Syntax Analysis in the Climacs Text Editor

Christophe Rhodes
Department of Computing
Goldsmiths College
University of London
New Cross Gate
London SE14 6NW, UK
c.rhodes@gold.ac.uk

Robert Strandh
LaBRI
Université Bordeaux 1
351, cours de la Libération
33405 Talence Cedex
FRANCE
strandh@labri.fr

Brian Mastenbrook
Motorola
1303 E. Algonquin Road
Schaumburg, IL, USA 60196
brian@mastenbrook.net

20th June 2005

Abstract

The Climacs text editor is a CLIM implementation of a text editor in the Emacs tradition. Climacs was designed to allow for incremental parsing of the buffer contents, so that a sophisticated analysis of the contents can be performed without impacting performance. We describe two different syntax modules: a module for a sequentially-defined syntax of a typical programming language, doing the bulk of its parsing in a per-window function; and an interactive editor for a textual representation of lute tablature, recursively using a per-buffer function for its parsing.

1 Introduction

The field of advanced text editors is a crowded one, with a long history and an apparent ability to cause passionate argument. Climacs is philosophically part of the Emacs editor tradition, which has spawned many variants with many different approaches to buffer management, incremental redisplay, and syntax analysis. Emacs itself traces its lineage to TECO, where Emacs was originally implemented as a set of TECO macros. More information about text editing in general, and particulars of some editors we shall not discuss further, can be found in [7, 8, 12, 18] and references therein.

Climacs' syntax analysis is a flexible protocol which can be implemented with a full language lexer and parser. GNU Emacs, the most commonly used Emacs-like editor, uses regular expressions for its syntax analysis. However, regular expressions cannot be used to parse the general case of non-regular constructs such as Common Lisp's nestable `#|  |#` block comments. The lazy application of those regular expressions will also lead to additional erroneous parses even when nesting is not taken into account when the parser starts after the opening `#|` then the closing `|#` will be treated as the start of an escaped symbol name. Even if the regular expression parses the whole block comment correctly, other expressions can still match on the contents of the comment, leading to issues when the first character in a column in the block comment is the start of a definition. Emacs users quickly learn to insert a space before the open parenthesis to work around Emacs' font-lock deficiencies.

The Climacs text editor is a combination of frameworks for buffer representation and parsing, loosely coupled with a display engine based on the Common Lisp Interface Manager (CLIM) [11]. It includes the Flexichain library [14], which provides an editable sequence representation and mark (cursor) management, and an implementation of the Earley parsing algorithm [5], to assist in the creation of syntax-aware editing modes. An application can combine a particular implementation of the buffer protocol, the syntax protocol, and its own display methods to produce a sophisticated editor for a particular language.

The rest of this paper is organised as follows: we discuss the Climacs buffer protocol, which provides a standard interface to common text editor buffer operations, in section 2. The syntax protocol, which we discuss in section 3, provides a mechanism for attaching a lexical analyser and parser to the text editor, and provides for defining methods to draw syntax objects in the Climacs window. In section 4 we present some details of the implementation of syntactic analysis of editor buffers for various programming languages, including Common Lisp; in section 4.2, we discuss an application with Climacs at its core to support editing a textual representation of lute tablature. We discuss avenues for further development in section 5.

2 Buffer Protocol

The Climacs buffer protocol abstracts the operations performed on an editor buffer – or any editable sequence of arbitrary objects – from the implementation of those operations on a given data structure. This protocol is a set of generic functions for modifying and reading the contents of a buffer, and setting and retrieving marks in the buffer. The protocol abstraction is independent of any particular implementation, allowing flexible representations of buffers.

Currently Climacs uses a single `cursorchain` from the Flexichain library as the editable sequence representation for standard buffers. A `cursorchain` is a circular gap buffer with an arbitrary number of cursors in the buffer. Climacs uses these cursors as the implementation of marks in the buffer. The single gap-buffer implementation is used by many other editors, including GNU Emacs. Flexichain improves on this by making the gap buffer circular in its underlying representation; the start of the sequence is stored in a separate slot, along with the beginning of the gap.

