Translating the Ding dictionary to FreeDict TEI

Bachelor's Thesis

Einhard Leichtfuß

Christian-Albrecht University of Kiel
Programming Languages and Compiler Construction Group

Advised by: Priv.-Doz. Dr. Frank Huch

2020-10-14

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, daß ich die vorliegende Arbeit selbstständig verfaßt und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Kiel, den 14 Oktober 2020

Abstract

The Ding dictionary is the largest known digital German-English dictionary that is free as in Free Software. The FreeDict project publishes various free digital bilingual dictionaries in a custom dialect of the TEI format, together with tools to derive dictionaries in other formats. Included in the set of FreeDict dictionaries is a German-English dictionary that is based off a notably outdated version of the Ding dictionary. The original program used to translate the Ding dictionary is unfortunately lost. A prior attempt at rewriting such a translating program was made by Sebastian Humenda, member of the FreeDict project.

This thesis presents a new translating program targeted at the currently latest stable version of the Ding dictionary, 1.8.1, thereby explicitly representing most of the information present in the Ding dictionary and inferring some further information that is not explicitly represented in the Ding dictionary.

Contents

Chapter 1

Introduction

Human interaction largely depends on the use of natural language. In today's world, there are numerous different natural languages, where most people only are capable of conversing in very few of them. At the same time, the world is growing more and more interconnected, with people of many different native tongues desiring to interact with one another. Therefore, people often need to resort to using a second, non-native language, most frequently English, which has emerged as the primary international language during the 20th century [47].

Bilingual dictionaries are among the most useful tools to facilitate intercourse between humans without a common native language. This thesis aims to make one particular *free*[1] German-English dictionary—the *Ding* dictionary—more easily accessible, by incorporating it into the *FreeDict project*, that is, translating it to the *FreeDict TEI* format.

1.1 The FreeDict project

> *"The FreeDict project strives to be the most comprehensive source of truly free bilingual dictionaries. They are not just free of charge, but they give you the right to study, change and modify them, as long as you guarantee others these freedoms, too. Founded in 2000, FreeDict nowadays provides over 140 dictionaries in about 45 languages and thanks to its members, grows continuously." [16]*

The FreeDict project publishes these dictionaries in a custom variant of the *TEI* format, as described by the *TEI Guidelines* [38] (see section 1.3). Some of the dictionaries are handwritten, others are imported from other free dictionary sources by means of automated translation from the respective other format. Furthermore, the FreeDict project notably provides tools [14] to automatically convert its dictionaries into other formats[2] readily to be used by dictionary programs. In particular, the resulting dictionaries in the respective target format are also made freely available.

1.2 The Ding dictionary

The Ding Dictionary is a free digital German-English dictionary that is primarily promoted as part of the equally free *Ding* program. Composed of 197,771 *lines*, where a single *line* may

[1] free as in Free Software

[2] As of now, the *DICT* [44] and *Slob* [40] formats are supported.

contain more than one *entry*[3], it is most likely the largest free digital German-English dictionary. Both the program and the dictionary are principally written by Frank Richter [35] and made available under the terms of the GPL[4]. Additionally to the Ding program, there is also a web site[5] providing similar access to the Ding dictionary.

The Ding dictionary is provided in a custom format, which is meant to be processed by (variants of) *grep*[6]. In fact, Ding is an acronym for ***DI***ctionary ***N***ice ***G***rep [35]. The mentioned Ding front-ends essentially are front-ends to grep that add some Ding specific formatting to the matched lines. Therefore, the Ding format is not required to follow a very strict syntax, which generally makes parsing difficult. While there exists a syntax specification [34], it leaves out many details and seems in general outdated. The syntax of the Ding dictionary is actually quite rich; however, most of it is expected to be parsed by a human reader—when using one of the Ding front-ends.

A simple, but nontrivial line from the Ding dictionary might look as follows:

```
Ding {n}; Sache {f} | Dings {n} [ugs.] :: thing | thingy; dingus
```

It expresses a relation between German and English words, precisely that

- "`Ding`" and "`Dings`" are neuter nouns in singular form,
- "`Sache`" is a feminine noun in singular form,
- "`Dings`" is used colloquially (German abbrev.: "`ugs.`"),
- "`Ding`" and "`Sache`" are translation equivalents to "`thing`",
- "`Dings`" is a translation equivalent to "`thingy`" and "`dingus`".

Note how in this example, annotations are only present on the German *side*. This is not the case with all lines in the Ding dictionary; however, the German side is generally much richer annotated.

Note further that the Ding dictionary is naturally bidirectional: The above line may be found equally by grep'ing for any of "`Ding`" and "`thing`".

In the Ding program, the result of such a query would look as follows:

Ding *{n}*; Sache *{f}*	thing
Dings *{n}* *[ugs.]*	thingy; dingus

See subsection 1.3.1 on how the corresponding TEI looks like.

1.2.1 Incorporation into FreeDict

The current German-English dictionary of the FreeDict project is based on an old version of the Ding dictionary; for which the corresponding translation program is unfortunately lost. One of the FreeDict project's members, Sebastian Humenda, attempted to write a new translation program[7], starting in 2017. Due to a lack of time, he suggested on 2020-05-08 to instead replace the dictionary with another one for that a translation program was already available [24] (see also subsection 1.2.2).

[3]The Ding website claims a count of "approximately 326,000 entries" [35]. The resulting TEI dictionaries (German-English and English-German) each contain more than 400,000 entries

[4]GNU General Public License: A *copyleft* Free Software license. [43]

[5]`https://dict.tu-chemnitz.de/`

[6]A common command-line regex-matching program. [11]

[7]The program's code is available in the `importers/ding2tei` folder of FreeDict's tools git repository [14]

After some superficial investigation, I offered to try writing a new translation program myself, which would be using the *Haskell* language and likely the *Alex* lexical analyser generator and the *Happy* parser generator. The result is this thesis and the associated *ding2tei-haskell* [8] program.

1.2.2 Comparison with on-line dictionaries

The likely most common way to access dictionary data today is by use of an on-line dictionary.

There exist several commercial dictionary websites, the most notable German-English dictionaries likely being those on `leo.org` and `dict.cc`.

While these two will show to be larger in size, the Ding dictionary has the notable advantage to be provided under a free license. It can hence be used off-line, by any program, and with any modifications applied. I personally consider dictionaries critical infrastructure and believe that such should ideally be free.

LEO

As of 2013-01-22[9], the English-German dictionary on `leo.org`—which likely is *bidirectional*[10]—contains 787,419 entries [27]. It thereby had roughly twice[11] the size of the Ding dictionary at that time.

Entries on `leo.org` may relate either single words, composed expressions or examples, where the latter are separated. Searching for a word seems to cause an infix search. The LEO dictionary is therefore quite similar to the Ding dictionary, even though the infix search likely does not allow searching for potentially hidden syntactic elements such as the language separator (`::`) in the Ding dictionary. However, the LEO dictionary also encompasses at least some syntax that is not machine-processed, and can therefore explicitly be searched for [26].

Dict.cc

The `dict.cc` web site essentially provides the infrastructure for its users to collaboratively build a dictionary.

As of 2020-08-29, the bidirectional German-English dictionary on `dict.cc` contains 1,213,314 entries[12], which is roughly three times[13] the size of the Ding dictionary.

As presented on the About Page [23], the German-English dictionary is actually based on an earlier version of the Ding dictionary. According to the Download Page [22], it was provided to the owner of `dict.cc` under a more permissive license than the GPL, allowing the site owner to publish a modified version under a more restricted version. Despite the possibility to download the dictionary data, it is therefore not usable under a free license, according to the site owner[14] [22].

[8] The code is equally available in the tools repository [14] at `importers/ding2tei`, albeit currently in the branch `ding2tei-haskell-rewrite`.

[9] I am not aware of a source for a recent count.

[10] See definition 2.1.3.

[11] These numbers should be taken with a grain of salt, since they likely denote different entities; for LEO, it is unknown of which kind these entities are.

[12] The dictionary data can be obtained automatically via E-Mail [22], albeit in a slightly obfuscated form.

[13] See footnote 11 on why this comparison is flawed. In particular, since the `dict.cc` dictionary only relates single words and expressions—unlike both the Ding dictionary and TEI—the count is naturally much higher for that.

[14] This is actually debatable, since the dictionary data was originally available under the terms of the GPL and grew with the help of volunteers—as it still does [45]. It seems reasonable that these volunteers—from back then—hold copyright on their additions, while these could only have happened under the terms of the GPL. In this case, the dictionary data as it can be downloaded today, due to the terms of the GPL, must be provided

Although the German-English dictionary on `dict.cc` is built atop the Ding dictionary, it retains little of its syntax and structure. In particular, the syntax is much simpler; the dictionary essentially consists of word pairs, where both may have annotations.

Due to the large number of users, who are able to add—and in particular to fix—incorrect entries, the dictionary is generally of better quality, but lacks some of the rich structure of the Ding dictionary.

Wiktionary

The *Wiktionary* is a project just like Wikipedia, but for dictionaries. Its dictionaries are not bilingual by design; however, many entries contain translations into other languages. [57]

In fact, there exists a chain of Free Software projects (DBNary [36], WikDict [2] and FreeDict's WikDict importer[15]) that translate the Wiktionary's translations to FreeDict TEI. As noted in subsection 1.2.1, the alternative to using the Ding dictionary as source for the German-English FreeDict dictionaries would have been to use the Wiktionary data.

1.3 TEI

The *Text Encoding Initiative (TEI)* is an international community that primarily develops the *TEI Guidelines* aimed at the encoding of any digital text[16] [5, 38]. The current version are the *TEI P5 Guidelines*, which will be considered exclusively in this thesis.

The Guidelines in particular contain a chapter on dictionaries (chapter 9). In the following, the language (or format) described by the TEI Guidelines, in particular the chapter on dictionaries, shall be referred to as simply *TEI*, unless further qualified.

The TEI format allows to represent the original text in all its particularities, though enriched with information that cannot be easily derived from the original text by a machine. As an example (which is not dictionary related), consider the following encoding of a phrase, as found in the TEI Guidelines:

```
<cl type="relative"
 function="clause_modifier">Which frightened
 both the heroes so,<cl>They quite forgot their quarrel.</cl>
</cl>
```

As one might guess from the example above, TEI uses XML as markup language, though the choice of this markup language is not inherent to TEI. SGML was used previously and this choice may be altered again [38].

1.3.1 FreeDict TEI

The FreeDict project uses a custom dialect of TEI based on the TEI Guidelines' chapter on dictionaries [15]. It shall be referred to as FreeDict TEI here and later also simply as TEI.

In contrast to generic TEI, FreeDict TEI is not meant to represent purely presentational data. In the context of the Ding this means that, among others, the previously seen separators (`::`, `;`, `|`), and braces surrounding grammar annotations should not be preserved.

There are some further restrictions that ease automated handling by the FreeDict tools.

To get an idea how the format looks like, recall the example Ding line from section 1.2:

```
Ding {n}; Sache {f} | Dings {n} [ugs.] :: thing | thingy; dingus
```

under these terms.

[15] Code available at `importers/wikdict` in FreeDict's tools repository [14].

[16] For example, I could have specified this thesis in TEI, but I haven't.

Instead of translating this right away to FreeDict TEI XML (which quickly becomes huge), we first look at an abstract version of how this can be represented in TEI:

```
entries:
 * "Ding"
     - translation: "thing"
     - synonym: "Sache"
     - related: "Dings"
     - part-of-speech: noun
     - number: singular
     - gender: neuter
 * "Sache"
     - translation: "thing"
     - synonym: "Ding"
     - related: "Dings"
     - part-of-speech: noun
     - number: singular
     - gender: feminine
 * "Dings"
     - translation: "thingy"
     - translation: "dingus"
     - related: "Ding"
     - related: "Sache"
     - part-of-speech: noun
     - number: singular
     - gender: neuter
     - usage[register]: "ugs."
```

Note how this is clearly a *directed*[17] dictionary. A diametrical version can be obtained similarly.

The corresponding (redacted) TEI XML, as produced by the final translation program, looks as follows:

```
<TEI xmlns="http://www.tei-c.org/ns/1.0" version="5.0">
  <teiHeader xml:lang="en" /><!-- redacted -->
  <text xml:lang="en">
    <body>
      <entry xml:id="Ding.1">
        <form>
          <orth>Ding</orth>
        </form>
        <gramGrp>
          <gen>neut</gen>
        </gramGrp>
        <sense>
          <cit type="trans">
            <quote xml:lang="en">thing</quote>
          </cit>
          <xr type="syn">
            <ref target="#Sache.1">Sache</ref>
          </xr>
          <xr type="see">
            <ref target="#Dings.1">Dings</ref>
          </xr>
        </sense>
      </entry>
      <entry xml:id="Sache.1">
        <form>
          <orth>Sache</orth>
        </form>
        <gramGrp>
```

[17] See definition 2.1.3.

```
          <gen>fem</gen>
        </gramGrp>
        <sense>
          <cit type="trans">
            <quote xml:lang="en">thing</quote>
          </cit>
          <xr type="syn">
            <ref target="#Ding.1">Ding</ref>
          </xr>
          <xr type="see">
            <ref target="#Dings.1">Dings</ref>
          </xr>
        </sense>
      </entry>
      <entry xml:id="Dings.1">
        <form>
          <orth>Dings</orth>
        </form>
        <gramGrp>
          <gen>neut</gen>
        </gramGrp>
        <sense>
          <usg type="reg">ugs.</usg>
          <cit type="trans">
            <quote xml:lang="en">thingy</quote>
          </cit>
          <cit type="trans">
            <quote xml:lang="en">dingus</quote>
          </cit>
          <xr type="see">
            <ref target="#Ding.1">Ding</ref>
          </xr>
          <xr type="see">
            <ref target="#Sache.1">Sache</ref>
          </xr>
        </sense>
      </entry>
    </body>
  </text>
</TEI>
```

1.4 Goals

The foremost goal is to write a program that translates the Ding dictionary, as specified in the Ding format, to FreeDict TEI.

Primarily, the latest stable version of the Ding dictionary, 1.8.1, is targeted.

In the following, the words *must* and *should* are used deliberately, to denote obligatory and facultative goals, respectively. The word *should* may in particular also mean that there is also a value in partial fulfillment of the respective goal.

1.4.1 Validity of the output

The resulting TEI must be valid FreeDict TEI and it should work well with most of the FreeDict tools.

1.4.2 Quality of the translation

The resulting TEI should retain most of the information encoded in the Ding dictionary, excluding any presentational aspects. This information should mostly be encoded explicitly and as specific as possible in TEI. For example,

- grammar annotations such as `{f}` or `{pl}` must be properly classified (gender, number) and encoded explicitly (`<gen>`, `<num>`) using this classification,
- usage annotations such as `[coll.]` should be classified into common recommended categories such as *register* or *domain* and encoded accordingly (e.g., `<usg type="reg">coll.</usg>`).

Information that either cannot be reliably extracted from the Ding dictionary or cannot be expressed in FreeDict TEI should either not be represented at all or retained in verbatim (as part of plain text nodes) in the resulting FreeDict TEI. The presence of such information is a natural consequence of the Ding dictionary being only partially intended as machine-readable. An example is the following line from the Ding dictionary:

```
Ackerwagen- und -maschinenreifen {m} :: implement tyre [Br.] / tire [Am.]
```

It is unclear how the hyphens (`-`) are to be interpreted[18] and similarly it is unclear which entities the slash (`/`) separates, in other words, the left and right scopes of the slash cannot be reliably determined.

