Perl Programming Introduction

Fergus Toolan.

Review Questions

- How do you list the logged in users on a UNIX system?
- How do you list the contents of a directory?
- How do you change to your Home Directory?
- How do display an entire text file?
    - The beginning of the file.
    - The end of the file.
- What key is used to move to the next screen in more?
- How do display all files (including hidden files) in a directory?

Project

- Three Deliverables
  - Pre-Processor DATE
  - Retrieval System DATE
  - Documentation DATE
- All must be handed in on time.
- All files should be in a directory called project in your home directory.
- Marks will be deducted for late submission of work.
- The project is worth 30% of the course so it is worth spending time on.
- Some code will be provided to get you started.

Project

- http://ftoolan.ucd.ie/unix - any news about the project will appear here.
- There is no lab specifically for the project. In the normal labs TIME PERMITTING the demonstrators will help out. However they are there to teach the lab class rather than help with the project.
- There will be help available for the project in the form of demonstrator's office hours. These are
  - Fergus Toolan - B1.04 - Mon 2-4pm - fergus.toolan@ucd.ie
  - Lorna Kane - B1.04 - Tue 2-4pm - lorna.kane@ucd.ie
  - Martina Naughton - B1.03 - Wed 2-4pm - martina.naughton@ucd.ie
  - Aidan Finn - B2.18 - Fri 3-5pm - aidan.finn@ucd.ie
- We will try to organise someone to do some time on Thursday so that someone is available each day.
- If you want to organise a time outside of these hours email the demonstrators.

- Remember that in the week where a deadline is due the demonstrators will be fielding many questions so try to get organised and not leave everything to the last minute.

Overview

- What is Perl?
  - Why do we need it?
  - How does it work?
- My First Program
- Input/Output
- Scalars, Arrays and Hashes
- Control Structures
- Procedures
- Regular Expressions

What is Perl?

- Practical Extraction and Report Language
    - Pathologically Eclectic Rubbish Lister
    - Also coined by Larry Wall - creater of Perl
- An interpreted Programming Language
- Extremely good for Text Processing - indeed it was designed for this.
- Created in the late 70's / early 80's
- Current version is Perl 5.8.*
- Perl 6 is due for release any day now (although they have said that since 2000)
- Perl is an imperative programming language with some OO stuff.

How does it work?

- Perl is interpreted!
- What does this mean?
    - Write your source code in a normal file
    - Save as filename.pl
    - perl filename.pl - invokes the command interpreter
    - The source is read by the Perl interpreter which preprocesses the code
    - The code is executed line by line
    - NB: It is not compiled - no executable is built
- Interpreted Languages are normally slower than compiled languages
- It is a full programming language - no matter what people tell you.
- It is just optimised for text processing
- General Rule: Know your programming languages. We all have our favourites but sometimes our favourites can't do what you want. For instance Perl is more at mathematical functions - use C :-(instead.

Recommended Reading

- Google web search "Perl Tutorial"
- Perl - Black Book (a.k.a. The Bible) Stephen Holzner, Coriolis Press.
- Programming Perl - Larry Wall, Tom Christiansen & Randall L. Schwartz - O'Reilly Press.
- Any books on Regular Expressions - or the grep man page.

My First Program

```
#!/usr/bin/perl -w
#############################
# Commments go after the # symbol
#############################
print "Hello World\n";
```

- Code Breakdown
  - `#!/usr/bin/perl`

    The location of the Perl Interpreter
  - -w - display warning messages in execution
  - `print "Hello World\n";`
    print the string in quotes
  - Note that there are no brackets - they are not essential in Perl.

Variables

- Perl is weakly typed
  - This is new to you - Java is strongly typed
  - i.e. we must declare variable types int, String etc.
  - There is no need to do this.
  - Perl provides only three “types”;
- Scalars
  - Stores single data items - ints, strings, filehandles,...
- Arrays
  - Stores arrays of data items
- Hashes
  - Associative Arrays - similar to Hash Tables

Scalars

- Standard variables preceeded by $ sign.
    - Examples
    - $i = 0;
    - $name = ”fergus”;
- Lets modify our first program to use variables

```
#!/usr/bin/perl -w
###########################

$name = "fergus";
print "Hello $name\n";
```

Input from STDIN

```
#!/usr/bin/perl -w
###########################
$name = <STDIN>;
print "Hello $name\n";
```

- Now we run it and check the output

```
[root@ftoolan example]# perl test.pl
Enter your name:
fergus
Hello fergus

[root@ftoolan example]#
```

- Note that we have an extra newline at the end. Where did that come from?

