An Introduction to Perl

Scalar Data
Two basic scalar data types in Perl are numbers and strings. Both integers and floating-point numbers have internally the same format. Strings are sequences of characters. They may be enclosed in single or double quotes. We will almost exclusively use double quotes. There is a special value `undef`, which is initially assigned to all variables. It acts as zero when applied to numbers, and as an empty string when applied to strings.

Variables and Operators
A scalar variable holds a single scalar value. Names of scalar variables begin with $ followed by a Perl identifier, e.g. $fred, $barney,$_

Assignment Operator: =
Arithmetic Operators: +, -, *, /
String Operators: . –string concatenation (a dot), x –string repetition
Numeric Comparison Operators: ==, !=, <,>,<=,>=
String Comparison Operators: eq, ne, lt, gt, le, ge

There are two special features of Perl worth mentioning at this point.
1. Interpolation of scalars into strings: Any scalar variable inside a double quoted string is replaced by its value.
   Example:
   $string="world";
   print "hello $string 
"; # prints hello world and switches to a new line
2. Automatic conversion between numbers and strings: Arguments of an arithmetic operator are automatically recognized as numbers, and arguments of a string operator as strings.
   Example:
   $fred=5;
   $barney=4;
   print $fred*$barney; # prints 20
   print $fred x $barney; # prints 5555
   print $barney x $fred; # prints 44444
   print $fred . $barney; # prints 54

If Else Control Structure
Both if and if-else structures are similar as in C++. Curly braces are however necessary.
if(…){
  ...
}

if(…) {
  …
} else {
  …
}

**While Control Structure**
While structure is also similar to C++. Again block curly braces are required.
while (…) {
  …
}

**Standard Input**
Input in Perl is very convenient. Let us look at several examples of reading procedures from the standard input device (which typically is user’s keyboard).

```perl
$line=<STDIN>
# read a full line from standard input and store it in variable $line
Example:
$line=<STDIN>;
if($line eq "\n") {  
  print "This was an empty line.\n";  
} else {  
  print "This line was not empty.\n";  
}
# A line of input is read and one of the two strings is printed out,
# depending on what the input was.
while(defined($line=<STDIN>)){  
  print “This line was: $line \n”;  
}
# Lines of input are printed out right after inputting. Once the end-of-file is
# reached the line-input operator returns `undef` and the while loop stops
# execution.
while(<STDIN>){  
  print “This line was: $_ \n”;  
}
# The above code is a shortcut of the previous reading procedure. Now each line
# of input is automatically stored inside Perl’s default variable $_.
while(<>){  
  print “This line was: $_ \n”;  
}
# One may also use the diamond operator <> for reading. The effect is the same
# as previously. However the diamond operator has the advantage of allowing
# command line arguments.

**Lists and Arrays**
Lists are also user friendly in Perl. Arrays are variables storing lists.

**List Literals (representations of lists)**

Examples:

- `(1,2,3,4,5,6,7,8,9,10)` # a list consisting of numbers 1,2,…,10
- `(1..10)` # the same list as the one above, now created by the `range` operator
- `($a ..$b)` # the range determined by current values of $a and $b

**List Assignment**

Examples:

- `($fred, $barney)`=(`$barney, $fred`); # values are swapped
- `@numbers`=(`1..10`); # list (1..10) is stored inside an array

**Accessing List Elements**

Example:

- `@numbers`=(`1..100`);
- `print $numbers[0];` # prints 1
- `print $numbers[10];` # prints 11
- `print $numbers[100];`
- `# nothing is printed out, the value of $numbers[100] is undefined`
- `print $numbers[$#numbers];`
- `# $#numbers is the largest index; number 99 is printed, it is the last element of`
- `# the list

**Reading into a List**

Example:

```perl
while(<>){
    $lines[$index]=$_;
    $index=$index+1;
}
print $#lines+1;
print @lines;
```

# Consecutive lines of input become elements of a list stored inside array @lines.
# After reading we print out the number of input lines, and then the whole list
# itself.

**Hashes**

Hashes are similar to arrays, but instead of numbers we use strings for indexing.
Hash indices are called keys.

**Accessing Hash Elements**

Example:

```perl
$numbers {“one”}=”1”;
```
$numbers{"two"}="2";
$numbers{"three"}="3";

Hash Assignment
%numbers=("one","1","two","2","three","3");
#A list is assigned to a hash.

%inverse_hash=reverse %numbers;
# Hash %numbers gets reversed, keys become values and values become keys.
# After reversing the hash is assigned to another variable %inverse_hash;

Function Each
Example:
while(($key, $value) = each %numbers){
    print "$key $value\n";
}
#All successive key-value pairs are returned by function each, assigned to
#variables $key, $value and then printed out.

Foreach Control Structure
Examples:
foreach $word (@words){
    print "$word\n";
}
#Variable $word takes the successive values of @words. All elements
#of the list are printed in order.

foreach (1..10){
    print "$_\n";
}
#Prints all numbers from 1 to 10.

Split Function
Examples:
@fields=split " ", $string;
#Words, which are separated by spaces inside the string $string, become separate
#elements of the list @fields.

while (<>){
    foreach (split) {
        print "$_\n";
    }
}
#Words, which are coming form the input become successive values of the
#default variable $. They are printed out in order, each word in a separate line.