6 Implementing a tagger for Urdu

In this chapter, I will describe the work undertaken to design, develop and test a tagger for Urdu based on the tagset outlined in chapters 3 and 4 and the rule-based tagging methodology selected in chapter 5. My aim in this chapter is to demonstrate how the optimal set-up for a rule-based Urdu tagger was arrived at. A secondary aim is to evaluate what, if anything, can be learned about the linguistic nature of Urdu based on the process of designing the tagger in itself[1], as will be discussed in section 6.4.4.

As pointed out by van Halteren and Voutilainen (1999: 109) in the section quoted at length at the outset of the previous chapter, the task of tagging may be divided into three tasks. These are tokenisation – where the tokens of the text are segmented from one another as a preparatory step for analysis – followed by analysis of each token in isolation and the assignment of one or more potential tags to it, and finally disambiguation based on context to determine which tag is appropriate for that token. Although a large part of the discussion in this chapter will be devoted to the process of rule-based disambiguation (6.2.4, 6.4), for reasons outlined in the previous chapter, I will also discuss the programs that tokenise and analyse the text (see 6.2.2 and 6.2.3 respectively). Prior to that, I will discuss the computational structure into which these programs fit (see section 6.2.1). However, before moving on to these central issues, I will discuss a necessary preliminary factor: the difficulty of measuring the success of a tagger or any component of it.

[1] It goes without saying that the *output* of the tagger – the tagged texts and corpora – will be of use in investigating the linguistic structure of Urdu. However, it lies beyond the scope of this thesis to consider applications for the tagged text in any depth.

6.1 Measuring performance in a tagger experiment

As outlined in section 5.6.1, there is no universal agreement on an appropriate standard measure of tagger performance, making it very difficult to compare the results reported by different studies of part-of-speech tagging. However, within a single study the various factors which make comparison problematic can be controlled for. Furthermore it is clearly necessary for some measure of performance to be used to evaluate the tagger or components of it. In this section I will outline the measure used in this chapter.

The most developed model of rule-based tagging, Constraint Grammar (see section 5.2.2), uses a dual measure of precision and recall. It might therefore be supposed that for a tagger such as the one discussed here, utilising rule-based disambiguation, the precision/recall measure would be preferable. However, I intend instead to use the measures of accuracy (or "correctness") and ambiguity. This is for two reasons. Firstly, I do not consider precision to be as perspicuous a measure as ambiguity in terms of conveying at a glance how many extraneous analyses have been preserved[2]. Secondly, accuracy would appear to be more commonly in the majority of the studies reviewed in the previous chapter used than precision/recall. Therefore, for the highly problematic comparison between this study and others to be as legitimate as possible given the circumstances, the use of accuracy (and ambiguity where appropriate) is preferable.

The definitions of accuracy and ambiguity which I will utilise in this chapter

[2] In commenting that correctness/ambiguity are easier to understand than precision/recall, van Halteren (1999b: 81) essentially concurs with my impression here.

are effectively the same as those of van Halteren (1999b: 82)[3]. I will consider *accuracy* as being equal to the number of tokens that have received the correct tag, divided by the total number of tokens. This will be expressed as a percentage to one decimal place, in accordance with established tradition. I will consider *ambiguity* as being equal to the total number of tags that have been received by all tokens, divided by the number of tokens – that is, ambiguity equals the mean number of tags per token. This will be expressed to two decimal places. Some toy examples in English will suffice to demonstrate these principles:

- The_DET cat_NOUN sat_VERB on_PREP the_DET mat_NOUN

  Accuracy 100%, ambiguity 1
- The_DET cat_**ADJ** sat_**NOUN** on_PREP the_DET mat_NOUN

  Accuracy 66.7%, ambiguity 1
- The_DET cat_**ADJ/NOUN** sat_**NOUN/VERB** on_PREP the_DET mat_NOUN

  Accuracy 100%, ambiguity 1.33
- The_DET cat_**ADJ/ADV** sat_**NOUN/ADJ** on_**PREP/VERB** the_DET mat_NOUN

  Accuracy 66.7%, ambiguity 1.5

A related issue is the question of what level of performance is to be seen as a "successful" result to a tagger experiment? This is not unproblematic. As can be seen from the above toy examples, the ideal accuracy is 100% and the ideal ambiguity is 1. However, the success rates reported for the taggers discussed in the previous chapters

[3] Note however that van Halteren is among those who prefers the term "correctness" to "accuracy".

suggest that this is unattainable in practice. As outlined in section 5.7.1, accuracy rates of around 97%, or greater if some ambiguity remains, have been attained using most of the various disambiguation methodologies. However, Church and Mercer (1993: 9) suggest that even if no disambiguation technique at all were used, an accuracy of 90% could be achieved simply by selecting the most common tag for each token. This measurement appears to have been made for English, and there is of course no guarantee that the same will apply to Urdu.

