

Creating Perl scripts for the semi-automated mark-up of legacy taxonomic works

ver. 1.3

Thomas Hamann

Copyright: Document copyright © Thomas Hamann/Naturalis Biodiversity Center 2013-2016. This document is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported (CC BY-SA 3.0) license.

This project was subsidized in part by the EU project “pro-iBiosphere” (Grant agreement 312848).

Introduction

The following document explains what a potential script developer should keep in mind when developing scripts for semi-automated mark-up of legacy taxonomic works with FlorML and Perl, using regular expressions only. It covers how the legacy taxonomic work should be analysed, where useful information of the use of Perl can be found, and what aspects of Perl are important when creating scripts.

Table of Contents

Creating Perl scripts for the semi-automated mark-up of legacy taxonomic works	1
Introduction	1
Analyzing legacy taxonomic works	3
Introduction	3
Identifying useful text	3
Keys	5
Headings and subheadings	5
Figure captions and figure references	6
Other keywords	6
The absence of specific text.....	7
More options	7
Determining the optimum script order	7
Creating scripts	10
The programmer's mindset	10
Why Perl?.....	11
Required programs	11
Perl setup	11
Notepad++ setup	11
Saving scripts.....	13
Script basics.....	14
Comments.....	14
Perl pragmas to use	15
Opening and closing files	15

The <code>while</code> loop	16
Printing to the output file	17
Prepending or appending text to a file	18
The <code>if-else</code> loop.....	19
Matching and replacing text using regular expressions	20
Special characters	20
Grouping and backreferencing	21
Alternate choices within groups and grouping within groups	22
Grouping without backreferencing.....	22
Character classes.....	23
Look-ahead and look-behind assertions and positive and negative matching	23
Modifiers	24
Useful strategies for script design	24
Order of regular expressions within scripts.....	24
Order of terms within groups	24
Fixing after the fact	25
Reuse earlier scripts.....	25
Debugging scripts.....	25
Types of scripts	26
Clean-up scripts.....	26
Mark-up scripts	26

Analyzing legacy taxonomic works

Introduction

Before you even think of creating scripts, you should have a look at the printed legacy taxonomic work to get an idea of the type of contents it features. By "printed legacy taxonomic work" the actual volumes are meant. Scans can also be used, but you will likely find it is more difficult to get an overview when working purely from a computer screen (it is also slower).

If you are working on volumes that are part of a series, you should check whether there are (very) large differences in contents between the books. If so, you may be better off by considering each volume a single book that needs its own separate set of scripts. However, such cases are fairly rare.

If the books can be subdivided into one or more groups with the same type of contents, scripts can be created that will be used for the whole series of books. These scripts may evolve during use on the various volumes, gaining more detail and abilities with each new volume processed.

In both cases, basic scripts can be created by identifying useful text in the volume and using that to define an optimal script order and to base the scripts off.

Identifying useful text

When leafing through a volume, you should ask yourself questions like:

- **What kind of global contents does this volume have?**
 - How is the contents globally arranged?
 - Is all contents taxonomic in nature? Or is there other contents present? If so, what kind? And where is it located?
 - Are Addenda/Errata present?
 - Is contents mentioned in tables of contents or indexes unique? Or can it be derived from the remaining contents of the volume? If so, is there any reason it could not be left out of the marked up volume?
 - What other contents can be left out? Page headings, etc.
- **What about the general text contents of this volume?**
 - Where are references or citations located? Are they in lists or not? Or both? Do they only exist as part of another piece of text, are they appended at the end of sections, or do they exist in their own section(s)?
 - What about the figures? Where are they located? Is there a figure on the front page? Are they all line art, or are some photographs? Do they all have captions?
 - Is there any other content present that may qualify as a figure? What about signatures? Hand-drawn tables? Legacy glyphs/symbols that have no modern equivalent?

- What kind of text is present? Only nomenclature, taxonomic descriptions, distribution and ecology information? Or more? If more, what?
- Are keys present? If so, which format do they use? Indented¹ or not? Dichotomous or polytomous?
- Are tables or lists obviously present? What about footnotes? Stand-alone headings?
- **If Addenda/Errata are present, what format do they have?**
 - Do they contain figures?
 - Can they be eliminated by incorporating the changes they suggest into the main text before mark-up²? Or does this require the expert intervention of a taxonomist?
- **What about the details?**
 - What kind of headings are present? Are they all stand-alone, or are some inline with the paragraph's text following them?
 - Are writers mentioned on separate lines for certain sections?
 - Is the nomenclature separated into heterotypic synonyms, with homotypic synonyms on the same line? Or is no distinction made between heterotypic and homotypic synonyms and is there only one name per line in the nomenclature?
 - What kind of names are given in nomenclature? Only binomial, or different ones, too? Are there names that do not comply with current nomenclatural rules?
 - Where are types located? Inline with the name they belong to, on separate lines within the nomenclature, or elsewhere in the text?
 - How are the descriptions build up? What punctuation is used to separate individual characters?
 - Is the distribution information a list of localities, or is it a normal sentence?
 - Is the habitat/ecology information a list of habitats, or is it a normal sentence? Are altitudes mentioned? Flowering etc. periods?
 - What punctuation is used to separate individual references or citations? What order is used within individual references or citations? Can the year or some other characteristic be used as a cue for atomizing this information?
 - What format do figure captions use? What about references to figures?
 - Where can you find information on specimens and specimen gatherings? In types? In separate lists of specimens given for each taxon? In a separate list at the end of the volume? In figure captions? Elsewhere? Which format(s) are used?
 - Where are vernacular names given? In nomenclature only? Or in separate text sections? Both? What formats are used? Can you discover on your own what the local languages are? If not, does a taxonomist or flora editor know?

¹ Indented keys are best manually converted to linked keys at the document clean-up stage (see text preparation.doc), as the white space used for the indents is not the most favourite thing of the average OCR engine. Fixing the indents takes more time than converting the key to another format, especially in keys with dozens to hundreds of leads.

² This is preferred. Do this prior to the document clean-up stage (see text preparation.doc).

- Do tables or lists have any special contents?
- What format do footnotes use? Do they have any special contents, such as nomenclature or descriptions?
- Is there any text between square brackets, or other text that seems to be a comment by the author of the taxonomic work?
- Is superscript or subscript text present³? If so, elsewhere than references to footnotes?
- Is there text present that is bolded or in italics on purpose³? Is this explicitly mentioned anywhere? Is it in descriptions or keys, involving characters? If elsewhere, is it used to emphasize a certain taxon above others?

This is quite a list, and obviously you will not be asking yourself these questions all the time. It is merely a list of the more common topics that you may encounter, and having read it before you start will make you more aware of what you may encounter, hopefully leading to thinking things like "Hey, I read something about that before" when you spot something in the text.

Some of the things mentioned in the list are excellent to use as cues for scripting. The more common ones are explored below.

