A. Albright

Class 1: Introduction

(1) Getting software

- Text editors
  - AlphaX for Mac: http://www.maths.mq.edu.au/~steffen/Alpha/AlphaX/
  - SciTE for Windows: http://scintilla.sourceforge.net/SciTEDownload.html
- Perl for Windows:
  - http://www.activestate.com/Products/ActivePerl/

(2) Navigating the command line (a few basics)

Function	Unix	DOS
Change directories	`cd` *destination*	`cd` *destination*
Go up one level	`cd ..`	`cd ..`
Print current directory	`pwd`	`cd`
List files in current directory	`ls`	`dir`
Display a text file	`more` *filename*	`type` *filename*
Run a Perl program	`perl` *filename*	`perl` *filename*

- More at: http://ist.uwaterloo.ca/ec/unix/comparison.html

(3) hello1.pl

```
print "Hello world!\n";
```

(4) hello2.pl

```
$greeting = "Hello world!";
print "$greeting\n";
```

(5) hello2b.pl

```
$world = "Hello";
$hello = "world!";
print "$world $hello\n";
```

(6) hello3.pl

```
$greeting[0] = "Hello";
$greeting[1] = "world!";
# The following two lines do exactly the same thing
print "$greeting[0] $greeting[1]\n";
print "@greeting\n";
```

(7) hello3b.pl

```
@greeting = ("Hello", "world");
# The following two lines do exactly the same thing
print "$greeting[0] $greeting[1]\n";
print "@greeting\n";
```

(8) simplemath.pl

```
$x = 1;
print "The value of \$x is $x\n";
$x = $x + 2;
print "The value of \$x is $x\n";
$x = $x * 2;
print "The value of \$x is $x\n";
```

```
$x = $x / 3;
print "The value of \$x is $x\n";
$x = $x - 1;
print "The value of \$x is $x\n";
$x++;
print "The value of \$x is $x\n";
$x--;
print "The value of \$x is $x\n";
```

(9) Concatenating text:

```
$greeting = "Hello" . " " . "world!";
```

(10) loop1.pl

```
# A for loop from 1 to 10
for ($i = 1; $i < 11; $i++) {
     print "$i\n";
}
```

(11) Syntax: `for (initial state, condition, operation) { ...}`

- Here, initial state is for $i to have value of 1
- Condition is to keep going as long as $i is less than 11
  - `x < y` means x is less than y
  - `x <= y` means x is less than or equal to y
  - Similarly, `x > y`, `x >= y` for x greater than (or equal to) y
  - `x == y` means x equals y
- Each time we run the loop, we add one to $i (`$i++`)
- The stuff to do is between curly braces: { ... }

(12) hello4.pl

```
@greeting = ("Hello", "world!");
for ($i = 0; $i <= 1; $i++) {
     print "$greeting[$i] ";
}
print "\n";
```

(13) hello5.pl

```
@greeting = ("Hello", "world!");
for ($i = 0; $i <= $#greeting; $i++) {
     print "$greeting[$i] ";
}
print "\n";
```

(14) cv.pl

```
@consonants = ('p','t','k','b','d','g','f','s','z','m','n','l','r');
@vowels = ('a','e','i','o','u');
# Let's also keep track of how many words we have generated
$number_of_words = 0;
# Loop through consonants
for ($c = 0; $c <= $#consonants; $c++) {
     # Loop through vowels
     for ($v = 0; $v <= $#vowels; $v++) {
          # Print out this CV combination
          print "$consonants[$c]$vowels[$v]\n";
          # Add one to the number of words
          $number_of_words++;
     }
}
print "\nGenerated a total of $number_of_words words\n";
```

(15) cvcv.pl

```
@consonants = ('p','t','k','b','d','g','f','s','z','m','n','l','r');
@vowels = ('a','e','i','o','u');
$number_of_words = 0;
for ($c1 = 0; $c1 <= $#consonants; $c1++) {
   for ($v1 = 0; $v1 <= $#vowels; $v1++) {
      for ($c2 = 0; $c2<= $#consonants; $c2++) {
         for ($v2 = 0; $v2<= $#vowels; $v2++) {
            print "$consonants[$c1]$vowels[$v1]$consonants[$c2]$vowels[$v2]\n";
            # Add one to the number of words
            $number_of_words++;
         }
      }
   }
}
print "\nGenerated a total of $number_of_words words\n";
```

(16) Control structures

- `if (`*condition*`) { ...}`
- `if (`*condition*`) { ...}`
  `else { ...}`
- `if (`*condition*`) { ...}`
  `elsif (`*condition*`) { ...}`
  `else { ...}`
- `unless (`*condition*`) { ...}`

Conditions:

`$x == $y`	x equals y (numeric)
`$x != $y`	x doesn't equal y (numeric)
`$x eq $y`	x equals y (strings)
`$x ne $y`	x doesn't equal y (strings)