Information that is deemed hard to extract from the Ding dictionary may be considered equally to such information that cannot be reliably extracted, and therefore dropped or retained in verbatim.

If, however, the reliable information extraction is only hindered by a small number of inconsistencies in the syntax, these should be fixed in the Ding dictionary.

Enrichment

The resulting TEI should be enriched with information that is not evident from the Ding, but can be reliably inferred. For example, the annotation of a gender (e.g., `{f}`), when not combined with `{pl}`, implies that the annotated *unit*[19] is in the singular form.

1.4.3 Quality of the Code

This is not a one-shot program. We shall see that at the end of this thesis that there remains a lot of space for improvement. Also, the Ding dictionary is evolving, later versions might require adaptation. Similarly, the expressiveness of FreeDict TEI might increase, such that more information can be expressed explicitly. Further, this program is to become part of the FreeDict project, there may be other people wanting to read or alter it.

The code should therefore be maintainable and well documented. One particular goal is that it allows understanding and some altering for people who are less experienced in programming or the chosen programming languages and tools.

[18] In fact, even I—as a human and speaker of German—have some trouble identifying the set of words described by use of these hyphens.

[19] See definition 2.1.1.

Efficiency

There are no high expectations towards efficiency of the code. It is not meant to be run often. In fact, the program is unlikely to be more often run than compiled.

Nonetheless, the program should not exceed certain reasonable bounds, both regarding runtime and memory usage. Ideally, it would translate the Ding dictionary on a not-too-ancient machine in less than an hour and without the need for swapping due to overly high memory usage. Note that testing can be performed on subsets of the input dictionary.

1.4.4 Further optional goals

Spanish-English dictionary

There exists also a Spanish-German dictionary that claims to be in the same format as the Ding dictionary [18]. Ideally, the translating program could handle this as well.

Other versions of the Ding dictionary

The latest stable version, 1.8.1, that is primarily targeted in this thesis, stems from 2016. There is also a more recent, albeit not tremendously larger, *devel* version. Unfortunately, there is no record of old *devel* versions; therefore, no devel version could serve as a base for the to be written program. Due to a lack of a clearly defined syntax, any changes in *devel* could make its parsing fail.

Nonetheless, it would be very valuable to also be able to parse and translate such a *devel* version.

Older versions of the Ding dictionary shall not be considered. They might only cause obstacles that are not relevant anymore.

Chapter 2

Syntax overview and analysis

Before attempting to translate between two languages, it is usually a good idea to learn these languages. This applies to natural languages in particular, but also to formal languages.

Therefore, in the following, the syntax of both the Ding dictionary and TEI are unfolded. This happens incrementally, where the two formats step down the ladder of complexity in parallel, so they can be compared and related with one another at each of these levels of detail.

This process is interleaved by section 2.3, detailing the methods of syntax analysis.

2.1 Preliminary definitions

Definition 2.1.1 (language, unit)**.** *A* language *refers to a natural language, in particular to a language of the Ding dictionary, English or German.*

A unit *is a single entity in a* language *that is atomic in the context of a dictionary. The class of* units *encompasses single words, composed expressions, single phrases, but also sequences of phrases, which might for example form an interaction between two speakers.*

Furthermore, a language *is identified with the set of* units *that stem from it.*

Definition 2.1.2 ((bilingual) dictionary)**.** *A* dictionary *is any kind of structure that relates units with one another. Unless specified otherwise, a* dictionary *refers to a* bilingual dictionary*, relating units of two distinct* languages*.*

Definition 2.1.3 (direction, source, target)**.** *A dictionary may have a* direction *with its two languages distinguished as* source *and* target *language.*

This is the case when a dictionary serves as a (partial) map from structures of the source *language to such of the* target *language.*

A dictionary that has a direction *is also called* directional*, and* directionless *otherwise. A dictionary with a single* direction *is called* monodirectional *and one with two directions* bidirectional*.*

Note that being directionless and bidirectional usually expresses the same, only from a different point of view.

2.2 Dictionary structure

The structure of the Ding dictionary fundamentally differs from that of TEI. Therefore, before getting into any specifics of the Ding and TEI formats, three different basic structures of bilingual

dictionaries shall be introduced and compared, including those that correspond to Ding and TEI.

When enriching these structures with examples, the Ding syntax will be used. This is possible because the Ding's structure will show to be the most general one.

Note that the following dictionary categories' terms are not standard terms, but rather customary to this thesis.

2.2.1 Relational dictionary

Definition 2.2.1 (relational dictionary)**.** *A* relational dictionary *D to two languages A and B is a relation in the mathematical sense between these two, that is,*

$$D :\subseteq A \times B.$$

This is arguably the simplest dictionary structure, though quite limited, as we will see when comparing with the others. It is clear that such a relational dictionary is directionless, the order of A and B does not matter.

Example 2.2.1.

```
Rolle :: part
Rolle :: role
Rolle :: roll
Ballen :: roll
```

Note how single units with several translations are required to be divided into several elements of the dictionary.

Neither the Ding dictionary nor TEI adhere to this structure; however, a dictionary of this structure can be naturally derived from both (and likely any bilingual dictionary).

2.2.2 Traditional dictionary (~ TEI)

TEI in general tries to allow for representation of any existing body of text, usually originating in print form. Therefore, in the case of dictionaries, the structure is essentially that found in traditional print dictionaries. Note that it does allow for different types than bilingual dictionaries, where the structure is mostly the same; we shall only consider the latter though.

Definition 2.2.2 (traditional dictionary)**.** *A* traditional dictionary *D is directional and contains mappings from key units in the source language A to non-empty sets of value units in the target language B. This can be expressed as*

$$D :\subseteq A \times (\mathcal{P}(B) \setminus \{\varnothing\}).$$

A single element of D relates a unit in A to equivalent units in B. Note that a single key $a \in A$, which is identified by its orthography, may have several sets of related units. This is the case where there are *homographs*[1]. Note further that not each unit $a \in A$ needs to occur as key in D.

Example 2.2.2.

```
Rolle :: role; part
Rolle :: roll
Ballen :: roll
```

[1] Homographs: Orthographically identical words/units with different pronunciation and/or meaning. [49]

From the point of view of computer science, one might prefer to see a traditional dictionary as a multi-map from A to sets of units in B, that is,

$$A \to \mathcal{P}(\mathcal{P}(B) \setminus \{\varnothing\}).$$

Compare example 2.2.2 to example 2.2.1 to see how traditional dictionaries are more expressive than relational dictionaries: Synonymous units in the target language may now be grouped.

Traditional dictionaries can be seen as a generalization of relational dictionaries, even though the former are monodirectional, unlike the latter. Given a relational dictionary $D_r \subseteq A \times B$, this can naturally be embedded in the space of traditional dictionaries, as

$$D_t := \{(a, \{b\} \mid (a, b) \in D_r\}.$$

An alternative embedding would be

$$\widetilde{D}_t := \{(a, \{b \mid (a, b) \in D_r\}) \mid a \in \operatorname{dom} D_r\}.$$

However, this would mean that potential homographs with distinct meanings (such as "`Rolle`" in example 2.2.1) are grouped together.

Most traditional print dictionaries further allow the subdivision of the values to a given key into several senses (and [recursive] sub-senses). This is also the case with TEI.

2.2.3 Semantic dictionary (~ Ding)

In comparison to print dictionaries, digital dictionaries are much less restrained, for example it is not imperative to have a natural order on its entries. The Ding dictionary takes advantages of this. Its basic structure is that of a *semantic dictionary*.

Definition 2.2.3 (semantic dictionary)**.** *A* semantic dictionary D *links sets of units in both languages together that all share a common sense, that is,*

$$D :\subseteq (\mathcal{P}(A) \times \mathcal{P}(B)) \setminus \{(\varnothing, \varnothing)\}.$$

Note that this definition allows elements in the dictionary that only link together units of a single language.[2] A corresponding alternative definition would be

$$\widetilde{D} :\subseteq (\mathcal{P}(A) \setminus \{\varnothing\}) \times (\mathcal{P}(B) \setminus \{\varnothing\}).$$

One may also consider such a semantic dictionary as a partial function from an abstract set of senses S to entities in both languages,

$$D^f : S \nrightarrow (\mathcal{P}(A) \times \mathcal{P}(B)) \setminus \{(\varnothing, \varnothing)\}$$

(or conversely). However, the nature of such an abstract sense is entirely unclear, and it is probably best defined by its value under D^f.

Example 2.2.3.

```
Rolle :: role; part
Rolle :: roll
Rolle; Ballen :: roll
```

[2] Semantic dictionaries can effectively be monolingual. Such dictionaries are known as *thesauri* [54]. In particular, the *Wordnet* [31]—a great free dictionary of English—may be considered (a generalization of) a monolingual semantic dictionary. There actually are some Ding entries that may be considered monolingual; both thesauri and the Wordnet shall serve as reference on why such entries nevertheless are meaningful.

Similarly to how relational dictionaries can be seen as a special case of traditional dictionaries, these can be considered a subset of semantic dictionaries. Given a traditional dictionary $D_t \subseteq A \times (\mathcal{P}(B) \setminus \{\varnothing\})$, this can be embedded as

$$D_s = \{(\{a\}, b) \mid (a, b) \in D_t\}.$$

Note that this also induces a natural embedding of relational dictionaries into the space of semantic dictionaries.

Note further that the embedding of a semantic dictionary into the space of traditional dictionaries is not generally possible.[3] In this aspect, the Ding format is therefore clearly superior to TEI.[4]

2.3 Syntax analysis

Neither the Ding nor the FreeDict TEI syntax is thoroughly documented. Before describing these, we shall therefore direct our attention on the means used to identify sufficiently strict syntax for both languages.

Note that the prior section on the dictionaries' structures partially depends on the results found by these means.

2.3.1 Ding

As mentioned in the introduction, there exists a quite old specification of the Ding syntax [34], which lacks many details though and is outdated in some aspects. For example, it specifies that braces always contain grammar information, while in practice, they may also contain inflected forms of a verb. Hence, this syntax specification is considered unreliable and not used as primary source of information on the syntax.

Instead, the syntax is inferred from the given dictionary in several steps. This includes searching for patterns that are described in the just mentioned Ding syntax specification.

Case studies

Without a reliable syntax description, the first step in inferring a syntax is to have a look at individual elements of the given text. In the case of the Ding dictionary, the primarily distinguishable entities are lines.

Automated analysis

Next, recognized syntax elements that do not seem obvious in semantics or in particularities of its syntax may be analyzed automatically. This is done primarily using the *sed* language [9] (see also section 3.2), which particularly allows to perform regular expression matching and replacement. For example, we may use this to list all contents of braces, together with the count of the respective occurrences, with the aid of some other small tools. This particular information can then be used to identify both grammar information and inflected forms inside braces, the used separators and the possible (separated) values of grammar annotations.

[3] The later translation of Ding to TEI will split up non-traditional entries into several entries and link these with one another, see subsection 2.4.2.

[4] While we can losslessly translate a Ding dictionary to TEI, as noted in footnote 3, this comes at the cost of data duplication.

During this process, there will also be identified many inconsistencies and bugs in the Ding syntax. In many cases, we consequently need to decide on what precise syntax should be considered correct. The goal is to have a rather strict syntax where there are not too many ways to convey the same meaning. Note how this process is therefore quite interwoven with the preprocessing as described in section 3.2, where the now identified bugs are to be fixed—also using sed.

Rectification based on syntax violations

Finally once there is a partial implementation of the main program, we can use that to identify violations of our current syntax. Such may either cause to refine our syntax—likely accompanied by further sed based syntax analysis—or to improve the main program, where the preprocessor is to count as part of it.

2.3.2 TEI

Since TEI, as defined by the TEI Guidelines [38], is meant to represent any kind of text, it is quite general and liberal in structure. FreeDict TEI is a stricter subset of TEI, which nevertheless retains many liberties. While the TEI Guidelines are quite thorough, the FreeDict TEI specific documentation as specified mostly in its Wiki [15] lacks information on how to handle many particularities. This gap can partially be filled by existing dictionaries of the FreeDict project [13] that may serve as examples.

Validation

Additionally, the FreeDict project provides XML schemata[5], which may be used to verify the syntax to a certain degree.

Human interaction

Questions that could not be resolved with sufficient clarity using the above methods, I have discussed with members of the FreeDict project, in particular Sebastian Humenda. There arose actually a lot of such questions, many of which I asked on the FreeDict mailing list [25], others via IRC[6]. A majority of these questions, together with any answers (some of which by myself), I have gathered in a page on the FreeDict wiki [12], to allow for a structured overview of both the remaining questions and the answers—where the latter may very well be useful to the FreeDict project in general, and might be used to refine its documentation.

TEI Lex-0

Another standard that describes a subset of TEI is *TEI Lex-0* [6]. In comparison to FreeDict TEI, the documentation is closer to completeness. There was a short lived debate on the FreeDict Mailing List on whether TEI Lex-0 should replace FreeDict TEI. Unfortunately, unlike FreeDict TEI, TEI Lex-0 “was created to handle "retrodigitized dictionaries"” and was consequently considered unfit [1]. Nevertheless, it may serve as a useful resource wherever the syntax of FreeDict TEI and TEI Lex-0 do not collide.

[5]The XML schemata are located in the `shared/` folder of the git repository holding non-imported dictionaries [13].

[6]Internet Relay Chat [50]

2.4 Syntax overview

In this section, the syntax of both the Ding dictionary and TEI is described, excluding some specialties[7]. In the course of this, we shall start defining the respective Haskell data types pertaining to the nodes of the *abstract syntax trees* (*ASTs*) of both Ding and TEI.

While there exist more modern approaches on AST representation, such as presented in a 2008 paper by McBride [29], I have chosen to take the more traditional approach of generally associating to each element of the syntax a single Haskell type.

Knowing that both the Ding and TEI language represent dictionaries, we may start by defining a common dictionary type:

```
data Dictionary header element = Dictionary
  { dictHeader  :: header
  , dictSrcLang :: Language
  , dictTgtLang :: Language
  , dictBody    :: Body element
  }
```

Since we only are concerned with bilingual dictionaries, there always need to be two languages, which we distinguish as source and target languages. In case of a directionless dictionary—as may be considered the Ding dictionary—we need to define a direction arbitrarily.[8] The header may contain any auxiliary information such as the version or the author of the respective concrete dictionary. Unlike the source and target languages, the type of the header is specific to the respective concrete dictionary type; therefore, the `Dictionary` is polymorphic in this aspect. The core part of a dictionary is naturally its body, composed of a set of elements, which also are particular to the respective dictionary type. The body is defined as follows:

```
newtype Body element = Body [element]
```

Note that unlike stated above, we do not define it as a set of elements, but rather a list or sequence of such. In practice, there may very well be dictionaries where the order of its elements matter. Even though this is not the case for both Ding and TEI, a list may very well serve the purpose of representing a set; we shall only need trivial set operations that can efficiently be performed on a list representation.

2.4.1 Ding

In the following, the Ding syntax, as identified using the means described in subsection 2.3.1, shall be explicated. Note that this syntax only applies to the German-English Ding dictionary as supplied with the Ding program. As mentioned in subsection 1.4.4 of the introduction, there also exists a Spanish-German dictionary that claims to be specified in the same format. Syntax analysis has shown though that the syntax is in fact, while similar, different (see section 4.1.4).

