# 9 Target Schemas and Input/Output Models

We have already shortly described the input/output requirements our system should fulfill (Sec. 7.2.1). In this chapter we will cover them in more detail, discussing the problems that arise in this context and how they can be solved.

One specific kind of input are the *target schemas* that define which kinds of information should be extracted from the input texts. They will be treated in the first section; in the following sections we will discuss which kind of text formats should be allowed as input and which formats and annotation styles should be supported for answer keys (expected attribute values). Finally we will turn to the question of output: how should the results of the extraction process be serialized to allow both evaluation and further processing?

## 9.1 Target Schemas

As stated before, information extraction differs from related areas in being schema-based: it requires a predefined *target schema* that specifies which kinds of information should be extracted and how they should be stored. The definitions that a target schema needs to provide depend on the task or tasks (cf. Sec. 3.1) that should be handled.

In the case of our system, which is only meant to handle the *extraction of explicit information* (*fragment extraction*—cf. Sec. 7.1.1), target schemas become very simple: all that is required is a list of attributes that are to be extracted. There is no need to provide any explicit semantics or characteristics of the attributes to extract, since the system is trained from a set of annotated training texts and is expected to *learn* what it needs to know. Attributes do not need to be explicitly typed, since no *value normalization* is performed.

*Relationship resolution* would be necessary to combine extracted attribute values into non-trivial tuples and to resolve dependencies between relations. Since this is another task that is beyond the scope of this work, only two simple cases of relations are currently allowed: *text-as-tuple* and *single-attribute relations.*

**Text-as-tuple** means that there is only a single relation (with any number of attributes) in the target schema and that each text corresponds to at most one tuple in this relation, either “naturally” or due to a suitable segmentation step during preprocessing (a document without any information to extract correspond to zero tuples). If there are several candidate extractions for an attribute found in a text, all except one will have to be discarded—our approach will handle this by keeping the most probable candidate, as is usual for statistical approaches. This scenario applies to most standard IE tasks, among them the *Seminar Announcements* and the *Corporate Acquisitions* corpora that will be used for evaluation.

**Single-attribute relations** means that the target schema comprises several relations, but each of them only has a single attribute, i.e. all attributes are independent of each other. In this case, there is no need for relationship resolution. This scenario applies, for example, for the extraction of named entities (NE) and related tasks, e.g., extraction of biomedical entities—it has been used to train this system as a NE recognizer for the weakly hierarchical approach (cf. Chapters 14 and 20).

These two scenarios correspond to two different evaluation modes supported by our system (*"one answer per attribute"* vs. *"one answer per occurrence"*)—we will explain these modes in Section 15.2 at the beginning of the evaluation part. In other, more complex, cases, relationship resolution will have to be performed as a postprocessing step after our system has finished its work.

There are no theoretical constraints on the number or the characteristics of attributes that can be defined, but, as stated in the previous chapter, the quality of extraction results generally depend on the suitability of attributes for automatic extraction—we can expect problems with attributes whose values are very heterogeneous, very vaguely defined or very long. The number of training examples is another factor that will influence results, since trainable algorithms require a sufficiently large and representative set of examples to learn reliable characteristics. Hence, attributes that only occur infrequently should probably not be included in target schemas, unless the training corpus is very large.

## 9.2 Formats for Input Texts

### 9.2.1 Plain Text and Structured Documents Formats

Traditionally, information extraction corpora have been made available in plain text format, without any markup. However, since the advent of the Web and its "lingua franca" *HTML*, extraction from structured documents formats such as HTML has become more important. While some families of approaches such as *wrapper induction* (cf. Sec. 5.3) rely heavily or even primarily on markup information (HTML tags), other modern approaches (e.g. *ELIE*, cf. Sec.4.4) continue to ignore this kind of information, expecting conversion of such documents to plain text format prior to processing.