Climacs also provides three purely functional (or fully persistent) buffer implementations, all based on functional data structures [1, 17]. The underlying data structure is a balanced binary tree with an abstracted-away rebalancing scheme, supporting sequence operations needed by the Climacs buffer protocol at reasonable $O(\log\ n)$ efficiency. The first implementation, `binseq-buffer`, uses one tree whose leaf nodes (buffer elements) can be arbitrary objects. An optimised implementation, `obinseq-buffer`, uses less space but buffer elements must be non-`nil` atoms. Finally, `binseq2-buffer` combines the previous two implementations, by using a tree whose leaf nodes contain the optimised trees representing lines; the benefit of this implementation are faster ($O(\log\ n)$, compared with $O(n)$) operations dealing with lines and columns. All three implementations enable simple and inexpensive undo/redo operations because older buffer versions are kept as a whole in memory, so there is no need to store editing operations to facilitate undo. Besides the undo operation simplification, the persistent buffer implementations facilitate further purely functional operations on Climacs buffers. The space cost of these implementations is not negligible, though it is alleviated by sharing significant portions of older buffer versions with newer versions.

It is anticipated that Climacs will provide several other buffer implementations, one of which will use a sequence of lines organised into a tree for quick access. In this structure, a line can be considered opened or closed. When a line is opened, it is represented as a Flexichain `cursorchain`. All editing operations are performed on open lines, and a fixed number of lines are kept opened according to a least-recently-used scheme. When a line is closed it is converted to a vector for efficient storage and access. If the line contains only `base-char` objects this vector is a `base-string`; otherwise, it is unspecialised.

This structure has the advantage of efficient line-based access in the buffer, in contrast to a single gap buffer implementation, where determining the line number of a mark or placing a mark at a specific line is $O(n)$, as the entire buffer must be scanned to determine how many newlines precede the mark. The proposed structure also provides much better behaviour than that of a single gap buffer when a user is editing two disparate sections

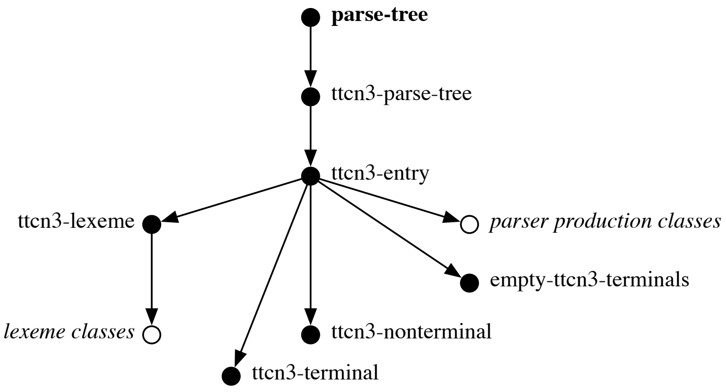


Figure 1: Organisation of classes used by a typical syntax

in a large file. With a single gap buffer, the gap must be moved to the point of edit before an edit operation is allowed; when the buffer is large and edit operations occur frequently at multiple locations in the buffer, this requires a substantial amount of copying between edits. In this situation single-gap-buffer editors such as GNU Emacs will noticeably pause between edits to move the gap. A structure which contains a sequence of lines and keeps the most recently used lines open as gap buffers can operate as a multi-gap buffer with automatic gap placement, while not suffering poor performance when accessing a specific line in the buffer.

The efficiency of Climacs buffers depends of course on the implementation of the buffer protocol that is used. Space efficiency will also depend on the implementation of Common Lisp which is used to run Climacs. In Steel Bank Common Lisp, the type `character` is represented with the full 21 bits used by the Unicode character space. External character encodings are converted to UTF-32 when a file is read in to memory. If the characters are stored in a specialised array, this will net a worst case space efficiency of four bytes of every byte in the file. However, the time advantages of this representation outweigh the space inefficiency for our purposes. Searching for an individual character in a sequence of n characters encoded in UTF-8 (or other variable-length encoding) is $O(n)$, because each individual character must be examined to determine the number of octets which are stored to represent that character.

