CSC 360 Study Guide
Learning Perl, 6th Edition

Chapter 1: Introduction

Excellent overview of Perl and source of Perl resources online. See the simple "Hello World" program: '`print`' is used to produce output and # indicates a comment. In Perl versions 5.10 and later, you can use '`say`' instead of '`print`'; must also include the line '`use 5.010;`' at the beginning of the program.

Chapter 2: Scalar Data

Numbers – all numbers are stored as doubles; arithmetic operators are +, -, *, /, %, **, for addition, subtraction, multiplication, division, modulus, and exponentiation respectively.

Strings – sequences of characters enclosed in either single quotes or double quotes; backslash escapes (e.g. \n or \t, see Table 2.1, p. 26 are interpreted inside double quotes but are treated as standard characters inside single quotes. Perl now supports Unicode, but you must specify 'use utf8;' The string concatenation operator is the period '.' and the string repetition operator is 'x'. Perl automatically converts numbers to strings and strings to numbers as required.

Scalar variables begin with a $ sign, followed by an underscore or letter and then possibly more letters, digits, and underscores.

Assignment is accomplished with = operator; there are also the +=, -=, *= , /=, %=, and .= operators.

Variable interpolation in strings -- Double quoted strings that contain a variable name will have the variable replaced by its value.

See Table 2-2, p. 35-36 for operator precedence and associativity.

The comparison operators are ==, !=, <, <=, >=, and > for numeric values and eq, ne, lt, le, gt, and ge for strings. See Table 2-3, page 37.

The if statement – if (condition) { ... statements …} Curly brackets are required. The 'else' block is optional.

The line input operator -- <STDIN> is used to get input from the keyboard as a string. This typically has a newline character at the end of it that can be removed with the 'chomp' operator.

The while loop – while (condition) { ... statements ... }

Scalar variables have the special value *undef* before they are assigned a value. This will be used as 0 or an empty string in expressions. The *defined()* function can test a variable for the *undef* value.

Chapter 3: Lists and Arrays

A list is an ordered collection of scalars. An array is a variable that contains a list. The individual values can be strings, numbers, or `undef` values, and need not all be of the same type. Subscripts are integers and always begin at 0. Arrays are automatically extended to accommodate previously non-existent elements. The last subscript of an array named `rocks` is given by `$#rocks`. You can remove the tail or extend an array by modifying the value of this variable. You may also use negative subscripts to refer to elements from the end of the array.

List literals contain 0 or more values in parentheses separated by commas, but also may contain references to other variables or ranges of values: `(1, 2..5, 7, "larry")` is the same as `(1, 2, 3, 4, 5, 7, "larry")`. The qw shortcut stands for "quoted words" or "quoted by whitespace", so `("Larry", "Moe", Curly")` is the same as `qw / Larry Moe  Curly/`. The delimiter need not be a `'/'`, but could be almost any punctuation character.

List literals may appear on the right hand side of an assignment and may be used to assign to an array by using the '@' before the array name: `@stooges = qw / Larry Moe Curly/`

The `push` and `pop` operators allow you to add and remove elements at the end of an array, so `$fred = pop( @array )` removes the last element from `array` (permanently) and assigns it to `$fred`, while `push( @array, $a)` would put `$a` as the last element of `array`.

The `shift` and `unshift` operators work on the beginning of an array in the same way that push and pop work on the end of the array.

The `splice` operator allows you to insert and delete elements from the middle of an array.

Entire arrays will be interpolated into quoted strings using @array: if `@array = qw {one two three}`, then `"zero @array end"` will be the same as `"zero one two three end"`.

The `foreach` loop gives a convenient method to iterate over all the elements in a list or an array. If you do not specify a control variable, Perl will use the default variable `$_`.

The `reverse` operator takes a list (or an array that contains a list) and returns the list in reverse order.

The `sort` operator likewise returns a list in sorted (ASCIIbetical) order. This will probably not give the desired results if the list contains numbers.

Perl 5.12 and later allow the use of the `each` operator on arrays, returning the subscript and the value.

