Across Languages and Cultures 4 (1), pp. 109–132 (2003)

PERL SCRIPTING IN TRANSLATION PROJECT MANAGEMENT

GRZEGORZ CHRUPAŁA

Intercultural Studies Group, Universitat Rovira i Virgili,
Placa Imperial Tarraco 1, E-43005 Tarragona, Spain
E-mail: grchrupc7@docd4.ub.edu

Abstract: The field of translation in general and of technical translation in particular has become increasingly dependent on the use of various electronic tools. Computer assisted translation uses specialized applications that partly automate the production of multilingual software and documentation. However, for some needs these applications are not flexible enough. In these situations programming becomes indispensable. Scripting programming languages such as Perl provide a perfect platform for rapid solution of specific, short-term problems in a fully customizable way.

Key words: translation, scripting, programming, CAT, Perl

1. INTRODUCTION

The process of producing a technical translation normally involves a series of subtasks. Sometimes all these are performed by one and the same person but most commonly they are assigned to professionals who are specialized in some subset of the tasks. This text focuses on processes mostly carried out by either translation managers or Information Technology professionals. Thus we will be dealing neither with specifically translatological problems nor with the logistics of technical translation. Rather, we will be talking about those parts of the translation process most dependent on the digital support of documents and consequently heavily relying on computers and computational techniques. We will also mainly focus on the processing of "static" electronic documents and only tangentially touch upon issues related to localization (i.e., translation and adaptation of software).

1.1. CAT Tools

Ever since most texts started to be produced on digital rather than physical support, electronic tools automating translation processes have been gaining importance. These tools include specialized editors, translation-memory management applications, terminology management applications, alignment tools and format-filtering components. All or some of these can be built into a single modular application that provides an integrated Computer Assisted Translation (CAT) environment. Some well-known applications include Trados Translators Workbench, STAR Transit or Atril's Déjà Vu.

CAT applications have proven enormously useful and have consistently improved productivity in translation and arguably also translation quality. By means of re-using existing resources, translation memories relieve technical translators from some of the repetitive nature of their job, permitting them to concentrate on the more creative parts of translation. Alignment tools make it possible to create translation memories from parallel texts in a semi-automatic way. Filters and specialized editors with import/export functions permit the translator to work in a familiar environment without needing to learn all the different applications that are currently used to produce documentation. Finally, the terminology management applications integrated into CAT tools streamline both the terminologist's and translator's work by providing a flexible digital support for specialized dictionaries.

However, by their very nature, these applications have some limitations. They are designed so that they can be used by an averagely computer literate translator or project manager. They are almost exclusively Graphic User Interface (GUI) based and as such provide little extensibility. The functions that are available are those and only those that the GUI designers programmed into the user interface, and these tend to be the most common, routine tasks required by the majority of users most of the time.

There are situations when existing CAT tools are inadequate on their own, for example:

(1) Your processes are untypical and as such are not provided for by common CAT suites;
(2) You need to perform tasks that are too specific to be covered by a CAT application.

If you are regularly in situation (1), then you probably need a long-term CAT solution tailored to your untypical needs. This involves software development and falls outside the scope of the present work. However, if you find yourself in situation (2) you don't really need a heavy-duty long-term solution: you just want a quick and easy-to-program fix. And this is where scripting languages show their power.

1.2. Scripting

A script is a series of commands that automates relatively uncomplicated tasks involving electronic data. There are different ways to write scripts.

- You make a file with a series of commands provided by your operating system. On Unix these are called shell scripts; on Windows they are known as batch files.
- You write a program in a programming language of your choice.

Often scripts are written in scripting languages which provide features that make it easier and quicker to write, to debug and to use such relatively non-complex programs. Some of these features are:

Dynamic typing: You do not need to predeclare your variables or specify their type. So if you want to store some integer value in variable `myint` you do not need to specify earlier that `myint` is a variable of type `integer`. You can just say `myint = 8`, use the variable, and maybe later assign a string to it.

Interpreting rather than compiling: There is no need to first compile the program in order to use it. You just type your script in a text editor, save it and run it by invoking an interpreter which translates the source code on the fly and executes the result. This means you are able to develop and test your programs faster.

High-level programming: You do not need to know or care about low-level operations such as memory allocation. You can concentrate on your data and what needs to be done about them. Your programs are compact and easy to understand.

Another important factor to consider when choosing a language for scripting is how easy it is to learn. Since you are going to try to find quick solutions you do not want to spend too much time just trying to figure out the basics.

2. PERL

Languages often used for scripting include Awk, Tcl, Scheme, Perl, Python and Ruby. Languages such as C, C++ and Java are good for systems and applications programming, but are awkward when you need a small, quick solution because they lack some or all of the features enumerated above. In this text I will use examples in Perl.

2.1. What is Perl?

Perl is a high-level general purpose programming language developed by Larry Wall (Wall et al. 2000). It is widely used in quick prototyping, system utilities, software tools, system management tasks, database access and world wide web programming. Is has also been employed in corpus and computational linguistics since one of its main strengths is its comprehensive support of text processing and powerful regular expressions.