Keys

Polytomous linked keys (not indented keys) usually use almost entirely unique formats within most legacy taxonomic works, formats that are not used elsewhere in a volume. Therefore writing a script that only marks up keys is simple.

Headings and subheadings

Headings and subheadings are probably one of the easiest cues to exploit for script writing. They generally have a format that is extremely consistent throughout a volume and even a series of volumes. They are almost always located at the beginning of a line and in all capital letters or followed by some specific form of punctuation. Furthermore, they generally precisely identify the contents that follows them. This means you can use them to 'select' a piece of text for processing by a script. Subheadings that are inline with the paragraph that follows them are especially suitable for this approach.

However, because there still is a minor amount of variation in formats, the following approach is more interesting:

- 1) Identify all (or most) possible variations of a certain (sub)heading within a volume.
- 2) Write a script that adds basic mark-up to all identified variations of that heading. This basic mark-up is the same for all variations.
 - a. When the subheading is inline with the paragraph that follows it, the script can also immediately insert basic mark-up for the paragraph.

³ Superscript, subscript, bolded or italics text has its format stripped off during the document clean-up stage. It will not be used for scripts. During proofreading such information will be added back in manually.

- 3) Now write a script that selects only the previously marked up text using the mark-up instead of the heading as a cue and then inserts more detailed mark-up.

As it is possible to select for a single specific (sub)heading, different sections of the script can do similar things for different (sub)headings without any interference between the different parts of the script.

Figure captions and figure references

Figure captions (the text below a figure) can be exploited in a similar vein to (sub)headings. Most of them start with "Figure"⁴ at the beginning of a line, usually involve some numbering (Roman or Arabic) and some specific punctuation, meaning the beginning of each figure caption can be used as a selector.

Figure references are more complex. Usually they involve abbreviations such as "Fig." or "Pl.". However, these abbreviations are also used in literature references, citations and sometimes normal text. Obviously this may cause problems. Fortunately, the environment of the figure references can often be used as an additional cue. They basically occur in three different situations:

- On separate lines, starting with "Fig." or "Pl." or sometimes "Map".
- At the end of text paragraphs, usually separated from the previous sentence using specific punctuation (e.g. long dashes).
- Between parentheses within a text paragraph.

Furthermore, the format of the figure references (especially punctuation) is usually different than the format used in literature references and citations.

Other keywords

Besides the previously mentioned options it is also possible to use specific keywords to 'select' a specific piece of text. This is very useful when certain paragraphs lack headings. A few examples:

- 1) Taxonomic descriptions usually lack a (sub)heading telling you it is a description. It is possible to use the surrounding text to add the basic mark-up to the description, but this can be rather error-prone. There is another way, and that is to use keywords in the description that seldom occur elsewhere, combined with avoiding previously marked up text.

Surprisingly, many habit-related keywords or keyword combinations only occur in the description. It also helps is that the description is usually written in shorthand, e.g. starting with "Plants diecious" instead of "The plants are diecious".

⁴ "Photograph", "Plate", etc.

- 2) In distribution data that lacks a heading, keywords regarding the distribution on the scale of the world (e.g. "northern hemisphere") may work very well, combined with avoiding previously marked up text.
- 3) A list of collector names can be used to find and mark-up specimen information when combined with the way specimen information is formatted.

The absence of specific text

A less intuitive option to select text for processing in a script is to search for text that does not contain specific keywords. This allows you to skip certain text sections. It was actually already mentioned, as this is what makes it possible to avoid previously marked up text.

However, using this is not limited to broadly avoiding other text or previously marked up text; it can be used to select specific text if that specific text never contains keywords or formats used elsewhere. This is very useful.

More options

As may be apparent, useful text in legacy taxonomic works is not limited to alphanumeric text itself, but also includes punctuation. It is often very useful in helping with the selection of specific text sections in combination with keywords. It can also be combined with information other than specific keywords. Brackets, dashes, etc. are often also useful, as are digits.

However, scripts are not limited to using the letters, digits, and punctuation that were already present in the printed work. Fleetingly mentioned was that the start of a line can be used as part of a pattern. So can the end of a line.

It is not required to use the exact characters or digits given in the text. Instead, so-called character classes can be used. These can, for example, include all capital letters, or those capital letters that are exclusively used in Roman numerals, or only non-accented lower case letters, or specific digits, or combinations of punctuation and specific letters.

Special characters with special functions are available, meaning that for example the boundaries between words in a sentence can also be used in a pattern. Furthermore, there are modifiers available that allow to search for repeated use of a letter or digit, amongst other options.

All of these options can be combined to create extremely powerful yet fairly simple search patterns. More information on these possibilities is provided in the second part of this manual, Creating scripts.

Determining the optimum script order

To avoid having to write separate scripts for marking up and eventually atomizing every single type of data contained in a legacy taxonomic work, similar tasks can be grouped into a single script. However, this has a consequence in that scripts cannot

be run in a more or less arbitrary order⁵. Instead, there is an optimum script order, which depends on four factors:

- 1) The number and quality (meaning similarity, mostly) of the bits of text you can use as cues for your scripts.
- 2) The number of mark-up tasks for a specific text section that can be performed in a more or less standalone fashion (with a single script).
- 3) The degree of data atomization you want to achieve.
- 4) Whether the text needing mark-up is globally present (affecting the whole document) or only locally.

The first point means that text that can easily be identified as something specific is easier to mark up first, while text that presents only a few cues is more difficult to mark up, and in some cases should initially be left alone. It follows that anything with a specific (sub)heading should be marked up before anything else lacking specific (sub)headings. It is possible to do this within one and the same script, with some judicious use of the option to avoid text that has already been marked up.

The second point means that text that can be completely marked up using a single script has precedence over text that will need a second pass by another script. For example, the mark-up of keys can be performed entirely with a single script, and therefore can be performed very early in the mark-up process.

The third point, the degree of data atomization you want to achieve, is important because text that will have to be atomised still shares its basic mark-up with text that does not to be atomised. It is easier to add this basic mark-up to all text requiring it at once, and to have a second script do the atomisation of those parts that require such atomisation⁶.

Fourthly, some mark-up tasks may seem simple and stand-alone, but may affect the whole document instead of only a limited part of it. Performing these tasks early on, before atomisation, can needlessly complicate earlier mark-up tasks. An example of this is the mark-up of HTML entities (e.g. & instead of &).

Furthermore, it is better to keep scripts as simple as possible, by splitting up the tasks into smaller subtasks that do not try to do too many things at once. If required, several subsequent scripts can be used. Keeping code simple also improves readability.

Combining all of the above leads to a basic order of scripts that is shown in Figure 1. Note that the "Remaining text mark-up"-task may be partially manual if you really lack cues for use in scripts.

The next part of this manual explains how to create scripts in more (technical) detail.

⁵ Depending on the cues you have available in the original text, the option of an arbitrary script order is rather unlikely anyway.

⁶ It is possible to do both tasks with a single script, but this causes some complications that substantially complicate debugging.

