

PyVallex: A Processing System for Valency Lexicon Data

Jonathan Verner^{1,2}, Anna Vernerová¹

1: Charles University, Faculty of Mathematics and Physics, Institute of Formal and Applied Linguistics

2: Charles University, Faculty of Arts, Department of Logic

1: Malostranské náměstí 25, 118 00 Prague 1, Czech Republic; 2: Celetná 20, 110 00 Prague 1, Czech Republic
jonathan@temno.eu, vernerova@ufal.mff.cuni.cz

Abstract

We present PyVallex, a Python-based system for presenting, searching/filtering, editing/extending and automatic processing of machine-readable lexicon data originally available in a text-based format. The system consists of several components: a parser for the specific lexicon format used in several valency lexicons, a data-validation framework, a regular expression based search engine, a map-reduce style framework for querying the lexicon data and a web-based interface integrating complex search and some basic editing capabilities. PyVallex provides most of the typical functionalities of a Dictionary Writing System (DWS), such as multiple presentation modes for the underlying lexical database, automatic evaluation of consistency tests, and a mechanism of merging updates coming from multiple sources. The editing functionality is currently limited to the client-side interface and edits of existing lexical entries, but additional script-based operations on the database are also possible. The code is published under the open source MIT license and is also available in the form of a Python module for integrating into other software.

Keywords: dictionary management, lexicographic software, electronic dictionaries, valency lexicons

1. When The Old Tools No Longer Suffice

In the early 2000s researchers at the Institute of Formal and Applied Linguistics in Prague (then the Center for Computational Linguistics) started developing the valency lexicon Vallex aiming to validate the existing valency theory of the Functional Generative Description (Panevová, 1974; Panevová, 1975; Panevová, 1980; Panevová and Skoumalová, 1992) and to pursue further syntactic research (Lopatková et al., 2002). Early on, a lexicon named PDT-Vallex was forked from Vallex and used for maintaining consistency of the tectogrammatical annotation of the data in the Prague Dependency Treebank 2.0 (Hajič et al., 2006; Hajič and Honetschläger, 2003). The PDT-Vallex was fully integrated with the tree editor and search tool TrEd¹ (Pajas and Štěpánek, 2008) and its native format is now XML-based. In contrast, the Vallex project still uses the original pipeline based on a text-based format converted by a sequence of scripts to an intermediate representation as XML-data (Žabokrtský, 2005) and final presentations as web-pages (with individual lexicon entries stored in html files) and a PDF/printed lexicon (produced via TeX). The annotators are reluctant to move away from the text-based format which they can use with a common text-editor (with custom syntax highlighting definition files relying on a set of markup conventions); this situation is not unique to the Vallex project, e.g. (Benko, 2019). On the other hand, the set of custom scripts for format conversions, validity checking, and custom search are not well documented and have a significant maintenance cost. The scripts are written using tools/languages which are not as widely popular today as they were when the project started (*perl*, *awk*, *xslt*), making it hard for new team members to actively take part in further development. This is particularly problematic when need for a significant new functionality arises. A case in point is the ability to validate link-type attributes for which the old pipeline has no efficient mechanism. In recent years

we have seen a sharp rise in the number of internal links between different lexical units within Vallex when phenomena such as lexical-semantic alternations and light verb constructions became the central topics of our research. It has now become necessary to monitor changes in the data and check that they do not break the links or produce inconsistencies. This validation requirement now also extends to cross-lexicon links since Vallex now has three additional sister projects, all linking back to its data: NomVallex, a lexicon of deverbal nouns; RU-Vallex, a lexicon of Russian verbs; and PL-Vallex, a lexicon of Polish verbs. Integrating such cross-lexicon link validation into the old pipeline seems close to impossible.

2. Motivation & Design

As noted above, the main motivation for designing a system from scratch was that the mostly unstructured nature of the previous system made it difficult to add new functionality and maintain the system. The design of the system did not allow for easy integration into other tools which was compounded by almost non-existing documentation. On the other hand, the older system provided significant functionality which the new system needed to replicate. An additional strict requirement was that annotators could continue working as they were used to, in other words, using the text-based data format and Subversion revision system, for creating new entries as well as editing old ones. The above considerations lead to the following design goals for the new system:

Modularity To make adding new functionality easier, the system should be structured into components, each with a well defined public interface (API). The components should only communicate with each other through this API. This allows any new functionality to modify only the relevant parts of the system and thus lowers the barrier for new contributors—a developer does not need to have precise knowledge of the whole system to add/modify functionality to/of a single component.

