Intro to Perl

Practical Extraction and Reporting Language
CIS 218
Perl Syntax

• Perl is an “interpretive” script language. As opposed to BASH, CSH etc which are interactive. Perl actually “precompiles perl code into an interim binary format.

  - Perl is case sensitive
  - All Perl statements end in semicolon “;”, except code blocks.

• Default output to STDOUT with the “print” command:

  - print “Hi”, “there”, “\n”;
  - “say” command is the same as “print” except it automatically inserts an <LF> or “\n” at the end of output. Requires use of “use v5.10;” (or similar depending on release) statement.
    
    use v5.10;
    say “Hello World”;

• Perl scripts begin with Perl “magic” statement:

    #!/usr/bin/perl –w
    use v5.10;
    print “Hello World\n”;
    say “Hello World”;

• Can invoke Perl commands directly from command line as follows:
  perl -w -e 'print "Hello World\\n"
perl -w -e 'use v5.10; say "Hello World""
Scalars

- SCALARS - mathematical term meaning single value.
  - Scalar – a single value variable
  - Scalar types – numbers, strings

- VARIABLES: Named location in memory that holds a value(s).
  - Variable names are case sensitive in Perl, always preceded by “$”.
    (i.e. no lefty/right rule)
  - Variable assignment “=”.  
  - Example: \$variable=value;

- If you use a scalar that is undefined (or undef’ed), perl will stringify or numify it based on how you are using the variable:

  An undefined scalar stringifies to an empty string: 
  An undefined scalar numifies to zero: 0
Numbers

• NUMBERS – usual notation as in algebra, parentheses can be used to change standard operator precedence order.

• Numeric operators:
  + (add), - (subtract),
  * (multiply), / (divide),
  ** (exponent), % (modulus),
  ++ (increment), -- (decrement)

• $radius = 50;
  $pi = 3.141592;
  $area = $pi*($radius**2);
  print $area;

• $counter=10;
  $counter--;
  print $counter; .... Prints 9
Strings

• Usually enclosed in double quotes so it is treated as a “word” otherwise can cause problems with commands requiring a single word syntax.

```perl
$name="Fred Flinstone";
p$name; .... Prints “Fred Flinstone”
```

• Parentheses used to enforce order and assign a list of words.

• String assignment and concatenation (using “.”) is simple string substitution:

```perl
#!/usr/bin/perl -w
($firstname, $middleinitial, $lastname) = ("Fred ", "W", "Flinstone");
p$firstname, $middleinitial, $lastname;
# String concatenation
$name=$firstname.$middleinitial.$lastname;
p$name;
```

• Default Variable $_ - referenced by Perl if no explicit variable name specified

```perl
#!/usr/bin/perl –w
$_="Yabba dabba doo!!";
p$;
Quoting Strings

- Single quotes can also be used but suppresses variable substitution same as Bourne Shell:

```perl
$name="Fred Flinstone";
print $name;  .... Prints “Fred Flinstone”
print ‘$name’; .... Prints “$name”
```

- \ can be used to quote single characters

- Also can use for specifying alternate delimiters such as forward slash “/”, parentheses “(, “)” or curly braces “{]” :

  - q (single quote, suppresses variable substitution)
  - qq (double quote, allows variable substitution)
  - qw (quote a word)
  - qx – same as “backticks” or command substitution

q, qq are used for a list, qw for single word
@q = qw/this is a test/ is the same as @q = (‘this’, 'is', 'a', 'test')

```perl
perl -e ‘$name=qw/Fred Flinstone/; print $name."\n";’
perl -e ‘$name=q/Fred Flinstone/; print $name."\n";’
perl -e ‘$name=qq/Fred Flinstone/; print $name."\n";’
```
Arrays

- **Array** – A named list of variables usually indexed by a value. @ sign starts array variables.

- You use the equals (=) sign to assign values to array variables just like scalar values.

- Individual Arrays items are indexed by number starting with 0 and referenced as a scalar ($).

```perl
@emptyArray = ();
@numberArray = (12, 014, 0x0c, 34.34, 23.3E-3);
@stringArray = ("This", "is", 'an', "array", 'of', "strings");
@mixedArray = ("This", 30, "is", 'a', "mixed array", 'of', 0x08, "items");
print @emptyArray 
; print @numberArray; print 
; print @stringArray; print 
; print @mixedArray; print 
;

$array = (1..5);
print @array; print 
;
print $array[0]; print 
; print $array[1]; print 
; print $array[2]; print 
; print $array[3]; print 
; print $array[4]; print 
;

@smallArrayOne = (5..10); .. Is range operator
@smallArrayTwo = (1..5);
@largeArray = (@smallArrayOne, @smallArrayTwo);
print @largeArray;
```

- Default array @ _ - referenced by Perl if no explicit array
Hashes

- **Associative Array Variables (hash):** a hash is an array using a non-numeric index (KEY). The term "Hash" refers to how associative array elements are stored in memory.

- Associative array names start with the % character. Is actually a paired list in the form of: ("key", "scalar value"). Or by using the "value"=>"key" list construct.

- An internal table is used to keep track of which keys are defined. If you try to access an undefined key, Perl will return a null or blank string.

- Lists are dynamically extended by Perl. Perl will extend the associative array as needed when you assign values to keys as a list or singly as a scalar.

