Context-Sensitive Editing for the MEDFORD Metadata Language

Liam Strand[1,0009-0005-9712-1689], Andrew Powers[2,0009-0002-4878-9752], Polina Shpilker[1,0000-0002-6761-7326], Lenore Cowen[1,0000-0001-6698-6413], Alva Couch[1,0000-0002-4169-1077], and Noah M. Daniels[2,0000-0002-9538-825X]

[1] Department of Computer Science, Tufts University, Medford, MA 02155, USA,
[2] Department of Computer Science and Statistics, University of Rhode Island, Kingston, RI 02881, USA
noah_daniels@uri.edu

Abstract. The MEDFORD metadata description language was designed to enable creation of structured metadata in an ordinary text editor. This paper provides a tool that improves the ease of writing correct MEDFORD within the popular VSCode editor. This tool provides syntax highlighting and autocompletion features that generate real-time feedback in the form of suggestions which improve the discoverability of language semantics, as well as useful and informative error messages. This tool leverages the Language Server Protocol, and thus is easily ported to other text editors.

1 Introduction

The recent MEDFORD metadata description language is designed to create structured metadata for research objects (e.g. manuscripts, datasets, software, experiments) in a format that is simple to read and write for both humans and machines [6]. Designed to be easy to learn, the MEDFORD language can be used by researchers of any domain to describe metadata using any standard text editor. In MEDFORD, the structured metadata fields are indicated by a controlled vocabulary following the `@` character. A MEDFORD file is then coupled with the BagIt standard [4] to allow users to reference or optionally package their data along with their metadata [6].

The `medford` parser checks completed MEDFORD files for syntactic and semantic errors. In keeping with our goal of making it easy for humans to write correct MEDFORD files in a basic text editor, we wanted also to build an extension to that text editor to help users write correct MEDFORD files by giving real-time feedback during editing. This feedback improves the *discoverability* of the MEDFORD language by suggesting auto-completions, and also highlights errors as they are made.

In recent years, the proliferation of user-friendly programming languages (e.g. Python, Typescript, Dart) and data-description languages (e.g. TOML, YAML, JSON) has led to the development of user-assistance tools for popular

text editors. Beyond simple syntax highlighting, these tools have typically suggested completions for partially-typed keywords (including variable, function, or method names), provided inline display of warnings and errors (such as from a syntax checker, type checker, or linter), and provided quick access to build tools and documentation [7]. Effective language tools ease the learning curve of an unfamiliar language to a user [3]. In this paper, we develop such a user-assistant tool for the MEDFORD metadata language.

Our tool leverages the Language Server Protocol (LSP), a JSON-based schema for communication between a text editor and a language-specific parser, introduced by Microsoft within the last decade. Before the existence of the LSP, providing a language-specific parser required a Cartesian explosion of software to be written: for every combination of editor, language, and operating system, a custom plug-in required development and testing. This served as a barrier to adoption of new text editors, new languages, and less-popular operating systems.

A language server implementing the LSP can, in principle, serve as a "Rosetta stone" or translation hub between any text editor that supports the LSP and any language for which an LSP implementation exists. Thus, only one language server needs to be written per language, rather than the Cartesian product of languages, editors, and operating systems. In practice, some editor-specific tools may still need to be developed, but they are much simpler and smaller than an entire plug-in specific to an editor, language, and possibly operating system. Below we describe the MEDFORD editor tool in the context of the popular Microsoft Visual Studio Code editor (henceforth VSCode), but its use of the LSP means it is also easily portable to other editing environments.

2 MEDFORD's Vocabulary

Oftentimes, when beginning to write technical syntax or code in a new language or format, a common hardship is the lack of knowledge of the language [3]. The MEDFORD language is designed to be extended *by users as they write,* where users are always free to make up new `@` tags. The MEDFORD parser treats any such tag that is unknown to it as a "note" tag and passes the content through unchanged. However, MEDFORD becomes much less useful if some users are using `@author` while others are using `@contributor` and others `@writer`; thus a common vocabulary of MEDFORD terms known to the parser and standardized (as well as standard formats for dates and GPS coordinates, and so on) are greatly preferred. In fact, the parser will protect and restrict these standards for its syntactic checking of MEDFORD files.

We have documented all the MEDFORD tags which are known to the current (1.0) version of the MEDFORD parser, in a complete user manual and tutorial [5]. We have designed editor support for this set of MEDFORD tags, making use of the tools built into the average IDE in the form of support for the language server protocol.