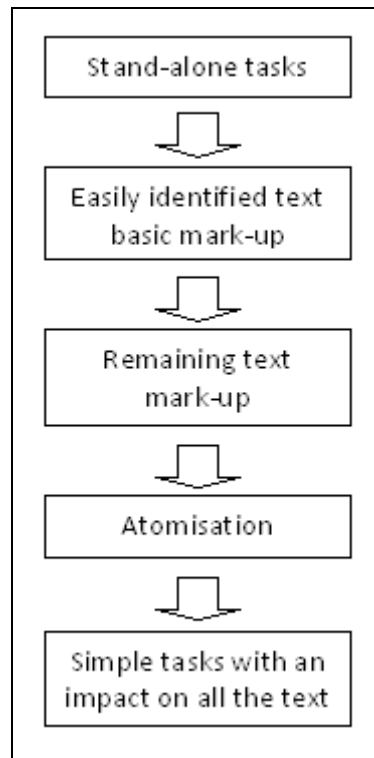


Figure 1: Basic order of scripts.

Creating scripts

This section provides a basic guide on how to create scripts for the purpose of semi-automatically inserting XML into legacy taxonomic works.

If you have never used Perl before, it is suggested that you first learn some basics using one of the books available online. A useful one is <http://www.perl.org/books/beginning-perl/>, of which you should read the first 6 chapters and do the examples given (not only in your head, actually do them on your computer, not only by typing them into your text editor, but by running them). Chapter 5 contains almost all the basic information you will need.

Modern Perl is also a good book (and fun if you have some additional time): http://modernperlbooks.com/books/modern_perl/chapter_00.html

The more technical information found in <http://perldoc.perl.org/> is also very useful, especially the following:

<http://perldoc.perl.org/perlrequick.html>

<http://perldoc.perl.org/perlretut.html>

<http://perldoc.perl.org/perlstyle.html>

...and the documentation they link to.

If you have trouble reading the more technical documentation, learn to read it⁷, because you will need it at one point. Also remember that the documentation does not cover everything, some things need to be tried to discover whether they work or not. Some of those are covered in this manual.

Convention:

In this manual, Perl code is shown in `this font`.

The programmer's mindset

When you are writing scripts (programming) you need to have a certain mindset that is quite similar to switching to a foreign language that is not native. This means that when writing Perl you think in Perl. The legacy taxonomic text you are working on stops being a normal text full of words and sentences you can comprehend and becomes something more; a text full of exploitable patterns that provide cues for the insertion of XML.

The objective of first doing the text analysis and then only starting to create a script is to try to get you in the right mindset. You need to be analytic. Consider the

⁷ Search for terms on Google or Wikipedia if needed.

patterns that are presented to you and the XML you want to insert as a puzzle. Use Perl to find a way to complete your puzzle.

If you are a taxonomist and you have trouble with this, try looking at the legacy taxonomic text as if it were an organism to describe. Organisms have characters (= patterns) that can be used to describe them; some characters are more important than others and can be used to precisely determine what species you are looking at. What you want to find in the text are those characters, those patterns that are essential to the text and its atomisation.

Why Perl?

It has been suggested that other programming languages, such as Python, can also be used to perform these tasks. Although this is indeed true up to a degree, Perl has the advantage of having been created specifically to handle text. Any text. Plain text, HTML, XML, foreign writings, etc. Several features that are practical when handling text are very simple to implement in Perl. For example, Unicode support is a matter of adding *only two* distinct lines at the top of a script. Furthermore, Perl makes it possible to do the same things in various ways, therefore suiting various types and strains of users. Finally, it has really good documentation and a community that is friendly to newbies and oldbies alike.

Required programs

Perl setup

Perl must be installed if you want to test and debug your scripts. Follow the instructions given in **script use.doc** to install it. The same document explains how to run Perl scripts.

Notepad++ setup

Notepad++ (<http://notepad-plus-plus.org/>) is a free advanced text editor that supports many different programming languages and has syntax highlighting (displaying various parts of the code in different colours). If you are not allowed to install programs on your computer, there is a Portable Apps version⁸ available at http://portableapps.com/apps/development/notepadpp_portable.

Go to Notepad++'s "View" menu and switch on the following three options (see also Figure 2):

- "Word wrap", which wraps the text if a line is longer than your screen is wide.
- Under "Show Symbol", choose:
 - "Show White Space and TAB"
 - "Show Indent Guide"

⁸ You will need to install Portable Apps (<http://portableapps.com/>) to a USB stick or your home directory. Normally this does not require administration rights.

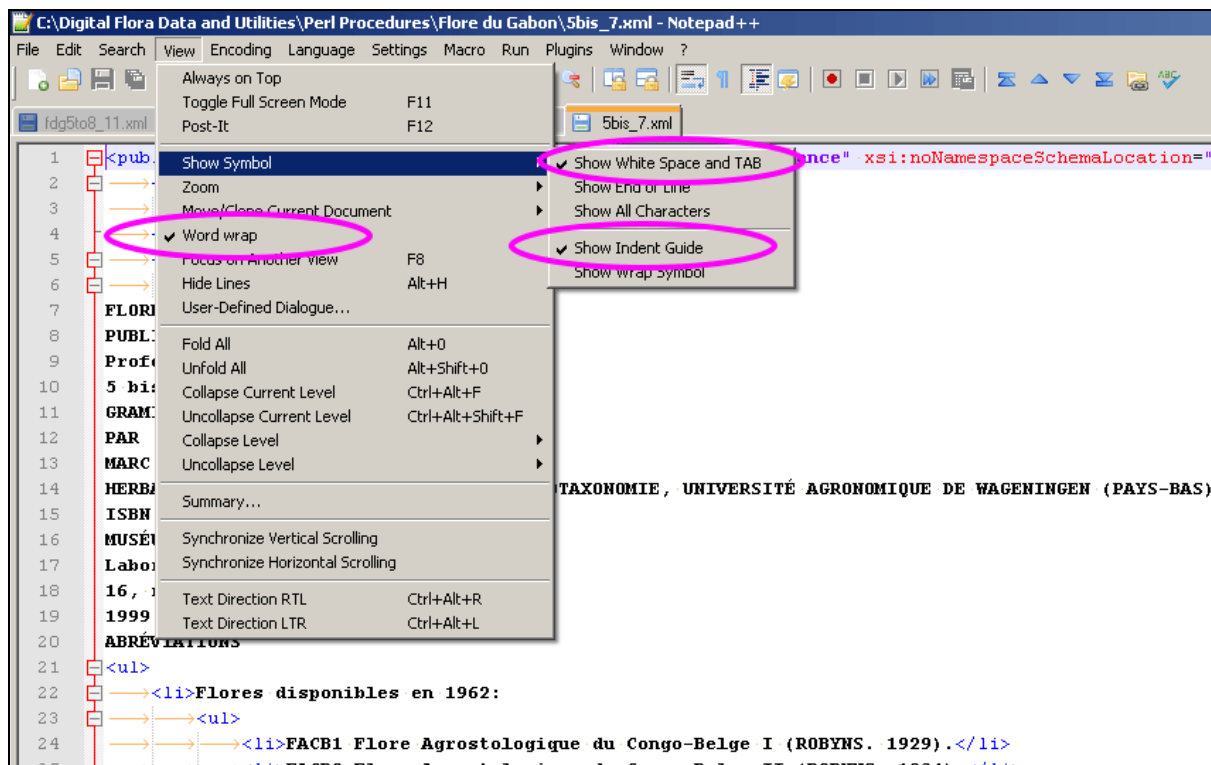


Figure 2: Helpful options in Notepad++.

