaning*
^	beginning of string
$	end of string
\b	word boundary
\B	*not* a word boundary

- Examples:

```
/^fred/    # matches fred only at the beginning of the string
/betty$/   # matches betty only at the end of the string
/fred\b/   # matches fred, but not freddy
/\bwiz/    # matches wiz and wizard, but not twiz
/\bFred\B/ # matches Frederick but not “Fred Flinstone”
```

Using a Different Delimiter

- By default `/` is used to delimit REs
- To match an expression containing slashes, you can backslash-escape them

```
# Match /etc/passwd
if ($file =~ /\/etc\/passwd/) {
        …
}
```

- Alternatively, you can use a different character such as `:` or # as the delimiter, by explicitly giving the `m` (“match”) prefix:

```
# Match /etc/passwd
if ($file =~ m:/etc/passwd:) {
        …
}
```

- Now the forward slash isn’t special

Variable Interpolation in REs

- Variables that appear in regular expressions are substituted (interpolated) *before* the RE is scanned for other special characters (metacharacters)
- Allows you to construct REs from computed strings in addition to literals

```
$sentence = ”Every good bird does fly”;
print “What should I look for? “;
$string = <STDIN>;
chop ($string);     # remove trailing newline
if ($sentence =~ /$string/) {
        print ”$string occurs in $sentence\n”;
else {
        print “”$string not found\n”;
}
```

- `$string` can also contain metacharacters, which are interpreted normally

More Pattern Matching Variables

- Recall that character sequences that match subexpressions in parentheses are assigned to `$1, $2, $3, …` in sequence
- In addition, after a successful match the entire text that matched is stored in the variable `$&`
- All of the text before the match is assigned to `$`` `
- All of the text following the match is assigned to `$'`
- Example:

```
$_ = "H35j78";

if ($_ =~ /\d+/)
        print "$` - $& - $'"
}

# Output is: H - 35 - j78
```

Functions

- Defining a function
- Invoking a function
- Arguments
- Return values
- Local variables
- Example: Advanced sorting

Defining a Function

- Also called subroutines, or sometimes just “subs”
- General construct:

```
sub my-subname {
        statement_1;
        statement_2;
        ...
}
```

- Function definitions can appear anywhere in the program
  - Need not occur before they are called
- Separate namespace from variables, so you can have a subroutine named `foo` along with variables `$foo`, `@foo` and `%foo`
- By default (almost) all variable references in a function are *global*

Invoking a Function

- To invoke a function, precede the function name with &

```
&say_hello;       # invokes the function say_hello

$a = 3 + &radius; # part of an expression

for ($x = &start_val; $x <= &end_val; $x += &incr) {
        …
}
```

- Functions can call other functions (including themselves)

Arguments

- Arguments may be passed (in parentheses) to a function
- Any arguments passed to the function appear in the special array `@_`
- `@_` is local to the function
  - If there is a global variable `@_`, it is saved and restored after the function exits
- No formal (dummy) parameters

```
sub print_msg {
        print “First argument:  $_[0]\n”;
        print “Second argument: $_[1]\n”;
}

&print_msg(“foo”, 42);
```

- Note that `$_[0]` is unrelated to `$_` !

Return Values

- A function returns a value to the code that called it, which may be assigned or used in some other way
- The return value of a function is the value of the last expression evaluated in the body of the function

```
sub double_a {
        $a *= 2;
}

$a = 3;
$c = &double_a; # $c is now 6
```

- Can also use `return (val);`
- The returned value can be a scalar or a list

Return Values

- Another example:

```
sub add {
        $sum = 0;
        foreach $n (@_) {
                $sum += $n;
        }
        $sum; # Required!
              # could also use return($sum);
}

$a = &add(4,3);        # $a is now 7
$b = $a + &add(1..5); # $b is now 7+1+2+3+4+5=22
```

- Without the last line `$sum` (or the return statement), the last expression evaluated would be `foreach`, resulting in a null return value
- If `$sum` did not exist before invocation of `add`, it pops into existence when `add` is first invoked

Local Variables

- By default, most variables are global in Perl
- `@_` is local to each function, however
- Can define other local variables using

```
local ($var1, $var2, …)
```

