Tabular JSON: A Proposal for a Pragmatic Linguistic Data Format

Adam Roussel
Department of Linguistics
Ruhr University Bochum
roussel@linguistics.rub.de

Abstract

Existing linguistic data formats tend to be very general and powerful yet difficult to use on a day-to-day basis, so that practitioners often reach for underpowered ad-hoc text formats that require error-prone string parsing. We propose a pragmatic JSON-based linguistic data format that is flexible enough to cover most types of linguistic annotations and scenarios. It avoids the need for string parsing, as the serialized data representation is trivially convertible to tabular data structures that are immediately usable in data analysis applications.

1 Introduction

While there are very many data formats that have been introduced for use with linguistic data, they seem to either be highly general and capable of representing any kind of annotation yet unwieldly to use, thus requiring extra software to translate between the abstract underlying data model and a more application-specific und user-friendly view of that data, or they are easy to work with but very limited in the kinds of annotations that they support. There seems to be room for practical data formats that lie somewhere in the middle, ones that are lightweight and easy to use, yet flexible and capable of supporting a range of possible annotations. This is the sort of format that Tabular JSON is intended to be.

2 Related Work

Specialized data formats for linguistic corpora include ones such as Salt (Zipser and Romary, 2010) or Paula XML (Dipper, 2005; Dipper and Götze, 2005; Chiarcos et al., 2008). These are capable of representing any kind or nearly any kind of annotation, since their main purpose is the exchange of corpus data between systems and the long-term storage of data. However, due to their generality, they are complex formats and are not suitable as everyday working formats. Generally, some kind of specialized software is required to translate the general representation on disk to something usable in a given application scenario.

Formats such as UIMA CAS XMI[1] and FoLiA (van Gompel and Reynaert, 2013) are somewhat less complex and more human-readable, with UIMA CAS making greater use of stand-off representations. Both support a broad range of possible annotation types. Though new type systems may be defined with UIMA CAS, there doesn't appear to be a straightforward way to add custom annotation types to FoLiA. Both of these formats are nevertheless complex enough to warrant the use of specialized software in order to produce and consume data in these formats, DKPro-Cassis (Klie and de Castilho, 2024) and the FoLiA Python library, respectively.

Finally, perhaps the most widely used formats are those derived from the CoNLL-X formats (Buchholz and Marsi, 2006), CoNLL-U[2] most prominent among them. These are all text formats that have one token per line, tab-separated fields, and sentences separated by empty lines. The variants may have different numbers of columns, which contain different kinds of data – this is generally determined by the variant name, but the CoNLL-U Plus format allows for the number and names of columns to be specified in a special header line.

Though the format is fairly human-readable, parsing (and re-parsing) it is inefficient and error-prone, as different users are bound to overlook different edge cases. These problems are compounded in cases where more complex types of annotations are to be represented and new ad-hoc representations are invented to accomodate them within the confines a single column.

[1] https://uima.apache.org/
[2] https://universaldependencies.org/format.html

3 Description

3.1 Guiding principles

The main goal of Tabular JSON is to be a format that is practical for daily use:

- Minimize ad-hoc parsing.
- Require no special software.
- Support a variety of annotation types.

Reading and writing the data should not require users to do string parsing, which is inefficient and error-prone. Ideally, once the data are parsed, they remain in a structured form and can be read or written with no further format-specific knowledge or software required: a general-purpose JSON parser should suffice.

A practical format for linguistic data needs to be able to represent a broad range of annotation types: It is not uncommon to have annotations that have the shape of spans or relations between tokens or spans. These types of entities, among others, ought to be naturally representable making hacks to represent them on a token-wise basis unnecessary.

3.2 Data model

It's important to distinguish between the data format that is used on disk and the data model that is expressed in that format. To some extent, they're related, since some formats are not capable of representing some kinds of logical entities. Trees don't go well with CSV, for instance, but they're a natural fit for XML.

In the interests of practicality we use a data model that is essentially tabular, which ensures seamless compatibility with common data analysis packages. Our model follows broadly the principles of "tidy data" (Wickham, 2014): Tidy data is characterized by observations or basic units of analysis being represented in rows and various variables or properties of those units of analysis being represented in columns. Tidy data is easier to work with, to reshape and to analyze, and it works well with vectorized operations, such as are used in R or Pandas.

