8

How to Generalize the Task of Annotation

STEVE FLIGELSTONE, MIKE PACEY
and PAUL RAYSON

8.1 Introduction

In the last 20 years, UCREL's principal technique for automatic grammatical analysis has been a probabilistic one. Training methods for this technique usually rely upon large bodies of text analysed in advance either completely by hand or by machine and then corrected by grammarians (see Garside *et al.* 1987, Black *et al.* 1993).

Such probabilistic algorithms have achieved high success rates when applied to part-of-speech (POS) tagging. However, with a baseline of 96–97 per cent accuracy, the amount of data needed to train more accurate models increases exponentially, and it is not clear that training from a hand-corrected 1000–million word corpus would decrease errors dramatically. Therefore, as a complement, and occasionally as an alternative, to probabilistic methods, UCREL increasingly employs **template analysis** techniques, with programs such as JAWS and the rule-based component of CLAWS (described in Fligelstone *et al.* 1996) all using closely related and essentially similar template-based techniques to reduce errors and/or ambiguity. Template-based methods are applied more generally, without a statistical counterpart, in semantic annotation: for example in linking nouns with textually-related adjectives or verbs (see Wilson 1993, Wilson and Rayson 1993).

With further projects developing along similar lines, in order to avoid further duplication of programming effort when implementing template methods, and to develop a flexible system for developing and evaluating rule sets, we decided to build a general purpose rule interpreter or **Template Tagger**. It has already been tried out on the problems for which we have individual tools, but has also found application in new analytical tasks. In due course, deployment of the tagger in new problem areas will enable us to see what further features could usefully be incorporated.

In the following sections, we describe the main characteristics of the

Template Tagger, discuss its development to date, and refer to specific areas of application.

8.2 Framework for a Template Tagger

8.2.1 General aims

Our goal in developing the Template Tagger was to create a general-purpose program which could be used to apply rules for text annotation irrespective of the particular analytical task in question, and in a way which could utilize, and if necessary amend, any existing mark-up in the text input. The program was required for immediate application to a particular multi-level analysis project,[1] but was designed from the outset with the intention of using it for other tasks, as described in Sections 8.4.1–3.

This was achieved by making explicit various aspects of the annotation procedure which are implicit in the task-specific systems referred to above. To illustrate this point, let us consider a simple rule from the contextual pattern rule set (see Section 7.5), more commonly known as the CLAWS idiomlist:

```
a DD231, great DD232, many DD233
```

Without going into too much detail about the underlying system, the basic purpose of this rule is to tag any instance of *a great many* as a complex determiner (DD2),[2] rather than to allow the system to tag each separate word on a word-by-word basis. What is significant for our argument here is the brevity of the rule, made possible by the fact that the system operates within a tightly confined framework of possible input and output tokens, and tagging operations. In this case the system 'knows' that *a great* and *many* are parts of the lexical **input**, and that the DD231, DD232, etc. are candidate POS tags, i.e. **output** tokens, which must in this case *replace* any earlier list of candidate tags in the event of a match.

Slightly different conventions are used with the JAWS system to format a rule used to identify an active, as opposed to passive, use of a past participle:

```
VH*   R*   V*N[PERF]   {by}
```
[3]

In this rule, white space, rather than commas, is used as a separator between input tokens; the output token is indicated by the use of square brackets; and the effect, or 'action' of the rule, is not to add anything to a list of candidate tags, but to modify the existing VVN POS-tag in such a way as to mark it as definitely active, rather than ambiguously active-passive, in other words, to replace one POS tag with another (more informative) one. Note also that in this rule the input tokens are a mixture of wildcarded[4] POS tags (VH*, R*, etc.) and lexical items ('by'), and that in this rule scheme, the use of curly brackets is used to indicate lemmas.[5]

Both these systems are able to use very terse rule formats because they each operate within a confined, albeit different, framework. Although the default tagging operation in each system is subtly different, it is always the same, so needn't be spelled out, and the possible types of input allowed by each system are so confined (in CLAWS' case only words and POS-tags are present, and in JAWS' case, only these plus lemmas), that mere position or use of brackets can suffice to avoid any ambiguities.

