"=~?"

(or: How We Learned to Stop Worrying and Love the Binding Operator)

Jason Lee*

14 June 2001

Abstract

This paper describes the Legal Data Markup Software project for the Legal Information Institute at Cornell Law School, the goal of which was to produce a system that could markup an ASCII text copy of the United States Code provided by the House of Representatives with XML tags. Topics discussed include using Perl to mark up hard-to-parse text, teaching Perl on-the-fly, lessons learned from using Perl as a platform for iterative software engineering, and perhaps just a few instances of (almost) infinite recursion.

1 Introduction

For centuries, the measure of a barrister was the size of his bookshelf. The entire body of law was kept in nice, solid leatherbound books that generally sat on a shelf and looked suitably pretentious.

With the rise of the Internet, electronic documents have joined their ink-and-paper counterparts. However, the legal profession has been slow to adopt the electronic medium, relying on third-party sources to make selected legal information available online–often through proprietary interfaces and fee-based services. The Legal Information Institute at Cornell Law School seeks to provide an alternative by expanding the body of legal information publicly available online.

The goal of the Legal Data Markup Software (LDMS) student project at Cornell University was to create a tool for the Legal Information Institute that could markup the United States Code (USC) provided by the United States House of Representatives.

*The author is currently affiliated with Cornell University, and can be contacted via e-mail at <jcl53@cornell.edu>.

2 Background

In order to understand the LDMS project, a little bit of historical background is required. Specifically, the goals of Cornell's Legal Information Institute and the current distribution method of the U. S. Code were primary factors in determining the scope and orientation of the project.

2.1 The Legal Information Institute

The Legal Information Institute (LII) at Cornell Law School was established in 1992 by Peter W. Martin and Thomas R. Bruce. The Legal Information Institute is primarily concerned with research related to the use of digital information technology in the context of legal information, with the aim of making law generally accessible, not only to legal professionals, but to anyone who can access the Internet. Furthermore, although it seeks to do so in partnership with other key actors, autonomy is key: under no circumstances are activities of the LII to be under control or direction of these other actors.

The LII was established with a $250,000 USD multi-year startup grant from the National Center for Automated Information Research, and is currently supported by grants, gifts, and consulting projects undertaken by the co-founders[1].

2.2 The U. S. Code

The U. S. Code (USC) is published electronically by the Office of the Law Revision Counsel. This set of laws covers the general and permanent laws of the United States, excluding executive orders, judicial proceedings, treaties, and laws enacted by state or local governments[2].

The electronic publication of the USC is a relatively new development; the structure of the ASCII-encoded full text of the USC bears testament to its origins as a paper-only document. As discussed below (see Section 2.2.2, on page 3), the full text as provided contains very few machine-friendly structural cues, and is likely identical to the text provided to the publishers of the ink-and-paper version of the USC.

2.2.1 General Properties of the USC

The USC in ASCII-encoded text format[1] encompasses 50 titles, ranging in size from 600 bytes to 45 megabytes. The total collection size is approximately 565 megabytes[2].

[1] The USC as ASCII-encoded text is available at <`http://uscode.house.gov/download.htm`>. Unfortunately, there is currently no option to download the entire collection as a single archive; each of the 50 titles must be downloaded separately.

[2] When all was said and done, the LDMS system was able to markup all major divisions of the text in the space of approximately 2 hours.

```
-CITE-
     1 USC TITLE 1 - GENERAL PROVISIONS                              01/23/00

-EXPCITE-
    TITLE 1 - GENERAL PROVISIONS

-HEAD-
    TITLE 1 - GENERAL PROVISIONS

-MISC1-
      THIS TITLE WAS ENACTED BY ACT JULY 30, 1947, CH. 388, SEC. 1, 61
                                STAT. 633
    Chap.                                                             Sec.
    1.    Rules of construction                                         1
    2.    Acts and resolutions; formalities of enactment; repeals;
      sealing of instruments                                          101
    3.    Code of Laws of United States and Supplements; District of
      Columbia Code and Supplements                                   201
                           POSITIVE LAW; CITATION
      This title has been made positive law by section 1 of act July
    30, 1947, ch. 388, 61 Stat. 633, which provided in part that:
    ''Title 1 of the United States Code entitled 'General Provisions',
    is codified and enacted into positive law and may be cited as '1 U.
    S. C., Sec. - - .' ''
```

Figure 1: Original USC text.