Optionally, you can start the Plugin Manager, which can be found in the "Plugins"-menu under "Plugin Manager" => "Show Plugin Manager", and install the "Compare"-plugin (Figure 3).

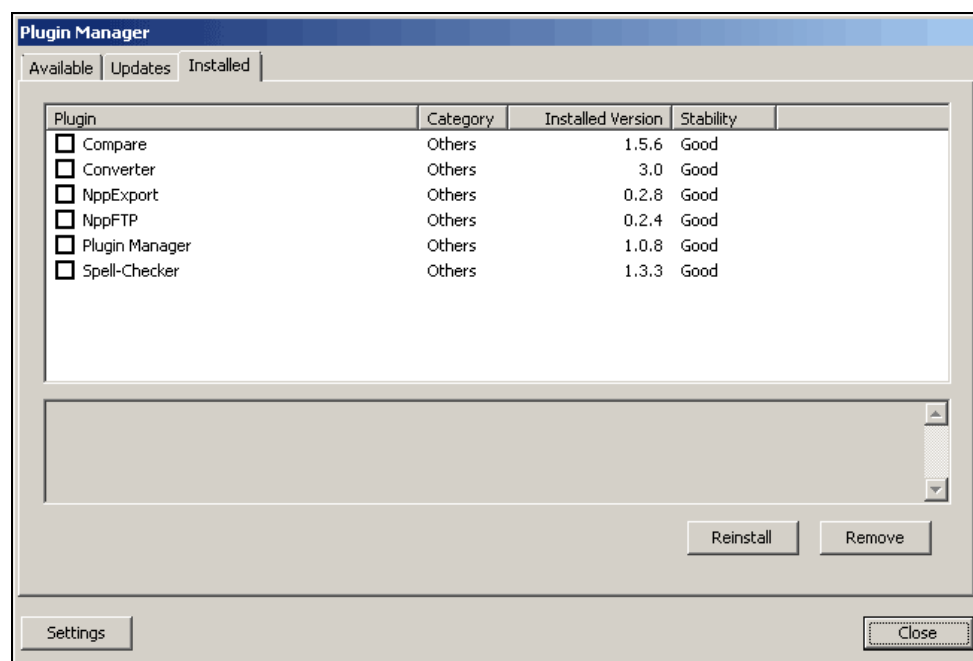


Figure 3: Notepad++'s Plugin Manager.

Saving scripts

When you want to save a script, you click on "Save As..." in the "File"-menu to open the "Save As"-window. In this window, you click on the dropdown box next to "Save as type" and select "Perl source file (*.pl, *.pm, *.plx)" in the list (Figure 4). Then you give your script an appropriate name (Figure 5) – use **.pl** as the file extension! Notepad++ will now recognize this file as a Perl script, and apply the correct syntax highlighting.

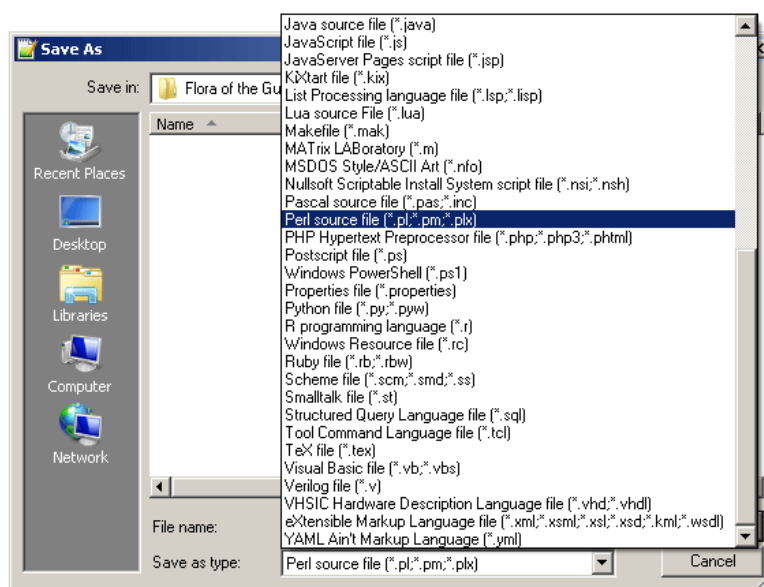


Figure 4: Saving a Perl script with the correct extension.

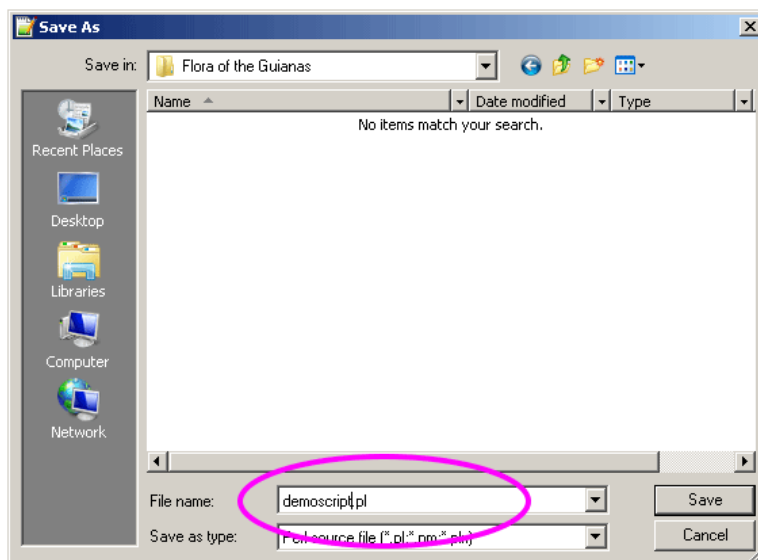


Figure 5: Saving a Perl script.

Script basics

The following explains the basic building blocks that should be present in every script used for the insertion of XML.

Comments

One of the most important things in programming, if not the most important thing, is to properly comment your programs. Comments should not only say what something is, they should describe what something does, or rather what it intends to do, in common, non-cryptic language. This is especially the case if the code that is being commented is complicated. It helps if you also mention on what data the code is supposed to operate.

Code that does not work as intended with comments that explain what it should do is easier to fix than code with no or bad comments.

In Perl, a comment is preceded by a hash (#), e.g.

```
# This is a comment.
```

Comments can be put on separate lines above code or it can follow code.