But what if we wish to apply not only POS tags, but also semantic tags, or an additional level of grammatical tags or dependency tags? And what if we wish such tags to be available as input against which to match and fire further rules? And what if we don't always wish the same action to be undertaken, but sometimes one action (e.g. 'add tag to list') and sometimes another (e.g. 'overwrite existing tag with new tag'), or yet another, (e.g. 'delete this tag')? And finally, what if we wish to perform all these tasks with a single system, rather than having to create customized software for each task?

In this case, we would need to re-think the rule format cited above, in order to make it quite clear to a generic system

- which parts of the rule specify input and which output
- at what level of input/output the particular tokens are significant
- precisely what action the rule is to perform.

We might represent the JAWS rule as in Figure 8.1. Although making such information explicit leads to less succinct rule formats, it enables us to design a flexible multi-purpose system required for implementing novel annotation regimes.

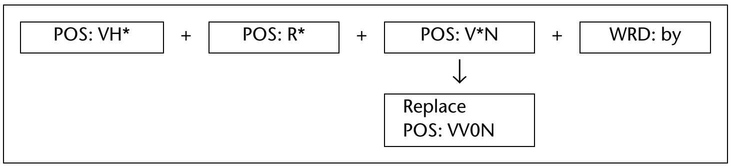

Figure 8.1 Explicit representation within a template rule

8.2.2 Multiple levels of annotation

In the foregoing section we have referred to distinct **levels** of annotation (see Section 1.5), such as 'the POS tag level'. As a general purpose tagger, the program should support multiple levels of annotation, but we did not wish to pre-judge the range of types of annotation which would be present in the input, or which the tagger would be required to apply. We see the normal operation of the tagger as accepting text with *n* levels of annotation and allowing the user to add one or more levels of annotation to this, building on levels already contained in the input, or simply to alter existing levels of annotation.

The term 'level' is used here to signify a *type* of information. Just as there is no pre-defined set of levels, nor is there any implicit structure or hierarchy of levels. The user is at liberty to use any number of levels, with any significance appropriate to the task and the rules written for it. It would be perfectly possible, if rather pointless, using such a system, to tag all words beginning with 'a' as having the value 'A' at some arbitrary level 'first letter' for which the code FSL has been chosen by the user. The following single rule would suffice.

```
Pattern(WRD: a*/A*)    Action(Inserttag(FSL: A))
```

Assuming that the Template Tagger is able to identify the WRD level in its input stream, the only special information required by the program to apply such a rule is a declaration in its configuration file that the levels WRD and FSL are to be used (see Section 8.2.3).

In its original conception, the Template Tagger allows any number of levels to be declared and used, and rule sets have been developed which manipulate annotations at several levels simultaneously, allowing, for example, grammatical decisions to be based on semantic features and vice versa.

The only obligatory analytical level is the word (or punctuation item) itself (level: WRD). All other levels, and the three-letter codes used to represent them, are introduced at the user's discretion.

For the program to know to which level a particular input token belongs, one of two approaches may be taken. Either all input must be formatted in such a way that each item is explicitly labelled, or the program must be adapted to 'understand' a particular input format. So far, we have taken the latter approach, since we already have well-established text formats in constant use, but the requirement for a particular input 'parser' for any given task is obviously at odds with the goal of a truly generic tool. However, care has been taken to adopt a modular approach to the program construction (see Section 8.3.2), in order to ensure that the heart of the program is universally applicable, with changes only to the input and output modules required to handle new input formats and output requirements. A default output format in which each token of output is labelled for level is also available.

8.2.3 User control

We have already seen that the user must declare which analytical levels are to be handled by the program. Experience has shown that there are a number of other factors which it is useful to be able to vary with rule-driven analyses of the kind the Template Tagger would be used to perform. These include:

1. whether to split the rules into separate files (either for ease of editing, modularity, or selective re-application),
2. in what order to apply the various rule tables,
3. whether to stop searching a rule table once a rule has fired successfully or to continue and apply some heuristic for overlapping items, and
4. how to apply any action specified when a rule fires.