```perl
%associativeArray = ("Dec 2" =>"Jack A.", "June 2"=>"Joe B.", "Feb 13"=>"Jane C.");
%associativeArray = ("Jack A.", "Dec 2", "Joe B.", "June 2", "Jane C.", "Feb 13");
$associativeArray{"Jennifer S."} = "Mar 20";

print "Joe's birthday is: " . $associativeArray{"Joe B."} . "\n";
print "Jennifer's birthday is: " . $associativeArray{"Jennifer S."} . "\n";
```

- The key is specified as the first value in the paired list. The second value is the value returned on reference. Individual Arrays items are indexed by the non-numeric key and referenced as a scalar ($). The keys directive can be used to extract the list of keys from an associative array.

```perl
- %pets = ( fish=>3,cats=>2,dogs=>1,);
  foreach my $pet (keys(%pets))  {print "pet is '$pet'\n";}
```

- As with other variables, the hash has a default value referenced by Perl if no explicit array @_.

User Input

- INPUT from Command Line: $variable=<STDIN>;
  Gets keyboard input up to Return key <LF> and assigns to $variable

- chomp drops <LF> from $variable

```perl
#!/user/bin/perl -w
print "Enter Shoe Size";
<size>=<STDIN>;
chomp $size;
print "Your shoe size is $size \n";
```

- Alternative example:

```perl
perl -e 'print "Enter Shoe Size:"; chomp ($size=<STDIN>); print "your shoe size is $size\n";'
```

- Note the combined function of chomp and STDIN – the first command in parentheses to change order of execution.

- chop is related to chomp except it removes ANY trailing character from a string.

- Example reading and writing:

```perl
while (<STDIN>) { print($_); }
while ($inputLine = <STDIN>) { print($inputLine); }
```

... Note STDOUT is default output destination

Note: can be used by redirecting a file from the command line: perl -w myperl < someinputfile > someoutoutfile
Conditions, statement blocks, local variables

- All code terminated in {} are statement “blocks”, standalone blocks of code
  
  ```
  { statement1;
    statement2;
    statement3; }
  ```

- Conditional tested within parentheses for `if`, `when` or `until`. If true block statements are executed:
  ```
  if (condition) { commands; }
  while (condition) { commands; }
  until (condition) { commands; }
  ```

- Statement block also specifies scope of local variables defined by `my`. You can (but don’t have to) declare a variable before using it, the most common way is with the `my` function. `my` simultaneously declares the variables and limits their `scope` (the area of code that can see these variables) to the enclosing code block:
  ```
  my ($radius) = 50;
  my ($pi) = 3.141592;
  my $area = $pi*($radius**2);
  print $area;
  ```
Conditional Comparisons

- **Function**                      **String**     **Numeric**
  equal to                      eq          ==
  not equal                    ne          !=
  less than                    lt          <
  greater than                 gt          >
  less than or equal to        le          <=
  greater than or equal to     ge          >=
  comparison (<=, ==, >)      cmp          <=>

- If you use a string compare for two numbers, you will get their alphabetical string comparison. Perl will stringify the numbers and then perform the compare. E.G. when you compare the strings ("9" lt "100"). String "9" is greater than (gt) string "100". Number 9 is less than (<=) number 100.

- If you use a numeric operator to compare two strings, perl will attempt to numify the strings and then compare them numerically. Comparing "John" <= "Jacob" will cause perl to convert "John" into a number and fail.
Conditional logic

- The higher precedence logical operators '&&', '||', and '!' operators.

<table>
<thead>
<tr>
<th>function operator</th>
<th>usage</th>
<th>return value</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND</td>
<td>$one &amp;&amp; $two</td>
<td>if ($one is false) $one else $two</td>
</tr>
<tr>
<td>OR</td>
<td>$one</td>
<td></td>
</tr>
<tr>
<td>NOT</td>
<td>! $one</td>
<td>if ($one is false) true else false</td>
</tr>
</tbody>
</table>

- The lower precedence logical operators 'and', 'or', 'not', and 'xor' operators.

<table>
<thead>
<tr>
<th>function operator</th>
<th>usage</th>
<th>return value</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND</td>
<td>$one and $two</td>
<td>if ($one is false) $one else $two</td>
</tr>
<tr>
<td>OR</td>
<td>$one or $two</td>
<td>if ($one is true) $one else $two</td>
</tr>
<tr>
<td>NOT</td>
<td>not $one</td>
<td>if ($one is false) true else false</td>
</tr>
<tr>
<td>XOR</td>
<td>$one xor $two</td>
<td>if ( ($one true and $two false) or ($one false and $two true))</td>
</tr>
</tbody>
</table>

- Strings "" and "0" are FALSE, any other string or stringification is TRUE
- Number 0 is FALSE, any other number is TRUE
- all references are TRUE
- undef is FALSE
Functions

- **Functions:** (or subroutines) are blocks of codes that are given names so that you can use them repeatedly as needed. Functions place code into pieces called *modular programming*.

- A function definition is very simple. It consists of:
  ```perl
  sub functionName {  block of code }
  ```

- Subroutines are usually called on the right side to return a calculated value using `return`. Values are passed to the subroutine using the default array `@_`, and are references as members of the array `$_[index]`.

For instance, if your program has a function that calculates the area of a circle the following line of code might be used to call it. Inside the `areaOfCircle()` function, the parameter array is named `@_`. All parameters specified during the function call are stored in the `@_` array so that the function can retrieve them. The `@_` array is used like any other array. Individual array items are referenced as scalars `$_[index]`:

