Efficient Integrated Tagging of Word Constructs

Andrew Bredenkamp
Frederik Fouvry
Dept. Language and Linguistics
University of Essex
Wivenhoe Park
Colchester
Essex CO4 3SQ
United Kingdom
{andrewb,fouvry}@essex.ac.uk

Thierry Declerck
IMS
University of Stuttgart
D-70174 Stuttgart
Germany
thierry@ims.uni-stuttgart.de

Bradley Music
Center for Sprogteknologi
Njalsgade 80
DK-2300 Copenhagen S
Denmark
music@cst.ku.dk

Abstract

We describe a robust text-handling component, which can deal with free text in a wide range of formats and can successfully identify a wide range of phenomena, including chemical formulae, dates, numbers and proper nouns. The set of regular expressions used to capture numbers in written form ("sechsundzwanzig") in German is given as an example. Proper noun "candidates" are identified by means of regular expressions, these being then rejected or accepted on the basis of run-time interaction with the user. This tagging component is integrated in a large-scale grammar development environment, and provides direct input to the grammatical analysis component of the system by means of "lift" rules which convert tagged text into partial linguistic structures.

1 Motivation

1.1 The problem : messy details

Messy details are text constructs which do not lend themselves well to treatment by traditional techniques for linguistic analysis, whence their 'messiness'. Typical examples are numbers, codes or other (sequences of) word-forms which can occur in many variations (often infinite), making impossible a comprehensive treatment by traditional means.

There are various types of phenomena classified as messy details which can be subclassified according to at what level of generality as regards text structure they occur, viz. general format level, sentence level and word level phenomena. *General format level* phenomena occur over sentence boundaries, example being headers, meta-comments and tables. Phenomena classified as *sentence level* occur within a single sentence, but cannot be considered word constructs of a fixed nature. These are more 'linguistic' than the usual messy details, but are considered messy details since they lend themselves to partial analysis via a similar type of pre-processing. Examples of these are the use of parentheses and commas which can be used within practical implementations as a basis for segmentation during pre-processing.

Word level phenomena are usually the most frequently occurring messy details, including such things as dates, document references of various sorts, codes, numbers and proper nouns. For any realistic application these types of construct must be processed efficiently, the alternative being coding them individually in some lexicon and/or implementing sets of grammar rules for parsing them syntactically.

This problem area was given priority in the EU-sponsored LSGRAM project (LRE 61-029) which aimed to integrate an approach to messy details into a large-scale grammar implementation. The coverage of the grammars developed was based on corpus analyses within each language group of the project, these revealing a large number of messy details of the types mentioned. What was called for then was an efficient means of identifying word-level messy details (or *word constructs*) such that they could be processed in a general way, avoiding additional grammar rules and the need for an infinite number of lexical entries.

2 The basic approach

2.1 Identification using regular expressions

The types of word construct of interest here lend themselves well to identification by matching regular expressions over each input sentence (considered as a record), tagging them as specific instances of general phenomena (e.g. dates, numbers, etc.).

awk is a programming language specifically designed for in this type of string manipulation (matching, replacement, splitting). It has special provisions for treating text in the form of records. awk reads input record by record, matching user-defined regular expressions and executing corresponding actions according to whether a match has been found.

The regular expressions can be stored in variables and reused to build more complex expressions. This is important, as some of the phenomena we were attempting to match were complex (see below) and occurred in a variety of formats.

The awk-implemented tagger developed for this project, tagit, can be used as a stand-alone tagger for SGML texts. It has been integrated within the text handling procedures of the ALEP system after sentence recognition and before word recognition. When a pattern matches against the input, the matched string is replaced with a general tag of the relevant type (e.g. DATE, NUMBER). Subsequent tagging and morphological parsing then skip these tags, and further processing (i.e. syntactic analysis) is based on the tag value, not the original input string.

2.2 Sample case : currency patterns in German

tagit has been integrated into the German LS-GRAM grammar for the identification of word constructs occurring in the mini-corpus taken as the departure point for the work of the German group, consisting of an article on economics (taken from the weekly newspaper "Die Zeit"). As usual when using 'real-world' texts, many messy details were found, including dates and numbers used within percentages and, as would be expected from the text type, within amounts of currency. These occur both with and without numerals, e.g. "16,7 Millionen Dollar", "Sechsundzwanzig Milliarden D-Mark". The text examples are especially problematic given the German method of expressing the ones-digit before the tens-digit, e.g. "Sechsundzwanzig" is literally "six-and-twenty".