The Flexichain library was designed to be able to take advantage of specialised lisp vectors for compact storage, though this possibility is not used by the current Climacs buffer implementation. Instead Climacs uses an unspecialised vector for its storage, which uses one machine word per element, either as an immediate value or as a pointer to a larger element. More space-efficient buffer implementations are possible, should it be necessary; for instance, it is conceivable that a buffer implementation might choose to compress sections of the buffer which are not in use.

3 Syntax Protocol

Climacs is designed to allow multiple implementation strategies for buffer parsers and syntax-aware display mechanisms. The set of hooks that Climacs provides to allow this is the syntax protocol. A syntax in Climacs is a class and set of methods which provide a lexical analyser, parser, and display methods for a buffer. The incremental parser associated with a syntax creates and updates a parse tree of a buffer's contents, and provides a mechanism for drawing these parsed objects in a window. The parser accepts lexemes produced by an incremental lexical analyser. Display is handled by drawing methods implemented using the CLIM high-level display toolkit.

Though an implementation of a syntax is free to choose its own parser implementation strategy, lexical analysis and parsing of the buffer is typically done in an object-oriented fashion. The lexer operates on objects in the buffer, usually characters, and returns objects of various classes. Each parser production is represented by a class. In complex syntaxes, the parser rules can be quite complicated and involve arbitrary code, but for a simple grammar the parsing rules can be entirely represented by matching on the classes returned by the tokeniser and parser. Figure 1 shows the organisation of classes in the TTCN-3 grammar.

The syntax analysis is divided between a per-window and a per-buffer function. Performing the analysis per-window is best suited to analysis of text where the parse tree will be used only for display and editing, as it is less important if the parse tree for off-screen text is up-to-date at every point during an edit. The per-buffer approach is appropriate when the parse tree will also be used for some other display or analysis of the text in the buffer, though it is also used for invalidating a region of the previous parse based on the extent of the region damaged by an edit.

Climacs includes a parser generator that implements the Earley [5] algorithm. There are many advantages of this algorithm in the context of text editing. Perhaps most importantly, no grammar preprocessing is required, so it is not necessary for the entire grammar to be known ahead of time. This means that the user can load Lisp files containing additional syntax rules to complete the existing ones without having to apply any costly grammar analysis. Other advantages include the possibility of handling ambiguous grammars, since the Earley parsing algorithm accepts the full class of context-free grammars. This feature is crucial in certain applications, for instance in a grammar checker for natural languages. Implementations of the Climacs syntax protocol may, but are not required to, use the provided Earley parser: any algorithm with an explicit representation of the parser state is suitable for use by an incremental parse system like that of Climacs' syntax protocol.

It should be noted that the Earley parsing algorithm is relatively slow compared to table-based algorithms such as the LR shift/reduce algorithm. Worst-case complexity is $O(n^3)$ where n is the size of the input. It drops to $O(n^2)$ for unambiguous grammars and to $O(n)$ for a large class of grammars suitable for parsing programming languages. Additionally, the complexity is often proportional to the size of the grammar (which is considered a constant by Earley), which can be problematic in a text editor. We have yet to determine whether the implementation of the Earley algorithm that we provide will turn out to be sufficiently fast for most Climacs syntax modules. Other possibilities include the Tomita parsing algorithm [15] which provides more generality than LR, but which is nearly as fast in most cases.

4 Syntaxes

We describe examples illustrating two different approaches to syntax analysis in the Climacs editor. Per-window parsing is used by the provided modes for HTML, Common Lisp, Prolog, and a Testing Control Notation (TTCN-3). Each of these syntaxes is implemented with the provided Earley parser [5]. The lute tablature editor uses a per-buffer function for its syntax analysis and implements a simple state-machine parser for its regular notation.