As already observed, there are many different ways scripts can be written and used. You could use shell scripts to do most of what we are going to cover. But using a programming language such as Perl has a few clear advantages:

Perl is available for virtually any operating system. Whether you are on Mac, Linux, Windows or anything else, you can use most of the same code. With shell/batch scripts you are restricted to the set of commands provided by your operating system, and on some of them this can be very limiting. Perl is a full-blown programming language, so you can basically achieve anything that is programmable. There is a large repository of modules and scripts publicly available on CPAN (Comprehensive Perl Archive Network) at http://www.cpan.org/. If you need to do something, it is probable that someone has already written a program that does what you need.

Perl is covered either by the GNU General Public License or Artistic License, which ensures that it remains free, open-source software. You can download the source code from CPAN. Binary builds for many platforms are also available (for example from http://www.activestate.com, for Windows, Linux and Solaris).

While Perl is very versatile and perfectly suitable for most types of tasks discussed in this text, some people might prefer to use a different scripting language. For example Ruby, developed by Yukihiro Matsumoto, is very similar in spirit to Perl but has fuller and higher level support for object-oriented programming. Another choice is Python, which also provides a nice, simple interface to object-orientation. Another very flexible language is Scheme, which is a small, simplified dialect of the functional programming language Lisp. These three languages have a smaller user-base than Perl and for this reason there is less reusable code publicly available for them. Whichever one you choose, most of the techniques illustrated with Perl code in this text should be transferable to Ruby and Python and probably also to other languages.

2.2. The Perl Basics

This section is not meant to be a Perl tutorial: there are already many very good tutorials and books available: the classic reference is Programming Perl by Larry Wall et al. (2000). Another very useful source is the Perl Cookbook (Schwartz and Christiansen 1998). Below we simply illustrate some chosen features of the language.

Note: At the time of writing the latest stable release of Perl was 5.8.0 and the code in this text was tested on this version. A major rewrite of the language is underway. This modernized Perl will be called Perl6.

2.2.1. Syntax

In Perl, statements normally end with a semicolon and anything following `#` until the end of line is ignored (this is used for comments). Blocks are enclosed in curly brackets. There are many predefined variables with names consisting of punctuation characters.

2.2.2. Variables

Variables are where you can store data; in Perl they have a different prefix depending on the type of value they contain:

- `$` (dollar sign) means the value is a scalar, e.g., a single number, string or a reference to an object;
- `@` (at sign) means the variable contains an array – roughly, an ordered list with an integer index assigned to each element;
- `%` (percent sign) is used to mark hashes – or unordered sets of `key => value` pairs whose keys are unique scalars.

An example should make their use clear:

```
$firstName = ‘Nim’;
$surname = ‘Chimpsky’;
@fruit = (‘apples’, ‘bananas’, ‘coconuts’, ‘guavas’);
%gloss = (
    apples => ‘manzanas’,  bananas => ‘plátanos’,
    coconuts => ‘cocos’,  guavas => ‘guayabas’);
print “$firstName $surname likes”,
 “$fruit[0] and $fruit[1].\n”;
print “A $surname le gustan “,
 “las $gloss{apples} y los $gloss{bananas}.\n”;
```

The above program will print out the following two lines:

```
Nim Chimpsky likes apples and bananas.
A Chimpsky le gustan las manzanas y los plátanos.
```

2.2.3. Functions

Functions in Perl accept arguments or lists of arguments and return values. For example the `split` function can take a regular expression as its first argument and a string as the second one, and will return a list of strings resulting from dividing the original string wherever it matches the regular expression provided.

`join` does (more or less) the opposite: it takes a string and a list of strings and returns a string which results from concatenating the elements in the list, and putting the first argument string in between them. Some functions are used more for their side-effect rather than for their return value. The `print` function returns `1` if successful, but also prints its arguments on the current output (e.g., the terminal window). And functions can be combined so that what one returns becomes the argument(s) of the other. For example, if variable `$myMates` contains the string

```
‘tom@host.net;dick@host.net;harry@host.org’,
```

then this line:

```
print( join(“\n”, split(/;/, $myMates) ) );
```

will print the following:

```
tom@host.net
dick@host.net
harry@host.org
```

The special character `\n` means a newline. In the above line, first the string is split at semicolons, the resulting list of strings is then joined with newlines, and the resulting string is printed to the current output device.

You can define your own functions (then they are also called subroutines) with `sub`.

```
sub FahrenheitToCelsius {
    my $f = $_[0];
    return ((5/9) * ($f-32));
}
print ‘Water boils at’, FahrenheitToCelsius(212),
 ‘degrees Celsius’, “\n”;
```

You retrieve any arguments given to your function from the special variable `@_`: its first element is `$_[0]`, second `$_[1]`, etc. You can also simply say `shift(@_)` to remove and retrieve the first element of `@_`, and even abbreviate it to just `shift` (`@_` will be understood).

Above we have saved the argument given to the subroutine in a locally scoped variable `$f`. What makes it locally scoped (visible only inside the enclosing block) is the use of `my`. The function will return the last expression evaluated or whatever you specify as an argument to `return`: in this case the temperature converted to the Celsius scale.

