What is Perl?

• Perl stands for ‘Practical Extraction and Report Language’
• General purpose scripting language developed by Larry Wall in 1987.
• Has many of the characteristics of C, the various Unix shells, as well as text processing utilities like sed and awk
A very basic Perl script

Start up your favorite text editor and call this ‘hello’ and enter in the following two lines.

```perl
#!/usr/local/bin/perl5
print "Hello world!\n";
```

After saving this file, exit the editor and do the following:

```bash
>chmod u+x hello
```

• Perl programs or ‘scripts’ are not compiled, but interpreted.
• As such, the u+x permission must be set to run the script.

We run this script simply by typing:

```bash
>hello
```

If ‘.’ (current directory) is not in your path, then you must invoke the program as follows:

```bash
>./hello
```

Assuming no mistakes you should get:

Hello world!

So what's going on?
print "Hello world!\n";  # produces output on screen

• \n is the newline character which puts the cursor at the start of next line
• A semi-colon is needed at the end of (almost) every line in a Perl script.
• Comments can be put on any line, and must start with a # character.

Let’s modify our hello script top make it interactive.

#!/usr/local/bin/perl5
print "What is your name? ";
$name=<STDIN>;
chomp($name);
print "Hello there $name.\n";

If we run this, we get

> ./hello
What is your name? Tim
Hello there Tim.
>
So what’s happening here?

First prompt the user for their name.

```perl
print "What is your name? ";
```

then take input from the user

```perl
$name=<STDIN>;
```

This takes a line of standard input and assign it to the variable `$name`

(We’ll discuss variable nomenclature in the next section.)

Since the line of standard input includes a \n at the end (when we hit `ENTER`) this gets removed or ‘chomped’ by the command

```perl
chomp($name);
```

Finally, we say hello

```perl
print "Hello there $name.\n";
```
Perl Variables and Operators

In Perl, there are three basic data types:

- Scalars
- Arrays
- Associative arrays (also called hashes)

Unlike C or Pascal, there is no need to specify names or types of variables at the beginning of a program.

Scalars

Scalars consist of integer or floating point numbers or text strings.

The variable names begin with a $ followed by a letter or _ (underscore) and can consist of either letters (upper or lower case) or _ or numbers.

Ex:

```perl
$x = 3.5;
$name = "Tim";
$A_very_long_and_silly_looking_variable_name = 2;
```
All numbers in Perl are double precision floating point numbers (integers too!)

Ex:
```
$x=3;
$y=-5.5;
$z=6.0E23;  # exponential notation for 6 x 10^23
```

One can also work in Octal (base 8) or Hexadecimal (base 16) as well.

As for strings, the only two types are single and double quoted.

Ex:
```
$x = "Hello
";  # Hello followed by newline
$y = 'Hello
';  # literally Hello
```

Within double quotes, special characters like \n, are interpreted properly.

Ex:
```
\n  newline
\t  tab
\"  literally "
\\  literally \n```

For single quoted strings, however, what’s in quotes gets printed as is.

```php
$y = 'Hello
Goodbye';
print $y;
```

So if we have

```php
print "Left	Middle	Right\n";
```

we get

Left  Middle  Right

For single quoted strings, however, what’s in quotes gets printed as is.

```php
$y = 'Hello\nGoodbye';
print $y;
```

yields

Hello\nGoodbye  ⇐ the \n did not get treated as a newline

Also, if you wish to embed variables inside strings and have the value substituted in properly, you must use double quotes.

Ex:

```php
$name=“Tim”;
$greeting=“Hello $name\n”;
print $greeting;
```

will produce

Hello Tim
The typical operators for numerical values are present:

```
+, -, *, /
```

There is also an exponentiation operator,

```
2**3;  # 8 since 2^3 = 8
```

as well as a 'modulus' operator for taking remainders

```
5 % 2;  # 1, since 5 divided by 2 leaves remainder 1
```

Additionally, there are the autoincrement `++` and autodecrement `--` operators as in C.