Input Continued

- The newline character from the input was also taken into Perl. How do we avoid this?
  - chop - removes the last character.
  - chomp - removes a trailing newline character.

```
#!/usr/bin/perl -w
###########################
$name = <STDIN>;
chomp $name;
print "Hello $name\n";
```

- Now we run it and check the output

```
[root@ftoolan example]# perl test.pl
Enter your name:
fergus
Hello fergus
[root@ftoolan example]#
```

Arrays

- Arrays contain many data elements accessed through an index.
- Arrays in Perl (like Java) begin at position 0;
- Where scalars used $ arrays use @ and then the name.
- To initialise an empty array we use
  - `@arrayname = ();`
- To initalise an array with elements we use
  - `@arrayname = (1, 2, 3);`
- To access a particular element in an array we use
  - $arrayname[1] - accesses value at position 1 in the array
  - Note we are now in scalar context as we are accessing a particular value.

Printing the Values in an Array

```
#!/usr/bin/perl -w
###################################
# Prints the values in an array   #
###################################
@values = (1, 2, 3, 4);
$i = 0;
while ($i < $#values + 1)
{
  print "$values[$i]\n";
  $i++;
}
```

- $#values stores the last index in the array
- Therefor the length is $#values + 1
- Another way to find the length is
  - `$length = @values;`

Exercise

- Sum the values in an array...

Hashes

- Similar to a hash table in Java (Data Structures II course)
- All items are identified by a unique key.
- The hash structure in Perl is more correctly called an Associative Array.
- It does not allow collisions - data will be overwritten.
- This gaurantees that each key is unique - can be very handy.
- To initialise a hash we use the following syntax

  -
    ```
    %hashname = ();
    ```
- It is the same as arrays but uses % instead of @
- To inititalise a hash with elements we use

  -
    ```
    %hashname = ( "k1" => 1,
          "k2" => 2
    );
    ```

Hashes

- To access an element in a hash we need to know the key of that element.

```
#!/usr/bin/perl -w
##########################
%h = (  "k1" => 1,
"k2" => 2
);
print "$h{k1}\n";
print "$h{k2}\n";
```

- Note the use of $ to access an individual element in the hash. Like arrays when we access a particular element we are dealing in scalar values.
- Sometimes we want to access all elements in a hash.

Accessing Hash Elements

```
#!/usr/bin/perl -w
################################
# Print all elements in a hash
################################
%h = (  "k1" => 1,
"k2" => 2,
"k3" => 3
);

foreach $key (keys %h)
{
  print "$h{$key}\n";
}
```

- What order will the elements be printed in...

Accessing elements in a hash

- Here is the output from my machine

```
[root@ftoolan example]# perl test.pl
2
1
3
[root@ftoolan example]#
```

Exercise

- Do something using hashs..

Control Statements

- All imperative languages provide two major types of flow control statements
  - selection - if, switch, case,...
  - iterative - while, for, do, repeat,...
  - There is also recursion but we won't be mentioning that again :-)
- Selection statements allow us to select a certain block of code to be executed based on some condition.
- The general form of a selection statement is

```
if (condition is true) {
Do Something
}
else
{
Do Something Else
}
```

Control Statements

- All iterative statements contain initialisation, condition and progress steps.

```
for (initialisation, gaurd, progress)
{
Do Something
}
```

Selection - IF Statment

```
#!/usr/bin/perl -w
$name = <>;
chomp $name;
if ($name eq "fergus")
{
  print "I like that name\n";
}
else
{
  print "I don’t like that name\n";
}
```

Selection - IF Statment

- We can also have multiple conditions.

```
if ($name eq "fergus")
{
  print "I like that name\n";
}
elsif ($name eq "joe")
{  # We can use as many elsif clauses as we like. Note spelling!!
  print "Thats not too bad\n";
}
else
{
  print "I don't like that name\n";
}
```

Iterative Statements

- We will use the while statement.
- To sum the values in an array we use the following

```
@values = (1, 2, 3);
$i = 0;
while ($i < @values)
{
  $sum = $sum + $values[$i];
  $i++;
}
print "$sum\n";
```

- Where is the mistake...