```perl
$areaOfFirstCircle = areaOfCircle($firstRadius);

$radius = $_[0];
$areaOfFirstCircle = areaOfCircle(5);
print("$areaOfFirstCircle\n");
```

```perl
sub areaOfCircle {  $radius = $_[0];  return(3.1415 * ($radius ** 2));   }
```
Non-sequential Control Statements

- **The If Statement**

  The syntax for the if statement is the following:

  ```
  if (CONDITION) {     # Code block executed    # if condition is true.
  elsif (CONDITION_TWO {    # Code block executed    # if condition two is true.
  else {    # Code block executed    # if condition(s) are false.

  Also has a “natural language” single command format:
  {single command} if (CONDITION);
  ```

  ```
  if ($word1 eq $word2) {
    print "match\n";
  } else {
    print "No match\n";
  }
  ```
Non-sequential Control Statements

• **While Loops:** repeat a block of statements while some condition is true. There are two forms of the loop:

1) where the condition is checked before the statements are executed (the do..while loop)

   ```
   do { STATEMENTS } while (CONDITION);
   ```

2) where the condition is checked after the statements are executed (the while loop)

   ```
   while (CONDITION) { STATEMENTS }
   continue { STATEMENTS }
   ```

   ```
   $\text{firstVar} = 0;
   do {
      print("inside: firstVar = $\text{firstVar}\n");
      $\text{firstVar}++;
   } while ($\text{firstVar} < 2);
   ```
Non-sequential Control Statements

- **Until Loops**: loops are used to repeat a block of statements while some condition is false. There are two forms of the until loop:

1) one where the condition is checked after the statements are executed (the do...until loop).

   do { STATEMENTS } until (CONDITION);

2) one where the condition is checked before the statements are executed (the do...until loop)

   until (CONDITION) { STATEMENTS };

```perl
$firstVar = 10;
until ($firstVar > 20) {
    print("inside: firstVar = $firstVar\n");
    $firstVar++;
}
print("outside: firstVar = $firstVar\n");
```
Non-sequential Control Statements

• **For Loops:** looping a specific number of times.

```plaintext
for (INITIALIZATION; CONDITION; INCREMENT/DECREMENT) {
    STATEMENTS;
}

for ($firstVar = 100, $secondVar = 0; $firstVar > 0; $firstVar--, $secondVar++) {
    print("inside: firstVar = $firstVar  secondVar = $secondVar\n");
}
Non-sequential Control Statements

- **Foreach loops** are used to iterate commands on each element of an array (list). Same as for loop in csh, bash.

```
foreach LOOP_VAR (ARRAY) { STATEMENTS}
```

- The loop variable is assigned the value of each array element, in turn until the end of the array is reached. Let's see how to use the foreach statement to find the largest array element.

```
sub max {
    my($max) = shift(@_);

    foreach $temp (@_) {
        $max = $temp if $temp > $max;
    }
    return($max);
}
```

```
print max(45..121, 12..23) . "\n";
print max(23..34, 356..564) . "\n";
```
Loop Control

- **last Keyword:** is used to exit from a statement block. This ability is useful if you are searching an array for a value. When the value is found, you can stop the loop early.