```
$a=2;
++$a;  # $a now equals 3
--$a;  # $a now equals 2 again
$a++;  # $a now equals 3
$a--;  # $a now equals 2 again
```

Note, these also can be applied to decimal values as well.

Ex:

```
$b=3.5;
++$b;  # $b now equals 4.5
```
One also has the operators $+= -= *= $ for 'in place' assignment.

Ex:

```
$a=3;
$a+=2;  # $a now equals 5;
$a*=4;  # $a now equals 20;
```

as well as a $**= $ operator for in place exponentiation.

For strings, there is a concatenation operator for combining two strings

It is given by  . (a period)

Ex:

```
$x="Hello";

$y="There";

$z=$x.$y;  # $z is now "HelloThere"
```
We saw earlier the \texttt{chomp()} operator which removes a trailing newline character if one is present.

Ex:

\begin{verbatim}
$a="Hello There\n";
chomp($a);  # $a now equals "Hello There"

$b="Hi There";
chomp($b);  # $b still equals "Hi There"
\end{verbatim}

There is also the operator, \texttt{chop()}, which removes the last character in a string, whether it is a newline or not.

Making Comparisons

If we wish to compare two scalars then we must choose the appropriate comparison operator.

<table>
<thead>
<tr>
<th>Comparison</th>
<th>Number</th>
<th>String</th>
</tr>
</thead>
<tbody>
<tr>
<td>equal</td>
<td>(==)</td>
<td>eq</td>
</tr>
<tr>
<td>not equal</td>
<td>(!=)</td>
<td>neq</td>
</tr>
<tr>
<td>less than</td>
<td>(&lt;)</td>
<td>lt</td>
</tr>
<tr>
<td>greater than</td>
<td>(&gt;)</td>
<td>gt</td>
</tr>
<tr>
<td>less than or equal</td>
<td>(&lt;=)</td>
<td>le</td>
</tr>
<tr>
<td>greater than or equal</td>
<td>(&gt;=)</td>
<td>ge</td>
</tr>
</tbody>
</table>

We'll see examples of these used in the section on control structures.
Arrays

In Perl, arrays are lists of scalar values, either strings, or numbers.

Array names, as a whole, are prefixed with the @ sign followed by a letter or _ and can consist of either letters, numbers, or _ characters.

They can be created and modified in a variety of ways, the simplest is to just list the elements in the array.

Ex:

```perl
@X=(5,11,-6,12);
@People=("Tom","Dick","Harry");
@DaysOfWeek=("Mon","Tue","Wed","Thu","Fri","Sat","Sun");
@stuff=("Hi",3.1415,6,"Bye\n"); # mix and match!
```

Array elements are indexed starting from 0 and are accessed as follows:

Ex:

```perl
@X=(5,11,-6,12);
print "$X[2]\n";
```

yields

```
-6
```

That is, if the array is named @X then the i\(^{th}\) element is $X[i]
Adding elements to an array can be done in several ways.

Ex:

```perl
@People=("Tom","Dick");
@People=($People,"Harry")
```

So now,

```perl
@People=("Tom","Dick","Harry");
```

Note, if one instead did

```perl
@People=("Harry",@People);
```

then

```perl
@People=("Harry","Tom","Dick");
```

One can also add an element by means of the array index.

Ex:

```perl
@X=(3,8,-2);
$X[3]=5;
```

So now

```perl
@X=(3,8,-2,5);
```

That is, we have added a **fourth** element to the array. (at array index 3)
One can also copy arrays in a very intuitive manner.

```plaintext
@Names=('Tom','Dick','Harry');
@CopyOfNames=@Names;

So now,

@CopyOfNames=('Tom','Dick','Harry');
```

One can also take a 'slice' of an array.

Ex:

```plaintext
@Planets=('Mercury','Venus','Earth','Mars',
          'Jupiter','Saturn','Uranus',
          'Neptune','Pluto');

@InnerPlanets=$Planets[0,3];

So now, @InnerPlanets=('Mercury','Venus','Earth','Mars');

(Keep in mind, element 0 is the first element in the array.)
```
Combining two arrays is also very easy:

Ex:

```plaintext
@People=("Tom","Dick","Harry");
@MorePeople=("John","Jim");
@Combined=(@People,@MorePeople);
```

So now,