Correct Loop

```
@values = (1, 2, 3);
$sum = 0; # We must initialise sum to gaurantee correctness.
$i = 0;
while ($i < @values)
{
  $sum = $sum + $values[$i];
  $i++;
}
print "$sum\n";
```

- The line $i = 0; is the initialisation stage. We set the counter to the index of the first array element (remember arrays begin at position 0)
- The gaurd is $i ¡ @values in other words while i is less than the length of the array.
- $i++ is the progress step. If this is removed we have an infinite loop.

File I/O in Perl

- TRIVIAL compared to Java.
- Method for reading.
  - Open File.
  - Read File line by line.
  - Close the file.
- Method for writing.
  - Open the File selecting overwrite or append methods.
  - Write to the File.
  - Close the File
- As in any language that uses files closing the file is vital.
- All systems have a limit on the number of open files they allow.
- One linux system I run can have 106,009 open files at once.

Reading a File

```
#!/usr/bin/perl -w
################################
# Reads the contents of a file #
################################
open FH, "<file.txt"; # opens file.txt with a Filehandle
                      # called FH.
$line = "";
while ($line = <FH>)  # Read a line at a time until
                      # the EOF character.
{
  print "$line\n";    # print the line
}
close FH;       # Close the file.
```

Writing to a File

```
#!/usr/bin/perl -w
############################
# Write Hello World to a file
############################
open FH, ">file.txt"; # open file note > symbol this means
                      # overwrite. >> means append.
print FH "Hello World\n";  # print to the file, Note no
                           # comma after FH.
close FH;
```

String Operators - split and join

- Perl is very good for handling text. It provides some handy built-in functionalityfor this.
- Two commonly used commands are split and join.
- split is similar to the StringTokenizer in Java.
- It splits the string on a particular delimiter (tab, space etc)
- The results are stored in an array.
- Join takes an array as argument and joins the elements to form a string.

split

```
$text = "10,24,1,5";
@values = split ",", $text;  # comma is the delimeter.
$i = 0;
$sum = 0;
while ($i < @values)
{
  $sum = $sum + $values[$i];  # note no casting needed.
  $i++;
}
```

join

```
@values = (1, 2, 3);
$text = join " ", @values; # joins them using a space.
print "$text\n";

# Could do this is one step.

print join " ", @values;
```

Exercise

- Create a file with a line of text in it.
- Write a Perl script to print the files contents with one word per line.
- Eg:

```
INPUT TEXT: This is the text of the file
OUTPUT:
This
is
the
text
of
the
file
```

- Hint use split.
- When we come to regular expressions we will see a better way of doing this using the substitution operator.

Procedures

- Procedures are like methods in Java.
- Code is bundled together to perform a particular operation.
- Procedures may take parameters and can return a result.
- Parameters are passed as an array using a special variable `@_`
- We don't need to define the parameters that are passed in although we can.
- Procedures are called as follows

  ```
  &procedure_name(Parameters);
  ```

- Note & symbol

Writing a Procedure

- Here are procedures to sum two values

```
sub sum1 { # note no parameter list
  $a = $_[0];   # get the first arg
  $b = $_[1];   # get second arg.
  $sum = $a + $b; # add them
  return $sum;    # return the value.
}
# A better way??
sub sum2 {
  $a = shift;   # Gets first element in array and removes it.
  $b = shift;
  return ($a + $b);
}
# An even better / faster way
sub sum3 {
  return ($_[0] + $_[1]);
}
```

Passing Arrays and Hashs to Procedures

- So far we have only passed scalar values to a procedure. How do we go about passing an array or hash to a procdure?
- Using references
- We pass the memory address rather than the array/hash to the procedure.
- A memory address is a scalar value.
- We can recreate the structure in the procedure.
- The following code gets a reference to an array A

```
$aref = \@A; # The \ symbol gives the reference.
```

- Next we will write a procedure to sum the values in an array.