- **next Keyword:** lets you skip the rest of the statement block and start the next iteration. One use of this behavior could be to select specific array elements for processing and ignoring the rest.

- **redo Keyword:** causes Perl to restart the current statement block. Neither the increment/decrement expression nor the conditional expression is evaluated before restarting the block. This keyword is usually used when getting input from outside the program, either from the keyboard or from a file. It is essential that the conditions that caused the redo statement to execute can be changed so that an endless loop does not occur.

- **goto Keyword:** lets your program jump directly to any label. This is bad programming (“spaghetti code”).
There are four basic operations that you can do with files: OPEN, CLOSE, READ, WRITE.

OPEN

```
open(filehandle, filepathname) || die "Cannot open file $!
; ... logical OR to cancel prog with error
```

```
on(myfile, "c:\windows\system32\sometextfile.txt");
on(myfile, "/home/user/somefile.txt");
```

READ (<>)

```
on(myfile, "somefile");
while (defined($line=<myfile>)) { print $line; } ... record at a time w/variable
while (<myfile>) { print; } ... record at a time w/default variable
@file=<myfile>; ... entire file as an array, subject to array size and memory limits
```

WRITE

```
print filehandle (list);
print STDOUT (list);
print (list); ... Note STDOUT is default output destination
```

CLOSE

```
close (filehandle);
```

binmode(filehandle); ... bypasses EOR processing – i.e. <LF> or <CR><LF> in writing text records.
Requires byte counting and a supplied buffer
File I/O Examples

• open(myfile, “somefile”); ... entire file as an array to STDOUT
    @file=<myfile>
    print @file;

• open (SOURCE, “<source.txt”) || die “$!”;
  open (DEST, “>destination.txt”) || die “$!”;
  while (<SOURCE>) { print DEST ($_); }
  close (SOURCE);
  close (DEST);

• open (SOURCE, “source.txt”) || die “$!”;
  open (DEST, “>>destination.txt”) || die “$!”;
  @file=<SOURCE>
  print DEST (@file);
  close (SOURCE);
  close (DEST);

• open (SOURCE, “source.txt”) || die “$!”;
  open (DEST, “>destination.txt”) || die “$!”;
  @_=<SOURCE>
  print DEST (@_);
  close (SOURCE);
  close (DEST);
### Variable context

<table>
<thead>
<tr>
<th>Expression</th>
<th>Context</th>
<th>Variable</th>
<th>Evaluates to</th>
</tr>
</thead>
<tbody>
<tr>
<td>$scalar</td>
<td>scalar</td>
<td>$scalar, a scalar</td>
<td>the value held in $scalar</td>
</tr>
<tr>
<td>@array</td>
<td>list</td>
<td>@array, an array</td>
<td>the list of values (in order) held in @array</td>
</tr>
<tr>
<td>@array</td>
<td>scalar</td>
<td>@array, an array</td>
<td>the total number of elements in @array</td>
</tr>
<tr>
<td>@array[$x]</td>
<td>scalar</td>
<td>@array, an array</td>
<td>the ($x+1)$th element of @array</td>
</tr>
<tr>
<td>@array[$x, $y]</td>
<td>list</td>
<td>@array, an array</td>
<td>a slice, listing two elements from @array</td>
</tr>
<tr>
<td>%hash</td>
<td>list</td>
<td>%hash, a hash</td>
<td>a list of alternating keys and values from %hash</td>
</tr>
<tr>
<td>$hash{$x}</td>
<td>scalar</td>
<td>%hash, a hash</td>
<td>the element from %hash with the key of $x</td>
</tr>
<tr>
<td>@hash{$x, $y}</td>
<td>list</td>
<td>%hash, a hash</td>
<td>a slice listing two elements from %hash</td>
</tr>
</tbody>
</table>
System Calls

- `system "command"`; ....launch a child process in Perl to run a command
  `system "whoami"`;

- `exec "command"` .... replacing the Perl proces and terminating
  `exec "whoami"`;

- Default destination is STDOUT. Can capture output using “backticks” instead of doublequotes - `` instead of ""
  `$date = "date";
  my $now = `$date`;
  print "Time now is: $now";

- `open (filehandle, "| command")` .... Send data to/from an external command
  `open(MAIL, "| mail -s Test rjtaylor@csc.oakton.edu ") || die "mail failed: $!\n";
  print MAIL "This is a test message";

- `open (filehandle, "command | ")` .... Receive data from an external command
  `open(PS,"ps -e -o pid,stime,args |") || die "Failed: $!\n";
  while ( <PS> ) { print $_; }`