```plaintext
@Combined=("Tom","Dick","Harry","John","Jim");
```

There is a built-in `sort()` function for sorting the elements of an array.

Ex:

```plaintext
@People=("Tom","Dick","Harry");
@People=sort(@People);
@People now equals ("Dick","Harry","Tom");
```

- Note, the elements of the array are converted to strings before sorting.
- By default, the sorting is based on the ASCII value of the strings.
- There is also a way to sort arrays in numerical order.
Associative arrays

An associative array is a structure consisting of pairs of scalars, a key and a value, such that each value is associated to a key.

Associative array names, as a whole, begin with % followed by a letter or _ and can consist of either letters or numbers or _ characters.

As with regular arrays, individual elements are accessed with a $.

Typically, associative arrays are created and augmented on the fly, just by giving key and value pairs.

Ex:

```
$Grade{"Tom"}="A";
$Grade{"Dick"}="B";
```

note {} instead of [] for associative arrays
That is, `$Grade` is an associative array with (at the moment) two key and value pairs, which were given by the two assignment statements.

We could have also done this with the following statement:

```
%Grade = ("Tom" => "A","Dick" => "B");
```

Also, the keys and values can be different scalar types.

For example:

```
$blah{"xyz"}=1;
$blah{3.14}="π";
```

yes, a decimal can be a key!
(It’s a scalar.)
A very useful function to apply to an associative array is `keys()`.

As the name suggests, this returns all the keys in a given associative array in ordinary array form.

Ex:

```perl
%Grade=("Tom"=>"A","Dick"=>"B","Harry"=>"C");

@Students=keys(%Grade);

@Students now equals ("Tom","Dick","Harry")
```

Undefined Values

If a scalar value is referenced but has not been assigned a value, Perl gives it the default value of `undef` which literally means undefined.

So, for example, if `$a` has not been defined, then

```perl
print "$a";
```

will produce no output, but will not generate an error either.
Likewise

@X=(3,7,9,2);
print "@X[10]";

will produce no output.

The point being that any array element not yet defined has the value \texttt{undef}

And if

%Grade="Tom" => "A","Dick"=>"B";

then \texttt{$Grade\{\textit{Harry}\}} \texttt{is undef} since we have not given it a value.

Perl Control Structures

In Perl, there are a variety of loop structures and conditionals. Some of the syntax is similar to C.

All of these are built around what's known as a statement block which is simply a sequence of statements, surrounded by \{ and \}

Conditionals

Ex:

```perl
$entry=<STDIN>;
chomp($entry);
if($entry eq "Thank You"){
   print "Your Welcome!\n";
}
```
The conditional itself

$entry eq "Thank You"

is within parentheses and the value returned is either true or false.

If true, then the block within { and } is executed.

Before going further, here is a basic guide as to what is true or false in Perl:

- "0" and "" (the empty string) and undef are false.
- all else is true*

What Perl does, is to first convert any scalar to string, then apply the above rules.

* Note, "0.0" evaluates to true since, as a string, "0.0" is not "0"
One also has an else construction.

```perl
print "What\'s the password? ";
$entry=<STDIN>;
chomp($entry);
if($entry eq "SHAZAM"){
  print "Access Granted\n";
}else{
  print "Incorrect Password!\n";
}
```

If the conditional is true, ($entry eq "SHAZAM") then the print statement inside the first set of { and } is executed,

otherwise the "Incorrect Password!" message gets printed.

Note, in Perl, all blocks following conditionals need to be enclosed in { } even if they consist of just one statement! (unlike C)

Also, one does not need to put a semicolon after the closing } in a conditional.
One can also combine conditionals using

```
&& logical and
|| logical or
```

```
if(($day eq "Monday") && ($time eq "7AM")){
    print "Time to get up!\n";
}
```

Logical **not** is given via `!`

```
if(!($password eq "SHAZAM")){
    print "Access Denied\n";
}
```

---

**Loops**

One has many of the familiar loop constructions.

Consider the following examples.

Ex:

```
$n=1;
$sum=0;
while($n<=10){
    $sum = $sum + $n;
    $n++;
}
print "The sum of the numbers from 1 to 10 is $sum\n";
```
likewise

```php
$n=1;
$sum=0;
until($n==11) {
    # use == for numerical comparison!
    $sum = $sum + $n;
    $n++;
}
print "The sum of the numbers from 1 to 10 is $sum
";
```

There is also a `for` statement.

Ex:

```php
$sum=0;
for($n=1;$n<=10;$n++) {
    $sum = $sum + $n;
}
print "The sum is $sum
";
```

The general syntax is:

```
for(initial_expression;test_expression;increment_expression) {
    statement block
}
```
There is a nice generalization of `for()` used to loop over the elements of an array.

Ex:

```perl
@People=("Tom","Dick","Harry");
foreach $person (@People){
    print "$person
";
}
```

yields (as you might expect)

- Tom
- Dick
- Harry

One can use the `foreach()` function together with the `keys()` function to examine the contents of an associative array.

Ex:

```perl
%Grade=("Tom"=>"A",
    "Dick"=>"B",
    "Harry"=>"C"
);
@People=keys(%Grade);
foreach $person (@People){
    print "$person received a $Grade{$person} \n";
}
```

i.e.

- `keys(%Grade)` is the array ("Tom", "Dick", "Harry") extracted from the associative array `%Grade`;
Basic Perl I/O

As we saw in our interactive script:

```perl
#!/usr/local/bin/perl5
print "What is your name? ";
$name=<STDIN>;
chomp($name);
print "Hello there $name.\n";
```

we can take input from the keyboard and assign it to a variable name.

This is done using what's known as the STDIN filehandle which is how Perl deals with external files in general.

In this case, we asked for one line of standard input from the user.

However, if we wish to process a sequence of lines of standard input we can use, for example, a `while()` loop.

Ex: Suppose we want to take each line of input and surround it with brackets `[ ]`

```perl
while($line=<STDIN>){
    chomp($line);
    print "[{$line}]\n";
}
```
The lines of input can come from different sources than just the keyboard.

Ex: (Let's make that chunk of code into a script called 'bracket')

```perl
#!/usr/local/bin/perl5
while($line=<STDIN>){
    chomp($line);
    print "[$line]\n";
}
```

Now make this executable, with the chmod command as before.

[Think: What would happen if we forgot the chomp statement?]

Now try this,

```bash
>cat bracket | ./bracket
```

```perl
[#!/usr/local/bin/perl5]
[while($line=<STDIN>){}
[    chomp($line);
[    print "[$line]\n";]
[}

Literally, we sent the lines of the script itself (with the cat command) through the script, and each line was surrounded by brackets []
Regular Expressions (a.k.a. 'regexps')

- One of the most powerful features of Perl.
- Process text using what are known as regular expressions.
- Regular expressions are a means of doing pattern matching on strings.

The general syntax for a pattern is

```
/pattern/
```

where `pattern` is the text pattern we are trying to describe.

The general syntax to see if a string matches a certain pattern is:

```
$x =~ /pattern/
```

pattern matching operator

To see if `$x` contains the word `hello` we might write:

```
if($x =~ /hello/){
    #do something
}
```

i.e. If the pattern matches, then the conditional has value true.
By default, pattern matching is case sensitive, so the following strings would match:

```
>x="hello there"
>x="I just called to say hello"
>x="Othello by William Shakespeare"
```

yes! this is a match

but something like

```
>x="Hello to you!"
```

would not (the capital H makes a difference)

One way to make the pattern more flexible is to use alternation.

Ex:

```
>x =~ /th(is|at)/
```

matches if $x matches either

```
this or that
```

The ( | ) allows us to choose one or more possibilities.
Regular expressions allow us to be quite general about the patterns we look for.

Ex: Match all strings which have the letter \texttt{a} followed by at least one text character. (i.e. something other than \texttt{\textbackslash n})

\begin{verbatim}
/a./
\end{verbatim}

the letter \texttt{a} \hspace{1cm} any text character

So these would match

"apple"
"this and that" \# in several places actually

but not

"a"
"ha" \{ no character after the \texttt{a} }\)

We can also match on multiples of characters.