- Takes a list of variable names and creates local instances of them
- Inside the function, local variables *mask* any global variables with the same name(s)
  - Values of global variables are saved, and restored after the function exits
- `local` can also be used inside ordinary code blocks `{ … }`
- In Perl 5, `my` is (essentially) a synonym for `local`

Local Variables

- Example:

```
sub greater_than {
        local ($n, @values) = @_; # create some local variables
        local (@result);          # to hold the return value
        foreach $val (@values) {  # step through arg list
                if ($val > $n) {  # is it eligible?
                        push (@result, $val); # include it
                 }
        }
        return (@result);         # return final list
}

@new = &greater_than(55,@list);    # @new gets all @list > 55
@foo = &greater_than(5,1,5,15,30); # @foo is (15,30)
```

- Note assignment of @_ to other local variables for readability
- Could use `my` in place of `local` here

Example: Advanced Sorting

- By default, `sort` sorts the elements of a list according to their ASCII values
- You can change this behavior by
  - Writing a function that sorts according to some other rule
  - Telling `sort` to use this function
- The sorting function should assume two arguments `$a` and `$b`, and return
  - any negative number if `$a` is "less than" `$b` (i.e., if `$a` should come before `$b` in the sorted list)
  - zero if `$a` "equals" `$b` and
  - any positive number if `$a` is "greater than" `$b` (i.e., if `$a` should come after `$b`)

Example: Advanced Sorting

- Numeric comparison:

```
sub numerically {
        if ($a < $b) {
                -1;
        } elsif ($a == $b) {
                0;
        } elsif ($a > $b) {
                1;
        }
}

@new = sort numerically (@list); # tells sort to use numerically
                                 # in sorting the list

# Shorthand:

sub numerically {
        $a <=> $b;  # the “spaceship” operator; same as the
}                   # above
```

Exercises 3

17. Construct a regular expression that matches
    - at least one `a` followed by any number of `b`'s
    - any number of backslashes followed by any number of asterisks
    - three consecutive copies of whatever is contained in `$whatever`
    - any five characters, including newline
18. Write a program that accepts a list of words on `STDIN` and searches for a line containing all five vowels (a, e, i, o, and u).
19. Modify the above program so that the five vowels have to be in order.
20. Write a program that looks through the file `/etc/passwd` on `STDIN`, printing the real name and login name of each user. (Hint: use `split` to break each line up into fields, then `s///` to get rid of the parts of the comment field that are after the first comma.)

Exercises 3

21. Write a subroutine that takes a numeric value from 1 to 9 and returns its English name (i.e., one, two …). If the input is out of range, return the original value as the name instead.
22. Taking the subroutine from the previous exercise, write a program to take two numbers and add them together, printing the result as "`Two plus three equals five`." (Don't forget to capitalize the first letter!)
23. Create a subroutine that computes factorials. (The factorial of 5 is 5! = 5*4*3*2*1 = 120.) Try this using a normal subroutine and a recursive one (i.e., a subroutine that calls itself).
24. Build a function that takes an integer and returns a string that contains the integer displayed with a comma every three digits (i.e., 1234567 should return 1,234,567).

File and Directory Manipulation

- Filehandles
- Opening a filehandle
- Using filehandles
- File tests
- Moving around the directory tree
- Globbing
- Operations on files

Filehandles

- A filehandle is the name of an I/O connection between your Perl process and the outside world
- Already seen `STDIN` and `STDOUT`/`STDERR`
  - I/O connections to keyboard and screen
- Filehandles have a separate namespace from other Perl entities
  - Can have `$foo`, `@foo`, `%foo`, `&foo`, as well as filehandle `foo`
- Recommended style is to uppercase filehandles, but this is not required

Opening a Filehandle

- The operation

```
open (HANDLE, "filename");
```

  opens the file *filename* and attaches it to *HANDLE*
- A prefix to *filename* controls whether file is opened for reading, writing, appending, etc.
- Returns true or false (actually `undef`) indicating success or failure of the operation
  - Can fail due to, e.g., permissions, file not found, etc.

```
open (FH, "myfile") or die "can't open myfile!";
```