VERY important section on Scalar and List context. The name of an array returns a list in a list context and the number of elements in a scalar context. Be sure to read and understand this section and all of the examples in it.

In a scalar context, `<STDIN>` returns the next line of input, while in a list context, `<STDIN>` returns the remainder of all of standard input up to the end-of-file marker.

Chapter 4: Subroutines

Subroutines are defined by the word sub, followed by the name of the subroutine, then the block of code in curly braces which makes up the body of the subroutine. A subroutine is invoked by preceding its name with the '&' symbol. The value of the last expression in the subroutine is the value that is returned. The parameters to a subroutine are stored in the (local) array `@_`, which can be tested to see how many actual parameters were sent. By default, all Perl variables are global in scope. They can be declared local to a subroutine using the `'my'` operator. The return operator can specify a value to be returned, instead of the last expression in the subroutine. A subroutine may be invoked without the preceding '& if the compiler can determine that it really is a subroutine call (the definition came earlier or the parameters are inside parentheses). The '&' is required if your subroutine has the same name as a built-in function. In perl 5.10 and later, we may use the `state` declaration to indicate that a (private) variable should retain its value from one invocation of the subroutine to the next.

Chapter 5: Input and Output

We have seen <STDIN> earlier. A typical loop to process standard input until end-of-file might be:

```
while( defined($line = <STDIN> ) ) { ... process $line ... }
```

Perl gives a simple shortcut :

```
while( <STDIN> ) { ... process $_ ... }
```

or in a list context:

```
foreach( <STDIN> ) { ... process $_ ... }
```

The diamond operator, <>, can be used to process one or more files named on the command line. Technically, the diamond operator gets the name from the @ARGV array. @ARGV is automatically initialized with the list of command-line arguments, if any.

Output can be formatted using *printf*, instead of *print*. This is very similar to *printf* in C and C++.

Filehandles are names for an I/O connection between your program and the outside world, recommended to be in all caps. There are six special filehandle names: `STDIN, STDOUT, STDERR, DATA, ARGV, ARGVOUT`. Input, output redirection can be done with < and > respectively. Output can be piped to another program with `|`.

Opening a filehandle: `open CONFIG, “dino”;` is the same as `open CONFIG, “<dino”;`
both open the file named “dino” for input through the filehandle CONFIG;
`open BEDROCK, “>fred”;` opens “fred” for output and
`open LOG, “>>logfile”;` appends output to the end of existing file “logfile”

The three-argument version of opening a file handle is preferred:

```
open CONFIG, '<', 'dino';
open BEDROCK, '>', 'fred';
open LOG, '>>', 'logfile';
```

also see the discussion of specifying the file encoding.

Closing a filehandle: `close BEDROCK;`

Bad Filehandles: record the status of opens with: `my $success = open LOG, “>>logfile”;`

Now you can test `$success` to see if the open was successful or not. A better way is to use *die*:

Fatal Errors with die: Consider the code:

```
if(  ! open LOG, “>>logfile”) {
     die “Cannot create logfile: $!”;
}
```

This attempts to open the filehandle; if it fails your program terminates, giving the message “Cannot ...” as well as the error message from the operating system stored in the variable $!.

Warning messages: you can use *warn* just like *die* to issue warning messages, but program is not terminated.

Beginning with version 5.10, Perl has an autodie pragma:

```
use autodie;
open LOG, '>>' 'logfile';
```

will automatically cause your program to die and produce diagnostic message.

Using Filehandles: to read from the file “/etc/passwd”:

```
if( ! open PASSWD, “/etc/passwd” ){
      die “Oh Crap, ($!)”;
}
while( <PASSWD> ){
     ... process each line ...
}
```

To write to a file opened for output, using either *print*, or *printf*:

```
print LOG “  .. whatever ...”;
printf STDERR (“%d percent complete.\n”, $done/$total * 100);
```

You can use the *select* statement to change the default output filehandle. Setting the special variable `$| = 1;` forces the output buffer to be flushed after each output operation.

The `say` keyword is similar to `print`, but it adds a newline.