In order to deal with this phenomenon, regular expression patterns describing the currency amounts were defined in awk. First, patterns for cardinals were specified, e.g.[1]

```
two_to_nine = "([Zz]wei|[Dd]rei|[Vv]ier|"\
   [Ff]unf|[Ss]echs|[Ss]ieben|[Aa]cht|[Nn]eun)"
one_to_nine = "([Ee]in|"two_to_nine")"
card = "("one_to_nine")"
number = "[0-9]+(,[0-9]+)?"
range = "("number"|"card")"
```

The actual pattern used in the implementation is more complex and goes up to 999, but the example shows the principle. Given this set of variables, the pattern assigned to card can match the text version of all cardinal numbers from 1 to 999, e.g. "Drei", "Neunzehn", "Zweiundzwanzig", "Achthundert Fünfundvierzig", etc. The value assigned to range can match number, optionally with a comma as decimal point, e.g. "99,09". The following patterns are also needed :

```
amount = "(Millionen|Milliarden)"
currency = "(Mark|D-Mark|Dollar)"
curmeasure = "(("amount"( "currency")?)|"\
   "("currency"))"
measure = "("range" "curmeasure")"
```

The last two patterns described define measure being the succession of a cardinal number (as a digit or a string) followed by curmeasure, being the concatenation of amount and currency. But both of them, amount and currency, are defined as being optional. So that inputs like "30,6 Milliarden Dollar", "Zweiundzwanzig Dollar" or "Dreiundvierzig Milliarden Dollar" are automatically recognized. But the definition of 'measure' disallows the tagging of "Zweiundzwanzig" as a 'measure' expression. The tag provided for this string will be the same as for any other cardinals.

tagit applies these patterns to each record within the input, assigning the appropriate tag information in case a match is found. Further processing is described below.

3 Extension for proper nouns : interactive tagging

Proper nouns present another problem that falls under messy details. A small extract from the corpus used for the English grammar showed a wide range of possible proper noun configurations : "James Sledz", "Racketeer Influenced and Corrupt Organizations", "Sam A. Cali", "Mr. Yasuda", "Mr. Genji Yasuda", ...

Regular expressions can catch several of those cases, but it is difficult to get certainty, e.g. "Then Yasuda ..." vs "Genji Yasuda" : one can never be sure that an English word is not a name in another language. Since this is a pre-processing treatment, there is no disambiguating information present, and fully automatic tagging cannot be

[1]Umlauted characters and "ß" are matched by the system, though they are not shown here.

Note that regular expressions are specified as strings and must be quoted using pairs of double quotes. Variables are not evaluated when they occur in quotes, so quoting is ended, and then restarted after the variable name, whence the proliferation of double quotes within the complex patterns.

Some awk syntax : "=" is the assignment operator, parentheses are used for grouping, "|" is the disjunction operator, "?" indicates optionality of the preceding expression, "+" means one or more instances of the preceding expression, square brackets surround alternative characters (possible specified as a range, e.g. "[0-9]").

done, unless the program can have access to either some lookup facility and/or can interact with a human user.

3.1 Patterns for proper nouns

For financial texts, the domain of our reference corpus, the proper nouns are company or institution names and person names. Product and company names can be very unconventional. Therefore the regular expressions need to be rather generous. The interaction with the user and the dictionaries will provide a way to tune the effect of these expressions.

We defined the proper noun regular expression to be nearly anything, preceded by a capital. Person names can contain initials, and they might be modified by titles ("Mr", ...) or functions, business names can be modified by some standard terminology (like "Ltd."). Lower case words are allowed if they are not longer than three characters (for names containing "and" etc.).

3.2 Interacting with the user

Tagging proper nouns presents a special problem, since, unlike the case of numbers and dates, there is a great deal of uncertainty involved as to whether something is a proper noun or not. Therefore a natural extension to `tagit` was the implementation of an interactive capability for confirming certain tag types such as proper nouns.[2]

If a proper noun is found, then the tagger first does some lookup to limit the number of interactions during the tagging. We used the two following heuristics :

1. Has it already been tagged as a proper noun ? If so, do it again.
2. Has it already been offered as a proper noun, but was it rejected ? If so, and if it occurs at the beginning of a sentence, reject it again.

Those two checks are kept exclusively disjunctive. If a word occurs both as a proper noun and as a "non-proper noun", the user will be asked if he or she wants it to be tagged. This allows one to use different name dictionaries for different texts.

If the program itself is certain that a proper noun is found, then it tags it and goes on to a next match. Otherwise it asks the user what to do with the match that was found. There are two possible answers to this question :

1. The user **accepts** the match as a proper noun. The program tags it, stores it for future use, and proceeds.

   When the match is not entirely a proper noun, the matching string can be edited. This consists of removing the words before and/or after the first proper noun in the match.[3] The remaining substring of the match is tagged as a proper noun and stored. The words before the first word are skipped (and also stored); everything that comes after the tagged proper noun is resubmitted.
2. The user **rejects** the match that is offered. The program stores it (as a "non-proper noun") and proceeds.

4 Integration with linguistic analysis

The ALEP platform (Alshawi et al., 1991) provides the user with a Text Handling (TH) component which allows a "pre-processing" of input. An ASCII text will first go through a processing chain consisting in a SGML-based tagging of the elements of the input. The default setup of the system defines the following processing chain : the text is first converted to an EDIF (Eurotra Document Interchange Format) format. Then three recognition processes are provided : paragraph recognition, sentence recognition and word recognition. The output from those processes consist of the input decorated with tags for the recognized elements : 'P' for paragraphs, 'S' for sentences, 'W' for words (in case of morphological analysis, the tag 'M' is provided for morphemes) and 'PT' for punctuation signs. Some specialized features are also provided for the tagged words, allowing to characterize them more precisely, so for example 'ACRO' for acronyms and so on.

