Data Structures in Lexicography

PHD THESIS DRAFT 2024-02-29

Michal Měchura

Abstract

Dictionaries are reference tools which humans turn to in order to satisfy their information needs with respect to the vocabulary of a natural language. This thesis takes a critical look at the formal methods used in modern digital lexicography for representing the structure of dictionaries in software. We do this on two levels: data languages and data models.

At the level of data languages, the thesis identifies several problems with how XML is used for encoding dictionaries. We analyse how dictionary content is fundamentally different from other kinds of text and we argue that XML is not a good fit for the requirements of lexicography. Crucially, we observe that other data languages such as JSON and YAML do not meet those requirements either. We propose an alternative language called Name-Value Hierarchy (NVH) which represents dictionary content more efficiently.

At the level of data models, the thesis critiques the over-reliance on tree structures in lexicography. Tree structures, while easy to serialise in languages such as XML, do not provide a satisfactory representation for some phenomena that occur in dictionaries, such as entry-to-entry cross-references. We propose an alternative, partially graph-based data model called DMLex which is currently in the process of being standardised by OASIS.

This thesis can be seen as a contribution to the digitisation of lexicography. Currently, digital lexicography is mainly about text encoding: a shallow, document-oriented form of digitisation. This thesis shows the way towards a more deeply digitised, data-centric lexicography of the future.

Prologue

My long career in computational lexicography,[1] all those years when I have been providing IT support and building IT infrastructures for various dictionary projects, has given me an opportunity to think criticially about the data structures we use for representing lexicographic content. This thesis is the outcome of that critical thinking.

A good PhD thesis is one which identifies a problem and proposes a solution. This happens twice in this thesis.

- The first ocassion is in **Block II 'Data Languages'** where I identify several problems with how XML and other languages are used for encoding lexicographic content, and I propose a new formal language called NVH (*Name-Value Hierarchy*) which fits the needs of lexicography better.

- The second ocassion is in **Block III 'Data Modelling'** where I identify a number of problems with the tree-based data models that are commonly used for representing dictionaries, and I propose an alternative data model based on a hybrid between tree structures and graph structures. This proposed data model is called DMLex (*Data Model for Lexicography*) and is now on its way to becoming an official OASIS specification.

Preceding the two blocks, **Block I 'Context'** serves as an introduction: it defines the kind of lexicography that this thesis is about, and sets it in the context of broader digitisation tendencies that are in progress in the discipline at present.

[1] Sometimes also called "e-lexicography". This term appeared sometime in the 2010s but seems to be falling out of use now, no doubt because all lexicography is "e-" now.

Acknowledgements

I am grateful to friends and colleagues in the NLP Centre[1] and in Lexical Computing[2] for encouraging me to develop the ideas that eventually made it into this thesis, in particular X, Y and Z. Lexical Computing has supported my research financially over the years: the company has a history of sponsoring academic research in disciplines relevant to it and I have been able to benefit from that. Lexical Computing has also kindly taken over the development of *Lexonomy*, an open-source dictionary writing system I had created in 2016, after it became clear that Lexonomy would not be a topic in my thesis and that I did not have the capacity to be the project's maintainer. One of the two main contributions of this thesis, the formal language known as NVH, is a direct result of my work with Lexical Computing, and has now been adopted by the company internally for all its lexicographic work.

An equal amount of gratitude goes to friends and colleagues in the wider international lexicographic community whom I have been meeting and exchanging ideas with at conferences such as eLex and the EURALEX congresses, and in Europe-wide projects such as ENeL (*European Network for eLexicography*) and ELEXIS (*European Lexicographic Infrastructure*). Most of this thesis can be traced back to talks and papers I had presented at these forums over the years. In particular, I am grateful to my fellow members in LEXIDMA (*Lexicographic Data Models and API*), a technical committee in OASIS: X, Y and Z. Together, we have developed DMLex (*Data Model for Lexicography*), the second of the two main contributions of this thesis.

---

[1] Centre for Natural Language Processing at Masaryk University's Faculty of Informatics.

[2] Lexical Computing is one of the campus companies hosted by the Faculty of Informatics. It is known mainly for the *Sketch Engine* corpus query system.

Publications

This is a structured and commented list of publications I have authored and talks I have given that are relevant to this thesis. This list can be understood as a timeline of sorts for the ideas that eventually culminated in this thesis.

On the digital transformation in lexicography

TBD

On digital dictionary publishing

TBD

On dictionary writing systems and the dictionary-making process

TBD

On XML and other data languages in lexicography

TBD

On lexicographic data models

TBD

BLOCK I: CONTEXT

1 Introduction to lexicography and its data structures

This chapter will define the object of the thesis: the domain of *human-oriented lexicography*. We will do this from two angles. First, we will set human-oriented lexicography apart from other kinds of lexicographic activities in natural language processing (NLP) and in language technology. Second, we will go on a short guided tour through the types of content that are usually found in human-oriented dictionaries: entries, senses, definitions and many others. This will be a introduction to the kinds of objects we will be attempting to represent computationally in the rest of this thesis.

1.1 What is a dictionary?

The typical product of lexicography is a *dictionary*. A dictionary can take the form of a printed book or, more likely these days, a digital artefact presented to users on the screens of their computers and mobile phones. This section will clarify how human-oriented dictionaries differ from other digital artefacts that provide information about words such as NLP-style "language resources", wordnets, framenets and word embeddings.

1.1.1 Dictionaries versus "language resources"

The term lexicography is used in this thesis to refer to the dicipline which produces dictionaries and other language-related reference works *intended for consumption by human users*. The emphasis on human users is important here. This thesis is not about machine-understandable language resources such as semanic networks, frame lexicons and morphological databases. The discipline which produces such datasets is sometimes called lexicography too but it is a different offshoot of lexicography which this thesis will not deal with.

Why is it important to dinstinguish between dictionaries for humans on the one hand and language data for machines on the other? After all, both disciplines have roughly the same goal: to organize information about the

words of one language or another, and to make that information available to whoever needs it. Why does it matter, then, whether the consumer is a human person or a computer application? Your initial intuition may be that once a dictionary originally written for humans is encoded on a computer, it will immediately become useful for applications in natural language processing such as word-sense disambiguation and machine translation. Conversely, you might think that a computational lexicon such as WordNet or FrameNet should be useful for humans to look up words in. True, such crossovers are not completely unheard of. Dictionary content originally produced for humans *is* ocassionally found useful for other applications: a classical example is the grammatical coding system in *Longman Dictionary of Contemporary English* (published in 1978) which found extensive use in natural language processing in the 1980s (Boguraev and Briscoe 1987). And perhaps there are people who do satisfy their linguistic information needs by looking words up in WordNet and FrameNet instead a *Collins*, a *Merriam-Webster* or a Wiktionary. But such synergies are rare. On the whole, the two camps are separate: there are dictionaries for humans, there are lexical resources for machines, and there is little overlap between them.

This is not just an artefact of ignorance, of factions failing to talk to each other. There are deeper reasons for this. Human-oriented dictionaries tend to communicate facts about language in ways which, while not explicit enough for computer applications, are optimized for human cognitive abilities. Computational lexicons, on the other hand, tend to represent lexical knowledge in ways which can be rather alien to humans, especially humans who are not computational linguists. And this divide, if anything, is getting wider with the arrival of machine-learned language models such as word embeddings: these represent lexical knowledge in ways even more alien and more inaccessible to inspection than "classical" hand-crafted, WordNet-style resources.

In fact, the structure of *any* machine-oriented lexical resource has almost nothing in common with the structure (both micro- and macro-) of a typical human-oriented dictionary. For this reason, it is valid to study the data structures of human-oriented dictionaries separately from the data structures present in computational lexicons. The purpose of this thesis is to study the former without getting distracted by the latter.

1.1.2 Dictionaries versus "lexical databases"

Dictionaries do not simply list off all facts that can be known about a word. Good dictionary authors are careful about what they include and exclude in a dictionary entry, depending on the dictionary's purpose and target audience: whether it is a dictionary for decoding or encoding, a dictionary for native speakers or for learners, if learners then which level, and so on. The principle of *selection of information* is important in dictionary production, it is how lexicographers respond to the intended *lexicographic function* of the dictionary (in the sense defined by Tarp 2008) and to the perceived *information needs* (in the information-scientific sense of the term) of its end-users.

In contrast to this stands the concept of *lexical* (or *lexicographic*, or *lexicographical*) *database*: a "structured resource that contains as much lexicographic information as possible regarding words and lexical units in a language" (Horák and Rambousek 2018, p. 185). These databases are not dictionaries. They are something from which a dictionary may be derived later, either automatically by setting a few parameters, or manually by human lexicographers. Examples of such databases include Cornetto (Dutch: TBD ref), DANTE (English: Convery et al. 2010, Rundell and Atkins 2011) and EKILex (Estonian: TVD: ref). *Wikidata Lexemes* can also be undertood as such a database, this one being multilingual. Unlike a dictionary, a lexical database is there to meet the information needs of the *lexicographers*, not the dictionary end-users. A lexical database is neutral in terms of its lexicographic function: it can fulfill multiple functions depending on what the lexicographer decides to include or exclude.[1]

Are the structures of lexical databases within the scope of this thesis? That depends. Some lexical databases, such as Cornetto, are organized in structures reminiscent of machine-oriented WordNet-like resources where the main organizing principle is a *network* of lexical units. These databases are relatively far away (structurally speaking) from the eventual dictionary that may be derived from them. For that reason, they are out of scope here.

[1] The role of a function-agnostic database in the dictionary production process is detailed in section 4.2.2 of Atkins and Rundell (2008). For a critical discussion of the concept see Bergenholtz and Nielsen (2013).

Then there are lexical databases such as DANTE whose data structure is very similar to that of a finished dictionary: the primary organizing principle is a collection of *entries* subdivided into *senses*, with very little explicit networking between them. A good way to think about a lexical database like DANTE this that it is a semi-finished human-oriented dictionary, an "almost-dictionary" which needs some amount of editing (but no structural transformation) before it is ready to start fulfilling some lexicographic function and meeting the information needs of end-users. The structures of such databases are very much within the scope of this thesis.

1.2 What is inside a dictionary?

Now that we have defined which kinds of dictionaries are in scope for this thesis, let us take a look inside them: let us look at the kinds of content typically found in dictionaries. This section will be useful to readers who do not have a background in lexicography, as an introduction to the *domain* we will be modelling in the remainder of this thesis.

Figure 1-1 A typical dictionary entry

1.2.1 Entries and headwords

A dictionary is a collection of **entries**. Dictionary entries are not continuous text (like paragraphs in a novel or encyclopedia). The style of the language inside them tends to be "telegraphic", the only items that consist of full-formed sentences are usually the examples (and sometimes the definitions, for that see below). They are highly structured, they resemble multi-level bulleted lists rather than continuous text, and they are rarely read from beginning to end in their entirety: a typical user, once they have located the entry which they believe will satisfy their information need, will briefly scan the entry before eventually zooming in on the part that interests them.

In printed dictionaries the entries have traditionally been arranged alphabetically by the entries' headwords, in electronic dictionaries the entries are stored in some form of database and brought on the user's screen in response to what the user has searched for. The principle by which entries

are organized in a dictionary is called the dictionary's *macrostructure*, while the internal structure of the entries themselves is the dictionary's *microstructure*.

Each entry typically begins with a **headword**. The rest of the entry describes the headword, typically by giving a numbered list of the headword's **senses**. The way a dictionary approaches the lexicon of a language can therefore be described as *semasiological*: it starts from a word and asks, which meanings does this word have? (The opposite would be an *onomasiologial* approach: starting from a defined meaning or concept and asking, which words express this meaning? This is the approach taken in terminology studies and also in various wordnets and framenets).

Most of the time, the lexicographic concept of *headword* is identical to what a lexicologist or a computational linguist might call a *lemma*: a canonical word form, such as the nominative singular of a noun or the infinitive of a verb, which represents a possibly very large set of inflected word forms (all the cases of a noun, all the tenses of a verb). But not all headwords are lemmas. Sometimes lexicographers decide to give headword status to individual word forms if they believe that this will be useful for the user. An example would be Czech *vlas* 'a hair' whose plural *vlasy* has meanings that are partially unpredictable: not just 'hairs' but also 'headhair'. In such a situation the lexicographer may well decide that *vlasy* needs its own entry. In addition to that, sub-word units such as suffixes and prefixes are often treated as headwords too, and in recent years we see a tendency to treat multi-word items, such as various idioms and set phrases, as headwords as well.[1]

The headword is often followed by one or more of the following items:

- **Grammatical labels** which indicate its part of speech (= whether it is a noun, a verb, an adjective etc.) and other grammatical properties such as noun gender or verbal aspect.

[1] At that point *headword* becomes a bit of a misnomer: the term implies a single-word expression whereas modern dictionaries may contain entries headed by multi-word "headwords".

- Various **inflected forms** of the headword: the plural of nouns, the past tense of verbs and so on. In richly inflected languages dictionaries do not usually list *all* possible inflected forms of the headword because such lists would be too long and distracting (for example, for a Czech noun that would be up to 14 word forms: seven grammatical cases in the singular and seven again in the plural). Lexicographers normally choose a handful of representative forms to indicate to the user how the word inflects in broad terms.

- An indication of the headword's **pronunciation**. In printed dictionaries this is often given as a transcription in (some variant of) the International Phonetic Alphabet (IPA) or in some kind of simplified "phonetic spelling". In digital dictionaries pronunciation is increasingly being offered as sound recordings which the user can listen to.

1.2.2 Senses

The largest part of a dictionary entry is usually taken up by an ordered list of senses. Each sense represents one of the possibly many meanings of the headword. The senses are typically presented as a bulleted or numbered list. In printed dictionaries the senses were often all presented inline, without line breaks between them, to save space. Modern on-screen dictionaries are usualy more generous in their use of whitespace and the senses are formatted as visually separated block-level elements with generous whitespace between them.

The senses of a lexicographer are not necessarily the same thing as the senses of a lexicologist or a cognitive linguist. In disciplines where researchers are interested in how a person's mental lexicon is organised, it is seen as important do distinguish between situations when two "readings" of a word constitute separate senses and when not. For example, the two readings of *bank* (a financial bank versus the bank of a river) would be considered separate senses (because you can never mean both at the same time, not even in an ambiguous sentence like *we finally reached the bank*) but the two readigs of the financial sense (an institution versus a building) woi;d not be considered separate senses (because you can mean both at the same time, as in *I work in a bank*) – these two readings would more likely be described by a

lexicologist as *facets*, *microsenses* or *ways-of-seeing* of the same sense. But most of this is only of limited relevance to a lexicographer's job. A lexicographer may well decide to present facets to the dictionary user as seperate senses (or subsenses), or to merge two senses into one, if he or she believes that this will be useful to the end-user (= that it will satisfy the user's information needs). This illustrates an important point about human-oriented lexicography in contrast to lexicology: a lexicographer's mission is not necessarily to express facts which are true with respect to some linguistic theory but to present those facts in ways which are going to be useful, helpful, relevant and easy to understand for a human dictionary user. So: the term *sense* does not have much of any deep theoretical meaning in lexicography, it is just a unit of organising information: basically nothing more than a synonym for "an item on a list":[1] just as a *headword* is not the same thing as a *lemma.*

The order in which the senses are presented usually matters: lexicographers often deliberately arrange senses accordng to some principle such as frequency of use (most common senses first), chronology (historically oldest – or newest – senses first), a percieved "basicness" (literal senses first, metaphorical extensions after), age of acquision (in children's dictionaries) or learner level (eg. according to CEFR[2]).

High-frequency headwords typically have a lot of senses while low-frequency headwords often only have one. In traditional printed dictionaries, when an entry contains only a single sense, if often happens that the sense is not marked up in any obvious way from the rest of the entry. In modern born-digital dictinaries, the senses of an entry are always marked up explicitly (typically using some XML element such as `<sense>`), even if there is only one of them.

---

[1] In lexicology, the superordinate term for both *sense* and *facet* is *reading*. In German lexicographic literature the term for *sense* is sometimes the word *Leseart* 'a way of reading'. This is – from a lexicologist's point o view – perhaps a more adequate term for the lexicographic "item on a list".

[2] *Common European Framework of Reference*, a popular system for ranking the skills of second-language learners in levels ranging from A1 (absolute beginner) to C2 (native-like fluency).

A sense typically consists of items such as the following:

- A **definition**: a sentence which explains the meaning of that sense. Definitions are usually in the same language as the headword. Different dictionaries have different *defining styles*. Some definitions are formal and written strictly in such a style that one could almost replace the headword with them ("collide: to hit something violently"[1]) while others use a more chatty, full-sentence style which re-uses the headword ("if two or more moving people or objects collide, they crash into one another"[2]). The latter is popular in pedagogical dictionaries for second-language learners and some authors prefer calling such definitons *explanations* rather than *definitions*. In addition to (or instead of) definitions/explanations, senses sometimes start with short "mini-definitions" (called variously indicators, signposts or glosses) whose purpose is to help the user locate the desired sense quickly when scanning a long entry visually.

- **Translations** of the headword if the dictionary is bilingual. Some translations are simple and straighforward equivalents of the headword, but in complicated cases, when the target language does not have a straighforward equivalent, these can be *explanatory translations* which read almost like definitions, but in the target language. Translations can be just strings of text, or they can be annotated with additional information such as grammatical labels, transcriptions of pronunciation and so on, depending on the purpose of the dictionary.

- **Example sentences** which show how the headword is used in context. Lexicographers include example sentences in a dictionary for many diffrent purposes. In a pedagogical dictionary for second-language learners, the purposde of examples is to show models of good usage which the learner would do well to follow. In dictionaries for native speakers the purpose of examples may be less to serve as a model of good usage and more to clarify the sense, together with the definition. And in historical dictionaries the examples may be there primarily to attest, to

[1] From Cambridge Advanced Learner's Dictionary & Thesaurus

[2] From Collins COBUILD Advanced Learner's Dictionary

prove that the sense exists or existed. Example sentences may or may not come with source attributions, may or may not have sound recordings of pronunciation, and may or may not be translated.

- **Collocates** of the headword, that is, words that often accompany the headword in real-world language use: adjectives that often modify the noun, verbs that often have this noun as subject or object, and so on. Traditionally, it has *not* been very common for dictionaries to have collocates (unless it was a specialized collocations dictionary) but it has been becoming more common in recent decades, mainly because collocates are relatively easy to extract from corpuses and because they are often the basis on which senses are identified.

- **Cross-reference** to other (senses of other) entries in the same dictionary, such as links to synonyms and antonyms.

The purpose of all this sense-level content is to describe meaning: the semantics and the pragmatics of the headword. In contast, the purpose of entry-level content such as part-of-speech labels and pronunciation transcriptions is to describe the formal properties of the headword: its morphology, morphosyntax, phonology and orthography. An ideal which lexicographers sometimes aspire to is to have a clear division between formal and semantic properties of the headword: the formal properties belong at the entry level and are understood to be shared by all the senses, while the semantic properties belong at the sense level and are specific to each individual sense. In practice, this ideal is sometimes relaxed, and senses sometimes contain formal information too. For example, if a noun has two different plural forms and these are associated with two separate senses, then the plurals may be given at the level of the senses, rather than at the level of the entry. Later in this thesis, in Chapter 7 when we go on to describe the DMLex data model, we will see that it *is* possible to handle such situations without giving up on the ideal of strict separation between formal and semantic properties.

Senses are often arranged in a flat list without hierarchy, but in some more complex dictionaries there may be a hierarchical list of senses and subsenses. The entry structure in such dictionaries can be understood as recursive:

senses can contain other senses. In Chapter 6 we will discuss recursion in more detail.

1.3 Making dictionaries machine-readable

Dictionaries are a very specific text type. Representing them on computers – which is something people have been doing since the 1980s if not earlier – is a different task from text encoding in the likes of HTML, LaTeX or DocBook. In a text encoding scenario, the typical goal is to mark up things like paragraphs and itemized lists, as well as inline structures such as stretches of "strong" "emphasized" text (≈ bold and italic). Digital lexicography aims at a higher level of abstraction, the goal is to represent the domain-specific content types discussed above, such as senses and definitions, as well relationships between them. Blocks II and III of this thesis deals with the various challenges that come up when attempting to do that.

2 A short history of digitisation in lexicography

The preceding chapter has introduced human-oriented lexicography as the domain of interest for this thesis. Data modelling is a *digitisation* activity: it is one of the things we do to migrate a domain into the digital medium. In this chapter, we will review to what extent lexicography has become digitised in the last few decades. This should set the rest of this thesis in its broad context.

For most of their history, dictionaries have existed as printed books. Today, however, the popular image of "the dictionary" as a book is outdated and hugely out of sync with how lexicography is actually done. Today's lexicography is a discipline where everything happens on computers, either fully automatically or in interaction with humans: this applies to how dictionaries are made (using corpus query software and dictionary writing systems) as well to how dictionaries are delivered to end-users (as websites and mobile apps). The printed dictionary market has shrunk to a shadow of its former self while online dictionaries rule the day. Most dictionary projects today are designed as digital-only, with no printed output planned. Like many other disciplines, lexicography is going through a digital transformation. The purpose of this chapter is to clarify how far advanced we are in this transformation and how much of it is still ahead of us.

The process of making and delivering a dictionary is something which unfolds in stages. The first stage is when we are *discovering* facts about words, these days typically from a corpus. The second stage is when we are *organising* these facts into the form of dictionary entries. The final stage is when we are *delivering* dictionaries to human users on their screens or (rarely) on printed pages. In this chapter I will argue that not all stages have been digitised equally. Although the first stage – *discovery* – has been digitised throroughly and in some sense "completely", the remaining two stages – *organisation* and *delivery* – have only been digitised rather superficially so far and there is untapped potential in them yet.

2.1 From citation slips to corpus query systems

To say something about a word, the lexiographer must *know* something about it first. Pre-digital lexicographers relied on their introspection and their own subjective judgment to produce lists of the meanings a word has, to compose example sentences, and so on. From the 19th century onwards this started becoming more objective and empirical with the introduction of citation slips and various reading programmes (Atkins and Rundell 2008, section 3.2). And, from late 20th century onwards, these analog tools have started being replaced by methods from corpus linguistics and from natural language processing.

The use of corpuses and computational methods for lexicography was pioneered in the 1980s by the now legendary COBUILD project (Sinclair 1987). Today, putting NLP at the service of lexicography – for the purposes of *knowledge acquisition* – is a well-established research programme (Horák and Rambousek 2018, section 12.3.1). Computational methods have given lexicographers previously unheard-of superpowers such as automatic discovery of collocations based on various statistical measures, automatic word-sense discovery through clustering of collocates, automatic discovery of synonyms, antonyms and other semantic or paradigmatic relations, and even finding "good" dictionary examples based on heuristics such as "prefer short sentences with simple words in them". Corpus-based lexicography is now the standard, practically all dictionary projects begin by deciding which corpus to work from. The process of compiling a dictionary entry almost always begins with using a corpus query system such as Sketch Engine (Kilgarriff et al. 2004, Kilgarriff et al. 2014) to discover facts about the headword.