Another problem lies with the trade-off between accuracy and ambiguity. Many strategies which remove extraneous tags may also at some point remove accurate tags which ought to be preserved. Thus as ambiguity falls, so too will accuracy. Conversely, a "better safe than sorry" strategy of preserving any tag which might conceivably be correct will lead to very high accuracy, but also very high ambiguity. It is not clear exactly what level of ambiguity would be great enough to debase the corresponding achievement in accuracy. Likewise, it is not clear what loss of accuracy would be too high a price to pay for unambiguous output. Is 98.3% with ambiguity 1.41 better or worse than 95.5% with ambiguity 1.19? Conceivably, the answer will depend on the intended purpose of the tagger's output.

Given the intractable nature of these difficulties, I will not attempt to quantify answers to the questions posed above. Instead, I will adopt an essentially *ad hoc* approach to such trade-off problems, justifying each decision as it is taken.

Where accuracy and ambiguity figures are quoted in the discussion in the remainder of this chapter, they have been assessed by comparison with the benchmark version of the training dataset created by manual tagging (see 4.5). The comparison

has been done automatically using a program called *Comparetag*[4]. This program loads each line[5] of each file being compared, together with the corresponding line from the benchmark corpus, and adds any errors or ambiguities found to a running total. When this process is complete, the final accuracy and ambiguity figures are calculated and printed to a report file. This also contains copies of lines where errors were found, to facilitate analysis of what is being tagged incorrectly. The program is dependent on identical tokenisation between the two files, so this must be ensured (by hand where necessary) prior to running Comparetag.

6.2 A description of the tagger system

6.2.1 General system philosophy and architecture

Prior to describing the algorithms underlying the various components of the Urdu tagging system, I will describe here the general philosophy of the system before going on to detail the structure of the system.

The system is named *Unitag*. This name has been chosen because the tagger

[4] This program, like all those written for this project and described in this chapter, was written in the C programming language using the Microsoft Visual C++ programming environment and compiler. Virtually all code written by myself conforms to ANSI standard C. The programs were written to run on a desktop PC (Pentium processor with 128Mb of RAM) operating under Microsoft Windows NT. User interface with the programs is in most cases via command-line arguments. Source code for all the programs is freely available and has been distributed with the EMILLE Corpus.

[5] See 6.2.1.3 below for a description of the vertical layout of tagging files.

functions solely and entirely on two-byte Unicode text[6]. So far as I am aware, it is the only tagger yet designed for which this is the case. Unitag is conceived, not as a tagger *per se*, but as a program for managing calls to other programs, each of which handles a stage of the tagging process.

The rationale for this is that, as stated above, it is common to divide tagging into the stage of tokenisation, initial tag analysis, and tag disambiguation. This computational task is more tractable to the programmer if handled by several different programs, each called in turn by Unitag[7]. However, handling each of these tasks by means of a separate program also allows them to be used independently where this is of benefit. For example, Verticalise, the tokeniser described in section 6.2.2 below was also used to prepare the raw text to be manually tagged (see also Chapter 4).

A further benefit to realising each stage of the task as an independent program is that it opens the possibility of reusing component programs. For example, tokenisation is a fairly low-level technique. Once one suitably well-performing tokenisation program has been written, there should be no reason for anyone with the opportunity to use that program[8] ever to write another. Thus, anyone wishing to perform tagging on Unicode text could make use of Unitag and Verticalise, and perhaps write their own programs for the more complicated procedures of analysis and disambiguation. This would save time otherwise spent, in effect, reinventing the wheel. It should be noted that, although it was written to form the basis of the Urdu

[6] Files used by Unitag containing filenames or program names, however, are required to be in ASCII, as ASCII text is needed for the system call.

[7] This was accomplished simply by using the C library function *system( )* to issue instructions to the operating system.

[8] In this context it should be noted that all the programs described in this chapter are freely available to all on request.

tagging system, Unitag itself is entirely language-independent. It consists of a functional typology of disambiguation systems (for not all perform the same task), a structure into which the programs fit, a formalism for these programs to communicate with one another, and a system for declaring which programs are to be used at each stage. In the following sections, each of these aspects of Unitag will be described.

6.2.1.1 Classification of disambiguation systems within Unitag

Although all taggers have some component which can be described as performing disambiguation, not all work in exactly the same way, though their effect on the text is equivalent. All disambiguation systems receive text which is tagged ambiguously – that is, in which any given token is tagged with one or more tags, of which one is correct. They all output text in which as many as possible of the incorrect tags have been removed. However, they differ in how they do this.

Some, for example Constraint Grammar taggers, the early TAGGIT system, or de Marcken's (1990) probabilistic system, remove incorrect analyses without necessarily selecting a single tag per token. This does not remove all the ambiguity; it is commonly referred to as n-best tagging (see also section 5.2.1 in the previous chapter).