2.2.4. Control Structures

Control structures are what makes programming languages such a powerful tool. The most important ones are if-statements and loops.

If-statements

These allow you to execute some commands or not depending on some condition. In Perl they have the following form:

```
if ( condition1 ) {
  action1
} elsif ( condition2 ) {
  action2
} else {
  default action
}
```

The `elsif` and `else` blocks are optional.

Loops

These can take many different forms. The most useful ones are `foreach` and `while` loops. With `foreach` you can iterate over a list and do something with each of its elements. For example, to print the fruit dictionary stored in `%gloss` we can use the following loop:

```
foreach my $fruit ( keys %gloss ) {
    print “In Spanish $fruit is $gloss{$fruit}\n”;
}
```

We use `keys` to make a list of the keys in `%gloss` for the loop to iterate over. On each iteration the next element of the list is assigned to the `$fruit` variable, which we then print out and use to access the values in `%gloss`.

With `while` loops you can execute a block for as long as a certain condition holds. The general form is this: `while ( condition ) { action }`. This control structure is often used to process, for example, each line or each paragraph in a file in turn.

2.2.5. Input/Output

For Perl programs to be useful they need to be able to communicate with the outside world. One way to exchange data is by means of a terminal window. The user can input data with the keyboard and the program can print information on the screen. This is normally done by reading from the STDIN (standard input) and writing to STDOUT (standard output) handles. We can get data from the user in the following way:

```
print “What language do you want to use?\n”;
$answer = <STDIN>;
chomp($answer);
if ( $answer eq ‘English’) { print “Hello\n”}
elsif ( $answer eq ‘Polish’) { print “Cześć\n”}
else { print “I don’t speak $answer\n”}
```

A few things to note here: <> is used to read from a filehandle; the chomp function deletes the return (\n) character from the string keyed in by the user; the eq operator is used to compare strings (for numbers == would be used).

Perl can also read and write data to handles assigned to files. You can open a filehandle with the open function like this:

```
open( $read, “<”, ‘fileA.txt’);
open( $write, “>”, ‘fileB.txt’);
open( $append, “>>”, ‘fileC.txt’);
while ( my $line = <$read> )  {
    unless ( $line eq “\n”) { print $write $line }
}
```

The filehandle will be accessible via the first argument you give to open (a scalar variable in this case but you can also use a bare-word identifier such as IN instead), the second argument specifies if the file is to be opened for reading, (over)writing or appending, and the third argument is the name of the file. Above we have used a while loop to bind each line read from $read to the $line variable and then print it out to $write if it does not consist of a single newline. unless is, of course, like a negated if-statement. There is a very useful special null filehandle which you can use with a while loop like this: while (<>) {block}. With this loop you can give your Perl script a list of files on the command line and it will treat them as if they were one continuous file and read them (unless you specify otherwise they will be read line by line):

```
open( $append, “>>”, ‘appended.txt’);
while( my $line = <> ) { print $append $line }
```

If your script is called append.pl, then the contents of file1 to file4 will be appended to the file appended.txt.

```
> perl append.pl file1 file2 file3 file4
```

The contents of `file1` to `file4` will be appended to the file `appended.txt`. If your shell does wildcard expansion, you can invoke the script like:

```
> perl append.pl *.html
```

and Perl will read all files in the current directory whose names end in `.html` and append them to `appended.txt`.

2.3. The Perl Library

One of the main advantages of Perl over other scripting languages is its vast library. Perl has been widely used for many years and has accumulated a substantial body of code developed by users for their needs and submitted to CPAN. Most of this code is covered by the same licensing terms as Perl itself.

2.3.1. Using Modules

Perl makes code reuse easier by providing a simple module system. A module is a container for code that provides a user interface to the functionality it implements. Some modules are included in the Perl distribution: these form the Standard Library. Others are available for download on the Internet, most notably at CPAN sites. These have to be installed before they can be used. There is one module, CPAN.pm, that automates finding, retrieving and building modules from CPAN.

A module's interface is normally described in the documentation that accompanies it. The simplest way to start using a module is to read this documentation, and put a `use` statement at the top of the script. It has the following form:

```
use <ModuleName>
```

For example, in order to be able to use the `Encode` module to convert between different text encodings simply say `use Encode` at the beginning of your script.

Modules can use two different types of interface:

- Functional – The module exports functions that the user can call in the same way as built-in Perl functions.

- Object-oriented – In this style of interface, the user creates objects defined by the module and calls methods on these objects. One needs a basic understanding of OO-programming in order to use this style of interface.

Module names are usually nested, that is, the part before `::` indicates the broad category to which the module belongs. For the purposes of text and natural language processing, the following categories, among others, contain useful code: `Text`, `HTML`, `XML`, `String`, `Lingua`. Use of modules is illustrated in some script examples in Section 4. For example, we use `HTML::TokeParser` to process HTML files (Section 4.1), and `XML::Writer` in order to produce XML output (Section 4.2.1).

3. REGULAR EXPRESSIONS

Regular expressions are a tremendously useful feature of Perl and one of the principal reasons for its popularity. Their use, however, is not restricted to Perl: the Unix operating system supports them, many text editors do as well, and even some word processors and CAT tools do, too. Regular expressions are available in many programming languages but Perl's version is often regarded as the most powerful one.