The corpus turn in lexicography has introduced three major categories of innovations. **Firstly,** they enabled the existence of superhumanly large corpuses which have been unachievable with analog tols: in other disciplines such large datasets are called *Big Data*. **Secondly,** they brought statistical methods that can be used to analyze these corpora more objectively than a human lexicographer could, and at the same time bring to light knowledge that a human person might not even notice. **Thirdly,** new models of human-computer interaction have emerged, concepts such as *keyword in context* and

word sketch, which allow the human lexicographer to take note of the outputs of the corpus methods and understand them.

We can therefore say that the initial stage of the entire lexicographic process – *knowledge acquisition* – has already been digitised so thoroughly and so deeply that we have actually redefined it into something quite different from what it was in pre-digital times. Today's corpus tools are not just better versions of paper-based citation slips and reading programmes: they are qualitatively different, delivering results that would have been unachievable without them. The NLP methods used for extracting knowledge from corpuses will certainly continue to improve incrementally, but it seems that no major new innovations are likely to emerge in this area: the potential offered by the digital medium has been exploited more or less fully here.

As a parallel trend, we sometimes see efforts to complemet corpus data with insights from research on how people use online dictionaries (Lew and de Schryver 2014), especially search log analysis (De Schryver et al. 2006), and with insights from studies in psycholinguistics and language acquisition such as word prevalence and age of acquisition (Lew and Wolfer 2024). These play a role mainly in deciding which headwords to include in a dictionary and in prioritizing which headwords the lexicographers should process first. For everything else, “corpus is king” and will probably remain on its throne for the foreseable future.

2.2 Rise of the robot lexicographers

The lexicographer’ job is to “translate” knowledge from the corpus into the form of a dictionary entry that is going to be comprehensible and useful to the intended end-users. Until recently, the everyday reality for working lexicographers has been to do the “translating” manually: on one screen, the lexicographer watches the results of the corpus analysis (using a corpus query system such as Sketch Engine) and, on another screen, he or she compiles the dictionary entry by typing, copying and pasting short pieces of text into a prepared structure in a dictionary writing system such as Lexonomy. All knowledge from the corpus passes through the mind and fingers of the person in front of the keyboard before it becomes a dictionary. Human minds and

fingers are the bottlenecks of the lexicographic process – they are what makes dictionary projects take so long and cost so much money – so it is no wonder that there is a push towards automation here.

Automation in this area began modestly over a decade ago in the form of ergonomic improvements such as *tickbox lexicography* (Kilgarriff et al. 2010) in Sketch Engine (which makes it possible to batch-copy content from the corpus tool into the dictionary) and *content pulling* (Jakubíček et al. 2018) in Lexonomy (which allows the lexicographer to "pull" content from the corpus into the dictionary entry on request).

More recently, we have been seeing more radical attempts at automation, when people are experimenting with the automatic generation of entire dictionary entries and entire dictionaries, either "at once" (the *One-Click Dictionary* method, Jakubíček et al. 2018, in Sketch Engine which makes it possible to generate an entire proto-dictionary from the corpus), or "gradually" when dictionaries are generated step by step in interaction with a human editor (the *Million-Click Dictionary* method, Jakubíček et al. 2021). Experience so far shows that it is possible to make the lexicographic process go faster and cheaper this way (Baisa et al. 2019), importantly *without* having to accept inconvenient trade-offs affecting the quality of the resulting dictionary.

However, nothing is without consequences. Here, like everywhere else, increased automation forces a certain redefinition of what we are actually doing. **Firstly,** the role of the lexicographer is shifting from the role of a "driver" of the entire process to the role of a post-editor, someone who only corrects the computer's mistakes and intervenes where the computer does not know how to proceed. This transformation, which is only just beginning in lexicography, is already far advanced in other language-related disciplines, for example in translation (the role of the translator is changing to the role of a machine translation post-editor) and in copywriting (the role of people who produce marketing texts, such as product descriptions in online shopping, is changing to the role of authors of templates from which machines then generate finished texts). **Secondly,** there is a tendency to simplify the structure of dictionary entries. Dictionary entries generated by automatic methods tend to have a flatter structure (without a complex hierarchy of

senses and subsenses) and contain a narrower repertoire of content types than dictionaries compiled by humans. In other words, automatically generated dictionaries are shaped by what *can* be obtained from the corpus rather than what lexicographers ideally *want* to have in a dictionary. Whether this bothers the end-users and whether they even notice is an open question.

All these trends are relatively new and are far from established practice yet. Some dictionary makers are experimenting with these while are others are not even aware of them yet. An additional, recently emerged trend which is even further from everyday practice yet, is using generative AI to automate certain lexicographic tasks (De Schryver 2023), especially those that have previously resisted automation, such as definition writing. Lexicographic definitions are notoriously difficult to extract from corpuses (Kovář et al. 2016, Stará 2019) because authentic non-dictionary texts do not normally contain sentences of that type ("a window is a space usually filled with glass in the wall of a building"). It seems that, with clever prompting and clever use of few-shot learning techniques, large language models such as ChatGPT are able to generate dictionary-style definitions to a standard comparable to those written by human lexicographers (Lew 2023).

The summary is that this particular stage of the lexicographic process – the stage when lexicological knowledge is being converted into lexicographic content – is currently undergoing rapid innovation and is being subject to a strong push towards automation. This will probably force a redefinition and renegotiation of the roles of humans and machines in the entire process.

2.3 Dictionary writing systems and what is inside them

The classical data structure for lexicographic content is an *entry*. Section 1.2 has given an introduction to the types of content that dictionary entries usually contain. Lexicographers typically use specialised software, a *dictionary writing system*, for editing a dictionary. Current widely used dictionary writing

systems are the IDM Dictionary Production System,[1] TLex,[2] iLex (Erlandsen 2010) and Lexonomy[3] (Měchura 2017, Rambousek et al. 2021). For a comprehensive recent review of dictionary writing systems see Abel 2022.

Using specialised dictionary software is commonplace on dictionary projects today. At first glance, it might seem that this is another example of deep and thorough digitization: nobody seriously considers writing a dictionary in an ordinary word processor any more. But, in the rest of this thesis, I will argue that we have not yet exhausted all the potential that the digital medium offers. Practically all current dictionary writing systems represent dictionary entries as isolated tree structures, usually encoded in XML – the software is not much more than a glorified XML editor. Block III of this thesis is a critique of this position: I argue there that keeping dictionaries in a purely tree-structured data model imposes certain inconvenient limits and causes problems which could be solved by re-engineering dictionaries into a more flexible, partially graph-based data model.

Concrete proposals for how to do this are layed out in Block III, therefore we will not delve deeper into the topic at this stage. Let it just be said at this point that, from a data-modelling perspective, lexicography has only undergone a rather shallow form of digitisation so far, and that there is much to be done yet.

2.4 The future of human-dictionary interaction

When dictionaries migrated from the pages of books onto computer and phone screens in the last two decades, it was a big change for the better for the end user. The main improvement is that it made searching faster and easier. In paper dictionaries, the user had no choice but to be his or her own search engine: people searched by turning the pages with their fingers, navigating alphabetically. This is a process which takes time and puts a cognitive load on the person: your attention is distracted from whatever

[1] `https://dps.cw.idm.fr/index.html?the_entry_editor.htm`

[2] `https://tshwanedje.com/tshwanelex/`

[3] `https://www.lexonomy.eu/`

you were doing before, such as reading or writing, by having to search the dictionary. Computers have allowed us to take this cognitive burden off ourselves and outsource it onto a machine. This has made it much easier for people to use dictionaries.[1]

Some people are still deeply fascinated by this innovation. But for most computer users today, many of whom are digital natives, this evolutionary step is something that happened a long time ago. Digital dictionaries are the normal state of affairs. It is now time to start asking what the next evolutionary step in human-dictionary interaction will be.

One emerging user requirement is **aggregation**: people increasingly express a desire to search many dictionaries at once. The current situation is that, in each language and in each language pair, users usually have a choice of multiple online dictionaries and dictionary-like products which may or may not satisfy their current information need: the user has to visit each website individually to see if it has the information the user is looking for. This can be an arduous slog around the Internet: every dictionary website is a little different, some are user-friendly and ergonomic, others not so much, you have to know them, know their addresses, know the strengths and weaknesses of their search algorithms. It is a large cognitive load. Can it be automated?

One strategy is to use a generic search engine like Google – but generic search engines often misconstrue a lexicographic enquiry for an encyclopedic one: "tell me about the word *cat*" versus "tell me about cats". Another option is to use one of the few existing dictionary-specific meta-search engines and aggregators such as OneLook[2] and the European Dictionary Portal[3] (Měchura 2017) – but their problem often is that they do not cover the languages, language pairs or individual dictionaries the user wants.

---

[1] Anecdotal evidence suggests – even if there is no data to prove it – that people consult dictionaries more often today, in the digital era, than they used to back when all dictionaries were on paper. This must be because dictionaries are easier to use now: human nature dictates that the easier something is, the more are people likely to do it.

[2] `https://onelook.com/`

[3] `http://www.dictionaryportal.eu/en/`

The road to a better aggregation of dictionary websites is currently blocked by several obstacles. One obstacle is the absence of widely respected standards for exchanging dictionary metadata on the Internet: a machine-readable vocabulary which any dictionary website could use to tell the world about the headwords it contains, in which languages they are, and so on. This is a technical hurdle. The second obstacle is more human: publishers tend to be reluctant to make their content available to third parties. Most organisations that publish online dictionaries today prefer to do so on their own websites, under their own logos, with their own identities. This is understandable for commercial publishers, but non-commercial and academic institution have this tencency too. In spite of these handicaps, some form of aggregation on tomorrow's "Internet of dictionaries" is probably unavoidable. For once, it is what users want and, secondly, it is already happening in other information discpllines, mainly in libraries and in scientific publications: open metadata, all kinds of portals and metasearch engines are already commonplace there today.

A second emerging trend is for dictionaries to become integrated into other tools, even to such an extent that the dictionary becomes **invisible** (Medveď et al. 2023). The motivation is again to minimise the cognitive load associated with consulting a dictionary. A dictionary is something people use when they are doing something else, typically reading or writing. While reading or writing, an information need may emerge in the reader's or writer's mind, a need which must be satisfied before the user can or wants to continue: perhaps because he or she does not understand a phrase or is not sure how best to express an idea. This is when people decide to go to a dictionary, but this comes at the cost of becoming distrated and perhaps losing track of what you were doing before.

This is why we are beginning to see experiments with digital tools which eliminate the need to "go" anywhere at all: the user can satisfy his or her information needs right there in the current context, without having to – for example – switch to a different browser tab or window. An example is the experimental tool ColloCaid (Lew et al. 2018) which, while writing in a second language, suggests typical collocations on the spot, without the need to go anywhere or search for anything. Writing tools are not a new genre by any means (spellcheckers and grammar checkers have existed for decades), what

is new is the fusion between them and subsets of what would traditionally be called "lexicography" (in ColloCaid's case, the cataloguing of collocations). While writing tools are a well-known genre, "reading tools" are not and that is perhaps why nobody has built a hypothetical "clicktionary" yet, a tool which would let the user click on any word anywhere and which would not only bring up the correct dictionary entry but would also highlight the correct sense inside the entry.

Both these trends – the trend towards aggregation and the trend towards invisibility – are in extremely early stages yet. The current state of the art is more mundane and prosaic: we have dictionary websites and dictionary apps which, while offering a slightly better user experience over printed dictionaries, are not really bringing anything qualitatively new, anything the printed dictionaries were not doing too. Interaction between humans and dictionaries is therefore an area in a relatively shallow state of digitisation, an area where radical innovation is waiting to happen yet.

2.5 Summary: digitisation deep and shallow

This chapter has analysed the process of making and delivering a dictionary as something that unfolds in stages, and we have shown how the different stages have become digitised to different depths. The initial stage of the process – *knowledge acquisition* – is now so deeply digitised that hardly any qualitatively new developments are expected any more, while all the other stages – from what happens in dictionary writing systems to what eventually lands on an end-user's screen – are still in a shallow state of digitisation and there is potential for qualitative jumps to completely new levels.

This is normal for any industry which is undergoing a digital transformation. In an influential book on digital transformations in business, Ross et al. (2021) distinguish between two stages of digitisation: early digitisation is when existing business models and processes are merely *improved* by becoming digital, while the later stage, when the business truly becomes a "digital business", is when the digital infrastructure enables the discovery of completely new, digital-only models and offerings which never existed

before. This corresponds to the distinction between *shallow* and *deep* digitisation in this chapter.

The fact that shallow comes before deep appears to be a feature of technological progress generally. Pemberton (2023) makes a similar observation on innovations that happened long ago: “Whenever a new technology is introduced, it imitates the old. Early cars looked like horseless carriages because that is exactly what they were. [...] It took a long time for cars to evolve into what we now know.” And: “For the first 50 years, [printed] books looked just like manuscripts: hand-writing fonts, no page numbers, no table of contents, or index. Why? That was what was expected of a book at the time. [...] After about 50 years, readable fonts were introduced.”

When an old technology imitates the old and only improves it a little, that is a shallow form of innovation. In lexicography, one of the areas affected by shallow innovation is the data structures used for representing lexicographic content. The rest of this thesis is going to be about making proposals for deep innovation in this area.

BLOCK II: DATA LANGUAGES

3 Lexicography versus XML

This chapter takes a critical look at how XML is used in lexicography and asks the question, why do dictionary entries often end up looking so complex when encoded in XML? The main reason for the perceived complexity of XML-encoded dictionaries is *purely structural markup*: XML elements which contain other XML elements instead of human-readable text. The over-abundance of purely structural markup in lexicography is caused by the nature of lexicographic content, much of which is inherently headed. XML has no support for headedness and neither do other commonly used languages such as JSON and YAML. In this chapter we propose a number of constraints and extensions to XML, JSON and YAML which add support for headedness into these languages.

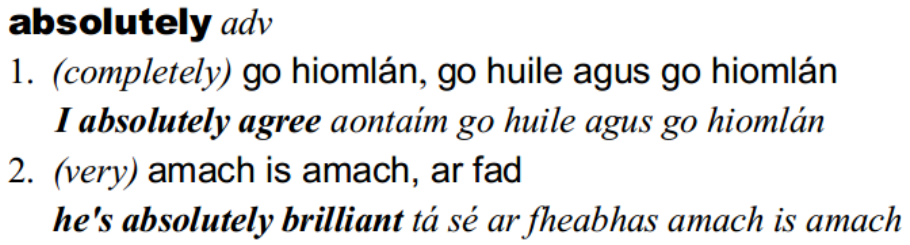
3.1 Introduction: dictionaries and XML

Lexicography is the discipline of creating dictionaries (where by dictionaries we mean books, websites and apps where human users look up information about words). In modern lexicography, dictionary entries are usually encoded in XML (W3C 2008). Each dictionary entry is typically its own XML document, and each such XML document conforms to an XML schema created for that particular dictionary. An example can be seen in Listing 3-1 below which shows how a dictionary entry from a bilingual dictionary would typically be encoded in XML. For comparison, Figure 3-1 shows how the same entry would eventually be presented to a human user.

Listing 3-1 A dictionary entry encoded in XML

```
<entry>
    <headword>absolutely</headword>
    <pos>adv</pos>
    <sense>
        <gloss>completely</gloss>
        <translation>go hiomlán</translation>
        <translation>go huile agus go hiomlán</translation>
        <exampleContainer>
            <example>I absolutely agree</example>
            <translation>aontaím go huile agus go hiomlán</translation>
          </exampleContainer>
    </sense>
    <sense>
        <gloss>very</gloss>
        <translation>amach is amach</translation>
        <translation>ar fad</translation>
        <exampleContainer>
            <example>he's absolutely brilliant</example>
            <translation>tá sé ar fheabhas amach is amach</translation>
        </exampleContainer>
    </sense>
</entry>
```

Figure 3-1 A human-readable rendering of a dictionary entry encoded in XML



Notice that the XML encoding is relatively high-level: it encodes the structure of the entry, not its appearance on screen or on paper. There are XML elements to indicate where the headword is, where one sense ends and another begins, and so on. So, we can define dictionary encoding as the activity of taking an inventory of lexicographically relevant content items such as headwords, part-of-speech labels, senses and translations, and expressing them formally in a language such as XML.

XML is the most commonly used encoding language for dictionaries today. As lexicography began digitising itself in the late 1990s and early 2000s, XML seemed like an obvious choice: for example, an early seminal paper on dictionary encoding (Ide et al. 2000) extols the virtues of XML and does not even consider any alternatives—to be sure, no workable alternatives to XML existed in the early years of digital lexicography. XML was already popular for text encoding in general, and its underlying tree-like object model fitted in nicely with pre-existing thinking in theoretical lexicography where dictionary entries were modelled as tree structures (Wiegand 1989).

It is the 2020s now and lexicography has long transitioned from paper to screens. The focus has moved from retrodigitising old paper-bound dictionaries to producing new born-digital ones. There have been advances in automation, so that we no longer talk of writing dictionaries but generating them from data and then post-editing them (Jakubíček et al. 2018). There have been quantitative advances in both scale (how many dictionaries are produced, how large they are) and speed (how quickly). XML is still with us in this new world.

This chapter asks whether XML is still fit for the job. Some of the recent advances in digital lexicography have given rise to scenarios and use cases which were not there in the early years, such as the need to change dictionary schemas frequently during the lifetime of a project, or to make dictionaries more easily processable by machines (as opposed to merely legible to humans). The purpose of this chapter is to show that XML makes some of these tasks unnecessarily difficult, and to look for alternatives.

3.2 The dark side of XML in lexicography

XML has many properties which make it a good language for encoding dictionary entries, for example the fact that XML preserves the order of elements, or that XML has out-of-the-box support for inline markup. Later in this thesis (in Chapter 4) we will give a detailed analysis of those features of XML which are good for dictionary encoding. At this point, however, we are going to concentrate on occasions when the use of XML in lexicography is more hindrance than help.

Since its emergence in the late 1990s[1] and despite its popularity, XML has been subject to passionate criticism from many quarters (Carlson 2007). The usual objection is that XML is a "verbose" language, which is another way of saying that XML documents tend to have an inconveniently high ratio of tags to content: it takes a lot of tags to encode a little content.

Some of the perceived verbosity of XML is caused by superficial design decisions in the syntax of XML, in particular the fact that the name of each element needs to be given twice, first in the opening tag and then again in the closing tag, which is obviously redundant. This is, however, not the only reason why XML looks and feels verbose. There are other, less superficial reasons for the perceived verbosity of XML, reasons which have less to do with the syntax and more with the underlying data model. Nowhere is this more apparent than in lexicography, as we will show in the rest of this section.

3.2.1 Purely structural markup and matryoshkization

We will concentrate here on one less obvious cause of verbosity in XML: the multi-layered embedding of elements inside other elements inside yet more elements, a phenomenon we call *matryoshkization*.[2] Listing 3-2, which shows how a pair of translations would typically be encoded somewhere inside a bilingual dictionary, demonstrates matryoshkization in practice.

[1] The W3C XML recommendation, the de-facto standard for XML, was published in 1998.

[2] A *matryoshka* is a popular Russian wooden toy in the form of a doll. When the doll is opened it reveals a smaller doll inside, which in turn has another smaller doll inside, and so on.

Listing 3-2 A pair of translations encoded in XML

```
<translationGroup>
  <translationContainer>
    <translation>leasú</translation>
    <pos>n-masc</pos>
  </translationContainer>
  <translationContainer>
    <translation>athchóiriú</translation>
    <pos>n-masc</pos>
  </translationContainer>
</translationGroup>
```

The only XML elements here that contain actual human-readable content are `translation` (the translation's wording) and `pos` (its part of speech). The remaining XML elements are purely structural, used for grouping other elements together:

- The `translationContainer` element groups `translation` and `pos` elements together.
- The `translationGroup` element groups several `translationContainer` elements together.

Let us walk ourselves through the hypothetical steps which may have led a schema designer to designing the schema in this way.

Step 1. In the beginning, the requirement was to encode translations. This can be done very easily with just one type of element which we can call `translation`: Listing 3-3.

Listing 3-3 Two translations

```
<translation>leasú</translation>
<translation>athchóiriú</translation>
```

Step 2. Then the schema designer realised that we need to encode part-of-speech labels for each translation, using an element we can call `pos`: Listing 3-4.

Listing 3-4 Two translations and two POS labels

```
<translation>leasú</translation>
<pos>n-masc</pos>
<translation>athchóiriú</translation>
<pos>n-masc</pos>
```

Step 3. But, to indicate which part-of-speech element belongs to which translation, the schema designer decides to group each pair under a common parent. A popular naming convention in lexicography is to call such elements containers, for example `translationContainer`: see Listing 3-5. This has introduced one level of matryoshkization into the entry schema: one layer of purely structural markup.

Listing 3-5 One layer of purely structural markup

```
<translationContainer>
    <translation>leasú</translation>
    <pos>n-masc</pos>
</translationContainer>
<translationContainer>
    <translation>athchóiriú</translation>
    <pos>n-masc</pos>
</translationContainer>
```

Step 4. At this point, the schema designer notices that the code in which translations are encoded is quite long. It occurs to him or her that it might be a good idea to wrap all translation containers inside yet another layer of purely structural markup, so that it becomes easier to collapse and expand in an XML editor. In lexicography, when an element's only purpose is to group a list of elements of the same type together, a popular naming convention is to call it a group, for example `translationGroup`: Listing 3-6.

Listing 3-6 Two layers of purely structural markup

```
<translationGroup>
  <translationContainer>
    <translation>leasú</translation>
    <pos>n-masc</pos>
  </translationContainer>
  <translationContainer>
    <translation>athchóiriú</translation>
    <pos>n-masc</pos>
  </translationContainer>
</translationGroup>
```

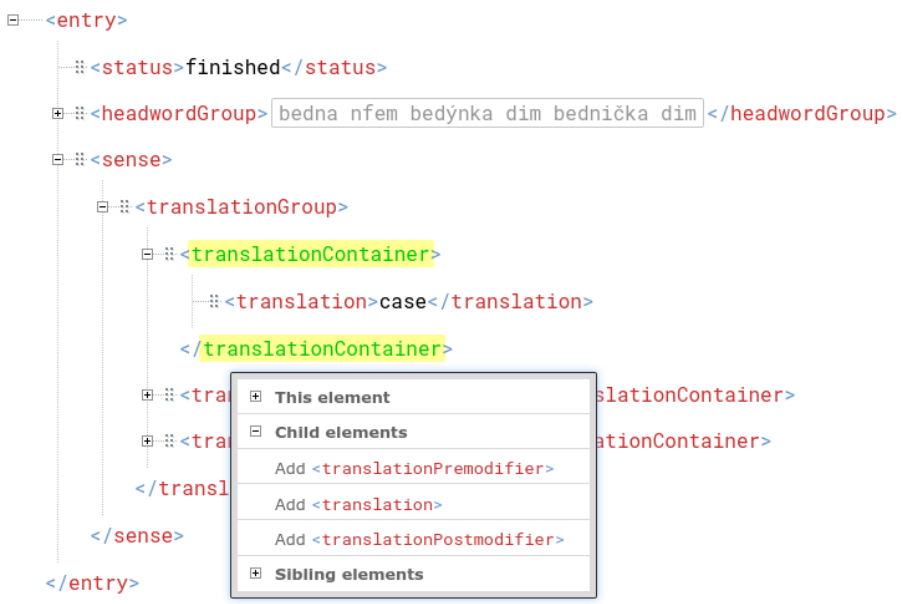
We have ended up with two layers of purely structural markup in the entry schema. The source code of our entries has become difficult for humans to read and navigate while editing. Most of the tags are purely structural, while tags which surround actual human-readable content are the minority.