Others always select a single tag from the available options. Most Markov model taggers operate in this way. Yet a third type do not narrow down ambiguous input at all; instead they take unambiguous input and reduce the number of errors that it contains. Of this type are Brill's transformation-based tagger, and the IDIOMTAG unit within the CLAWS system, for example.

These three types are not incompatible. For example, Brill's tagger can take as

its input the output of another tagger (Brill 1995: 545); and as Chanod and Tapanainen (1995a) demonstrate, different types of disambiguation system can be linked in serial.

It would therefore seem desirable to allow Unitag to call any of these types of program. To clarify the following discussion, I will define these three types of disambiguation system as follows:

- A *disambiguator* receives ambiguous input and removes some of the ambiguity, without necessarily selecting a single tag.
- A *decider* receives ambiguous input and selects a single tag per token.
- An *improver* receives unambiguously tagged input and alters the tagging in some way to reduce the number of errors.

The order in which these three types of system are listed is the order in which they must operate. Any other order would make no sense. An improver cannot function unless a decider has already removed all ambiguity; a disambiguator cannot operate after a decider as it requires ambiguity in the input. So that each module in the tagging system may be referred to unambiguously, I will define the term *analyser* to refer to the program which provides the initial ambiguous set of tags to the tokenised input.

On the basis of the components now defined, and their necessary relative ordering, the structure of the Unitag system may now be defined.

6.2.1.2 The structure of Unitag

The overall structure of the Unitag system can be represented diagrammatically as below:

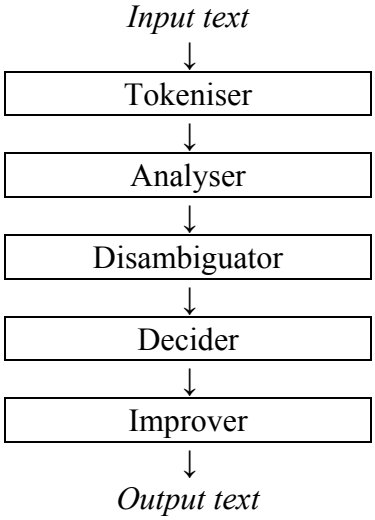

Fig. 6.1

An additional stage of reallocating probabilities will be discussed below.

Whilst the tokeniser and analyser are compulsory in the Unitag system, all the other units are optional (with the restriction that an improver *must* be preceded by a decider). Therefore, minimally, one could define an instantiation of Unitag which produced non-disambiguated text straight from the analyser. Of course, this would not be of particularly great use. Maximally, one could define an instantiation of Unitag which called, for instance, a Constraint Grammar-type disambiguator, a Markov model decider and a transformation-based improver, thus utilising the benefits of every approach. In the instantiation of Unitag which constitutes the tagger for Urdu, only a disambiguator is used, because, as has been explained, a rule-based procedure is to be used. However, it is my belief that the facility of future work on tagging in Urdu will be greatly enhanced by creating the Urdu tagger in a framework which is

ultimately very open to having new components added to it.

There follows a toy example in English of what might be the output at each stage, if a disambiguator, decider and improver were all specified in an instantiation of Unitag.

Tokeniser	The	cat	sat	on	a	bright	green	mat
Analyser	The art	cat N V R J	sat N V R J	on P	a art	bright N V R J	green N V R J	mat N V R J
Disambiguator	The art	cat N J	sat N V R J	on P	a art	bright R J	green N V R J	mat N V R J
Decider	The art	cat N	sat V	on P	a art	bright J	green N	mat N
Improver	The art	cat N	sat V	on P	a art	bright J	green J	mat N

In this example, the analyser (perhaps by reference to a lexicon) recognises the non-lexical words and assigns them the correct category, but assigns all the lexical words a set of open category tags. The disambiguator (perhaps using a rule that says verbs and adverbs do not follow articles) is able to reduce the ambiguity for the words "cat" and "bright". The decider, its task thus simplified somewhat, is then able to select using a Markov model what it considers the most likely string of tags for the sentence. It is not quite right, however, and one of the remaining two errors is picked up by the improver and corrected.

It should be noted that if more than one of the disambiguator, decider and improver are utilised within a particular instantiation of Unitag, that does not imply

that they shoulder an equal burden of the work. For example, if one wished to define a tagger like Brill's within Unitag, one could call a fairly naïve decider, and most of the actual work would be done by the improver (thus approximating Brill's process of an initial-state annotator producing inaccurate but unambiguously tagged text which is then corrected using learned rules: see Brill 1995).