Prior to this new work, we found anecdotally, in feedback from domain scientsts (coral researchers) who were informally testing MEDFORD in their re-

search groups, that users had trouble utilizing MEDFORD to its fullest potential. Our editor support helps users by solving the following problems:

1. Users must learn how the parser sees a file, i.e., what parts of the document have special meaning. Users need to develop an intuition for how the document is structured.
2. Users need to learn what major and minor tokens are known to the parser.
3. The parser's feedback is separated both temporally and spatially from the document being written. The user must learn how to understand the parser's error messages in order to fix them.
4. More advanced features such as macro substitution were never exposed to the user, impeding discoverability of such features.

Our work to build an editor support tool for MEDFORD alleviates these problems while retaining the language's flexibility and extensibility. A major contribution of the current work is the incorporation of autocompletion into our VSCode extension for the language. As the user is writing syntax, suggestions will appear similarly to how search engines display suggested searches when making a query. In MEDFORD, this feature has been implemented into the language server, and thus is present in the VSCode extension for the language. The advantage of this method is that the user will begin to learn the language in real time with the suggestions shown to them. In addition, a built in hover feature shows more information about the already-written syntax when the cursor is placed over it. Primarily, the benefit of this feature for MEDFORD is that it allows for users to easily see the related tokens of any given token, which are a key part of the language.

One key concept in MEDFORD is that *major tokens*, such as `@Contributor`, associate with subsidiary tokens, called *minor tokens*, such as `@Contributor-Role`. A minor token by itself makes no sense without the "parent" major token, but in designing MEDFORD, we deliberately avoided any sort of nesting or hierarchy (for instance, we did not want nested brackets). In this context, the hover feature allows the editor to suggest related minor tokens only once the user has already typed a major token. This prevents users from having to outright use documentation unless necessary and introduces an organic way for them to learn new minor tokens for major tokens that they may use frequently. As a new user, one may not be aware of every minor token associated with a given major tokens, and the benefits of this feature help to mitigate this problem. Using this feature, rather than memorize or consult the user manual, users can instead discover new tokens organically using the provided autocompletion and hover capabilities.

3 Seeing A Document Through the Parser's Eyes

A key part of attaining fluency in MEDFORD is understanding how the parts of a document are assigned semantic meaning. For example, it is important that a user understand that the metadata line:

```
@Contributor-Role Author
```

is formed from the keywords `@` and `-`, the expected major token `Contributor`, the expected minor token `Role`, and the metadata itself `Author`. Without any help, these distinctions are not immediately clear.

A strong intuition for document structure allows users to quickly and confidently make changes in the appropriate places. We help users develop this intuition by highlighting the different parts of a line in different colors. The syntax highlighter uses colors defined by the user's editor theme, so it does not make sense to discuss specific colors. Instead we will use terminology like "keyword color" or "expected token color" to describe how text is highlighted.

```
26  @Contributor A. Powers
27  @Contributor-Association Department of Computer Science
28  and Statistics,
29  University of Rhode Island
30  @Contributor-Role
```

Fig. 1. MEDFORD syntax highlighting. Here, `@` and `-` are highlighted with the keyword color, `Contributor` and `Role` with the expected token color, and `Author` with the color representing metadata values. Note that the metadata associated with a token (in this case, Contributor-Association) is allowed to span multiple lines.

For the example above, as shown in Figure 1, we highlight `@` and `-` with the keyword color, `Contributor` and `Role` with the expected token color, and `Author` with the color representing metadata values.

In this simple example, the highlighting is straightforward, but much more complicated metadata strings are possible, using macros, inline LaTeX, comments, and expected and user-defined tokens. The highlighting system helps elucidate the meaning of complex nested syntax.

4 Real-Time Feedback

We cannot expect users to write MEDFORD documents perfectly on their first attempt; even experienced MEDFORD users are likely to make small errors on a regular basis. The current paradigm for identifying these errors involves the user first writing the document to the best of their ability, then submitting that document to the parser for validation. Though we attempt to make the resulting error messages clear and specific, having the errors appear separated from the document text removes their context and makes resolving the errors more challenging.

A new user could be easily overwhelmed by dozens of lines worth of error messages after writing a few major-token blocks. This discouragement and overwhelming of new users is well-studied (and not new) in the context of introductory programming courses [2], and carries over even to non-programming languages such as MEDFORD.

