Legal Knowledge and Information Systems
G. Sileno et al. (Eds.)

© 2023 The Authors.
This article is published online with Open Access by IOS Press and distributed under the terms
of the Creative Commons Attribution Non-Commercial License 4.0 (CC BY-NC 4.0).
doi:10.3233/FAIA230969

Executing United States Bills into Law: A Working Application in the United States House

Sela MADOR-HAIM [a,1] and Ari HERSHOWITZ [b]
[a] *XCiteDB LLC*
[b] *Govable*

Abstract. We describe a system to execute U.S. federal bills into law, as part of a working application for the United States House. This system is based on a formal grammar we developed, which achieves greater than 94% accuracy in parsing amendatory phrases in bills; it is in production at the United States House of Representatives and is used to produce official 'Comparative Print' reports showing how a bill would amend current law. The grammar consists of two components: *CItation Modeling and Processing Language (*CIMPL), which captures citations to current law as they are found in bill amendments, and *AMendment Processing Language (*AMPL), which includes directive language to amend the text referred to by the CIMPL phrase. Here, we describe the analysis that led to development of the grammar, and provide an overview of how the grammar is applied to execute proposed amendments in the Comparative Print Suite of the U.S. House of Representatives. This application is available to all Members and staff at the internal site, compare.house.gov. Both the executable grammar and the full, point-in-time U.S. law dataset upon which they act, are publicly described here in technical detail for the first time.

Keywords. legislation, congress, Natural Language Processing, Amendment,

1. Introduction

The "Ramseyer Rule", dating to January 28, 1929, requires committees of the U.S. House of Representatives to prepare a report for any bill that passes out of committee, showing how that bill would affect the current law [1]. The report, called a "Ramseyer Report", has traditionally been produced in a labor-intensive process, involving a great deal of legislative research and manual execution of amendments. In the case of large bills, committee experts spend weeks applying thousands of amendments to hundreds of statutes. The process was partially automated more than a decade ago, using a pattern-matching tool based on regular expressions. While it was a great improvement over manual amendment execution, the pattern matching approach has several limitations: it does not model the semantics of the amendatory instruction; it may fail on simple variations of the amendatory grammar; and it requires the addition of new patterns when new amendatory phrases

[1]Corresponding Author: Sela Mador-Haim, selama@gmail.com

are encountered. Starting in the 115th Congress, the House Clerk's Office embarked on a project to build a software tool for producing reports showing how a bill would change current law.

The formal grammar we developed for this project shows significant improvement over the previous pattern-matching system. It has been successfully implemented in a software tool called the "Comparative Prints Suite" that was released to staff and Members of the U.S. House of Representatives beginning in October 2022 [2]. The tool is used to produce official reports showing how a bill would amend current law. This is a valuable asset to analysts and Members of Congress, because bills are written in a way that often lacks context of the law to be amended.

We started this work by analyzing amendatory phrases in U.S. Congressional bills over a 13-year period to develop a Categorial Grammar lexicon that could express the semantics of amendments in two inter-related executable languages, *CItation Modeling and Processing Language (*CIMPL) and *AMendment Processing Language (*AMPL). We also developed an amendment processing engine (runAMPL) to execute these languages, using a new point-in-time XML database (XCiteDB) to retrieve and store the target text. This paper focuses on the analysis of amendatory phrases and the structure of CIMPL and AMPL. The runAMPL processing engine and database design may be described in future publications.

2. Related work

2.1. Natural Language Processing of Amendments

NLP (Natural Language Processing) has been effectively applied to extract references [3] and to parse the hierarchical structure of legislative texts [4]. However, there has been relatively little practical work to process amendments. One early work describes APS [5], which uses ATN (Augmented Transition Network) to parse amendments in natural language and execute them. The result is relatively limited in its expressive power and types of amendments supported due to the limitations of the encoding. In OPAL [6] and Menslegis [7], amendments are extracted (and in case of OPAL, parsed), but not executed.

There are also two older systems, mostly undocumented, that were used by the U.S. Congress to parse and execute amendments: the mini-Ramseyer tool, and the AIP (Amendment Impact Program) tool. These tools have not been formally described, but based on our knowledge of them, both use pattern-matching based on regular expressions to parse amendments. Their accuracy has been highly limited, leading the U.S. House to seek improved methods [8].