There are several rules which apply to programs to be called by Unitag:

1. *The programs must be callable from the command line.*

   At present, Unitag and all other programs described in this chapter are called from MS-DOS. There is however no reason why they should not be recompiled for, say, Unix, and linked via a Unix shell script. However, command-line arguments are indispensable, as they are at the core of how Unitag communicates with the programs it calls. Windows programs, for instance, cannot be called in this way.
2. *They must handle two-byte Unicode text.*

   This is the very *raison d'être* of Unitag.
3. *The first command-line argument after the program name must be the filename of the text file to be processed. The second must be the filename in which the output is to be placed.*

   This is so that Unitag is able to construct the correct system calls.
4. *The programs must handle one file at a time.*

   While Unitag itself is capable of running in batch mode on a set of files specified in another text file, it does this on one file at a time. Therefore, the programs called must also be capable of handling a single file, specified as a command-line argument.
5. *The programs must handle text which is laid out in the Unitag file format, as*

described in the following section.

Again, this is to ensure effective communication between the different programs called by Unitag.

Whilst the first two of these restrictions are evidently necessary, it might seem that the other three might needlessly exclude from incorporation in Unitag programs which have already been written that do not conform with their requirements. However, this is not the case. A program called by Unitag may, naturally, itself call another program. So if, for instance, one wishes to incorporate into Unitag a disambiguator which uses a non-Unitag file format, and which takes the name of the file to be tagged as its third argument, this would not be impossible. A program would have to be written to call the disambiguator, having reordered the arguments and mapped the file to be tagged into the disambiguator's preferred input format. It would also have to map the format back to Unitag format after the disambiguator had run on it. This would be computationally trivial. The instantiation of Unitag would then be defined so that it calls this new, intermediary program. In this way, almost any program that performs the function of analyser, disambiguator, decider or improver can be included within an instantiation of Unitag.

It would be possible to go further with this, and incorporate programs that do not even use Unicode, if the intermediary program maps the file to some suitable eight-bit format and back again. It is also be possible to make the Unitag model more flexible. For example, one might wish to use more than one disambiguator – quite legitimately, as there would be ambiguity remaining after the first one for the second

one to reduce[9]. In this case, an intermediary program would be needed that ran first one, then the other, on each file. This program would then be called by Unitag, which would treat the intermediary exactly as a single-program disambiguator.

6.2.1.3 The Unitag file format

As stated earlier, there is a set formalism used within Unitag for the various programs to communicate with one another. This is the Unitag file format, within which information about each token is stored in a particular layout, so that each program knows exactly what input it is receiving from the previous program in the chain. I will in this section describe and (where necessary) justify the format.

It is anticipated that the text to be tagged will be either plain text without any markup or text marked up in SGML/XML. The EMILLE Corpus[10] is in the latter format, as are most text corpora nowadays. Therefore, it will for now be assumed that this is the case without further comment. Should it be desired to employ Unitag on text in some format radically different to plain text or SGML, it might prove necessary to pre-process the text into a format suitable for Unitag. The role of the tokeniser in Unitag (see the next section) is to transform the SGML into the Unitag format, which as a matter of definition includes tokenisation.

The layout of a file in Unitag format is vertical, i.e. each token is laid out on a

[9] It would also be legitimate to use more than one improver. However, the nature of the decider is such that using more than one runs contrary to its definition.

[10] See section 1.3.

separate line[11]. Whilst the exact content of each line varies slightly depending on what point in the tagging process the file has reached, some things are consistent throughout. The following is an example of line from a file in the Unitag format:

s00004 w001 <body> *VE NULL

At the start of the line are two serial numbers which identify the token[12]. The first is a segment number. A segment consists of a string of less than 1,000 tokens, divided by the tokeniser in any way that the programmer deems appropriate[13]. The tokens within a segment are indicated by the word number, which is the second number on the line. Both numbers are followed by a space character (Unicode 0020). This two-number system has been adopted to facilitate finding particular lines in the file with ease.

After the second space character, there follows the token itself (in the example above, an SGML element). Then, there is a horizontal tab character (Unicode 0009). The following three characters are a code for "last modified by". In this case, *VE indicates that Verticalise has been the last program to modify this line. The code need

[11] This is very similar to one of the output formats produced by the CLAWS tagger, which I find easy to work with. Since this format is easily mapped to some other format at any point, it seemed acceptable to follow my personal preference on this point.

[12] This identification is not necessarily unique. The numbers are assigned by the tokeniser, so if the tokenisation is corrected by some later module (e.g. the analyser) one or more adjacent tokens may have the same number. As the numbers are intended for human reference primarily, rather than machine processing, this is not seen as a problematic issue.

[13] In text processed by Verticalise (see section 6.2.2), a new segment begins with the token after each SGML element. Since in the EMILLE corpus, each utterance or sentence begins and ends with an SGML element, this seemed a reasonable way of breaking up files into manageable segments.

not only indicate the program, but may also indicate which procedure within the program which has last altered the line. In post-editing, if a human changes a line, they should insert their initials here. Note that these responsibility markers must be exactly three characters in length.