2.2.2 Structure of the USC

The USC is ordered hierarchically, with titles at the top and statutes at the bottom. The number of levels (e.g., subsection, chapter, &c.) between title level and statute level is variable. Furthermore, the name of each division between title and section is also variable. Additionally, even the numbering scheme of divisions is variable; some divisions are identified by Roman numerals, others may be identified by letters (uppercase or lowercase), and still others may be identified by Arabic numerals[3]. Within each statute can exist several paragraphs of text, as well as tables and cross-references.

In the electronic version, several single-dash delineated tags (`-CITE-`, `-EXPCITE-`, &c.) are present, most likely as formatting guides for the publisher of the paper version. These field tags (often referred to as "dashtags" or "dashlines" by the developers) divide the document into several non-nested sections at the top-most level. These proved to be useful for dividing each title up into manageable chunks.

3 The Legal Data Markup Software Project

In the fall of 2000, a team of student software developers in the Software Engineering course at Cornell were assembled for the purpose of developing a software solution that could markup the USC as provided with XML tags.

[3]Even more confusing are those sections which are identified by combinations of these three.

```
-CITE-
    26 USC TITLE 26 - INTERNAL REVENUE CODE                          01/05/99

-EXPCITE-
    TITLE 26 - INTERNAL REVENUE CODE
    .

-HEAD-
    TITLE 26 - INTERNAL REVENUE CODE

-MISC1-
                  ACT AUG. 16, 1954, CH. 736, 68A STAT. 3
      The following tables have been prepared as aids in comparing
    provisions of the Internal Revenue Code of 1954 (redesignated the
    Internal Revenue Code of 1986 by Pub. L. 99-514, Sec. 2, Oct. 22,
    1986, 100 Stat. 2095) with provisions of the Internal Revenue Code
    of 1939. No inferences, implications, or presumptions of
    legislative construction or intent are to be drawn or made by
    reason of such tables.
```

Figure 2: Field tags in the United States Tax Code.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE LDMS SYSTEM "LDMS.dtd">
<!-- This file created by LDMS v1.0 -->
<!-- The dtd is ./LDMS.dtd -->
<LDMS>
<STRUCTDIV name="TITLE" vlevel="0" hlevel="0">
<TITLEDATA>
<NAVGROUP>
<CITE titlenumber="1">
     1 USC TITLE 1 - GENERAL PROVISIONS                    <DATE ACTUAL="2000-01-23">01/23/00</DATE>
</CITE>