4.1 Per-Window Syntaxes

Climacs currently provides four syntaxes using a per-window parsing function. Of these the Prolog syntax is the most complete and implements the entire ISO Prolog syntax. The HTML, Common Lisp, and TTCN-3 syntaxes are somewhat less complete in their implementation. Each syntax is free to implement its lexical analyser and parser in the manner which is most convenient for its grammar. All of these syntaxes use the provided implementation of the Earley parsing algorithm, but each provides its own set of macros for defining parser rules to accommodate the level of analysis to parse the language's grammar.

Implementing the Prolog syntax proved a good test of the established framework. Firstly, and most importantly, ISO Prolog [10] is not a context-free grammar; *terms* have an implicit priority affecting their parse.[1] The implementation of Earley's algorithm, however, was able to address this additional complexity with no difficulty.

Another area of difficulty is the fact that parsing a Prolog text can change the grammar itself through the use of the `op/3` directive. The inclusion of

```
:- op(100,xfy,<>).
```

in a Prolog text means that, after parsing this directive, the token `<>` must be recognised as an right-associative operator with priority 100 in the grammar. This is achievable by keeping a cache of parsed `op/3` directives, and maintaining and invalidating it in parallel with the parse of the buffer itself.

While the Prolog syntax included with Climacs is a demonstration of the potential of strongly syntax-aware editors, it is not yet a practical tool for Prolog programmers, as most implementations of Prolog deviate from strict ISO compliance. On the level of surface syntax, interpretation of quoting rules and escape sequences in strings are very variable, while additionally treatment of operators in currently-available Prologs can differ markedly from the standard requirements; this means that working code written for these Prologs can be flagged as a parse error by Climacs. Nevertheless, work is underway to use Climacs' syntax analysis to provide a front-end for a Prolog development environment.

At present, one parse error implies the invalidation of the rest of the file. This adds a burden on the mode implementor that the syntax analyser be both bug-free and correspond with reality; a slightly-buggy or incomplete syntax mode can severely impair the utility of the editor. We plan to implement a resynchronisation method for parsers, which would allow the parse to continue at the next valid parsable state in the buffer; see section 5.

The Testing and Test Control Notation 3 (TTCN-3) language [6] is a language which captures detailed test specifications. TTCN provides both a textual "core" grammar and a graphical presentation format similar to Message Sequence Charts (MSCs) [9]. Climacs currently provides an editor for a subset of the TTCN-3 core language.

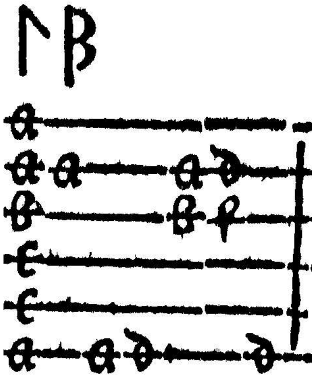
The TTCN-3 syntax is implemented with a high-level macro which defines classes and adds syntax rules using the syntax protocol for each terminal and non-terminal in the grammar. The syntax of this macro resembles the BNF form in which the official core grammar is specified, with the difference that optional productions and repeated productions in the form of "any number of" or "one or more of" receive their own non-terminal entry in the grammar. In addition, this macro defines basic display functions for the syntax objects produced by the parser, with language keywords appearing in a separate colour.

Much like the other per-window syntax modules, the one for Common Lisp uses high-level macros, some of which come with the Earley parser framework in order to build a highly modular grammar for parsing all of the Common Lisp language. This syntax module was built by a group of undergraduate students at the University of Bordeaux, and is currently being tested and improved.