3.3 Data formats

A suitable tabular representation could then be implemented in any of a number of formats – CSV, JSON, SQLite, Parquet, XML, etc. are all perfectly capable of representing a sequence of objects with some fixed set of attributes. We chose JSON over the alternatives because it is immediately and intuitively usable and keeps the parsing of text formats to a minimum.[3]

CSV (and TSV) formats may seem like an obvious choice for a tabular-oriented format, but there is no standardized form of CSV, instead numerous mutually incompatible dialects, using different delimiters, quoting strategies, etc., which makes parsing it error-prone. Furthermore, in order to include metadata about a given document and multiple tables for different kinds of data, you end up needing multiple files for each document.

SQLite is a great alternative to textual formats in some ways: The data can be more efficiently read and written to disk, and larger-than-memory data can also be processed easily. And with SQL, there is a powerful query language built-in. However, users often want to be able see their data in a text editor directly, as is possible with a text-based format, and they may not wish to use SQL.

While XML is more verbose and can be more effort to work with, it also has some clear advantages: XML is more stable, and there are more established standards for describing and verifying XML data.

Ultimately, JSON has advantages that seem to outweigh the strengths of XML. For JSON there is a high-quality package included in the Python standard library, which straightforwardly maps JSON values onto Python data structures. The JSON you see on disk is essentially equivalent to the Python data structures you get simply by loading that JSON data. The format's design, by relying on flat tables primarily, allows for a similarly immediately usable data structure in R using `jsonlite` or in Julia with `JSON.jl`. Furthermore, there are many great general-purpose JSON tools – `jq`, `jello`, `visidata`, etc. – that can also be used for working with Tabular JSON directly. In this way, users are free to choose the tools that work best in a given scenario, but this doesn't mean that they need to spend time and effort parsing text formats and fixing the attendant bugs.

Of course, as noted above, this tabular data model could be implemented in any of the formats mentioned above. In some situations, it may be preferable to implement this data model in one of those formats or some other format instead.

[3] Of course, JSON is, like XML, a text-based format, however it is a well-known and well-specified standard notation, for which there are myriad reliable parsers available. They can be reliably parsed in a way CSV, for instance, cannot.

3.4 Design principles

The need for software that translates between a comprehensible user-facing data model and one that is appropriate for serialization is avoided by making the serialization first-class: The serialization *is* the data model and is intended for direct interaction by users. We think of this as an *exterior-first* approach (Chu, 2023).

Different kinds of data are generally stored in separate tables, each with a set of keys appropriate to that kind of data. Each of these tables includes references, either to single tokens (= `token`) or to a range of tokens (= `begin` and `end`) in the main tokens table. In this way, some kinds of data are stored in a stand-off style.

The design aims to avoid some of the usability issues that accompany stand-off annotations by having these references be by index, thus enabling fast and easy retrieval of both single tokens and spans of tokens (via slicing). This also makes it easier to join the data in different tables using built-in operations in common data analysis packages. All of the index references use 1-based indexing, so they are directly usable in R, Julia, and Lua as-is, but some minor adjustments are required in Python. In general, any empty values are to be omitted, so they are distinguishable from empty strings and values such as `"_"`. The other strategy we employ is to only store some data in stand-off fashion: Annotations that apply to single tokens are simply included in the main tokens table.

Other advantages of stand-off representations are preserved. Different annotation layers can be easily added or removed without disturbing the others, and it is also possible, e.g., to have multiple instances of the same kind of annotation in order to store the annotations from different annotators in the same file.

3.5 Data layout

Each document in a corpus is represented by a single JSON object, which can either be stored as its own file or as a line in a JSON Lines file. This top-level contains, minimally, a `metadata` object and a `token` array:

```
{"id": "o9234f78",
 "metadata": { ... },
 "token": [ ... ], ... }
```

Whereas the metadata object is a collection of key–value pairs, adaptable to the needs of a given project, the token array is what we will call a "table":

`property`	Token annotations
`relation`	Relations between tokens
`span`	Spans over tokens
`set`	Sets of tokens
`spanset`	Sets of Spans over tokens
`hierset`	Hierarchical sets of spans over tokens

Figure 1: Annotation types.

an array of objects, which all have roughly the same set of keys. This is a row-based data representation that is readily translated directly into a data frame data structure by common data analysis packages.

Further annotations are included in tables under additional top-level properties and have forms that are determined by the type of data that these annotations represent. All of these additional tables refer to the main `token` table by means of indices. The format specifies a fixed set of possible annotation types (Figure 1), and each annotation type has a particular set of required keys.