Chapter 6: Hashes

A *hash* is similar to an array, except that the subscripts, called keys, are strings instead of integers, and must be unique. Individual values in a hash are accessed by enclosing the key value in curly braces, similar to square brackets and integer subscripts for arrays. To refer to the entire hash, use the percent sign (`'%'`) as a prefix. When assigning a list to a hash, the values in the list occur in pairs as “key”, “value”, “key”, “value”, etc. The assignment can also use the “big arrow” technique whereby the “key” and value are separated by '=>', instead of a comma. Thus, the list is (key => value, key => value, ...).

The *keys* function applied to a hash returns the list of key values, in no particular order. The *values* function returns the list of values in the same order as the keys function returned. In a scalar context, these functions return the number of entries in the hash. You can iterate over all pairs in a hash using the following loop:

```
while( ($key, $value) = each %hash ) {... body of loop ...}
```

The *exists* function allows you to determine if a particular key value is defined in the hash. The *delete* function will delete the key and its value from the hash.

There is a special hash, `%ENV`, that includes the environment in which your program is executed.

Chapter 7: In the World of Regular Expressions

Regular expressions specify patterns to be used for matching with strings: These can be specific characters enclosed in delimiters (usually /'s), often combined with metacharacters and quantifiers:

dot (.) matches any single character, except \n
backslash (\) is used to 'escape' other metacharacters
star (*) means to repeat the previous item zero or more times
plus (+) means to repeat the previous item one or more times
question mark (?) means the preceding item is optional
parentheses are used to group items
vertical bar (|) means 'or' when it appears between two items

Note that Unicode characters know about themselves and the properties they have. You can test for membership in a property, using `\p{PROPERTY}` where `PROPERTY` is the name of the class. So, to match any kind of space, you use `\p{Space}`.

See discussion of parentheses and *back references* to refer to which parts of the string actually matched the pattern. Perl 5.10 and later allow you to refer to a back reference using `\g{N}`, where `N` is the number of the back reference.

Character class – sequence of characters enclosed in square brackets [], matches any one of these characters; can specify a range: so `[a-zA-Z]` matches any letter. The caret, `^`, negates the class.

Shortcuts: In Perl 5.6 and earlier, only ASCII characters were supported. There were several shortcuts for certain frequently used character classes: `[0-9]` abbreviated as `\d`; `\w` (a "word" character) stands for `[a-zA-Z0-9_]`, `\s` (a "space" character stands for `[\f\t\n\r ]`. These are negated with the `^`, as `[^\d]` matches a non-digit character. The negations have shortcuts: `\D`, `\W`, and `\S` respectively. These shortcuts can still be used in perl 5.14 and later by using the `/a` modifier to indicate that we want the old ASCII interpretation. Perl 5.10 also added some new shortcuts: `\h` for horizontal whitespace, `\v` for vertical whitespace, `\R` for any type of linebreak. Since Unicode is now supported, you should avoid using these shortcuts in new code, but they will be prevalent in older code.

Chapter 8: Matching with Regular Expressions

Matches with m//: matches with / / don't need the 'm'; any other delimiter does need the 'm'.

Match Modifiers: /i means case insensitive match; /s allows newlines to be matched; /x allows you to add whitespace to the pattern to make it more readable; multiple modifiers can be combined; others will be introduced later.

Perl 5.14 adds modifiers to specify character interpretation: /a for ASCII, /u for Unicode, and /l for locale. This can cause some difficulty because Unicode characters don't have a unique "upper case" value, and program execution will differ based upon different values of the locale.

Anchors force a pattern to match at a particular location within the string. The \A anchors at the beginning of the string and \z anchors at the end; \Z anchors at the end but allows for a trailing newline; /m modifier allows for multi-line strings. Word anchors allow matches at the beginning or end of a word (a \w kind of word) \b anchors at the beginning or end of a word; \B is a non-word boundary, i.e. matches wherever \b would not.

Binding Operator, =~: instead of using $_, =~ uses the string on the left for matching to the pattern on the right; looks like an assignment, but isn't.

Interpolating into patterns: A regular expression is double quote interpolated as if it were a double-quoted string.