¹<https://ufal.mff.cuni.cz/tred/>

Extensibility The system should provide as much as possible of the functionality already present in the older system, and moreover be designed from the start to be flexible and easily extensible. In particular, it should have a well-defined data-model and provide an interface to allow accessing the data from other programs (either through a library interface or, e.g., a REST API).

Maintainability The design should aim to minimize the maintenance cost of the system. In particular the system should be well documented, the components should be covered by an automated test suite and the code style should be as uniform as possible and follow best practices.

While many of the needed functionalities could be achieved through existing Dictionary Writing Systems (WDS; the concept is discussed e.g. in (Abel, 2012)), these systems were ruled out by the requirement that annotators be allowed to continue using their current workflows.

Given the above, we have decided to implement the system in the *Python 3* programming language. Its advantage over *Perl* (the main language of the previous system) is that it provides much better language support for structured programming. It is also becoming much more popular in the NLP community, which makes it more likely that new contributors will be able to work with the code. Although Python (as is the case also for Perl) is a weakly typed language, we have opted to enforce the use of type hints via the *mypy* static type checker (The MyPy project, 2019) run before every commit. This makes the code effectively strongly-typed. In the interest of maintainability, we also enforce a uniform coding style through the use of a commit-hook which runs *autopep* (Hattori et al., 2019) on the committed code.

We expect that most annotators will learn to use selected features of PyVallex over time, especially the search/filtering functions, tabular output and editing of existing lexical units. The old system of custom-made scripts will be used alongside PyVallex for some time, until all of its functionalities are fully implemented and tested.

3. Data Layer

A valency lexicon consists of a collection of *Lexemes*. A Lexeme represents a group of related lexical units that share the same lemma or (as in the case of Vallex, NomVallex, and other lexicons) a group of derivationally related lemmas. Each lexical unit corresponds to a single meaning of these lemmas. Each lexical unit can be annotated with a number linguistic properties, e.g. semantic (a gloss of the given meaning, indication of primary and metaphorical meanings), syntactic (does the given lexical unit enter syntactic structures such as passive, reflexive and reciprocal constructions?), and features specific to valency lexicons (the valency frame and annotation of individual valency complementations with a functor, obligatoriness and a list of possible forms of expressions; indication of control, i.e. for a given lexical unit realizing one of its complementations through an infinitive, indicate which other valency complementation is referentially identical with the subject of this infinitive).

The data layer definition is provided in the

`vallex.data_structures` module² and consists of the following classes: `Lexicon` (representing a collection of *lexemes*), `Lexeme` (representing a single *lexeme*), `LexicalUnit` (representing a *lexical unit*, i.e. a single meaning), `Attrib` (representing a linguistically relevant property of a *lexical unit*). The `Attrib` class also has several specializations: `Frame` (representing the *valency frame*, represented as a list of *complementations*), `Lemma` (representing the *lemma* or set of lemmas), and `SpecVal` (representing the changes between the valency of two related units, e.g. a verb and a deverbal noun or a verb and its translation to another language). It is expected that further specializations will be defined to deal with particular properties.

Each of the above classes can store textual comments (which can be used to explain the reasoning behind a specific annotation, mark the data element as work in progress, etc) and they all provide a method to convert the data into a JSON representable structure.

4. Core components

The core of the system consists of a *parser module* which takes care of parsing the data and constructing the data-layer—an in-memory representation of the data; a *search module* which provides query capabilities; a *script module* which provides a framework for running data validation and batch processing scripts; and an *output module* which converts the in-memory representation into various output formats. Each of these components is implemented in a python submodule of the main `vallex` module and the components are mostly independent of each other.

4.1. The parser

The parse module takes care of parsing the specialized format (Figure 1) used by annotators when creating the lexicon data. The format is designed to be easily editable in any text editor and concise enough to facilitate manual creation.

The parser itself is split into a tokenizer function producing a stream of tokens and several parse methods. The methods take care of constructing the `Lexeme`, `LexicalUnit` and `Attrib` classes. It is designed in such a way that adding a specialized parser for a newly introduced property is just a matter of writing a single function which is given the body of the attribute and returns an instance of (a descendant of) the `Attrib` class. To integrate the function into the parser, it is enough to decorate it with a provided Python decorator.

4.2. Search

The search module provides query capabilities to the system.