<EXPCITE level="1">
<DIVEXPCITE>
 TITLE 1 - GENERAL PROVISIONS
</DIVEXPCITE>
</EXPCITE>

<HEAD>
    TITLE 1 - GENERAL PROVISIONS
</HEAD>

</NAVGROUP>
```

Figure 3: USC with XML tags.

3.1 Goals of the LDMS Project

Simply stated, the aim of the project was to markup the USC with XML tags. However, there were several specific requirements that had to be met for this project:

- Accuracy, accuracy, accuracy!
- Structure resolution had to extend to the paragraph level.
- Any unparseable text had to be marked as such for manual handling.
- Software was to be easily portable to multiple platforms.
- Software was to be easily maintainable and extendible.

TABLE OF CONTENTS

This Table of Contents is inserted for convenience of users and was not enacted as part of the Internal Revenue Code of 1986.

SUBTITLE A - INCOME TAXES

Figure 4: Title 26 table of contents, displaying a hierarchy of Title, Subtitle, Chapter. Statutes are always present at the lowest level.

3.1.1 Accuracy, Accuracy, Accuracy!

Due to the nature of the legal profession, accuracy in tagging was paramount. Lawsuits have been won and lost based on the precise location (and, in effect, role) of a piece of text, and even the position of a single comma can radically change the meaning of a statute[4].

3.1.2 Structure Resolution

The USC, as described above, can be seen as a hierarchical structure with titles at the top and statutes at the bottom. For legal applications, paragraphs, tables, and cross-references must also be recognized as such and tagged. All divisions between and including the title level and paragraph level were to be marked up and given unique identifiers. The DTD that defined how the USC was to be marked up needed to take this into account.

The challenge here was to develop a system that could be generally applied to all 50 titles. It would have been impractical to create one DTD per title; a good deal of interpretative flexibility was needed in both the DTD and the processing program. This flexibility, of course, was subject to the first mandate: accuracy.

3.1.3 Unparseable Text Handling

Unfortunately, the USC has had a history of exhibiting unusual errors. Sections may be accidentally misplaced or misnumbered, duplicate sections may appear, and section ordering can become unpredictable. In these cases, human intervention may be required to sort out the problems. To this end, the LDMS

[4]Those interested in the linguistic particularity of the legal profession may find Lawrence Solan's book, The Language of Judges[4], an intriguing read.

software was to identify sections that simply didn't fit and tag them as being questionable.

3.1.4 Software Portability

The target platform was unknown at the time the software was under development, as the LII was evaluating several different hardware/OS platforms to augment its current complement. Ideally, the software would run on as many platforms as possible out-of-the-box, so that anyone with a computer and free time could use it.

3.1.5 Software Maintainability/Extendibility

In the future, the LDMS software may be extended or modified to support different legal information collections. Furthermore, the structure of the USC as provided may change slightly. Therefore, it was important to allow new functionality to be easily added.

3.2 Personnel

All seven developers on the LDMS project were students at Cornell University. The project was overseen by Thomas Bruce, co-director of the LII, and William Arms, who taught the associated course.

3.2.1 Background Knowledge

In terms of what the developers knew before beginning work:

- 7 were proficient in Java.
- 7 knew what XML was.
- 6 knew how to write an XML document.
- 2 knew how to create an XML DTD.
- 3 had heard of CVS.
- 2 knew what CVS was.
- 1 had used CVS before.
- 7 had heard of Perl.
- 4 knew what Perl was.
- 3 actually knew Perl.
- 1 knew what the -T and -w switches did.
- 0 had extensive experience with Perl modules.

3.2.2 Team Structure

Although the team had no official leadership or management roles, one developer ended up being responsible for scheduling meetings, another took up the role of integrating the modules into a cohesive whole, and a third has since become responsible for writing an HTML backend for the software.

Each team member was given responsibility for a certain set of Perl modules, with the amount of regular expression work required proportional to the developer's familiarity with Perl.

3.3 Development Methods

3.3.1 Using the Iterative Model

The software was developed using an iterative[5] model–many successive iterations of testing and refinement–as opposed to a more traditional waterfall model, in which the final requirements of the system is determined at the outset and remains fairly stable.

Although the corpus that the system was to operate on was known in its entirety, the possibility of human error in generating the USC (formatting errors, spelling errors, and even the odd misplaced comma) made predicting the requirements for future versions of the USC practically impossible.

Additionally, the sheer size of the collection ensured that there would always be one more case to plan for.

3.3.2 Version Control

This project used CVS for version control. Although the modularization of the code meant that most modules were the sole responsibility of one developer, using CVS allowed the team to be more flexible in terms of focus. If needed (it was) developers could combine efforts on a module that required plenty of work to be done.

Additionally, if any last-minute fixes were required before a test cycle (they were), the developer was not required to check in his work. Test cycles were able to proceed almost independently of development.

3.3.3 Testing

Due to the size of the USC, direct inspection of the system's accuracy was impossible. Therefore, a random sampling[6] of 20 statutes per test cycle was

[5] Or, as some would call it, the "trial and error and trial and error and trial and..." model.

[6] Sampling locations were determined by a d1000 modulo the number of items on each level of hierarchy (e.g. d1000 % 50 at title level). Due to the inconstant hierarchical depth among titles, an iterative method was used; the topmost item–the title–was identified by (d1000 % 50), then the second-level item was identified by (d1000 % number of items in the next level in that title). This process was repeated until a statute (as numbered in the USC) or other item (if unnumbered, given provisional numbers starting after those assigned to statutes) was identified.

used; total accuracy was measured as a percentage of statutes marked up without error. Other measures included subdividing the accuracy by type (statute, cross-reference, table, &c.) and processing rate[7] (in kB/s).

4 Why Perl?

Several languages were originally proposed for use in development, namely Java, C, and Perl[8]. Perl was chosen primarily because it minimizes dependence on the underlying hardware, has built-in regular expression support, can be modularized, and does not require compilation. Here several of the team's expectations about the benefits of Perl are enumerated.

4.1 Platform-Neutral

It was expected that since Perl is not explicitly compiled, it is dependent primarily on the interpreter, rather than on the underlying architecture. This would (theoretically) allow the software to run just about anywhere a Perl interpreter could run, with minimal modification.