Match Variables: Capturing Parentheses not only group items, but also tell the regular expression engine to remember what was in the substring that matched what was in the pattern. Thus $1 includes the part of the string that matched the pattern in the first pair of parentheses, $2 contains the matching part from the second pair, etc.

Noncapturing Parentheses: You can follow a left parenthesis with ?: to indicate that it is not a capturing parenthesis.

Named Captures: Perl 5.10 and later allows us to provide labels for the captures and saves the text that matched in a hash named %+, with the label used as the key into the hash.

Automatic match variables: There are three match variables that you always get: $`, $&, and $'. The part of the string that matched a pattern is stored in $&; $` contains what preceded the match and $' contains whatever is after the match.

Quantifiers: A quantifier means to repeat the preceding item a certain number of times, like *, +, ?. So `/a{5,15}/` matches from 5 to 15 a's, `/(fred){3,}/` matches 3 or more repetitions of `fred` and `/\w{8}/` matches exactly 8 word characters. Thus `*` is the same as `{0,}`, the `+` is the same as `{1,}`, and `?` is the same as `{0,1}`.

Precedence of meta-characters: 1. Parentheses used for grouping and capturing. 2. Quantifiers (*, +, ?, and numeric ones with curly braces. 3. Anchors (\A, , \Z, \z, ^, $, \b, \B) and sequences of characters. 4. Vertical bar (|) 5. The atoms: characters, character classes and back references. See Table 8.1, page 151.

The program on page 152 can be used to test potential patterns to verify that they match input strings as expected.

Chapter 9: Processing Text with Regular Expressions

Substitution operator s/ / / functions like a "search and replace." If the closing / is followed by g, then the substitution is global. The /i and /s modify the substitution as before. \U forces the substitution to

uppercase, and \L forces lowercase, \E turns off case shifting, in lowercase, \l and \u, they affect only the next character

The `split` operator breaks up a string based upon a separator, producing a list. The `join` function performs the opposite of split. It takes two parameters: the first is the separator to be inserted between the items in the array (or list) in the second parameter.

When a match m/ / is used in a list context it produces a list of the match variables, or an empty list if no matches occurred.

Non-Greedy Quantifiers: Each greedy quantifier has a non-greedy counterpart, itself followed with a ?.

The /m modifier allows matches to include newlines (think multi-line).

Updating many files: see example in text on page 164-166.

In place editing from the command line: Cool, but fairly obscure. Uses command-line switches for Perl invocation.

Chapter 10: More Control Structures

The *unless* statement is similar to an *if* statement, except that the the statement block will be executed unless the condition is true. It is like using an *if* statement with the condition negated or having the only statements in the *else* block of the *if*. You could have an *else* block with an *unless*, but it's confusing.

The *until* loop is similar to a *while* loop, but with the termination criteria negated:
`until ( $j > $i ) { ... }` is the same as `while( $j <= $i ) { ... }`

Expression Modifiers: *if, unless, until, while, foreach:* These follow other statements, for example:

```
print ""$n is a negative number.\n" if $n < 0;
&error("Invalid input") unless &valid($input);
$i *= 2 until $i > $j;
print " ", ($n +=2) while $n < 10;
&greet($_) foreach @person;
```

Only a single expression is allowed on either side of the modifier.

Naked Block Control Structure: A sequence of statements inside curly braces. Variables declared within it are local to it.

Elsif clause Similar to Ada or Visual Basic; avoids embedding *if / else* statements inside *else* blocks

Autoincrement and Autodecrement: `++` and `--` operators just like C++; both pre- and post- operators.

The for Control Structure: for(initialization; test; increment) { loop body }

Loop Controls: *last* (exits the loop, like *break* in C++); *next* (jumps to the bottom of the loop, like *continue* in C++); *redo* (go back to the top of the loop block); *Labeled blocks* (attach labels to loop blocks to identify them, usually all caps; used to refer to that loop with *last*, *next*, etc.)

Ternary operator, ?: expression ? if_true_expr : if_false_expr

Logical operators: && means AND; || means OR; if the left side of one of these determines the entire value of the expression, then the right side is not evaluated, i.e., it is short circuited.