4.2.1. Queries

A query is a collection of conditions and the result of the query is a set of lexical units each of which meets all of the conditions of the query³. Each condition consists of

²For historical reasons, the main package is called `vallex` although most of the functionality is not specific to the 'Vallex' lexicon.

³Experience seems to suggest that disjunctive conditions are not used very much. A limited spectrum of negative conditions

```

: id: blu-v-brát-vzít-1
~ impf: brát (si) pf: vzít (si)
+ ACT(1;obl) PAT(4;obl) ORIG(od+2;opt)
  LOC(;typ) DIR1(;typ) RCMP(za+4;typ)
-synon: impf: přijímat; získávat
        pf: přijmout; získat
-example: impf: brát si od někoho mzdu
          pf: vzal si od něj peníze za práci
-note: volné si
       mohli brát na odměnách.COMPL
       měsíčně 26 až 40 tisíc korun
-recipr: ACT-ORIG
-diat: no_poss_result no_recipient
       pf: deagent: peníze navíc se
           musí odněkud vzít
           passive
       impf: deagent
           passive za práci se bere mzda

```

Figure 1: Sample lexical unit in the textual format (simplified)

a *selector* and a *pattern*. For each lexical unit the *selector* is passed to a `match_key_values` function which constructs a list of strings. A lexical unit satisfies the condition if at least one of the strings matches the *pattern*, which is a regular expression. The *selector* is a dot-separated list of strings. The standard implementation of the `match_key_values` function interprets the first element of the selector as an attribute name which it retrieves. It then passes the rest of the selector to the attribute's `match_key_values` function to construct the list of strings to be matched against. The standard implementation of the attribute's `match_key_values` method returns the attribute's textual representation if the *selector* is empty, its source form if the selector is 'src' and otherwise treats the selector as a path through the attributes structure treated as a tree. It resolves the path and returns the value present in the relevant node. For example a query consisting of the single condition

```
example.pf=.*od.*
```

would match the lexical unit shown in Figure 1 whose `example` attribute has the following structure

```

{
  'impf': [ 'brát si od někoho mzdu'],
  'pf': [ 'vzal si od něj peníze za
          práci']
}

```

The *selector* `example.pf` would retrieve the `example` attribute and from it its `pf` node which contains the string `vzal si od něj peníze za práci`. This string incidentally matches the regular expression `.*od.*`.

For discoverability purposes, each attribute has a method which returns a lists of all valid paths inside its structure. Note that besides the internal structure of attributes output

can be formulated. At the cost of complexity of the query language, it would be easy to extend it to also additional types of queries.

by the parser (Section 4.1.), selectors may also access computed properties and results of tests (Section 4.3.); in particular, it is possible to formulate a query for all units that failed a given test, e.g. a query of the form

```
error.lvc_references=.
```

would match⁴ all lexical units that failed the test shown in Figure 4.

4.2.2. Executing queries

The search module contains a `grep` method to execute queries. It additionally contains a `filter` method which allows pruning each lexical unit in the result set so that it contains only the properties/attributes a user is interested in. Another method is provided to compute various histograms. The `histogram` method takes three arguments: the first argument is a collection of lexical units over which the histogram is computed. The second argument is a *selector* which produces a list of strings from each lexical unit in the same way as is done when evaluating queries. The last argument is a regular expression which extracts the values to be counted from each string from the list. Figure 2 shows how the UI displays the results of computing the histogram for the `frame.functor` selector with the trivial pattern `(.*)`.

4.3. Scripts

The script module (`vallex.scripts`) provides a framework for running simple procedures over the valency lexicon data. The scripts are loaded by the framework from a configurable directory. Each file in this directory is a Python source file containing the definitions of the procedures. The system currently recognizes five kinds of procedures: *test*, *transform*, *compute* and *map/reduce*.

Test The test procedures are used to implement data validation for the lexicon. Each test function receives a lexical unit as its argument.⁵ It checks whether the unit satisfies the test and raises an appropriate exception if it doesn't. The framework iterates over all lexical units passing them in turn to each test procedure and collecting the results. The results are saved in the in-memory representation and can then be displayed to the user (Figure 3). The results are also annotated with the docstring of the test-procedure which can be used to provide human readable explanation of the failed result. An example data validation test is provided in Figure 4.

Transform The transform procedures can be used to implement one-time lexicon-wide changes, e.g. renaming an attribute. They receive a lexical unit which they can modify and return.