(Also `$x > $y`, `$x < $y`, `$x >= $y`, `$x <= $y` for numbers)

(17) cvcv2.pl

```
@consonants = ('p','t','k','b','d','g','f','s','z','m','n','l','r');
@vowels = ('a','e','i','o','u');
$number_of_words = 0;
for ($c1 = 0; $c1 <= $#consonants; $c1++) {
   for ($v1 = 0; $v1 <= $#vowels; $v1++) {
      for ($c2 = 0; $c2<= $#consonants; $c2++) {
         for ($v2 = 0; $v2<= $#vowels; $v2++) {
            if ($c1 eq $c2) {
               print "*$consonants[$c1]$vowels[$v1]$consonants[$c2]$vowels[$v2]\n";
            } else  {
               print "$consonants[$c1]$vowels[$v1]$consonants[$c2]$vowels[$v2]\n";
               # Add one to the number of words
               $number_of_words++;
            }

         }
      }
   }
}
print "\nGenerated $number_of_words legal words\n";
```

(18) Pattern matching:

```
if ($mystring =~ /searchstring/) { ... }
```

A few things to learn as you need them:

- `[ab]` means "either a or b" (a, b); this can be expanded, so `[abc]` = either a, b, or c, etc...
- `[^a]` means "anything other than a"; `[^ab]` means "anything other than an a or a b", etc. (set negation)
- `a*` means "any number of a's (from 0 to infinity)" (nothing, a, aa, aaa, aaaa, aaaaa, ...)
- `a+` means "one or more a's" (a, aa, aaa, aaaa, aaaaa, ...)
- `ab+` means "an a, followed by one or more b's" (ab, abb, abbb, abbbb, ...)
- `(ab)+` means "one or more consecutive occurrences of ab" (ab, abab, ababab, abababab, ...)
- `a?` means "an optional a"
- `^a` means "an a at the beginning of the string"
- `a$` means "an a at the end of the string"
- `.` (period) means "any character"

Also:

`\w`	Matches a "word" character (alphanumeric plus "_")
`\W`	Matches a non-word character
`\s`	Matches a whitespace character
`\S`	Matches a non-whitespace character
`\d`	Matches a digit character
`\D`	Matches a non-digit character
`\b`	Matches a word boundary
`\B`	Matches a non-(word boundary)

More information can be found at:

- http://www.wdvl.com/Authoring/Languages/Perl/PerlfortheWeb/perlintro2_table1.html
- http://etext.lib.virginia.edu/helpsheets/regex.html
- http://www.perldoc.com/perl5.6/pod/perlre.html

(19) patternmatch.pl

```
if ("blah" =~ /a/) { print '/a/' . "\n"; }
if ("blah" =~ /^a/) { print '/^a/' . "\n"; }
if ("blah" =~ /ba/) { print '/ba/' . "\n"; }
if ("blah" =~ /b.a/) { print '/b.a/' . "\n"; }
if ("blah" =~ /[a-h]*/) { print '/[a-h]*/' . "\n"; }
if ("blah" =~ /^[a-h]*$/) { print '/^[a-h]*$/' . "\n"; }
if ("blah" =~ /[a-m]*/) { print '/[a-m]*/' . "\n"; }
if ("blah" =~ /^[a-m]*$/) { print '/^[a-m]*$/' . "\n"; }
```

(20) cvcv3.pl

```
@consonants = ('p','t','k','b','d','g','f','s','z','m','n','l','r');
@vowels = ('a','e','i','o','u');
$number_of_words = 0;
for ($c1 = 0; $c1 <= $#consonants; $c1++) {
   for ($v1 = 0; $v1 <= $#vowels; $v1++) {
      for ($c2 = 0; $c2<= $#consonants; $c2++) {
         for ($v2 = 0; $v2<= $#vowels; $v2++) {
            $word = "$consonants[$c1]$vowels[$v1]$consonants[$c2]$vowels[$v2]";
            unless ($word =~ /$consonants[$c1].$consonants[$c1]/) {
               print "$word\n";
            }
         }
      }
   }
}
```

(21) cvcv4.pl

```
@consonants = ('p','t','k','b','d','g','f','s','z','m','n','l','r');
@vowels = ('a','e','i','o','u');
$number_of_words = 0;
for ($c1 = 0; $c1 <= $#consonants; $c1++) {
   for ($v1 = 0; $v1 <= $#vowels; $v1++) {
      for ($c2 = 0; $c2<= $#consonants; $c2++) {
         for ($v2 = 0; $v2<= $#vowels; $v2++) {
            $word = "$consonants[$c1]$vowels[$v1]$consonants[$c2]$vowels[$v2]";
            if ($word =~ /$consonants[$c1].$consonants[$c1]/) {
               print "$word\tC1=C2\n";
            } elsif ($word =~ /$vowels[$v1].$vowels[$v1]/) {
               print "$word\tV1=V2\n";
            } elsif ($word =~ /[pbmf].[pbmf]/) {
               print "$word\tTwo labials\n";
            } elsif ($word =~ /[iu]$/) {
               print "$word\tFinal high vowel\n";
            } else { print "$word\n"; }
         }
      }
   }
}
```