2.2. Amendment Processing Languages

To our knowledge, no formal languages exist for processing legislative amendments. However, some languages describe document changes in other domains. For instance, Google's diff-match-patch [9] uses offsets and strings to describe changes, but it lacks features like tree structure handling, requires the original text for offsets, and breaks if text offsets are changed. Conversely, stylesheet or query languages such as XSLT [10]

and XQuery [11] can describe any transformation, but are harder to generate and read. Tools that compare structured-text documents [12,13,14] use *Edit Scripts* to capture changes between two documents. These scripts emphasize document structure with limited support for text changes within nodes.

3. Analysis

Our analysis covered amendatory language from U.S. federal bills between 2005 to 2018, excluding bills only at the "introduced" stage. We focused on later-stage bills due to their standardized drafting. Of the 28,014 bills analyzed, we identified amendatory phrases using keywords like "is amended." This resulted in 284,094 amendatory phrases. After normalizing each phrase by removing specifics like quoted text and numbers, we found 606 keywords with 10 or more occurrences. A subset of 130 words covered 97.3% of all phrases. We then categorized these keywords based on their role in amendments. The list below provides some categories:

Action verbs inserting, striking, adding, redesignating, deleting, transferring
Other verbs appears, relating, amending
Character period, semicolon, comma, colon, dash
Location paragraph, section, subsection, heading, item, sentence
Other nouns place, sequence, term, matter, order, time, word
Pronouns it, this, those, either, their
Prepositions before, after, through, in, except, up
Adjectives following, new, all, such, first
Adverbs respectively, further, accordingly, appropriately
Function words by, and, the, at, as, of, to, for

We examined the context of each word in the amendment phrase, by collecting 3-grams (three word phrases). We defined a lexicon based on this analysis that would capture nearly all of the desired variation in amendatory phrases. Below we describe the lexicon and elements of the grammar; the full grammar files are available upon request.

4. CIMPL and AMPL

Our approach is to translate each English-language amendment into an executable machine-readable language. Since no existing formalized language fit our needs, we designed two formal languages with similar syntax and semantics: *AMendment Processing Language* (*AMPL*) for text and document transformations, and *CItation Modeling and Processing Language* (*CIMPL*) for specifying the citation, or the part of the law that is amended. Every CIMPL expression is also an AMPL expression, making CIMPL a subset of AMPL. We separate the two grammars, in order to allow independent pre-processing and retrieval of citations as needed. As an example, consider the amendment:

> Section 983(f)(8) of title 18, United States Code, is amended— by striking "or" at the end of subparagraph (C);

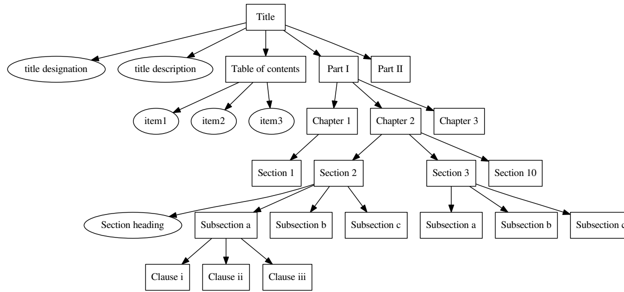

Figure 1. Structure tree of a legal document

The first part, before *"is amended"*, specifies what is amended, and it is translated to the CIMPL expression: `select law("united states code"): title("18"): section("983(f)(8)");` The second part, following *"by"*, states how the text is amended, and it is translated to the following AMPL expression:

```
strike is_end_of([subparagraph( "(C)")] ,search("or"));
```

As seen above, a CIMPL expression is usually straightforward. It starts with a `select`, followed by a location expression starting with `law()` function, specifying the amended law. Then it goes down the hierarchical structure using a colon operator. The AMPL code consists of one or more statements. In this example, the statement is `strike`, with the parameter `is_end_of([subparagraph( "(C)")]` specifying the text `‘‘or’’` at the end of subparagraph `(C)`.