⁴The pattern consists of a single dot so it matches any unit with a *non-empty* value of `error.lvc_references`.

⁵Actually, there are four sub-types differing in what argument is passed — a collection of lexicons, a single lexicon, a single lexeme or a lexical unit; for simplicity here and also in the paragraphs dedicated to other types of procedures we discuss only the sub-type receiving a lexical unit.

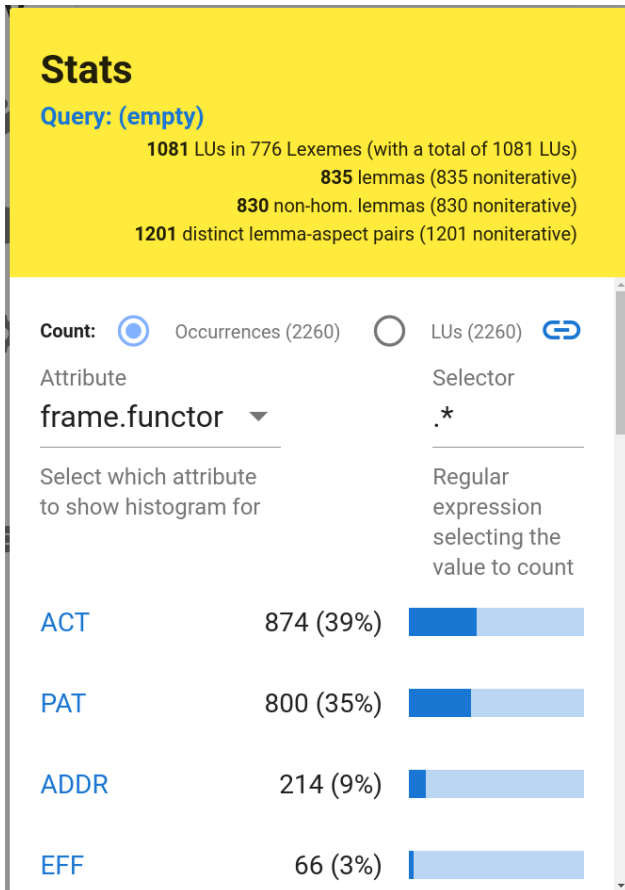


Figure 2: The web-based UI showing a histogram of the frame functors

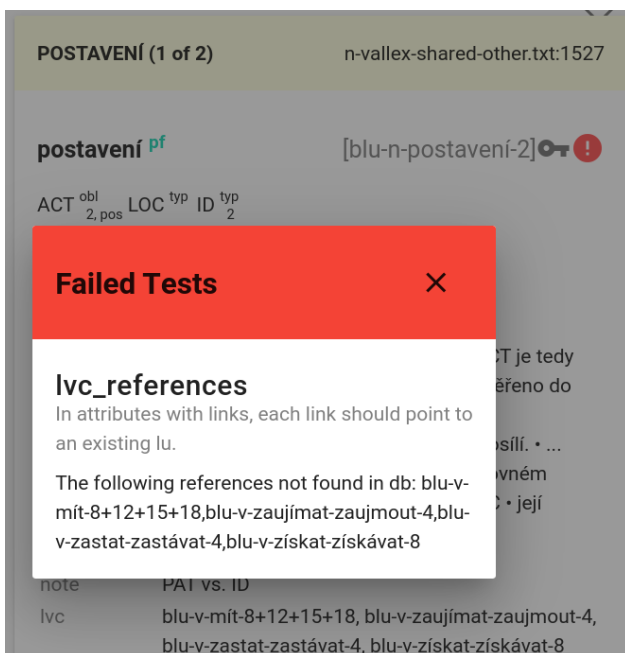


Figure 3: The web-based UI showing the result of a data-validation run on a lexical unit.

```
@requires('lumap')
def test_lu_lvc_references(lu, lumap):
    """
    In attributes with links, each link
    should point to an existing lu.
    """
    failures = []
    lvc_variants = [k for k in lu.attrs.keys()
                    if k.startswith('lvc')]
    if not lvc_variants:
        raise TestDoesNotApply
    applies = False
    for attrib in lvc_variants:
        if not \
            isinstance(lu.attrs[attrib]._data,
                       dict):
            continue
        refs = lu.attrs[attrib]._data['ids']
        if refs:
            applies = True
            for ref in refs:
                if not ref._id.startswith('@') \
                    and ref._id not in lumap:
                    failures.append(str(ref._id))
    if failures:
        raise TestFailed("The following \
            references not found in db: "
            + ','.join(failures))
```

Figure 4: An example of a data-validation procedure

Compute The compute procedures are similar to the transform procedures, but are used to implement dynamically computed properties which are *not* saved back to the lexicon on disk.