(22) readfile1.pl

```
#Read a file, print its line to the screen.
$input_file = "sample.txt";
open (INFILE, $input_file) or die "The file $input_file could not be found\n";

# Loop, continuing as long as lines can be read from the file
while ($line = <INFILE>)
{
  $line_count++;
  print "$line_count  $line";
}

close INFILE;
```

(23) What should this do? (and what is the problem?)

readfile3.pl

```
$input_file = "sample.txt";
$output_file = "sample-output.txt";

open (INFILE, $input_file) or die "The file $input_file couldn't be found\n";
open (OUTFILE, ">$output_file") or die "The file $output_file couldn't be written\n";

# Loop, continuing as long as a line can be read successfully from the file
while ($line = <INFILE>)
{
    $count = 0;
    $lines++;
    while ($line =~ m/[aeiou]/) {
        $count++;
    }
    print "Line $lines: $count vowels\n";
}

close INFILE;
close OUTFILE;
```

(24) Other useful operations

`chomp($x)`	removes newline (`\n`) from end of line
`lc($x)`	converts $x to lower case
`@fields = split(/\t/, $x)`	splits string $x into an array, using tab as a delimiter
`($var1, $var2) = split(/\t/, $x)`	assigns split fields to different variables
`$x =~ s/search/replace/`	searches $x for `search` and replaces with `replace` (1st instance only)
`$x =~ s/search/replace/g`	searches $x for `search` and replaces with `replace` (all instances)

(25) checkmath.pl

```
# This script reads in a series of arithmetic statments,
# and checks whether they are correct
# It is extremely limited, in that it only handles statements with 2 operands

$input_file = "math.txt";
open (INFILE, $input_file) or die "Can't open input file: $!\n";

$correct_answers = 0;
$incorrect_answers = 0;

CHECK_ANSWER:
while ($line = <INFILE>) {
    chomp($line);
    # We'll assume that statements have the form:
    #      x OPERATION y = z
    # So, let's first start by getting the left sides and result.
    # We can split at the equal sign (removing also any spaces around it
    ($operation, $given_answer) = split(/\s*=\s*/,$line);

    # Now, parse out the operation, so we can check it.  We want to split at a +, -, * or /
    # Since these are all "special" symbols in regular expression syntax,
    # we need to "protect" them by putting a backslash before each.
    # As before, we also include the spaces (\s*) as part of the delimiter
    @operands = split(/\s*[\+\-\*\/]\s*/, $operation);

    if ($operation =~ /\+/) {
        $operator = "plus";
        $real_answer = $operands[0] + $operands[1];
    } elsif ($operation =~ /\-/) {
        $operator = "minus";
        $real_answer = $operands[0] - $operands[1];
    } elsif ($operation =~ /\*/) {
        $operator = "times";
        $real_answer = $operands[0] * $operands[1];
    } elsif ($operation =~ /\//) {
        $operator = "divided by";
        $real_answer = $operands[0] / $operands[1];
    } else {
        # If we got here, there was no +, -, *, or / found
        print "Error! The operation $operation was not found to have an operator\n";
        next CHECK_ANSWER;
    }

    if ($real_answer == $given_answer) {
        $correct = 1;
        $correct_answers++;
    } else {
        $correct = 0;
        $incorrect_answers++;
    }
    print "$operands[0] $operator $operands[1] is $real_answer\t";
```

```
    print "\t(The given answer of $given_answer is ";
    unless ($correct) { print "NOT "; }
    print "correct)\n";
}

# We are now done with the file, and can calculate summary statistics.
print "\nTotal of $correct_answers correct answers, and $incorrect_answers incorrect answers.\n";
print "\t(Overall score: ".($correct_answers*100 / ($correct_answers+$incorrect_answers))." percent)\n";
```

(26) hepburn.pl

```
$input_file = "Japanese-ToConvert.txt";
open (INFILE, $input_file) or die "Warning! Can't open input file: $!\n";

while ($line = <INFILE>) {
    # Crucial rule ordering: this needs to go first
    $line =~ s/hu/fu/g;

    # The major difference is use of <y> after t,s,z
    $line =~ s/ty/ch/g;
    $line =~ s/sy/sh/g;
    $line =~ s/zy/j/g;
    # Also, palatalization before i
    $line =~ s/ti/chi/g;
    $line =~ s/si/shi/g;
    $line =~ s/zi/ji/g;
    # And assibilation of t before u
    $line =~ s/tu/tsu/g;

    print "$line";
}
```