Each document specifies the annotations it includes using the `annotations` key in the metadata object.

```
"metadata": {
  "annotations": {
    "lemma": {"type": "property"},
    "line2": {"type": "span",
     "description": "Secondary line ref."},
    ...
  }
},
```

The inclusion of this metadata tells users what top-level keys to expect and, due to the `type` value, what keys to expect in the associated tables. Additional keys besides `type` and `description` are allowed here, so there is a place for other useful information, e.g. provenance, etc. The specification provides for a set of standardized property names to be used for common annotations to aid in interoperability.

Token annotations. Some annotations, namely all those that apply strictly to single tokens, are included in the token table directly as `property` annotations. This includes such things as lemmas, normalized word forms, and POS tags.

```
{"id":"t2","form":"cats","lemma":"cat",
    "pos":"NOUN", ... }
```

A special case of token annotations is covered by the `object` annotation type. This annotation type

Name	Type	Desc.
form	property	Surface form
lemma	property	Lemma
pos	property	POS
dependency	relation	Dependencies
sentence	span	Sentences
coreference	spanset	Coreference

Figure 2: Some common annotations and their types.

describes a column in the tokens table that contains arbitrary JSON data. It is provided as an escape hatch, e.g. for representing sub-token-level data.

Relations between tokens. Dependencies are stored in a dependency table, whose rows are of the relation type, since dependencies are expressed as relations between tokens. Each row in this table has two references to the tokens table, from, in this case referring to the head token, and to, referring to the dependent token.

```
{"from":2,"to":1,"label":"det"}
```

Spans over tokens. Spans are always represented with a pair of properties, begin and end, which refer to the main token table. Sentences are represented as spans over tokens:

```
{"id":"s1","begin":1,"end":6,
    "label":"decl"}
```

Besides sentences, all layout information, such as that concerning page or paragraph or line boundaries, is represented as spans over tokens. This annotation type is also used for things like quotations and headings, where these are present.

Sets of spans over tokens. Entities of the spanset annotation type are a kind of span, so they contain begin and end properties, however in addition to this they also have an set property, which is the same for all members of a set. This annotation type is useful for representing things like coreferences:

```
{"set":"c1","begin":1,"end":2}
{"set":"c1","begin":4,"end":4}
```

Further information about the entity covered by such a span may be provided using the optional label property.

Hierarchical sets of spans over tokens. Each entry in a hierset-type table denotes a span of tokens, and so it has begin and end properties. In order to express hierarchical data structures, the entries must be able to refer to one another – these are all non-terminals. To this end, each entry must have an ID and a parent property, which specifies the node above it in the tree. This could be useful for representing consituency trees or discourse structure (note that the spans need not be limited to a single sentence or coincide with sentence boundaries). E.g.:

```
{"id":"c1","begin":1,"end":4,"label":"S"}
{"id":"c2","begin":1,"end":1,"label":"NP",
    "parent":"c1"}
{"id":"c3","begin":2,"end":4,"label":"VP",
    "parent":"c1"}
{"id":"c4","begin":3,"end":4,"label":"NP",
    "parent":"c3"}
```

4 An Example Use Case

In the course of a larger project there is often a need to convert data between various formats, and so conversion applications are written that tend to converge on a particular architecture: There are various reader and writer modules, some internal data model, and optionally a set of transformations that can be applied to that data model. There are existing applications for this purpose, such as Pepper,[4] why not use that?

One reason is that modules would need to be written in Java, and it could be that your team doesn't have expertise in Java or may not want to use it. (Note that this is not due to any issue with Java per se but could be the case for any particular implementation language.) Another reason is the internal data model, Salt, which, though very general and powerful, is not a representation that you would use for any other purpose, so that any modules you write are only useful in this Pepper context.

There are two main aspects of Tabular JSON, which make it useful in such a scenario: One is the exterior-first design and the other is the tabular data model.

What in most conversion scenarios is an internal data model is external in the case of Tabular JSON; the internal representation is identical to the serialization. This means that different parts of a conversion, annotation, and analysis pipeline need not know about one another. They can be developed independently from one another and could even use different programming languages altogether, yet all

[4] https://corpus-tools.org/pepper/

of these independent components have access to the same, complete underlying data structures. All of the various modules only need to know about Tabular JSON. This is the advantage of an exterior-first orientation.