After the responsibility marker is another space character. After that follows the tag or tags which are currently assigned to that token. In this case, since the "token" is actually an SGML tag, it receives the special NULL tag, indicating that this is not a unit which can meaningfully be tagged. There then follows a carriage return (Unicode characters 000d, 000a) before the next line.

The first line in the file is preceded by the Unicode byte-order mark, Unicode FEFF, as is standard for many Unicode files. Other than that, all lines follow the format detailed above, with the following variations in what occurs in the part of the line where the tag is given.

Prior to the analyser running on the file, it is possible for there to be nothing at all here (although there is still a space after the responsibility marker)[14]:

s00010 w001 چاہیں	*VE

After the analyser has run, up until the point at which the decider runs, there may be several tags:

s00010 w001 چاہیں	*VE VVSV2 VC2

The tags must consist of characters other than control characters, white space,

[14] I am now using Urdu examples from files in the benchmark corpus, in some cases manipulated slightly to illustrate a point.

the underscore character (Unicode 005F) or the forward slash character (Unicode 002F). They need not be from the ASCII range – any Unicode characters may be used. Between each tag is a space. This might seem to preclude the use of any tagset where tags which are not single strings of letters and numbers are utilised, such as the tagsets used in Constraint Grammar tagging (see 5.2.2). However, it would again be computationally simple to map such a tagset to one which is acceptable in the Unitag file format, and back again, as necessary.

After the decider has run, there are two possibilities for a line. It may contain a single tag, in which case that is the tag allocated to that token. Alternatively it may be a string of tags in which the first one, preceded by an underscore character, is the tag allocated to that token and the remaining tags represent rejected potential tags. The reason that the possibility of retaining the rejected tags on the line is allowed for in the Unitag file format is that it is possible to conceive of a situation where either an improver or some manual post-editing process makes use of the information in the tags rejected by the decider[15]. However, as no instantiation of Unitag discussed in this chapter uses this feature, I will not discuss it further. The two possible formats are as follows (the responsibility marker is that of a hypothetical decider program):

s00010 w001 چاہیں	*DE VVSV2
s00010 w001 چاہیں	*DE _VVSV2 VC2

Given that Unitag is intended to allow the usage of both rule-based and probabilistic systems within a single framework, it is necessary for some information

[15] This feature, like the layout in general, was inspired by the CLAWS vertical layout, although I am unaware whether the CLAWS layout was motivated by the same considerations that influenced my decision here.

about the relative probabilities of the different tags to be communicable between the different programs called by Unitag. This would normally take the form of tag frequencies, for instance if the analyser uses information in a lexicon to give the relative frequencies of the different tags it has suggested. However, there are other possibilities: for example, if a Markov model disambiguator marks on the different tags what it judges to be their probabilities in context. In either case, the probability of a tag is expressed as a percentage to no decimal places, which is laid out following the tag and separated from it by a forward slash character, as illustrated below:

s00010 w001 چاہیں *LE VVSV2/50 VC2/50

These probabilities are optional at all levels: a tagged file need not contain them, and if a file does then not every line in the file nor every tag on the line need have them. This stipulation is necessary because, of course, some analysers will not provide these probabilities, for example analysers designed to work with rule-based disambiguators.

However, so that any module which *is* probabilistic in nature may handle the file format robustly it is a condition of the format that where tags have no probability given, they are assumed to have an equal share of the probability remaining on that line. So for example, if two tags are given on the line and neither has a stated probability, then both are assumed to have a probability of 50%. If two tags are given, one with a stated probability of 75% and the other without stated probability, then the latter is assumed to have a probability of 25%. If four tags are given, one stated as 20% and one as 25%, then the remaining two are assumed each to have a probability of 27.5%.

It should be noted that the possibility of combining probabilistic and non-

probabilistic modules introduces a potential problem. If a non-probabilistic disambiguator eliminates some superfluous tags, the probabilities on a line will no longer total 100. This might pose problems for a subsequent Markov model decider or disambiguator. In such a case, the instantiation of Unitag might require an intermediary program to be used which called a program to adjust the probabilities so that they total 100 on each line again.

Although the Unitag file format is primarily for use internally within the program, I have used the same format for manual tagging, and the final output is left in that format. I have written a program, *Deverticalise*, which converts the Unitag format to a more widely-used SGML format (each line being condensed as follows: <w pos="TAG">TOKEN</w>), but this is not included in the Unitag architecture so that post-editing may be undertaken on the vertically formatted text.

6.2.1.4 Defining an instantiation of Unitag

I have at numerous points used the notion of an "instantiation" of Unitag. An instantiation of Unitag is defined simply as a specified set of programs working in the appropriate positions within the Unitag structure. The instantiation is specified in an ASCII text file, which is read by Unitag and used to construct its system calls. It contains the names of each of the programs, plus information about any additional arguments required by calls to those programs subsequent to the filename of the file to be tagged.

A Unitag instantiation file has the following form:

- On a separate line each, the analyser, disambiguator, decider and improver are

specified.