On the top level, the Ding dictionary is composed of a header and a list of *lines*. These shall be defined from the bottom up—alongside examples and the corresponding Haskell types—starting with simple lines and ending with general lines, which is where the `Line` data type shall be defined. Knowing that there will be defined such a type, we can already define the Haskell type for a Ding dictionary:

[7] Among the syntax particularities, we shall later only have a look at the most important ones—annotations—in section 2.5.

[8] For the Ding dictionary, the language occurring on the left side (German) shall be considered the source language and the language on the right side (English) the target language. That is, upon initial parsing. The dictionary may very well be mirrored programmatically.

```
type Ding = Dictionary Ding.Header Ding.Line
```

Remember that the Ding dictionary generally adheres to the structure of a semantic dictionary, which is the most general of the presented structures. Therefore, it may in particular contain lines that adhere to the structure of a relational dictionary's element. Such a line would simply relate two units. The simplest (and most prominent) line is hence:

```
Ding :: thing
```

Any line is separated by a double-colon (`::`), which separates the German *side* from the English *side*. The particular line above indicates that the German word "Ding" is equivalent to the English word "thing".

Further, the units in a line may be annotated[9] with additional information, for example as follows:

```
Dings {n} [ugs.] :: thingy [coll.]
```

In this example, `{n}` is a grammar annotation specifying that "`Dings`" is a neuter noun, while `[ugs.]` and `[coll.]` are usage annotations that both specify that the respective unit is used colloquially[10]

This motivates the following definition of a unit in the context of the Ding dictionary.

Definition 2.4.1 (Ding unit)**.** *A* Ding unit *is a unit that is annotated with further information. Such annotations are either prepended to the unit (in the case of parentheses-enclosed prefix collocates) or appended to the unit (in all other cases) using special markup such as braces or slashes.*

A Ding unit *may also simply be referred to as a* unit*.*

The `Unit` type distinguishes different types of annotations, as they are introduced in section 2.5:

```
data Unit = Unit
  { unitHeadword    :: String
  , unitPlain       :: String
  , unitGrammar     :: [GrammarInfo]
  , unitUsages      :: [Usage]
  , unitPrefixes    :: [String]
  , unitSuffixes    :: [String]
  , unitAbbrevs     :: [String]
  , unitInflected   :: Maybe InflectedForms
  , unitReferences  :: [String]
  , unitExamples    :: [Example]
  }
```

Note that `Example`s are special in that they may be considered annotations in the AST, albeit not being explicitly present in the Ding dictionary like the other annotations; instead they will be added in the enrichment step (see subsection 3.6.2).

In order not to be restricted to elements of a relational dictionary, and to allow for general elements of a semantic dictionary, define the following.

Definition 2.4.2 (group)**.** *A* group *is a set of units in a single language that share a common translation* group *in the respective other language of the dictionary. The units in a* group *shall be considered synonyms.*

[9] See section 2.5.

[10] *ugs.* is the German abbreviation corresponding to *coll.*.

In Haskell, a group is implemented as follows.

```haskell
newtype Group = Group [Unit]
```

Note that the obsolete Ding specification [34] only refers to "similar" units in what I call a group. While the syntax clearly mandates that such units share a common translation group, it is less clear that the units in a group indeed are synonymous. The decision on whether or when they are is somewhat important, because in the TEI format we will need to classify the relation between units in a group, synonymy being one of the options.

There is one particular class of cases where the synonymy is debatable, as depicted by the following example:

```
Studentin {f}; Student {m} :: student
```

For most nouns describing persons, the German language has gendered forms, that is, for each of the masculine and feminine gender there is a word that both has this grammatical gender and describes people of this social gender. The English language in most cases instead only has a single word for all social genders. In fact, there is quite a debate in Germany on gender-neutral language, for several reasons [41], one of them being the perceived need for truly neutral (generic) forms, as the English language has them—currently, in a generic context, the masculine forms are most frequently used. Alternatively, the feminine form[11] or newly developed forms may be seen. I personally take the stand that in direct comparison to the masculine gender, the feminine gender is a better choice to express genericity—and therefore a (synonymous) option. Further, I do not see no harm in blurring the lines between the genders a little. Thus, I have decided to not distinguish differently gendered forms from other synonyms.

Definition 2.4.3 (Ding entry)**.** *An* entry *is a pair of groups, each in one of the dictionary's languages, where each of them contains translations to the respective other* group*.*

The corresponding Haskell type:

```haskell
data Entry = Entry Group Group
```

A richer line may now contain a single entry relating two groups:

```
Dings; Dingsda :: thingy; dingbat; dingus
```

The units in a group are separated by semicola.

As may have become apparent from the separate notion of an entry, the Ding dictionary actually has one more layer than semantic dictionaries.

Definition 2.4.4 (Line)**.** *A* line *is composed of a non-empty sequence of entries, where these entries, or rather the units contained therein, have similar meanings.*

Note that the entries in a line are ordered, this will be important later.

In Haskell, a line is defined as:

```haskell
data Line = Line [Entry]
```

The representation of a line containing multiple entries in Ding is not as simple as the others':

```
Ding; Sache | Dings :: thing | thingy; dingbat; dingus
```

[11] As an example, the German ministry of justice recently authored a bill wherein the feminine gender is used generically—which was soon rejected by the interior ministry. [37]

The corresponding separator is the vertical bar; it does not separate entries though, but instead the entries, separately on each side. The line's entries are formed by combining the left and right groups, in order. In particular, the numbers of vertical bars on both sides are required to match. The formation of a line from two lists of groups can be easily expressed in Haskell:

```haskell
makeLine :: [Group] -> [Group] -> Line
makeLine = Line . map (uncurry Entry) . zip
      -- = Line . Data.List.zipWith Entry
```

This implementation is not ideal, since it would silently drop excess groups if any of the lists of groups is longer than the respective other. The actual implementation throws an error in this case.

2.4.2 TEI

In contrast to the definition of Ding dictionaries, TEI dictionaries shall be introduced from the top down. Also, the definitions will not be illustrated with concrete examples; these tend to grow quite large, as seen in the introduction. Because not only general TEI but also FreeDict TEI are quite liberal in structure, the TEI syntax as specified in the following is even more restricted to accommodate for the needs of the Ding dictionary.

As noted in the introduction, TEI is specified in XML. TEI shall therefore primarily be defined along the XML elements it is made of. XML itself is not introduced; the definitions shall be quite explicit, so even if XML is unbeknownst to the reader, they should be able to grasp the definitions. Further, the TEI AST shall be developed alongside the XML it corresponds to.

In order to define TEI entities as XML elements, we use a simple custom XML schema language.

Syntax definition 2.4.1 (XML element schema)**.** *The schema of an XML element* `el` *is made up of the opening tag (*`<el>`*), a content schema and the closing tag (*`</el>`*), in this order. The opening tag may contain attributes with placeholder values, which are written in capital letters, or fixed values, which contain small letters. These attributes are considered mandatory.*

Syntax definition 2.4.2 (XML content schema)**.** *The content schema is made up of either placeholder text or a list of element schemata. Any of the the latter may be* redacted*, in which case an empty-element tag shall be used (which may contain attributes, just like a regular element schema).*

Further, any redacted *element schema may be enclosed in brackets (*`[]`*)—to indicate that the presence of a corresponding element is optional—or in braces (*`{}`*)—to indicate that there may occur a list of corresponding elements, of any length (including 0).*

Note that using this schema language, regular empty elements are not permitted. We will not need them.

The interpretation of element schemata is quite natural: An element is valid in a schema iff the schema can be transformed into it by first expanding bracket- and brace-enclosed redacted elements to a list of such redacted elements—of a length that the brackets/braces permit—and thereafter replacing placeholder attribute values and placeholder text with any specific text and finally replacing redacted element schemata with any concrete elements. In the case of a set of related element schemata, redacted element schemata may only be replaced according to the corresponding schema in the set, if present. An element schema corresponds to a redacted element schema iff the names match and the latter's specified attributes are a subset of the former's, where potential fixed values must match and any fixed value must be retained during the replacing.

In the following, we shall define a set of element schemata describing the TEI language, that is, the set of TEI dictionaries (of the restricted form that we consider here). Some of the occurring redacted element schemata will not be defined as proper element schemata. This will in particular be the case with any redacted element schemata representing annotations. This is not to indicate that these redacted element schemata may have arbitrary content though; they should be rather considered as unspecified here. If desired, the corresponding element schemata may be inferred from the code accompanying this thesis.

We start on the top level.

Definition 2.4.5 (TEI dictionary)**.** *A* TEI dictionary *is composed of a header and a list of* TEI entries*:*

```
<TEI xmlns="http://www.tei-c.org/ns/1.0" version="5.0">
  <teiHeader xml:lang="en" />
  <text xml:lang="SRCLANG">
    <body>
      {<entry />}
    </body>
  </text>
</TEI>
```

Note that TEI entries are distinct from Ding entries. In the context of TEI, TEI entries may be simply referred to as entries.

The TEI header is notably larger than the Ding header and shall not be explicated here. Because we only intend to represent TEI dictionaries that are derived of Ding dictionaries, we do not need to represent the rather complex TEI header explicitly in the TEI AST. Instead, we can later directly derive the TEI header's XML form from the Ding header.

Therefore, using a soon to be defined `Entry` type, the `TEI` AST type can be defined as follows:

```
type TEI = Dictionary Ding.Header TEI.Entry
```

As announced in subsection 2.2.2, a TEI dictionary generally adheres to the structure of a traditional dictionary, that is, it maps key units—*headwords*—in a source language to (potentially several) sets of value units in the target language. A single entry essentially represents a single such mapping, but allows for a little more structure and information:

Definition 2.4.6 (TEI entry)**.** *A* TEI entry *has a unique identifier and consists of a* form *that most prominently contains the headword, a list of* senses *to this headword, and optionally some grammar information pertaining to all senses of the headword:*

```
<entry xml:id="ID">
  <form />
  [<gramGrp />]
  {<sense />}
</entry>
```

The identifier by convention should be of the form `HW.n`, where `HW` is the headword of the entry and `n` is a positive decimal number.

Both the translations and any further annotations will be allotted to specific senses of a headword and therefore to be found in a `sense` element. Note that a `gramGrp`, if present, contains a non-empty list of elements specifying grammatical properties. This is why the Haskell type is slightly different in structure:

```
data Entry = Entry
  { entryIdent    :: Ident
```

```
  , entryForm     :: Form
  , entryGrammar  :: [GrammarInfo]
  , entrySenses   :: [Sense]
  }
```

We will later want to refer to specific entries within TEI; therefore, unique identifiers are essential. Note that the headword inside the form cannot serve this purpose; a TEI dictionary may very well contain several *homographs*[12].

Note further that any TEI entry resulting from the Ding dictionary will contain exactly one *sense*. While Ding does not offer a directly corresponding syntactical element, one might consider to group certain *homographs*. I discussed this with the people of the FreeDict project; however, the question remains open. Hence, the structure for several senses is provided, but not used yet.

Definition 2.4.7 (form)**.** *A* form *contains a single unit representing the headword of an entry, potentially together with some related forms, which are specified using nested* form *elements that adhere to a different schema:*

```
<form>
  <orth>HEADWORD</orth>
  {<form type="TYPE"/>}
</form>
```

We shall use such nested `form` elements to specify inflected forms (`@type="infl"`) and abbreviations (`@type="abbrev"`) to the respective headword. This motivates the following Haskell type:

```
data Form = Form
  { formOrth      :: String    -- ^ headword
  , formAbbrevs   :: [String]
  , formInflected :: Maybe InflectedForms
  }
```

Definition 2.4.8 (sense)**.** *A* sense *groups information to a headword that is particular to a specific sense of that headword. It may contain a list of translations and essentially any annotations that may be found in a Ding dictionary. Such annotations refer to the headword (and the particular sense); annotations that exclusively refer to a translation may be annotated within a translation element. Further, it may contain references (`xr`) to other entries.*

The `sense` element is subject to the following schema, with the added requirement that it must contain at least one nested element:

```
<sense>
  [<gramGrp />]
  {<usg />}
  {<cit type="trans"/>}
  {<cit type="example"/>}
  {<xr type="TYPE"/>}
  {<note />}
</sense>
```

The ability to refer to other entries shall be used to link entries together that stem from the same Ding line (`@type="see"`) or Ding entry/group (`@type="syn"`). Note that the latter is necessary because we will have to split up Ding entries into several TEI entries whenever there is more than one unit in the source group of that entry, since TEI entries only permit a single keyword.

[12] See footnote 1 on page 10.

The corresponding Haskell type quite naturally reproduces the structure of the above schema, albeit again with the exception of `gramGrp`.

```
data Sense = Sense
  { senseGrammar      :: [GrammarInfo]
  , senseUsages       :: [Usage]
  , senseTranslations :: [Translation]
  , senseExamples     :: [Example]
  , senseReferences   :: [Reference]
  , senseNotes        :: [String]
  }
```

Note that grammar information can both be annotated at the sense and at the entry level. Naturally, it should be specified at the sense level if it is particular to a sense, and at the entry level if it pertains to all senses of the respective entry.

At last, let's have a look at how translations are represented:

Definition 2.4.9 (translation)**.** *A* translation *is a single unit together with some annotations pertaining to that translation:*

```
<cit type="trans">
  <quote xml:lang="TGTLANG">TRANS_UNIT</quote>
  [<gramGrp />]
  {<usg />}
  {<cit type="abbrev"/>}
  {<cit type="trans"/>}
  {<cit type="example"/>}
  {<note />}
</cit>
```

Note how the above schema is quite similar to that of a sense. Both allow for representation of most of the Ding units' annotations. The Haskell type again quite naturally represents the above structure:

```
data Translation = Translation
  { translationOrth      :: String
  , translationGrammar   :: [GrammarInfo]
  , translationUsages    :: [Usage]
  , translationAbbrevs   :: [String]
  , translationInflected :: Maybe InflectedForms
  , translationNotes     :: [String]
  }
```

2.5 Annotations

The Ding dictionary contains annotations, of many different kinds. As noted in subsection 2.4.1, they are generally present on the level of units.

These annotations can mostly be specified in a quite similar form in TEI; therefore, we only define one family of data types for them. Nonetheless, the Ding syntax shall be used to introduce them:

Parenthesis expressions

Ding units may contain text enclosed in parentheses ("`()`"). We differentiate three cases:

- Prefixing parenthesis expressions are considered prefix collocations, that is, optional text.

- Infix parenthesis expressions are equally considered optional text, but because they cannot be properly represented in TEI, they shall be retained in verbatim as part of the unit's text.
- Suffixing parenthesis expressions are considered a generic note. In practice, they may have a lot of meanings, including that of an optional suffix. In most cases, the latter does not apply though; therefore, we use the catch-all note term.

Grammar annotations

Grammar annotations are enclosed in braces ("`{}`"). They may contain one or more grammar keywords or expressions, separated by semicola, commata or slashes.

Individual grammar annotations are of the type `GrammarInfo`, which may be considered a grammar AST—for single grammar annotations. It represents both single grammar keywords—of type `GramLexCategory`—such as `{f}`, and grammar expressions such as `{+Dat. [ugs.]}`[13].