So the single input "John sees Mary." after being processed by the TH component will take the form :

```
<P> <S> <W>John</W>
        <W>sees</W>
        <W>Mary</W>
        <PT>.</PT>
    </S>
</P>
```

`<P>` and `</P>` mark the beginning and the respective ending of the recognized paragraph structure. The other tags must be interpreted analogously.

In the default case, it this this kind of information which is the input to the TH-LS component (Text-Handling to Linguistic Structure) of the system. Within this component, one specifies so-called 'ts_ls' (text structure to linguistic structure) rules, which transform the TH output into

[2]The graphical interface to the interactive tool has been implemented in Tcl/Tk.

[3]To *extend* the matches, the user would need to change the regular expressions.

partial linguistic structure (in ALEP terminology, this conversion is called *lifting*). The syntax of these *lift rules* is the following :

```
ts_ls_rule( <ld>, <tag_name>,
            [<features>], <tag_content> )
```

where : `<ld>` is a Linguistic Description (LD); `<tag_name>` is the name of an SGML tag (e.g. 'S', 'W'); `<features>` is a list of feature-value descriptions of the tag's features; `<tag_content>` is the atomic content of the string within the tag (optional in the *lift rule*).

This kind of mapping rule allows a flow of information between text structures and linguistic structures. So if the input is one already having PoS information (as the result of a corpus tagging), the TH-LS is the appropriate place to assure the flow of information. This allows a considerable improvement of parse time, since some information is already instantiated before the parse starts.

The TH component of the ALEP platform also foresees the integration of user-defined tags. The tag <USR> is used if the text is tagged by a user-defined tagger, as is done when processing messy details.

When `tagit` matches a pattern against the input, the matched string is replaced with an appropriate USR tag. Thus "Dreiundvierzig Milliarden Dollar" is matched by the pattern `measure` (see above), and is replaced by the SGML markup `<USR VAL="Dreiundvierzig Milliarden Dollar" LEVEL=M TYPE=MEASURE>Dreiundvierzig_Milliarden_Dollar </USR>`

Note that the matched sequence is copied into the attribute VAL and that in the data content spaces are replaced by underscores. For some pattern types, a generalized representation of the matched sequence is computed and stored in an attribute CONV. For instance, when the pattern for dates matches the input "March 15, 1995", CONV is assigned a standardized version, i.e. CONV="95/03/15".

This version with USR tags inserted is then processed by the set of lift rules. The following general lift rule does the conversion for all USR tags :

```
ts_ls_rule(
ld:{ sign => sign:{
        string => STRING,
        synsem => synsem:{
           syn => syn:{
              constype => morphol:{
                 lemma => VALvalue,
                 lu => TYPEvalue } } } } },
'USR', ['TYPE'=>TYPEvalue, 'VAL'=>VALvalue],
STRING ).
```

Here we can see the mapping of information between the user-defined USR tag (the attributes of which are listed in the last line of this rule) and the linguistic description ('ld'—'linguistic description', a structured type within the Typed Feature System), using the rule-internal variable TYPEvalue: the value of the attribute TYPE is assigned to the lexical unit ('lu') value of the linguistic description. After applying this rule to the result of matching "Dreiundvierzig Dollar", the ld is the following :

```
ld:{ sign => sign:{
          string => 'Dreiundvierzig_Dollar',
          synsem => synsem:{
             syn => syn:{
                constype => morphol:{
                   lemma => 'Dreiundvierzig Dollar',
                   lu => 'MEASURE' } } } } }
```

Although the original input sequence is available as the value of the feature `lemma`, further processing is based solely on the `lu` value `'MEASURE'`, thus making it possible to have a single lexical entry for handling all sequences matched by the pattern `measure` shown above. The definition of such generic entries in the lexicon keeps the lexicon smaller by dealing with what otherwise could only be coded with an infinite number of entries. In addition, treating such word constructs as a single unit gives a significant improvement in parsing runtime, since only the string `'MEASURE'` is used as a basis for further processing, instead of the original sequence of three words. Finally, runtime is also improved and development eased by the fact that no grammar rules need be defined for parsing such sequences.

5 Conclusion

The implementation described here handles a variety of word-level messy details efficiently, speeding up overall processing time and simplifying the grammars and lexica. General format level and sentence level phenomena can be handled in a similar way. Within our project, reimplementation using a more powerful tool `perl` is taking place, allowing further extensions to the functionality.

We maintain that user-interaction combined with some table lookup is the only viable approach to the robust tagging of free texts. The fact that an interactive tagging tool can be so easily integrated in to the linguistic processing system is of obvious and considerable benefit.

References

Alshawi H., Arnold D. J., Backofen R., Carter D. M., Lindop J., Netter K., Pulman S., Tsuji J., Uszkoreit H. 1991. Eurotra ET6/1: Rule Formalism and Virtual Machine Design Study (final report). CEC