- To close a file and release the filehandle, use

```
close (FILEHANDLE);
```

Examples

```
open(PWD, ”/etc/passwd”);  # open for reading; fails if
                           # /etc/passed doesn’t exist or can’t
                           # be read

open(FH, ”>myfile”);       # write-only; if myfile doesn’t
                           # exist it is created, else clobbered

open(LOG, ”>>logfile”);    # append mode; if logfile doesn’t
                           # exist it is created

open(TOPRINTER, ”| lpr”);  # can also “print” to UNIX pipelines

open(FROMPIPE, ”ls -l |”); # or read from them

close(LOG);
```

Using Filehandles

- Once a filehandle has been opened for reading, you can read from it by enclosing the filehandle in <>, just as for `STDIN`:

```
open (FH, ”myfile”);    # open myfile for reading
while ($line = <FH>) { # read a line from the file
     print “$line”;     # echo it to STDOUT
}
```

- As before, `<FH>` reads the next line from the file in a scalar context
- In an array context, `<FH>` all remaining lines of the file are read and placed in an array
- Newlines are retained (can use `chomp ()` to remove them)
- Returns `undef` (hence “false”) if there are no more lines to read

Using Filehandles

- To write/append to a filehandle, give the filehandle as the first argument to `print`:

```
open (LOGFILE, ”>>build.log”);

print LOGFILE ”Finished building application\n”;
```

- Note: no comma after the filehandle!
- `STDOUT` is the default filehandle for `print` and `printf`

File Tests

- Can test for existence, ownership, permissions, etc. of files and directories
- General form of test is

```
-X file
```

  where `X` is some character and *`file`* is the file or directory to be tested
  - `file` can be a name or a filehandle
- Most tests return true/false, though some return numbers (e.g. `-s`, which returns the size in bytes of a file)

```
$x = ”/etc/passwd”;
if (-e $x) {     # does the file exist?
        print “Crack some passwords!\n”;
}
```

File Tests

- More examples:

```
if (-r $file && -w $file) {
        # $file exists and I can read and write it
        . . .
}

chop ($fname = <STDIN>);
if (-A $fname > 7) {  # last accessed more than seven days ago?
        print “Say goodbye to $fname...\n”;
        unlink $fname; # delete the file
} else {
        print “$fname accessed recently!\n”;
}

if (-e) { # same as if (-e $_), of course!
        …
}
```

File Tests

File Test	*Meaning*
`-r`	file or directory is readable
`-w`	file or directory is writable
`-x`	file or directory is executable
`-o`	file or directory is owned by user
`-e`	file or directory exists
`-z`	file or directory exists and has zero size
`-s`	file or directory exists and has nonzero size (return value is size in bytes)
`-f`	entry is a plain file
`-d`	entry is a directory
`-T, -B`	file is text, binary
`-M, -A`	modification, access time (in days)

Moving Around the Directory Tree

- When your Perl program is launched it inherits the environment of its parent, (usually the shell) including the current directory
- To move to another directory, use `chdir("dirname");`
- Return value is true/false indicating whether the change was successful

```
print "Where do you want to go? ";
chop ($where = <STDIN>);

if (chdir $where) {
        # we got there
        …
} else {
        # didn't make it for some reason
        …
}
```

Globbing

- Can expand shell wildcards (“globbing”) by putting the globbing pattern inside <>
- Example:

```
@list = </etc/host*>;
```

  returns a list of all filenames in `/etc` that begin with `host`
- In a scalar context it would return the next filename that matches, or `undef` if no others remain

```
while($next = </etc/host*>) {
        $next =~ s#.*/##; # remove part before last slash
                          # (note use of # as delimiter)

        print “one of the files is $next”;
}
```

Globbing

- Multiple patterns are allowed inside the glob, for example

```
@foo_bar_files = <foo* bar*>;
```

- Generally, anything you could send to the shell for expansion will work in a glob
- Note: looks similar to regular expressions, but the meaning of the various metacharacters is very different!