3.1. What Are Regular Expressions?

Regular expressions are a way of describing strings in a compact way. Their main use is to search texts for sequences of characters that match certain patterns and possibly replace them with other strings. The notion of regular expressions comes from formal language theory and automata theory. They are capable of describing the same languages as the related notion of regular grammars – the most constrained type of grammars in the Chomsky hierarchy – and such languages (regular languages) are also the ones expressible by means of finite state automata. This means that regular expressions can be computationally implemented as this kind of automata (Jurafsky and Martin 2000).

As already noted, there are different applications that use regexes (as the name regular expressions is often abbreviated) with slightly differing syntax. Here I will concentrate on the Perl version of them.

3.2. Perl Regex Syntax

A regular expression is a pattern a string can be matched against. All the alphanumeric characters match themselves, e.g., `/April 29/` matches with the string `April 29`. There are also characters with special meaning. The most important ones are:

- `.` is the wildcard – matches any character. So `/.ove/` matches `dove, love, move, qove,` etc.
- `[]` defines a character class. The regex `/[ieaou]/` matches any "vowel" letter.
- `/[a-z]/` matches any lowercase English letter.
- `-` (hyphen) is used to indicate ranges inside `[]`.
- ^ (caret) as a first character inside square brackets negates the character class. `/[^ieaou]/` will match anything that is not a "vowel" letter. Outside of square brackets, the caret matches beginning of line.
- `$` matches end of line. `/^$/` matches an empty line.
- `{}` encloses number of times previous character has to appear. `/20{4}/` matches `20000`.
- Kleene star (or asterisk) indicates previous character appears zero or more times. `/20 *000/` matches `20000, 20 000,` etc.
- `+` previous character has to appear at least once. `/^.+$/` matches a non-empty line.
- `?` previous character has to appear zero times or once. `/p?sicosis/` matches `psicosis` or `sicosis`. If it follows another modifier such as `*` or `+`, then it controls the greediness of the match. `/<.*>/` will match as many times as it can (in the following string, it will match all of it: `'<p></p><br />'`). `/<.*?>/` will match as few times as it can: so the first time it will match `<p>`, the second `</p>,` etc.
- `|-` separates alternatives. `/Ch(om|imp)sky/` will match either `Chomsky` or `Chimpsky`.
- `()` parentheses are used for grouping and for capturing matches. With `/(Marie) +(Curie)/` we can capture and store `Marie` in `$1` and `Curie` in `$2`. This can be used to replace all occurrences of `Marie Curie` with, for example `Curie, Marie`.
- \ the backslash can have two meanings:
  - o Treat the next special character as literal. `/\*/` will match an asterisk.
  - o Treat the next alphanumeric character as special. Some of these are:

– `\n` newline
– `\t` tabulator
– `\s` whitespace (newline, tab or space)
– `\w` alphanumeric character
– `\d` same as `[0-9]`

The above are just the bare bones of Perl regular expressions. There are many more useful extensions. They are used in Perl programs with different modifiers that follow them. The two most important ones are `i`, which makes a pattern case-insensitive and `g,` which makes the regex match all occurrences of a matching string instead of just the first one. The string to be scanned is specified with the `=~` operator. If you just want to match, precede the regex with an `m`; if you want to substitute, prepend an `s`. An example will clarify;

```
$string = ‘M&aacute;s Pl&aacute;tanos que
Maracuy&aacute;s’;
$string =~ s/&aacute;/á/;
print $string, “\n”;
$string =~ s/&aacute;/á/g;
print $string, “\n”;
if ($string =~ m/plátanos/i) { print “ñam, ñam\n”}
```

The first substitution will only replace the first HTML entity `&aacute;` with the corresponding accented letter. The second substitution will replace all the matches because of the `g` modifier. The if-statement will not modify the string: it will simply execute its block if the condition is true – in this case if the string matches the case-insensitive regular expression. So we will get these three lines:

```
Más Pl&aacute;tanos que Maracuy&aacute;s
Más Plátanos que Maracuyás
ñam, ñam
```

Another useful operator is `e` which evaluates any expressions in the regex. This allows you to give the match to a function and substitute the result. The following regular expression multiplies all numbers in `$string` by 2.

```
$string =~ s/([0-9]+)/$1*2/eg
```

Together with the null filehandle and one more special variable `$^I`, regular expressions give us very flexible search-and-replace possibilities. What `$^I` does is to turn on the in-place edition. For example, the following code makes backup copies of all files given to it on the command line and then replaces prices in pesetas with prices in euros.

```
use utf8; use locale;
$^I = ‘.bak’;
while (<>) {
  s/([0-9])[\. ]([0-9])/$1$2/g;
  s{([0-9]+) *p(ese)?tas}
    { sprintf(“%.2f”, $1/166.38) . ‘€’}egi;
  print;
}
```

The string you assign to `$^I` is suffixed to each backup file name. Inside the `while` loop there are three lines which seem a bit cryptic. They make use of another implicit variable `$_` similar to `@_`, but scalar rather than array. The while loop assigns each line of input in turn to `$_`. Functions inside the loop are assumed to take this variable if no argument is provided.