Sum Array Values

```
@values = (1, 2, 3, 4);
$total = &sumArray(\@values); # Pass in the reference.
print "Total: $total\n";

sub sumArray {
  $aref = shift; # the reference to the array
  $i = 0;
  $sum = 0;
  while ($i < @$aref)   # @ before the ref to tell Perl
                        # to treat it as array
  {
    $sum = $sum + $$aref[$i];  # use extra $ to treat as
                               # scalar
    $i++;
  }
  return $sum;
}
```

Sum Values in a Hash

```
%values = ( k1 => 1, k2 => 2, k3 => 3, k4 => 4);
$total = &sumHash(\%h);
print "TOTAL: $total\n";

sub sumHash {
  $href = shift;
  $sum = 0;
  foreach $key (keys %$href)
  {
    $sum = $sum + $$href{$key};
  }
  return $sum;
}
```

Exercise

- Practice, Practice, Practice,...
- Create a telephone book using a Hash. Read names and numbers from a text file and then allow the user to query it for the number they are looking for.
- This will not be marked however some of the procedures will be very applcable to the project and it will get you practice.
- Demonstrators will be available during speicifed Office Hours to help on this.
- For those looking for a challenge extend the telephone application to deal with multiple phone numbers.
- There are numerous ways of doing this however the way I would recommend is to use a hash of arrays. this is by far the most difficult but also the most worthwhile.

Regular Expressions

- These are what give Perl its real power.
- A Regular Expression (regexp, RE) is used to find patterns in text.
- We can use them to replace all occurences of one string with another, to extract content from formatted pages - e.g. web page titles.
- I said there was a better way to write the last exercise using regular expressions

```
#!/usr/bin/perl -w
open FH, "<file.txt";
$line = <FH>;
$line =~ s/\ /\n/g;
print "$line\n";
close FH;
```

RE's

- Perl was designed to handle text matching.
- RE's are the thing that allow this.
- They are considered a very arcance part of Perl.
- Also considered difficult to master.
- However when you have mastered them they give great power.
- They allow you to perform pattern matching.
- The pattern might contain wildcards.
- There are two regular expression operators
    - The match operator m//
    - The substitution operator s///

m//

- The m// operator tries to match a pattern in a piece of text.
- The syntax of m// is

```
 $text =~ m/pattern/modifiers
```

- For example to search a string input for the word exit we use

```
if ($text =~ m/exit/i) {
  exit;
}
```

m//

- The i modifier means ignore case it will match exit, Exit, EXIT,...
- You can negate the m// operator i.e. it returns true if we don't match the pattern

```
if ($text !~ m/exit/i) {
  # Do Something...
}
```

- In scalar context m// returns true or false.
- We can also use it in list context

```
$text = "Here is the text";
@a = ($text =~ m/\b[^A-Z]+\b/g);
print join " ", @a;
```

- In the above example we return all the matching words - we are searching for lowercase words.

- I.e. The output is " is the text"
- The g modifier is used to match more than once.
- If we left this out the returned value would be "is". I.e. It would just match the first occurence.

s///

- This is used for substitution.
- The general form is

```
$text =~ s/old_pattern/new_pattern/modifiers;
```

- As an example lets replace young with old in a string.

```
$text = "Very young.";
$text = s/young/old/g;
print "$text\n";

[root@ftoolan unix]# perl test.pl
Very old.
[root@ftoolan unix]#
```

m// & s///

- These operators start matching from the left.

```
$text = "Pretty young, but not very young";
$text =~ s/young/old/;
print "$text\n";

[root@ftoolan unix]# perl test.pl
Pretty old, but not very young
[root@ftoolan unix]#
```

- How do we make this s/// replace every occurrence of young?

m// & s///

- Use the g modifier

```
$text = "Pretty young, but not very young";
$text =~ s/young/old/g;
print "$text\n";

[root@ftoolan unix]# perl test.pl
Pretty old, but not very old
[root@ftoolan unix]#
```

RE Overview

- The Regular Expression is the pattern you wish to match.
- This lecture will cover the following RE topics.
  - Characters
  - Character Classes
  - Alternative Match Patterns
  - Quantifiers
  - Assertions
  - Other Special Topcis and Helpful RE's

```
$text = "Perl is the Subject";
$text =~ /\b([A-Za-z]+)\b/;
print "$1\n";
```

- The $1 variable contains the first match i.e. Perl

Characters

- In a Regular Expression each character matches itself unless it is a meta-character.

```
while ($input = <>)
{
  if ($input =~ /exit/i) {
    exit;
  }
}

Or more correctly

while ($input = <>) {
  if ($input =~ /^exit$/i) {
    exit;
  }
}
```

Special Characters

- \n - newline
- \t - tab
- \d - digit
- \D - non-digit
- \s - whitespace
- \S - non whitespace
- \u - uppercase
- \l - lowercase
- \w - word character (alphnumeric and "_")
- \W - non word character