4.2 Per-Buffer Syntax: a tablature editor

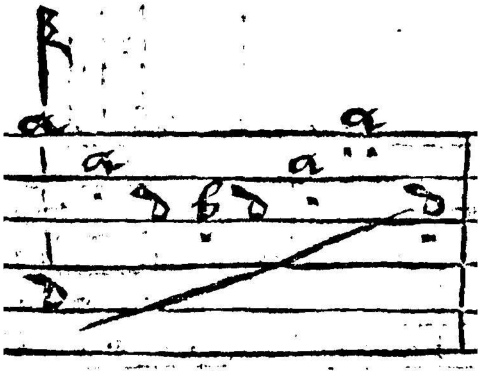
TabCode [2] is a textual format for description of lute tablature, a form of musical notation. In its simplest form, it is a sequence of whitespace-delimited independent words, where each word represents either a set of fret–string coordinates for the player's left hand specifying the note or chord to be played or alternatively some other element of musical notation (such as a barline); figure 2 shows a fragment of tablature, and demonstrates its *TabCode* encoding. It is also possible to encode more complex elements of lute tablature notation in *TabCode*. Ornaments, fingering marks, beaming, connecting lines and other complex elements can all be accommodated

[1] Formally, the grammar could be made context-free by introducing a large number of new production rules.



```
Qa1a2b3c4c5a6
Ea2
a6
d6
a2b3
d2f3
d6
|
```

Figure 2: An extract from ‘Fantasia Ioannis Dooland Angli *Lachrimae*’, Jean-Baptiste Besard, *Thesaurus Harmonicus* (1603), f.16v, and its *TabCode* encoding.



```
Sa1(E)d5(C5:8)
a2.
d3
b3.
d3
a2.
a1:
d3.(C-5:4)
|
```

Figure 3: An extract from ‘Lachrime by I. D.’ from *A New Booke of Tabliture*, published by William Barley (London, 1596), E1r, and its *TabCode* encoding. The parenthesised characters encode the lines joining and spanning the example, while the individual punctuation characters refer to the fingering marks.

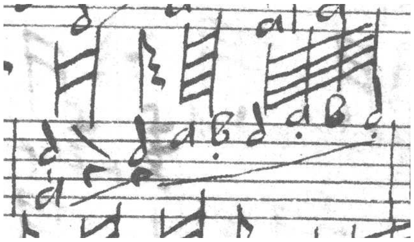


Figure 4: Extract from ‘An Almand by mr Jo Dowland Bacheler of musique’, *The Board Lute Book* (GB:Lam MS 603), f.13. Note in particular the connecting lines in this bar, joining chords within beams to an unbeamed chord.

(see figure 3 for examples of some of these more complex elements). *TabCode* has been used to produce scholarly editions of lute works [3] and to assist in computer-based musicological studies (as in [4] for example).

The *TabCode* language itself has developed to provide a terse and intuitive encoding of tablature, rather than a well-formed grammar for parsing. Simple *TabCode*, as in figure 2, presents no problems. In such a simple case, each chord is an optional rhythm sign (a capital letter), followed by zero or more notes as fret–string pairs (letter–number combinations). Adding ornaments and fingering marks to this structure is simple, as they are merely optional modifiers to each note, and can be parsed as such.

More complex to model are beams[2] and connecting lines, which have their own semi-independent identity, despite being encoded in *TabCode* as modifiers to individual tokens. In particular, the existence of beams and connecting lines means that we cannot parse a buffer into a sequence of tabwords and thence into hypothetical higher-level structures such as `beamed-group` and `connected-pair`, because these higher-level structures can overlap in non-trivial ways, as in figure 4. Instead, we deal with these modifiers by invoking a parser on the sequence of parsed buffer elements to generate parallel sequences of beams, connecting lines and other such tablature elements.

The tablature editor that has been developed in parallel with Climacs is in use for a project to catalogue European lute music, and additionally supports research [16] into more advanced notations and extensions of *TabCode* (to provide a means to encode editorial comments or alterations, or markup of distinguishing features of a specific source manuscript).

When the tablature editor is in use, the user has in addition to the usual editor window a scrollable graphical display of the tablature encoded within the buffer. Individual tablature elements are presented to this CLIM pane, allowing the graphical display to provide mouse-activated commands, such as one which moves the buffer cursor to the beginning of the tabword encoding a particular element.