Perl 5.10 provides the defined-or operator //

Chapter 11: Perl Modules

This chapter will not be on the test. It is straight forward to read and comprehend.

Chapter 12: File Tests

This chapter will not be on the test.

See Table 12-1, p. 204-205 for all of the possible file tests and their meaning.

Perl has a built-in virtual filehandle, _, (an underscore) that contains all of the information about the last executed file test, so that _ can be used for additional tests on the same file to increase efficiency. Starting with Perl 5.10, you can "stack" the file tests.

The *stat* function will give lots of details about a file. The *lstat* function gives the same information about a link. The *localtime* function will convert a timestamp integer into a human readable format.

Bitwise operators: & is bitwise 'and', | is bitwise 'or', ^ is bitwise 'xor', << is bitwise shift left, >> is bitwise shift right, ~ is bitwise negation (complement)

Chapter 13: Directory Operations

This chapter will not be on the test.

The *chdir* command changes the working directory, just like the shell *cd* command, for example:

```
chdir "/etc" or die "Cannot chdir to /etc: $!";
```

Globbing is the action of the shell expanding filename patterns into matching filenames. The *glob* operator will do this inside your program, as a list, expanded just like the shell would. An alternate syntax is using the angle brackets <> instead of *glob:* `@all_files = <*>;` is the same as `@all_files = glob "*";`

Directory handles are like filehandles opened with *opendir* instead of *open,* read from it with *readdir* instead of *readline*, and close it with *closedir.* Reading from a directory handle reads the names of all of the files in that directory (including dot-files), in no particular order.

If you need to process directories recursively, use the `File::Find` library.

unlink operator will remove files, for example:

```
unlink “one”, “two”; or unlink glob “*.o”;
```

rename operator is used to rename files:

```
rename “old”, “new”;
```

Excellent discussion of directories, inodes, hard and soft links in Unix file systems. Use the *link* function to create hard links; the *symlink* function to create symbolic or soft links; the *readlink* function to read a soft link; and the *unlink* function to remove both hard and soft links.

mkdir function creates a new directory within an existing one, for example:

```
mkdir “fred”, 0755 or warn “Cannot make fred directory: “ $!;
```

rmdir is similar to *unlink* and is used to remove **empty** directories.

chmod changes file permissions, just like shell command (permissions should be in **octal**).

chown changes file ownership and group membership, for example:

```
chown 1004, 100, glob “*.o”;
```

or:

```
chown getpwnam(“merlyn”), getgrnam(“magic”), glob “*.o”;
```

time function returns current system time; *utime* function changes the various times associated with a file

Chapter 14: Strings and Sorting

The *index* function will search one string for the occurrence of another, returning the location where it begins or -1 if it is not found. It will report the location of the first occurrence that it finds, but you can specify where to start the search as a third parameter. The *rindex* function returns the last occurrence; it's optional third parameter effectively gives the **maximum** return value.

The *substr* operator is used to extract a substring from a larger string:

```
$part = substr( $string, $initial_position, $length);
```

If the requested length is too large, just the remainder of the string is returned; you can also omit the third parameter to force returning the remainder of the string. If the starting location is negative, then the counting is from the end rather than the beginning.

The *substr* function can also be used to replace a portion of a string:

```
my $string = “Hello, World”;
substr( $string, 0, 5) = “Goodbye”;
```

Now, `$string` contains `“Goodbye, World”`

There is also a version of the *substr* operator with the fourth parameter as the replacement string:

```
my $new_value  = substr( $string, 0, 5, “Goodbye”);
```

The *sprintf* function works just like *printf*, except that its output goes into a string.