The trouble is, however, that the purely structural markup is not redundant. It (or most of it) is there to encode lexicographically relevant facts, such as the fact that this part-of-speech label belongs to this translation. The matryoshkization seems unavoidable, a necessary consequence if one wants to encode the facts one wants to encode. In the author's experience, lexicographers (and more importantly, IT professionals working in lexicography) often tacitly accept highly verbose XML as a necessary evil, as an inconvenience which needs to be accepted because there is no other way.

3.2.2 Matryoshkization versus your entry editor

A frequent counter-objection is that matryoshkization is not a problem because editing tools can hide the verbosity from the human lexicographer. It is, of course, possible in principle to create editorial user interfaces which do not expose the human lexicographer to the verbosity of the underlying XML. In practice, however, this is almost never done. All dictionary writing systems in wide use today (see section 2.3) are basically schema-driven XML editors where the lexicographer is fully exposed to the verbosity of the purely structural markup (example in Figure 3-2).

Figure 3-2 A typical lexicographic XML editor (Lexonomy).



To "hide" the XML from the lexicographer, one needs to develop a customised editorial UI which is specific to that particular dictionary (or, more accurately, to that particular entry schema). This can be a non-trivial software development task, especially if one considers the necessity to maintain the UI throughout the lifetime of the project and to keep it synchronised with changes to the schema. Most dictionary projects do not have the staff or the budget for such software development effort. Most dictionary projects simply procure an off-the-shelf dictionary writing system and customise it with their own entry schema. Dictionary writing systems typically do not even allow much more customisation than that. The only widely used dictionary writing system where the XML can be hidden behind a custom-built entry editing "widget" is Lexonomy, but this feature is rarely used there—precisely for the reason that developing and maintaining the widgets is expensive.

Therefore, it is invalid to claim that matryoshkization does not matter because it can be hidden. Matryoshkization cannot easily be – and rarely is – hidden from human lexicographers. Matryoshkization is a real and existing inconvenience on many dictionary projects.

3.2.3 Matryoshkization versus schema migration

The fact that entries are difficult to read and navigate for human lexicographers is not the only consequence of matryoshkization. Another consequence is that almost every change to the entry schema renders existing entries invalid.

Let us illustrate that by returning to the hypothetical example of a schema designer who is in the process of designing an entry schema for a new dictionary project. In **Step 1**, the designer has designed a schema which allows translations to be encoded in the simplest possible way, using just one type of element called `translation`: Listing 3-7.

Listing 3-7 Translations without POS labels

```
<translation>leasú</translation>
<translation>athchóiriú</translation>
```

The project starts and several hundreds of entries are encoded using this schema. Then the requirements change and it transpires that we need to add part-of-speech labels to some (but not all) translations. The schema designer goes back to the drawing board and follows through with **Steps 2 and 3**: the schema is changed so that translations are now to be encoded in a `transationContainer` element which can have two child elements, one `translation` and zero or more `pos`: Listing 3-8.

Listing 3-8 Translations with optional POS labels

```
<translationContainer>
    <translation>leasú</translation>
</translationContainer>
<translationContainer>
    <translation>athchóiriú</translation>
    <pos>n-masc</pos>
</translationContainer>
```

The schema designer has made two changes into the schema: (1) the new `pos` element type is now an optional sibling of `translation` and (2) the new `translationContainer` element type has taken the place of `translation`, "demoting" it to the role of its child. The first change does not cause pre-

existing entries to be invalid, but the second one does. The consequence is that all previously encoded entries are now invalid as per the new schema—including, frustratingly, those entries where we are not planning to add any part-of-speech labels. So, changing the schema was only half the work: we also need to write a schema migration script to make the existing entries valid again.

Every time we add new non-structural element types into an entry schema, such as `pos` in our example, the change usually does not cause pre-existing entries to be invalid (as long as the new element is optional). But when we add new purely structural markup into the schema, like we did when we introduced `translationContainer`, the schema becomes matryoshkized, all pre-existing entries become invalid and we need to fix that with a schema migration script. In other words, matryoshkization not only makes entries verbose, it also makes schema migrations more difficult.

A possible counter-objection is that this (= the necessity to write schema migration scripts every time we matryoshkize the schema) is unavoidable because the matryoshkization itself is unavoidable: there is no other way to encode what we want to encode than through purely structural markup. To be sure, this inconvenience is not unique to XML, schema migration scripts are common everywhere data is managed, in particular in relational databases. But that is beside the point. Avoidable or unavoidable, matryoshkization (and the necessity for schema migration scripts) is a hindrance to agility in the dictionary building process: it prevents the schema designer from making changes to the schema unreluctantly and frequently, in response to evolving project requirements.

3.2.4 Look-ahead matryoshkization

Experienced schema designers are often keen to avoid having to change the entry schema halfway through a project. For that reason, schema designers often choose to matryoshkize the schema even if there is no need for it yet, a phenomenon we can call look-ahead matryoshkization. For example, when designing a schema for encoding translations, the designer may introduce the purely structural element `translationContainer` from the very start, as in

Listing 3-9, even though there is no need for it and a `translation` element on its own would do. The designer is hoping to future-proof his or her schema: should a requirement for a `pos` sibling to `translation` emerge in the future, he or she will be able to introduce it into the schema without invalidating existing entries and without having to write a schema-migration script. This is perhaps wise and prudent—but if that requirement never emerges, then we have ended up with a dictionary full of XML-encoded entries which are more verbose than they need to be.

Listing 3-9 Translations with look-ahead matryoshkization

```
<translationContainer>
    <translation>leasú</translation>
</translationContainer>
<translationContainer>
    <translation>athchóiriú</translation>
</translationContainer>
```

3.2.5 Summary: XML in lexicography

Dictionary entries, when encoded in XML, tend to be overly verbose due to a phenomenon called matryoshkization. Matryoshkization is caused by the presence of purely structural markup. In addition to verbosity, matryoshkization also causes difficulties during schema updates.

Some degree of matryoshkization and purely structural markup can be observed in practically every discipline where XML is used, but (arguably) it is more prevalent in lexicography than anywhere else. So, in the next two sections, we are going to analyse in more detail the patterns of purely structural markup which occur often in lexicography and we will ask the question, what is so special about lexicographic data that makes matryoshkization so prevalent?

3.3 Patterns of purely structural markup

We can define purely structural markup as such XML elements which contain no text nodes as their direct children: all their child nodes are other XML elements. We have seen how too much structural markup leads to the

phenomenon of matryoshkization, which is a special subcase of the phenomenon of verbosity for which XML is often criticised. Let us now review the patterns of purely structural markup that commonly occur in lexicography. Broadly speaking, there are two patterns: the ‘list’ pattern and the ‘headed’ pattern.

3.3.1 The ‘list’ pattern of purely structural markup

Listing 3-10 Example of the ‘list’ pattern

```
<translations>
  <translationContainer>...</translationContainer>
  <translationContainer>...</translationContainer>
  <translationContainer>...</translationContainer>
</translations>
```

The first pattern is where a parent element wraps a sequence of child elements which are all of the same type. It is there because the designer of the schema probably thought it useful to group elements of the same type under a common parent element, like in **Step 4** of our fictional but realistic schema design process.

The usefulness of this grouping is debatable. The group thus created does not seem to represent any lexicographic fact which a lexicographer might want to communicate to the dictionary’s end-users. The parent wrapper is almost always unnecessary in the sense that it conveys no information which could not be inferred: the fact that there exists a list of translations is obvious from the fact that there is a sequence of `translation` elements in the entry. Grouping them under a common parent does not contribute any new information.

Unnecessary grouping of this kind can be found in XML outside lexicography too and tends to be advised against in XML styleguides (Ogbuji 2004). The ‘list’ pattern can almost always be explained away as a bad practice, and the dictionary schema can be made less complex by simply removing the purely structural elements.

3.3.2 The ‘headed’ pattern of purely structural markup

Listing 3-11 Example of the ‘headed’ pattern

```
<translationContainer>
  <translation>athchóiriú</translation>
  <pos>n-masc</pos>
  <usage>formal</usage>
</translationContainer>
```

The second pattern is where a parent element wraps child elements of different types, one of which can be considered the “head” and the others can be seen as providing additional information about the head. An example is `translationContainer` which can be said to be headed by `translation`, while the other children `pos` and `usage` provide additional information about the head.

Unlike the list pattern, the headed pattern cannot be explained away as a bad practice. Its purpose is to encode lexicographic facts which the lexicographer wants to communicate to the end-user, for example the fact that this `pos` element belongs to this `translation` element. The purely structural `translationContainer` element is a tool for encoding that fact.

Whenever during the process of designing an entry schema for a dictionary a requirement arises to encode something which appears to have a “head” plus a few other elements that provide additional information about the head, the headed pattern of purely structural markup is a popular choice—as it was for our fictional schema designer in **Step 3** above.

Why is the headed pattern of purely structural markup so popular in lexicography? The reason is that much of lexicographic content inherently is headed: we will show multiple examples of that in the following section.

3.4 The headedness of lexicographic data

n XML, at an abstract level, every XML element can be seen as a pair of two things: a name and a value. The name is what we have in the opening and closing tags, while everything between the tags is the value which can be

either plain text, or a list of child elements, or a mixture of both (so-called mixed content), or it can be empty. But the point is that an XML element always consists of exactly two things: a name and a value, even if the value is complex.

In lexicography, on the other hand, much of the content we encounter could more efficiently be modelled as a triple, as a group of three things: a name, a value, and a list of modifiers containing zero, one or more other such triples. The name and the value together are the head. Many content objects in lexicography are inherently headed, but headedness is difficult to model in XML without purely structural markup. Let us look at some examples of lexicographic content objects which are headed.

3.4.1 Translations are headed structures

Listing 3-12 A typical XML encoding of a translation

```
<translationContainer>
  <translation>athchóiriú</translation>
  <pos>n-masc</pos>
  <usage>formal</usage>
<translationContainer>
```

Listing 3-13 The same translation in concise pseudocode

```
translation: athchóiriú
  pos: n-masc
  usage: formal
```

In many bilingual dictionaries, translations are given simply as strings of text with no other information. Such translations are not headed, of course. But, in an encoding-oriented dictionary (ie. a dictionary which tells you how to express something in a language in which you are not fluent), translations are often decorated with grammatical annotations (part-of-speech labels) and pragmatic annotations (usage labels). Such translations are headed: the `translation` element together with its plain-text value is the head, while the other elements (`pos` and `usage`) are modifiers of the head. Purely structural markup (in the form of a parent element such as `translationContainer`) is often used to encode this in XML.

3.4.2 Example sentences are headed structures

Listing 3-14 A typical XML encoding of an example sentence

```
<exampleContainer>
  <example>Ich nehme den Regenschirm mit.</example>
  <source>bib-147_12</source>
  <translation>I'll take my umbrella with me.</translation>
</exampleContainer>
```

Listing 3-15 The same example sentence in concise pseudocode

```
example: Ich nehme den Regenschirm mit.
  source: bib-147_12
  translation: I'll take my umbrella with me.
```

In many dictionaries, example sentences are not just strings of text: they come with additional content such as bibliographical references (to tell us where the example comes from), usage labels (to tell us, for instance, that this sentence is colloquial) and translations. In other words, dictionary examples are headed structures: the `example` element together with its plain-text value is the head, while the other elements are modifiers of the head. Some of the modifier elements can be headed structures too: for instance, it is imaginable that translations could have their own modifiers, as in Listing 3-16.

Listing 3-16 Example sentences have translations which have usage labels

```
example: Ich gehe auf Nummer sicher.
  translation: I'll play it safe.
    usage: informal
  translation: I'll stay on the safe side.
    usage: neutral
  translation: I will err on the side of caution.
    usage: formal
```

3.4.3 Collocations are headed structures

Listing 3-17 A typical XML encoding of a collocate

```
<collocation>
  <collocate>make</collocate>
  <example>I have made a mistake.</example>
  <example>Everybody makes mistakes.</example>
</collocation>
```

Listing 3-18 The same collocate in concise pseudocode

```
collocate: make
  example: I have made a mistake.
  example: Everybody makes mistakes.
```

It is becoming common for dictionaries to contain information about the collocates of the headword: words which often occur together with the headword in real-world language use. For instance, inside the entry for the headword ‘mistake’ we might find a block of information that tells us that the headword collocates with the verb ‘make’ (as in ‘to make a mistake’), and then gives us some additional information about this collocation, such as some usage labels or a few example sentences. So, in a dictionary entry, collocations are headed structures: the `collocate` element together with its plain-text value is the head, while the other elements are modifiers of the head.

3.4.4 Senses can be headed structures too

Listing 3-19 A typical XML encoding of a sense

```
<sense>
  <definition>an institution where you store money</definition>
  <translation>banque</translation>
  <example>I got a large loan from the bank.</example>
</sense>
```

Listing 3-20 The same sense in concise pseudocode

```
definition: an institution where you store money
  translation: banque
  example: I got a large loan from the bank.
```

In lexicography, a dictionary entry is typically subdivided into one or more senses. A sense is a container for things such as definitions, translations and examples. Normally, a sense is not a headed structure because there is no obvious “head”: no single element inside the sense where we could say that all other elements are its modifiers. In XML, senses are practically always encoded by means of purely structural markup: there is a `sense` element which has no plain-text children of its own, but has many child elements such as `definition`, `translation` and `example`.

But is it true that senses are not headed structures? There is a case to be made that definitions are the heads of senses. A definition says that such-and-such meaning of the headword exists, and the remaining elements inside the sense can be understood as providing additional information about that meaning.

Not all dictionaries contain definitions. But, in those that do, it is possible to understand senses as headed structures. In an XML encoding of senses, the `sense` element is yet another incarnation of the ‘headed’ pattern of structural markup.

3.4.5 Entries can be headed structures too

Listing 3-21 A typical XML encoding of an entry

```
<entry>
  <headword>bank<headword>
  <partOfSpeech>noun</partOfSpeech>
  <sense>an institution where...</sense>
  <sense>a stretch of land...</sense>
</entry>
```

Listing 3-22 The same entry in concise pseudocode

```
headword: bank
  partOfSpeech: noun
  sense: an institution where...
  sense: a stretch of land...
```

We can perform the same re-analysis on entries as we did on senses. Entries do not seem like obviously headed structures: they are simply containers for various elements such as headwords, part-of-speech labels and senses.

But one of then does stand as a possible candidate for being the head: the headword! It is, after all, called a headword for one good reason: its purpose is to head the entire entry, while the rest of the entry is about the headword. On that analysis, even entire dictionary entries can be understood as headed structures, and the very existence of an `entry` element in XML-encoded dictionaries can be understood as an incarnation of the 'headed' pattern of structural markup, a consequence of matryoshkization.

3.5 How to encode headedness in XML

We have seen in the previous section that headed content structures are far from uncommon in lexicography: it so happens that much of dictionary content is inherently headed. And, in the sections before that, we have seen that to encode headed structures in XML, purely structural markup (more specifically, the 'headed' pattern of purely structural markup) is commonly used in lexicography, and that this is problematic because it has negative implications on readability and because it causes complications during schema updates.

The question to ask now is, are there other ways to encode headedness in XML? Is it possible to encode headed structures in XML without recourse to purely structural markup? In this section we will evaluate several options, some obvious and some less so.

3.5.1 Strategy 1: parentless sequencing

We have said before that the purpose of purely structural markup (in the 'headed' pattern) is to group elements together: to indicate which `pos` belongs to which `translation` and so on. Theoretically, it might be possible to achieve the same goal without purely structural markup, by relying only on the listing order of elements, as in Listing 3-23.

Listing 3-23 Two headed structures encoded as parentless sequencing

```
<translation>leasú</translation>
<pos>n-masc</pos>
<translation>athchóiriú</translation>
<pos>n-masc</pos>
```

In this scenario, we would "know" that each `pos` element belongs to its nearest preceding sibling `translation` element. The problem with this approach is that this fact is not encoded explicitly in the XML, and tools processing this XML in the future may not "know" it as we "know" it now: to an XML parser, `pos` and `translation` are simply siblings and nothing else. We would need to program additional logic on top of the XML parser to make that explicit. So, parentless sequencing defeats the purpose of encoding entries in XML in the first place: to take facts which are implicit and make them explicit.

3.5.2 Strategy 2: mixed content

Yet another suggestion is to encode headedness as mixed content. Mixed content is a strategy used in XML to encode inline markup, a typical example is tags such as `b`, `i` and `a` in HMTL: see Listing 3-24.

Listing 3-24 HTML with mixed content

```
<p>
  This is <b>very</b> important.
</p>
```

To say that an XML element has "mixed content" is another way of saying that its child nodes are a sequence of text nodes and elements. This is a good strategy for encoding inline markup. Could it be a good strategy for encoding headedness, as in Listing 3-25?

Listing 3-25 A headed structure encoded as mixed content

```
<translation>
  athchóiriú
  <pos>n-masc</pos>
  <usage>formal</usage>
</translation>
```

The problem is that there is no formal distinction (to an XML parser) between the head (= the element's first child) and the modifiers (= the element's other children). If we ask an XML parser to give us the text of the `translation` element, it will give us a concatenation of all the text node descendants, which is the string `athchóiriú n-masc formal` (with whitespace collapsed).

The problem becomes more apparent if the head's value contains inline markup, like in Listing 3-26. Here, the `example` element has four children: the text to implement electoral (with a trailing space), followed by the `h` element, followed by two more elements. An XML parser has no way of knowing that the first two children are part of the head's value and the others are not.

Listing 3-26 A headed structure, encoded as mixed content, where the head has inline markup

```
<example>
  to implement electoral <h>reform</h>
  <source>EU legislation</source>
  <translation>leasú toghchánach a chur i bhfeidhm</translation>
</example>
```

The mixed content strategy is only one step away from purely structural markup. The one step is to take those children that constitute the head's value and wrap them in yet another element, as in Listing 3-27.

Listing 3-27 XML with purely structural markup

```
<exampleContainer>
  <example>to implement electoral <h>reform</h></example>
  <source>EU legislation</source>
  <translation>leasú toghchánach a chur i bhfeidhm</translation>
</exampleContainer>
```

This is an improvement on the mixed content strategy because the head value is now explicitly demarcated from the rest. But the downside is that our schema is now matryoshkized, with all the disadvantages we have identified above.

3.5.3 Strategy 3: children as attributes

A simple suggestion that might occur to a schema designer wanting to avoid purely structural markup is to use XML attributes instead: the head would be encoded as an XML element and all its children would become its attributes, as in Listing 3-28.

Listing 3-28 XML with the children as attributes

```
<translation pos="n-masc" usage="formal">
  athchóiriú
</translation>
```

The problem with this suggestion is that it does not scale beyond a few simple examples. This is because XML attributes come with several inconvenient limitations:

- Attribute names have to be unique, meaning that there can never be, for example, two pos attributes or two usage attributes in an element.

- Attribute values are plain text with no structure. So, it is impossible for an attribute to have its own attributes, or any other kind of child nodes, or to contain a list of values. In other words, an XML attribute is similar to an XML element in that it is a name-value pair, but with the additional limitation that the value must be plain text.

3.5.4 Strategy 4: heads as attributes

He other way around to encode values as attributes with a pre-agreed name such as `value`. Children are then encoded as normal XML elements, as in Listing 3-29.

Listing 3-29 XML with the head as an attribute

```
<translation value="athchóiriú">
  <pos value="n-masc"/>
  <usage value="formal"/>
</translation>
```

This encodes headedness successfully but has an even larger problem than the previous strategy: now all values must be plain text, in-line markup is impossible everywhere.

3.5.5 Conclusion: headedness in XML

The conclusion for this section is that even though it is possible to find strategies in XML to avoid purely structural markup and/or to represent headedness, each strategy comes with its own trade-offs. These trade-offs may or may not be acceptable to the schema designer depending on the requirements of the project, for example whether in-line markup is needed or not.

3.6 How to encode headedness in other data languages

Lexicography abounds in headed structures but the markup language we use in lexicography most often, XML, was never designed for it and can only accommodate it awkwardly. This is unfortunate. But can we perhaps find another markup language to use in lexicography instead of XML, one that can encode headedness more gracefully? In this section we will evaluate JSON and YAML as currently popular alternatives to XML, we will also look at SGML as XML's historical predecessor, and we will also look at one less well-known language called NVH. In each case we will ask whether the language is able to encode headed structures without purely structural markup, and if not, how the language would need to change to support headedness.

3.6.1 Headedness in SGML

XML's historical predecessor was SGML (ISO 8879:1986, Goldfarb and Rubinsky 1990). Invented primarily as a text markup language, SGML was[1] more complex than XML, but this complexity enabled many markup minimisation

[1] We are talking about SGML in the past tense, as if SGML no longer existed. This is of course not true, SGML still exists. The past tense here is only a reflection of the fact that SGML is rarely used anymore, at least for new projects.

features which, in retrospect, made SGML into a language which supported headedness.

One of SGML's markup minimisation features was the ability to omit closing tags. Early versions of the HTML standard had a similar feature. So, it was possible to write code like in Listing 3-30.

Listing 3-30 SGML with minimised markup

```
<translation>athchóiriú
<pos>n-masc
```

The parser would implicitly "assume" the missing closing tags from its knowledge of the document schema. If the schema says that the `translation` and `pos` elements can only have text content and no child elements, then obviously they must be siblings and the parser will read the code as if the closing tags where there, like in Listing 3-31.

Listing 3-31 SGML without minimised markup

```
<translation>athchóiriú</translation>
<pos>n-masc</pos>
```

This features of SGML made it possible to write less verbose code, but it still does not turn SGML into a headedness-supporting language. The markup minimisation feature which does turn SGML into such a language is something called implicit elements. In SGML, is was possible the specify in the document schema that certain element tags can left out altogether, even though the parser would still "assume" them to be there. Let us demonstrate that on an example where we take a matryoshkized XML fragment and re-encode it in SGML. We start with a fragment like in Listing 3-33.

Listing 3-32 SGML with all elements explicit

```
<translation>
  <value>athchóiriú</value>
  <pos>n-masc</pos>
  <usage>formal</usage>
<translation>
```

Then, in the document schema, we specify that the `value` element is implicit. It now becomes possible to leave its opening and closing tags out, as in Listing 33.

Listing 3-33 SGML with an implicit element

```
<translation>
  athchóiriú
  <pos>n-masc</pos>
  <usage>formal</usage>
<translation>
```

This looks similar to our attempt to encode headedness in XML through mixed content, but the trick is that this is not mixed content. When parsing this code fragment, the SGML parser will understand from the schema that

1. `translation` is not allowed to have any text content, and

2. `translation` is required to have as its first child an element called `value` which is required to have text content.

These facts will trigger the SGML parser into interpreting the code as if the `value` element were actually there, like in the previous code sample. All this means that SGML could, in principle, be used in lexicography to encode headed structures in a such a way that schema migration does not cause problems. Let us assume we start with a simple entry schema where translations are encoded like in Listing 3-34.