Since we are presenting an alternative view of the whole buffer’s contents, we analyse the entire buffer’s contents on every edit, rather than bounding our analysis by the extent of the visible textual area. To mitigate the efficiency concerns that this might suggest, it should first be noted that the typical length of a *TabCode* file is of the order of 200–300 words, which requires only little time to parse on modern hardware. However, such a parsing scheme would stress the display engine if a complete redraw were forced on every edit, so we have implemented some obvious optimisations. The extent of the edit, along with its typical locality of effect, are used to limit the damaged region as before, so preserving the identity of unaffected tabwords; this identity can then be used in a cache informing CLIM’s incremental redisplay mechanism.

We handle parse errors on a word-by-word basis, so that even during editing the vast majority of a *TabCode* buffer can be graphically presented, rather than only up to the current location; by returning our best guess at

[2] the term ‘beam’ is used for the grid-like representation of repeated rhythm signs, adapted from standard music notation.

the intent of a particular word and resynchronising at the next whitespace, we can preserve the tablature view mostly unchanged for most editing operations, and highlight individual errors for the user's attention.

To assist the editorial process, we have also implemented MIDI audio feedback. We provide a command to render the entire tablature in sound, and several gestures to play individual chords: one intended for use during the initial entry of the encoding, to act as a rapid error-detection aid, and a motion command and mouse gesture to assist revision and navigation. At present, this MIDI support is based on Apple's CoreMIDI framework on Mac OS X; a port to alsa-lib on Linux is in progress.

5 Future Work and Conclusions

Climacs is already a very capable editor, especially given the relatively small amount of work (only a few person-months) that has been put into it so far. Using CLIM (and in particular the McCLIM [13] implementation) as the display engine has allowed the project to progress much more rapidly than would otherwise have been possible. It should be noted that Climacs development has also revealed some serious limitations and performance problems of the McCLIM library. Nevertheless, we maintain that using CLIM and McCLIM was the best choice, and in fact advantageous to other McCLIM users as well, as the deficiencies in the McCLIM implementation are being addressed and other improvements made for use with Climacs.

Due to its reliance on fairly well-defined protocols, the Climacs text editor framework is flexible enough to allow for different future directions. Turning the Common Lisp syntax module into an excellent programming tool for Lisp programmers is high on the list of priorities, for several reasons: first, it will encourage further work on Climacs; second, the Common Lisp syntax module is likely to become one of the more advanced ones to exist for Climacs, given that Climacs has unique and direct access to the features of the underlying Common Lisp implementation.[3] Thus, the Common Lisp syntax module is likely to exercise the Climacs protocols to a very high degree. This will allow us to improve those protocols as well as their corresponding implementations.

The TTCN-3 grammar is currently defined on the core textual language. For a large subset of this language, there is a direct correspondence between TTCN-3 textual notation and TTCN-3 Graphical Representation (GR) diagrams. Implementing a live-updating TTCN-3 GR display of a parsed buffer will, in addition to being a useful application, serve as a demonstration of the utility of maintaining a full parse tree of a buffer.

Another important future direction is the planned implementation of the buffer protocol. Representing a line being edited as a Flexichain can greatly improve the performance of some crucial operations that currently require looping over each buffer object until a newline character is found. Other operations that are currently prohibitive include knowing the line- and column number of a given mark.

One disadvantage of the current parsing scheme is that a single parse error prevents analysis of the rest of the buffer, which is potentially disturbing to a user's workflow. For relatively simple grammars such as *TabCode*, it is simple enough to resynchronise at the next token, whereas for more complex grammars the resolution is less clear. Providing a framework for customisable resynchronising of the parser after a parse error would allow for more user-friendly editing.