The other important aspect is that this lingua franca is intended to be directly usable itself, since it would otherwise just be one more additional format to deal with. This is the motivation behind the tabular data model, which is geared towards the way data analysis frameworks treat data and is also a more intuitive way of representing data than graph-based representations, such as Salt.

5 Conclusion

There is a tension between, on the one hand, a general and powerful format that can represent adequately any kind of data, but which inevitably must depend on some software layer that can translate between this general format and a usable internal model, and on the other hand, a simple and lightweight yet limited format that may not be sufficient for many applications, forcing the invention of error-prone ad-hoc solutions.

The JSON-based format described in this paper is intended as a practical, lightweight format for linguistic applications that has minimal dependencies and is directly usable, because its on-disk form is essentially identical to a usable internal data structure. The use of a known set of data types allows users to reason about the data and work with it in this form directly – practically erasing the distinction between a serialized form and internal data model. This frees users from having to parse ad-hoc text-based formats or depend on particular specialized software.

The complete specification of the format and a JSON schema for validation are available at the project's public repository, accessible at this URL: `https://gitlab.rub.de/comphist/tabular-json`. See also Appendix A for a complete example document.

6 Limitations

The specification establishes a set of standardized property names for basic kinds of annotations, such as POS and lemmas, in order to aid interoperability. However the number of standardized names remains quite small currently, so that interoperability in practice is limited. Though we plan to expand this set in future iterations, some challenges remain: It is foreseeable that different projects may not agree on the naming scheme and prefer different names (e.g. `feats` vs. `infl` vs. `morph`, etc.) or that different projects may wish to model the same information in different ways, for instance, modelling named entities as token properties vs. spans. Further, one of the goals of this format is to enable the storage and use of novel and not yet established kinds of annotations, and it is impossible in principle to come up with property names for these things in advance.

Most of the different kinds of annotations that the format provides for are stored as stand-off annotations, which, while it has its advantages, is an impediment to human-readability. If one wants to know which tokens belong to a given sentence, say, one must follow the references from the sentences array back to the tokens array. We try and make this as simple and straightforward as possible by the use of indices that can be used to directly retrieve a single token or slice of the tokens array in such cases. However, what is simple programmatically isn't necessarily easy for humans, and this is one of the reasons that projects may prefer to model some things differently than in the specification.

7 Ethical Considerations

As this work presents a data format, I see its ethical dimensions being primarily those of freedom, fairness, and re-use: Users should not be required or 'nudged' to use particular proprietary software in order to work with a given data format, such as with, say, Excel files. Since it relies only on the simple and well-documented JSON standard, data in our format are usable from any programming language environment with no special dependencies, which make the data reusable and offers potential users a high degree of flexibility. The format should also be usable with any kind of language data, historical or modern, due to the use of UTF-8.

Acknowledgments

Thanks to the reviewers for their helpful feedback. Gefördert durch die Deutsche Forschungsgemeinschaft (DFG) – SFB 1475 – Projektnummer 441126958.

References

Sabine Buchholz and Erwin Marsi. 2006. CoNLL-X shared task on multilingual dependency parsing. In

Proceedings of the Tenth Conference on Computational Natural Language Learning (CoNLL-X), pages 149–164, New York City. Association for Computational Linguistics.

Christian Chiarcos, Stefanie Dipper, Michael Götze, Ulf Leser, Anke Lüdeling, Julia Ritz, and Manfred Stede. 2008. A flexible framework for integrating annotations from different tools and tag sets. In *Traitement Automatique des Langues, Volume 49, Numéro 2 : Plate-formes pour le traitement automatique des langues [Platforms for Natural Language Processing]*, pages 217–246, France. ATALA (Association pour le Traitement Automatique des Langues).

Andy Chu. 2023. Oils is exterior-first (code, text, and structured data). https://www.oilshell.org/blog/2023/06/ysh-design.html. Accessed 2024-05-04.

Stefanie Dipper. 2005. XML-based stand-off representation and exploitation of multi-level linguistic annotation. In *Proceedings of Berliner XML Tage 2005 (BXML 2005)*, pages 39–50, Berlin, Germany.

Stefanie Dipper and Michael Götze. 2005. Accessing heterogeneous linguistic data – generic XML-based representation and flexible visualization. In *Proceedings of the 2nd Language & Technology Conference: Human Language Technologies as a Challenge for Computer Science and Linguistics*, pages 206–210, Poznan, Poland.