Listing 3-34 This SGML fragment validates in both schemas

```
<translation>
  athchóiriú
<translation>
```

If we then update the schema such that

1. `translation` is no longer allowed to contain text content, and

2. `translation` is required to a contain an implicit element called `value` (as well other optional children such as `pos` and `usage`)

then the original entries are still parsed as valid: the SGML parser "assumes" the implicit element to be there. We can matryoshkize the schema without having to matryoshkize the data, and no schema migration scripts are needed.

To the author's knowledge, however, this property of SGML was never taken advantage of in lexicography. Lexicography began digitising itself at a time when SGML had already peaked in popularity and XML was seen as its successor. And, to be sure, the flexibility of SGML came at a cost, as SGML was computationally hard to implement: all the markup minimisation features made it difficult to write parsers for SGML. XML evolved out of SGML to solve precisely that problem, as a subset of SGML which is more easily processable by machines. In its evolution from SGML to XML, the language gained machine processability and became easy to adopt, but lost support for headedness and gained on verbosity.

3.6.2 Headedness in JSON

As a serialisation format for data, JSON (ECMA 404, ISO/IEC 21778:2017) is often claimed to be more easily human-readable than XML. JSON is definitely less verbose than XML, mainly because the names of objects do not have to be repeated at the end of every object, which makes JSON significantly faster for (uncompressed) transmission than XML (Nurseitov et al. 2009). Listing 3-35 shows how an entry fragment might be encoded in JSON.

Listing 3-35 How a translation might be encoded in JSON

```
{
  "translationContainer": {
    "translation: "athchóiriú",
    "pos": "n-masc",
    "usage": "formal"
  }
}
```

Apart from this, however, JSON has the same problem as XML: it does not support headed structures. The code in listing 35 is JSON's equivalent of matryoshkization and purely structural markup: translationContainer is the purely structural element because it is an object which contains no literal text as its immediate child, all its children are other objects.

None of the strategies discussed for XML in Section 3.5 have equivalents in JSON. The parentless sequencing strategy is impossible in JSON because JSON requires the names inside an object to be unique: Listing 3-36 is illegal in JSON. The mixed content strategy is not an option either because JSON does not allow mixing literal values with name-value pairs: Listing 3-37 is also illegal in JSON. The only way to represent mixed content in JSON is to use array syntax `[...]` which comes with its own share of purely structural markup. And finally, the remaining two options discussed for XML which make use of attributes have no equivalents in JSON because there is no such thing as attributes in JSON.

Listing 3-36 Parentless sequencing (illegal in JSON)

```
{
    "translation": "leasú",
    "pos": "n-masc",
    "translation": "athchóiriú",
    "pos": "n-masc"
}
```

Listing 3-37 Mixed content (illegal in JSON)

```
"translation: {
    "athchóiriú",
    "pos": "n-masc",
    "usage": "formal"
}
```

We have seen how, in XML, every element is basically a name-value pair, where the value can be a literal value or a list of children. In JSON, every member is similarly a name-value pair. The name appears before the colon : and the value after it, where the value can be either a literal or a complex object. The underlying object model of JSON is therefore similar to that of XML. When we ignore the superficial differences in the syntax of the two languages, there are only two relevant differences in their object models (after Bourhis et al. 2020, Section 2.3): data elements in JSON are unordered whereas in XML they are ordered, and the keys inside a JSON object must have unique names whereas in XML the children of a parent are not required to have unique names.

In theory, it would be possible to extend the JSON language so that name-value pairs can optionally become triples consisting of a name, a value and an object containing the children. Listing 3-38 shows what a data fragment might look like when encoded in such an extension of JSON. This would introduce built-in support for headedness into JSON. This is, however, only a hypothetical speculation as no such JSON extension exists.

Listing 3-38 A hypothetical extension of JSON to support headedness

```
"translation: "athchóiriú" {
    "pos": "n-masc",
    "usage": "formal"
}
```

3.6.3 Headedness in YAML

A popular serialisation language which is even less verbose than JSON is YAML. YAML was designed deliberately to be as human-readable and human-writable as possible. Where other languages use (curly, pointy...) brackets and quotation marks to demarcate where things begins and end, YAML uses whitespace and indentation. If data encoded in JSON look and feel like source code in JavaScript or some other C-style language, then data encoded in YAML looks and feels like source code in Python. Listing 39 shows how an entry fragment might be encoded in YAML.

Listing 3-39 How a translation might be encoded in YAML

```
translationContainer:
  translation: athchóiriú
  pos: n-masc
  usage: formal
```

This is undoubtedly as “unverbose” as we can get from any serialisation language. But, crucially, this still does not encode the fact that the string `athchóiriú` is the head of the whole structure. Same as in the JSON example, `translationContainer` is a purely structural element.

Like XML and JSON, YAML has no support for headedness, and the only way to encode headed structures is either to matryoshkize the data through purely

structural elements, or to accept some other trade-off. The strategy of parentless sequencing and the mixed content strategy are not possible in YAML (without introducing their own purely structural markup), and the two strategies based in attributes are not possible either because there is no concept of attributes in YAML.

As a thought experiment, how would the syntax of YAML need to change to be able to accommodate headed structures? It would have to be possible for an object to have both a literal value and a list of children, like in Listing 3-40.

Listing 3-40 A hypothetical extension of YAML to support headedness

```
translation: athchóiriú
  pos: n-masc
  usage: formal
```

This is illegal in YAML, but it is in fact the same syntax we have used throughout this chapter to illustrate headed structures. An extension like this would turn YAML into a serialisation language which supports headedness.

3.6.4 Headedness in NVH

NVH (Name-Value Hierarchy)[1] is a less well-known data language conceived by the author of this thesis and developed by computational lexicographers in Masaryk University and in Lexical Computing, a company which makes software for lexicography. NVH is used by Lexical Computing in-house during the semi-automated production of dictionaries (Jakubíček et al. 2018), an agile process where frequent schema updates are common.

The syntax of NVH is similar to YAML, so that an NVH document may (if certain constraints are met) also be a valid a YAML document. Additionally, NVH differs from YAML in that it implements the proposal suggested in the previous section: an element in NVH is allowed to have both a literal value and a list of children, like in Listing 3-40. Listing 3-41 shows what a complete dictionary entry looks like when encoded in NVH.

[1] `https://www.namevaluehierarchy.org/`

NVH is the only markup language in existence designed specifically with headedness in mind. Unlike SGML, which supports headedness at the expense of increased parsing complexity, NVH documents are as simple to parse as YAML or JSON. This is because NVH is built not on the notion of name-value pairs but on the notion of name-value-children triples.

Listing 3-41 An entire entry encoded in NVH

```
headword: house
  pos: noun
  phon: haʊs
    soundfile: house.mp3
  sense:
    definition: a built structure with walls and a roof for living in
    label: Construction
    translation: hiša
      pos: femininеNoun
    translation: dom
      pos: masculineNoun
      label: informal
    collocation: a large house
      translation: velika hiša
      example: We bought a large house.
        translation: Kupili smo veliko hišo.
```

3.7 Conclusion

This chapter has challenged the age-old orthodoxy in computational lexicography that dictionary data is best encoded in XML. XML is widely used in lexicography but, on closer inspection, it turns out not to be the best fit for its requirements. We have analysed what lexicography actually needs from a markup language, with special attention to the inherent headedness of much of lexicographic content. We have seen how widely used languages such as XML, JSON and YAML have no built-in support for headedness, and how attempting to represent headed data in these languages results in an undesirable proliferation of matryoshkization and purely structural markup.

A schema designer who wishes to avoid purely structural markup has a number of options. Within XML, there are strategies for avoiding structural markup, but these come with trade-offs which may or may not be acceptable. Outside XML, we have shown that other well-known languages, namely JSON

and YAML, are no better than XML at meeting the needs of lexicography. The conclusion is that the needs of lexicography would best be met either by a return to SGML, or by an adoption of the less well-known language NVH.

4 Towards a lexicographic data language

The previous chapter brought Name-Value Hierarchy (NVH) onto the stage, a language created specifically for encoding headed content. In this chapter, we return to NVH to introduce its syntax and its design principles more fully. Following this introduction we will contrast NVH against other data languages (XML, SGML, JSON, YAML) and we will evaluate how well or badly each meets the needs of lexicography.

4.1 The design of NVH

The idea for the language now known as NVH emerged during discussions between the author of this thesis and colleagues in Lexical Computing. Its origins can be traced to our frustration with the inability of XML to represent headed content efficiently, without matryoshkisation and without frequent schema migrations.[1] In addition to good headedness support, NVH has borrowed various features from other languages. From YAML, NVH takes its simple, human-writeable syntax. From XML, NVH takes its order-preserving nature and the ability of nodes to have multiple children with the same names – more about all this below. The result is a formal language which is optimized for the needs of a modern, agile, digital-first lexicography.

There is no formal specification of NVH (yet). There is a website[2] which contains an informal description of the language and a GitHub repository[3] which hosts two implementations of an NVH parser: one in JavaScript (written by the author of this thesis) and another in Python (written by colleagues in Lexical Computing). This section summarizes the consensus in this small but growing NVH comunity as to the syntax of the language.

---

[1] What opened the door for something like NVH was a thought experiment: what if XML attributes could have *structure* inside them, just like XML elements do? What would a language that allows this look like? In the end the language ended up looking not at all like XML: if anything, NVH is more similar to YAML.

[2] `https://www.namevaluehierarchy.org/`

[3] `https://github.com/michmech/nvh`

4.1.1 A short introduction to the syntax of NVH

Listing 4-1 A code fragment in NVH

```
example: to implement electoral reform
  source: EU legislation
  translation: leasú toghchánach a chur i bhfeidhm
    label: formal
    label: legal
  translation: athchóiriú toghcháin a chur i bhfeidhm
    label: informal
```

Listing 4-1 shows a fragment of code in NVH. Each line represents the **head** of an NVH **element**. The head consists of the element's **name**, followed by a colon, followed by the element's **value**.

There are no formal rules for what constitutes a valid element name in NVH. The usual practice is that they are "technical" names (like variables in a programming language, or element names in XML), written without whitespace, in *camelCase*.

The element's value is a single-line string without line breaks. Any leading or trailing whitespace is ignored. The value can be an empty string.

Each element can have any number of **child elements**. The hierarchy between parent elements and child elements is indicated by indentation from the left. For example, in Listing 4-1, the element `example` has two child element, `source` and `translation`. The fact that an element can have *both* a value and a list of children is what gives NVH its ability to represent headedness without matryoshkization.

There are no formal rules for whether the indentation should be done using tabs of spaces, and if spaces, how many. The JavaScript parser accepts one tab character or two spaces as one level of indentation. The Python parser accepts one tab character or one space as one level of indentation.

NVH does not require a valid NVH document to have a single top-level element. Both NVH parsers are able to parse NVH documents consisting of several top-level elements – typically, several dictionary entries.

NVH has optional support for inline markup, implemented in the JavaScript parser but not in the Python parser. Inline markup is treated as ordinary child elements, and the text the markup applies to is identified in a stand-off way, as Listing 4-2 illustrates: the `headwordHighlight` element marks up an occurrence of the headword, and the `collocateHighlight` marks up the occurrence of its collocate. The `@1` at the end of the line does two things: it tells the parser that this element is inline markup, and the number tells the parser that it applies to the first occurrence of the substring. As the child elements under `collocateHighlight` show, the markup elements can have their own hierarchies of children and descendants underneath themselves, like any other NVH elements.

Listing 4-2 NVH with inline markup

```
example: We bought a larger house in the village.
  headwordHighlight: house @1
  collocateHighlight: larger @1
      lemma: large
      pos: adj
```

4.1.2 Key differences between NVH and YAML

NVH looks superficially similar to YAML but there are important dfferences which make the two languages incompatible.

- In NVH, an element can have both a value and a list of children. In YAML this is illegal: an element can have one or the other but not both.
- In NVH, the children of an element are not required to have to have unique names: an element can have multiple children with the same name. This is illegal in YAML and has to be represented using YAML's formalism of *lists* instead (similar to arrays in JSON).
- NVH is an order-preserving language: the order in which elements are given is guaranteed to go through parsing and subsequent serialization unchanged, and the position of an element in relation to its siblings is part of the object model produced by the parser. This is not the case with YAML (except for list items).

All of this means that the code in Listing 4-1 would not pass for well-formed YAML. To represent the same content in YAML one would have to resort to various forms of matryoshkization as shown in Listing 4-3.

Listing 4-3 The same data as in Listing 4-1 but in YAML

```
example:
  text: to implement electoral reform
  source: EU legislation
  translations:
    - text: leasú toghchánach a chur i bhfeidhm
      labels:
        - formal
        - legal
    - text: athchóiriú toghcháin a chur i bhfeidhm
      labels:
        - informal
```

4.2 Desiderata for a lexicographic data language

NVH was created to match the needs of lexicography better than other languages. Is the language actualy meeting this goal? If we were to create an ideal encoding language for lexicography, what features should the language have? Which features of XML, SGML, JSON, YAML and NVH would we like to bring into this new language? This section will list some criteria and evaluate each language against them.

4.2.1 Avoiding purely structural markup

Avoiding purely structural markup is important in lexicography for human readability and as a form of preparedness for future schema updates.

- **XML** encourages purely structural markup. To avoid it in XML, one has to resort to strategies which come with trade-offs (see Section 3.5).
- **SGML** makes it possible to avoid purely structural markup thanks to its markup minimisation features. However, this brings an increased complexity for parsing.

- In **JSON** and **YAML**, purely structural markup is practically unavoidable — although the extension proposed in Sections 3.6.2 and 3.6.3, which would add headedness support to the languages, would also remove the need for most purely structural markup.
- **NVH** makes it relatively easy to avoid purely structural markup thanks to its built-in support for headedness.

4.2.2 Headedness

Support for headedness is obviously a high-priority requirement for a lexicographic markup language, given how prevalent headedness is dictionaries.

- **XML** has no built-in support for headedness, except when using one of the attributes-based strategies, which however comes at the expense of the ability to represent in-line markup on the either head element or on the child elements.
- **SGML** has built-in support for headedness if the encoding makes use of SGML's markup minimisation features. This comes at the expense of easy machine processability: parsing SGML is a complex task.
- **JSON** and **YAML** have no built-in support for headedness either. The languages would need to be extended along the lines suggested in Sections 3.6.2 and 3.6.3 in order to support headedness.
- **NVH** has built-support for headedness, but at the expense or making in-inline markup difficult: same as XML when combined with attributes-based strategies.

4.2.3 Explicit listing order

One requirement which is important in lexicography is preserving order. The order in which items are listed needs to fixed, remembered during parsing, and guaranteed to survive every parsing-serialisation roundtrip. Having

things listed in a given order is almost always an implicit requirement when encoding lexicographic data.

- **XML**, **SGML** and **NVH** meet this requirement perfectly. The "order matters" principle is part of the design of the languages.
- In **JSON** and **YAML**, the children of a parent are not in any explicit order. For example, in JSON, every object is basically a collection of key–value pairs, and this collection is unordered. In practice JSON and YAML parsers and serializers often do preserve the order of items, but this is not guaranteed. The only way to fix the order of items is to encode them as an array (in JSON) or as a list (in YAML), which brings its own share of purely structural markup.

4.2.4 Non-unique child names

It is common in lexicography that a content object has multiple children of the same kind, for example an entry contains several senses, a sense contains several translations. To encode this without purely structural markup, the language has to allow the children of an element to have non-unique names.

- In **XML** and **SGML**, non-unique child names are allowed. It is possible, for example, for an `entry` element to have multiple children named `sense`, or for a `sense` element to have multiple children named `translation`.
- In **JSON** and **YAML**, non-unique child names are not allowed, and so code fragments such as listings would be invalid in JSON and YAML. To remodel it into legal JSON or YAML one would need to resort to some form of purely structural markup, for example using array syntax `[...]` in JSON.
- **NVH**, in spite of its superficial similarity to YAML, does allow non-unique child names. So a code fragment like Listing 4-5, although invalid in YAML, is valid in NVH.

Listing 4-4 Invalid JSON with non-unique child names

```
{
  "sense": {
     "gloss": "completely",
     "translation": "go hiomlán",
     "translation": "go huile agus go hiomlán"
  }
}
```

Listing 4-5 Invalid YAML (but valid NVH) with non-unique child names

```
sense:
    gloss: completely
    translation: go hiomlán
    translation: go huile agus go hiomlán
```

4.2.5 Inline markup

One of XML's strong points is its good support for inline markup. Here XML shows its origins as a markup language (as opposed to a serialisation language). This heritage proved itself useful in the early stages of digitisation in lexicography when dictionary entries were treated rather like small documents, consisting of running text which needed to be marked up. Dictionary encoding used to be like text encoding in the early stages of its digitisation, and XML's support for inline markup was useful in that scenario.

Since then, dictionaries have evolved away from the text encoding paradigm. Dictionary entries have ceased to look like running text with markup and have started to look more like structured data records. Consequently, the need for inline markup has diminished. Inline markup is now used fairly rarely in lexicography. The only application where inline markup plays a role (in some dictionaries) is to mark up the occurrences of headwords (and sometimes collocates) inside example sentences. In other wors, in-line markup is a low-priority requirement in lexicography: other requirements, such as headedness or an explicit listing order, are more important.

- **XML** and **SGML** have good built-in support for inline markup, as explained.

- In **JSON** and **YAML**, inline markup is difficult to encode as neither language has any built-in support for it. One must either matryoshkize the data (turn a string into an array of strings and objects) or invent a custom-built formalism (using some form of markdown, or stand-off markup based on start and end indexes).

- **NVH** has no built-in support for inline markup in the current form of the language, but there is a convention for representing in-line markup through stand-off annotation, as show in Listing 4-6. The `@` character identifies the element as in-line markup of its parent, and the index number after it identifies which occurrence of the substring is supposed to be marked up (in case there are multiple occurrences). This convention may become part of the specification of NVH in the future.

Listing 4-6 NVH with inline markup

```
example: We bought a larger house in the village.
  headwordHighlight: house @1
  collocateHighlight: larger @1
      lemma: large
      pos: adj
```

4.2.6 Easily machine-processable

A data language is easily machine processable if it is relatively easy to write parsers for it, if the language is (in some sense of the word) simple. This implies a subjective judgment, but the following is probably a fair summary.

- **XML** (arguably) is easily machine-processable if one ignores optional complications such as namespaces.

- **SGML** (arguably) is not easily machine-processable due to its markup minimisation features which require the parser to have access to the schema and to perform inference during parsing in order to infer closing tags and implicit elements.

- **JSON**, **YAML** and **NVH** (arguably) are easily machine-processable.

4.2.7 Human-friendly

We have discussed in Section 3.2.2 how human-friendliness is important in lexicography because human editors are usually exposed to the full verbosity of the markup language. A data language is human-friendly to the extent that it is human-readable and human-writeable. What that actually means may differ from human to human, but in the author's opinion, a human-friendly language should (1) have as little syntactic punctuation (such as pointy brackets, curly brackets) as possible and (2) indicate structure by something highly visual, such as whitespace and indentation, instead of paired brackets.

- **XML** and **SGML** (arguably) possess a low degree of human-friendliness due to the fact that they contain a lot of syntactic punctuation (although some of it can be minimised in SGML) and because structure is indicated by paired tags, which may not correspond to whitespacing and indentation.

- **JSON** scores better than XML and SGML on human-friendliness because it contains less syntactic punctuation, but worse than YAML and NVH because it still does contain some syntactic punctuation and because structure is indicated by paired brackets.

- **YAML** and **NVH** are practically the same in this respect and both possess a high degree of human friendliness. There is almost no syntactic punctuation, and structure is indicated through indentation.

4.2.8 Non-desiderata

TBD: long text support, data types other than text, namespaces.

4.2.9 Scorecards

This preceding section has listed off a lexicographic "wishlist" of criteria for an ideal serialisation language, and evaluated briefly how each language meets or does not meet the criteria. The results are summarised in Table 4-1 for XML and in Table 4-2 for the remaining languages. Answers that assume

a subjective judgment are labelled with a question mark. In Table 4-2, the column labels *JSON with extensions* and YAML *with extensions* means JSON and YAML with extensions suggested in Sections 3.6.2 and 3.6.3.

As we have seen, there is not a single data language in existence today which would tick all the boxes on the wishlist, although NVH and SGML come close. An interesting observation is that XML, in spite of being widely used in lexicography, is not the best possible fit for the requirements of the field, due mainly to its lack of support for headedness.

Table 4-1 A lexicographic scorecard for XML

	XML with matryoshkization	**XML with parentless sequencing**	**XML with mixed content**	**XML with children as attributes**	**XML with heads as attributes**
Avoid purely structural markup	No	Yes	Yes	Yes	Yes
Headedness	No	No	No	Yes	Yes
Explicit listing order	Yes	Yes	Yes	No	Yes
Non-unique child names	Yes	Yes	Yes	No	Yes
Inline markup	Yes	Yes	Yes	No	No
Easily machine-processable	Yes?	No?	No?	Yes?	Yes?
Human-friendly	No?	No?	No?	No?	No?

Table 4-2 A lexicographic scorecard for languages other than XML

	SGML	JSON	JSON with extensions	YAML	YAML with extensions	NVH
Avoid purely structural markup	Yes	No	Yes	No	Yes	Yes
Headedness	Yes	No	Yes	No	Yes	Yes
Explicit listing order	Yes	No	No	No	No	Yes
Non-unique child names	Yes	No	No	No	No	Yes
Inline markup	Yes	No	No	No	No	No?
Easily machine-processable	No?	Yes?	Yes?	Yes?	Yes?	Yes?
Human-friendly	No?	No?	No?	Yes?	Yes?	Yes?

4.3 Conclusion: notations matter

TBD.

BLOCK III: DATA MODELLING

5 Lexicographic data modelling

The previous block of chapters dealt with data languages such as XML and NVH: a relatively low level of representation. While the formal notation you choose for representing the contents of a dictionary matters, what matters even more is *what you do with it*: how you choose to model the various phenomena that occur in dictionaries. That is what the next block of chapters will be about: about modelling lexicographic content at a higher level of abstraction, independently of any notation.

5.1 Introduction: what are we modelling *for*?

When lexicography started digitising itself in the 1990s, the typical approach to data modelling was **text markup**: to demarcate where e.g. one sense ends and another begins. The purpose was to provide a level of abstraction above formatting, so that we can decide later whether e.g. headwords will be in bold or not, whether senses will be numbered or not. But in recent years we have been seeing the emergence of data models which are more ambitious than that: data models which are designed to help with **data management** (e.g. by guaranteeing that cross-references will always be valid), and data models which make dictionaries accessible for **alternative uses** than just showing entries to humans, such as various NLP tasks. This can be seen as a sign of maturity in the industry, as a step up from the relatively shallow, document-oriented digitisation of the 1990s towards a more deeply digitised, data-centric lexicography of the future.

5.2 Data-modelling standards in lexicography

This section provides an overview of the data modelling standards that are well known and widely used in lexicography today. We will be referring to them from multiple locations in the following chapters.

5.2.1 TEI (and TEI-Lex0)

The *Dictionaries* chapter of *Guidelines of the Text Encoding Initiative* (TEI Consortium 2007) is a popular XML-based encoding scheme for dictionaries, especially for retro-digitised ones. TEI-Lex0 (DARIAH 2021) is a customised version of the guidelines which imposes several additional constraints on it, in order to guarantee interoperability between dictionaries.