All these choices can be controlled by the user of the system in the **configuration file**, with defaults to enable the novice to achieve simple results without too much knowledge. The configuration file contains two sections, the first listing the levels relevant to the current task, and the second outlining the tagging strategy to be employed.

The only purpose served by the first section in its present form is to provide a check list so that the program may detect when an invalid rule is encountered, i.e. one which contains a reference to an invalid level. In due course the error-checking function of this section could be extended by including names of files containing validation rules for values at the various levels. This could include tag-lists, for closed label sets such as POS tags, or more general format rules for more open-ended annotations. A further type of declaration which belongs in this section concerns the level-specific formatting for the as yet unimplemented 'indexing' function, referred to in the following section.

The second section tells the program where to find the rules and what to do with them; specifically, for each rule file named, how many times to cycle through the rule set before proceeding to the next file, and whether to search to the end of the file in question in all cases ('through' mode), or whether to quit the rule table as soon as a rule is fired ('hit' mode).

8.2.4 A range of tagging operations

The action part of a rule is carried out when a rule fires successfully. Depending on the application, we might need to replace completely the contents of one level (e.g. a complete disambiguation of a set of POS tags) called 'HardInsert', or add a tag or marker to a list of existing values (called 'Append'). 'SoftInsert' adds a tag only if there is currently no value at the specified level (i.e. it can't overwrite or append). This is useful for ensuring that potentially recursive rules fire only once, such as when identifying the opening of a noun phrase. A function 'Void' has the effect of deleting any existing values at the specified level. We also envisage a 'Remove' function which would remove one value from a list on one level for partial disambiguation purposes, a 'Promote' function which would re-order values stored at a particular level, an 'Index' function which would generate numerical indices for linking items, whether adjacent or non-adjacent, e.g. for anaphoric links or indexed subject-object linking, and functions for modifying existing tagging, e.g. by the addition of a subscript.

One of the most challenging aspects of the development of this software has been to devise and define the minimum range of operations which will cater for most if not all of our tagging requirements.

8.3 Creation of the Tool

8.3.1 Development of the Template Tagger to date

Version 1 of the Template Tagger was written in 1994–95 to apply multiple levels of annotation to text already annotated by CLAWS and subsequently by JAWS (see Fligelstone 1995). Version 1 represents in many ways a prototype, with limited efficiency and some functions as yet unimplemented (see Section 8.2.4), but does adhere to the generic design principles outlined in the previous section.

Later in 1995, the Template Tagger was selected as the tool with which to undertake tag correction and enhancement work on the British National Corpus (BNC).[6] Unfortunately, the scale of the tagging task (some one hundred million words of input, applying several hundred rules) meant that the slow speed of the prototype, which was itself still under development, was too restrictive for the task in question.

Version 2 was thus commissioned; a more streamlined, cut-down version of the prototype. The main concession to efficiency was the dropping of unlimited LEVELS and VALUES. In the prototype, any number of these could be used. In Version 2, levels are hard-coded, and are limited to those relevant to the BNCTE project, and for any token, a maximum of six values may be stored at any given level. Version 2 thus represents a step forward in efficiency but a step back from the goal of a truly generic tool. It is to be hoped that a future Version 3 will marry the virtues of both. All versions have been developed in C on a UNIX platform.

In the following sections we discuss in more detail some features of the

Template Tagger's operations and design, with reference to the issues of flexibility and efficiency.

8.3.2 Input-output modularity

The Template Tagger was initially required to process a single format of corpus, the 'vertical' JAWS format, containing one word or punctuation item per line of input, along with a POS tag (the JAWS tag) and a string representing the lexical headword or lemma. However, the program was always intended to have wider applicability, so input and output modularity was implemented from the start.

The basic concept behind the Template Tagger is that any word-unit in a corpus can be split up into a number of LEVELS corresponding to different facets of information about the word (see Section 1.5). Each level may have one or more values (a familiar example of a multi-value level occurs in CLAWS vertical-format text, where each word may have a number of 'candidate' POS tags, listed in order of likelihood).