Jan-Christoph Klie and Richard Eckart de Castilho. 2024. DKPro Cassis – reading and writing UIMA CAS files in Python.

Maarten van Gompel and Martin Reynaert. 2013. FoLiA: A practical XML format for linguistic annotation – a descriptive and comparative study. *Computational Linguistics in the Netherlands Journal*, 3:63–81.

Hadley Wickham. 2014. Tidy data. *Journal of Statistical Software*, 59(10):1–23.

Florian Zipser and Laurent Romary. 2010. A model oriented approach to the mapping of annotation formats using standards. In *Workshop on Language Resource and Language Technology Standards, LREC 2010*, La Valette, Malta.

A An Example Document


```
{
  "id": "doc1",
  "metadata": {
    "title": "Example document",
    "year": "2024",
    "version": "1.0",
    "annotations": {
      "pos": {
        "use": "pos_xpos"
      },
      "pos_xpos": {
        "type": "property",
        "model": "en_core_web_sm",
        "source": "Spacy"
      },
      "pos_upos": {
        "type": "property",
        "model": "en_core_web_sm",
        "source": "Spacy"
      },
      "lemma": {
        "type": "property",
        "description": "omitted when
            same as form",
        "model": "en_core_web_sm",
        "source": "Spacy"
      },
      "sentence": {
        "type": "span"
      },
      "line": {
        "type": "span"
      },
      "coreference": {
        "type": "spanset",
        "source": "ajr"
      },
      "dependency": {
        "type": "relation",
        "model": "en_core_web_sm",
        "source": "Spacy"
      }
    }
  },
  "token": [
    {
      "id": "t1",
      "form": "The",
      "lemma": "the",
      "pos_xpos": "DT",
      "pos_upos": "DET"
    },
    {
      "id": "t2",
      "form": "cats",
      "lemma": "cat",
      "pos_xpos": "NNS",
      "pos_upos": "NOUN"
    },
    {
      "id": "t3",
      "form": "slept",
      "lemma": "sleep",
      "pos_xpos": "VBD",
      "pos_upos": "VERB"
    },
    {
      "id": "t4",
      "form": ",",
      "pos_xpos": ",",
      "pos_upos": "PUNCT"
    },
    {
      "id": "t5",
      "form": "and",
      "pos_xpos": "CC",
      "pos_upos": "CCONJ"
    },
    {
      "id": "t6",
      "form": "they",
      "pos_xpos": "PRP",
      "pos_upos": "PRON"
```

```
    },
    {
      "id": "t7",
      "form": "purred",
      "lemma": "purr",
      "pos_xpos": "VBD",
      "pos_upos": "VERB"
    },
    {
      "id": "t8",
      "form": "softly",
      "pos_xpos": "RB",
      "pos_upos": "ADV"
    },
    {
      "id": "t9",
      "form": ".",
      "pos_xpos": ".",
      "pos_upos": "PUNCT"
    }
  ],
  "coreference": [
    {
      "set": "c1",
      "begin": 1,
      "end": 2
    },
    {
      "set": "c1",
      "begin": 6,
      "end": 6
    }
  ],
  "line": [
    {
      "id": "l1",
      "begin": 1,
      "end": 4
    },
    {
      "id": "l2",
      "begin": 5,
      "end": 9
    }
  ],
  "sentence": [
    {
      "id": "s1",
      "begin": 1,
      "end": 9
    }
  ],
  "dependency": [
    {
      "id": "dep1",
      "from": 2,
      "to": 1,
      "label": "det"
    },
    {
      "id": "dep2",
      "from": 3,
      "to": 2,
      "label": "nsubj"
    },
    {
      "id": "dep3",
      "from": 3,
      "to": 3,
      "label": "root"
    },
    {
      "id": "dep4",
      "from": 3,
      "to": 4,
      "label": "punct"
    },
    {
      "id": "dep5",
      "from": 3,
      "to": 5,
      "label": "cc"
    },
    {
      "id": "dep6",
      "from": 7,
      "to": 6,
      "label": "nsubj"
    },
    {
      "id": "dep7",
      "from": 3,
      "to": 7,
      "label": "conj"
    },
    {
      "id": "dep8",
      "from": 7,
      "to": 8,
      "label": "advmod"
    },
    {
      "id": "dep9",
      "from": 7,
      "to": 9,
      "label": "punct"
    }
  ]
}
```