As apparent from the definition of the `Unit` type in subsection 2.4.1, the set of grammar annotations to a unit are—like most annotations—encoded as a list. One might consider to instead encode all grammar information of a unit in a single tree data structure. While this might seem advantageous—in particular during further processing, as happens during enrichment—I have decided against this, for two reasons: First, both in the Ding dictionary and in TEI, grammar annotations occur in lists; we would hence need to construct that tree, only to deconstruct it later—in most cases, the list of annotations may in fact be be transferred unaltered (i.e., the enrichment has no effect).[14] Second, most units have very few or no annotations, it is consequently a waste of both time and memory to annotate each of them with a full grammar tree.

Inflected forms

Inflected forms are equally enclosed in braces. They occur exclusively on the English side and are of the form `{simple_past; past_participle}`, where `simple_past` and `past_participle` are made up of a comma separated list of simple past and past participle forms, respectively.

The only means of differentiating inflected forms from a set of grammar annotations—which are also enclosed in braces and may also be separated by semicolon—is to rely on the above mentioned grammar keywords.

Usage annotations

Usage annotations are enclosed in brackets ("`[]`"). In contrast to grammar annotations, they may contain arbitrary text—provided any contained brackets, braces and parentheses are matched. However, syntax analysis has shown that most usage annotations occur repeatedly, the most frequently occurring contender being `[ornith.]` with 8,971 occurrences.

TEI allows for the subdivision of usages into several categories such as *domain*, *register* or *region*. Such subdivision needs unfortunately to happen manually, that is, a function needs to be written that performs the categorization based on the concrete usage string. Despite the high number of occurrences for many of the usages, their overall number is quite large; therefore, categorization remains unfinished. Usages that are not categorized yet or are considered unfit for all of the standard categories are specified as being of the fallback *hint* category.

[13] Unfortunately, usage annotations to grammar annotations, as seen here, cannot be represented in TEI.

[14] One might consider it an advantage to naturally get a fixed order on the annotations, when translating to TEI (e.g., part of speech before gender). However, one might also prefer the original order should to be preserved.

Abbreviations

Abbreviations are enclosed in slashes, as in "`example /ex./`". In some cases, they may also be of the form "`/ abbrev /`".

Since slashes may also occur as simple alternative indicators, as they usually do in written text, the recognition of abbreviation annotations is generally non-trivial. See subsection 3.3.1 on how it is done.

References

Units may be annotated with references to other units—which are not guaranteed to occur in the dictionary—in the form `~word`.

These references can be easily represented in TEI. Note that, while the representation uses mostly the same TEI syntax as the references generated during translation (see section 3.4), they should not be confounded with these.

Angle-bracket expressions

Angle-bracket expressions are exceptions by several respects. Syntactically, they are enclosed in between angle brackets ("`<>`"). In contrast to all other annotations, they seem to apply to groups, not units.

According to the obsolete Ding syntax specification [34], they show wrong or old spellings of headwords. Syntax analysis has shown though that this is rather rare; in practice, they mostly show alternative (valid) forms or synonyms, sometimes also a singular form to the annotated group, which contains plural forms.

Due to the many different meanings, angle-bracket expressions are dropped during parsing. One might consider to use a heuristic instead; in particular most singular forms are easy to relate to their respective plural form. Also, one might consider some of the occurrences bugs and fix them accordingly and/or report them upstream.

Chapter 3

Implementation

Now that we are aware of the structure of both the Ding dictionary and TEI, we shall be concerned with the transformation of the former into the latter, and the implementation thereof. After all, the main goal of this thesis was stated as writing a translation *program*.

3.1 Structure of the Program

The program needs to perform the following basic steps:

1. Read the Ding input and represent it in an internal data structure (abstract syntax tree / AST).
2. Translate that Ding AST to a TEI AST.
3. Convert the TEI AST to TEI XML.

Additionally, there will be an initial preprocessing step, and an enrichment step, located between steps 1 and 2 from above.

After introducing the folder structure right below, we shall have a closer look at the above named steps.

3.1.1 Module & folder structure

In the following, the folder structure is briefly introduced.

Upon request from the FreeDict project, the code is made part of the main FreeDict tools git repository [14] and located at `importers/ding2tei`. As of now, it lives in a separate branch, `ding2tei-haskell-rewrite`.

The top level structure—below `importers/ding2tei`—is loosely based on a "semi-standard" in the Haskell Wiki [21]:

- `src/`: The code needed to translate the dictionary.
- `utils/`: Code that is to be used for syntax analysis.
- `doc/`: Textual documentation describing some of the more important aspects of the two languages' syntax and the program.
- `todo/`: Less well structured documentation on issues that remain to be (fully) solved.

Below `src/`, the code is organized as follows:

- `preprocess/`: The preprocessor scripts.
- `Language/Ding`: All code pertaining to or working on the Ding dictionary exclusively.
- `Language/TEI`: All code pertaining to or working on TEI dictionaries exclusively.
- `Data/NatLang`: Data structures that are common to both the Ding and TEI dictionaries.
- `App/`: The code to translate from the Ding AST to the TEI AST.

Most of the program is to be written in Haskell; therefore, the folder hierarchy largely matches the hierarchy of Haskell modules[1].

3.2 Preprocessing

As noted earlier, the Ding dictionary contains many inconsistencies. These are considered bugs in the input data and we therefore do not want to have the main program handle them. Instead, we shall handle these inconsistencies in a preprocessing step. The main program can then expect a rather well formed syntax and does not need to account for the lot of special cases that might otherwise arise.

Another notable reason for the separation is that such bugs should ideally be reported upstream and fixed there. If they are fixed in a separate preprocessing step, these fixes can easily be incorporated upstream.

Tool choice

For the preprocessing step, the *sed* [9] language shall be used, as is also used for the identification of syntax (see subsection 2.3.1). Sed is an acronym for *stream editor* and is quite similar in syntax to the *ed* [32] editor. Sed is most known for its ability to search and replace using regular expressions, which we will primarily use.[2] Note though that sed can do much more and is in fact Turing-complete [3].

To be precise, we shall use *GNU sed*, which has some useful extensions, with the `-E` flag, which specifies to use POSIX extended regular expressions.

Linguistics

A notable part of the syntax errors required a certain knowledge of either the English or German language. Furthermore, there are also pure linguistic errors, such as typographic ones—which both may be spotted occasionally by hand or by searching for certain patterns using sed. When unsure in linguistic questions, I mostly relied upon Wikipedia and the Wiktionary [57], in some cases other on-line dictionaries. In one particular case, this did not suffice:

```
wenig zersetzte Streu fressend; makrohumiphag {m} [zool.] :: macrohumiphagous;
    ↪ macrosaprophagous
```

The `{m}` annotation usually would indicate that `makrohumiphag` is a male noun, which seems unlikely from both the context and the lower case spelling. In the lack of a definition of "`makrohumiphag`" or any of its given translations in the common places, I was finally able to find "`macrohumiphagous`" used in a biology paper—as an adjective [39].

[1] The Haskell module structure is largely modelled after one proposed in the Haskell Wiki [20].

[2] The searching capabilities of sed essentially match those of the previously introduced grep program, which actually got its name from the `g/re/p` ed command that allows to search for the ***r***egular ***e***xpression `re`. [48]

3.3 Lexing and Parsing

The first step of the actual Haskell program is step 1 as specified in the introductory section 3.1. The task at hand is to convert a stream of characters to the Ding AST as defined in subsection 2.4.1. As the heading indicates, this step is usually separated into two successive smaller steps, namely *lexing* (or *scanning*) and *parsing*. There do, however, also exist so called *scannerless parsers* [53].

Lexing is the process of converting the input stream of characters into a stream of so called *tokens*, such as separators, keywords and literals. In the context of the Ding dictionary, such would be for example `::`, `pl` and `flower`, respectively.

A parser usually operates on such a stream of tokens and—in abstraction from the concrete syntax—generates an abstract syntax tree.

Both the lexer and the parser could be written by hand, but for nontrivial languages, as is the Ding language, it is much more handy to use additional tools that are specifically targeted to such use.

Tool choice

There are two common options:

1. Using lexer and parser generators that allow a rather natural specification of the input syntax and from that generate (Haskell) code.

2. Using a parser combinator library that eases the writing of a parser right in Haskell. Such a library usually directly works on the stream of characters, so requires no separate lexer.

In the Haskell ecosystem, the common choice for a lexer generator is *Alex* [7] and for a parser generator *Happy* [28]. These are essentially Haskell derivations of the well known *lex* and *yacc* tools for the *C* programming language.

The most prominent parser combinator library in Haskell is *Parsec*. However, there exist a number of others, in particular *Megaparsec*, a fork of Parsec that particularly claims to be more efficient [30].

In the following, Alex+Happy and Megaparsec shall be compared. Happy may generate a parser to any *LALR(1)*[3] grammar, which is to be specified in a form quite similar to the *Backus-Naur-Form* (*BNF*) [42].

In contrast, Megaparsec allows to parse any *LL(1)*[4] grammar. Neither of the language classes LL(1) and LALR(1) is a subset of the other [4]; however, many languages can be expressed by grammars of both types, including the language of the Ding dictionary. Both grammar classes—and the corresponding tools—have their respective advantages:

- In contrast to LL(1), LALR(1) allows for left recursion [52], which often helps to specify the grammar in a more natural way.

- Megaparsec is a Haskell library; therefore, it allows for all the flexibility of Haskell, while Happy has its own specification language. For example, using Megaparsec, it is possible to define parametrised rules.

[3] LALR(1) stands for Look-Ahead LR parser (with 1 token look-ahead) and is a weaker form of an LR parser (***l***eft-to-right, ***r***ightmost derivation) that can be implemented with notably less memory than a general LR(1) parser. [51]

[4] LL(1) signifies that parsing happens from ***l***eft to right and produces a ***l***eftmost derivation. [52]

- Megaparsec may be used without a separate lexer. Most notably, this reduces the number of different components and technologies. However, this also has the effect of functionality usually specific to the lexer being interwoven with such that is specific to parsers, which can be an advantage.
- When used without a separate lexer, it is cumbersome to match keywords using Megaparsec. When overlapping, the common prefix needs to be extracted, or else—at the cost of efficiency—keywords that are prefixes of other keywords need to be specified before these. Note that it is often desirable to subdivide lists of keywords that may occur in the same place into several sub-lists, using separate rules. The word *before* does relate to single keywords; therefore, the whole concatenation of those lists needs to be ordered accordingly.

  Note that this problem could likely be avoided by using another parser combinator library, for example *Earley* [10], which allows for parsing any context-free language [46].

The decisive issue—leading to the choice of Alex and Happy—was finally the ability to handle slashes as they occur in the Ding dictionary. As described in subsection 3.3.1, while these seem generally difficult to parse, Alex has certain features that allow to do so relatively easily. I was unable to find a viable solution for Megaparsec, except by altering the Ding syntax and modifying it accordingly during preprocessing.

Similarly, the whitespace handling as required by the Ding dictionary would need uncommon usage of Megaparsec. Usually, whitespace is considered as to be ignored separator, while in the Ding dictionary, it may be part of a unit's text, where the precise amount of whitespace (usually zero or one single space character) matters.

3.3.1 Lexing

As noted before, the first step in the construction of the AST is to convert a stream of characters to a stream of tokens, where a token essentially groups and labels a sequence of characters.

Introduction by example

Let's consider an example Ding line:

```
Dings {n} [ugs.] :: thingy; dingus
```

In this line, we wish to recognize the separators ("`::`", "`;`", "`{`", "`}`", "`[`", "`]`") and the terminating newline character as individual tokens and further all occurring *words* as text tokens, except for "`n`".

Definition 3.3.1 (word)**.** *A* word *is a maximal sequence of characters that does not contain* special characters*—such as separators—and no whitespace.*

The "`n`" is to be matched as grammar keyword. Note that the lexer is not able to infer this from the surrounding braces, instead it simply knows that the word "`n`" is a grammar keyword; if it were to occur outside of braces, it would equally be recognized as such. It is the parser's job to recognize grammar tokens enclosed in braces as actual grammar annotations. In contrast, "`ugs.`" is not recognized as keyword, because usage annotations—as indicated by the brackets—may in general contain arbitrary text (see section 2.5).

The main reason to not allow whitespace in text tokens is to allow for the recognition of keywords that are not separated by special characters.[5]

The token type is now defined as follows:

[5] We have not yet seen such keywords; in fact, the only such keyword is currently "`to`", which is often prefixed to verbs.

```haskell
data Atom = NL          -- \n
          | LangSep     -- ::
          | Semi
          | OBrace
          | CBrace
          | OBracket
          | CBracket
          | GramKW GramLexCategory
          | Text String
          | ...
```

Note that the type is not named `Token`—we shall soon define a wrapper type around `Atom` named `Token`. Note further that the `GramKW` constructor takes a value of type `GramLexCategory` which is not simply an enumeration of keywords, but rather the type of a quite flat grammar AST (see section 2.5). AST construction is generally the job of the parser; however, since it can already be done here, there is hardly a point in inventing a new intermediate type for that a direct injective correspondence to the grammar AST would exist.

Composability

One particularity that was already seen with "`n`" is that some tokens may be only of special meaning in special context, whereas in other context they should be treated like a regular text token. Further, in the latter case, it should also be possible to join these tokens with their surrounding (text) tokens. This is particularly also the case with commata ("`,`"), as exemplified by the following lines:

```
Abschließend ... :: To conclude, ...
Stampfkartoffeln {f,pl} :: mashed potatoes
```

In the first line, the comma is simply part of the unit, whereas in the second line, it serves as a separator in between braces. We also are reminded of the fact that units may be composed of more than one word, as defined above.

First, to allow for treating all tokens as text tokens, we define a function to convert tokens to their respective string representations:

```haskell
atomToString :: Atom -> String
```

Note that this requires the correspondence function from strings to tokens to be injective, or else `atomToString` cannot generally give the original string—being the inverse of that function (restricted to the latter's image). Non-injectivity may actually be acceptable in certain cases—when two strings are considered equivalent or one of them a misspelling of the other.

Second, to enable joining of tokens, we might want to define the following function:

```haskell
joinAtoms :: Atom -> Atom -> Atom
```

If taking this route, we soon encounter the question of how to separate the joined tokens by whitespace. While regular text tokens are usually separated by a single space, a comma for example is usually not preceded by space and succeeded by a single space. However, if serving as a (German) decimal point, there shouldn't be any space inserted. The most general way to deal with this issue is therefore to remember the spacing between tokens. This brings us to the previously announced `Token` type:

```haskell
data Token = Token String Atom
```

The first argument of the `Token` constructor is to denote the whitespace preceding[6] the token. Note that newline characters are not considered whitespace in this context, they are regular tokens.

With this type, we should now be able to define a `joinTokens` function. Considering such a function as an operator, we realize that it is associative, and therefore we may consider `Token` as a semigroup:

```
instance Semigroup Token where
  (Token ws1 atom1) <> (Token ws2 atom2) = Token ws1 $
    Text $ atomToString atom1 ++ ws2 ++ atomToString atom2
```

The result of joining two elementary tokens is naturally a text token—if a token has a special meaning, it looses that when joined with surrounding tokens. The associativity law is clearly fulfilled.