Matching Special Characters

- We can match special characters in a regular expression.
- The example below searches for one or more digit characters.

```
$text = "ID: 123";
$text =~ /(\d+)/;
print "$1\n";

[root@ftoolan unix]$ perl test.pl
123
[root@ftoolan unix]$
```

Matching any Character

- To match any character we use . This matches any character except newlines (see s modifier later)
- If we want to replace all character with an asterisk * we use

```
$text = "Matching any character.";
$text =~ s/./*/g;
print "$text\n";

[root@ftoolan example]# perl test.pl
***********************
[root@ftoolan example]#
```

Matching . Character

- Characters like . are called meta-characters.
- These include \ — ( ) [{ ^$ * + ? .
- To match these we must escape them i.e. use \.
- Here is an example to match a . at the end of a line.

```
#!/usr/bin/perl -w
################################
# Print all elements in a hash
################################
$text = "Matching meta-characterers.";
if ($text =~ m/\.$/) {
  print "Full-stop found\n";
} else {
  print "No full-stop\n";
}
```

```
[root@ftoolan unix]# perl test.pl
Full-stop found
[root@ftoolan unix]#
```

Character Classes

- Sometimes we want to match one or more of a particular set of characters.
- We can create our own character class by enclosing the characters in
- For instance a character classes for vowels is [aeoiu]

```
$text = "Here is the text";
if ($text =~ /[aeoiu]/) {
  print "Yep, we got vowels\n";
}
```

- To negate a character class we use ^ For instance

```
$text = "Here is the text\n";
```

```
if ($text =~ /[^ab]) {
  print "There are characters other than a or b\n";
}
```

Alternative Match Patterns

- Sometimes we wish to match alternative patterns i.e. one or the other.
- To do this we use —
- Extending our exit example to accept quit or stop also.

```
while ($input = <>) {
  if ($input =~ m/exit|stop|quit/i) {
    exit;
  }
}
```

- To ensure that it is the only input on the line we use

```
while ($input = <>) {
  if ($input =~ m/^(exit|stop|quit)$/i) {
    exit;
  }
}
```

Quantifiers

- These are like wildcards in the OS.
- matches zero or more times.
- + matches one or more times.
- ? matches zero or one times.
- {n} matches n times.
- {n,} matches at least n times.
- {n,m} matches at least n times but no more than m times.

Quantifiers

- Replace multiple e's with one e.

```
$text = "Hello from Peeeeeerl";
$text = s/e+/e/g;
print "$text\n";

[root@ftoolan unix]$ perl test.pl
Hello from Perl
[root@ftoolan unix]$
```

- Note that the e in Hello was also replaced.
- If we didn't use a g modifier the string would look identical. Why?

Quantifiers

- A commonly used quantifier is . This matches zero or more characters.
- Take for example extracting titles from a HTML page.

```
$text = "<TITLE>Fergus' Homepage</TITLE>";
$text =~ m/<TITLE>(.*)<\/TITLE>/i;
print "$1\n";


[root@ftoolan example]# perl test.pl
Fergus' Homepage
[root@ftoolan example]#
```

Quantifiers

- Matching words of 4 or more characters.

```
$text = "This is some text with long and short words";
@a = ($text =~ m/\w{4,}/gi);
print join " ", @a;
print "\n";

[root@ftoolan example]# perl test.pl
This some text with long short words
[root@ftoolan example]#
```

Assertions

- Assertions (or anchors) match conditions in Strings not characters.
- We have already seen \b, ând $
- These match a word boundary, start of line and end of line respectively.

```
$text = "Here is some text";
$text =~ s/\b([A-Za-z]+)\b/There/;
print "$text\n";

[root@ftoolan.ucd.ie unix]$ perl test.pl
There is some text
[root@ftoolan.ucd.ie unix]$
```

Modifiers

- The modifiers are the letters that appear at the end of m// or s///.
- We have already seen g and i. Making the match global and case insensitive respectively.
- Others include:
  - e: indicates that the right-hand side of s/// is code to evaluate.
    ```
    $text = "This is SOME text with long and short words";
    $text =~ s/\b(SOME)\b/lc $1/ge;
    ```
  - m: Lets ând $ match embedded \n characters
  - s: Lets . match newlines.