The first substitution regex removes any spaces or thousand-separator points from numbers. The second converts pesetas to euros: a dot operator (`.`) is used to concatenate the result of the division with the euro symbol string. The print function finally prints the modified line back to the file. This is a very crude script and it could be improved in numerous ways, but it shows how with a few lines of code we can perform some useful and not so simple text operations.

4. SCRIPTS IN TRANSLATION

Having picked up some Perl or other scripting language, you can start automating some of the more tedious tasks you encounter in translation project management. In this section, some chosen examples of such computationally improvable processes will be reviewed. One factor to bear in mind is that Perl scripts are text-oriented: it is very easy to deal with text formats such as plain text, markup (HTML, SGML/XML, LaTEX, RTF), or CSV. It is less straightforward to process directly proprietary binary formats such as those typically output by office suites. You can still work with those if you save the document in some text format, or if you find a library module that supports your format.

4.1. Search and Replace

The most obvious application of scripts in translation project management is probably to use them to bulk search-and-replace. This sort of task is so common that there are even some specialized applications that do just this. All they do, and more, can also be achieved with Perl scripts. A script which does something uncomplicated such as search-and-replace will mostly fit on a single line, so it is unnecessary to save it in a file: you can simply type on the command line.

A common search-and-replace operation involves deleting hard linebreaks from plain text files (such as README files) while at the same time conserving the double or multiple linebreaks that separate paragraphs (useful if you want to edit a plain text file in a word-processor). One solution is to conserve the double newlines and delete the rest. Here is a oneliner that does just this:

```
> perl –i“.bak”–p –00 –e “s/\n([^\n])/ \1/g”myfile.txt
```

This is pretty dense. Here is what is going on in this line: The `–i` option is the command line equivalent of the `$^I` variable: it makes a backup file named `myfile.txt.bak`. The next two options are `–e` which tells Perl to evaluate the code that follows between quotes, and the `–p` option which wraps this code in a `while(<>){...print}` loop. The `–00` option sets the default record separator to the null string, which makes the script read the file paragraph-by-paragraph instead of line-by-line (so we can check what follows a newline!). This is equivalent to saying `$/ = ‘‘` in your script. Then there is the actual code: a substitution regex that replaces with spaces all those newlines that are followed by a character other than a newline (`\1` is the same as `$1` except the latter will not work on command line). The same code written in a less elliptical style would read:

```
$^I = ‘.bak’;
$/ = ‘‘;
while( my $text = <> ) {
    $text =~ s/\n([^\n])/ $1/g;
    print $text;
}
```

But it is often more convenient to type such simple, one-off operations directly on the command line without having to save them in a file.

You might use the following oneliner to remove HTML markup from all `.htm` and `.html` files in a directory.

```
> perl -i’.bak’ -p0777e “ s/<.*?>//g” *.htm*
```

Here we have combined three options `–p,–0777` and `–e`. The `–0777` makes Perl slurp the whole file at once instead of line-by-line. This is necessary, because some HTML tags can span more than one line. The code that removes the tags is rather simplistic and will not work for some complicated cases such as tags embedded in HTML comments and the like. To treat these cases properly it is better to use a module such as HTML::Parser or HTML::TokeParser. For example, the following script will strip HTML tags form the file `index.html` in a less error-prone way and will put the result in `index.txt`.

```
use HTML::TokeParser;
open($strip, “>”, ‘index.txt’)
  or die “Can’t open: $!”;
$p = HTML::TokeParser->new(‘index.html’)
  or die “Can’t open: $!”;
while( my $token = $p->get_token ) {
  if ( $token->[0] eq ‘T’) {
    print $strip $token->[1];
  }
}
```

The module parses the input file and returns tokens of type ‘S’ (start tag), ‘E’ (end tag), ‘T’ (text), etc. You then simply check the type of token returned and print it out or not. The module does most of the work for you.

Another interesting oneliner will insert thousand separators into all numbers in a file or a series of files.

```
perl -i “.bak” -pe “while (s/(\d)(\d{3}[\.\s])/\1 \2/)
{}” *.txt
```

This code inserts separator spaces but it could put in dots or commas depending on the locale. It works by introducing a second internal `while` loop that repeatedly scans the line for sequences of four digits followed by a decimal period or a space and inserts a space after the first digit. When all the necessary spaces are inserted it prints the line and goes to the next one by virtue of the external (implicit) `while` loop. Note that the scanning and inserting is packed into a single substitution regex in the condition to the internal `while` loop; that is why the block that follows is empty.

There are many more things that can be done with simple compact scripts like the ones above. Many of them are very project-specific and thus of limited general interest. According to your needs, they can easily be written on the fly.

4.2. Acquiring Terminology

An area where scripting is a virtual must is terminology acquisition. The terminology management systems that normally come with CAT applications are perfectly suited to deal with already existing, highly structured terminological resources that reside in databases.