It might come in handy to also have a neutral element, and thereby a monoid structure. Such in particular provides us with the `mconcat` function to merge arbitrary lists of tokens into a single token. As neutral element, we might be tempted to choose an empty text token (`Token "" (Text "")`); however, we can easily verify that this is not a left identity:

```
Token "" (Text "") <> Token "␣" (Text "Ding")
  = Token "" (Text "␣Ding")
```

Instead, we introduce a separate constructor for `Token`, namely `EmptyToken`, that takes no argument, and amend the definition of `(<>)` to treat `EmptyToken` as neutral element. This provides us with the desired `Monoid` instance:

```
instance Monoid Token where
  mempty = EmptyToken
```

Note that this `Monoid` instance may only be defined together with a corresponding `Semigroup` instance as defined above. This allows for the binary `Monoid` operator, `` `mappend` ``, to be implicitly defined as `<>` from the `Semigroup` instance.

Alex specification

Alex allows to perform token recognition using regular expressions. An Alex specification at its core consists of a list of *rules*, which in the simplest case are made of a regular expression and an *action*—a function to convert any matched string to the corresponding token—each. The matching occurs using the maximum-munch strategy, that is, the prefix of maximal length is matched. Thereafter, Alex continues with the remaining suffix. In case of overlapping rules for some input, Alex chooses the rule that matches the longest prefix. If these lengths are equal, earlier rules are preferred. [7]

To avoid the explicit handling of whitespace in each rule, we introduce an intermediate token type representing either whitespace or an `Atom`:

```
data SimpleToken = RegularToken Atom
                 | Whitespace String
```

A stream of `SimpleToken`s can later be easily converted to a stream of tokens using a function of the following type:

```
mergeWS :: [SimpleToken] -> [Token]
```

[6] We could equally well annotate the whitespace *succeeding* a token. Also, one might consider annotating a count of space characters instead—tabs, not to speak of \v et al., are rare—or even only the presence of whitespace using a boolean value.

This function is to merge each `Atom` with any directly preceding `Whitespace` into the corresponding `Token`—final `Whitespace` is to be dismissed.

A subset of Alex' rules may now look as follows:

```
[$white # \n]+    { Whitespace }
\n                { const $ RegularToken NL }
::                { const $ RegularToken LangSep }
\{                { const $ RegularToken OBrace }
```

As seen, on the left are regular expressions, and on the right—in braces—are the corresponding actions. Remember, these are functions that are provided with the respectively matched string. To any sequence of non-newline whitespace[7], we simply apply the `Whitespace` constructor. For the separators, we instead use the `const` function because we do not need the string value of the matched token to determine the token value.

This is not the case for text tokens. As noted earlier, as text tokens we intend to match words, that is, sequences of characters that do not contain separators or otherwise special characters, nor whitespace.

Alex also allows for the specification of character sets and macros, so we define a macro for words[8]:

```
$textChar = $printable # [$specialChar $white]
@word = $textChar+
```

Ignoring for now that some words should be recognized as keywords, we may now define a rule for `Text` tokens:

```
@word      { RegularToken . Text }
```

Keywords

There are two common ways to perform the keyword identification in the lexer:

1. We can give regular expressions for individual keywords, or—less likely—sets of related keywords.
2. We can change the rule for `@word` above by providing it with an action that distinguishes keywords from regular text tokens.

The second option allows for decomposition, make the Alex specification smaller. Unfortunately, there is an issue with this second option—being that there are also multi-word keywords, such as "`no pl`". This issue could be resolved by identifying each of such a multi-keyword's constituent words as individual keywords, and delegating the recognition of their co-occurrence to the parser. This would introduce a level of detail to the parser that I consider undesirable. The first option is thus preferred and chosen, manifesting for example as follows:

```
"f"        { const $ RegularToken $ GramKW $ Gender Feminine }
```

Note that all keyword rules need to be listed before the general text token rule as introduced earlier. This is because most keywords also match this rule—otherwise we could never have considered the second option above. The result is that single words that are keywords are—due to being specified earlier—recognized as such. On the other hand, words that are not keywords, but have a prefix which is, are properly recognized as regular text tokens, due to Alex' preference of the longest matching sequence.

[7] `$white` is a predefined character set representing any whitespace character and `#` is the set difference operator. [7]

[8] `$printable` is a predefined character set of printable characters. [7]

Multi-word keywords

Multi-word keywords—being the reason for individual keyword rules—require a little more caution. Consider the keyword "`no pl`", where one might want the corresponding rule to look as follows:

```
"no␣pl"    { const $ ... }
```

If we do now provide the Alex generated lexer with input prefixed by the sequence of words "`no plural`", it will happily[9] recognize our keyword "`no pl`", succeeded by the regular text token "`ural`". This is not what we want. The problem with multi-word keywords is that for partial matches as just exemplified, there is no generic multi-word matching rule that takes preference, as is the case with single-word keywords. We therefore need to specify explicitly that the last word of a multi-word keyword is ended.

Fortunately, Alex provides us with a mechanism to do just this: *contexts*. For each rule, we may give a left and a right context, where the former must be a character set, while the latter may be an arbitrary regular expression. These contexts have the effect that the rule only matches when—next to the matching of the main regular expression—the preceding character matches the left context character set and the succeeding character stream matches the right context regular expression. Note that for efficiency reasons it is advised to not make extensive use of right contexts. [7]

For our purpose, we do only need to check the single next character, it should be one that may not occur in a word. We therefore define a character set for the complement of a word's characters:

```
$word_end = ~$textChar
```

Subsequently, we may use this character set—also being a regular expression—to enhance the "`no pl`" rule as desired:

```
"no␣pl" / $word_end    { const $ ... }
```

Keeping track of positions

The lexer as introduced above should not fail on any input. It is, however, very well possible that the sequence of tokens it generates is not accepted by the parser. In this case, we would like to be told the position of the offending token in the input.

Alex allows for keeping track of these positions by using the "posn" *wrapper* instead of the "basic" *wrapper* that we used until now. Using the "posn" wrapper changes the required type of actions to the following:

```
AlexPosn -> String -> Token
```

An `AlexPosn` is a triple of absolute position, line and column. Since we are happy[10] with the line and column, we define a type that only stores these:

```
Position = Position Int Int
```

Further, we want to provide each `Token` with such a `Position`; therefore, we change its definition to the following:

[9] happily: from happy, not Happy [55]

[10] happy: adjective to happiness [56], not to be confused with Happy, which is a homograph, except for capitalization.

```
Token = Token String Position Atom
      | EmptyToken
```

We also need to modify the definition of the intermediate `SimpleToken`:

```
data SimpleToken = RegularToken Position Atom
                 | Whitespace String
```

Note how we do not record the position for whitespace; the position of a `Token` should be that of the wrapped `Atom`.

To avoid handling the position individually in each rule's action, we define a wrapper for regular tokens:

```
regularToken :: (String -> Atom) -> AlexPosn -> String -> SimpleToken
regularToken f p s = RegularToken (toPosition p) (f s)
```

The function `toPosition` translates a `AlexPosn` to a `Position`. With that new function, we now only need to wrap each regular token action in a call to `regularToken`.

For whitespace, since we are not interested in the position, we may simply give the action `const Whitespace`—thereby dropping the undesired first argument.

Specialty: slashes

Slashes serve multiple purposes in the Ding dictionary. These are exemplified in the following units:

```
to adopt/pass an amendment
der zeitliche Ablauf / die zeitliche Abfolge von etw.
greatest lower bound /glb/ (set theory)
```

We may infer three corresponding different usages of slashes:

1. *Strong slashes*, usually separating single words, with the meaning of an alternative between these.

2. *Weak slashes*, sometimes separating more than single words, equally with the meaning of an alternative between these.

3. *Opening* and *closing slashes*, surrounding an abbreviation (see section 2.5).

In the example units above, we might identify all slashes by their surrounding spacing. This is the basic idea, which shall be refined a little: We say that a slash is *left-free* or *right-free* if the preceding or succeeding character, respectively, is either whitespace or within a particular set of characters that serves a similar purpose. For example, a slash is also left-free, if it is preceded by an opening parenthesis; similarly a slash is right-free, if it is succeeded by a closing parenthesis.

With these definitions we may say that a strong slash is neither left- nor right-free, while a weak slash is both left- and right-free, and opening and closing slashes are exclusively left- or right-free, respectively.

Given character sets `$slashLeftFree` and `$slashRightFree`, we may now easily differentiate the different kinds of slashes in Alex, using the left and right contexts introduced earlier:

```
  $slashLeftFree ^ "/" / $slashRightFree  { ... WeakSlash }
  $slashLeftFree ^ "/"                    { ... OSlash }
                   "/" / $slashRightFree  { ... CSlash }
                   "/"                    { ... StrongSlash }
```

Note that there are actually some more particularities to be taken into account; however, these shall not be discussed here. For details on these, consult the file `doc/ding.slashes` and/or the concrete Alex specification. Note further that the above made distinction between different types of slashes originally stems from exemplary observation and is not used consistently. In particular, slashes that are strong by syntax may separate not only single words but groups of words; conversely, weak slashes frequently separate single words. It is therefore not in general possible to automatically infer the left and right scopes of strong and weak slashes. Additionally, there is quite a number of opening and closing slashes that do not belong to an abbreviation. Most of such cases can rather easily be identified in the parser, which allows for fixing them individually in the preprocessor.

3.3.2 Parsing

Given a stream of tokens as produced by the lexer from above, the next step consists of constructing a Ding AST from that token stream.

As noted earlier, we use the Happy parser generator for this purpose. Happy essentially allows to specify a grammar in Backus-Naur-Form [42], together with some Haskell code that describes how the grammar's rules correspond to the to be built AST. [28]

I initially specified the grammar for a whole Ding dictionary, that is, its header and a list of lines. Noticing that this caused the parsing step to consume a lot of memory and therefore most likely not to happen lazily, I decided to parse the header separately and have Happy only parse a single line. Concerning the memory usage, this unfortunately did not help much, since the remainder of the program requires to read in the whole Ding dictionary before it may start writing the body of the resulting TEI dictionary, see section 4.1.3.

The *start symbol* of our grammar is hence `line`, for which a single rule exists:

```
1 line : groups '::' groups
```

In this rule, `line` and `groups` are non-terminals, while `'::'` is a terminal.

Terminals—also *Happy tokens*—are defined in an initial section as follows:

```
1  %token '\n'             { Token _ _ NL }
2         '::'             { Token _ _ LangSep }
3         '{'              { Token _ _ OBrace }
4         ...
5
6         tok_gramPOS      { Token _ _ (GramKW (PartOfSpeech _)) }
7         tok_gramGender   { Token _ _ (GramKW (Gender _)) }
8         ...
9
10        tok_text         { Token _ _ (Text _) }
```

To each Happy token, there is associated a Haskell `Token` pattern, which may contain underscores to match any fitting value. In particular, we ignore the `Token`s' position and preceding whitespace. Further, we do not match each of the grammar keywords individually, but rather classify them. Note how for the separators, their Happy token names match the corresponding input string. This is not at all necessary, albeit quite common and useful. In particular, it is therefore not necessary to introduce all of the separators' token names here.

AST construction

In the following, the AST construction alongside the grammar rules shall be exemplified using the single `line` rule:

```
line :: { Line }
line : groups '::' groups      { makeLine $1 $3 }
```

We need to think of every *symbol*—*terminal* or *non-terminal*—as holding a value, of a fixed type. The non-terminal for which rules are to be defined may be annotated with the corresponding Haskell type beforehand. Next to each rule, we may—in braces—give Haskell code that constructs a value of the left hand's type from the values of the right hand. Values on the right hand may be accessed as `$1` through `$n`, in order of appearance. The value of a terminal is naturally that of the corresponding `Token`[11], as matched in the preamble, while the value for a non-terminal is either defined explicitly, preceding the corresponding rules, or inferred from these.

Knowing that the non-terminal `groups` is of type `[Group]`, we may infer that `makeLine` needs to have the type `[Group] -> [Group] -> Line`. In fact, we have seen a definition of `makeLine` at the end of subsection 2.4.1.

Note how the `line` rule does not explicitly represent the syntax of Ding—the `groups` non-terminal represents any number of "`|`" separated groups. We might alternatively give a `line` rule as follows:

```
line :: { Line }
line : group '::' group
     | group '|' line '|' group
```

This would ensure the numbers of groups on both sides match; however, the AST construction would need to be specified in a quite unnatural way, since the groups forming an entry in a line are matched by their order from left to right, not from outermost to innermost or conversely. Another alternative would be to use an attribute grammar, which Happy supports, to essentially count the number of groups on the left side and mandate that exact number on the right side. This approach shall not be discussed further.

Parsing of units

Next to the simple example of a line, we shall discuss in a little more detail the parsing of units—consisting each of a unit string and associated annotations.

Given the syntax description from subsection 2.4.1, consider the following examples:

```
Dings {n} [ugs.]
mashed potatoes {pl}
```

From the grammar point of view, a unit consists of a sequence of text tokens and a succeeding sequence of annotations, of potentially differing types. We might describe a unit by the following set of rules:

```
unit : unitText
     | unit gramAnnot
     | unit usageAnnot
     | ...
```

We would need separate rules for `unitText`—specifying a sequence of text tokens[12]—and all the different types of annotations.

We might further give the AST construction code as follows:

[11] In fact, Alex also allows for associating tokens with only constituents of the corresponding Haskell tokens. We do not use this ability though, since for most tokens, we want in particular to be able to access the annotated preceding whitespace.

[12] `unitText` might actually also include non-text tokens, such as commata or grammar keywords. For reasons of simplicity we ignore these here.

```
unit : unitText          { fromToken $1 }
     | unit gramAnnot    { $1 `plusGramAnnot` $2 }
     | unit usageAnnot   { $1 `plusUsageAnnot` $2 }
     | ...
```

Here, `fromToken` creates a simple `Unit` without any annotations from a `Token`, assuming that `unitText` uses the `Token`'s monoid structure to yield a single token. Further, the `plus*Annot` functions each add an annotation of the respective type to a unit.

Unfortunately, the Ding syntax—concerning units—is not actually as simple as depicted in subsection 2.4.1. In particular, annotations may also occur in the midst of a unit, as exemplified by the following Ding units:

```
Legende {f} zum Bild
tyre [Br.] / tire [Am.] equipment
```

Considering the first example, the `{f}` annotation is clearly meant to apply to the preceding text, "`Legende`". Seeing the succeeding text, we might consider this a collocation and therefore make it a separate annotation, that is, treat the unit similarly to:

```
Legende {f} (zum Bild)
```

This could not be generalized though, since in the second example, the suffix "`equipment`" cannot be seen as an optional collocation. Alternatively, we might consider ignoring the early location of the annotation, that is, consider it equivalent to:

```
Legende zum Bild {f}
```

Going through the list of units containing infix annotations[13], we may see that most of them may be validly interpreted this way. However, the clear contender is again the second example from above, particularly the presence of the slash, introducing two alternative—one might say contradicting—annotations. As explicated in subsection 3.3.1, handling slashes properly is difficult, if not impossible.