The pattern-matching engine within the Template Tagger operates on a sentence (or in later versions, 'unit') level. The Template Tagger reads in a sentence,[7] converts it into an internal data structure, processes it, and outputs the results. The internal data structure for a sentence is generic enough to allow for different formats of corpus. Handling a new corpus style is thus simply a matter of writing new input and output routines, in other words, customizing the tool to be able to assign to the appropriate level each token occurring in the input stream, and formatting the output according to requirements.

Ideally, we would like to create a system which would allow the user to 'explain' to the program how to interpret various input formats, but this is too ambitious at the present time. For now we have confined ourselves to the creation of routines selected by command-line options appropriate to the various formats with which we regularly work. A next step would be to create an input mode which expects all input to be labelled for level. The usefulness of this will be proportional to the extent to which there is agreement and take-up of standard conventions for producing explicitly labelled annotated text (see also Chapter 16 of this volume).

8.3.3 Rule-matching algorithm

The rule-matching algorithm is the heart of the Template Tagger. As has been mentioned, rule matching currently works on one sentence at a time – an attempt is made to match each rule provided by the user at each position in the current sentence, starting with the first word. If several rule files are to be used, the sentence is processed in its entirety using one file before proceeding to the next file and returning to the start of the sentence. If each rule file contains a different type of rule, this has the effect that the sentence, and by extension, the text, is subjected to one form of analysis before being subjected to another.

Each Template Tagger rule is composed of one or more 'cells', designed to match one or more adjacent items in the sentence. Multiple cells in a rule attempt to match adjacent items in the sentence. Each cell comprises a 'pattern' section and an optional 'action' section. The pattern section details a set of criteria that an input item (i.e. a word or punctuation item plus any associated annotation) must meet for the rule to match. To increase the power of pattern matching, a variant of UNIX-style regular-expressions may be used, including wild cards to represent multiple characters and negation (e.g. POS: NOT a noun).

The action section of the cell is optional, and contains a set of one or more operations to be carried out on the matching word if and only if the rule as a whole fires (i.e. if every cell in it matches). The most basic, and to date the most commonly used operation is 'HardInsert', which erases all current values (if any) at the specified level and replaces them with the value(s) specified in the rule. This operation is useful for adding completely new information, for enriching existing annotation (e.g. replacing general tags with more precise ones), and for tag correction. Actions may be included in any or all of a rule's cells.

An extension to the rule matching system is the optional cell. An optional cell does not have to be matched in order for the rule as a whole to fire. The optional cell takes a number argument, which specifies how many input items it may maximally match. The Template Tagger rule in Figure 8.2 (overleaf) is basically a rendition of the JAWS rule in Figure 8.1, except that the adverb cell (cell 2) is optional, and may match up to three consecutive adverbs. It is possible to include an action within an optional cell, but if the optional cell matches more than one item of input, the same action will be applied to all of them.

The optional cell device produces the problem of possible multiple matches for a rule from a single starting point (i.e. the rule may fire by either matching or omitting to match the optional cell(s)). In such instances, the Template Tagger employs a simple strategy of choosing the longest match. If there are competing match permutations of equal length from the same rule (a rare occurrence, in our experience to date), the Template Tagger will choose the rule-match which matches the cell(s) closest to the beginning of the rule.

The longest match principle also requires that care be exercised in writing rules containing optional cells in a medial position. For example a rule

```
<RULE>
        <NAME> Perfect-1
        <CELL>
                <PATTERN>
                        <LEVEL> POS
                        <VALUE> VH*
                </PATTERN>
        </CELL>
        <CELL>
                <OPTIONAL> 3
                <PATTERN>
                        <LEVEL> POS
                        <VALUE> R*
                </PATTERN>
        </CELL>
        <CELL>
                <PATTERN>
                        <LEVEL> POS
                        <VALUE> V*N
                </PATTERN>
                <ACTION>
                        <OPERATION> HardInsert
                        <LEVEL> POS
                        <VALUE> VV0N
                </ACTION>
        </CELL>
        <CELL>
                <PATTERN>
                        <LEVEL> WRD
                        <VALUE> by
                </PATTERN>
        </CELL>
</RULE>
```

Figure 8.2 Template Tagger rule with optional cell

intended to capture a noun and the next finite verb, regardless of intervening text, must contain an optional cell to represent that intervening text not as anything at all, but as anything which isn't a finite verb. Otherwise, it is the last finite verb in the sentence which will be matched by the rule, not the first one following the noun. Injudicious use of optional cells can cause whole sentences to be 'swallowed' by rules in this way.