However, while plain text is the least common denominator of all text formats, it is too limited for our purposes. Converting structured text formats to plain text throws away information which might turn out to be useful during the information extraction process—a concern expressed by the *"Structure matters"* conjecture (cf. Sec. 8.1). The same problem might occur if one structured format is converted into another. For example, HTML, the "lingua franca" of the Web, only contains a very limited set of generic structuring elements (some kinds of lists, section headers, and emphasized texts, and not much more). Conversion of richer formats such as *DocBook* [Wal99] or LaTeX is thus necessarily a lossy process—both comprise a much richer set of structural predefined elements (e.g. footnotes or appendices); DocBook also contains elements with a more semantic focus (such as *caution*, *important*, *note*, *tip*, or *warning*), while LaTeX allows extending the element set with any self-defined commands.

Because of this, deciding to use a richer predefined text format instead of plain text would not fix the basic issue—the richer the chosen format, is harder is becomes to write adequate converters for it, and in no case it would be able to cover self-defined commands such as supported by TEX adequately since their semantics is not known to a converter.

### 9.2.2 XML as "Universal" Input Format

The preferable alternative is thus to decide on a *meta-format* that can capture the *syntax* of a rich variety of formats instead of trying to unify their *semantics*. The obvious choice for a meta-format is *XML* [XMLa], the generic markup language that has gained widespread acceptance as a meta-format for any kind of both loosely and highly structured data.

Many current text formats such as HTML, DocBook [Wal99], *TEI P4*[1] and the *OpenDocument Format* [ODF][2] are already based on XML or its predecessor and superset SGML. XML-based formats do not need any preprocessing (once they fulfill the assumption we will discuss below in the last section of this paragraph), while SGML-based formats can be converted automatically to XML if the used *Document Type Definition (DTD)* is known.

Plain text could be trivially converted to XML by wrapping a whole document into a root element and escaping all characters that require escaping. However, while plain text does not contain any *explicit* structural markup, it frequently follows implicit conventions to express emphasis, lists elements and other structural information (e.g. in e-mails). Aiming to make this implicit markup available for the information extraction process requires a more complex heuristic conversion process—we will return to this point in Section 12.1.

For the purposes of this work we will consider it sufficient to accept plain text, HTML, and any XML-based formats as input, since other formats can generally be easily converted into one of these formats.

When extracting from XML documents, we assume that all information to extract is available in the textual content of the document—we will not try to extract from the values of XML attributes (XML elements and attribute name/value pairs *are* used for building the context representations of the tokens to classify as described in Sec. 12.2, but all tokens to classify are taken from the textual content of the document.) This corresponds to the content model of typical XML-based text formats such as (X)HTML or DocBook, where attribute values are used to store meta-information about the text or fragments of the text (e.g. formatting details or hyperlinked URLs attached to a text fragment), but not for storing visible text fragments.

[1] A format defined by the *Text Encoding Initiative* for for use in humanities, social sciences and linguistics [SM04].

[2] Used by OpenOffice <`http://www.openoffice.org/`> and other text processing applications. Actually, ODF files are ZIP archives bundling several compressed files, but the actual textual content of the document (with some style and formatting information) is stored in an XML file within the archive.

## 9.3 Input Formats for Answer Keys

Answer keys contain the "true" (gold-standard) information on the attribute values that can be found in a text. In the training stage they are used to train the system so it can learn how to recognize attribute values; during evaluation, the attribute values proposed by the system are compared with the answer keys to calculate performance metrics (cf. Chap. 15). During the extraction phase, obviously, they are unavailable to the information extractor.

The most simple and most robust way of providing answer keys is by storing them inline within a document. If this is not possible, they can be stored externally in a separate file or a database, along with information that allows locating them in the input text. These two options will be discussed in the following subsections.

### 9.3.1 Support for Inline Annotations

Answer keys are text fragments (strings) from an input text. They can be marked inline within a text by inserting a marker identifying the begin and another marker identifying the end of each attribute value. The markers must also reveal the attribute name for each attribute value (at least the begin marker—for end markers this information is redundant since we assume attribute values to be continuous and non-overlapping, cf. Sec. 8.3).

The most common way of doing this is by wrapping each answer key within a pair of XML tags: `<att>...</att>`, where `att` is the name of the attribute. This is also the style of inline annotation we support in our system.