Normally, each script file should start with some commented lines (Figure 6).

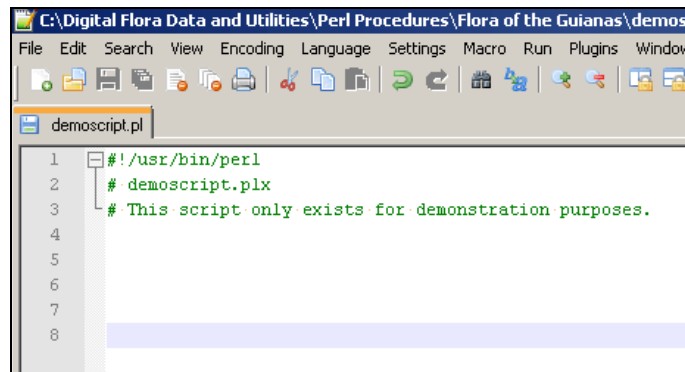


Figure 6: Required comments at the start of a Perl script.

`#!/usr/bin/perl` is used only on UNIX systems, but you should include it in your scripts to ensure compatibility even if your scripts are developed in Windows. Perhaps someone else will want to use your scripts on a UNIX or UNIX-like system (e.g. a Mac).

The second and third lines give the name of the script and a description of what the script serves for.

Note that Notepad++ shows comments in green.

Perl pragmas to use

The following pragmas are used in each Perl script:

```
use warnings;  
  
use strict;  
  
use utf8;  
  
use open ':encoding(utf8)';
```

What `use warnings;` and `use strict;` do is described here:
http://www.perlmonks.org/?node_id=111088

`use utf8;` allows you to use Unicode UTF-8 characters (see <http://www.unicode.org/>) within your scripts, while `use open ':encoding(utf8)';` enables the proper processing of Unicode-encoded text files.

These pragmas are added to the top of your file, just after the initial comments (Figure 7).

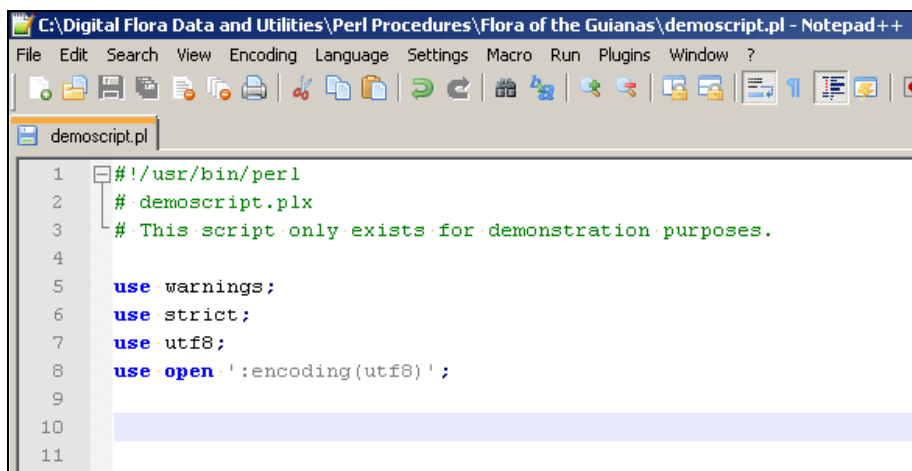


Figure 7: Perl pragmas each script should have.

Tip: Every line that is not a comment should end on a semicolon.

Opening and closing files

Figure 8 shows the Perl code that is required to open and close files.

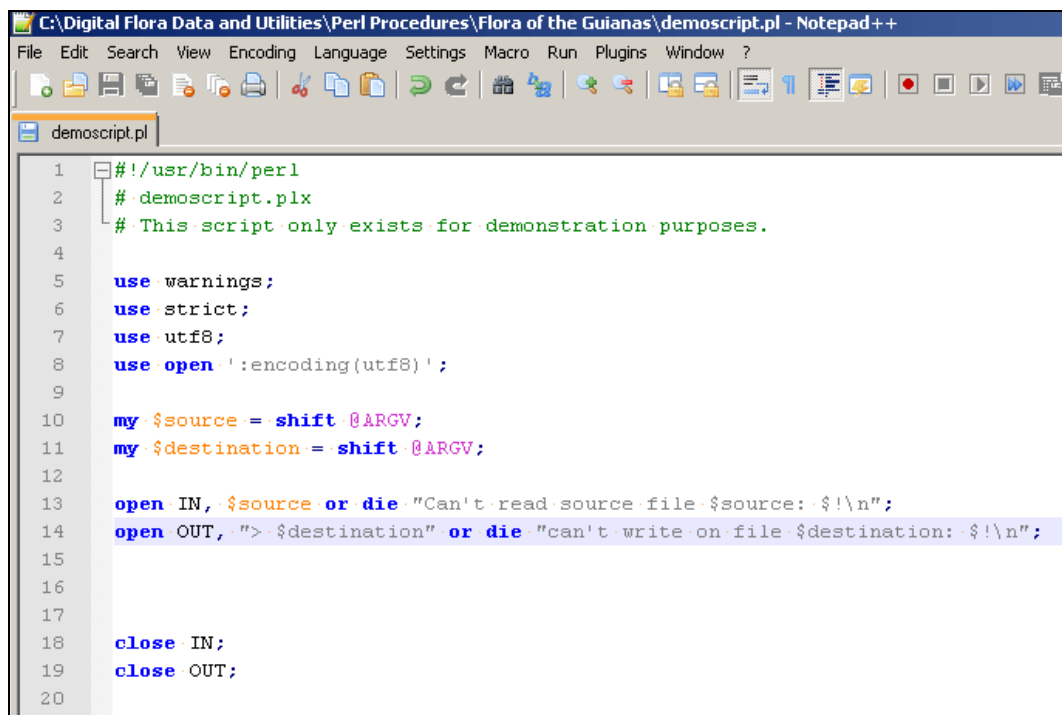


Figure 8: Code for opening and closing files.

```
my $source = shift @ARGV;
```

```
my $destination = shift @ARGV;
```