They also provide some import features that help to acquire or upgrade terminology in legacy formats. But these features are fairly useless if you have to import such notoriously unstructured sources as PDF documents or electronic glossaries in HTML distributed among hundreds of files. These are normally optimized for visual appeal rather than for structuring and separating information. Below we are going to see examples of how these sources can be processed by means of custom Perl programs. Relatively uncomplicated examples have been chosen so as to be able to focus on the general techniques rather than have to deal with the nitty gritty of specific cases.

4.2.1. A Spanish-Czech Glossary

In the following example, we are given two files to process. They both specify what different parts of a car are called: one is in Spanish, the other one in Czech. They look like this:

Spanish:

```
F  8 - Conmutador kick-down
F  9 - Conmutador para control del freno de mano
(01114)
F 10 - Conmutador contacto puerta trasera izquierda
(01708)
F 11 - Conmutador contacto puerta trasera derecha
(01709)
```

Czech:

```
F8 - spínač pohybu pedálu akcelerace
F9 - spínač kontrolky ruční brzdy
F11 - pravý dveřní spínač zadní
```

What we want is a single file in an easily importable format. One choice is to build a simple, flat text file with one record per line, with terms separated by tabulators:

```
Conmutador kick-down    spínač pohybu pedálu akcelerace
```

With a Perl script this will be a pretty easy task, but there are some things to bear in mind. First, the exact format of the identifier is not the same in the two files (in Czech there are no spaces between the letters and numbers). Second, the Spanish file includes some parenthesized numbers that we do not want in the target file. Also, not all Spanish terms have a Czech equivalent. The main task will be to build up a hash of hashes, with the normalized identifiers as keys and hashes of `filename => term` pairs as values. We use filename to indicate which language a given term corresponds to. Then we can iterate over this hash and print it out to the target file. A possible solution would be the following:

```
foreach my $file ( @ARGV ) {
  open( my $fh, "<", $file )
    or die "Can't open: $!";
  while ( my $record = <$fh> ) {
    next if $record =~ /^$/;
    my ( $key, $term ) = split( ' - ', $record );
    $key =~ s/\s//g;
    $term =~ s/(\(\d+?\))|(^\s+)|(\s+$)//g;
    $glossary{$key}{$file} = $term if $key ne '';
  }
  close( $fh );
}

open( my $out, ">", 'es-cz.txt')
  or die "Can't open: $!";
foreach my $key ( sort( keys %glossary ) ) {
  print $out join( "\t",
                   values %{ $glossary{$key}}
                 ), "\n";
}
```

The first `foreach` loop iterates over the special array `@ARGV` which contains the elements given to the script as command-line arguments. If the script is invoked like this:

```
> perl myscript.pl spanish.txt czech.txt
```

then the `@ARGV` will consist of two elements. So this loop will first open the `spanish.txt` file, read the records and store them in the `%glossary` hash and then repeat this for the `czech.txt` file.

The `next` command inside the `while` loop skips empty lines. The first substitution regex removes whitespace from the identifiers and the second one deletes the parenthesized digits and any leading and trailing whitespaces from the terms. Then we open the output file, iterate over the keys of the `%glossary` hash and print out the values of the internal hashes joining them with tabulators.

As can be seen, in Perl the whole process is pretty trivial. However, relying solely on the import capacities of existing CAT tools it would be virtually impossible to do anything useful with the kind of data we had as input. But once they are converted by this script, they can be imported in a totally straightforward way.

This case was so easy in part because the source file contained only very basic information: just terms assigned to unique identifiers. But in other cases you may be faced with a more complex source that includes multiple languages, synonyms, definitions and a variety of other terminological information. Representing this in a flat file with one record per line can be difficult and awkward. In these cases what can be done is to use one of the markup text formats employed for import and export by the terminology application that we are trying to import into.

In MultiTerm by Trados, records are separated by ** and inside the record a two-level tagging scheme is used: the-first level tags mark the beginning of a language section, and inside each language section tags are used to mark fields such as term, definition, synonym, abbreviation and cross-reference. This kind of format is simple and easy to produce.

TermStar by STAR adopts a different solution: it uses the XML-based Martif standard. XML document types usually have a formal specification using the DTD (Document Type Definition) syntax, so their validity can be automatically verified. The Martif standard is specified in the Martif DTD.

This is both more flexible and more complex to produce than the MultiTerm scheme. And, at least in principle, it could be used by an application other than TermStar since XML is a widely supported standard.

Although writing well-formed XML documents can be complex, it is simplified to a large degree by the use of a helper module, such as for example XML::Writer. Below is part of a program that uses this module and that can be used to write a simple Martif document. This document will be well-formed, but we will not validate it against the Martif DTD, since for our proposes this is unnecessary. What is important is that the application that is to use the output file accepts it as valid and interprets it correctly.

```
use XML::Writer;
use IO;
my $out = IO::File->new( “>mtf.xml” );
my $w = XML::Writer->new( OUTPUT => $out, DATA_MODE =>
1 );
sub startMartif {
    my $w = shift;
    $w->startTag( ‘martif’);
    $w->startTag( ‘text’);
    $w->startTag( ‘body’);
}
```

```
sub endMartif {
    my $w = shift;
    $w->endTag( ‘body’);
    $w->startTag( ‘back’);
    $w->endTag( ‘back’);
    $w->endTag( ‘text’);
    $w->endTag( ‘martif’ );
}

sub writeTerm {
    my ( $w, $term, $termType, $def ) = @_;
    $w->startTag( ‘ntig’ );
    $w->startTag( ‘termGrp’ );
    $w->dataElement( ‘term’, $term );
    $w->dataElement( ‘termNote’, $termType,
                     ‘type’ => ‘termType’ );
    $w->dataElement( ‘descrip’, $def,
                     ‘type’ => ‘definition’ )
        if defined( $def );
    $w->endTag( ‘termGrp’ );
    $w->endTag( ‘ntig’ );
}
```