TBD: say more about how TEI is XMLy and tree-structured.

5.2.2 Lemon

Lexicon Model for Ontologies or *Lemon* (W3C 2016) is a scheme for representing dictionaries using Semantic Web technologies – which means that the data model is graph-based, not tree-based. The purpose of Lemon is not primarily to encode dictionaries for human consumption. Instead, Lemon's goal is to "provide linguistic grounding for ontologies": to provide information about how entities on the Semantic Web can be expressed in natural language. In other words, Lemon is mainly for machine-readable lexicons. That said, Lemon models information in a way which is similar to how human-oriented dictionaries model it: it works with objects such as entry and sense.

A lexicon in Lemon is a collection of objects of class `LexicalEntry`. Each entry has a headword (an object of class `Form`) and one or more senses (objects of class `LexicalSense`). Entries and senses can have other properties, such as part-of-speech labels, inflected forms, definitions and example sentences, but classes for these are not provided by Lemon: an implementor is free to choose any other vocabulary from the Semantic Web for representing these, for example the LexInfo ontology,[1] or even build their own.

Although the underlying metamodel of Lemon (and the entire Semantic Web) is a graph, dictionaries modelled in Lemon usually end up having a branching, tree-like structure like they do in XML.

TBD: say something about how *Wikidata Lexemes* uses Lemon.

[1] `http://lexinfo.net/`

TBD: say something about Lemon's new *Lexicography Module* (W3C 2019).

5.2.3 LMF

TBD: *Lexical Markup Framework* (ISO 24613-1:2024). Recently updated. Expressed in UML.

5.2.4 DMLex

The *Data Model for Lexicography* or DMLex (OASIS 2024) is being developed by the *Lexicographic Infrastructure Data Model and API* (LEXIDMA)[1] technical committee of OASIS, an organisation which oversees the development of open standards in the IT industry. LEXIDMA originated from the Europe-wide ELEXIS project.[2] The author of this thesis is the chair of LEXIDMA and DMLex's main author. DMLex uses a classical tree structure for the basic entries-and-senses skeleton of dictionary entries, and database-like relations for everything else. More details about DMLex will be given in Chapter 7.

5.2.5 Private schemas

TBD: in addition to the standards mentioned above, it is common practice for dictionary projects to be based on private, "home-baked" schemas. These tend to be expressed in XML and purely tree structured, in the fashion of TEI. A well-know example is DANTE (Convery et al. 2010, Rundell and Atkins 2011).

5.3 Design patterns in lexicography

Almost all the data models agree that the basic entries-and-senses hierarchy of an entry should be modelled as a tree structure, which is easily representable in every formalism including XML. A tree structure seems like a perfectly uncontroversial design pattern for this, with no obvious alternatives and no trade-offs.

[1] `https://www.oasis-open.org/committees/lexidma/`

[2] `https://elex.is/`

There is, however, some variation in how other, more complex phenomena are modelled. The rest of this block will be mainly about them. Chapter 6 will bring a thorough analysis of how hierarchies of subsenses and subentries are modelled in the various data models and will propose an alternarive, recursion-free design pattern which has been included in DMLex. Chapter 7 will them zoom in on DMLex and explain how it approaches other complex phenomena: how DMLex models entry-to-entry cross-refrences, how it models linguistic variation (such as variant spellings of a headword), how it deals with multi-word expressions, and others.

6 Avoiding recursion in the representation of subsenses and subentries

Recursion, and recursion-like design patterns, are used in the entry schemas of dictionaries to model subsenses and subentries. Recursion occurs when elements of a given type, such as `sense`, are allowed to contain elements of the same or similar type, such as `sense` or `subsense`. This chapter argues that recursion unnecessarily increases the computational complexity of entries, making dictionaries less easily processable by machines. The chapter will show how entry schemas can be simplified by re-engineering subsenses and subentries as relations (like in a relational database) such that we only have flat lists of senses and entries, while the *is-subsense-of* and *is-subentry-of* relations are encoded using pairs of unique identifiers. This design pattern losslessly records the same information as recursion (including – importantly – the listing order of items inside an entry) but decreases the complexity of the entry structure and makes dictionary entries more easily machine-processable.

6.1 Introduction

This section will introduce the concept of recursion in dictionary schemas and will set it in the wider context of dictionary encoding.

6.1.1 What is a dictionary schema

On a typical dictionary project, entries are encoded in XML (or some other formal notation), while the structure of the entries is controlled by a schema. A schema is a document which prescribes that, for example, each entry must begin with an `entry` element, that this element must contain exactly one `headword` element followed by one or more `sense` elements, that each `sense` element must contain zero or one `definition` element followed by zero or more `translation` elements, and so on. Schemas are usually expressed in machine-readable form such as DTD (Document Type Definition) and are used during the dictionary production process to guide human lexicographers in producing structurally correct entries.

Essentially, an entry schema defines two things: types and embedding constraints.

Types. A schema defines the existence of certain types of elements, such as `sense` and `definition`. Each element in each entry must be an instance of one such type.

Embedding constraints. A schema defines which elements can be contained or embedded inside which other elements, based their types. An embedding constraint consists of three facts:

- The fact that an embedding is allowed to occur, for example that an instance of `definition` is allowed to be embedded inside an instance of `sense`.
- The arity of the embedding, prescribing how many instances are allowed be embedded, for example zero or one `definition` inside a `sense`, one or more `sense` inside an `entry`.
- The listing order of the embedded elements, for example `headword` first and `sense` afterwards (and not the other way around) inside an `entry`.

6.1.2 Modelling dictionary entries as tree structures

The embedding of elements inside other elements creates a hierarchy of parent and child elements. A parent element contains zero, one or more child elements, and each child element can in turn be the parent element for further child elements. When such a hierarchy is encoded in XML, it becomes a formalisation of a mental model in which a dictionary entry is imagined as an (upside-down) tree. Modelling dictionary entries as tree structures has a long history in lexicography which pre-dates digitisation; a summary of this way of thinking is Wiegand (1989). When XML appeared on the scene in the 1990s, it became widely adopted for dictionary encoding because XML fits the tree-structured mental model nicely: for early thinking in the XML encoding of dictionaries see Ide et al. (2000) and Erjavec et al. (2000).

Most of the phenomena that occur in dictionaries lend themselves easily to being modelled as tree-structured hierarchies of parents and children. Indeed, the entire 'skeleton' of a typical dictionary entry, comprising headwords, senses, definitions and so on, is very obviously an (upside-down) tree. For such things, XML (or indeed any other tree-structured serialisation language, such as JSON or YAML) is a good and perfectly appropriate encoding formalism.

On the other hand, some phenomena such as cross-references (from one entry to another) are known to be difficult to implement in the tree-structure paradigm because, in a way, they 'reach out' of the tree and give rise to relations other than parent-child. To implement such relations in a dictionary writing system, some additional programming on top of the tree structure is usually required to enforce consistency (to avoid 'dangling' cross-references, that is, cross-references which go nowhere). This occasionally leads implementors to abandon the tree-structure model completely and to 'reimagine' the dictionary as a network or graph where relations other than parent-child are allowed to exist and where any element can be connected to any number of other elements, not just to its parent. An example of such an implementation is Maks et al. (2009). For a broader discussion of graphs versus trees in dictionary encoding see Měchura (2016).

So, in dictionary encoding, we have phenomena which can be modelled fully and losslessly by tree structures, and phenomena which cannot and for which a graph structure is more appropriate. In this article we are going to investigate something which is halfway between these two positions: the phenomenon of recursive embedding where there are senses inside senses or entries inside entries. These phenomena can be encoded in a tree-structure, so it is obviously not in the same category as cross-referencing discussed above. But the recursivity of the schema does cause computational complications, as we will discuss in the rest of this article.

6.1.3 Causes and types of recursion in dictionary schemas

One thing we often see in dictionary schemas is that they allow some form of recursive embedding, in other words, containing elements of one type inside elements of the same or similar type. Typical examples are **subsensing**

(a sense contains other, more specialised senses) and **subentrying** (such as when the entry for black hole is a subentry somewhere inside the entry for hole).

To allow recursive embedding in an entry schema is easy: one needs to design the schema in such a way that instances of – for example – `sense` are allowed to contain other instances of `sense`. When recursion is defined in this way, it can go on indefinitely: there is nothing stopping an embedded sense from containing yet more senses, and so on. In practice this potential is almost never exploited deeper than two or three levels, but the schema does allow it, at least theoretically.

Many schema designers find the prospect of potentially never-ending recursion worrying. So, another popular option, if one wants to allow recursive embedding in a schema, is something we might call *soft recursion*. In soft recursion, the schema defines two types, such as `sense` and `subsense`, and prescribes that instances of `sense` are allowed to contain instances of `subsense`, but instances of `subsense` are not allowed to contain further senses or subsenses. Apart from this, the types `sense` and `subsense` have the same or similar content models: they contain definitions, example sentences and so on. This is, strictly speaking, not recursion in the algorithmic sense, but it is similar to recursion, as it achieves the same effect, only with the guarantee of never running off beyond two levels of embedding.

In the rest of this article we will discuss how both types of recursion cause unnecessary complexity in dictionary schemas, making entries less easily processable by machines. The following section investigates the role of recursion in subsensing, and the next section after that will look at the role of recursion in subentrying.

6.2 Subsensing

This section looks at the practice of subsensing (= embedding senses inside other senses) in dictionary schemas. Listing 6-1 shows an example of an entry with subsensing: notice how one of the sense elements contains other sense elements. In this section we will analyse how dictionary schemas usually

enable subsensing by allowing recursion, and we will make an alternative proposal for encoding subsensing without recursion.

Listing 6-1 A dictionary entry with subsensing[1]

```
entry:
  headword: work
  pos: noun
  sense:
    definition: Activity involving effort...
    example: he was tired after a day's work
    sense:
      definition: Activity as a means of earning income; employment.
      example: I'm still looking for work
    sense:
      definition: The place where one is employed.
      example: I was returning home from work on a packed subway
    sense:
      definition: The period of time one spends in paid employment.
      example: he was going to the theatre after work
    sense:
      definition: A job.
      label: West Indian
      label: count noun
      example: I decided to get a work
  sense:
    definition: A task or tasks to be undertaken.
    ...
```

6.2.1 What are subsenses for?

Why do lexicographers sometimes decide to organise the senses of an entry into a hierarchical list of senses and subsenses, instead of a flat list of senses? Broadly, there are two kinds of motivation.

- **For modelling sense relations.** Deciding how to order and organise the senses inside an entry is a classical problem (or challenge) in lexicography. According to Kipfer (1983), an early writer on the subject, this is 'one of the most important decisions facing lexicographers'. Three broad strategies exist: (1) a chronological or historical ordering where senses are ordered according to how new or old they are in the language, (2) a usage-based or frequency-based ordering where most commonly

[1] Adapted from `https://www.lexico.com/definition/work`

used senses are listed first, and (3) something called 'logical' or 'analytical' ordering where senses are ordered and grouped according to how related they are to one another.

The purpose of the third type, the logical/analytical ordering, is to represent variations in semantic distance when certain senses of a polysemous headword are closely related (as often happens when it is an instance of regular polysemy – Apresjan 1974) while other senses of the same headword may be semantically more distant. The goal is to create a dictionary entry which 'flows' (can be read) as a coherent text, not just as a listing of mutually independent bullet points (Lew 2013, page 293). This approach typically implies a multilevel hierarchy of senses – 'tiered senses' according to Atkins and Rundell (2008, section 7.3.2) – where subsenses are allowed to be nested under a main sense. Entries that follow this strategy usually consist of a few broad senses, each of which contains a number of more detailed senses which are specialisations or metaphorical extensions of the main sense.

- **For navigating large entries.** When an entry contains so many senses that skimming through all of them at once is cognitively too demanding for the human user, the solution is to augment the entry with navigational aids (Lew 2013, page 295). Common navigational aids intended to help users locate the relevant sense in long polysemous entries include menus at the beginnings of entries (Atkins and Rundell 2008, section 7.2.1.3) and 'guidewords' (also called signposts, indicators or mini-definitions) at the beginnings of senses (Atkins and Rundell 2008, section 7.2.5).

  A hierarchical ordering of senses can also be understood as a navigational aid, similar to the hierarchical taxonomy of subject fields in a library catalogue or the hierarchical arrangement of topics in old-style internet directories such as Yahoo: the idea is that the user can iteratively 'zoom in' on the sense that interests him or her by moving from a manageable number of general top-level senses to more specialised senses deeper down the hierarchy.

These two motivations (modelling sense relations and providing a navigational tool) show that the presence of subsenses in dictionaries is well-motivated: the motivation is to meet the human user's needs and requirements. The mission of this article is not to argue against the existence of subsenses. Rather, this article argues against how subsenses are usually represented in machine-readable dictionaries, which is through recursion, and proposes a different, computationally simpler, method of representing subsenses.

6.2.2 What is wrong with recursive subsenses?

Dictionary schemas often implement subsensing either by defining a type (sense) and allowing recursion on it, or by defining two types (sense and subsense) and allowing soft recursion on them. Either way, we end up with dictionary entries in which the senses are not a flat list but a hierarchy.

Processing hierarchies computationally is more difficult than processing flat lists. In a hierarchy, the same kind of information (definitions, example sentences etc.) ends up being located at different depths inside the entry (starting from whatever the top-level element is) and possibly under parents of different names (sense or subsense). This is a distracting complication for digital agents (= software tools which process dictionary entries) such as dictionary writing systems or programs that extract data from dictionaries. The task the tool is given to do is usually complex enough in its own right, so it is no help if the recursive entry structure is making things even more difficult. From the perspective of someone who writes software tools for processing dictionary entries, it would be more convenient if each entry had only a flat list of senses with no embedding.

We have said that subsenses exist in order to satisfy genuine user needs: telling the user about sense relations and empowering the user to navigate long entries. But satisfying these needs through recursive data structures is an overly complicated solution. We are now going to present a technically simpler method which uses only a flat list of senses, without embedding.

6.2.3 The proposal

The method proposed here for encoding subsenses is based on a simple idea: the fact that one sense is a subsense of another sense does not need to be encoded by physically containing the one inside the other. The senses can be encoded as same-level siblings, and we can express the sense-subsense relationship between them by encoding it as a relation, like we do in relational databases. To get there, the following changes need to be made:

1. The list of senses is flattened, so that there is only a flat of senses, with no hierarchy.

2. Each sense is given a unique identifier (these can be generated automatically by the dictionary writing system and does not have to visible to the lexicographer).

3. In every location where one sense is supposed to appear as a subsense, we insert a placeholder instruction (`hasSubense`) which says 'take sense so-and-so and insert it here as a subsense'.

The result is shown in Listing 6-2. The identifiers created in step 1 begin with the hash character (e.g. `#work_noun_1`). Notice how the list of senses is flat, but the first sense contains placeholders with instructions for inserting subsenses there.

Listing 6-2 A dictionary entry where subsensing is treated as sense-to-sense relations

```
entry:
  headword: work
  pos: noun
  sense: #work_noun_1
    definition: Mental or physical effort done in order to...
    example: he was tired after a day's work
    hasSubsense: #work_noun_2
    hasSubsense: #work_noun_3
    hasSubsense: #work_noun_4
    hasSubsense: #work_noun_5
  sense: #work_noun_2
    definition: Activity as a means of earning income; employment.
    example: I'm still looking for work
  sense: #work_noun_3
    definition: The place where one is employed.
    example: I was returning home from work on a packed subway
  sense: #work_noun_4
    definition: The period of time one spends in paid employment.
    example: he was going to the theatre after work
  sense: #work_noun_5
    definition: A job.
    label: West Indian
    label: count noun
    example: I decided to get a work
  sense: #work_noun_6
    definition: A task or tasks to be undertaken.
    ...
```

The senses and subsenses are now encoded as a flat list, while their hierarchical arrangement needs to be inferred by following the chains of references between identifiers (#work_noun_1 etc). Notice that we have not lost any information in this transformation. We are still encoding the fact that a sense is a subsense of another sense, but we are doing it differently, without any form of recursion. Even the listing order of the subsenses is preserved in this new representation.

To show such an entry to human users, a software tool must first reconstruct the sense hierarchy from the ID-to-ID links, producing something like (1) again, and format that for display to the user.

Similarly, to present the entry to a lexicographer for editing, a software tool must first reconstruct the sense hierarchy from the ID-to-ID links, producing

something like (1) and then make it available for editing, for example in a conventional XML editor. When the lexicographer is finished with the entry, the tool must flatten the list of senses, give each sense an ID, indicate the sense-to-sense relations with ID-to-ID links, and store this flattened version.

In other words, the flattened version is an internal representation for machine processing, while the 'reconstructed' unflattened version, with its hierarchy of senses and subsenses, is for presentation to humans. In software engineering terms, the unflattened" version as in Listing 6-1 is the view model while the flattened version as in Listing 6-2 is the domain model of a dictionary entry. It is possible to convert losslessly between the two models. The unflattened model is more easily legible for humans while the flattened model is more easily processable by machines.

6.2.4 Realistic example: *sicher* in DWDS

Before we conclude our discussion of subsensing and move on to subentrying, let us look at a real-world example of an entry from a relatively modern, recently retro-digitised dictionary. We will show on this example how a multi-level hierarchy of senses, subsenses and 'subsubsenses' can be losslessly re-represented as a more easily machine-readable flat list of senses with relations between them, in accordance with our proposal.

The entry comes from a digitised version of Wörterbuch der deutschen Gegenwartssprache, published online as part of *Digitales Wörterbuch der deutschen Sprache* (DWDS). Some of the larger entries in this dictionary have a very 'branchy' hierarchy of senses and subsenses which goes down to more than two levels, and sicher is one of them. Only the first few senses from *sicher* are shown in Listing 6-3.

Listing 6-3 *sicher* in DWDS[1]

```
entry:
  headword: sicher
  pos: adj
  sense:
    definition: nicht von Gefahr bedroht, ungefährdet
    example: ein sicherer Weg
    sense:
      pattern: vor etw|jmdm sicher sein
      example: hier seid ihr vor der Entdeckung sicher
      sense:
        expression: sicher ist sicher!
        definition: lieber vorsichtig sein, lieber nichts riskieren!
        example: ich nehme den Regenschirm mit, sicher ist sicher!
    sense:
      expression: Nummer Sicher
      definition: Gefängnis
      example: in Nummer Sicher sitzen
  sense:
    definition: zuverlässig, verlässlich
    ...
```

If for the moment we ignore the fact that some of the subsenses are in fact subentries (a topic we will return to in the following section), then this sense hierarchy could be flattened as in Listing 6-4. We have ended up with an entry which, to a computer programmer at least, looks more easily processable, as there is only a flat list of senses, no hierarchy. It is now easier than before to write a program or script to iterate over the senses, for example to extract all definitions. The information about hierarchy is still there and can be used if needed, but it is encoded in such a way that it can be ignored when not needed.

[1] Adapted from `https://www.dwds.de/wb/sicher`

Listing 6-4 *sicher* with flattened senses

```
entry:
  headword: sicher
  pos: adj
  sense: #sicher_1
    definition: nicht von Gefahr bedroht, ungefährdet
    example: ein sicherer Weg
    hasSubsense: #sicher_2
    hasSubsense: #sicher_4
  sense: #sicher_2
    pattern: vor etw|jmdm sicher sein
    example: hier seid ihr vor der Entdeckung sicher
    hasSubsense: #sicher_3
  sense: #sicher_3
    expression: sicher ist sicher!
    definition: lieber vorsichtig sein, lieber nichts riskieren!
    example: ich nehme den Regenschirm mit, sicher ist sicher!
  sense: #sicher_4
    expression: Nummer Sicher
    definition: Gefängnis
    example: in Nummer Sicher sitzen
  sense: #sicher_5
    definition: zuverlässig, verlässlich
    ...
```

6.3 Subentrying

This section will deal with the phenomenon of subentrying where dictionary authors put entire entries (or things that look like entire entries) inside other entries. For the purposes of this article we will define subentry as any element inside a dictionary entry which has its own headword. When a subentry is present somewhere inside an entry, it overrides the entry's headword and provides its own.

6.3.1 What are subentries for?

Why do lexicographers sometimes decide to embed a subentry inside an entry? Broadly, there are two kinds of motivation.

- **To place multi-word expressions inside a single-word entry.** When a multiword expression appears inside an entry headed by a single-word headword, its appearance there may take many different forms.

One common practice is to make the multiword expression look like an ordinary sense, with a definition and a usage example, and nothing to distinguish it visually from other senses except the fact that it has its own (multiword) headword. Another option is for the multiword expressions to look like a 'mini-entry' located inside its host entry, with its own list of one or more senses. A multiword expression can be embedded inside one specific sense of its host entry, or it can be outside the sense hierarchy altogether, in a separate box at the end of the entry. The options are virtually endless: Atkins and Rundell (2008, 2page 54) gives a list of the most commonly used strategies. But regardless of how it is presented, whenever we see an element inside an entry which is headed by its own headword (even a multiword one), we can think of that element as a subentry.

- **To place secondary headwords and 'run-ons' inside the entry.** A secondary headword (Atkins and Rundell 2008, pages 492–94) is (typically) a single-word expression which is morphologically related to the main headword and which, in the lexicographer's opinion, needs to be be briefly described or at least mentioned in the dictionary, but does not deserve its own entry. An English example could be writing (as a noun) under write (as a verb) or run (as a noun) under run (as a verb). In other languages, secondary headwords that are sometimes seen in dictionaries are gender variants (German *Lehrerin* 'female teacher' under *Lehrer* 'teacher'), aspectual variants (Czech *přistávat* 'to be landing' under *přistát* 'to have landed') and others. Regardless of how such an element is presented or formatted inside the entry, we can think of it as a subentry: as something which, even though it is not an entry, has its own headword.

Every time a lexicographer decides to treat something as a subentry inside another entry (as opposed to treating it as its own independent entry), that decision is motivated: the lexicographer predicts that the user will benefit from seeing the subentry embedded in the host entry in its entirety, without having to navigate to a separate entry somewhere else in the dictionary. Therefore, the purpose of this article is not to argue against the existence of subentries in dictionaries. Instead, this article argues against how subentries

are usually represented in machine-readable dictionaries, which is through recursion, and proposes an alternative, computationally simplee, method.

Headword overriding

To understand why representing subsenses through recursion is problematic, it is necessary to understand the concept of headword overriding first. Normally, a dictionary entry is headed by a headword, and then the rest of the entry describes that headword. This seems logical and regular. But this logical and regular pattern is sometimes broken by things which **override** the headword.

A typical cause of overriding is the presence of a multiword subentry inside the entry. Its presence somewhere in the body of the entry changes the object of description: from that point onwards, we are no longer describing the headword we started with, we are now describing the multiword expression instead. In Listing 6-5, when we enter sense number 2, the object of description changes from the headword *sicher* to the multiword expression *sicher ist sicher!* and then, as we leave sense number 2, it changes back to the headword *sicher*.