8.3.4 *Invisibility*

When processing corpora, certain portions of the corpus may interfere with easy pattern matching. An example would be in spoken discourse where certain aspects of speech (coughs, fillers or pauses) have been transcribed, interfering with the flow of the text, and thus pattern-matching.

Invisibility allows the user to instruct the Template Tagger to ignore certain 'words' for the purpose of pattern-matching unless a cell's pattern is explicitly looking for it.

In Version 1, this took the form of a few hard-coded exceptions in the pattern-matching algorithm, notably the CLAWS NULL[8] tag and the CLAWS double-quotes tag. This element was enhanced in Version 2 to allow the user to specify a set of invisible words, based upon their POS values. Ultimately, invisible items should be user-definable with reference to any level or combination of levels.

Future work will focus around improving the TEI-conformant format output module, and increased user-control in the areas of corpus format and output options.

8.4 Areas of Application

8.4.1 Partial syntactic parsing

The first task to which the Template Tagger was put was not the one initially envisaged. Whilst it had been anticipated that the program would first be used for the purpose described in the next section, just as the program was approaching its first trials, Geoffrey and Fanny Leech outlined a set of rules which would produce, on the basis of POS-tagged input, a partial scheme of syntactic labels, not necessarily balanced, which they would then use as input to a further syntactic analysis program designed to complete the parsing task (cf. Garside and Leech 1985).

These rules lent themselves easily to conversion to the Template Tagger format, and the program was successfully deployed on this task. Two levels, LBR and RBR were introduced, to handle left (opening) bracketing and right (closing) bracketing respectively. Figures 8.3 and 8.4 contain examples of the kind of rules produced.

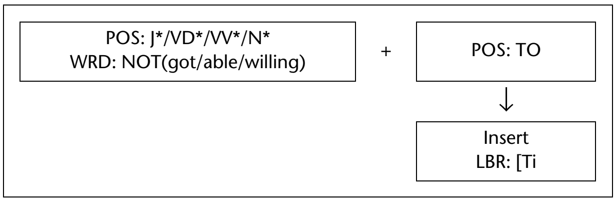

Figure 8.3 Rule to mark opening of an infinitive clause

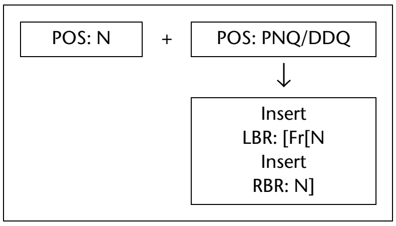

Figure 8.4 Rule to mark opening of a relative clause

8.4.2 Semantic tagging

The most elaborate use made of the Template Tagger to date has been in the creation of a multi-level annotated corpus with various kinds of grammatical and semantic information added. The progress made did not amount to a complete semantic analysis, as many more rules would have been required to make such a claim, but it none the less demonstrated the value of the program in applying a layer by layer analysis of text, using rules in later stages of the analysis which depended on information added at an earlier stage. All these stages could be incorporated into a single execution of the program so that the input text could be dealt with thoroughly, on a sentence-by sentence basis.

Input to the program was an enriched POS-tagged corpus which incorporated a column showing the lemma or lexeme (which allowed for more succinct rule writing than would have been possible had only the word been available), produced by the program JAWS, one of the earlier template-based annotation programs which in fact led to the development of the Template Tagger (see Sections 8.1 and 8.2 and Fligelstone 1995).

Rule files applied by the Template Tagger dealt with analytical issues such as clause boundary identification, subject identification, main dependencies, phrasal verbs, verb disambiguation, lexical look-up, fixed idioms, stereotypical sentences, and so on. Occasionally a rule would apply several levels of tagging at once (stereotypical sentences being a case in point), so that, for example, semantic information might be applied at a much earlier stage in the process than would normally be the case.