Operations on Files

- Some useful functions for performing operations on files/directories:
  - `unlink ("filename");`
    - Removes ("unlinks") a list of files

```
unlink (<*.o>); # just like 'rm *.o' in the shell
```

  - `rename ("file1","file2");`
    - Renames (moves) `file1` to `file2`

```
rename ("foo","bar"); # like 'mv foo bar' in the shell
```

    - Note: `rename ("file","directory")` is not allowed!

Operations on Files

- `mkdir ("dirname", mode);`
  - Creates a directory with permissions set by *mode*

```
mkdir ("foo", 0777); # creates direcoty foo with rwx
                     # permissions for all
```

- `rmdir ("dirname");`
  - Removes a directory

```
rmdir ("foo"); # just like 'rmdir foo' in the shell
```

- `chmod (mode,"file1","file2",…);`
  - Sets permissions for listed files to *mode*

```
chmod (0666,"foo","bar"); # gives everybody rw
                          # permissions for foo and bar
```

Operations on Files

- All of these functions return true/false indicating success or failure of the operation
- Examples:

```
unlink (”foo”) || die “Unable to delete foo”;


if (mkdir (”tmp”,0700)) {
        # creation was successful, proceed
        …
} else {
        # creation failed!
        …
}
```

External Processes

- Using backquotes
- `system ()`
- Output to and from pipes

Using Backquotes

- An external command can be run by placing it in backquotes: ` `
- The command output is becomes the value of the backquoted string

```
$now = ”the time is now: ” . `date`;
```

- Output of the date command is concatenated with the previous string
- If the backquoted command appears in an array context, you get an array of strings each of which is one line of the command output

```
@files = `ls -l`; # each element of @files contains
                  # one line of ls -l output
```

- Variable interpolation does occur inside backticks
- Look out for newlines (use `chop/chomp` if desired)

system()

- Another way to execute an external command
- If given a scalar, `system` passes it to `/bin/sh` for execution

```
system (“date”);
```

- The scalar can be anything `sh` can process, including multiple commands separated by semicolons
- If given a list, system takes the first item as the command, and subsequent items as arguments to that command

```
system (”grep”,”INTEGER”,”prog.f”);
```

- Note that shell processing (globbing) does *not* occur for these arguments:

```
system (”/bin/echo”,”*”); # just echos *
```

system()

- Where does the output of the command go?
- The shell inherits STDOUT and STDERR from the Perl process, so output normally goes to the screen
- This can be changed using ordinary sh redirects:

```
system ("a.out > outfile 2> err");

$where = "who_out" . ++$i;          # make a filename
system ("(date; who) > $where &"); # interpolation
```

- system returns the exit status of the command, usually 0 if no error occurred
- Backwards from normal convention:

```
system ("date > now") && die "cannot create now";
```

  - die invoked if command returns nonzero (i.e., true)

Output to and from Pipes

- Reading from a pipe:

```
open (PIPE, "ls -l |") or die "ls error!?!";
```

  - In a scalar context, <PIPE> returns a single line of output from `ls -l`
  - In an array context, <PIPE> returns all (remaining) output lines
  - The command is run when the filehandle is opened
  - Shell metacharacters other than | (e.g., redirects) are also processed
- Writing to a pipe:

```
open (LPTR, "| lpr");
```

  - `print LPTR ... ;` now sends to the standard input of `lpr`
  - When the filehandle is closed (or the script exits), the command is run

References

- Creating references
- Anonymous references
- Using references
- Passing references
- Nested datastructures

Creating References

- A reference is like a pointer in C
- It is a scalar object that holds the location of the data associated with some variable
  - Said to “point to” the variable
- You can create references to nearly any kind of data
  - Scalars, arrays, hashes, functions
- A reference is created using the \ operator

```
$sref = \$x;      # $sref points to the scalar $x
$aref = \@arry;   # pointer to an array
$cref = \0xFA0;   # reference to a constant
$code = \&myfun;  # pointer to a function
@reflist = \($a,$b,$c)
```

Anonymous References

- You can also directly create references to unnamed objects:

```
# array reference:
$aref = [1,”foo”,”a”,42];

# hash reference:
$href = {
        ”UCLA”, 1,
        ”tOSU”, 2,
}

# function reference:
$code = sub { print “Hello world\n”; }
```