Some older corpora such as the *Seminar Announcements* corpus (cf. Sec. 17.1) use such XML-style tags within plain text files.[3] Files annotated in such a way cannot be processed by standard XML parsers since they are not well-formed. We use the XML repair algorithm described in Chapter 13 to convert such files into regular XML.

### 9.3.2 Support for External Annotations

Inline annotation of answer keys is not always possible, for example, if an input text already contains XML markup there is the risk of overlapping conflicts. Also, any inline annotations contained in a text must be removed prior to preprocessing the text to ensure that no "forbidden" information about the true attribute values is leaked to the information extractor during evaluation.

Because of this, we support external annotation of answer keys as an alternative to inline annotations—any inline annotations are converted to external annotations prior to (pre)processing. External annotations need to carry four items of information: the text fragment to extract, the attribute name, a pointer to the text containing the text fragment, and a pointer to the position of the fragment within the text that

[3] Meanwhile, a variant of the *Seminar Announcements* corpus stored in regular XML files has been published by the University of Sheffield's *Dot.Kom Project* <`http://nlp.shef.ac.uk/dot.kom/resources.html`>, but it was not yet available when we started our experiments on that corpus.

```
Type:     cmu.andrew.official.cmu–news
Topic:    CEDA Spring Lecture Series
Dates:    13–Feb–95
Time:     3:30 PM
PostedBy: Edmund J. Delaney on 9–Feb–95 at 10:18 from andrew.cmu.edu
Abstract:

CENTER FOR ELECTRONIC DESIGN AUTOMATION SPRING LECTURE SERIES

The Center for Electronic Design Automation, CEDA, in the department of
Electrical and Computer Engineering will offer its first lecture
in its Spring lecture series on February 13, in the Adamson Wing, Baker ↩
   Hall.
The lecture begins at 3:30 p.m followed by a reception in Hamerschlag
Hall, Room 1112. Professors Rob A. Rutenbar and Wojciech Maly
will speak on "The State of the Center for Electronic Design Automation".
Funded in part by the Semiconductor Research Corporation, SEMATECH,
NSF, and by U.S. and international semiconductor companies, CEDA involves
12 faculty and 60 graduate students working on software tools to design,
verify and fabricate next–generation integrated circuits and systems.
```

Figure 9.1: Sample Input Text

allows anchoring the fragment. We store this information in the following relation (the underlined attributes form the primary key):

AnswerKeys (Type: string, <u>Text: string, Source: string, FirstTokenRep: integer</u>, Index: integer)

The first three attributes are straightforward: Type is the name of the attribute, Text is the text fragment to extract, Source is the file name of the text containing the answer key. Storing the anchoring information in a reliable way is more tricky. An intuitive idea would be to count the characters of the document and specify the index position of the first character of the text fragment to extract. However, if we count *all* characters including whitespace, we are bound to run into trouble, since the amount of whitespace in HTML/XML documents is usually insignificant and likely to change, e.g., during linguistic preprocessing. Ignoring whitespace and specifying the index position in regard to *printable* (non-whitespace) characters only is more reliable—this is the information stored in the Index attribute.

However, since the index position is still not a way to anchor answer keys that is sufficiently robust to always survive the preprocessing process described in Section 12.1, we do not use this attribute for locating attribute values, instead treating it as redundant information that is optional (may be NULL). The index position is unreliably since sometimes a few printable characters are deleted (probably never added) during preprocessing. For example, the *txt2html* converter we use for handling plain text documents will convert lines starting with a stand-alone '*' or '-' character into list item elements (`<li>...</li>`), deleting the item marker ('*' or '-'). Converters from other documents formats (which have not been used by us but might be configured by users for our system) might show a similar behavior. Thus we cannot rely on the number

```
Type:     cmu.andrew.official.cmu-news
Topic:    CEDA Spring Lecture Series
Dates:    13-Feb-95
Time:     <stime>3:30 PM</stime>
PostedBy: Edmund J. Delaney on 9-Feb-95 at 10:18 from andrew.cmu.edu
Abstract:

CENTER FOR ELECTRONIC DESIGN AUTOMATION SPRING LECTURE SERIES

The Center for Electronic Design Automation, CEDA, in the department of
Electrical and Computer Engineering will offer its first lecture
in its Spring lecture series on February 13, in the <location>Adamson ⤸
   Wing, Baker Hall</location>.
The lecture begins at <stime>3:30 p.m</stime> followed by a reception in ⤸
     <location>Hamerschlag
Hall, Room 1112</location>. <speaker>Professors Rob A. Rutenbar</speaker> ⤸
    and <speaker>Wojciech Maly</speaker>
will speak on "The State of the Center for Electronic Design Automation".
Funded in part by the Semiconductor Research Corporation, SEMATECH,
NSF, and by U.S. and international semiconductor companies, CEDA involves
12 faculty and 60 graduate students working on software tools to design,
verify and fabricate next-generation integrated circuits and systems.
```