By combining all the levels of analysis into a single all-embracing tagging process, it was all too easy to commit the error of including a rule which could never fire because it depended on patterns which could only be produced by rules occurring later in the run. To overcome the problem it was found useful to adopt the convention of grouping rule tables into sets which were labelled A, B, C and so on. This was purely a matter of convention with no programming implications. The first few rule files consulted were A-files, followed by the B-files, and so on. Although an A-file rule could be used to apply any level of annotation, the patterns which its rules sought in the text could only contain the kind of information available in the JAWS output. A rule which required information about heads and clause boundaries, largely applied by the A-rules, would therefore have to be included in a B-file, and rules requiring higher level input such as semantic tags would be confined to the C-files, and so on.[9] Figure 8.5 demonstrates the output format and the levels of information encoded.

Ref. No		Pos	Word	Lemma	Clause	Dependent	Head	Function-Semantics
0000182	030	PPMS1	He	HE		<1>	&Nn	SBJ-Human-Male
0000182	040	VABD	was	BE				
0000182	050	VV0G	wanting	WANT		<N	&Va	VRB-Wanting-1
0000182	060	TO	to	TO	+TI			
0000182	070	VV0I	know	KNOW		<V	&Va	VRB-Thinking-Know
0000182	080	CSW	whether	WHETHER	+FN			
0000182	090	PPMS1	he	HE		<2>	&Nn	SBJ-Human-Male
0000182	100	VM00	could	COULD				
0000183	010	VV0I	expect	EXPECT		<N	&Va	VRB-Wanting-5
0000183	020	PPY00	you	YOU		<V	&Nn	OBJ-Human
0000183	030	IF	for	FOR			&P	ADV-
0000183	040	NN1	lunch	LUNCH			&Nn	Concrete-Food
0000183	041	.	.	.				

Figure 8.5 Sample Template Tagger output

8.4.3 British National Corpus enrichment and correction

The Template Tagger was used as an important part of the British National Corpus Tag Enhancement project (BNCTE) described in more detail in Chapter 9. The BNCTE team used Version 2 to apply rules for the assignment of part-of-speech tags that were too complex for the CLAWS tagging formalisms. For this task it was necessary to use only the levels WRD, POS and DEC. DEC is the CLAWS **decision code** – a two figure code indicating which part of the program (lexicon, suffixlist, etc.) had assigned the tag.

It had been found that there were errors in the CLAWS tagging of the BNC that could not be correctly resolved using the CLAWS resources. The Template Tagger, however, was powerful enough to encode a large number of more complex rules. While it was not necessary to use the full range of functionality of the Template Tagger, at least not in terms of levels, the BNCTE team did have to formulate some very complex rules. A further new problem was how to apply hundreds of rules to a 100 million word corpus while ensuring that the sequence of rules was correct to achieve the desired effect. Careful preparation and testing had to be carried out in order to appreciate fully how the effects of different rules interacted.

For example, rules were written and applied to disambiguate words such as *before* and *after*, which may be tagged as subordinating conjunctions or prepositions, depending on their syntactic role in the sentence. Compare:

```
We met again after_CJS the ball was over.
We met again after_PRP the ball.
```

With the Template Tagger it was possible to formulate a rule that looked to the right to see if there was a finite verb within the clause, and if so, tagged the word as a subordinating conjunction. If there was no finite verb before the end of the clause/sentence, then the word was tagged as a preposition. Other rules also ran in conjunction with this, to correct special cases which would not be captured by the main rule, such as *before* occurring at the beginning of a sentence before a specific date like *1965*. It became apparent that it was preferable to run the rules that disambiguated finite verbs from non-finite verbs and nouns before the rules for *before* and *after*, so that the latter rules could properly identify finite verbs in the context.

Without a thorough syntactic parse, it was impossible to correct all errors, but the Template Tagger was crucial in the BNCTE project (see Chapter 9) for improving the accuracy rate of the automatic tagging in areas where the probabilistic formalisms of CLAWS and the restricted power of contextual pattern-matching rules had not been able to make an impression before.