To circumvent this issue, we could consider infix annotations equivalent to suffix annotations iff there occur no commata[14] in the unit's text. While this approach seems to work for most practical cases, there are still exceptions. Hence—that is, due to a a lack of a sufficiently precise syntax description—I decided to ignore[15] any infix annotations. Note how this can still cause misinterpretations, as with the following units[16]:

```
Bildüberschrift {f} / Bildunterschrift {f} [print]
Buchführung {f} und Fakturierung {f}
```

In each of these units, both `{f}` annotations clearly are meant to apply to the just preceding word, while the respective later one would be considered by the parser as applying to the whole unit. Such special cases can unfortunately not be easily reliably recognized. There could surely be used a heuristic—e.g., depending on the number of annotations describing the same property such as `{f}` and `{n}`—but false positives or negatives would hardly be avoidable. The misinterpretation of the just introduced special cases is therefore accepted in favor of a simpler parser.

Further, expressions enclosed in parentheses need to be an exception to this rule. As outlined in section 2.5, infix parenthesis expressions are to be kept verbatim, as part of the unit's text, as for example in the following unit:

[13] E.g., by searching for the regex "`(\}|\]) [[:alpha:]]`"—catching *most* infix grammar and usage annotations.

[14] The `<>` separator is similarly problematic (see section 4.3) and should thus also cause infix annotations to be dropped.

[15] Infix annotations are permitted, but not included in the AST.

[16] The second example additionally presents "`und`" as another slash-like separator (of many), exemplifying the difficulty to provide a complete syntax description properly classifying infix annotations.

```
distance (measuring) sensor
```

Note how this is clearly not ideal; I am unaware though of a better option.

One might consider to split the above unit up in two, as follows:

```
distance sensor
distance measuring sensor
```

These units would then likely become part of the same group, even though they are not necessarily synonymous[17].

Implementation

Having decided on how to treat infix annotations, it remains to modify the parser accordingly.

The first idea is to just keep the rules for `unit` as introduced earlier, where it is defined as `unitText` followed by any number of annotations. It seems we only need to modify `unitText` to allow for infix annotations. We shall not go into the details on how to do this; there are several options—which all cause the same problem: Upon having `Happy` create a parser from the annotated grammar, it will complain about either shift/reduce of reduce/reduce conflicts, causing it to generate a faulty parser. This is because Happy is a LALR(*1*) parser—it has a *look-ahead* of *1* token. Having read for example a text token and stumbling upon a usage annotation, that is, the `OBracket` token, Happy is unable to decide whether that usage annotation should be assumed an infix annotation as part of `unitText` or a suffix annotation as part of `unit`.

There are several ways to solve this problem, the likely easiest being the following. First, we realize that a grammar allowing infix annotations can actually be specified quite naturally:

```
1 unit : text
2      | unit text
3      | unit gramAnnot
4      | unit usageAnnot
5      | ...
```

Note how the non-terminal `unitText` has been replaced by the terminal `text`[18], which now may also occur after annotations.

Unfortunately, the above rules for `unit` do not represent the semantic difference between infix and suffix annotations, contrarily to a hypothetical implementation of our first idea. We shall, however, see that this distinction can easily be provided for by the annotated Haskell functions. Similarly to the first attempt, we annotate the above rules with Haskell functions:

```
1 unit : text              { fromToken $1 }
2      | unit text         { $1 `plusToken` $2 }
3      | unit gramAnnot    { $1 `plusGramAnnot` $2 }
4      | unit usageAnnot   { $1 `plusUsageAnnot` $2 }
5      | ...
```

The only new function is `plusToken`. To demonstrate how this allows for the distinction of infix and suffix annotations, consider again the following example Ding unit:

```
Legende {f} zum Bild
```

The parser operates from left to right, that is, it first recognizes "`Legende`", then annotates that as a feminine noun. Next, it recognizes a text token. The presence of this text token clearly implies that all formerly recognized annotations were infix annotations and may therefore be thrown away[19]. This is hence exactly the job of `plusToken`—amend the unit's text by the newly

[17] In this particular example, the units likely may be considered examples; however, this cannot be generalized.

[18] Again, in practice, we'd also want to consider other tokens that do not introduce annotations, such as commata. For reasons of simplicity, we continue to assume that these do not exist.

[19] Parenthesis expressions need to be treated slightly differently.

read text token and ditch all its annotations.

3.4 Translation

Now that we are able to generate a Ding AST from the input file, the next step is to translate this AST to a TEI AST:

```
ding2tei :: Ding -> TEI
```

This function should assume the Ding dictionary to be directed, that is, the resulting TEI directory will have as source language the first language of the Ding dictionary.[20]

Remember the definitions of these ASTs:

```
type Ding = Dictionary Ding.Header Ding.Line
type TEI  = Dictionary Ding.Header TEI.Entry

data Dictionary header element
  = Dictionary
      header
      Language     -- ^ source language
      Language     -- ^ target language
      (Body element)

newtype Body element = Body [element]
```

As mentioned in subsection 2.4.2, the header of a TEI dictionary is just retained from the Ding dictionary, further the source and target languages also remain unchanged. Therefore, we only need to translate the body, that is, a list of Ding lines to a list of TEI entries.

One particularity that we need to take care of is the allocation of identifiers. As noted in subsection 2.4.2, these identifiers should bear the form `HW.n`, where `HW` is the headword of the entry and `n` is a positive decimal number. These numbers should be chosen minimal such that for a given entry all previous entries with the same headword are provided with a smaller number. We therefore should process the Ding lines (and entries) in order while maintaining a state—storing for each headword the last given number, or equivalently the number of previous occurrences of that headword. The natural way to do this in Haskell is by use of a *state monad*:

```
type IdState = State IdMap
type IdMap = Map String Int
```

For the state to be passed through, we use a standard map from the `containers` package and for the state monad, we use the standard state monad from the `transformers` package. For the map, any headword that is not in the map should be considered as having value 0 under the map.

Note that a state monad essentially has the following form:

```
type State s a = s -> (a, s)
```

It modifies a state of type `s` and in parallel produces a value of type `a`. Note further that the above defined `IdState` is not a full type, but instead takes as argument the type of the value that the state monad should produce.

Most of the computation should now happen in the `IdState` monad, which gives us the type for the translation of individual Ding lines:

[20] In order to obtain a TEI dictionary of the other direction, the Ding dictionary needs to have its content mirrored first, which can be achieved quite easily.

```
convLine :: Ding.Line -> IdState [TEI.Entry]
```

Given this, we may define

```
convLines :: [Ding.Line] -> IdState [TEI.Entry]
convLines = liftM concat . mapM convLine
```

and subsequently the main translation function:

```
ding2tei (Dictionary header srcLang tgtLang (Body ls)) =
  Dictionary header srcLang tgtLang (Body tes)
 where
  tes = evalState (convLines dingLines) Map.empty
```

Note that a Ding line may be translated to several TEI entries, which is why `convLines` needs to concatenate the results.

Conversion of a Ding entry

Because the Ding dictionary is a semantic dictionary, while TEI dictionaries are traditional dictionaries, we cannot simply convert a Ding entry to a TEI entry. Consider the following Ding entry (or rather, Ding line containing a single entry):

```
Ding {n}; Sache {f} :: thing
```

Our source language is German, our Ding entry thus contains two keywords. Since TEI does not allow for several keywords per entry, we need to create two TEI entries from the above single Ding entry. We are, however, able to provide them with links to one another, as described in subsection 2.4.2.

The implementation of the Ding entry conversion shall not be explicated here, it is quite similar to the conversion of Ding lines described below.

Conversion of a Ding line

Remember that a `Line` is essentially a list of entries. Hence, before defining `convLine`, we consider the function `convEntries`, assuming that there is a function `convEntry`. The natural definition is analogous to that of `convLines`:

```
convEntries :: [Ding.Entry] -> IdState [TEI.Entry]
convEntries = liftM concat . mapM convEntry
```

We may also specify this more verbosely as:

```
convEntries :: [Ding.Entry] -> IdState [TEI.Entry]
convEntries []       = return []
convEntries (de:des) = do
  tes1 <- convEntry   de
  tes2 <- convEntries des
  return $ tes1 ++ tes2
```

Note how a single Ding entry may result in several TEI entries, as explained earlier in this section.

The above definition of `convEntries` allows to provide the resulting TEI entries with unique identifiers. We do additionally want to use these identifiers though, to refer to other TEI entries. In particular, TEI entries that stem from the same Ding entry shall refer to one another as synonyms, and such that stem from the same Ding line, but different Ding entries shall refer to one another as related entries.

Currently, we are concerned with the translation of a list of entries (i.e., a line), where references to related entries are of interest, that is, Ding lines with more than one entry. Therefore, consider the following example line:

```
Ding | Dings | Krempel :: thing | thingy | things
```

In this case, there is a direct correspondence between Ding entries and TEI entries, no Ding entry results in multiple TEI entries. The above definition of `convEntries` would provide each of these three entries with an identifier, from left to right. If we do now want to provide the TEI entry bearing the headword "`Dings`" with references to the other two entries, we need to access in particular the identifier for "`Krempel`", which is to be calculated after the translation of the "`Dings`" entry is completed. Unfortunately, monadic transformations generally mandate an execution order, similar to that in imperative languages.

One way to solve this problem is by subdivision of the translation into two steps, where in the first, identifiers would be distributed to those Ding units that are to become headwords of TEI entries, while most of the Ding's structure is retained, and in the second step, the actual translation would happen, using these new identifiers to include references. This would require either defining a new intermediate language or adding some `Maybe` attributes to Ding units, which would initially be set to `Nothing`. Also, we'd have to traverse the AST twice in our code.

Fortunately, there is another way to solve the above problem. While monadic transformation, in particular when specified in `do` notation, looks quite similar to imperative statement sequences, it is different. A monadic action is still a regular function; it may be seen as a recipe to transform a state and produce a value. The combination of monadic actions therefore does not need to happen in some chronological order.

In simple terms, what we want to achieve is to combine two monadic actions, where each of the produced values may be accessed by the respective other monadic action. Ideally, we'd like to be able to write something like the following:

```
1 ma = do
2   a1 <- ma1 a2
3   a2 <- ma2 a1
4   return a2
```

While this is clearly invalid code—due to the early accessal of `a1`—we shall see that we actually can validly specify this in a quite similar way.

As a first step, we translate the above's intentions to a `let` expression. For reasons of simplicity, we assume that `State s a = s -> (a, s)`.

```
1 ma s0 =
2   let (a1, s1) = ma1 a2 s0
3       (a2, s2) = ma2 a1 s1
4   in  (a2, s2)
```

At first sight, this definition might look equally wrong, given the cyclic dependency between the two equations in the `let` expression. Fortunately, this is perfectly valid due to Haskell's laziness and implicit calculation of a fixed point. We may specify an equivalent definition without cyclic dependencies, but by use of the `fix` function:

```
1 ma s0 = fix $ \ (a2, _) ->
2   let (a1,  s1) = ma1 a2 s0
3       (a2', s2) = ma2 a1 s1
4   in  (a2', s2)
```

Clearly, we want to ensure that the fixed point exists and can be calculated in finite time. This is the case for our concrete problem: The identifiers given to a TEI entry do not depend on that

TEI entry's references to other TEI entries. We could therefore unwrap the `IdState` and specify the function `convEntries` using a `let` expression.

There is a still better way though. What `fix` is to regular functions, `mfix` is to monadic actions. Unlike `fix`, `mfix` is not directly defined. Instead, there exists a class `MonadFix`:

```
class Monad m => MonadFix m where
  mfix :: (a -> m a) -> m a
```

Given a `MonadFix` instance, we may now redefine `ma` from above as follows:

```
ma = mfix $ \ a2 -> do
  a1  <- ma1 a2
  a2' <- ma2 a1
  return a2'
```

This looks already much better. Furthermore, we may use the `RecursiveDo` GHC[21] extension [19, 8] that allows us to specify this quite naturally:

```
ma = do
  rec
    a1 <- ma1 a2
    a2 <- ma2 a1
  return a2
```

The `rec` keyword groups a list of lines in the `do` notation that should essentially be wrapped in `mfix`, as demonstrated above. Note that we could also use the `mdo` keyword in place of `do`, which would allow us to omit the `rec` keyword. I prefer the explicitness of `rec` though.

Fortunately, the state monad that we use provides an instance of `MonadFix`. Hence, using the newly gained knowledge, we may get to the following implementation of `convEntries`:

```
convEntries :: [Ding.Entry]
            -> [TRef.Ident]
            -> IdState ([TRef.Ident], [TEI.Entry])

convEntries []       _    = return ([], [])
convEntries (de:des) refs = do
  rec
    (refs1, tes1) <- convEntry   de  (refs ++ refs2)
    (refs2, tes2) <- convEntries des (refs ++ refs1)
  return (refs1 ++ refs2, tes1 ++ tes2)
```

To understand this definition, consider again the following example Ding line:

```
Ding | Dings | Krempel :: thing | thingy | things
```

The `convEntries` function is to sequentially process all suffixes of a Ding line, from longest to shortest, where a Ding line is considered a list of entries. Consider in particular the case where `convEntries` is applied to the suffix starting at `Ding :: thing`. The idea is now that `convEntries`

- is provided with references of type `TRef.Ident` that stem from previous entries in the same line (e.g., "`Ding.4`") and
- returns the TEI entries together with the corresponding references (e.g., "`Dings.2`" and "`Krempel.1`") as constructed from the given list of entries.

[21] Glasgow Haskell Compiler: The most widely used Haskell compiler.

Additionally, `convEntry` is provided with all references from surrounding Ding entries. The job of `convEntry` is subsequently to convert a single Ding entry to potentially several TEI entries, each of which must be annotated with references to the *related* entries, as given to `convEntry`. Also, `convEntry` needs to provide the generated entries with references to *synonymous* TEI entries—such that stem from the same Ding entry. This may be done in a quite similar fashion to the distribution of *related*-references and—as noted earlier—shall therefore not be discussed here.

Annotations

As noted in section 2.5, the annotations' structure is the same for both the Ding and TEI AST, it just needs to be put into other places (see subsection 2.4.1 and subsection 2.4.2). The concrete implementation is hence rather simple and shall not be discussed here.

3.5 TEI XML generation

In this step, we need to translate the TEI AST to textual XML. The actual generation of the XML shall not be discussed here, it is mostly unspectacular.

Tool choice

It is not convenient to generate XML by hand, in particular proper indentation is non-trivial. Therefore, we shall use an XML library, in particular the `xml` library [17]. This is a quite simple library—simple to use, but lacking some features. However, since we only need to print a rather simple XML tree, it shall suffice.

3.6 Enrichment

As specified in the introduction, we not only wish to represent in TEI what is explicitly represented in the Ding dictionary, but also include information that is only implicitly present.

As equally noted in the introduction, the Ding dictionary is directionless, while TEI dictionaries have a direction. Although the actual fixing of a direction happens in the translation step, some of the concrete enrichments also assume a direction. Therefore, the enrichment is split into two steps, `enrichUndirected` and `enrichDirected`, where the latter is to happen after the former. This would in particular be useful if it were desired to produce two TEI dictionaries—one for each direction—in a single run of the program. This is not currently implemented though.

The `enrichUndirected` step currently is only concerned with grammar annotations, while the `enrichDirected` step identifies examples and transforms the AST correspondingly.

3.6.1 Grammar

The grammar enrichment is further subdivided in two steps, which shall be explicated in the following.

Inferral

During *inferral*, grammar information that is not explicitly present in the Ding dictionary, but can be reliably inferred, is added, on the level of single units. For example, syntax analysis has

shown that words annotated with a gender, but no grammatical number, generally are in the singular form.