Map/Reduce The map/reduce procedures are used to perform more complicated analyses of the lexicon for which a simple search/histogram does not suffice. Each map/reduce procedure consists of a pair of functions: a *mapper* and an (optional) *reducer*. Each mapper receives a lexical unit as an argument and uses a framework-provided emit function to emit a collection of (key, value) pairs. The framework iterates over all lexical-units passing them to the mapper functions and collecting the resulting pairs. It then groups them by the *key* component and passes the groups to the reducer which can do further processing. A default reducer which just counts the number of values for a given key is provided by the framework and is used when no specialized reducer is provided by the user. Note that, although map/reduce is now commonly associated with parallel processing, here all the mappers and reducers are run sequentially—while running them in parallel would be possible, the small size of the data does not justify the additional complexity of parallel processing. We use map/reduce only as a familiar paradigm for structuring analysis code.

4.4. Output

The output module provides tools to export lexicon data in various formats. In addition to built-in JSON output (which is implemented in the data-layer), new formats can be de-

fined using *Jinja2* (Ronacher, 2019) templates. Currently only a single txt format is provided which outputs the in-memory representation in the same format that the annotators use. This can be used to check the fidelity of the in-memory representation (by comparing it with the original source) and for normalizing the sources. In the future other formats may be added, e.g., one of the XML-based internationally recognized formats such as XML:TEI or the RDF/XML serialization of the OntoLex-Lemon model.

5. The UI Layer

Although the PyVallex system provides a command line UI for performing searches, computing histograms and running batch scripts, it is expected that most of its users will prefer a nicer graphical user interface. We have decided to provide the GUI as a web-based interface. This has several advantages. First, it allows the system to be installed on a server and be accessible to users without forcing them to acquire the needed datasets or requiring them to maintain the installation. Moreover, web-based technologies are very common and basing the UI on them considerably lowers the barrier for new contributors/maintainers. It is expected that even a person without a detailed knowledge of the system would be able to contribute simple modifications to the UI in a short amount of time. Finally, using a webview widget provided by the *Qt library* (Qt Company, 2019), we can implement a simple local client based on the same code-base.

5.1. Implementation

The server-backend is a simple Python *WSGI* (Eby, 2010) application written in the *Bottle.py* (Hellkamp, 2019) microframework. It uses an *SQLite* database (Hipp, 2018) to store a JSON representation of the in-memory data⁶ and exposes a REST-based api which is consumed by the front end. The front end is a Javascript application written using the *Vue* framework (You and others, 2019) together with the *Vuetify* component library (Leider and others, 2019) to provide a familiar *Polymer*-style interface (Google, Inc., 2015).

5.2. User Features

5.2.1. Searching

By default, all lexemes in the lexicon are displayed in alphabetical order. However, the GUI provides an interface for formulating queries (Section 4.2.1.); to improve user experience, all existing selectors are listed in a drop-down list (Figure 5). To clarify the meaning of selectors, a help string may be provided; the user may also click on the key icon next to any lexical unit and see a full listing of the selector-value pairs for the given unit.

5.2.2. View modes

Currently, two view modes are provided for lexical units: in the Source View mode, the complete portion of the source file that corresponds to the given lexical unit is run through a simple regex-based syntax highlighter that mimics the function of the text editor known to the annotators. On

⁶The database is used solely as a method to allow safe concurrent access to the data.