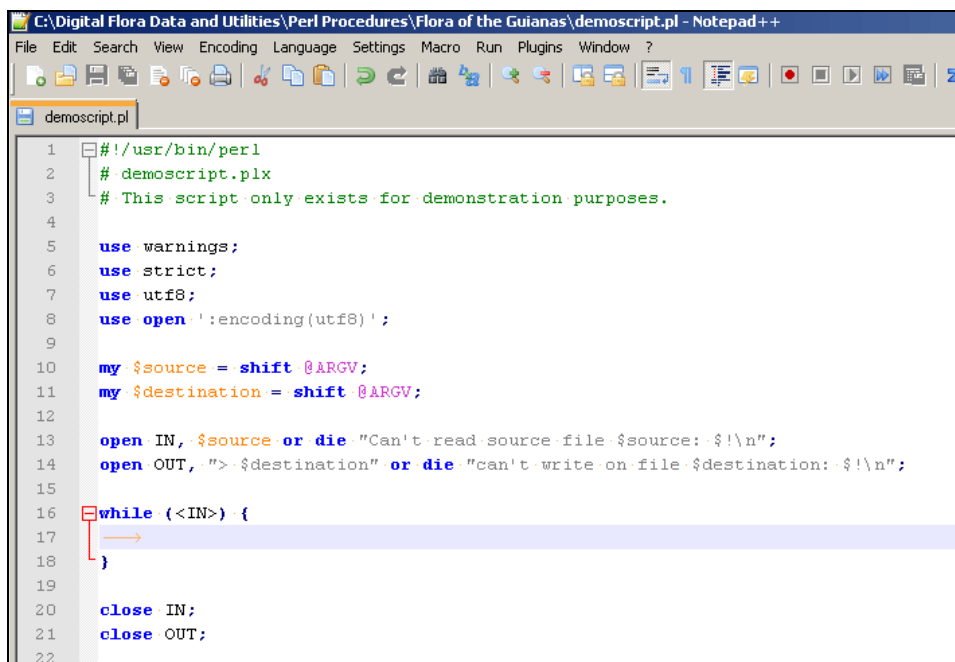
Each of these two lines declares a variable. The first one holds the input (source) file name and the second one the output (destination) file name. These are read from the information you enter at the Command Prompt when running a Perl script (see **script use.doc**). The two files names following the name of the Perl script are interpreted as an array of two values, which are read by `shift`.

The line starting with `open IN` opens the input file for *reading*, and gives an error message if the file cannot be found. The line starting with `open OUT` opens the output file for *writing*, and gives an error message if the file cannot be written.

`close IN;` and `close OUT;` close the input and output files. These two lines should always be the last two lines of a script.

The while loop

Figure 9 shows the script with a `while` loop in the script. What this does is that the code within the `while` loop will be executed as long as the input file (indicated by `<IN>`) has not ended. All code within the curly (`{` and `}`) brackets belongs to the `while` loop. To make this even clearer, the code belonging to that loop is indented by a tab. In Figure 9 no code is yet present, but the tab can be seen on line 17. The `while` loop processes the input file line by line.



```

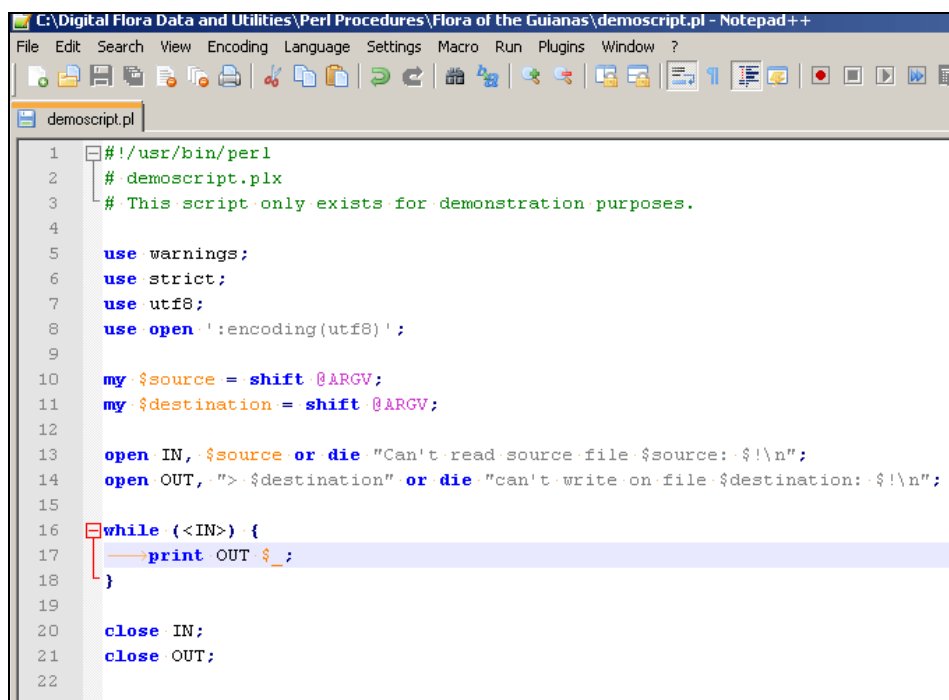
1  #!/usr/bin/perl
2  # demoscrypt.plx
3  # This script only exists for demonstration purposes.
4
5  use warnings;
6  use strict;
7  use utf8;
8  use open ':encoding(utf8)';
9
10 my $source = shift @ARGV;
11 my $destination = shift @ARGV;
12
13 open IN, $source or die "Can't read source file $source: $!\n";
14 open OUT, ">$destination" or die "can't write on file $destination: $!\n";
15
16 while (<IN) {
17     #
18 }
19
20 close IN;
21 close OUT;
22

```

Figure 9: The while loop.

Printing to the output file

If you were to run the program shown in Figure 9, you would discover that the output file is created, but that it is empty. This is because the input file is read, but nothing is done to the lines that are read, including writing those lines to the output file. To do this, the line `print OUT $_;` is added to the script (Figure 10).



```

1  #!/usr/bin/perl
2  # demoscrypt.plx
3  # This script only exists for demonstration purposes.
4
5  use warnings;
6  use strict;
7  use utf8;
8  use open ':encoding(utf8)';
9
10 my $source = shift @ARGV;
11 my $destination = shift @ARGV;
12
13 open IN, $source or die "Can't read source file $source: $!\n";
14 open OUT, ">$destination" or die "can't write on file $destination: $!\n";
15
16 while (<IN) {
17     print OUT $_;
18 }
19
20 close IN;
21 close OUT;
22

```

Figure 10: Printing to the output file.

Now the lines will be written to the output file. However, they are not yet modified. The following sections shortly explain the remaining Perl code that you will use, and then a few example scripts with explanations are given.

Prepending or appending text to a file

In some cases it might be required to add text to the beginning or the end of a file without changing the remaining text. In this case the following should be done:

- To prepend text, add a `print OUT` statement just before the `while` loop, followed by the text you want to add between quotation marks, followed by a semicolon.
- To append text, add a `print OUT` statement just after the `while` loop, followed by the text you want to add between quotation marks, followed by a semicolon.

The `while` loop should still contain a `print OUT` statement too, otherwise the remaining text contents is not written to the new file.