Additionally, the lists of grammar annotations are deprived of duplicated information, some of which may stem from the inferral.

Transferral

Many annotations are only present in a single unit in a Ding entry, while they often also apply to other units of the same entry. In particular, most of the annotations—barring inflected forms—are much more frequent on the German side of the Ding.

I therefore have identified a subset of possible grammar information that can—potentially only partially—validly be *transferred* between units in an entry. While it seems likely that that subset differs for transferral between units on the same side and units on different sides, this has shown not to be the case for the known annotations, in the context of the German and English languages.

3.6.2 Examples

Many of the entries in the Ding dictionary exemplify other entries. Such *examples* can unfortunately not be identified on the level of the syntax, as it is described in subsection 2.4.1.

Representation in TEI

Before going into the details of what an example constitutes, we shall have a look at how we can represent examples in TEI, which may restrict us in what to consider an example.

FreeDict TEI allows to add examples within a `sense` element. Such examples may have several forms. Until lately, I was assuming that the only permitted form was that expressed by the following *XML element schemata*[22]:

```
<cit type="example">
  <quote xml:lang="SRCLANG">SRC_EXAMPLE</quote>
  {<cit type="trans" />}
</cit>
```

```
<cit type="trans">
  <quote xml:lang="TGTLANG">TGT_EXAMPLE</quote>
</cit>
```

Note that the schema for `<cit type="trans" />` here should be distinguished from that for translations as defined in definition 2.4.9.

Essentially, we may specify a single example in the source language—that exemplifies the surrounding entry's headword—and add any number of translations in the target language. Rather lately, I was made aware that it is also possible to specify examples as a list of any number of examples in both languages. We will not pursue this option here.

Case studies

Consider a few lines from the Ding dictionary that may be considered as containing an example. Note that they are redacted.

First, consider the following line.

[22] See definition 2.4.1.

```
Aufenthalt | Dieser Aufenthalt war nicht eingeplant. :: stop | This stop wasn't
    ↪ scheduled.
```

Listing 3.1: line containing a definite example entry

It consists of two syntactically minimal entries, where the second one clearly is to be read as an example to the first, rather than its own independent entry. The exemplifying becomes apparent in particular by the fact that for each language, the only unit of that language in the first entry appears literally in the second entry. Furthermore, the second entry relates phrases, which are usually not seen in a dictionary as independent headwords.

Compare this to the following line.

```
redaktionell | Redaktionssitzung :: editorial | editorial meeting
```

Listing 3.2: line containing two regular entries

At first, one might consider the second entry equally an example to the first; however, the second entry is also useful on its own. One might very well search for "`Redaktionssitzung`"—which one couldn't even find by any of its constituents, namely "`Redaktion`" and "`Sitzung`", if it were only present as an example to "`redaktionell`".

Next, have a look at a completely different line.

```
ein erfreulicher Anblick | Schön, dich zu sehen! :: a sight for sore eyes | Good
    ↪ to see you!
```

Listing 3.3: line containing two composed expressions

Both entries contain composed expressions and these are of similar complexity. It would therefore be unnatural to consider one of them an example of the other.

Finally, consider a less redacted version of the line 3.1.

```
Aufenthalt; Halt | Dieser Aufenthalt war nicht eingeplant. :: stop; stopover;
    ↪ layover | This stop wasn't scheduled.
```

Listing 3.4: line with nontrivial groups

This line contains groups with more than one unit, where not all of them occur as infixes in the alleged example entry. Remember that entries with several units in the source language are split into several TEI entries; therefore, the question arises which of these TEI entries the potential example should be annotated to.

Identification

With the above lines in mind, we shall now decide on what to consider an example, that is, by which characteristics it is to be identified.

There are two basic means of identification:

1. Define an example by its properties (e.g., number of words, presence of interpunctuation).
2. Define examples with respect to the units they exemplify (e.g., require that an example is an infix of another unit).

Both means are reasonable, so they shall be combined.

Definition 3.6.1 (expression)**.** *A unit is called an* expression *if it either*

a) *contains interpunctuation, or*

b) *is composed of three or more space-separated words, where annotations do not count.*

Definition 3.6.2 (example unit)**.** *An expression is said to* exemplify *another unit if the latter is not an expression and infix of the former. In this case, the expression is called an* example unit*.*

We impose the additional restriction that units may only exemplify units within groups left of them. This choice can be justified by syntax analysis, but also by the obsolete Ding syntax specification [34], according to which what I call a line consists of an initial main entry together with some auxiliary entries that in particular may include examples. Having relaxed that definition generally, it is natural to allow examples to not only refer to the very first entry but also to later, albeit preceding ones.

Next to the exemplifying unit, we also want to give its translations, which leads us to the following definition.

Definition 3.6.3 (example)**.** *An* example *is composed of an example unit and the corresponding translation group, that is, the set of units it translates to.*

Goal

For each line, examples should be identified as described above, removed from that line and annotated to the units they exemplify. Note that a single example may exemplify more than one unit.

Examples are identified on the unit level. That is, an entry's source group may contain both examples and non-examples. In the latter case, the whole entry should be removed from the line; in the former case, we only need to remove units from the source group. This is, because for each Ding unit in the source group, a separate TEI entry is to be generated.

Implementation

Since we require examples to be right of the units they exemplify, we process a line from left to right. A unit is clearly a non-example—and therefore to be retained—iff it is not an example to any of the preceding (non-example) units.

For each entry in a line, and each unit in its source group, we need to test whether it exemplifies any of the preceding non-example units. If it does, we need to amend the exemplified unit with the newfound example and remove the example unit from its group. If a group ends up empty this way, the enclosing entry is to be removed.[23]

Note how the alteration of the previously processed units is non-trivial, they have already been processed after all. The solution to this shall be a stack on which any entry is pushed after initial processing. This stack is wrapped in a state monad[24] and can therefore be changed during the processing of later units in the same line. We implement this as follows:

```
1 type EntryStackState = State [Entry]
2
3 push :: s -> State [s] ()
4 push x = modify (x:)
```

Assuming a `handleEntry` function, we may now define a function to process a list of entries:

```
1 handleEntries :: [Entry] -> EntryStackState ()
2 handleEntries = mapM_ handleEntry
```

[23] Groups that were empty since the beginning can safely be removed, the surrounding entry has no non-empty representation in TEI anyways; only entries with empty target groups may be represented in TEI.

[24] See section 3.4 for a brief introduction to state monads.

Note that the list of resulting entries is only present as the state inside the monad and needs to be extracted from there in the calling function.

When testing a unit that is an expression for being an example, we provide it with a list of the preceding entries—the stack's content. To allow for both annotating the examined potential example unit to units it exemplifies and to log whether any such annotation occurred—that is, whether we found an example unit—we use a `Writer` monad[25]. This `Writer` monad should allow for writing a boolean value, where several boolean values should be combined using $\wedge$. We therefore effectively want to consider the `Bool` type a monoid with `False` as neutral element and `&&` as operator, which can be implemented as follows:

```
type ChangeWriter = Writer Changed

newtype Changed = Changed Bool

instance Semigroup Changed where
  (Changed b1) <> (Changed b2) = Changed $ b1 || b2

instance Monoid Changed where
  mempty = Changed False
```

This allows us now to define a function that updates the stack's entries with respect to a single potential example unit, and to identify whether this actually is an example:

```
updateEntries :: Unit -> [Unit] -> [Entry]
              -> ChangeWriter [Entry]
updateEntries pex pexTrans = mapM $ updateEntry pex pexTrans
```

It is provided with the to be examined unit, the list of its translations and the entries to be considered as being exemplified.

The `updateEntry` function subsequently needs to inspect each unit in the source group of the entry it is provided with—for being exemplified by the equally supplied potential example unit.

[25] A writer monad is like a generic state monad, except it does only allow for writing, in the form of amending a state with values of a monoid using that monoid's operator.

Chapter 4

Conclusion

4.1 Reflection on the goals

In this section, the goals, as stated in section 1.4 of the introduction shall be reflected upon.

4.1.1 Validity of the output

To test the resulting TEI XML files for validity and interoperability with the FreeDict tools, we need to insert them into the directory structure of the the FreeDict dictionaries' git repository [13] (locally), thereby replacing the old files.

The results are:

- `make validation` and `make qa` succeed. In particular, the syntax is valid according to the provided XML RelaxNG schema.
- The `teiaddphonetics` script—responsible for adding phonetics information to the resulting TEI—fails. The reason has been identified as a bug in that script, not allowing certain special constellations of special characters. While the adding of phonetic information is required by the Makefile to be executed before translating the TEI result to any output format, this step can be simulated by just putting a symbolic link in the right place[1].
- Building both output formats (*DICT* [44] and *Slob* [40]) succeeds.

Additionally, the FreeDict project's member Sebastian Humenda kindly commented on a preliminary result, identifying a few flaws, which since have been fixed.

4.1.2 Quality of the translation

With a few exceptions, all information that is marked up explicitly in the Ding dictionary (e.g., by "`::`", "`|`", "`{`") is represented explicitly in the TEI output. The exceptions are annotations in angle brackets, which are used quite inconsistently (see section 2.5) and infix annotations, which are mostly dropped (see subsection 3.3.2), since they cannot be generally represented in FreeDict TEI.

Additionally, the enrichment notably increases the grammar annotations and identifies some examples.

[1] See `README`.

In general, the result can in my opinion be seen as good and usable, albeit there remain many possibilities for improvement, as depicted in section 4.3. In particular, the resulting dictionaries seems superior to the current German-English and English-German FreeDict dictionaries; not only due to the larger data set, but also due to richer annotation (e.g., cross-references).

4.1.3 Quality of the Code

The module structure is rather fine grained and the code is generally well equipped with documentation. The general documentation, located in `doc/`, is quite limited though. Some files in the `todo/` directory may also serve as documentation on the present code; however, the quality of documentation therein is often not overly high. Additionally, this thesis is also intended as documentation.

Some non-trivial Haskell constructs, such as `MonadFix`, are used. These generally make the code shorter, but in parralel also likely harder to grasp for people not experienced in Haskell. However, should I ever hand maintainership of the resulting dictionary over to somebody else (of the FreeDict project), it is likely that they only want to change certain bits, to accommodate for changes in the upstream Ding dictionary. In most cases, this will require amending the preprocessor, which is written in sed and bears very little structure—editing it should be trivial, given a minimal knowledge of sed. Further, the syntax of either Ding or FreeDict TEI may change or be interpreted differently; in most cases this will only affect the part of the AST that is shared between the Ding and TEI AST (in particular, annotations), or the construction of TEI XML. In either case, the to be edited code is of quite simple structure. Also, I have provided a brief introduction to Haskell[2] to ease the understanding of the code. Furthermore, the syntax of Ding is given in (annotated) BNF; at least its understanding does not require any Haskell or Happy knowledge; modification may cause some hard to grasp error messages though.

Efficiency

On my not-too-recent machine, the main program takes approximately 1 minute and 40 seconds to translate the Ding dictionary to TEI, when compiled with `-O2`. The compilation itself takes about a minute, the preprocessing about 10 seconds. The run-times of `alex` and `happy` are negligible (less than a second). Overall, the run-times are definitely within reasonable bound.

Concerning the memory usage, the main program consumes about 4.5 GiB of memory when working on the full Ding dictionary. This is quite a lot. One might expect that due to Haskell's laziness, the memory usage should be easy to keep in bounds; unfortunately there is one small problem: The TEI header needs to contain the number of TEI entries—which is unknown until the whole list of TEI entries is constructed. As the TEI header is located at the top of the TEI output, the program cannot start writing any of the body—and thereby freeing memory—until most of it is computed.

While the memory usage could likely notably be reduced by replacing the `String` type with more a space-efficient type such as `ByteString` [33], this seems less important when looking at the resource consumption by the FreeDict tools when applied to the TEI output of the main program:

All nontrivial tools that were tested consumed more than the above 4.5 GiB of memory, with a peak of 7.7 GiB for `teiaddphonetics`. Concerning run-time, the *DICT* exporter ran for about 24 hours, while the other tools took less time, some of them even a moderate amount. Note that my program cannot influence the resource consumption of these tools, except by producing

[2] `doc/haskell_intro`

smaller output, which is not possible without stripping information. Instead, the *DICT* exporter in particular should likely be investigated in order to make it more efficient.

4.1.4 Further optional goals

Spanish-English dictionary

Syntax analysis has shown that the Spanish-German dictionary's format, which is advertised as that of the Ding dictionary, is actually different. It is nonetheless similar and actually much simpler. Note that the Ding program can easily handle it, simply because the Ding program (i.e., grep) can handle anything. Notable incompatibilities are:

- The number of group separators ("`|`") sometimes differs between the two sides of a line.
- There are annotations of the form "`/SOME_WORD`" at the end of lines, which presumably are to apply to the whole line.

Due to the simplicity of the structure, I was able to write a simple sed script to adapt the syntax[3], albeit with some loss of information. This script is not to be considered a solution, but rather a proof of concept. Instead, the main program should be adapted to also allow for the syntax of this Spanish-German dictionary, likely only when supplied with a special flag.

Other versions of the Ding dictionary

The written program exclusively works with the latest stable version of the Ding dictionary, 1.8.1. Adapting it to the unstable *devel* version likely mostly requires amending a copy of the preprocessing scripts.

4.2 Reflection on choices made

Lexing and Parsing tools

In the Ding dictionary, some syntax elements may only occur on either of the German or English side, it would therefore be useful to be able to distinguish these sides in the parser. However, most syntax is equal on both sides; thus, it is not viable to set up a complete rule set for both languages. Instead, we'd like to have rule families, parametrised by language. This is unfortunately not possible in Happy, while it would have been when using Megaparsec.

As an alternative measure, one might consider to use the Happy-supported attribute grammars [28], however—while they do allow to pass arguments (*inherited* attributes) from top to bottom—they do not allow to have these arguments alter the actual grammar rules, but only the annotated functions used to build the AST from the grammar's rules. This may in particular be used to throw errors in case of unexpected syntax depending on the respective side's language; however, this is often not sufficient, as certain syntax might occur on both sides, albeit with different semantics.

Note that the main reasons that led to the decision for Alex and Happy over Megaparsec were actually reasons to choose Alex over the Lexing capabilities of Megaparsec. It might therefore have been a better choice to combine Alex and Megaparsec, even though this would deprive Megaparsec of one of its core advantages.

[3] `src/preprocess/es-de/syntax.sed`

XML library

The `xml` library has shown to be very limited. In particular, the XML header needs to be added manually, and—when pretty-printing—XML content made up of a list of both text and element nodes is unconditionally treated equally to a list of element nodes, that is, causing superfluous whitespace to be added between them. This issue was circumvented by providing a function that combines such a sequence of text and element nodes into the corresponding XML string literal, and subsequently treating that literal as single XML element.

Other XML libraries likely would not have required to resort to such tricks.

4.3 Future work

As noted earlier, the Ding dictionary is in large parts meant to be parsed by the human reader and therefore the structure is in many cases quite hard to parse by a machine. Hence, the possibilities of improvement are mostly endless—many of them requiring (notably) more of a linguistics background than of a computer science background. This is not to say though that all potential improvements are hard to implement.

Next to the remaining optional goals reflected upon in section 4.1, the following issues particularly deserve further work. Note that the below list is far from complete. Refer to the files in the `todo/` directory for more.