Using References

- To use a reference (dereference it, in C parlance), just put the appropriate type indicator in front of the reference variable

```
$foo = 12;
$sref = \$foo;       # $sref points to foo
print “$$sref”;      # prints 12
$$sref = “bar”;      # $foo is now “bar”

@arry = (1,2,3);
$aref = \@arry;      # $aref refers to @arry
@$aref = (2,4,6);    # changes @arry
$$aref[2] = ”gaak”; # reference to individual element
```

Passing References

- Can also pass variables by reference to functions

```
sub doubler {

        my @reflist = @_;  # local array for incoming pointers

        foreach (@reflist) {
                $$_ *= 2;  # $_ is a reference,
                           # $$_ is what it points to
        }
}

&doubler (\$a, \$b, \$c);
```

Passing References

- Can also return references, of course

```
sub arrayinit {

        my @a = (1,3,5);
        my @b = (2,4,6);

        return (\@a, \@b);
}

($aref, $bref) = &arrayinit ();
```

Nested Datastructures

- Perl does not support arrays of arrays directly, but you can create an array of references, each of which refers to an array
- The operator `->` can be used to dereference array pointers

```
$aptr = [1,2,3];   # anonymous array reference
$$aptr[0] = 3.14;  # changes 0th element of the array
$aptr->[0] = 3.14; # same thing

# here is an array of pointers, each of which refers
# to a hash consisting of a single key/value pair:

$ptr = [{“cow”,”purple”},{“llama”,”blue”}];

$ptr->[1]->{“llama”} = “scarlet”;
$ptr->[1]{“llama”} = “scarlet”;    # equivalent
```

Exercises 4

25. Write a program that reads a filename from STDIN, then opens the file and prints its contents preceded by the filename and a colon. For example, if the file fred contains the lines aaa, bbb and ccc, the output should be

```
fred: aaa
fred: bbb
fred: ccc
```

26. Write a program to read in a list of filenames and then display which of them are readable, writeable and/or executable, and which ones don't exist.
27. Write a program that accepts a list of filenames and finds the oldest file among them. Print the name of that file along with its age in days.
28. Write a program to change directory to a location specified as input, the print a listing of the files there. Do not show a list if the directory change doesn't succeed; in this case simply warn the user.

Exercises 4

29. Write a program that works like `rm`, deleting the files given as command line arguments. (Be careful testing this!)
30. Write a program to parse the output of the data command to get the current day of the week. If it is a weekday, print “get to work,” otherwise “go play.”
31. Using references, build a structure (as in C) that represents a circle image. Your circle struct should contain four data items: the x and y coordinates of the center point, the radius and the color. (Hint: the struct can be a hash where the keys are the names of the data items and the values are reference variables pointing to the actual data.)

Some Other Topics of Interest

- The Perl debugger (`perl -d`)
  - Also highly recommend using `perl -w` (print warnings) during development!
- Advanced process management
- Handling binary data (`pack` and `unpack`)
- Formats
- Database interfaces
- CGI
- Many more features, functions and options in virtually all the areas we've covered so far...

Some Additional Features in Perl 5

- Perl compiler: can compile Perl code into a machine-native executable
- Lexical scoping of variables: variables may be declared with a lexical scope
- Regular expression enhancements: pattern grouping can be done without using backreferences, and whitespace and comments can be embedded in a RE
- Loadable code modules: the Perl library in Perl 5 is defined in terms of packages, and there are also many 3rd-party packages available throuth CPAN
- Object oriented programming
- And more...

For Further Information

- *Learning Perl, 2nd ed.*, Randal L. Schwartz (O'Reilly and Associates, 1996)
- *Programming Perl, 2nd ed.*, Larry Wall, Tom Christiansen, and Randal L. Schwartz (O'Reilly and Associates, 1996)
- *Advanced Perl Programming*, Sriram Srinivasan (O'Reilly and Associates, 1997)
- *Perl 5 Desktop Reference*, John Vromans (O'Reilly and Associates, 1996)
- `http://www.perl.com`
- `http://language.perl.com`
- `http://language.perl.com/info/documentation.html`
- `http://language.perl.com/CPAN`