Advanced Sorting: You can direct Perl to sort in some order other than ASCIIbetical order by setting up a *sort-definition subroutine* or *sort subroutine,* for short. Essentially, all this subroutine has to do is specify the order of two values `$a` and `$b`. If `$a` should precede `$b`, the subroutine should return `-1`; if `$a` should follow `$b`, then it should return `1`; if it doesn't matter, then it returns `0`. Thus, for a numeric sort:

```
sub by_number {
     #sort subroutine, thus assume $a & $b are pre-loaded
     if( $a < $b ) { -1 } elsif ( $a > $b ) { 1 } else { 0 }
}
```

This is used this way: `my @result = sort by_number @some_numbers;`

The *spaceship* operator, `<=>`, is Perl's shorthand for a numeric comparison:

```
sub by_number { $a <=> $b }
```

The three-way string-comparison operator is *cmp*. To create a case-insensitive string sort, use:

```
sub case_insensitive { "\L$a" cmp "\L$b" }
```

For Unicode strings, you probably want to use:

```
use Unicode::Normalize;
sub equivalents { NFKD($a) cmp NFKD($b) }
```

For simple sort subroutines like these, you can include them in-line, without declaring a subroutine:

```
my @numbers = sort { $a <=> $b } @some_numbers;
```

For a descending order numerical sort, you can use *reverse*:

```
my @numbers = reverse sort { $a <=> $b } @number_array;
```

or:

```
my @numbers = sort { $b <=> $a } @number_array;
```

To sort the keys in a hash, say `%score`, based upon the descending numerical values stored in the hash:

```
sub by_score { $score{ $b } <=> $score{ $a }
```

To sort the keys in a hash ASCIIbetically in the event of numerical ties, you can define:

```
sub by_score_and_name {
     $score{ $b } <=> $score{ $a } # by descending score
          or
     $a cmp $b           # ASCIIbetically, if tied numerically
}
```

See the example of a five-way sort of an array of library patron ID numbers using another subroutine, and three different hashes to resolve ties along the way.

Chapter 15: Smart Matching and given-when

This material will not be on the test and is specific to Perl version 5.10 and later.

The Smart Match operator, ~~, provides an easy way to perform simple comparisons as well as regular

expression matching, all with the same syntax. There are several examples and Table 15.1, page 250, gives the priority and type of match performed, based upon the type of operands.

The given-when statement is Perl's version of the switch statement in C/C++/C#/Java. See the examples.

Chapter 16: Process Management

This material will not be on the test.

system function launches a child process; it inherits STDIN, STDOUT, STDERR from parent, e.g.:

```
     system "date"; #sends current date and time to STDOUT
or:  system 'ls -l $HOME'          #note single quotes
```

Perl waits for the child process to complete before resuming execution, unless it is launched as background process in the shell with a trailing '&'. You should read the section about avoiding invocation of a separate instance of the command shell.

Shell environment variables, like PATH, are available through the special hash %ENV, which can be read or modified to be passed on to a child process.

exec function causes the Perl process itself to perform the action instead of creating a child process. When that command or program finishes, there is no Perl process to return to, so control is usually returned to a command prompt.

You can use **backquotes** to capture the output of another command, for example:

```
     my $now = `date`;   #gets "date" output with newline
```

to capture the command's STDERR output as well:

```
     my $all_output = `doit  -enable 2>&1`; #intermixes STDOUT & STDERR
```

Backquoted commands used in a scalar context return a single string, possibly including several newlines. In a list context, backquoted commands return a list containing one line of output (still including newline) per list element. For comparison:

```
     my $who_text = `who`;    #multiple lines in a single string
     my @who_lines = `who`;   #one line per array entry
```

You can use a process as a filehandle by using the *open* function with the pipe symbol, referred to as a *"piped open"*. For example:

```
     open DATE, "date |" or die "cannot pipe from date: $!";
or:  open MAIL, "| mail merlyn" or die "cannot pipe to mail: $!";
```

The first example is similar to : `date | your_program`
The second example is similar to: `your_program | mail`

Now, you can read from it just like a from a file: `my $now = <DATE>;`

```
or:  print MAIL "The time is now $now";
```

There is a rather esoteric section about *fork* and *waitpid* dealing with the fork system call and process ids.

You can send signals to processes with the *kill* function. You can receive signals through the special hash %SIG, which associates the name of a handler subprogram with the name of the signal that it is meant to process. See the example in the book.

Chapter 17 Some Advanced Perl Techniques

This material will not be on the test.