At the beginning of the script we have created the object `$w` that will be used to write out the data. Then three subroutines are defined which make it less tiresome to wrap the data in the appropriate Martif tags. The important subroutine is `writeTerm()` which can be used to write out a term group consisting of a term, its type (full form, synonym, abbreviation, etc.), and an optional definition. These subroutines could be modified to create application specific elements such as hyperlinks, cross-references or data on the person who created and modified a record. But we will keep the example simple. So far the program above does not do anything; what needs to be added is the part that actually processes the input data and writes them out using the methods of XML::Writer and the subroutines we defined. Consider the following fragment of a glossary in HTML format:

```
<br><a NAME=“carroorientable”></a><b>carro
orientable</b>
<br>&nbsp;<i>combinación de una corredera</i>
<br>&nbsp;<i>portaherramientas y de un plato
giratorio</i>
<br>&nbsp;&nbsp;DEU  drehbarer Werkzeugschlitten
<br>&nbsp;&nbsp;ENG  swivel top slide
<br>&nbsp;&nbsp;FRA  coulisse supérieure pivotante
```

The processing problems it presents are fairly typical of HTML documents: they are made to be displayed to human readers rather than be easily processed by machines. The information is not clearly structured as it is in XML – here we will use Perl regular expressions to extract it. We will add the following code to our program.

```
$/ = ‘<br><a’;
startMartif( $w );
while( my $rec = <> ) {
  my ($spa, $def, $langs ) =
    $rec =~
      m{NAME.+?<b>(.+?)</b>.+?<i>(.+)</i>(.+)}s;
  $def =~ s/(<.+?>|&nbsp;)//g;
  my ($deu, $eng, $fra) =
    $langs =~
      m/[A-Z]{3}\s\s(.+?)$/gm;
  writeTermEntry( $w, $spa, $def,
                  { ‘deu-de’ => $deu,
                    ‘eng-gb’ => $eng,
                    ‘fra-fr’ => $fra }
                ) if defined( $spa );
}
endMartif( $w );
$w->end();

sub writeTermEntry {
  my ( $w, $spa, $def, $langs ) = @_;
  $w->startTag( ‘termEntry’, ‘id’ => $id++ );
  $w->startTag( ‘langSet’, ‘lang’ => ‘spa-es’ );
  writeTerm( $w, $spa, ‘full form’, $def );
  $w->endTag( ‘langSet’ );
  foreach my $lang ( keys %{$langs} ) {
    $w->startTag( ‘langSet’, ‘lang’ => $lang );
    writeTerm( $w,
               $langs->{$lang},
               ‘full form’ );
    $w->endTag( ‘langSet’ );
  }
  $w->endTag( ‘termEntry’ );
}
```

We read the file record by record (by setting input separator to `<br><a’`). The main `while` loop extracts terms in Spanish, German, English and French, plus the Spanish definition, and passes these data to the `writeTermEntry()` sub-

routine which writes them out to the output file. A fragment of the resulting Martif file looks like this:

```
<langSet lang="spa-es">
<ntig>
<termGrp>
<term>carro orientable</term>
<termNote type="termType">full form</termNote>
<descrip type="definition">combinación de una corredera
portaherramientas y de un plato giratorio</descrip>
</termGrp>
</ntig>
</langSet>
```

As can be seen, this format allows for easy structuring of information by means of nested elements delimited by start and end tags. The file output by the script should be directly importable into a Martif aware application (such as TermStar) without the need to define any additional import options.

If you use a CAT or terminology tool that uses an XML standard such as Martif, it might be worthwhile implementing a helper module that would specifically serve to output data in this format. Above only a few subroutines were defined to produce an acceptable well-formed Martif file, but for serious work a module with complete support of a given DTD would be more useful. However, this falls beyond the scope of the present paper.

4.3. Other Areas of Application

In this section, we will briefly discuss other areas where scripting can prove of use, without going into the details of implementation. They are more complex than the scripts presented so far and draw on the existing body of research and on specific techniques in corpus and computational linguistics (Brew and Moens 2000; Jurafsky and Martin 2000). There are also dedicated software applications that provide the functions described below. However, if you need them only occasionally or if you want highly customized solutions it might be preferable to write your own scripts.

4.3.1. Custom Statistics

Modern CAT suites provide all sorts of fairly sophisticated statistics useful for project management tasks such as monitoring progress or pricing. Even so, you may sometimes need something they are not capable of. In these cases you can write a script that does the job.