A challenge in defining amendments is handling the law's hierarchical structure. The United States Code, for example, is typically organized into some combination of: Title, Subtitle, Division, Chapter, Subchapter, Part, Subpart, Section, Subsection, Paragraph, Subparagraph, Clause, Subclause, Item, Subitem. As shown in the diagram in Fig. 1, the document's structure is represented as a tree, where the subdivisions are the nodes, and each node can have multiple children ordered by label. Some special leaf nodes describe elements such as table of contents and headings that do not represent subdivisions. For example, in Fig. 1, the title designation and description are leaf nodes under Title, as well as the Title's table of contents (with table items as children).

Amendments work in both *structured text mode* and *flat text mode*. Structured text amendments work on the tree structure, deleting and inserting nodes, or replacing node's content. Flat text amendments view the document (or part of it) as a sequence of characters, ignoring structure, and select, strike and insert text using this flat-text view.

4.1. Context Selection and CIMPL

To model an amendment, we specify the document, and which part of the document is being amended. This is typically provided by the citation, translated into CIMPL. As stated earlier, CIMPL is a subset of AMPL, or more precisely, a CIMPL expression is a sequence of *select* statements in AMPL. For example, consider the amendment:

> Section 321 of the Energy Policy and Conservation Act (42 U.S.C. 6291) is amended— in paragraph (4), by striking ", determined in accordance with test procedures under section 323";

The text ``‘‘Section 321 of the Energy Policy and Conservation Act’’`` is the citation, translated into the CIMPL expression: select (law("Energy Policy and Conservation Act"):section("321"));. The language ``‘‘in paragraph (4)’’`` appears after the `is amended` phrase. This additional phrase is generally not considered as part of the citation, but further refines the scope of the amendment. We translate this phrase into an additional *select* statement:

```
select paragraph "(4)";|
```

Which is equivalent to adding the paragraph to the *CIMPL* expression, as in:

```
select (law("Energy Policy and Conservation Act"):section("321")):
    paragraph "(4)";
```

4.2. *Text Strike Statements*

In flat text mode, there are two types of basic actions: *strike* and *insert*. In AMPL, a strike is expressed with a *strike* statement consists of the keyword *strike* followed by an expression that specifies which text should be removed. For example, the amendment:

by striking “test procedures under section 323”;

would be translated into the AMPL statement:

```
strike search("test procedures under section 323");
```

In this example, we search for a string in the text, and strike it. However, a string may occur more than once in the text. For example, in `strike search("President")`, if the word “President” appears multiple times, striking all occurrences may not reflect drafter’s intent. Generally, when the drafter expects to strike multiple occurrences of the same text, they use language such as *by striking the word “President” each place it appears*. Without the phrase *each place it appears*, the word “President” should appear only once. We capture this distinction by using the directive `#all`. Semantically, the statements `strike search("President");` and `strike search("President")#all` are the same, except that in the first case (where the “all” directive is absent), it would perform an additional check and produce either a warning or an error.

In addition to simple *search* expressions, AMPL supports other, more complicated expressions to specify ranges of text we can strike. Due to space limitations, we do not list all the supported functions here, but a few examples include:

1. by striking “and” at the end
2. by striking “and” at the end of clause (iv)
3. by striking “the Director” the first place it appears
4. by striking “North Country Trail” and all that follows through “June 1975.”

The resulting AMPL statements are,

```
1. strike is_end(search("and"));
2. strike is_end_of(clause("iv"), search("and"));
3. strike nth(1,search("the Director"));
4. strike search("North Country Trail") through search("June 1975");
```

4.3. Text Insertion Statements

An amendment that inserts text inside a provision gets translated into an *insert* statement. In general, an *insert* statement has two parameters: an expression that specifies where to insert the text (position), and the text string that we want to insert. For example, when we have the amendment:

> by adding at the end the following: “and”

the AMPL code for the insertion is:

```
insert end "and";
```

In this case, the keyword *end* specifies inserting the text at the end of the selected provision. Instead of *end*, we can use other AMPL expressions to describe the location. For example *begin*, for inserting at the beginning of the provision. We can write `insert after(search("this")) "and that";`, for inserting “and that“ right after the word “this”, and similarly use `before(...)` for inserting text before a string. Furthermore, AMPL allows more complex position expressions. For example, `before(is_end(search(';')))` specifies a position right before the semicolon at the end of the provision. And the expression `after(nth(2,sentence()))` specifies the position following the second sentence.