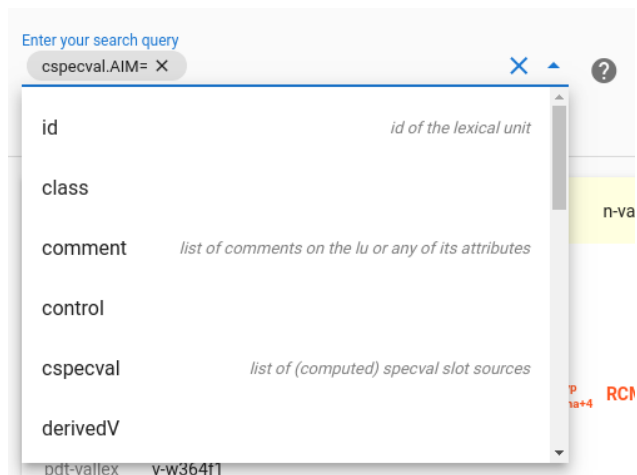


Figure 5: The search field

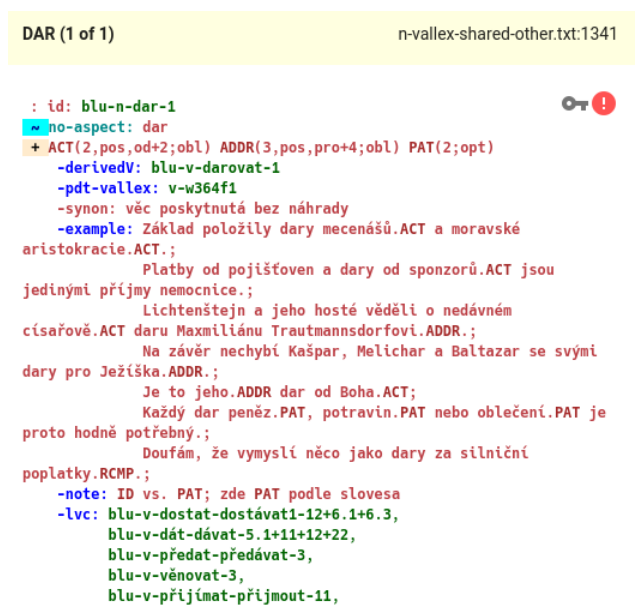


Figure 6: The display of a lexical unit (Source View)

the other hand, the Default View mode is based on a template that can access the complete output of the parser (Section 4.1.) as well as computed properties not present in the source files (Section 4.3.) and the full formatting capabilities of the *Vue.js* framework. A particularly useful feature of the Default View is the fact that linked lexical units (such as source verbs of deverbal nouns or translation equivalents in case of our Russian and Polish lexicons) can be displayed in a pop-up with a single click. The two views can be compared in Figures 6 and 7.

The `filter` method (Section 4.2.2.) may be triggered from the settings menu as a selection of attributes to be displayed. A simple interface for showing various histograms is also available (ibid.).

5.2.3. Basic Editing

Because of the assumption that the web-based GUI will not replace the existing annotation pipeline (text files stored in

DAR (1 of 1) n-vallex-shared-other.txt:1341

dar no-aspect [blu-n-dar-1]  

ACT^{obl}_{2, pos, od+2} ADDR^{obl}_{3, pos, pro+4} PAT^{opt}₂

ACT^{obl}_{1→2, pos, od+2} ADDR^{obl}_{3, pos, pro+4} PAT^{obl→opt}_{4→2} CAUS^{typ}_{k+3} AIM^{typ}_{na+4}

RCMP^{typ}_{za+4}

derivedV blu-v-darovat-1
 pdt-vallex v-w364f1
 synon věc poskytnutá bez náhrady

example Základ položily dary mecenášů.ACT a moravské aristokracie.ACT.
 • Platby od pojišťoven a dary od sponzorů.ACT jsou jedinými příjmy nemocnice. • Lichtenštejn a jeho hosté věděli o nedávném císařově.ACT daru Maximiliánu Trautmannsdorfovi.ADDR. • Na závěr nechybí Kašpar, Melichar a Baltazar se svými dary pro Ježíška.ADDR. • Je to jeho.ADDR dar od Boha.ACT • Každý dar peněz.PAT, potravin.PAT nebo oblečení.PAT je proto hodně potřebný. • Doufám, že vymyslí něco jako dary za silniční poplatky.RCMP.



Figure 7: The display of a lexical unit (Default View)

a Subversion repository), the current implementation does not provide editing functionality in the web-based interface. Nonetheless, a simple component for editing a single (pre-existing) lexical unit is available in the local client-side *Qt*-based GUI, so that annotators may quickly correct minor issues encountered while searching the data; the client initially reads data from the working copy of the Subversion repository and stores a JSON representation of this data in an *SQLite* database for safe concurrent access. The user can open an editor view (see Figure 8) and do some edits. When the editor view is closed, the system first checks that the state of the entry in the database has not changed since opening the editor, then parses the new textual form of the lexical unit and replaces the whole unit in the database, also updating its in-memory representation. In case the unit has changed since the editing window was opened, the user is warned and no changes are stored. Upon request, the contents of the *SQLite* database can be output in the textual form back to the subversion repository the data was originally read from (making sure that any files changed in the meantime are backed up; see Section 4.4.); it is the users' responsibility to commit the changes, resolve conflicts etc.