- Each specification consists of a single word saying what module is being specified, followed by a space, followed by the program name, followed by a line break.
- On the line after a program is specified are typed any command-line arguments which should follow the filename of the output file in the system call to that program, exactly as they would be entered at command line. If no arguments are specified, that line should contain the string "NULL". Therefore, the file as a whole consists of eight lines.
- If no program is to be specified for a given module, then instead of a program name the file should contain the string "NULL".
- Lines beginning with a forward slash are ignored as comments.

The following would therefore be a possible instantiation of Unitag for tagging Urdu:

```
/ Unitag for Urdu
/ Version testing new Urdu lexicon
analyser urdutag
new_urdu_lexicon.txt
disambiguator unirule
urdu_rulefile.txt
/ no decider used for now
decider NULL
NULL
improver NULL
NULL
```

No option is included to specify a tokeniser, as Verticalise (outlined in the following section) has been designed specifically to produce the Unitag format and

perform tokenisation at the same time. Note that Unitag will only call an improver if a decider has been specified.

The Unitag program itself is called by the following instruction on the command line: *unitag instantiation_filename raw_text_filename* . The first argument after the program name is the filename of the text file containing the instantiation details. The second argument is the name of the text file to be tagged. If an optional third argument consisting of a single L is present, then the second argument will be interpreted as an ASCII file containing a list of multiple files to be tagged all at once, with one filename on each line.

6.2.2 The design of the tokeniser program

Like the general Unitag program, the tokeniser program (called *Verticalise*) is language-independent and does not incorporate any particular linguistic knowledge. It takes in untokenised text and performs the following actions:

- Any character or stretch of characters consisting solely of white space characters is replaced by a token break (unless it occurs within an SGML element).
- A token break is inserted before and after every punctuation mark.
- A token break is inserted before and after every SGML element (which is treated as a single token).
- The text is laid out in the vertical format described in 6.2.1.3 above.
- All SGML elements are given the NULL tag.

As can be seen, Verticalise does not consider what the words actually are in

any way. Such fine-tuning to the tokenisation as splitting clitics off from the word they are attached to is left to the analyser to perform using the language-specific knowledge which it must necessarily possess.

6.2.3 An analyser program for Urdu

In the following section, I outline the structure of the analyser program written for the Urdu tagger, which assigns the initial set of contextually ambiguous tags to the tokens of the text. Unlike the Unitag and Verticalise programs, this program is language-specific; for this reason it is called *Urdutag*. It uses several means of analysis, including lexical lookup, character type analysis, and morphological analysis. These will be discussed in 6.2.3.1, 6.2.3.2 and 6.2.3.3 respectively.

6.2.3.1 Lexical lookup in Urdutag

Lexical lookup in Urdutag is fairly basic. A lexicon is specified as one of the program's command-line arguments and is held in memory by the analyser. The first step in the analysis of a given token is to look its wordform up on the lexicon list[16]. If it is found, then all the tags that the word in the lexicon possesses are assigned to the token in the text. Neither the character types nor the morphology of the word will be analysed if a wordform is found in the lexicon.

[16] Vowel diacritics are stripped from the word prior to the process of analysis, so a single lexicon can be used for both text without vowel marks and the much rarer vowelled texts.

No attempt is made to lemmatise[17] tokens. Although this might lead to a greater success rate in looking things up in the lexicon, there was a significant risk that I would bias the model away from the patterns of language as it really is. My model of the language, Schmidt's (1999) grammar, is not comprehensive enough to form the basis of a robust lemmatisation algorithm. Therefore, I would have to make judgements about the possible occurrence of, and predict the forms of, words that are not discussed or cited by Schmidt or by any other writer. As a non-native speaker I felt it imprudent to attempt this. It may be theorised that a larger lexicon may help offset the disadvantages of missing lemmatisation.

The format of the lexicon and the procedures used to optimise it will be discussed in 6.3.1 below, where I introduce the Unilex program (a companion tool to the Unitag suite) which was used to automatically derive and manage the lexicons described in this chapter.

6.2.3.2 Character type analysis in Urdutag

If no analysis is produced by lexical analysis, Urdutag attempts to allocate a tag by analysing the characters that make up the token. This algorithm is currently very primitive, detecting only the JDNU and FX categories:

- If all the characters in the token are numerals (Arabic or ASCII), the tag JDNU is assigned.
- If the token contains characters from outside the Arabic alphabet, the tag FX is

[17] To lemmatise a token is to remove any inflectional morphemes and derive the "root" or "basic" form of the word.

assigned.

If after both lexicon lookup and character type analysis, no tag has been assigned, the word is morphologically analysed. It is in this area, discussed in the following section, that the most knowledge about the language has been built in to the Urdutag program.

6.2.3.3 Morphological analysis in Urdutag

As with the tagset, the Urdu grammar of Schmidt (1999) was used as a model of the language to develop an algorithm for morphological analysis of Urdu tokens, which was implemented in the code of Urdutag.