Our plans for Climacs go further than creating an improved implementation of Emacs. We intend to make Climacs a fully-integrated CLIM application. This implies among other things using the presentation-type system of CLIM to exchange objects between Climacs and other CLIM applications such as an inspector application, a debugger pane, etc. We also hope that implementors of other CLIM applications such as mail readers, news readers, etc, will consider using Climacs for creating messages.

We are often asked whether Emacs-based applications such as VM and Gnus will be available for Climacs.

[3] The possibility of providing this level of editor integration for Prolog, given the existence of Prolog implementations embedded in Lisps, is also of interest.

We expect that replacements for these tools will use CLIM to provide a rich client user interface. Rather than fitting a mail reader into the framework of a text editor, applications can use CLIM's comprehensive user interface. Climacs will then provide a best-of-class buffer editor for these applications without restricting the ability of the application to work with graphics, graph layout, mouse interaction, multiple application panes, or other functions that CLIM does especially well.

Acknowledgements

The authors wish to thank Aleksandar Bakic for fruitful discussions. B.M. acknowledges Motorola's generous support. C.R. is supported by EPSRC grant GR/S84750/01.

References

[1] ADAMS, S. Functional pearls: Efficient sets – a balancing act. *Journal of Functional Programming 3*, 4 (1993), 553–561.

[2] CRAWFORD, T. Applications Involving Tablatures: *TabCode* for Lute Repertories. *Computing in Musicology 7* (1991), 57–59.

[3] CRAWFORD, T., Ed. *Silvius Leopold Weiss: Sämtliche Werke für Laute*, vol. 5–7. Bärenreiter, Kassel, 2002–.

[4] CRAWFORD, T., GALE, M., AND LEWIS, D. An Electronic Corpus of Lute Music (ECOLM): technological challenges and musicological possibilities. In *Conference on Interdisciplinary Musicology* (Graz, 2004), R. Parncutt, Ed., pp. 118–119.

[5] EARLEY, J. An Efficient Context-Free Parsing Algorithm. *Communications of the ACM 13*, 2 (1970), 94–102.

[6] THE EUROPEAN TELECOMMUNICATIONS STANDARDS INSTITUTE. *Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language*, 2003.

[7] FINSETH, C. A. *The Craft of Text Editing*. Springer-Verlag, 1991, 1999–. `http://www.finseth.com/craft`.

[8] GREENBERG, B. S. Multics Emacs (Prose and Cons): A commercial text-processing system in Lisp. In *LFP '80: Proceedings of the 1980 ACM conference on LISP and functional programming* (New York, NY, USA, 1980), ACM Press, pp. 6–12.

[9] INTERNATIONAL TELECOMMUNICATION UNION. *Message Sequence Chart (MSC)*, 1999.

[10] ISO. *Prolog – Part 1: General core*, 1995.

[11] MCKAY, S. CLIM: The Common Lisp Interface Manager. *Communications of the ACM 34*, 9 (1991), 58–59.

[12] PIKE, R. Acme: A User Interface for Programmers. In *USENIX Winter* (1994), pp. 223–234.

[13] STRANDH, R., AND MOORE, T. A Free Implementation of CLIM. In *International Lisp Conference* (San Francisco, 2002), Franz Inc.

[14] STRANDH, R., VILLENEUVE, M., AND MOORE, T. Flexichain: An editable sequence and its gap-buffer implementation. In *1st European Lisp and Scheme Workshop* (Oslo, 2004).

[15] TOMITA, M. An Efficient Augmented–Context–Free Parsing Algorithm. *Computational Linguistics 13*, 1–2 (1987), 31–46.

[16] WIERING, F., CRAWFORD, T., AND LEWIS, D. Creating an XML vocabulary for encoding lute music. In preparation.

[17] WILL, R. Algebraic Collections: A Standard for Functional Data Structures. `http://www.stud.tu-ilmenau.de/˜robertw/dessy/fun/`.

[18] WOOD, S. R. Z – The 95% Program Editor. In *Proceedings of the ACM SIGPLAN SIGOA symposium on Text manipulation* (1981), pp. 1–7.