Listing 6-5 Extract from the entry for *sicher* in DWDS[1]

```
entry:
  headword: sicher
  pos: adj
  sense:
    definition: nicht von Gefahr bedroht, ungefährdet
    example: ein sicherer Weg
  sense:
    expression: sicher ist sicher!
    definition: lieber vorsichtig sein, lieber nichts riskieren!
    example: ich nehme den Regenschirm mit, sicher ist sicher!
  sense:
    definition: zuverlässig, verlässlich
    example: ein sicherer Tresor
    ...
```

[1] Adapted from `https://www.dwds.de/wb/sicher`

Another frequent cause of overriding is when the object of description changes from the headword to a secondary headword (such as an inflected form, a variant form or a capitalised form). Listing 6-6 shows how the object of description changes from *bible* to *the Bible* as we enter sense number 1 and then it changes back to *bible* as we leave the sense.

Listing 6-6 The entry for *bible* in LDOCE[1]

```
entry:
  headword: bible
  pos: n
  sense:
    expression: the Bible
    definition: the holy book of the Christian religion
  sense:
    label: informal
    definition: the most useful and important book on a subject
    example: It's the anatomy student's bible!
```

6.3.2 What is and what is not overriding

Not all entry-internal elements which can be headed by something are necessarily overriding the headword. Consider the (changed) example from DWDS in Listing 6-7.

Listing 6-7 Extract from the entry for *sicher* in DWDS[2]

```
entry:
  headword: sicher
  pos: adj
  sense:
    definition: nicht von Gefahr bedroht, ungefährdet
    example: ein sicherer Weg
  sense:
    pattern: vor etw|jmdm sicher sein
    example: hier seid ihr vor der Entdeckung sicher
  sense:
    expression: sicher ist sicher!
    definition: lieber vorsichtig sein, lieber nichts riskieren!
    example: ich nehme den Regenschirm mit, sicher ist sicher!
```

[1] Adapted from `https://www.ldoceonline.com/dictionary/bible`

[2] Adapted from `https://www.dwds.de/wb/sicher`

Here, sense number 1 is an ordinary sense and sense number 3 is clearly an example of overriding. But what about sense number 2? It is headed by the grammatical pattern *vor etw|jmdm sicher sein* ('to be safe from sth|sb'). Here it is probably reasonable to argue that this sense element is describing (one sense of) the headword *sicher*: the object of description has not changed. The grammatical pattern is merely one of the properties of this sense of *sicher* that are being communicated to the user.

Another way to decide whether the head of an entry-internal element is or is not a subentry is to ask yourself whether it is likely that a dictionary user would search for it. Users might well input expressions such as *the Bible* or *sicher ist sicher* into the search box of an online dictionary, but probably not a grammatical pattern like 'to be safe from sth|sb'.

Another clue that can tell us whether we are dealing with an overridden headword or not is to ask whether it would be acceptable not to display the subentry inside the entry, and to provide a clickable hyperlink instead. In that hypothetical scenario, the user would be expected to click that hyperlink and this would take him or her to another screen where the subentry would be displayed. If this would be acceptable, then that is a clue that the subentry can function as an independent entry in its own right, its 'head' can function as a headword, and we are indeed dealing with an instance of headword overriding.

Either way, what ultimately decides the question (whether something is a subentry or not) is the intention of the lexicographer. In a well-designed dictionary schema the answer will be in the names and types of the elements used for encoding. In our examples here, a sense is a subentry if it has an expression, otherwise it is an ordinary sense.

6.3.3 What is wrong with overriding?

A human dictionary user, while he or she is skimming down a dictionary entry, is able to recognise when the object of description has changed and when it has changed back. Human dictionary users understand this easily from the way the dictionary entry is formatted on their screen and from

their knowledge of the language. This requires almost no extra effort for the human user.

As a software engineer, however, when building a program which will process the entries, having to deal with overriding (= with changes in what is being described) is an unwelcome complication. Recognising when overriding has occurred and when not requires some additional programming: the program must know that when it has entered a sense which has an expression then the object of description has changed to whatever this expression contains. It must also remember the previous object of description and know that, when it has left that particular sense, the object of description changes back.

So, even though headword overriding poses no problem for human dictionary users, it poses a real and existing complication for software and for people writing it. IT professionals would find it easier (and themselves more willing) to work with dictionary data if they could count on the fact that the object of description will never change inside an entry – in other words, that all senses inside one entry will be describing one and the same headword. Sadly, many dictionaries in existence today do not meet this assumption.

6.3.4 How dictionary schemas enable overriding

In a typical dictionary schema that allows entry overriding, there is usually an element somewhere in the schema which allows itself to be headed by something, to have something that resembles a headword. Having a headword is normally the privilege of entries, but the idea of headword overriding is that some elements inside entries have this privilege too.

In some dictionary schemas, the elements that are allowed to override the headword are ordinary senses. That is how it is in our two examples in Listings 6-6 and 6-7: the senses which act as ordinary senses and the senses which act as subentries are instances of the same type (sense). What turns a sense into a subentry is the presence of that one heading element (here named expression). Other dictionary schemas have a dedicated type for the elements that are allowed to override the headword, with a name such as subentry. All of these situations qualify as soft recursion. Last but not least, it is possible to enable headword overriding with proper (not soft) recursion, by allowing

instances of entry to contain other instances of entry, but this design pattern is not common.

In any case, regardless of the names of the elements and regardless of whether the recursion is soft or not, the problem is not caused by the existence of subentries as such. The problem is caused by the fact that the subentries are physically embedded inside other entries and that they override headwords there. In other words, the problem is caused by recursion.

6.3.5 The proposal

To get rid of recursion and to prevent headword overriding from happening, we must remodel subentries as relations. The following changes need to be made:

1. All entry-internal elements that override the headword (that is, all subentries) are taken out of the entries and promoted to the status of entries.

2. Each entry is given a unique identifier (these can be generated automatically by the dictionary writing system and can remain invisible to the lexicographer).

3. In every location where one entry is supposed to appear as a subentry, we insert a special instruction which says ‘take entry so-and-so and insert it here as a subentry’.

The result is shown in Listing 6-8. The identifiers created in step 1 begin with the hash character (e.g. `#the_bible`). Notice how the list of entries is flat, there are no subentries in the encoding. But subentries are still there implicitly, in the placeholders with instructions for inserting subentries.

Listing 6-8 A dictionary entry where subentrying is treated as entry-to-entry relations

```
entry: #bible
  headword: bible
  pos: n
  hasSubentry: #the_bible
  sense:
    label: informal
    definition: the most useful and important book on a subject
    example: It's the anatomy student's bible!
entry: #the_bible
  headword: the Bible
  sense:
    definition: the holy book of the Christian religion
```

Notice that we have not lost any information through the remodelling. We are still encoding the fact that an entry is a subentry of another entry, but we are doing it differently, without any form of recursion. Even the listing order of the subentries is preserved, so the subentry for *the Bible* has not lost its place as the first 'sense' of *bible*.

To show the entry for bible to end users, a software tool must first reconstruct the entry-subentry hierarchy from the ID-to-ID links, producing something like (6) again, and format that for display to the user. The same must be done when presenting the entry for editing to human lexicographers. The opposite must be done when saving the entry after editing: the composite entry needs to be decomposed into individual entries and any occurrence of subentries must be marked with placeholders (`hasSubentry`).

The decomposed version is suitable as an internal representation for storage and for machine processing because it does not use recursion and because headword overriding is guaranteed not to occur in it. The 'reconstructed' composite version, with its hierarchy of entries and subentries, is for presentation to humans. Once again, we see that the view model (the composed version) and the domain model (the decomposed version) are two sides of the same coin, with lossless conversion possible in both directions.

6.3.6 Realistic example: *sicher* in DWDS

For a more realistic example, let us return to the entry for *sicher* in DWDS. In Listing 6-4 we left it with its subsenses flattened and remodelled as relations. The only problem that remained with this entry is that senses 3 and 4 are subentries: they have their own headwords (encoded as `expression`) which override the headword of the main entry. We need to factor them out into their own entries, and link them to their original locations through `hasSubentry` relations. The result is in Listing 6-9.

Listing 6-9 *sicher* with decomposed subentries

```
entry:
  headword: sicher
  pos: adj
  sense: #sicher_1
    definition: nicht von Gefahr bedroht, ungefährdet
    example: ein sicherer Weg
    hasSubsense: #sicher_2
    hasSubentry: #sicher_4
  sense: #sicher_2
    pattern: vor etw|jmdm sicher sein
    example: hier seid ihr vor der Entdeckung sicher
    hasSubentry: #sicher_3
  sense: #sicher_5
    definition: zuverlässig, verlässlich
    ...
entry: #sicher_3
  headword: sicher ist sicher!
  sense:
    definition: lieber vorsichtig sein, lieber nichts riskieren!
    example: ich nehme den Regenschirm mit, sicher ist sicher!
entry: #sicher_4
  headword: Nummer Sicher
  sense:
    definition: Gefängnis
    example: in Nummer Sicher sitzen
```

6.4 Implementing the proposals

In this section we will discuss whether and how the design pattern proposed in this article can be implemented in various existing data models and encoding standards in lexicography. We will look at three formalisms: TEI-

Lex0, Lemon and DMLex. In each, we will investigate whether subsenses and subentries can be represented there in the recursion-free, relational fashion proposed here.

6.4.1 Subsenses and subentries in TEI-Lex0

The data model behind TEI-Lex0 is completely tree-structured. Subsensing is achieved by recursion: `<sense>` is allowed to contain other `<sense>` elements. Subentrying is achieved by recursion along with headword overriding: both `<entry>` and `<sense>` are allowed to contain other `<entry>` elements, and `<sense>` is allowed to contain `<form type="lemma">` (in other words: senses can have their own headwords). Listing 6-10 shows what the entry for *sicher*, which we know from Listing 6-3, might look like in TEI-Lex0.

Just because TEI-Lex0 allows recursion and headword overriding does not mean that we have to use it. It is possible in TEL-Lex0 to reduce sense hierarchies into flat lists and to refactor subentries into separate entries. The only thing missing in TEI-Lex0 is XML equivalents for instructions such as `hasSubsense` and `hasSubentry`, which we know from Listings 6-4 and 6-9. If such XML elements existed in TEI-Lex0, then a non-recursive encoding of the same content would like Listing 6-11 – notice occurrences of `<hasSubsense>` and `<hasSubentry>` there.

Another option, instead of inventing new XML elements, would be to avail of TEI's existing mechanism for cross-references which is based on `<xr>` and `<ref>` elements, as Listing 6-12 shows. This would entail inventing a new value for the `@type` attribute of `<xr>`, which is invalid in TEI-Lex0 (the legal values of that attribute are a closed list). It would also deviate from the intended semantics of `<xr>` which is navigation (providing clickable links), not transclusion (embedding content taken from elsewhere).

The verdict is that although TEI-Lex0 does not support recursion-free encoding of subsenses and subentries, introducing the necessary features into it (for example in an in-house implementation) would be relatively trivial.

The consequence of encoding subsenses and subentries this way would be that, every time a software agent is about to show the entry for *sicher* to a

human user, it needs to reconstruct the hierarchy of subsenses and subentries first. The upside of that trade-off is that the dictionary would be encoded in a way which is free of recursion and headword overriding, therefore computationally simpler and more easily processable for purposes other than displaying entries to humans.

Listing 6-10 *sicher* encoded in TEI-Lex0

```
<entry xml:id="sicher" xml:lang="de">
  <form type="lemma"><orth>sicher</orth></form>
  <gramGrp><gram type="pos">adj</gram></gramGrp>
  <sense xml:id="sicher_1">
    <def>nicht von Gefahr bedroht, ungefährdet</def>
    <cit type="example"><quote>ein sicherer Weg</quote></cit>
    <sense xml:id="sicher_2">
      <form type="pattern"><orth>vor etw|jmdm sicher sein</orth></form>
      <cit type="example"><quote>hier seid ihr vor der Entdeckung sicher</quote></cit>
      <sense xml:id="sicher_3">
        <form type="expression"><orth>sicher ist sicher!</orth></form>
        <def>lieber vorsichtig sein, lieber nichts riskieren!</def>
        <cit type="example"><quote>ich nehme den Regenschirm mit, sicher ist sicher!</quote></cit>
      </sense>
    </sense>
    <sense xml:id="sicher_4">
      <form type="expression"><orth>Nummer Sicher</orth></form>
      <def>Gefängnis</def>
      <cit type="example"><quote>in Nummer Sicher sitzen</quote></cit>
    </sense>
  </sense>
  <sense xml:id="sicher_5">
    <def>zuverlässig, verlässlich</def>
    ...
  </sense>
</entry>
```

Listing 6-11 *sicher* encoded in a (hypothetical) relational variant of TEI-Lex0

```
<entry xml:id="sicher" xml:lang="de">
  <form type="lemma"><orth>sicher</orth></form>
  <gramGrp><gram type="pos">adj</gram></gramGrp>
  <sense xml:id="sicher_1">
    <def>nicht von Gefahr bedroht, ungefährdet</def>
    <cit type="example"><quote>ein sicherer Weg</quote></cit>
    <hasSubsense sense="#sense_2"/>
    <hasSubentry entry="#sense_4"/>
  </sense>
  <sense xml:id="sicher_2">
    <form type="pattern"><orth>vor etw|jmdm sicher sein</orth></form>
    <cit type="example"><quote>hier seid ihr vor der Entdeckung sicher</quote></cit>
    <hasSubentry entry="#sense_3"/>
  </sense>
  <sense xml:id="sicher_5">
    <def>zuverlässig, verlässlich</def>
    ...
  </sense>
</entry>
<entry xml:id="sicher_3" xml:lang="de">
  <form type="expression"><orth>sicher ist sicher!</orth></form>
  <sense xml:id="sicher_3_1">
    <def>lieber vorsichtig sein, lieber nichts riskieren!</def>
    <cit type="example"><quote>ich nehme den Regenschirm mit, sicher ist sicher!</quote></cit>
  </sense>
</entry>
<entry xml:id="sicher_4" xml:lang="de">
  <form type="expression"><orth>Nummer Sicher</orth></form>
  <sense xml:id="sicher_4_1">
    <def>Gefängnis</def>
    <cit type="example"><quote>in Nummer Sicher sitzen</quote></cit>
  </sense>
</entry>
```

Listing 6-12 An alternative hypothetical encoding of relations in TEI-Lex0

```
<xr type="hasSubsense">
  <ref>#sense_2</ref>
</xr>
```

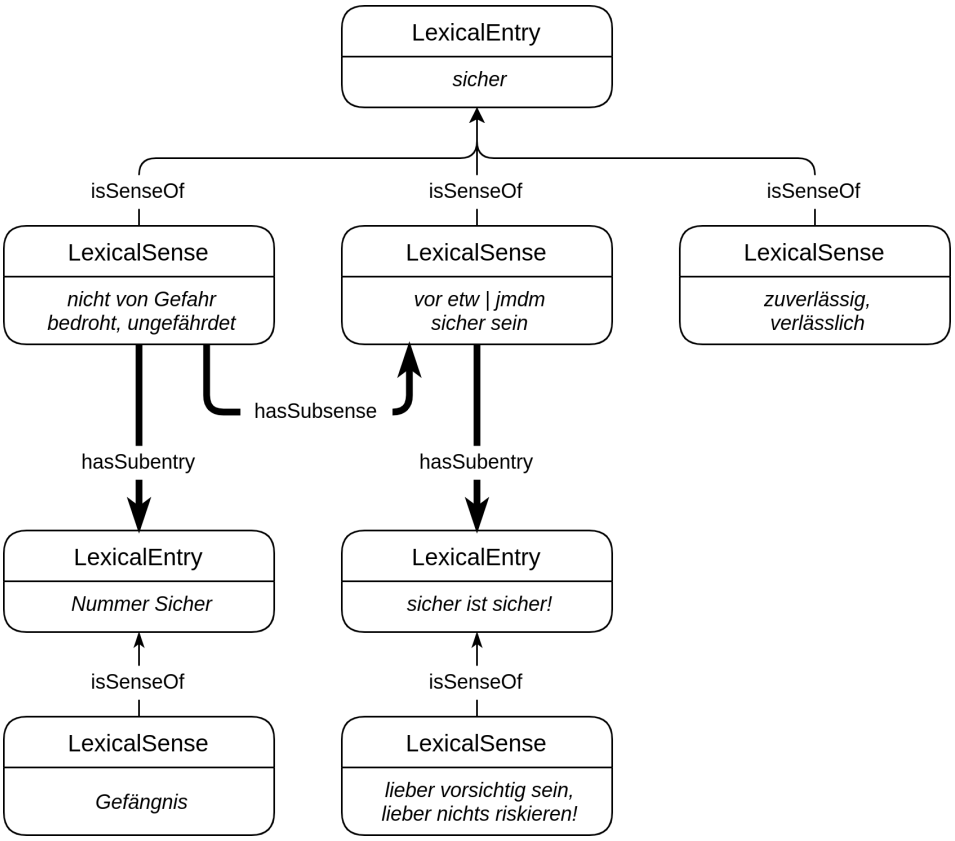
6.4.2 Subsenses and subentries in Lemon

Although the underlying metamodel of Lemon (and the entire Semantic Web) is a graph, dictionaries modelled in Lemon usually end up having a branching, tree-like structure like they do in XML.

Unlike TEI-Lex0, Lemon is very strict about how 'branchy' the trees are allowed to be. Recursive embedding is prohibited in Lemon: senses cannot have subsenses and entries cannot have subentries. Headword overriding is prohibited too, as senses are not allowed to have their own headwords. This makes Lemon computationally simpler than TEI-Lex0, but also less expressive. In fact, there is no way to represent subentries or subsenses in Lemon: these notions simply do not exist in Lemon. Lemon provides no facilities for declaring that, for example, one sense of a given entry is a subsense of another sense of the same entry. This is not surprising, given that subsensing and subentrying exist mainly to satisfy human users' needs (as discussed in Sections 6.2.1 and 6.3.1) while Lemon is intended for machine applications.

It would seem, then, that Lemon has the opposite problem from TEI-Lex0, as it cannot represent subsenses and subentries at all. But, given than Lemon is part of the Semantic Web ecosystem, it is relatively easy to extend Lemon with custom classes needed to represent the necessary links between senses and entries. Figure 6-1 shows how that could be done: thin lines and objects in thin outline are classes and properties provided by Lemon, while thick lines are properties provided by someone's own hypothetical extension to Lemon.

Figure 6-1 *sicher* modelled in Lemon, with extensions (thick lines) allowing the modelling of subsenses and subentries



The verdict is that although Lemon does not support any representation of subsenses and subentries, it can be extended to support the recursion-free, relational representation proposed in this article.

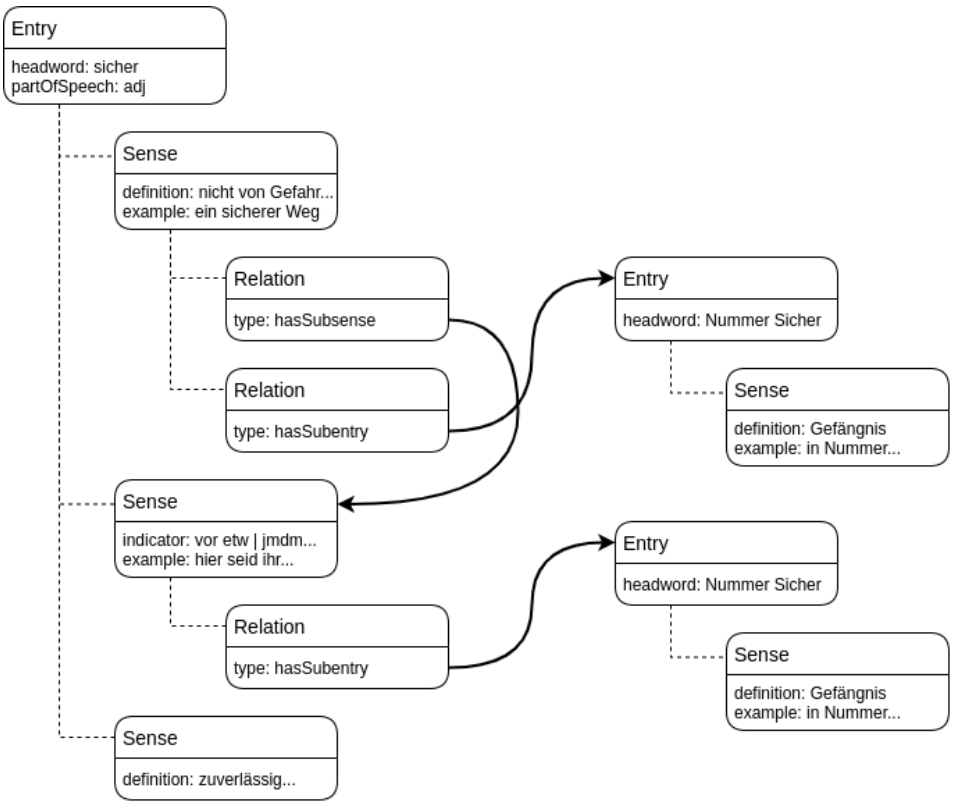
6.4.3 Subsenses and subentries in DMLex

TEI-Lex0 is very expressive in its ability to represent a wide range of lexicographic content types and their configurations – including subsenses and subentries – but the downside is its computational complexity caused by recursion and by headword overriding. Lemon, on the other hand, is computationally simple (there is no recursion and no headword overriding) but the downside is low expressivity, including crucially its inability to represent subsenses and subentries. Each of these two formalisms meets one

goal at the expense of the other. We will now look at a third formalism which aims to meet both goals at once: to be highly expressive and computationally simple.

DMLex does not allow recursive embedding or headword overriding in its tree structures: entries contain only flat lists of senses, and senses are not allowed to have their own headwords. But DMLex also defines a data type called *relation* which can be used to represent all kinds of relations which 'reach out' of the classical tree structure including synonymy, antonymy and variation. Crucially, this data type can also be used to model subsensing and subentrying: Figure 6-2 shows an example.

Figure 6-2 *sicher* modelled in DMLex



DMLex is similar to Lemon in that it is a hybrid between a tree-structured metamodel (for the entries-and-senses skeleton) and a graph-structured metamodel (for everything else). But DMLex differs from Lemon in that it

provides a richer inventory of data types for lexicographic content (definitions, example sentences etc.) and that it preserves the listing order of senses, subsenses and subentries. DMLex is therefore a faithful implementation of the design pattern proposed in this article: it prohibits recursive embedding, prohibits headword overriding, and represents subsenses and subentries relationally.

6.5 Conclusion

This chapter has made proposals for a relational remodelling of subsenses and subentries in dictionary schemas: things that were previously modelled as instances of recursive embedding are remodelled here as relations between non-recursive objects. What is the significance of this change?

6.5.1 A new design pattern

The goal of this chapter has been to propose an alternative design pattern for subsenses and subentries in machine-readable dictionaries. In software engineering, a design pattern – a term made popular by Gamma et al. (1995) – is an abstract and informal 'template' or 'recipe' for approaching a given software engineering challenge. The most popular design pattern employed for solving the challenge of subsenses and subentries today is recursive embedding (of senses inside senses, etc.). We have argued that this design pattern causes unnecessary complications further downstream and is therefore undesirable, and we have proposed an alternative design pattern, one based on treating subsenses and subentries as database-like relations. We have shown that this design pattern losslessly represents the same information but is less complex computationally.

A design pattern is not a complete data model: it is not something which can model all kinds of data in a given domain, or something which can be converted into computer code straight away. Consequently, we have not proposed a complete data model for dictionaries here. We have, however, shown how the recursion-free design pattern can be implemented in existing tree-structured encoding schemas without having to re-engineer them completely.