Figure 11 shows a script making use of this.

```

1  #!/usr/bin/perl
2  # pubtags.plx
3  # Pre- and appends the first and last XML tags to a file.
4  use warnings;
5  use strict;
6  use utf8;
7  use open ':encoding(utf8)';
8
9  my $source = shift @ARGV;
10 my $destination = shift @ARGV;
11
12 open IN, $source or die "Can't read source file $source: $!\n";
13 open OUT, ">$destination" or die "can't write on file $destination: $!\n";
14
15 # Prepends the first XML tags required by the FM XML scheme:
16 print OUT "<publication xmlns:xsi=\"http://www.w3.org/2001/XMLSchema-instance\"
17         xsi:noNamespaceSchemaLocation=\"\" xmlns:mods=\"http://www.loc.gov/mods/v3\"
18         lang=\"en\">\n\t<metadata>\n\t\t<defaultMediaUrl>http://wp5.e-taxonomy.eu/media/</defaultMediaUrl>\n\t<
19         /metadata>\n\t\t<treatment>\n\t\t\t<taxon>\n\t\t\t\t";
17 while (<IN>) {
18     # Prints all lines to file:
19     print OUT $_;
20 }
21
22 # Appends the accompanying closing tags:
23 print OUT "\n\t\t</taxon>\n\t\t</treatment>\n\t</publication>";
24
25 close IN;
26 close OUT;
27
28
29

```

Figure 11: Prepending and appending the first and last XML tags.

The `if-else` loop

The other loop used in many scripts is the `if-else` loop. This loop takes the form of:

```
if (condition) {  
  
    do something..  
  
}  
  
else {  
  
    do something else (or nothing)..  
  
}
```

What this loop does is to check whether a certain condition is true or not. If the condition is true, the code that is between the curly brackets following `if (condition)` is run. If the condition is false, the code between the curly brackets following `else` is run, or, if no code is provided there, the loop ends without doing anything. In the scripts used for XML mark-up, the condition will be attempting to match text in the line of the input file that is being processed. The code for the match goes in between two forward slashes, which tell Perl a text match is to be attempted.

There is an expanded version of this loop that allows you to add one or more `elsif` ("else if") statements in between the `if` and `else` part. This makes it possible to check the same text string against multiple options and to do different things depending on which match is true.

This loop is often used when the text that has to be marked up is located at a random location on a line and can also be present elsewhere than in the target text. In such cases you first try to match the target line using a `if-else` loop, and then mark up those parts of the line that need to be marked up.

These loops can be placed inside each other if required. An example is shown in Figure 12. The other code shown in this figure should become clarified in the next section.

How text matching works is explained very well in a lot of the online documentation and books, so the following section will only focus on those syntax options that are used in the scripts for XML mark-up.

```

24      → # Inserts basic character mark-up:
25      → # Format #1:
26      → if (/\. - [[[:upper:]] [[[:lower:]]]) {
27      →   s/( \. ) ([[:upper:]] [[[:lower:]]]/$1<\subChar><\char>\n\t\t\t\t\t<char class="">$3/g;
28      →   # Format #1.5:
29      →   if (/\. - [[[:upper:]] [[[:lower:]]]) {
30      →     s/( \. ) (-) ([[:upper:]] [[[:lower:]]]/$1<\subChar><\char>\n\t\t\t\t\t<char class="">$3/g;
31      →   }
32      →   elseif (/(:|;) ([[:lower:]] [[[:lower:]]]) {
33      →     s/(:|;) ([[:lower:]] [[[:lower:]]|\d+)/$1<\subChar>\n\t\t\t\t\t<subChar class="">$3/g;
34      →   }
35      →   else {
36      →   }
37      → }
38      → # Format #2 (Unique to FM Series II Vol. 2):
39      → elseif (/; - [[[:lower:]] [[[:lower:]]]) {
40      →   s/(:|;) ([[:lower:]] [[[:lower:]]]/$1<\char>\n\t\t\t\t\t<char class="">$3/g;
41      → }
42      → # Single sentence descriptions:
43      → else {
44      → }

```

Figure 12: An example of two `if-elseif-else` loops.

Matching and replacing text using regular expressions

Regular expressions can be used to match specific text patterns and replace them with different text or insert additional text (see <http://perldoc.perl.org/perlrequick.html> for a primer).

In the scripts used for XML mark-up, there are two locations where regular expressions are used:

- In the `if-else` and `if-elseif-else` loops, as mentioned above (for text matching only).
- In substitutions, which are explained below.

Substitutions try to match some text and then replace that text with other text. They take the global form of

s/text match/substituted text/;

s stands for, you guessed it, "substitution".

The text match can be any word or a text string, with or without special characters or grouping.

The substituted text can be any word or text string, with or without special characters, and may or may not involve the text that was matched. As the objective of the scripts in the insertion of XML, usually some or all of the matched text will be reused.

Special characters

Perl has several special characters that have a special meaning in regular expressions. They are the following:

. * ? + [] () { } ^ \$ | \

How they work is explained very well in Chapter 5 of Beginning Perl (<http://www.perl.org/books/beginning-perl/>). Below is a summary of those that are most employed in the script used for XML mark-up:

.	matches any character
*	preceding character or group is present 1 or more times
?	preceding character may be present 0 or 1 times
+	preceding character or group may be present 0 or more times
[] and { }	used for character classes
()	used for grouping
^	matches start of line
\$	matches end of line
	used for multiple options within a group
\	escape character

What you must keep in mind at all times is that when these characters are used in a text string normally (not as a special character) they need to be escaped by preceding them with a backwards slash (\) if you want to match that text string. This means the escape character itself should also be escaped, if it is present in normal text.

The escape character is also used in matching white space characters such as tabs (\t) and newlines (\n). These same white space characters can be used in the substitutions themselves to insert tabs and newline characters.

Grouping and backreferencing

When inserting XML into a text string it is very useful to be to group parts of the text string together. For this parentheses are placed around each of the groups.

For example, if you want to insert some XML tags in the middle of the text string "Flowers unisexual, mostly pedicellate; bracteole present or absent." you could write something like this:

```
s/(Flowers unisexual, mostly pedicellate;) (bracteole present
or absent\.)/$1<XML tags>$2/;
```

What this does is to put the first part of the sentence in one group and the second part in another. Note how the dot at the end of the text string is escaped to avoid that Perl sees it as a special character.

These groups can then be backreferenced in the substitution, which will include the text they contain in the substituted text. The first group is \$1, the second \$2.

The space is not in any group and therefore will have disappeared after the substitution.

So the resulting text would be:

```
Flowers unisexual, mostly pedicellate;<XML tags>bracteole
present or absent.
```

Alternate choices within groups and grouping within groups

Now, if you had two text strings 1) "Flowers unisexual, mostly pedicellate; bracteole present or absent." and 2) "Flowers bisexual, mostly pedicellate; bracteole present or absent." on two separate lines you could write two separate substitutions to insert the XML tags into the middle of each text string. Or you could write:

```
s/(Flowers (unisexual|bisexual), mostly pedicellate;)
(bracteole present or absent\.)/$1<XML tags>$3/;
```

This would result in the following two text strings

```
Flowers unisexual, mostly pedicellate;<XML tags>bracteole
present or absent.
```

```
Flowers bisexual, mostly pedicellate;<XML tags>bracteole
present or absent.
```

The vertical bar (|) is used to indicate there are multiple options for a match within a group. Furthermore, here you can see it is possible to group text within another group.