Translation Consistency: CAT is supposed to ensure that your translations are consistent: the same source language segments are translated in the same way each time they occur (unless context requires otherwise). This is due to the use of translation memory: you only translate a given segment the first time it occurs; on subsequent occurrence it will be retrieved from the memory and you can reuse it.

This does work with small projects where each target language is covered by an individual translator. However, when the project has to be distributed among various translators who all work into the same language simultaneously, inconsistencies may arise. You can try to detect them by writing a script that for each source segment tells you if different translations exist, what they are and in which place in the text they appear. It could also provide some context, e.g., the previous and the following segment. In this way you can tell if the variants are legitimate or not and correct them when appropriate.

Probably the easiest way to implement such a script would be to export your projects to the TMX (Translation Memory Exchange) format, which is an XML document type, and process these TMX files rather than work directly on the internal format of your CAT application.

Types and Tokens: A measure of the lexical variety of a text can be approximated by its type per token ratio. This is obtained by tokenizing the text (dividing it into words and optionally normalizing them to lower case) and then dividing the total number of word-tokens by the number of unique words (word-types). The first sentence in this paragraph contains 17 tokens, but only 15 types (‘a’ and “of” count just once for the type count). So the ratio is around 0.882.

This ratio on its own is only useful for comparing texts of the same length, since it tends to decrease as text length increases. The implementation of a script that provides type per token ratios for alphabetic languages would involve:

- Extracting the plain text from the document (e.g., in the case of HTML stripping the tags);
- Dividing the text into individual word-tokens using whitespace and punctuation as delimiters;
- Getting type and token counts and calculating the ratio.

The kind of statistics described above is probably more useful in translation studies than in technical translation management proper. However, some of the same techniques employed to obtain it can be used for rough terminology extraction, which can be of less restricted usefulness. More on this below.

4.3.2. Word Frequencies and Terminology Extraction

Word frequency information is often used to perform a rough, semiautomatic extraction of significant terms from a text. As a result we get a list of words that have a high probability of being specialized terminology. These could be given equivalents in the target language(s) and introduced into the project dictionary to improve terminological consistency.

A script which produces such a list of potential terminology could be implemented in a variety of ways, but the first two steps would be the same as in the one that calculates type per token ratio: extraction of plain text and tokenization. Additionally, each type would be assigned an integer indicating its frequency in the text. On the basis of this information, we could put on the terminology list all the words above a certain frequency, minus those on an exclusion list. Such a list could include known stop-words and/or words below certain length.

Alternatively, two cut-off points could be established in the rank order, excluding words below and above it. The words above would tend to be common function words, and those below to be irrelevant. The same procedure could be performed with bigrams (or trigrams) in addition to words in order to capture complex terms.

4.3.3. From Translation Memory to Glossary

Some translation projects involve documents that are highly fragmented lists of out-of-context strings. This is often the case in software localization or inventories of machine parts. In these cases it is often useful to be able to convert a TM (Translation Memory) into a glossary and vice versa. For example, in a software localization project it could be advisable to translate the user interface first. Then the resulting TM will consist of short strings that are unlikely to appear on their own in the user manual or the helpfiles. However, they will often occur embedded in larger segments. So you can reuse your resources more efficiently if you convert your TM to a glossary, so the translators can easily retrieve the strings using the terminology application that is part of the CAT suite. Depending on the CAT application in question this could be a non-trivial task (e.g., with Transit).

Such a conversion can be done with a Perl script. Again, processing the TMX file exported by the CAT application and writing the extracted data to a flat text file or a tagged terminology format would be the most straightforward way of implementing it.

5. CONCLUSIONS

In this paper we have looked at different ways in which small programs written in a scripting language can complement CAT solutions in translation project management. We have implemented some simple examples in Perl and discussed other possible uses of scripting. We have argued that the flexibility of a programming language is difficult to achieve with ready-made software solutions such as existing CAT applications. In the area of translation management there is much potential for improving efficiency and relieving humans of mechanical tasks.

Programming techniques for translation need not be invented from scratch: we can draw on existing code. For natural language processing there is all the experience accumulated in corpus and computational linguistics. For dealing with markup formats there is an even larger body of code developed in system administration, networking and other areas. As the field of translation is becoming more and more dependent on technology, translation managers will increasingly benefit from familiarity with computational techniques or from collaboration with IT professionals.

Acknowledgements

I would like to thank Anthony Pym, of the Universitat Rovira i Virgili, and Harm Smit for having read this paper and made many useful suggestions.

References

Brew, C. & Moens, M. 2000. *Data Intensive Lingusitics*. World Wide Web, http: //www.ltg.ed.ac.uk/~chrisbr/dilbook/dilbook.html.

Jurafsky, D. & Martin, J. H. 2000. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linuistics and Speech Recognition*. New York: Prentice Hall.

Schwartz, R. & Christiansen, T. 1998. *Perl Cookbook*. Cambridge, MA: O'Reilly and Associates.

Wall, L., Christiansen, T. & Schwartz, R. L. 2000. *Programming Perl*. Cambridge, MA: O'Reilly and Associates, third edition.