Should other things besides subsenses and subentries be re-engineered completely into relations? There is some evidence that cross-references, sense-to-sense links and other relationships which 'reach out' of the tree structure would benefit from such a move – refer back to the discussion in 6.1.2. On the other hand, for the basic entries-and-senses skeleton, a tree structure seems perfectly appropriate and re-engineering it into something else would be introduce unnecessary complications of its own. The ideal metamodel for dictionaries is probably hybrid: half tree-structured, half graph-structured.

6.5.2 Advantages and disadvantages

What do we gain and what do we lose when we transition from recursive embedding to relations? When subsenses and subentries are encoded through recursion (including soft recursion), they are encoded in a way which is **easily legible for humans** but **less easily processable by machines**. When we remodel subsenses and subentries into relations, then the balance is reverted: the entries are now **less easily legible for humans** but **more easily processable by machines**. As is often the case when recording information on computers, we can do things either in a way which is human-readable, or in a way which is computationally efficient, but we cannot have both.

Until now, the trend in lexicography has been to resolve such trade-offs in favour of human readability. Dictionaries were expected to be encoded in a structure which is close to their eventual appearance on screen and on paper. Showing the dictionary entry to a human user was the one and only imaginable culmination of the entire lexicographic effort. This is the lexicographic industry's traditional use-case. Those who wanted to use dictionaries for other use-cases, for example to extract data from them for an NLP application or to link several dictionaries together, were forced to deal with a complexity which, from their point of view, was unnecessary.

The proposal presented here resolves the trade-off the other way around, in favour of machine processability. The data models end up being simpler because there are fewer types in them (there is no need for types such as subsense and expression). The entries themselves end up being encoded in flat, non-recursive data structures where the same information can always be

found at the same depth and where there is no headword overriding. So, using a dictionary for purposes other than showing them to humans has become easier. On the other hand, because the encoding has become less legible for humans, the dictionary now requires additional processing before it can be shown to a human user (to unflatten and 'reconstruct' the hierarchies of subsenses and subentries).

For any dictionary project with ambitions beyond the traditional lexicographic use-case, the relational approach proposed here is the better option: it widens the potential audience, the advantages outweigh the disadvantages. A wider range of consumers are getting a chance to consume the data. The barriers to entry are lower, dictionaries encoded like this are now more easily understandable to IT professionals working not only in lexicography but also in disciplines such as information retrieval, digital humanities, computational linguistics, natural language processing and so on. Through relational remodelling of recursive structures, we can bring dictionaries to audiences outside the traditional consumers of lexicographic content.

7 On the design of DMLex

DMLex (*Data Model for Lexicography*, OASIS 2024) was mentioned several times in the previous chapters as one of the data modelling standards in lexicography. But DMLex is special among its peers because it offers a radically different approach to modelling many lexicographic phenomena from other standards. It is now time to turn to DMLex in more detail.

7.1 The thinking behind DMLex

The institutional background of DMLex has been described in Section 5.2.5. Let us only repeat here that DMLex is being created by LEXIDMA (*Lexicographic Infrastructure Data Model and API*), an OASIS technical commitee. The author of this thesis is the chair of LEXIDMA, and so it is not surprising that DMLex is shaped by the author's ideas about how data should modelled in lexicography. This section will outline what those ideas are, and the next section after it will show how DMLex approaches the modelling of specific phenomena.

7.1.1 Does the world need another lexicographic data standard?

Practically all existing data standards in lexicography (reviewed in Section 5.2), and practically all private schemas typically used on dictionary projects, have one thing in common: they model dictionary entries as tree structures. A tree structure is the metamodel behind XML: every XML document is basically an upside-down tree where elements branch into other elements which branch into even more elements. At first sight, this seems like a good fit for dictionary entries with their hierarchical arrangement of entries which contain senses which contain definitions and so on. But tree structures have disadvantages too. For example, they are inefficient at representing cross-references between entries (where the requirement is to prevent invalid cross-references to non-existent targets). Graph structures (such as relational databases and RDF graphs) are more efficient at modelling this. Another limitation of tree structures is that they force lexicographers into having to make uncomfortable decisions about the placement of multi-word subentries:

is it really necessary to decide whether *black hole* goes under *black* or *hole*? Can it not be under both? This is not possible in a tree structure (at least not easily, not out-of-the-box); graph structures better suited to modelling such situations.

The goal of LEXIDMA is to depart from the limitations of tree-structured data models and to propose a new approach to modelling dictionaries: one which is a hybrid between tree structures for the basic entries-and-senses hierarchy, and graph structures for everything else. One way to understand DMLex is as a catalog of design patterns which dictionary designer could take inspiration from if, for example, they are building a dictionary writing system and if their ambition is to handle the more complex phenomena – cross-references, multi-word subentries and similar – in a more intelligent way than before.

7.1.2 What is a data model?

Practicioners in lexicography are familiar with dictionary entries being encoded in XML, and with the internal structure of these entries being constrained by an entry schema (on dictionary projects this is typically a DTD). An entry schema is something which expresses the fact that, for example, an entry is supposed to contain exactly one headword followed by a list of one or more senses, that each sense is supposed to contain at most one definition, and so on.

A data model is like a schema, but more abstract. An XML schema is very closely coupled with the notation of XML. A data model, on the other hand, is not coupled with any specific notation or formalism. A data model is one level of abstraction above that: it is something which can be expressed in many notations and formalisms. DMLex is such a data model. It defines the structure of dictionaries (called 'lexicographic resources' in DMLex) in a way which can be "serialised" into many different formalisms. The DMLex standard comes with serialisations of itself into XML, into JSON, into a relational database, into an RDF graph, and last but not least into NVH.

7.1.3 What kind of data model is DMLex?

TBD: Conceptual, logical...

7.1.4 The metamodel behind DMLex

Every data model assumes a certain metamodel, a collection of axiomatic primitives the model is composed of. In XML, every model is a tree of *nodes* (of different kinds: elements, attributes, text nodes and others). In Entity Relationhip (ER) modelling, each model "pictures the world in terms of entities that have attributes and participate in relationships" (Halpin and Morgan 2008 page 8). UML class diagrams analyse everything in terms of classes which encapsulate data and behaviour, and Object-Role Modelling (ORM) "views the world in terms of objects playing roles in relationships" (Halpin and Morgan 2008 page 9). And DMLex?

DMLex sees each dictionary as a collection of objects of certain **types** such as `entry`, `sense`, `definition` – DMLex defines which types exist. In diagrams in this chapter, the type of each object is given in a shaded heading at the top (and sometimes at the bottom) of the box. The rest of the box contains the object's **properties**: DMLex defines which properties an object can have based in its type, and what their arities are (at least one, one or more, etc.). A property has a name (such as `headword`, `definition`, `listingOrder`) and a value. In diagrams, the property's name comes before the colon and the value after it, such as `headword: colour`. The value of a property can be either literal, typically a short string of text or a number, or a reference to another object. In diagrams, references are indicated as arrows which point from the property to the object the property refers to.

7.1.5 Relations, relations everywhere

Most of the object types defined by DMLex are straightforward equivalents of the kinds of content that have existed in dictionaries for centuries – `entry`, `sense`, `definition` and so on – and their instances refer to one another in a way which resembles a tree structure. There is one object type, however, which will seem unfamiliar to lexicographers: `relation`. DMLex uses `relation` objects to represent things that are difficult to represent satisfactorily in a tree structure, such as cross-references, and for things that *are* representable in a tree structure but the representations would be computationally complex.

Each `relation` object has a property named `type`. Every DMLex implementor is free to define the relation types that exist in their dictionary and how their software is supposed to handle them (for example when showing entries to humans). Then, each `relation` object has at least two members or, more accurately, refers to at least two `member` objects. Each `member` object contains a reference (in its `ref` property) to something in the same dictionary such as an entry or a sense, and optionally another property named `role` which specifies the role of this member in the relation. The implementor is free to define which roles exist for which relation types.

This is the mechanics DMLex makes available to implementors for representing relations between entries and/or senses. The rest of this chapter shows many example of how DMLex makes use of relations for various purposes.

7.2 How DMLex models selected phenonena

The rest of this chapter will be a gallery of how DMLex represents various phenomena that arise when planning the structure of a dictionary. The DMLex specification is a very long document and it is not possible to show everything in this thesis, but the phenomena chosen for demonstration here will hopefully suffice to give the reader an idea of how DMLex works. In each subsection, I will show the abstract structure using diagrams with boxes and lines between them. Each such diagram can be serialized into XML, JSON, NVH, a relational database and an RDF graph – DMLex defines how, but I will not be showing those serializations here, for the same reason of keeping the thesis within a reasnable length.

7.2.1 The basics: entries and senses

How does DMLex represent the basic skeleton of dictionary entries as things that are headed by headwords and then subdivided into senses? This is the only area where DMLex is relatively conservative and models this as a tree structure. Figure 7-1 shows a simple dictionary entry when viewed by a human user and Figure 7-2 shows how the same entry would be modelled in DMLex.

Figure 7-1 A simple dictionary entry rendered for human viewers

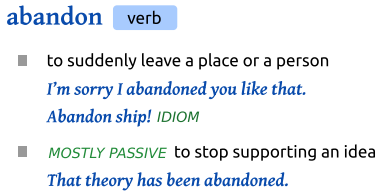
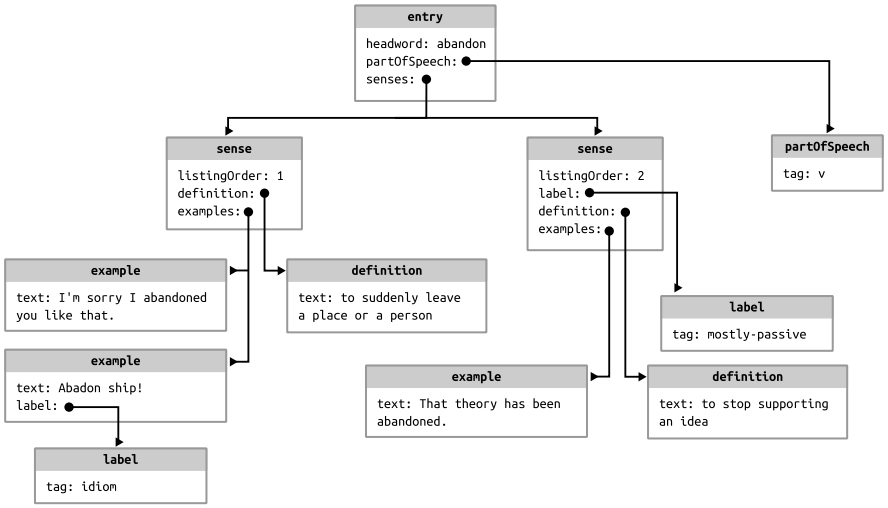


Figure 7-2 A simple dictionary entry modelled in DMLex



DMLex can be used for representing monolingual dictionaries like above, but it can also represent bilingual dictinaries like in Figures 7-3 and 7-4. DMLex can also represent *multilingual* dictionaries: those are dictionaries which have one source language and multiple target languages.

Figure 7-3 A simple bilingual entry

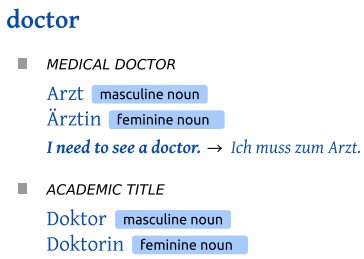
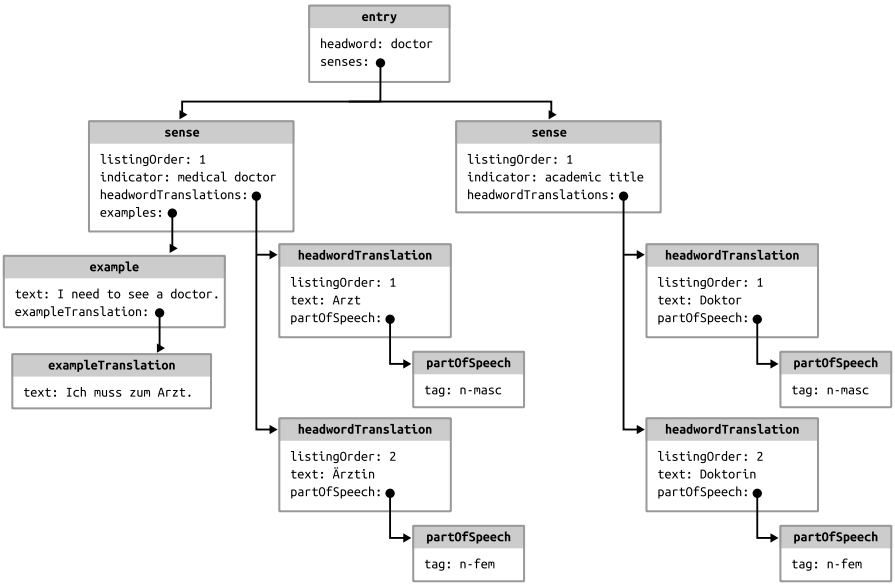


Figure 7-4 A simple bilingual entry modelled in DMLex



7.2.2 Cross-references

One very obvious use for DMLex's `relation` objects is modelling cross-references from entries to other entries. In classical XML-based lexicography, a cross-reference is similar to a hyperlink on the web: it is something which can be navigated, typically by clicking, to go from (some location in) one entry to (some location in) another entry. The way a cross-reference is typically represented in XML and other tree-structured models has two problems. First, it does not guarantee that the target of the cross-reference actually

exists. Second, it does not guarantee that the target contains a reciprocal cross-reference in the opposite direction (if reciprocity is expected).

DMLex solves this problem by redefining cross-references as relations. A good way to understand how this is different from the usual approach is to realise that DMLex does not actually model the cross-references, it models the relations that motivate them. The model states that there is a relation of a certain type between two or more entries which may or may not be shown to end-users as clickable cross-references, depending on the relation's type and on how the software is programmed to handle it.

Figure 7-5 shows an example. There are three entries for the headwords *lens*, *glasses* and *microscope*, with one sense each. The model uses relations to express the fact that there is a meronymy (part-whole) relation between *lens* and *glasses*, and also between *lens* and *microscope*. Notice that there are two relations of type `meronymy`: one between *lens* and *glasses* and another between *lens* and *microscope*. Each has two members pointing to the relevant senses, and each member has a `role` property to tell us which end of the relation it represents, the part or the whole. Figure 7-6 shows how these relations can be rendered as cross-references at the end of the appropriate senses of the appropriate entries.

Figure 7-5 Entries for *lens*, *glasses* and *microscope* modelled in DMLex

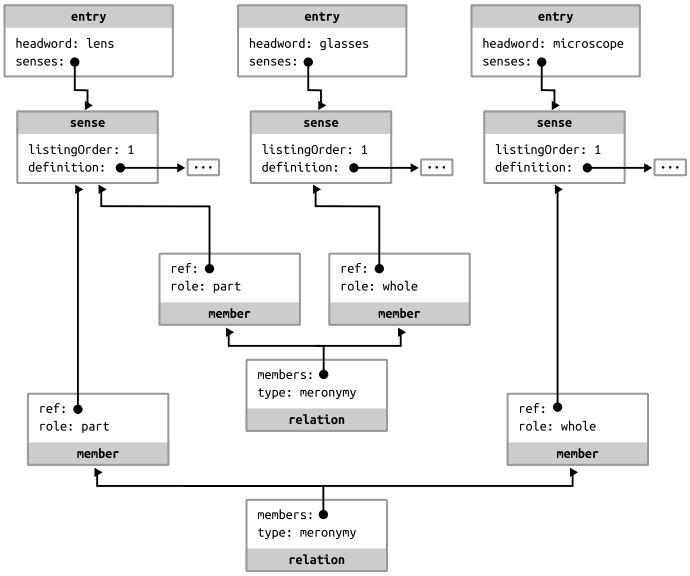


Figure 7-6 Part-whole relations rendered as cross-references

lens

- curved glass that makes things seem bigger
  - » **things that contain** lens
    glasses microscope

glasses

- an optical seeing aid
  - » **things contained in** glasses
    lens

microscope

- equipment for looking at very small things
  - » **things contained in** microscope
    lens

In addition to this, DMLex also gives the implementor all the mechanics needed to define that `meronymy` is one of the valid relation types in this dictionary, and that each relation of this type is supposed to have exactly two members, one with `role: part` and one with `role: whole`, each referring to a sense somewhere in the dictionary. These constraints are not shown in Figure

7-5 but can be expressed in DMLex using objects of types `relationType` and `memberType`.

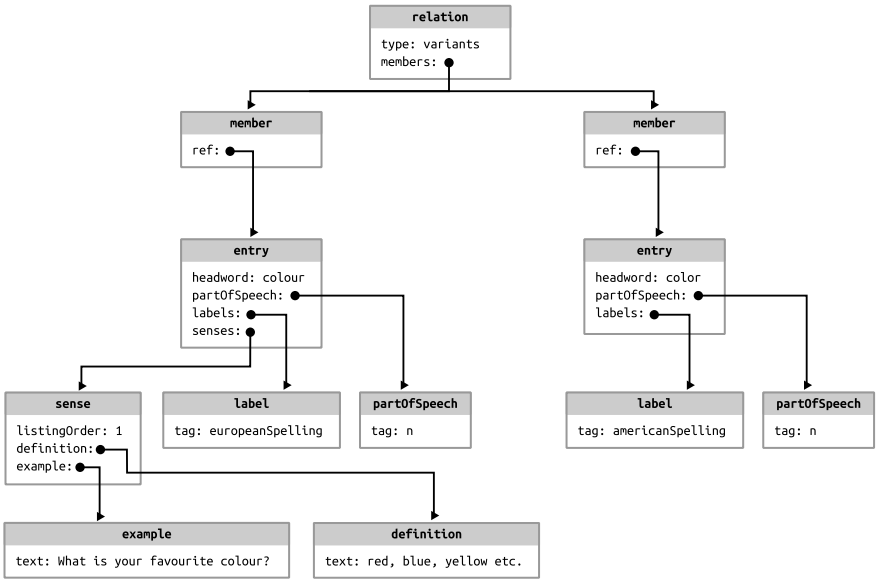
7.2.3 Multiple headwords per entry

The usual situation in dictionaries is that each entry is headed by a single headword. Occasionally, however, lexicographers produce entries headed by several headwords or headword-like things at the same time. This happens when lexicographers are describing spelling variants (*colour* and *color*), gender-paired nouns (German *Lehrer* 'male teacher' and *Lehrerin* 'female teacher'), aspect-paired verbs (Czech *přistávat* 'to be landing' and *přistát* 'to have landed') and so on. Figute 7-5 illustrates what such an entry typically looks like.

Figure 7-7 An entry with multiple headwords

DMLex, on other hand, is very strict about allowing only one headword per entry. That does not mean, however, that things like *colour* and *color* cannot be represented in DMLex. They can, but it has to be done differently. In DMLex, a separate entry has to be created to for each, and then the two entries must be connected with an object of type `relation`. Figure 7-8 shows an example.

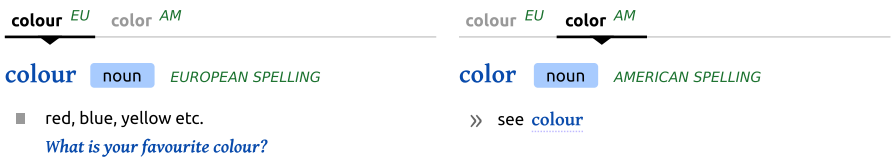
Figure 7-8 Entries for *colour* and *color* modelled in DMLex



Notice how the entry for *colour* contains a lot of information (a part-of-speech label and a sense) while the entry for *color* is sparse and skeletal: its purpose is only to serve as a member in the `relation` object. DMLex allows entries with no senses to exist for this purpose.

This is how the data is represented in DMLex internally, but it is not necessarily what a human dictionary user would see on their screen (or on a printed page). When a software agent (such as a website or a mobile app) is about to show one of these entries to a human user, it needs to follow through on all the `relation` objects and compose a "view" of the entry from them. Figure 7-9 shows a suggestion of what the entries for *colour* and *color* might look like when displayed on someone's screen. Another option is to merge the two entries into one "virtual" entry at display-time, effectively producing the same output as in Figure 7-7. Either way, it is up to the implementor to decide how their software will handle entries connected through these relations.

Figure 7-9 Suggested display of colour and color



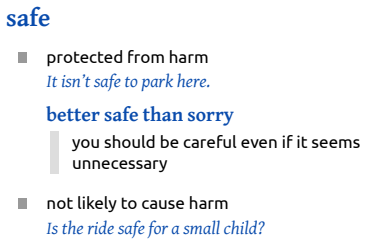
This may seem more complicated than necessary at first. Why have we, the authors of DMLex, decided to model things this way instead of simply allowing entries to have multiple headwords? The motivation for choosing this design pattern is **computational simplicity**. Multi-headword entries, while easily understandable for human dictionary users, cause inconvenient complications for automated processing, for example when sorting entries alphabetically or when matching dictionary entries to words in a text. These computational tasks become simpler if we are able to guarantee that each entry will always have exactly one headword. When designing DMLex, we decided to favour computational simplicity, even if it means that rendering entries for human viewers is now a more complex process.

This decision also means that the data model is simpler than it would be if it did allow multi-headwording: there are fewer object types and fewer properties in the model. There is no need separate object types and properties for variant headwords, secondary headwords and similar: everything is simply a headword (of its own entry). Also, one thing sometimes seen in multi-headword entries (in dictionaries that allow them) is that each co-headword almost has its own "mini-entry" describing its properties: labels like 'American spelling' as well as other things. This complicates the data model again because it becomes non-trivial to distinguish between information which applies to only one of the co-headwords and information which applies to the whole entry. DMLex eliminates these complications by imposing the restriction that each entry must have exactly one headword. For more complicated scenarios, multiple entries have to created (even if some of them will be "empty") and connected through `relation` objects.

7.2.4 Placement of multi-word subentries

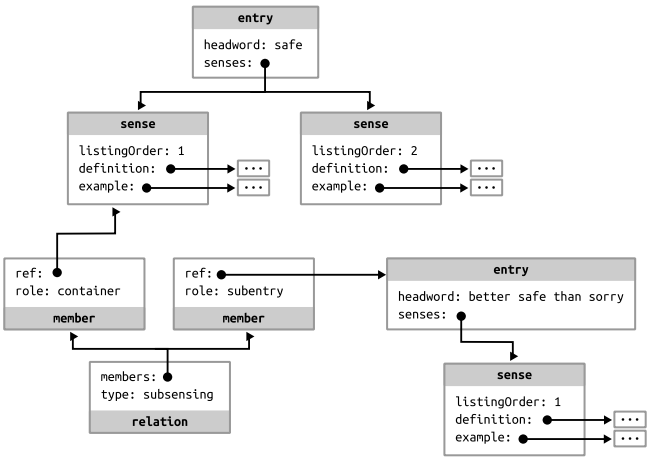
It is not unusual for dictionary entries to contain embedded subentries for multi-word items. Figure 7-10 shows a dictionary entry headed by the headword *safe* which contains a subentry headed by the multi-word expression *better safe than sorry*.