The inflectional morphology of Urdu is in some ways simple. There are no inflectional prefixes, only suffixes, and each word has at most a single inflectional suffix. Derivational affixes (which can also help to identify the category of a word) are also primarily suffixes. This makes the action of the morphological analyser computationally very simple[18]: it reads characters from the word, one by one, starting from the end, and matches the longest suffix it can. When it has matched a suffix, it assigns a set of tags associated with that suffix (or a default set of tags if no suffix is

[18] One slightly more complex aspect of the analysis, computationally speaking, is identifying clitics and splitting the token which contains them so clitic and host word can be tagged separately. However, despite its computational complexity, it is linguistically simple (since it merely involves spotting the relevant string at the start or end of a word) and I will not discuss it further here except to note that the process of spotting and splitting off clitics comes before suffix analysis. The clitics identified are *al–*, *-gunā*, and the various clitic forms of *hī* and *kō*, the last two being homographs in some contexts (see 3.5 and 3.12.1).

matched). These sets can be quite large, as there is a great deal of ambiguity among suffixes in use in Urdu. This was discussed to some extent in section 3.1.5, with regard to noun suffixes. But the problem is in evidence across most morphosyntactic categories. This means that inevitably, the ambiguity inherent in the wordform is reproduced in the output of the analyser.

To illustrate this, I will list all the suffixes mentioned by Schmidt (1999) which could be of use in allocating a tag. I deviate here from my usual system of transliterating Urdu, in order to represent more clearly the actual characters that occur in the Indo-Perso-Arabic text. So the Unicode character 06CC, *chōTī yē*, ی, which has been transcribed variously as *y*, *ī*, and (medially) *ē* and *ai*[19], is here shown as Y, regardless of its phonetic value; the character *vāō*, و, transcribed as *v*, *ū*, *ō*, or *au*, is here shown as V; *baRī yē*, ہ, is indicated by E, and *chōTī yē* with superscript *hamza*, ئ, usually left untranscribed, is indicated by [hoy] (for "hamza over yē"). I have also left out the vowels which are not present in the original word.

The suffixes listed below fall into three distinct groups: those native to Urdu, those confined to Arabic loanwords, and those confined to Persian loanwords. The

[19] Some added explanation is warranted here. In theory, *baRī yē* represents the *ē* and *ai* sounds, and *chōTī yē* the *y* and *ī* sounds. However, the medial forms of the two letters are identical, which has led some Urdu typists to use *chōTī yē* when typing a letter representing any of the four sounds in medial position. This procedure was adapted as standard on the EMILLE project, following the example set by the BBC's Urdu webpage (one of very few in the world that use Unicode) and at the suggestion of one of the typists on the project. This means that for the purposes of the morphological analyser, *baRī yē* (E) will be assumed to always occur finally, since if it was medial, *chōTī yē* (Y) would be typed instead. However, it remains a possibility that the system may be used to tag text where *baRī yē* has been used medially, and therefore medial *baRī yē* is changed to *chōTī yē* at the same time as any vowel diacritics are removed.

first group are contained in the following table. The symbol (***) indicates that all words falling into this category were "exceptions" of one sort or another, either by virtue of being considered "rare" by Schmidt or by going against another, more general pattern of suffix-tag match-up[20]. These exceptions are not implemented in Urdutag, and must therefore be held in the lexicon, as discussed at greater length below.

Table 6.1

Suffix	Tag indicated
Noun suffixes	
ā	NNMM1N (NPMM1N[21])
h	NNMM1N
Yh	NNMM1N
ā~	NNMM1N (***)
Y	NNUM1N NNUM1O NNUM1V NNUM2N (as occupational suffix – ***)
gāh	NNUF1N NNUF1O NNUF1V
(Note also that the preceding derivational suffix may be anticipated to also take inflectional suffixes to create relevant forms for NNUF2N, NNUF2O, NNUF2V)	
ā[hoy]E	NNUF1N NNUF1O NNUF1V (Persian feminine ending – ***)

[20] The suffixes *–Y* and *–gY* are identified as a Persian feminine ending by Schmidt (1999); however, she does not state whether the nouns it appears on are marked or unmarked (one might anticipate the former from their form, but the latter from their provenance). In the former case, there is nothing to distinguish this ending from that of the standard marked feminine noun; but in the latter case, all words of this type will be included in the lexicon as exceptions to the standard marked feminine pattern.

[21] A single proper noun tag is given here for purposes of demonstration. In fact, since Urdutag was written to tag using the U2 tagset, this was not implemented in the program. See also 4.2.2.4.