However, keep in mind that the more parentheses are used the more complicated a regular expression becomes to read. Try not to have more than three or four levels of grouping.

Grouping without backreferencing

Using multiple levels of grouping has one disadvantage: Every group counts towards the total number of groups that can be backreferenced. This is also why the first example above uses \$2 for the last group, while the second example uses \$3 for the last group. If you want to avoid this problem, you can add ?: in front of a match to make it match only and not count for backreferencing.

So using the previous example, that would give:

```
s/(Flowers (?:unisexual|bisexual), mostly pedicellate;)
(bracteole present or absent\.)/$1<XML tags>$2/;
```

And now the last group can be backreferenced with \$2 again.

That said, the previous example could also be written as:

```
s/(Flowers (?:(uni|bi)sexual, mostly pedicellate;) (bracteole
present or absent\.)/$1<XML tags>$2/;
```

Which may be simpler to use in some cases and shows grouping also works within words on character level.

Character classes

Using alphanumeric characters only has its uses, but it is not always known which characters you will encounter. This is where character classes come in. For example, if you wanted to match a single letter at the start of a line and add tags around it, you could write the following:

```
s/^(^)(a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z)/$1<
tag>$2<\tag>/;
```

Not only is this hard to read and error-prone to write, it is also a lot of work. The following does exactly the same thing:

```
s/^(^)([a-z])/ $1<tag>$2<\tag>/;
```

`[a-z]` is a character class that includes all lower case letters from a to z.

Likewise, `[A-Za-z0-9]` includes all upper case and lower case letters and digits 0-9.

You can also create your own character classes. For example if you wanted to match text that only consists of lower case letters and the ampersand symbol, you could create a character class like `[a-z&]`. For some character classes (and characters!) Perl has some nice shortcuts. For example, `[0-9]` can also be written as `\d`.

Now the character classes for lower case and upper case letters shown above are only for the non-accented letters. When processing legacy taxonomic works this may cause problems, especially if the taxonomic work is written in a language other than English. So instead of `[a-z]` and `[A-Z]` use the Posix character classes `[:lower:]` and `[:upper:]` respectively, unless you are absolutely sure that you will not encounter accented letters. These will work properly if you use any recent version of Perl.

Something that is not explained very well in the documentation is that Posix character classes can be combined with normal character classes quite easily. So the earlier example `[a-z&]` becomes `[:lower:]&` when using Posix.

Look-ahead and look-behind assertions and positive and negative matching

The basics of look-ahead and look-behind assertions are well explained in Chapter 5 of Beginning Perl (<http://www.perl.org/books/beginning-perl/>). They can sometimes be useful in substitutions, but are most useful in `if-else` loops.

By combining a match for the start of a line with a negative look-ahead assertion that checks for certain types of contents, it is possible to avoid already marked up text. An example of such a loop is shown below:

```
if (/^(?!\\t*<)(?![[:upper:]]+\\.<br \/>$)(?![[:upper:]]+<br \/>$)(?![[:upper:]]+<\\string>$)(?![[:upper:]]+<\\feature>$)(?!.+xmlns:xsi)/) {

}

}
```

Modifiers

In Figure 12 you can see that the substitutions have a `g` between the last forward slash and the semicolon. This `g` is a modifier that stands for "global" and tells Perl the substitution process should not stop after the first match and substitution, but should be performed for all matches in that particular line.

There are other modifiers, but normally none of them need to be used in the scripts.

Useful strategies for script design

Order of regular expressions within scripts

The order of the regular expressions within a script is important, especially at the atomisation stage. Once the basic mark-up for most text has been inserted based on easily exploited cues such as (sub)headings, what is left has to first be split up over several lines and prepared for atomisation. In atomisation scripts, the regular expressions for the most extensive patterns should come first, followed by sequentially simpler and simpler regular expressions.

This is because the simpler regular expressions generally can match a larger part of the contents that has to be atomised. Using those regular expressions first would cause a large number of mismatching to occur, which would take a lot of time to correct. This is especially a problem in nomenclature, literature references and citations.

Order of terms within groups

The order of the terms within groups where multiple options are given using the vertical bar (`|`) can be quite important for successful mark-up. When the simpler terms happen to repeat part of a more complex term, the order of terms should be similar to the order of regular expressions: first the complex terms, then the simpler ones.

When using Grouping without backreferencing, it is possible to use the vertical bar followed immediately by the closing bracket for the group to make the group (without backreferencing) optional, e.g.:


```
(?:large |small |)leaf
```

This will first check for “large leaf”, then for “small leaf”, and finally for “leaf” only.

Fixing after the fact

However, in some cases it is easier to first 'incorrectly' mark up some data, and then to write a regular expression that fixes those cases where the mark-up went wrong. This is especially the case when the data in question are difficult to split prior to mark-up or when you start processing a new volume and discover that previously working regular expressions fail in a repeatable manner due to a small format change that would otherwise require many regular expressions to be rewritten.

Reuse earlier scripts

Basing new scripts off older scripts that are known to work can be an effective way of reducing development time.

Debugging scripts

Debugging a script consists of two stages.

Firstly, the script should successfully run. If you added the proper pragmas at the start of your script and running your script is aborted due to syntax errors, an error message pinpointing the location of the problem should be displayed. Figure 13 shows an example of such a message.

```

C:\>c:
C:\>cd digital flora data and utilities
C:\Digital Flora Data and Utilities>cd perl procedures
C:\Digital Flora Data and Utilities\Perl Procedures>cd flore du gabon
C:\Digital Flora Data and Utilities\Perl Procedures\Flore du Gabon>perl features
fr.plx fdg34to35_7.xml test666.xml
Unmatched ( in regex; marked by <-- HERE in m/( <-- HERE </collector>( : )/ at fe
aturesfr.plx line 60.
C:\Digital Flora Data and Utilities\Perl Procedures\Flore du Gabon>

```

Figure 13: A syntax error halts running a script.

Secondly, you should check in the output XML file that the script did what you intended it to do. If not, modify the offending regular expression. Continue debugging until you are satisfied with the result.

If you do not debug, ever, you will likely be wasting a lot of time and end up with output files where you do not know what went wrong where. This is not a good perspective.

Types of scripts

There are two main types of scripts that you will have to create: clean-up scripts and mark-up scripts.

Clean-up scripts

When you use scripts to perform mark-up of legacy taxonomic works, you will likely find that a lot of problems are caused by errors or inconsistencies in the original text, either due to typos in the original work or due to OCR-errors. The purpose of the clean-up scripts is to reduce these problems by automatically fixing reoccurring errors.

There are two clean-up scripts. One fixes commonly occurring problems in the punctuation and standardises dashes and the like, while the other fixes OCR-errors that often occur in a particular taxonomic work.

Mark-up scripts

As shown in Figure 1 the mark-up scripts will have to be created in a certain order. The exact script order depends on the taxonomic work and needs to be custom-tailored to it.