6. Future work

Some features of our old pipeline, e.g. XML and PDF output, have yet to be implemented in PyVallex. PyVallex is currently only available in source code form⁷, although we plan to publish it as pre-built package in the PyPi package repository. The client-side *Qt*-based GUI is now distributed as a Windows and a Unix standalone executable which should, in principle, be runnable on multiple versions of the operating systems. Although this has some advantages (simplicity, easy updates), it has a significant performance cost—the single file is basically a self-extracting archive which must unpack itself every time the program is

⁷<https://gitlab.com/Verner/pyvallex>

Edit  

```

: id: blu-v-absolvovat-1
~ biasp: absolvovat
+ ACT(1;obl) PAT(4;obl)
-synon: zakončit, končit
-example: absolvovat studium
-use: prim
-class: phase verb
-diat: deagent %studium se absolvovalo s vyznamenáním% [made-up]
passive-být %DAAD ve svém rozhodnutí také určit, s jakým
výsledkem má být poslední státnice absolvována, aby mohlo být
stipendium vyplaceno.% [SYN]
poss-result-nconv-mít %Kurzy s počtem do 30 žáků musí
doprovázet 3 dospělé osoby, z nichž jedna musí mít absolvován kurz
první pomoci.% [SYN]

```

Figure 8: A simple editor

run. This choice should be revisited once we gather more user feedback.

While the client-side *Qt*-based GUI allows for basic editing, it would be nice to incorporate this into the web-based UI. This will require some form of user authentication/authorization to be implemented. Eventually, the editing capabilities could be extended to allow for more comprehensive edits (including creating new lexemes/lexical units). This would allow some annotators to use the PyVallex system as the main or only tool to create or edit lexicons, and external users to efficiently submit feedback on individual lexical units as well as suggest extensions of the lexicon.

Finally, the modular architecture of PyVallex which allows programmatic access to the data in a structured way makes it an ideal tool for script-based examination of multiple revisions of the same data. We therefore plan to implement additional tests for guarding the internal and mutual consistency of the annotation present in all four sister-projects that use the Vallex format: Vallex, NomVallex, RU-Vallex and PL-Vallex.

6.1. Extending PyVallex for other projects

A recent survey of lexicographic practices in Europe has found that most projects use an in-house DWS rather than an off-the-shelf solution (Kallas et al., 2019). Even though PyVallex is not primarily a DWS, it can be viewed as just another addition towards this trend. One of the arguments for an in-house solution was the fact that the lexicographers on our teams are happy with the current setup: the main source of truth about the lexicon data are files in the text-based Vallex format, shared between annotators in a Subversion repository. In contrast to XML-based formats such as XML:TEI (TEI Consortium, 2020) and the RDF/XML serialization of the OntoLex-Lemon model (Bosque-Gil et al., 2019), our format is compact and practical for direct and daily use by humans; moreover, it is easy to inspect changes in the data with general-purpose diff tools. This means that in order to build upon pre-existing software for the presentation, querying and editing of our lexicographic data, we would have to write custom conversion scripts to and from the native format of those tools, and create a pipeline allowing for continuous integration of changes

coming from the Subversion repository. However, once we have a parser for our format, it is natural to use the resulting internal representation of the data as the input for queries, scripts, and output functions. In short, the PyVallex framework is a result of the wish to enable developers to write a parser and then reuse the resulting representation for multiple purposes. Although most of the current development of PyVallex concentrates on the Vallex-specific functions, the general framework is ready to work with other data models as well. We thank the anonymous referees for pointing out that as a first step towards adoption of PyVallex by other projects, we could implement parsers and output functions for internationally recognized lexicon exchange formats. In fact, we could go even further and make sure that the internal representation of lexicon data in PyVallex reflects the OntoLex-Lemon model.

7. Acknowledgements

The research reported in this paper has been supported by the projects No. 18-03984S (*Between Reciprocity and Reflexivity: The Case of Czech Reciprocal Constructions*) and No. 19-16633S (*Valency of Non-verbal Predicates*) of the Czech Science Foundation (GAČR) and partially also by the LINDAT/CLARIN project of the Ministry of Education, Youth and Sports of the Czech Republic (project No. LM2015071).