Figure 7-10 An entry with a subentry



In Chapter 6 we discussed extensively how most dictionary schemas represent subentries by allowing some form of recursion (such that entries are allowed to contain other entries or entry-like things), and how this increases the computational complexity of dictionaries. This is why DMLex does not allow entries to be embedded inside entries. In DMLex, the multi-word subentry needs to be represented by its own entry which can be linked to its "mother entry" through a `relation` object. Figure 7-11 shows what that looks like when modelled in DMLex. The relation effectively says "please take the entry for *better safe than sorry* and place it under the first sense of *safe* when rendering the entry for *safe* for human end-users" – provided the software agent has been programmed to understand relations of `type: subentrying` this way.

Figure 7-11 An entry with a subentry modelled in DMLex



One highly beneficial consequence of this design pattern is that lexicographers no longer have to decide whether a multiword subentry such as *better safe than sorry* should be placed under *safe* or *sorry*. It can be under both, we simply need to create two relations, one between *better safe than sorry* and *safe*, and another between *better safe than sorry* and *sorry*.

Traditionally, deciding on the placement of multi-word items has been a perennial problem in lexicography (Bogaards 1990). In many cases the ideal solution would be to include a multiword subentry under several headwords, but this is difficult to accommodate in a tree-structured data model. The only way to include a multi-word unit in more than one entry is to duplicate it, but this is an inelegant solution. Most importantly, it opens up the potential for inconsistency: if a lexicographer makes a change to the subentry *better safe than sorry* under *safe*, there is no automatic way to propagate the change to the other copy under *sorry*.

One method to deal with this, which as been becoming increasingly popular in born-digital dictionaries, is to treat multi-word units as independent entries, in effect promoting them to the same level as single-word headwords. This approach 'solves' the problem of multiword item placement by deciding not to place them anywhere, and that is also its drawback: it strips the lexicographer of the ability to include a multiword item like *better safe than*

sorry under specific senses of the single-word entries. Instead, it delegates the placement question to the wwebsite's search algorithm, hoping that *better safe than sorry* will indeed appear somewhere on the user's screen when the user has looked up *safe* or *sorry*. This is far from ideal: the lexicographic function of *better safe than sorry* is not just to head its own (sub)entry but also – a lexicographer might decide – to serve as an illustrative example in specific senses of the words it is composed of. The lexicographer's desire to include it in a specific location inside one or more specific entries is well-motivated and the *treat-multiwords-as-headwords* method is only a workaround.

What is needed is a way for including a single multiword item in several locations inside several entries, without having to keep multiple copies of them in multiple locations. DMLex makes this possible. DMLex treats multiword units as independent entries (like the *treat-multiwords-as-headwords* method) but additionally provides the mechanism of `relation` objects which can trigger software agents into rearranging invididual entries into a hierarchy of entries and subentries at display-time.

7.2.5 Entry-internal sense relations

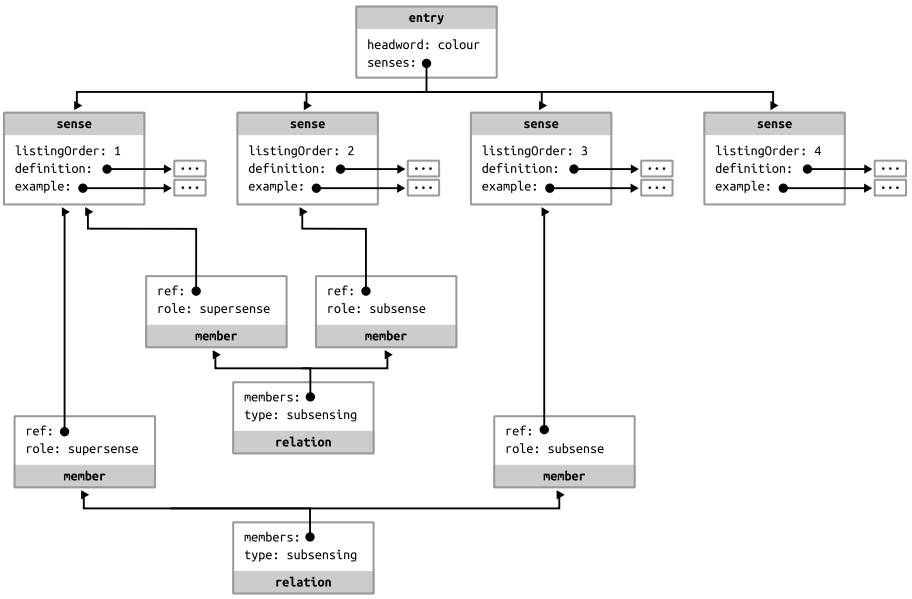
Another thing which is quite common in dictionaries is senses inside senses. An example can be seen in Figure 7-12: the entry for *colour* has two top-level senses, the first of which has two subsenses.

Figure 7-12 An entry with subsenses

We have discussed in Chapter 6 how dictionary schemas typically represent this by allowing recursion on senses. This is not allowed in DMLex. A DMLex

entry always contains only a flat list of senses but lexicographers have the option to create relations between senses that can trigger software agents into rearranging the senses into a hierarchy at display-time. Figure 7-13 shows what such a model looks like.

Figure 7-13 Subsenses modelled in DMLex



When lexicographers decide to arrange the senses of a word into a hierarchy of supersenses and subsenses, this is usually motivated by the existence of a *semantic relation* between the supersense and the subsense. The subsense is often a *specialization* of the supersense, or its *metaphorical extension*, or something in that vein. The exact nature of these semantic relations is typically not expressed explicitly in dictionaries, and only "hinted at" through the existence of subsenses. In DMLex, however, it is possible to be explicit about these things. A DMLex implementor can set up an inventory of types for relations between senses of the same entry, with types such as `specialization`, `metaphoricalExtension` and others, and lexicographers can use them to describe how the senses of each entry are related to one another. Then, during display-time, some of these relations can be configured to trigger the software into rearranging the senses into a hierarchy, while others

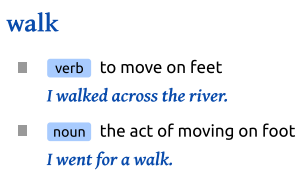
can remain invisible to the user but still available for automated tasks in NLP, for statistical analysis and so on.

7.2.6 Separation of form and meaning

DMLex enforces a strict separation between the formal properties of headwords (their orthography, morphology and phonology) and their semantic properties (their meaning and usage). The former is always recorded at the entry-level (as properties of the `entry` object) and the latter always at the sense-level (as properties of the `sense` object). This is far from common in lexicography. Dictionaries sometimes have part-of-speech labels attached to individual senses rather than to the entire entry, or sense-specific inflected forms of the headword attached to indiviual senses. Such things are prohibited in DMLex, for reasons which – as mentioned several times before – have to do with avoiding computational complexity. It makes things easier for IT tasks if it can be guaranteed that all the formal properties of the headword are at the entry-level and that no exceptions or overrides await further down inside the entry.

All of this does not mean that sense-specific formal properties cannot be expressed in DMLex. They can, but it has to be done differently. For example, let's assume a lexicographer wants to describe the headword *walk* which is both a verb and a noun. The typical approach in (English) lexicography is to create a single entry for this headword with one sense for the verb and another for the noun, like in Figure 7-14.

Figure 7-14 An entry with sense-specific parts of speech



This cannot be done in DMLex because it violates the principle of form/meaning separation. What can be done, however, is using relations to approximate this at display-time for the human end-user. The lexicographer

needs to create two separate entries for the two *walks* (which can be thought of as effectively treating them as homonyms, or as "spreading" the headword over two entries), connect them with a relation, and make sure all software is programmed to handle this relation such that the on-screen display looks almost like a single entry. Figure 7-15 shows the model and Figure 7-16 shows what the end-user might see.

Figure 7-15 How DMLex models a headword spread over two entries

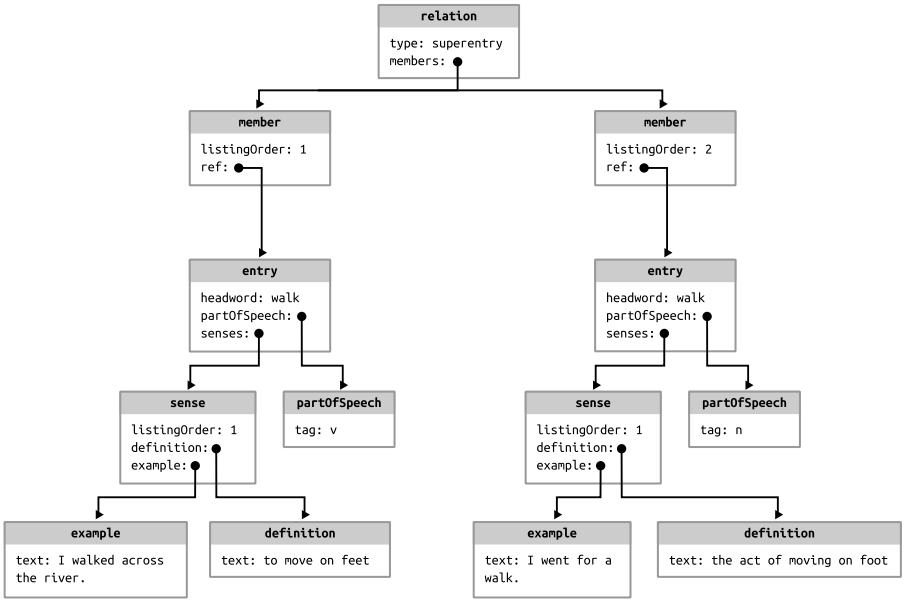
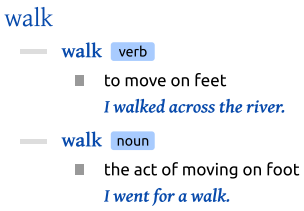


Figure 7-16 Suggested rendering of a headword spread over two entries



By remodelling the situation this way, we lose almost nothing compared to Figure 7-14: the user is still able to see both the verb and the noun simultaneously, in something that resembles a single entry. But internally, the

verb and the noun are represented as separate entries, which makes the data more easily usable for machine applications.

7.3 Conclusion: towards a data-centric lexicography of the future

TBD.

Epilogue

In the prologue to this thesis I mentioned how my career in computational lexicography has given me a chance to develop the thinking captured in this thesis. Although the thesis is an important milestone, it is not the end of the journey. How are the two innovations introduced here – NVH and DMLex – likely to develop in the future?

The next logical step for NVH would be to standardise the language, to write a full and formal specification. This has not been necessary so far while the user community is small, but may become necessary if and when the language becomes adopted more widely. But will it? That depends, among other things, on whether uses-cases can be found for it in disciplines other than lexicography. Is NVH's support for headedness a strong enough selling point outside lexicography? If it is, will the language need to be extended with additioonal features to meet the needs of other disciplines? If so, can that be done without the language becoming over-complicated?

As for DMLex, this thesis is being written at a time when the specification has not been fully approved yet: it is now a "Draft 02". But I do not see any obstacles to getting DMLex through to a fully endorsed OASIS specification soon. The main challenge will come after that: how will the industry welcome it, who and how will implement it? I do not indent to only watch DMLex's future development passively, I intend to have an active part in producing a working implementation in the near future. But more importantly, I will be eager to observe whether and how DMLex is going to influence the thinking of software engineers about lexicographic data models. In a sense, this is a more important goal than concrete implementations. Even if nobody ever produces a fully faithful implementation of DMLex, it will still be a success if DMLex serves as a sourcre of inspiration, as a catalog of reusable desgin patterns, for future builders of dictionary writing systems.

Human-oriented lexicography is an exciting discipline to be in right now. After a long stagnation in a superficially digitised, text encoding-style mode of operation, it is finally on its way to a future where dictionaries can fully exploit the possibilities of the digital medium. I hope that my thesis has been a valuable contribution to that transformation.

References

Abel, A. (2022) **'Dictionary writing systems'**, in: Hanks, P.; de Schryver, GM. (ed.) *International Handbook of Modern Lexis and Lexicography*, Springer, https://doi.org/10.1007/978-3-642-45369-4_111-1

Apresjan, J. D. (1974) **'Regular Polysemy'**, in: *Linguistics*, vol. 12, no. 142, pp. 5–32. https://doi.org/doi:10.1515/ling.1974.12.142.5

Atkins, B. T.; Rundell, Michael (2008) ***The Oxford Guide to Practical Lexicography***, Oxford University Press.

Baisa, V.; Blahuš, M.; Cukr, M.; Herman, O.; Jakubíček, M.; Kovář, V.; Medveď, M.; Měchura, M.; Rychlý, P.; Suchomel, V. (2019) **'Automating dictionary production: a Tagalog-English-Korean dictionary from scratch'**, in: *Proceedings of eLex 2019*.

Bergenholtz, Henning; Nielsen, Jesper Skovgård (2013) **'What is a Lexicographical Database?'**, in: *Lexikos*, vol. 23. https://doi.org/10.5788/23-1-1205

Bogaards, P. (1990,) **'Où cherche-t-on dans le dictionnaire ?'**, in: *International Journal of Lexicography*, vol. 3, no. 2, pp. 79–102.

Boguraev, Bran; Briscoe, Ted (1987) **'Large Lexicons for Natural Language Processing: Utilising the Grammar Coding System of LDOCE'**, in: *Computational Linguistics*, vol. 13, nos. 3-4, pp. 203–218. https://aclanthology.org/J87-3002/

Bourhis, Pierre; Reutter, Juan L.; Vrgoč, Domagoj (2020) **'JSON: Data model and query languages'**, in: *Information Systems*, vol. 89. 10.1016/j.is.2019.101478

Carlson, Kristofer J. (2007) **'The Case Against XML'**, http://www.krisandsusanna.com/Documents/the-case-against-xml.pdf

Convery, Cathal; Ó Mianáin, Pádraig; Ó Raghallaigh, Muiris; Atkins, Sue; Kilgarriff, Adam; Rundell, Michael (2010) **'The DANTE database (Database of ANalysed Texts of English)'**, in: *Proceedings of the 14th EURALEX international congress*, pp. 293–295.

Croft, William; Cruse, D. Alan (2004) ***Cognitive Linguistics***, Cambridge University Press.

DARIAH (2021) ***TEI Lex-0: A baseline encoding for lexicographic data, version 0.9.1***, https://dariah-eric.github.io/lexicalresources/pages/TEILex0/TEILex0.html

de Schryver, Gilles-Maurice (2023) **'Generative AI and Lexicography: The Current State of the Art Using ChatGPT'**, in: *International Journal of Lexicography*. https://doi.org/10.1093/ijl/ecad021

De Schryver, G.-M.; Joffe, D.; Joffe, P.; Hillewaert, S. (2006) **'Do dictionary users really look up frequent words? On the overestimation of the value of corpus-based lexicography'**, in: *Lexikos*, vol. 16, pp. 67–83.

ECMA 404 – The JSON data interchange syntax, http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf

Erjavec, Tomaž; Evans, Roger; Ide, Nancy; Kilgarriff, Adam (2000) **'The Concede Model for Lexical Databases.'**, in: *Proceedings of the Second International Conference on Language Resources and Evaluation (LREC)*. http://www.lrec-conf.org/proceedings/lrec2000/pdf/335.pdf

Erlandsen, Jens (2010) **'iLEX, a general system for traditional dictionaries on paper and adaptive electronic lexical resources'**, in: *Proceedings of the 14th EURALEX international congress*.

Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John (1995) ***Design Patterns: Elements of Reusable Object-Oriented Software***, Addison-Wesley.

Goldfarb, Charles F.; Rubinsky, Yuri (1990) ***The SGML handbook***, Clarendon Press and Oxford University Press.

Halpin, Terry; Morgan, Tony (2008) ***Information Modeling and Relational Databases***, Elsevier, https://doi.org/10.1016/B978-0-12-373568-3.X5001-2

Horák, Aleš; Rambousek, Adam (2018) **'Lexicography and Natural Language Processing'**, in: Fuertes-Olivera, Pedro A. (ed.) *The Routledge Handbook of Lexicography*, pp. 179–196.

Ide, Nancy; Kilgarriff, Adam; Romary, Laurent (2000) **'A formal model of dictionary structure and content'**, in: *Proceedings of the 9th Euralex International Congress*.

ISO 24613-1:2024 Language resource management: Lexical markup framework (LMF), Part 1: Core model, https://www.iso.org/standard/82014.html

8879:1986 – Standard Generalized Markup Language (SGML), https://www.iso.org/standard/16387.html

ISO/IEC 21778:2017 – The JSON data interchange syntax, https://www.iso.org/standard/71616.html

M. Jakubíček; M. Měchura; V. Kovář; P. Rychlý (2018) **'Practical Post-Editing Lexicography with Lexonomy and Sketch Engine'**, in: *Book of Abstracts of EURALEX International Congress 2018*.

Jakubíček, Miloš; Kovář, Vojtěch; Rychlý, Pavel (2021) **'Million-Click Dictionary: Tools and Methods for Automatic Dictionary Drafting and Post-Editing'**, in: *Book of Abstracts of the 19th EURALEX International Congress*, pp. 65–67.

Kilgarriff, Adam; Rychlý, Pavel; Smrž, Pavel; Tugwell, David (2004) **'The Sketch Engine'**, in: *Proceedings of the 11th EURALEX International Congress*, pp. 105–116.

Adam Kilgarriff; Vojtěch Kovář; Pavel Rychlý (2010) **'Tickbox Lexicography'**, in: *Proceedings of eLex 2019*, pp. 411–418.

Kilgarriff, Adam; Baisa, Vít; Bušta, Jan; Jakubíček, Miloš; Kovář, Vojtěch; Michelfeit, Jan; Rychlý, Pavel; Suchomel, Vít (2014) **'The Sketch Engine: ten years on'**, in: *Lexicography*, vol. 1, pp. 7–36.

Kipfer, Barbara A. (1983) **'Methods of Ordering Senses Within Entries'**, in: *Proceedings of the 1st EURALEX International Congress*.

Kovář, Vojtěch; Močiariková, Monika; Rychlý, Pavel (2016) **'Finding Definitions in Large Corpora with Sketch Engine'**, in: *Proceedings of LREC 2016*, pp. 391–394.

Lew, Robert (2013) **'Identifying, Ordering and Defining Senses'**, in: Jackson, Howard (ed.) *The Bloomsbury Companion to Lexicography*, pp. 284–302, Bloomsbury Academic.

Lew, Robert (2023) **'ChatGPT as a COBUILD lexicographer'**, in: *Humanities and Social Sciences Communications*, vol. 10. https://doi.org/10.1057/s41599-023-02119-6

Lew, Robert; de Schryver, Gilles-Maurice (2014) **'Dictionary Users in the Digital Revolution'**, in: *International Journal of Lexicography*, vol. 27, pp. 341–359. https://doi.org/10.1093/ijl/ecu011

Lew, R.; Wolfer, S (2024) **'What Lexical Factors Drive Look-Ups in the English Wiktionary?'**, in: *SAGE Open*, vol. 14,. https://doi.org/10.1177/21582440231219101

Lew, R.; Frankenberg-Garcia, A.; Rees, G.; Roberts, J.; Sharma, N. (2018) **'ColloCaid: A real-time tool to help academic writers with English collocations'**, in: *Proceedings of the XVIII EURALEX International Congress*, pp. 247–254.

Maks, E.; Tiberius, C.; van Veenendaal, R. (2009) **'Standardising Bilingual Lexical Resources According to the Lexicon Markup Framework'**, in: *Proceedings of LREC 2008*. https://research.vu.nl/en/publications/standardising-bilingual-lexical-resources-according-to-the-lexico

Měchura, Michal (2016) **'Data Structures in Lexicography: From Trees to Graphs'**, in: *Recent Advances in Slavonic Natural Language Processing*. https://michmech.github.io/pdf/raslan2016.pdf

Měchura, Michal (2017) **'Introducing Lexonomy: an open-source dictionary writing and publishing system'**, in: *Proceedings of eLex 2017 conference*, pp. 662–679.

Měchura, Michal (2017) **'How (not) to build a European Dictionary Portal'**, Final Conference of the European Network of e-Lexicography, https://www.youtube.com/watch?v=ORrGelo9ytU

Medveď, M.; Měchura, M.; Tiberius, C.; Kosem, I.; Kallas, J.; Jakubíček, M.; Krek, S. (ed.) (2023) ***Electronic lexicography in the 21st century: Invisible Lexicography***, Proceedings of the eLex 2023 conference, https://elex.link/elex2023/proceedings-download/

Nurseitov, Nurzhan; Paulson, Michael; Reynolds, Randall; Izurieta, Clemente (2009) **'Comparison of JSON and XML data interchange formats: a case study'**, in: *Caine*, pp. 157–162.

OASIS (2024) ***Data Model for Lexicography (DMLex) Version 1.0, Committee Specification Draft 02***, https://docs.oasis-open.org/lexidma/dmlex/v1.0/csd02/dmlex-v1.0-csd02.pdf

Ogbuji, Uche (2004) **'Considering container elements: When to use elements to wrap structures of other elements'**, in: *Principles of XML design*, IBM, https://www.ibm.com/developerworks/library/x-contain/index.html

Pemberton, Steven (2023) **'The Printing Press vs The Web: The Effects'**, The Web Conference, https://homepages.cwi.nl/~steven/Talks/2023/05-02-webconf/

Rambousek, Adam; Jakubíček, Miloš; Kosem, Iztok (2021) **'New developments in Lexonomy'**, in: *Proceedings of the eLex 2021*, pp. 455–462.

Ross, Jeanne W.; Beath, Cynthia M.; Mocker, Martin (2021) ***Designed for Digital: How to Architect Your Business for Sustained Success***, The MIT Press.

Rundell, Michael; Atkins, Sue (2011) **'The DANTE database: a User Guide'**, in: *Proceedings of eLex 2011*. https://elex2011.trojina.si/Vsebine/proceedings/eLex2011-32.pdf

Sinclair, John McHardy (ed.) (1987) ***Looking up: An account of the COBUILD project in lexical computing and the development of the Collins COBUILD English language dictionary***, Collins ELT.

Stará, Marie (2019) ***Automatická tvorba definic z korpusu***, Master's thesis, Masaryk University, https://is.muni.cz/th/i9wm5

Tarp, Sven (2008) ***Lexicography in the Borderland Between Knowledge and Non-Knowledge: General Lexicographical Theory with Particular Focus on Learner's Lexicography***, Max Niemeyer Verlag.

TEI Consortium (2007) ***TEI P5: Guidelines for Electronic Text Encoding and Interchange***, https://tei-c.org/Guidelines/P5/

W3C (2008) ***Extensible Markup Language (XML) 1.0***, https://www.w3.org/TR/2008/REC-xml-20081126/

W3C (2016) ***Lexicon Model for Ontologies: Community Report, 10 May 2016***, https://www.w3.org/2016/05/ontolex/

W3C (2019) ***The OntoLex Lemon Lexicography Module: Community Report, 17 September 2019***, https://www.w3.org/2019/09/lexicog/

Ernst Wiegand, Herbert (1989) **'Der Begriff der Mikrostruktur: Geschichte, Probleme, Perspektiven'**, in: *Wörterbücher: Ein internationales Handbuch zur Lexikographie*, de Gruyter, pp. 409–462.

YAML Ain't Markup Language (YAML) version 1.2, Revision 1.2.2, https://yaml.org/spec/1.2.2/