To mitigate user discouragement, we implemented a system that provides immediate and contextualized feedback to the user as they type. Users are made aware of their mistakes immediately and can choose to fix them immediately, without having their writing "flow" disrupted. Furthermore, error messages are much easier to understand and resolve because they appear in the context of the text to which they refer. In VSCode, an error message appears as a red squiggly line below the erroneous text, as in Figure 2, while other editors might display errors differently.

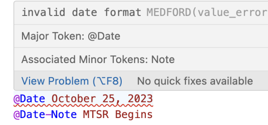

Fig. 2. MEDFORD error highlighting. In VSCode, the error is marked with a squiggly red underline.

The system is as permissive as possible. Users can keep writing even after the system displays error messages. Furthermore, the system continues to check for errors later in the document even after spotting a first error. When the system marks an error, it is not saying "stop everything and fix this", it is instead saying "you'll want to come back to this soon".

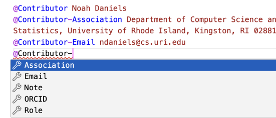

Fig. 3. MEDFORD autocompletion. In VSCode, possible completions appear as a drop-down menu.

In addition to marking errors that signify an invalid MEDFORD document, we also provide feedback for *likely* mistakes. An example of this would be the user indicating that an author is corresponding with a line like `@Contributor-Role Corresponding Author`, but failing to provide an associated `@Contributor-Email`. Similar likely mistakes, like including the template token `[..]` used for indicating metadata that must be inserted by the user, or writing a malformed DOI can also be caught with this system.

Finally, we enable *discoverability* by showing possible completions of tokens a user has typed, as illustrated in Figure 3. With this system for real-time feedback, users can write confidently, knowing that they will be made aware of any mistakes as soon as they make them.

5 Future Directions

We will add support to other platforms for writing MEDFORD files. In particular, any editor uses the Microsoft Language Server Protocol may be considered. This potentially includes Sublime Text, Neovim, Emacs, and some others. While the LSP implementation itself need not change, a lightweight "shim" is needed for some editors, while other editors can talk to the language server directly. Of course, testing is needed across all editors.

Beyond editor support, there is also interest in whether generative AI can be utilized to more intelligently correct or prompt users to document metadata fields. Recent developments in large language models [1] have enabled prompt-based generation of functioning code, unit tests, and documentation across a

number of programming languages. Of course, in the case of MEDFORD, writing a prompt with all the metadata would be as much work as writing the MEDFORD markup itself. However, we plan to investigate how large language models can be given a *data source*, such as the PDF of a journal article, and simultaneously extract the metadata and mark it up appropriately as valid MEDFORD. We can envision extending this concept to other data sources, such as photo collections, genomic or other sample data, and the like.

6 Availability

The MEDFORD parser and editor support are freely available at:

– https://github.com/TuftsBCB/medford
– https://github.com/TuftsBCB/medford-language-server
– https://github.com/TuftsBCB/medford-vscode

7 Acknowledgments

We thank the anonymous reviewers for their valuable suggestions. The initial development of MEDFORD was supported in part by the National Science Foundation under NSF grants OAC-1939263. OAC-1939795 and OAC-1940233. The current work was supported by a seed grant from the Tufts Data Intensive Study Center (DISC). Liam Strand thanks the Tufts University Summer Scholars program and the Fowler Family.

References

1. L. Floridi and M. Chiriatti. Gpt-3: Its nature, scope, limits, and consequences. *Minds and Machines*, 30:681–694, 2020.
2. M. Harman and S. Danicic. Using an interpreter to teach introductory programming. *WIT Transactions on Information and Communication Technologies*, 7, 1970.
3. A. J. Ko, H. H. Aung, and B. A. Myers. Design requirements for more flexible structured editors from a study of programmers' text editing. In *CHI '05 Extended Abstracts on Human Factors in Computing Systems*, CHI EA '05, page 1557–1560, New York, NY, USA, 2005. Association for Computing Machinery.
4. J. Kunze, J. Littman, E. Madden, J. Scancella, and C. Adams. The bagit file packaging format (v1. 0). Technical report, 2018.
5. A. Powers, L. Strand, L. Cowen, A. Couch, P. Shpilker, and N. Daniels. *MEDFORD User Guide*, Aug. 2023.
6. P. Shpilker, J. Freeman, H. McKelvie, J. Ashey, J.-M. Fonticella, H. Putnam, J. Greenberg, L. Cowen, A. Couch, and N. M. Daniels. Medford: A human-and machine-readable metadata markup language. *Database*, 2022, 2022.
7. V. K. Swarnkar and K. Satao. A survey on performance of different text editor, 2013.