4.2 Regular Expressions

It was expected that the built-in regular expression support would make the task of locating particular patterns that marked division boundaries easier. Also, it was expected that regular expressions could be used to make fairly general pattern-matching templates that could be used to match entire classes of material.

4.3 Modular

Perl can be modularized without worrying about differences between the system the modules are built on and the production system the modules are intended for. This was expected to provide several benefits, including division of responsibility, interchangeable components, and extendibility.

4.3.1 Division of Responsibility

Never underestimate the power of blame. Each module was assigned to a certain developer. If an error occurred in that module, fixing it could be made the responsibility of the developer responsible for that module. Compare to the situation that would have arisen had the system been monolithic.

[7] Processing time was measured using the *NIX 'time' command.
[8] Someone suggested x86 assembly. He was re-educated.

4.3.2 Interchangeable Components

If a new, improved algorithm for distinguishing footnotes from the surrounding text is developed, it can be substituted for the current footnote module if it has the same functions available. This substitution is not inherently dependent on library versions or OS functionality, unlike many solutions that require compiled code.

4.3.3 Extendibility

The processing chain can be changed simply by adding more modules. There is no need to recompile anything at all; the new modules only need be imported and called as appropriate.

5 The Perl Crash Course

At first, it was expected that each developer would find the appropriate resources pertaining to Perl on his own initiative. Unfortunately, it turned out that developers had certain reservations about studying Perl on their own (see section 8.1 on page 16 for the most compelling reservations). Thus, some active intervention was required.

5.1 Course Materials

5.1.1 Books

The common text used in the crash course was the Perl in a Nutshell book[3]. Had time been less essential, a text with more tutorial matter would have been used; the goal here, however, was to have developers learn the parts of Perl most relevant to the project in the least amount of time.

5.1.2 Tools

An email list was the major mode of communication among team members. However, the use of a whiteboard in discussing Perl syntax proved invaluable for demonstrating regular expression syntax.

5.1.3 Miscellaneous

A key component of teaching Perl on-the-fly was the generation of sample code for developers to look at and play around with.

5.2 Goals

5.2.1 Dispel FUD

One major hurdle was a general fear of Perl. The sentiment expressed by those developers who had not written in Perl was that it is a difficult language to

program in, hard to maintain, and not a "professional strength" solution[9].

5.2.2 Develop a Minimum Skill Set

For the purposes of this project, only a narrow set of skills were required, specifically those dealing with pattern-matching. Other issues were generally ignored; this worked for the most part, but led to several problems down the line (see Section 7, on page 13).

5.2.3 Apply Existing Knowledge

None of the developers were new to the programming field; most had over five years of experience with computers. Relating parts of this past experience to Perl would help to speed the process up.

5.3 Methods

Being an ad-hoc process, the crash course was fairly disorganized. However, several practices revealed themselves to be important.

5.3.1 Emphasis on Similarities

Perl syntax was presented in terms of differences from Java, with the exception of regular expressions and the binding operator. This turned out to be particularly useful in discussing flow control and variables, as loop structure is almost identical, and variables could be directly related to primitive Java types.

In all cases, the most similar elements were taught first. Topics covering subjects more unique to Perl (the binding operator, for example) were discussed later, always in the context of the project's goals. The aim was, of course, to cover only as much Perl as necessary.

5.3.2 Demonstrations

Use of Perl code was demonstrated through sample code that illustrated part of the minimum skill set. Typically, developers were walked through the sample code line-by-line, with further diagramming done on the whiteboard. This served both to foster familiarity with the syntax, as well as to show that Perl really isn't inherently more difficult than any other language.

5.3.3 Encouragement of Further Reading

Developers were strongly encouraged to keep reading on the topics discussed. This was important not only as an aid to instruction, but as an opportunity for developers to amass a library of reference material that could be consulted over the course of the project.

[9] The regular expressions convinced them otherwise.

6 How Perl Helped

In section 8.1 several qualities were listed that constituted reasons for the project software to be written in Perl as opposed to other languages. This section provides illustrations of how these qualities played into the process of building software using an iterative process.

6.1 Modules

6.1.1 Use of Pre-existing Modules

One feature that was not originally considered was the availability of Perl modules that provided existing functionality. The XML::Writer module by David Megginson was quite useful for formatting and validating the output XML. The LDMS system, however, does not take advantage of the current location functions; this is handled by an internal state machine.

6.1.2 Damage Control

At any given point in development, certain modules were considered safe to use and apt to the task at hand, whereas others were in the process of being refined and/or debugged. This was necessary due to the trial-and-error nature of iterative software engineering–it was critical to respond immediately to any features that deviated from the requirements.

Periodic (some would say constant) testing of the project software is also one of the primary practices of the iterative model of software engineering. However, with (usually) a third of the code undergoing revision at a time, there were often times at which one module or another was simply broken and not likely to be fixed before the next testing cycle.

Using Perl modules made it relatively painless to write dummy modules to stand in for broken modules for testing without the need to recompile the code. The process became a simple matter of renaming a few modules in the test tree just before a test cycle.

6.2 Regular Expressions

Perl's regular expression support was quite well suited to the task at hand (namely drawing attention to certain patterns in the text), and saved the team an enormous amount of work, as expected. What follow are a number of examples in which regular expressions saved time and effort.

6.2.1 Indentation Levels

For example, one reasonably consistent method of determining division level at resolutions finer than the statute level is to check the amount of indentation preceding each line of text. One general expression that came in handy looked like:

```
''SEC. 201. WAIVER OF REQUIREMENT FOR PARCHMENT PRINTING.
  ''(a) Waiver. - The provisions of sections 106 and 107 of title
1, United States Code, are waived with respect to the printing (on
parchment or otherwise) of the enrollment of any of the following
measures of the first session of the One Hundred Fourth Congress
presented to the President after the enactment of this joint
resolution (Nov. 20, 1995):
    ''(1) A continuing resolution.
    ''(2) A debt limit extension measure.
    ''(3) A reconciliation bill.
  ''(b) Certification by Committee on House Oversight. - The
enrollment of a measure to which subsection (a) applies shall be in
such form as the Committee on House Oversight of the House of
Representatives certifies to be a true enrollment.
```

Figure 5: Indentation that is indicative of hierarchical ordering.

```
                          WRITS OF ERROR
  Section 23 of act June 25, 1948, ch. 646, 62 Stat. 990, provided
that: ''All Acts of Congress referring to writs of error shall be
construed as amended to the extent necessary to substitute appeal
for writ of error.''

    Table Showing Disposition of All Sections of Former Title 1
-------------------------------------------------------------------------
Title 1 Former          Revised Statutes        Title 1 New Sections
 Sections                Statutes at Large
-------------------------------------------------------------------------
1                       R.S., Sec. 1            1
2                       R.S., Sec. 2            2
3                       R.S., Sec. 3            3
4                       R.S., Sec. 4            4
5                       R.S., Sec. 5            5
6                       June 11, 1940, ch.      6
                         325, Sec. 1, 54
                         Stat. 305
```

Figure 6: A three-part table.

```
/^(\s*)\S/
```

The parenthesized expression matches any whitespace at the beginning of a line. The matched whitespace can be retrieved with the pattern match variable `$1` and checked against the indentation level of the previous line to verify whether or not the depth of the current line is different from that of the previous line.

6.2.2 Table Identification

Tables are a part of the USC that tends to break efforts to recognize them. They are (often) delineated by lines of dashes–but not always! Tables may consist of two parts (title and data) or three (title, headings, data).

The key here is to recognize a line that consists of at least 35 consecutive dashes flanked by whitespace (if any) as a table boundary. This requirement can be expressed as:

```
/^\s*-{35,}\s*$/x
```

Perl's recurrence syntax, then, allows a good amount of flexibility in the recognition code without matching every line that contains a few dashes; due to the changing nature of the collection, it cannot be assumed that a line won't, for example, consist of only a dash–which would match a more general rule.

6.3 Unwound Code

Although concise code is generally a good idea in a production system, it hampers the revisions that are necessary using the iterative method. Fortunately, since there is more than one way to do it in Perl, code could initially be written in an expanded format, then rewritten more concisely (in an equivalent form) when it was considered stable. The sometimes bewildering array of equivalent options offered by Perl turned out to be quite an asset here.

6.4 Efficiency

By December of 2001, the LDMS system was able to process the entire USC in about 2 hours. Furthermore, memory use was conservative enough that each title could be processed in memory without the need to explicitly stream intermediate results to and from disk. Swap space was generally only used on titles over 10 megabytes–approximately 16% of the titles in the collection.

7 Gotchas

7.1 Build-Breaking and CVS

For many compiled languages, the basic standard by which one judges whether or not one can merge code into the repository is whether or not the code compiles. However, since there is no explicit compilation step in the process of running Perl code, developers tended to turn code in as-is, often breaking their module.

The resolution was to specify that code should be checked by running:

```
perl -c <filename>
```

This minimized the number of broken builds when used. A better solution would have been to make CVS run this command in lieu of the developers.

7.2 The Case of the Missing Function

At one point in the development process, the function that displayed command help simply vanished.

Yes, it still existed in the code, but nothing else could access it. Fiddling with the module namespace didn't work. Neither did moving it (and its associated perldoc) to a different location.

```
# Create private global variables.
my ($line) =""; #This concatenates each line(belonging to the text
                #associated with the -SOURCE- catchline)after removing
                #the new line character(\n)from each line and the white
                #spaces at the beginning of any line

my (@divLines) = ();      #each element of @divLines contains
                          #the text in $line separated by the
                          # delimiter ;


. . .


sub tagDivSource
{
   foreach(@_)
   {
       next if(/SOURCE/);
       chomp($_);
       $_=~s/^\s+//;
       $line.=' '.$_;
   }#end for
```

Figure 7: Snippets of fractal recursion code (2000-11-29).

The mystery was solved when a missing `=cut` was placed at the end of the perldoc block preceding the function.

Essentially, the function had been accidentally commented out while documentation was being added. The `=cut` line from the next function toggled out perldoc just in time for the rest of the program to be parsed.

7.3 Fractal Recursion

During one build cycle, the system took an abnormally long amount of time to process the titles; instead of taking approximately four hours, which was normal for that stage, the process required about a day.

The person in charge of integrating the modules decided to monitor a test run by himself. After about ten hours, he exclaimed something unintelligible and started typing furiously. Apparently that build was not only sucking up time, it had exhausted all the memory available on the development system. Linux had killed his SSH session in response.

Ultimately, the cause was found to be the only case of almost-infinite recursion anyone on the team had ever seen. "It's like a fractal or something," said one developer who saw what the code was doing.

The cause? A simple matter of scope. A temporary variable was needed to accumulate the input at an intermediate stage of processing. However, the developer had declared the variable at the top level of the module instead of inside the function that needed it.

The result? Each time this module was called, if the new input required it to do any work the new input was added to the old input, then the module went

```
sub tagDivSource
{
   my ($line) =""; #This concatenates each line(belonging to the text
                   #associated with the -SOURCE- catchline)after removing
                   #the new line character(\n)from each line and the white
                   #spaces at the beginning of any line