Special collocations

Many Ding units contain keywords such as "`sth.`", "`to sb.`", "`from sb./sth.`", "`etw.`", "`für etw.`", "`für etw./jdn.`"; as in the following units:

```
to award the contract to sb.
to accept sb.'s bid
to demand an explanation of sth. from sb.
to entrust sth. to sb. / sb. with sth.
etw. ablegen
für jdn. ein Vorbild sein
eine Gefahr/Bedrohung für jdn./etw. darstellen
```

Ideally, we would like these keywords—*collocations*—to be identified and treated similarly to annotations, in particular they should not be retained as part of the unit's text. This is in particular to allow finding for example "`etw. ablegen`" when only searching for "`ablegen`". Note that the "`to`" prefix is already recognized during parsing as a verb indicator and not considered part of the unit's text.

As seen in the above examples, those collocations appear usually, but not always, as prefixes or suffixes of units. Further, they may be combined, potentially with slashes, and also occur between other annotations, such as `{f}`. We have seen in subsection 3.3.1 and subsection 3.3.2 that the handling of slashes is generally difficult.

Another difficulty lies in the fact that such collocations (e.g., "`for sb.`") may also occur as individual units (the collocation would then be "`sb.`"). Because of this, and also the quite unlimited composability of these collocations' constituents (e.g. "`für/gegen jdn./etw.`"), we cannot simply have the lexer identify them and treat them similarly to annotations in the parser.

Also, it may be desirable to only allow certain valid compositions[4]; thereby denying for example "`für jdm.`"—which should instead be "`für jdn.`".

[4] See `todo/parsing.collocation-literals` for a partial syntax of valid compositions.

In general, properly identifying these collocations seems quite difficult. However, they also represent the single biggest flaw in the TEI result in my opinion and therefore it is likely worth the effort tackling this issue.

Improved example handling

The example handling as explicated in subsection 3.6.2 unfortunately catches only a minority of all units that are *expressions*.

At least some of these are not recognized because the heuristic only recognizes them as exemplifying a unit in the respective target language and not also in the source language. It should therefore be considered to change the representation in TEI to also allow these examples to be recognized.

Also, it may be worth amending the heuristic; for example, to also recognize umlaut modifications (e.g., "`Apfel`" ~ "`viele Äpfel essen`").

Slash and "<>" handling

While for slashes in general it is difficult to determine their scopes, it has been noted in subsection 3.3.1 that *strong* slashes usually have only single words in scope. One might decide to ignore the few exceptions or identify them by hand and fix them.

The "`<>`" symbol is similar to a strong slash, it specifies that the surrounding two words may be swapped [34]. In particular, the scopes are generally clear, although one needs to take care of slashes once again, as exemplified by the following unit:

```
|| to big up <> sb./sth.
```

Acknowledgements

This work is built upon the work of others.

It literally could not exist without a subject to work on; therefore, I wish to thank Frank Richter and any other contributors to the Ding dictionary for their valuable work and for sharing it as Free Software. Equally, I wish to thank the various contributors to the FreeDict project, for their continued successful endeavour to provide the public with "truly free bilingual dictionaries" [16], and for providing the infrastructure and support allowing me to translate the Ding dictionary to FreeDict TEI.

I particularly wish to express my gratitude towards Sebastian Humenda of the FreeDict project, who spent a great amount of time on dealing with the many questions of mine, and additionally Piotr Bański, who also answered some of them.

Finally, I wish to thank my advisor, Frank Huch, for his valuable suggestions and answers on my similarly numerous questions, both on programming and the general writing of a thesis.

Bibliography

[1] Piotr Bański and Sebastian Humenda. *[freedict] Re: Case inflections, verb and adjective forms*. Sept. 8, 2020. URL: https://www.freelists.org/post/freedict/Case-inflections-verb-and-adjective-forms,9.

[2] Karl Bartel. *Wikdict: About*. URL: https://www.wikdict.com/page/about (visited on 10/10/2020).

[3] Christophe Blaess. *Implementation of a Turing Machine as Sed Script*. 2001. URL: https://catonmat.net/ftp/sed/turing.txt (visited on 10/09/2020).

[4] Nigel P. Chapman. *LR parsing. theory and practice*. Cambridge: Cambridge University Press, 1987, pp. 86–87. ISBN: 052130413X.

[5] TEI Consortium. *TEI: About*. URL: https://tei-c.org/about/ (visited on 10/10/2020).

[6] DARIAH Working Group on Lexical Resources. *TEI Lex-0. A baseline encoding for lexicographic data*. URL: https://dariah-eric.github.io/lexicalresources/pages/TEILex0/TEILex0.html (visited on 10/10/2020).

[7] Chris Dornan, Isaac Jones, and Simon Marlow. *Alex User Guide*. Version 3.0. Aug. 11, 2003. URL: https://www.haskell.org/alex/doc/alex.pdf (visited on 10/11/2020).

[8] Levent Erk and John Launchbury. "A Recursive do for Haskell". In: *Proceedings of the 2002 ACM SIGPLAN Haskell Workshop* (Sept. 2002). DOI: 10.1145/581690.581693.

[9] Jay Fenlason et al. *sed(1). stream editor for filtering and transforming text*. Version 4.7. Dec. 2018. URL: https://manpages.debian.org/buster/sed/sed.1.en.html (visited on 10/09/2020).

[10] Olle Fredriksson. *Earley: Parsing all context-free grammars using Earley's algorithm*. Version 0.13.0.1. URL: https://hackage.haskell.org/package/Earley (visited on 10/11/2020).

[11] Free Software Foundation. *grep(1). print lines that match patterns*. Version 3.3. May 11, 2018. URL: https://manpages.debian.org/buster/grep/grep.1.en.html (visited on 10/09/2020).

[12] FreeDict project. *Discussion on (FreeDict) TEI*. URL: https://github.com/freedict/fd-dictionaries/wiki/discussion-TEI (visited on 10/09/2020).

[13] FreeDict project. *FreeDict dictionaries*. URL: https://github.com/freedict/fd-dictionaries (visited on 10/09/2020).

[14] FreeDict project. *FreeDict tools*. URL: https://github.com/freedict/tools (visited on 10/09/2020).

[15] FreeDict project. *FreeDict Wiki*. URL: https://github.com/freedict/fd-dictionaries/wiki (visited on 10/09/2020).

[16] FreeDict project. *Home — FreeDict*. 2020. URL: https://freedict.org/ (visited on 10/10/2020).

[17] Galois Inc. *xml: A simple XML library*. Version 1.3.14. URL: https://hackage.haskell.org/package/xml (visited on 10/09/2020).

[18] Zeno Gantner and Matthias Buchmeier. *Spanish-German vocabulary list for ding*. Version 0.0i Mon Mar 5 18:36:47 2012. URL: https://savannah.nongnu.org/projects/ding-es-de (visited on 10/09/2020).

[19] GHC Team. *Glasgow Haskell Compiler User's Guide. GHC Language features*. Version 8.10.1. 2015. URL: https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/glasgow_exts.html (visited on 10/11/2020).

[20] HaskellWiki. *Hierarchical module names — HaskellWiki*. 2012. URL: https://wiki.haskell.org/index.php?title=Hierarchical_module_names&oldid=45429 (visited on 10/13/2020).

[21] HaskellWiki. *Structure of a Haskell project — HaskellWiki*. 2012. URL: https://wiki.haskell.org/index.php?title=Structure_of_a_Haskell_project%5C&oldid=54914 (visited on 10/09/2020).

[22] Paul Hemetsberger. *dict.cc | German-English dictionary*. URL: https://www1.dict.cc/translation_file_request.php (visited on 10/08/2020).

[23] Paul Hemetsberger. *dict.cc: About/Contact*. URL: https://www.dict.cc/?s=about: (visited on 10/08/2020).

[24] Sebastian Humenda. *[freedict] Poll: replace deu-eng / eng-deu*. May 8, 2020. URL: https://www.freelists.org/post/freedict/Poll-replace-deueng-engdeu.

[25] Einhard Leichtfuß. *[freedict] Re: Poll: replace deu-eng / eng-deu*. Aug. 29, 2020. URL: https://www.freelists.org/post/freedict/Poll-replace-deueng-engdeu,19.

[26] LEO Dictionary Team. *ganzer | ganze - Translation in LEO's English ⇔ German Dictionary*. URL: https://dict.leo.org/german-english/ganzer+%7C+ganze (visited on 10/09/2020).

[27] LEO Dictionary Team. *LEO: Dict-Statistik*. Archived at https://web.archive.org/web/20130122233400/http://dict.leo.org/pages.ende/stat_de.html. Jan. 22, 2013. URL: http://dict.leo.org/pages.ende/stat_de.html (visited on 01/22/2013).

[28] Simon Marlow and Andy Gill. *Happy User Guide*. Version 1.18. 2009. URL: https://www.haskell.org/happy/doc/html/index.html (visited on 10/11/2020).

[29] Conor McBride. "Clowns to the Left of Me, Jokers to the Right (Pearl): Dissecting Data Structures". In: *SIGPLAN Not.* 43.1 (Jan. 2008), pp. 287–295. ISSN: 0362-1340. DOI: 10.1145/1328897.1328474.

[30] Megaparsec contributors. *megaparsec: Monadic parser combinators*. Version 9.0.0. URL: https://hackage.haskell.org/package/megaparsec (visited on 10/11/2020).

[31] George A. Miller. "WordNet: A Lexical Database for English". In: *Communications of the ACM* 38.11 (Nov. 1995), pp. 39–41. DOI: 10.1145/219717.219748.

[32] Andrew L. Moore and Antonio Diaz Diaz. *ed(1). line-oriented text editor*. Version 1.15. Jan. 2019. URL: https://manpages.debian.org/buster/ed/ed.1.en.html (visited on 10/09/2020).

[33] Bryan O'Sullivan et al. *xml: A simple XML library*. Version 0.11.0.0. URL: https://hackage.haskell.org/package/bytestring (visited on 10/10/2020).

[34] Frank Richter. *Aufbau der Übersetzungs-Datei*. July 2005. URL: https://dict.tu-chemnitz.de/doc/syntax.html (visited on 10/09/2020).

[35] Frank Richter. *Ding: A Dictionary Lookup program*. Version 1.8.1. Sept. 2016. URL: https://www-user.tu-chemnitz.de/~fri/ding/ (visited on 10/10/2020).

[36] Gilles Sérasset. "DBnary: Wiktionary as a Lemon-Based Multilingual Lexical Resource in RDF". In: *Semantic Web* 6 (May 2015), pp. 355–361. DOI: 10.3233/SW-140147.

[37] Tagesschau.de. *Geschlechtergerechte Sprache. Streit über Gesetzestext in weiblicher Form*. ARD. Oct. 12, 2020. URL: https://www.tagesschau.de/inland/streit-gesetzestext-weibliche-form-101.html (visited on 10/12/2020).

[38] TEI Consortium, eds. *TEI P5: Guidelines for Electronic Text Encoding and Interchange*. Version 4.1.0. Aug. 19, 2020. URL: http://www.tei-c.org/Guidelines/P5/ (visited on 10/09/2020).

[39] Anne Theenhaus and Matthias Schaefer. "The effects of clear-cutting and liming on the soil macrofauna of a beech forest". In: *Forest Ecology and Management* 77.1 - 3 (Sept. 1995), pp. 35–51. DOI: 10.1016/0378-1127(95)03580-4.

[40] Igor Tkach. *Slob*. URL: https://github.com/itkach/slob/blob/master/README.org (visited on 10/13/2013).

[41] Wikipedia. *Geschlechtergerechte Sprache — Wikipedia, Die freie Enzyklopädie*. 2020. URL: https://de.wikipedia.org/w/index.php?title=Geschlechtergerechte_Sprache&oldid=204426566 (visited on 10/10/2020).

[42] Wikipedia contributors. *Backus–Naur form — Wikipedia, The Free Encyclopedia*. 2020. URL: https://en.wikipedia.org/w/index.php?title=Backus%E2%80%93Naur_form&oldid=981305357 (visited on 10/09/2020).

[43] Wikipedia contributors. *Copyleft — Wikipedia, The Free Encyclopedia*. 2020. URL: https://en.wikipedia.org/w/index.php?title=Copyleft&oldid=980794227 (visited on 10/13/2020).

[44] Wikipedia contributors. *DICT — Wikipedia, The Free Encyclopedia*. 2020. URL: https://en.wikipedia.org/w/index.php?title=DICT&oldid=956132992 (visited on 10/13/2020).

[45] Wikipedia contributors. *Dict.cc — Wikipedia, The Free Encyclopedia*. 2019. URL: https://en.wikipedia.org/w/index.php?title=Dict.cc%5C&oldid=928265367 (visited on 10/08/2020).

[46] Wikipedia contributors. *Earley parser — Wikipedia, The Free Encyclopedia*. 2020. URL: https://en.wikipedia.org/w/index.php?title=Earley_parser&oldid=970168970 (visited on 10/11/2020).

[47] Wikipedia contributors. *English language — Wikipedia, The Free Encyclopedia*. 2020. URL: https://en.wikipedia.org/w/index.php?title=English_language&oldid=982362604 (visited on 10/10/2020).

[48] Wikipedia contributors. *Grep — Wikipedia, The Free Encyclopedia*. 2020. URL: https://en.wikipedia.org/w/index.php?title=Grep&oldid=982878639 (visited on 10/12/2020).

[49] Wikipedia contributors. *Homograph — Wikipedia, The Free Encyclopedia*. 2020. URL: https://en.wikipedia.org/w/index.php?title=Homograph&oldid=966897894 (visited on 10/10/2020).

[50] Wikipedia contributors. *Internet Relay Chat — Wikipedia, The Free Encyclopedia*. 2020. URL: https://en.wikipedia.org/w/index.php?title=Internet_Relay_Chat&oldid=982274231 (visited on 10/12/2020).

[51] Wikipedia contributors. *LALR parser — Wikipedia, The Free Encyclopedia*. 2020. URL: https://en.wikipedia.org/w/index.php?title=LALR_parser&oldid=941890158 (visited on 10/09/2020).

[52] Wikipedia contributors. *LL parser — Wikipedia, The Free Encyclopedia*. 2020. URL: https://en.wikipedia.org/w/index.php?title=LL_parser&oldid=982396718 (visited on 10/09/2020).

[53] Wikipedia contributors. *Scannerless Parsing — Wikipedia, The Free Encyclopedia*. 2020. URL: https://en.wikipedia.org/w/index.php?title=Scannerless_parsing&oldid=979074748 (visited on 10/11/2020).

[54] Wikipedia contributors. *Thesaurus — Wikipedia, The Free Encyclopedia*. 2020. URL: https://en.wikipedia.org/w/index.php?title=Thesaurus&oldid=979685055 (visited on 10/13/2020).

[55] Wiktionary. *happily — Wiktionary, The Free Dictionary*. 2020. URL: https://en.wiktionary.org/w/index.php?title=happily&oldid=59506058 (visited on 10/11/2020).

[56] Wiktionary. *happy — Wiktionary, The Free Dictionary*. 2020. URL: https://en.wiktionary.org/w/index.php?title=happy&oldid=60731129 (visited on 10/11/2020).

[57] Wiktionary. *Wiktionary, the free dictionary*. 2020. URL: https://en.wiktionary.org/wiki/Wiktionary:Main_Page (visited on 10/09/2020).