8.5 Conclusion and Further Development

It will be apparent from the foregoing account that we are still some way from the completion of a truly generic template tagging program, but it is encouraging that the three types of deployment discussed in the previous section have all been possible within the framework of the development of a single piece of software.

It remains to be seen whether eventually the general-purpose nature of our tool will be so well developed that it will be possible to bring it to bear on novel tasks without the need for modification. That may be hoping for too much, but what is clear is that as the Template Tagger matures it will become an increasingly useful analytical and experimental tool. There has already been mention in this chapter of features which are still at the planning stage or under development. As our experience in using the tool grows, so some of those features may be subtly re-defined or supplanted by more pressing concerns, but the following areas seem likely to receive attention:

- **Validation procedures** To date error checking is confined to confirmation that input and rules are well formed, and that there is no reference to spurious levels. With other tools it has been customary to check the content of information levels, e.g. to check for an illegal POS tag. Therefore some means of specifying legal and illegal content at the user-defined levels would be appropriate.
- **Conversion tools** As well as coping with corpus encoding formats, we need to take account of rule file formats that currently exist. For example, in order to save recoding the thousands of rules in the IDIOMTAG module of CLAWS we have automated their translation to the Template Tagger format. This will also aid the acceptance of the Template Tagger if it is to replace our current tools.

What this tool exemplifies is an approach based on the idea that useful labelling of text can be based on the treatment of significant fragments of text, sequences of items which may be specified in templates, without respect to the 'well-formedness' of the broader context. Such approaches promote robustness, as they are more tolerant of 'real language', though their analyses may be less 'neat' than those achieved by more traditional 'structural' parsers. Robust analysers now seem to fall into two distinct types: the probabilistic tagger, of which CLAWS remains an example, and the template based 'fragment' analyser, of which the work on Constraint Grammar (see Karlsson *et al.* 1995) is perhaps the most thoroughly worked out instance to date. The Template Tagger is in the same tradition, though less theoretically oriented, intended for deployment on a range of tasks to be determined by the user, and ultimately as a tool with which to develop new analytical methods.

Notes

1. Lancaster Database of Linguistic Corpora (an ESRC-funded project at Lancaster University, 1990–95). This project involved the creation of a half-million word corpus, drawn from the texts contained in the British National Corpus, annotated to include enriched POS-tagging, grammatical functional labels (Subject, Object, etc.), lexeme identification, principal dependencies, and some word-level semantic information (ESRC Project No. x205262001).
2. The final two digits in each tag turn the tag DD2 into what are called 'ditto tags': see the discussion of multiwords in Section 2.2 (1).
3. This rule states that given the sequence: any form of *have*; any adverb; any

past participle; the word *by*, then the past participle is an active 'perfect tense' participle.

4. A **wild card** (the term is borrowed from the card game Canasta) is a symbol whose function is to stand for **any** value from a range of possible values. Thus, in this case, the asterisk is a wild card symbol which can stand for, or match, any string of characters (including zero characters or one character) excluding a space. Wild cards are extremely useful devices for automated text annotation, in that they allow the use of a **partial** specification, which can match on an open-ended set of **full** specifications.
5. A further distinction which this framework allows is between lemma (or lexeme) and spelling. The use of uppercase within curly brackets would allow a match against any part of a lemma, rather than the exact form cited.
6. This work took place within the British National Corpus Tagging Enhancement project (BNCTE) at Lancaster University, funded by the EPSRC: see Chapter 9 of this volume.
7. The sentence or 'unit' constraint is currently imposed by the rule-matching engine. The input routines can actually be set to handle a sliding window of several sentences using a device known as the 'wheel' devised by M. E. Bryant, formerly of UCREL. The ability to apply rule-matching routines across sentence boundaries would open up the possibility of using the Template Tagger to experiment with rules for anaphoric linking, for example.
8. The CLAWS NULL tag is used to tag apparent words (normally preceded and followed by a space) which are not words in a linguistic sense, such as SGML tags..
9. See the multiple passes through contextual rules in the CLAWS tagger, Section 7.5 above.