4.4. Text Replacement

Text replacement is implemented in AMPL as a *strike* statement followed by an *insert*. For example:

> is amended by striking “hearing impairments” and inserting “deaf or hard of hearing,”.

This is translated into:

```
strike search("hearing impairments"); insert "deaf or hard of hearing";
```

Here, we have a *strike* followed by an *insert* statement that does not specify a position. An *insert* without a position inserts the text at the place of the previous statement (typically a strike, but can also be a previous insert). As discussed in Section 4.2, a strike range expression can match multiple occurrences of the same phrase. In this case, the subsequent *insert* statement inserts the new text in each of the matching positions for the strikes.

4.5. Structure-altering Statements

There are several actions we support on the structured document level. One of these actions is *rename*, which corresponds to changing the provision number (in bills this is commonly expressed as ‘redesignate’). *rename* l_1 *as* l_2 renames an element indicated by l_1 as l_2 by changing its label, but doesn’t move it in the tree. For example:

```
rename Clause("iii") as Clause("vi"); // Example 1
rename Clause({"i","ii","iii"}) as Clause({"vi","iii","v"}); // Example 2
```

In example 1, the label of Clause iii becomes Clause vi, but otherwise the tree remains unchanged. Example 2 shows multi-element rename, were Clause i becomes Clause vi, Clause ii becomes Clause iii and Clause iii becomes Clause v. Note that this would leave the nodes out of order, where Clause iii is after Clause vi.

We can also strike a structure element, which works just like striking a range. $strike\ Subsection(\text{"1"})$ is equivalent to $strike\ rangeof(Subsection(\text{"1"}))$. Similarly, *insert* can work on the structured document, either after striking of an element, or at a provided location. The *location* for insert is either $before(element)$ $after(element)$ or *end*. And finally, striking an element followed by an insert for a new element can be shortened as a *rewrite* statement. For example, $rewrite\ Subsection(\text{"1"})\ as\ < text >$ is equivalent to $strike\ Subsection(\text{"1"});\ insert\ < text >$;. We use the *rewrite* for brevity.

5. Parsing amendments to CIMPL and AMPL

The list of words identified in Section 3 allows us to use a compositional semantics [15] approach for parsing plain text amendments into a formal representation of the amendments grammar. Compositional semantics allows us to formally specify the semantics for each lexical item. We derive the semantics of the whole phrase as a composition of the individual lexical items. The formalism we use for specifying the compositional semantics for our lexicon is called Categorial Grammar [16]. In Categorial Grammar, words are categorized based on their grammatical roles. For each entry in the lexicon, we provide the text (which could be a word or a short phrase), the syntactic category of the item, and its semantics. A key attribute of Categorial Grammar is the correspondence between the syntax and semantics of each lexical element.

The syntactic categories we use correspond to the role of each lexical item in the amendment phrase, and also to the semantics of AMPL and CIMPL (see Section 4). Instead of broad grammatical categories such as nouns, we use finer categories such as `string`, `elements`, `ranges`, `positions`, `location` and `action`.

As an example, here are the entries for the word "before" in the lexicon:

```
"before" Positions/Ranges      \rns.before(rns);
"before" Loc/Elements          \elements.before_loc(elements);
"before" (Ranges\Ranges)/Ranges \rns.\rns1.is_before(rns,rns1);
```

As seen in this example, there are three entries for "before", each of them with different syntactic categories and semantics. The first has the category `Positions/Ranges`. The forward slash means that we expect a "ranges" phrase after the word "before", and the result is a `Positions` phrase. The expression `\rns.before(rns)` is a lambda expression, defining the semantics of this lexical item. The result is the function `before()` with the ranges phrase as argument. The next entry, with the category `Loc/Elements` is similar, when there is an elements phrase instead of a ranges phrase after the preposition.

In the third entry, the category `(Ranges\Ranges)/Ranges` means that it expects a "ranges" phrase after the preposition "before", and another "ranges" phrase before it. For example, if we have the amendment `striking \and" before \the plan"`, for the phrase `\and" before \the plan"`, the "and" is a "ranges" phrase, and so is "the plan". The result is `is_before(search("the plan"), search("and"))`.