**multipliers.**

* zero or more occurrences of the previous character
+ at least one of the previous character
? 0 or 1 instances of the previous character
{\texttt{n}} \texttt{n} instances of the previous character
{\texttt{m}, \texttt{n}} \texttt{m} and \texttt{n} instances of the previous character

Ex:

\begin{verbatim}
/be*t/
\end{verbatim}

would match \texttt{"bet"} and \texttt{"beet"} or even \texttt{"bt"}
If we change * to + then

```
/be+t/
```

matches "bet" and "beet" but not "bt"
since the e+ means at least one instance of the letter e

If we change this to say

```
/b.+t/
```

then this would match "boot", "belt", "bet", "bat", "b t" etc.
since .+ means match one or more of any character

Again, the pattern just has to exist somewhere in the string in order to match.

Say we wish to see if there is a vowel somewhere in a given string.

We could do this as follows.

```
if($x=~/[aeiou]/){
    print "Found a vowel!\n";
}
```

The [ ] indicates a specific class of characters which we want to match.

In this case, one of the five vowels.
If we wish to match any lower case letter, then we can use

```regex
/[a-z]/ # i.e. all the letter from a to z
```

to include upper case letters we use

```regex
/[a-zA-Z]/ # all letter from a to z and A to Z
```

Likewise, we can also match digits.

```regex
/[0-9]/
```

There are also a number of pre-defined classes one can use which have abbreviations.

<table>
<thead>
<tr>
<th>description</th>
<th>class</th>
<th>abbreviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>digits</td>
<td>[0-9]</td>
<td>\d</td>
</tr>
<tr>
<td>words</td>
<td>[a-zA-Z_]</td>
<td>\w</td>
</tr>
<tr>
<td>space</td>
<td>[ \n\r\t\f]</td>
<td>\s</td>
</tr>
</tbody>
</table>

literally a space
To negate a class, use \(^\) 

ex. \(^{x}\) everything but the letter x

\(\w\) non-word characters \([^{a-zA-Z_}\])
\(\s\) non-space characters \([^{\ \n\t\f}]\)
\(\d\) non-digit characters \([^{0-9}]\)

One can combine pre-defined classes to make larger classes.

Ex:

\(\$x=~/\w\d~/\)

matches words and digits

So, for example

\(/\w{5,10}/\)

word class

would match any ‘word’ of between 5 and 10 characters in length.

Ex: The following match,

“therefore it is true” and

"the rain in Spain"

these are what match
Anchoring patterns

Suppose we wish to specify where in a string a given pattern is matched. For example, say we wish to see if a given string, starts with a capital letter.

```
$sentence =~ /^[:A-Z]/
```

The `^` is to test if the pattern is matched at the beginning of the string.

Likewise, we could test if a certain pattern is matched at the end of a string.

i.e. Say we wish to check if a certain string ends with the letter e

We could use the following:

```
$x =~ /e$/;
```

So this would match if

```
$x = "the"
```

but not if

```
$x = "the rest"
```
One can also anchor a pattern at a **word boundary** using the directive `\b`

Such a boundary occurs at the beginning of a string (or end) or at a transition from a `\w` to a `\W` or vice versa.

Ex:

```
$x =~ /the\b/;
```

matches if

```
$x="the"  or  $x="the end"
```

but not

```
$x="then"
```

Matching somewhere that is **not** a word boundary can be done with `\B`.

---

**References for further information on Perl**

**Books**

- *Learning Perl* by Randal L. Schwartz & Tom Christiansen (O'Reilly)
- *Programming Perl* by Larry Wall, Tom Christiansen and Jon Orwant (O' Reilly)
- *Perl in a Nutshell* by Ellen Siever, Stephen Spainhour, and Nathan Patwardhan (O' Reilly)

**Web**

- [http://www.perl.com](http://www.perl.com)
- [http://math.bu.edu/people/tkohl/perl](http://math.bu.edu/people/tkohl/perl) → My Perl Page!
Introductory Perl

Boston University
Office of Information Technology

Course Number: 4080

Course Coordinator: Timothy Kohl