Matching Words

- How do we match a complete word in text.
- There are numerous methods.
  - \S - Matches non-whitespace characters
    `$text =~ /(\S+)/;`
  - \w - Matches alphanumeric and “_”
    `$text =~ /(\w+)/;`
  - \b - Word Boundaries
    `$text =~ /([A-Za-z]+)/;`
  - Character Classes - [A-Za-z]
    `$text =~ /(\b[A-Za-z]+)\b/;`

Matching the Beginnning of the Line

- Sometimes we want to match the beginning of a line.
- We use ôr \A

```
if ($text =~ /^\./) {
  print "Shouldn't start a sentence with a period.";
}

or

if ($text =~ /\A\./) {
  print "Shouldn't start a sentence with a period.";
}
```

- The difference is in multi-line mode (when the m modifier is used) m̂athces the beginning of every line but \A matches only the very start of the first line.

Matching the End of a Line

- Sometimes we wish to match the end of a line.
- We use $ to do this.

```
while ($text = <>) {
  if (m/exit$/) {
    exit;
  }
}
```

- The above code checks if the user has entered exit.
- Remember chomp function.
- This shows that $ matches everything up to the newline.
- Be careful with the m and s modifiers as they can change this behaviour.

Checking for Numbers

- We can use \d or a character class to check for digits

```
if ($number =~ /^\d+$/) {print "Number"; }
if ($number =~ /^[0-9]+$/ {print "Number"; }
```

- However this only checks for positive integers.

- To check for all numbers use.

```
if ($text =~ /^[+-]?\d+\.?\d*$/) {print "Number\n"; }
```

Checking for Letters

```
$text = "aBc";
if ($text =~ /^\w+$/) { print "Only word characters found\n"; }
# Note this will match ‘‘\_"

if ($text =~ /^[A-Za-z]+$/) { print "Only Letters Found\n"; }
# This will only match letters in the character class specified.
```

Finding Multiple Matches

- Use the g modifier

```
$text = "Here is the texxxt";
while ($text =~ m/x/g) {
  print "Found another x\n";
}

[root@ftoolan unix]# perl test.pl
Found another x
Found another x
Found another x
[root@ftoolan unix]#
```

Finding the nth Match

```
$text = "Name: Ann Name: John Name: Tom Name: Tim";
$match = 0;
while ($text =~ /Name: (\w+)/g)
{
  $match++;
  print "Match $match\t$1\n";
}
```

```
[root@ftoolan example]# perl test.pl
Match 1 Ann***
Match 2 John***
Match 3 Tom***
Match 4 Tim***
[root@ftoolan example]#
```

Matching in Multiple Line Text

- Using the s modifier.
  - Normally . matches any character but the newline
  - You can change this by using the s modifier.

```
$text = "here is some text\n";
$text =~ m/(.*)/s;
print "$1.";
```

```
[root@ftoolan example]# perl test.pl
here is some text
.
```

- However if you want to match over multiple lines you should use m.

Using the m Modifier

- Lets replace the beginning of the line with BOL and the End of the line with EOL.

```
$text = "This text has\nmultiple lines";
$text =~ s/^/BOL/g;
$text =~ s/$/EOL/g;
print "$text\n";

[root@ftoolan example]# perl test.pl
BOLThis text has
multiple linesEOL
[root@ftoolan example]#
```

Using the m Modifier

- Why did it only replace at the beginning and end of the string and not each line.
- \n in the middle was ignored.
- Change this with the m modifier.

```
$text = "This text has\nmultiple lines";
$text =~ s/^/BOL/mg;
$text =~ s/$/EOL/mg;
print "$text\n";
```

```
[root@ftoolan example]# perl test.pl
BOLThis text hasEOL
BOLmultiple linesEOL
[root@ftoolan example]#
```

- Here Perl sees the \n and ând $ match .

Greedy Quantifiers

- Perls Quantifiers are greedy by default.
- This means that they match the most characters possible.
- For instance

```
$text = "That is some text isn't it";
$text =~ s/.*is/That's/;
print "$text\n";

[root@ftoolan example]# perl test.pl
That'sn't it
[root@ftoolan example]#
```

- To make a quantifier less greedy you use ?

Greedy Quantifiers

```
$text = "That is some text isn’t it";
$text =~ s/.*?is/That’s/;
print "$text\n";

[root@ftoolan example]# perl test.pl
That’s some text isn’t it
[root@ftoolan example]#
```