6. Workflow and Implementation

We process bills in a custom, multi-stage Python-based, pipeline as follows:

1. Identify and extract amendments (XMLextract)
2. Parse English to CIMPL and AMPL
3. Retrieve target text from the current law database (XCiteDB)
4. Amend target text by executing the AMPL expression
5. Store back amended text

6.1. Extracting Amendments

As a starting point to process amendatory language in bills, we developed a tool called XMLextract. It uses pattern matching to identify amendments within a bill. Our analysis identified several trigger phrases that are used primarily in amendments, including: “is amended”, “is further amended”, and “are amended”. To extract the amendments, we look for such phrases. We benefit from the fact that the bills we process are in XML, and use the structural information to retrieve the full amendment for each phrase.

6.2. Parse to CIMPL and AMPL

After extracting the amendments in a bill, we parse each amendment, and generate the CIMPL and AMPL expressions for each amendment. The citation is the part that comes before the trigger phrase, and the amendment action is the part that comes after it, as described in Section 4. The two grammars are handled separately to allow citations to be retrieved and amended independently by different amendment phrases.

6.3. Retrieve Target Text

After obtaining the CIMPL and AMPL expressions for the amendment, we retrieve the part of the law to be amended. This presents a significant challenge. As described in the House Clerk’s report [2], “On the federal level, there is no single unified code like some of our state legislatures have or other national parliaments. When amending current law, federal legislation is drafted to several sources including the positive Titles of the U.S. Code, the Statutes at Large, and named Acts from the HOLC Statutes Compilations dataset. At this time, the ‘Changes in Existing Law’ application will illustrate changes to the U.S. Code, the current Statute Compilations, and some Statutes at Large.”

In the above example, we need to retrieve Section 983(f)(8) of Title 18 of the U.S. Code. An XML database can store all the laws in XML format, and would allow us to retrieve the specific part of the law that we want to amend (e.g. only Section 983(f)(8), and not the entire Title 18) at a particular point in time. For data retrieval, we use XCiteDB [17], an XML database that supports temporal versioning. This allows us to retrieve the structured text of the target law according to the bill’s date.

6.4. Execute the AMPL expression

To execute amendments, we use another tool we developed, called runAMPL. This tool receives an AMPL expression and the XML fragment for the target provision, and amends

Table 1. Results for 117th congress

	Number	Percent
Total	194151	100%
AMPL Parse Error	6267	3.23%
CIMPL Parse Error	4403	2.27%
Citation not found	23976	12.35%
Amendment failed	21905	11.28%
Parsed successfully	183481	94.5%
Completed	134254	69.15%

it by executing the provided AMPL code. Where execution of an amendment fails, it often indicates a drafting error, such as a typo in the amendment, or a conflict with another amendment. In such cases, the system issues a notification to the user, who can then correct the amendment.

6.5. Store Back Amended Text

Finally, after getting the amended provision from runAMPL, we store it back into the law database. XCiteDB supports branches, which allows us to store the amended text in a branch named after the amended bill (for example, `115hr1067ih`), so that we could retrieve the law as amended by each bill, without overwriting the original law on the main branch. This system, in theory, would allow us to automatically execute amendments as part of the consolidation or codification process. In practice, codification is done by the Law Revision Counsel and involves additional editorial work that requires human judgment, so this automated system is not currently used in the codificaiton process.

7. Evaluation

Table 1 shows the results when running the bills from the 117th congress. As seen in the table, only 5.5 percent fail due to amendment language which is not supported in the lexicon, and doesn't parse to AMPL and CIMPL, and 94.5 percent are parsed successfully. Out of the successfully parsed amendments, about 12 percent fail because the citation is not found, mostly because the database is incomplete and missing some laws. Similarly, 11.28 percent fail because the retrieved text is different than expected by the amendment (e.g. searched string not found). This can happen either because of data issues or drafting error (e.g. typo). The system identifies any potential errors and reports them to the users to analyze and correct in future versions of the bill, as appropriate.