Figure 9.2: Sample Text with Inline Annotations

of printable characters up to the start of an answer key to be exactly the same before and after preprocessing.

Instead we rely on the FirstTokenRep attribute for anchoring the text fragments. The value of this attribute is determined by tokenizing (cf. Sec. 12.3) both the text fragment to extract and the input text and indexing the occurrences of the *first* token of the text fragment in the text—for example, the FirstTokenRep attribute for a SPEAKER answer key **"Dr. Werner Koepf"** will be set to 0 if it starts at the first occurrence (= "0-th repetition") of the token "Dr" in the source text, or 1 if it starts at the second occurrence of this token. This method is reliable as long as attribute values do not start with an "endangered" token such as "*" or "-" which is very unlikely (in the corpora we used, this never occurred).

Figure 9.1 shown an example input text—a real example from the *Seminar Announcements* corpus (cf. Sec. 17.1) which was already shown as example in Section 3.4.[4] Figure 9.2 shows the same text with the inline annotations that are part of the original markup where the attribute values to extract are enclosed in XML-style tags.[5] Table 9.1 lists the answer keys for this file as external annotations.

[4] For the example given in Sec. 3.4, we omitted the last sentences and modified the headers to fit e-mail conventions.

[5] The original markup for this corpus also contains tags around each *paragraph* and *sentence*. They are not shown since they have not been used by our system nor (to our knowledge) by other systems evaluated on that corpus.

| **Type** | **Text** | **Source** | **FirstTokenRep** | **Index** |
|---|---|---|---|---|
| stime | 3:30 PM | cmu-news-2450 | 0 | 26 |
| location | Adamson Wing, Baker Hall | cmu-news-2450 | 0 | 91 |
| stime | 3:30 p.m | cmu-news-2450 | 1 | 101 |
| location | Hamerschlag Hall, Room 1112 | cmu-news-2450 | 0 | 110 |
| speaker | Professors Rob A. Rutenbar | cmu-news-2450 | 0 | 116 |
| speaker | Wojciech Maly | cmu-news-2450 | 0 | 122 |

Table 9.1: Example of External Answer Keys

## 9.4 Serialization of Extracted Attribute Values

The output of the information extractor is a set of proposed attribute values for each input text. The information we are interested in for each extracted attribute value is basically the same as for answer keys: the extracted text fragment, the attribute, and anchoring information pointing to the text containing the text fragment and its position in the text (the latter information is not strictly necessary for users that only want to work with the extracted data, but it is essential for evaluating the extractor and it will be useful for users interested in some background knowledge or doubtful about the extractions proposed by the system).

Additionally, the probability estimation assigned by the statistical system is of interest—users might want to filter extracted attribute values whose probability is below a certain threshold or they might re-check them manually. Thus extracted attribute values are stored in this relation:

> Extractions (Type: string, $\underline{\text{Text: string, Source: string, FirstTokenRep: integer}}$, Index: integer, Probability: real)

The first five attributes correspond to those of the AnswerKeys relation described above (Sec. 9.3.2); the Probability attribute stores a number in the $[0.0, 1.0]$ range giving the probability estimation of the extraction to be correct.