   my (@divLines) = ();      #each element of @divLines contains
                             #the text in $line separated by the
                             # delimiter ;
   foreach(@_)
   {
       next if(/SOURCE/);
       chomp($_);
       $_=~s/^\s+//;
       $line.=' '.$_;
   }#end for
```

Figure 8: A quick fix for fractal recursion.

back and processed the whole thing. Since the extra processing was not needed all the time, it did not proceed indefinitely; it eventually reached a point at which no more processing could be done, and handed the results off. However, at each stage it added more tags. The XML:Writer module was staggering under the sheer weight.

The quick fix, of course, was to declare the variables inside the function.

7.4 XML::Writer vs. the Public Health and Welfare

Title 42 is entitled: "The Public Health and Welfare". Contained within its 45 megabyte bulk are diverse topics ranging from environmental standards to space-based biomedical research to disaster relief. It is a particularly complicated title.

Although most titles are processed at an average rate of 30 kB/s[10], Title 42 happened to clock in at only 7 B/s. The next largest title, Title 26 (the federal tax code, roughly half the size), completed at a rate of 27 kB/s. There was obviously something happening in Title 42 that wasn't happening elsewhere.

The bottleneck turned out to be in output via XML::Writer–though the cause remains unclear[11]. Since this only occurs on one title, it's certainly not a big problem within the context of this project. However, other bodies of law which weigh in at over 100 megabytes per document may require subdivision or a different means of XML output.

[10]Time elapsed was measured using the 'time' command. Measurements were performed on a 350 MHz i686 with 128 MB RAM and 512 MB total swap space, running Linux 2.2.14.

[11]Preliminary testing on the author's home machine, with 512 MB physical RAM, indicates that the increased use of swap space necessitated by the size of Title 42 only degrades performance by approximately 10%. Another pending line of inquiry addresses the possible use of XML::Writer's data mode as a more efficient method of output.

8 Lessons Learned

8.1 "=~?" is Pronounced as "Huh?"

The most common explanation heard for why developers did not seek out information about Perl on their own was that they had looked at some Perl scripts and concluded that since they'd never seen most of the operators before, it required a completely different way of thinking.

They were right.

However, they also concluded that Perl would be a tough language to learn. This assumption seems to be shared by their colleagues in the M. Eng. program (at least, those who have never used Perl before). By contrast, Java is easy because "it is object-oriented".

The element of the crash course that seemed to be most effective in dispelling this assumption was the demonstration in which one developer wrote up the main command loop for the program in the space of ten minutes. Since the code didn't include any regular expression matching, it looked much like the sort of code the other developers were more familiar with.

Perl can be a friendly language[12]. It's all in the way one approaches it.

8.2 Perl Requires More than a Minimal Skill Set

The fractal recursion incident (see Section 7.3, on page 14) revealed that merely knowing topics relevant to the project at hand is not enough even for a project that does not make use of the more esoteric Perl features. Peripheral concepts, such as variable scope, can cause problems if not clarified at the outset.

Although Perl does not require the developer to actually know a little bit of everything, it helps.

8.3 Perl Can Speed Up the Iterative Cycle

By late November, iterations were occurring twice per week–every three days (in some cases even more frequently) the system was tested extensively on the USC. A major reason why such a short iterative cycle was possible was that Perl's regular expressions are flexible enough to process multiple cases with a single expression. Thus, fewer lines needed to be modified when bugs were found.

The project would have been substantially more difficult to debug had each possible case required unique handling code, not to mention it would have been a nightmare to maintain in the long run.

[12] Certainly some wags will say it's picky about who its friends are.

9 Conclusions and Parting Shots

9.1 Final Impressions

Even with the minimal feature set used by the project team, Perl proved to be an excellent choice for the system. There certainly are things which may have been better done differently, as discussed above, but these are not strictly concerned with problems of the language itself. Backends to the LDMS system are slated to be written in Perl, as well–its facility with manipulating text is greatly appreciated.

9.2 The Future of the LDMS Software

Although the project has been wrapped up[13], the software is still the subject of active development. A backend is being created to convert the XML-tagged USC into HTML, and plans are in the works for a search engine that takes advantage of the XML tags to efficiently index the collection.

Eventually, the software may interoperate with the Leda project at the LII: a distributed legal electronic document archive that will be based on open source software[14].

9.3 The Future of the Developers

The seven developers have since parted ways for the most part. Five of them are on their way to careers with large software companies that (in all probability) will make them quite rich. The other two, having decided to work on a separate project in the meantime, are on their way to founding a small software company that (in all probability) will make them quite poor.

References

[1] Legal Information Institute. "About the LII". Legal Information Institute, 2000. Legal Information Institute. 2000.
<http://www.law.cornell.edu/lil.html>.

[2] Office of the Law Revision Counsel. "About the Office and the United States Code". Office of the Law Revision Counsel, 2001. Office of the Law Revision Counsel. 2001.
<http://uscode.house.gov/about.htm>.

[3] Siever, E., S. Spainhour, N. Patwardhan. Perl in a Nutshell: a Desktop Quick Reference. O'Reilly & Associates, 1999.

[13] The project garnered an A in the class, plus several glowing comments by the instructor.

[14] The Leda project is still very much under development. However, it is being targeted for Red Hat Linux systems–in particular, glibc-2.2 + kernel 2.4.

[4] Solan, L. The Language of Judges. Chicago: University of Chicago Press, 1993.