8. Conclusion and Future work

This paper presents a system that employs a categorial grammar lexicon and two specially designed formal languages, AMPL and CIMPL, for processing amendments. This technology is implemented as part of a suite of software tools developed for the House of Representatives, which extracts and automatically executes amendments within bills. It also generates comprehensive reports showing the potential impact of these bills on

existing legislation. This system achieves greater than 94 percent success rate in parsing into executable grammar in U.S. bills. Phrases that cannot be automatically processed by the system often indicate errors in drafting or conflicting amendments, improving the overall efficiency and accuracy of the legislative process. Future work may extend this grammar to increase coverage, and support other jurisdictions and languages. We will also address current weaknesses in the current laws database, and provide more detailed automated feedback related to drafting errors.

8.0.1. Acknowledgements

We are grateful to Kirsten Gullickson and her team in the House Office of the Clerk, and E. Wade Ballou, Jr. House Legislative Counsel and his team for their vision and leadership in this work, and their generosity in sharing their deep expertise. We also thank colleagues from Xcential Corporation as well as other public sector team members for their feedback, testing, development and analysis.

References

[1] Deschler's Precedents;. (accessed April 19, 2023). https://www.govinfo.gov/content/pkg/GPO-HPREC-DESCHLERS-V4/html/GPO-HPREC-DESCHLERS-V4-3-7-3.htm#:~:text=Sec.%2060.%20Comparative%20Prints;%20The%20Ramseyer%20Rule.

[2] Office USHC. Comparative Print Project: Quarterly report to the Committee on House Administration; 2022. https://usgpo.github.io/innovation/resources/reports/Clerk-QR10-Comparative-Print-Project.pdf.

[3] De Maat E, Winkels R, Van Engers T. Automated Detection of Reference. In: Legal Knowledge and Information Systems: JURIX 2006: the Nineteenth Annual Conference. vol. 152. IOS Press; 2006. p. 41.

[4] de Maat E. Making Sense of Legal Texts; 2012. Ph.D. thesis, University of Amsterdam.

[5] Arnold-Moore T. Automatically processing amendments to legislation. In: Proceedings of the 5th international conference on Artificial intelligence and law; 1995. p. 297-306.

[6] Van Gog R, Van Engers TM. Modeling legislation using natural language processing. In: 2001 IEEE International Conference on Systems, Man and Cybernetics. e-Systems and e-Man for Cybernetics in Cyberspace (Cat. No. 01CH37236). vol. 1. IEEE; 2001. p. 561-6.

[7] Boella G, Di Caro L, Leone V. Semi-automatic knowledge population in a legal document management system. Artificial intelligence and Law. 2019;27:227-51.

[8] Statement of E. Wade Ballou, Jr., Legislative Counsel Office of the Legislative Counsel U.S. House of Representatives; 2021. https://www.congress.gov/116/meeting/house/110533/witnesses/HHRG-116-AP24-Wstate-BallouE-20200303.pdf.

[9] Diff-Match-Patch; 2021. (accessed May 2, 2023). https://github.com/google/diff-match-patch.

[10] Clark J, et al. Xsl transformations (xslt). World Wide Web Consortium (W3C) URL http://www w3 org/TR/xslt. 1999;103.

[11] Walmsley P. XQuery. O'Reilly Media, Inc.; 2007.

[12] DeltaXML;. (accessed May 2, 2023). https://www.deltaxml.com.

[13] Wang Y, DeWitt DJ, Cai JY. X-Diff: An effective change detection algorithm for XML documents. In: Proceedings 19th international conference on data engineering (Cat. No. 03CH37405). IEEE; 2003. p. 519-30.

[14] Gutiérrez-Soto C, Barra A, Landaeta A, Urrutia A. Change Detection by Level (CDL): An efficient algorithm to detect change on XML documents. In: 5th International Conference on Computer Sciences and Convergence Information Technology. IEEE; 2010. p. 1-7.

[15] Jacobson PI. Compositional semantics: An introduction to the syntax/semantics interface. Oxford Textbooks in Linguistic; 2014.

[16] Steedman M. Categorial grammar. Lingua. 1993;90(3):221-58.

[17] XCiteDB. XCiteDB A Time Machine for Structured Data; 2023. Https://xcitedb-web.vercel.app/home.