The first author was also supported by Charles University Research Centre program No. UNCE/SCI/022 and by the Progres grant Q14. Krize racionality a moderní myšlení.

This work has been using language resources stored and distributed by the LINDAT-CLARIN project of the Ministry of Education, Youth and Sports of the Czech Republic, project No. LM2015071.

We thank the anonymous referees for their valuable suggestions, and in particular the second referee for his very detailed report.

8. Bibliographical References

- Abel, A., (2012). *Dictionary Writing Systems and Beyond*. Oxford University Press, Oxford.
- Benko, V. (2019). LexiCorp: Corpus approach to presentation of lexicographic data. In *Proceedings of eLex 2019*.
- Bosque-Gil, J., Gracia, J., et al. (2019). The OntoLex Lemon lexicography module. Technical report.
- Eby, P. J. (2010). Python Web Server Gateway Interface v1.0.1. *Python Enhancement Proposals*, PEP 3333.
- Google, Inc. (2015). Polymer project. <https://www.polymer-project.org/>.
- Hajič, J., Panevová, J., Hajičová, E., Sgall, P., Pajas, P., Štěpánek, J., Havelka, J., Mikulová, M., Žabokrtský, Z., Ševčíková-Razímová, M., and Uřešová, Z. (2006). Prague Dependency Treebank 2.0 (PDT 2.0). LINDAT/CLARIN digital library at the Institute of Formal and Applied Linguistics (ÚFAL), Faculty of Mathematics and Physics, Charles University.
- Hajič, J. and Honetschläger, V. (2003). Annotation lexicons: Using the valency lexicon for tectogrammatical annotation. *The Prague Bulletin of Mathematical Linguistics (PBML)*, 79–80:61–86.
- Hattori, H., Myint, S., and Wendling, B. (2019). Autopep8 1.4.4 (2019-11-30). <https://pypi.python.org/pypi/autopep8/>.
- Hellkamp, M. (2019). Bottle: Python Web framework. <https://bottlepy.org/docs/dev/>.
- Hipp, D. R. (2018). SQLite. <https://www.sqlite.org/>.
- Kallas, J., Koeva, S., Langemets, M., Tiberius, C., and Kosem, I. (2019). Lexicographic practices in Europe: Results of the ELEXIS Survey on user needs. *Electronic lexicography in the 21st century (eLex 2019): Smart lexicography*, page 144.
- Leider, J. et al. (2019). Vuetify.js. <https://vuetifyjs.com/en/>.
- Lopatková, M., Žabokrtský, Z., Skwarska, K., and Benešová, V. (2002). Tektogramaticky anotovaný valenční slovník českých sloves. Technical Report TR-2002-15, ÚFAL/CKL MFF UK, Praha.
- Pajas, P. and Štěpánek, J. (2008). Recent advances in a feature-rich framework for treebank annotation. In Donia Scott et al., editors, *The 22nd International Conference on Computational Linguistics - Proceedings of the Conference*, volume 2, pages 673–680, Manchester, UK. The Coling 2008 Organizing Committee.
- Panevová, J. and Skoumalová, H. (1992). Surface and deep cases. In *Proceedings of the 14th conference on Computational linguistics, COLING'92*, volume 3, pages 885–889, Nantes, France, august. Association for Computational Linguistics.
- Panevová, J. (1974). On verbal frames in Functional Generative Description, Part I. *Prague Bulletin of Mathematical Linguistics*, 22:3–40.
- Panevová, J. (1975). On verbal frames in Functional Generative Description, Part II. *Prague Bulletin of Mathematical Linguistics*, 23:17–37.
- Panevová, J. (1980). *Formy a funkce ve stavbě české věty*, volume 13 of *Studie a práce lingvistické*. Academia, Praha.
- Qt Company. (2019). Qt 5.12.1 (2019-11-30). <https://www.qt.io/>.
- Ronacher, A. (2019). Jinja 2.10.3 (2019-11-30). <https://palletsprojects.com/p/jinja/>.
- TEI Consortium. (2020). TEI P5: Guidelines for electronic text encoding and interchange 4.0.0. Technical report.
- The MyPy project. (2019). Mypy 0.750 (2019-11-30). <http://www.mypy-lang.org>.
- You, E. et al. (2019). Vue.js. <https://vuejs.org/>.
- Žabokrtský, Z. (2005). *Valency Lexicon of Czech Verbs*. Ph.D. thesis, Charles University in Prague, Praha.