Y	NNUF1N NNUF1O NNUF1V (***)
gY	NNUF1N NNUF1O NNUF1V (Persian feminine ending – ***)
stān	NNUM1N NNUM1O NNUM1V NNUM2N
(Note that this derivational suffix may be anticipated to also take inflectional suffixes to create relevant forms for NNUM2O and NNUM1V)	
V	NNUM1N NNUM1O NNUM1V NNUM2N (***)
pn	NNUM1N NNUM1O NNUM1V NNUM2N
(Note that this derivational suffix may be anticipated to also take inflectional suffixes to create relevant forms for NNUM2O and NNUM1V)	
pā	NNMM1N[22]
Y	NNMF1N NNMF1O NNMF1V
Yā	NNMF1N NNMF1O NNMF1V
āhT	NNMF1N NNMF1O NNMF1V
āVT	NNMF1N NNMF1O NNMF1V
Yt	NNMF1N NNMF1O NNMF1V
(Note also that the three preceding derivational suffixes may be anticipated to also take inflectional suffixes to create relevant forms for NNUF2N, NNUF2O, NNUF2V)	
E	NNMM1O NNMM1V NNMM2N
[hoy]E	NNMM1O NNMM1V NNMM2N (nouns such as *rūpae*)
Y~	NNMM1O NNMM1V NNMM2N (***)
Yā~	NNMF2N
YV~	NNMF2O
Y~	NNUF2N
[hoy]Y~	NNUF2N

[22] It is not entirely clear from Schmidt's description whether this suffix indicates marked or unmarked nouns. However, consultation with one of my native-speaker informants confirms that, as one would expect from its form, this suffix indicates marked nouns.

V~	NNMM2O NNUM2O NNUF2O
V	NNMM2V NNMF2V NNUM2V NNUF2V
Adjective suffixes	
ā	JJM1N
ā~	JJM1N (***)
Y	JJF1N JJF1O JJF2N JJF2O
Y~	JJF1N JJF1O JJF2N JJF2O JJM1O JJM2N JJM2O RRJ (***)
E	JJM1O JJM2N JJM2O RRJ
Y	JJU (as a derivational ending – ***)
ā or Y	JJU (some Arabic loanwords – ***)
Vā~	JDNM1N
VY~	JDNM1O JDNM2N JDNM2O JDNF1N JDNF1O JDNF2N JDNF2O
Adverb suffixes	
[Unicode character U+064B]	RR (Arabic loans ending in the “tanvīn” character)
Verb suffixes	
nā	VVNM1N
nE	VVNM1O, VVNM2
nY	VVNF1, VVNF2
tā	VVTM1N
tE	VVTM1O, VVTM2N, VVTM2O
tY	VVTF1N, VVTF1O, VVTF2N, VVTF2O
tY~	VVTF2N
ā	VVYM1N
Yā	VVYM1N

E	VVYM1O, VVYM2N, VVYM2O
Y	VVYF1N, VVYF1O, VVYF2N[23], VVYF2O
Y~	VVYF2N
V~	VVSM1
E	VVST1 VVSV1
Y~	VVSM2 VVSV2
V	VVST2 VVIT2
[hoy]E	VVIA
[hoy]YE	VVIA

Also native to Urdu are a collection of related suffixes (*ā / lā / vā / lvā*) which increase the valency of verbs, creating transitive and causative verbs from intransitive roots (see Schmidt 1999: 158-174). Occurring before the verbal inflectional suffixes, it might be possible to analyse these suffixes, and use them to identify the word definitively as a verb (as opposed to the tentative identification which might otherwise result from the similarities between the verbal suffixes and other suffixes). However, this has not been attempted, as internal change to the verb root frequently accompanies the addition of these derivational affixes; the suffixes may also occur as infixes (e.g. *bōlnā > bulānā; TūTnā > tōRnā; biknā > bēcnā; nikalnā > nikālnā*). To allow for all these changes would be computationally tricky. Rather, the derived transitive or causative verb will be treated as a lexical item in its own right (much like *sit* and *set* in English).

Under the heading of "Arabic suffixes" fall various forms of the Arabic feminine suffix, *-ā / -t / -h / -āh / -āt / -Yt*, the dual/plural ending *–Yn*, and an adjective ending *–ānY*. The Persian suffixes consist of the following group. All indicate either

[23] Schmidt (1999) does not specify whether or not the *–Y* ending may represent a feminine plural perfective participle, but one of my native-speaker informants reports that it can.

unmarked nouns or unmarked adjectives[24].

Y, gY, kār, gār, gr, cY, bān, Vān, gāh, ābād, stān, zār, ch, īch, k, dān, dānY, dār, āVr, Vr, Yār, Yr, mnd, Y, h, ānh, nāk, gYn, ān, gān, Yān

As can be seen, some endings (especially *Y*, *E*, *ā, Y~* and *V*) are highly polysemous. This problem only gets worse when Arabic and Persian suffixes are taken into account, since in many cases, the tendencies demonstrated by the borrowed affixes are opposite to those in the native vocabulary (e.g. *–ā* indicates masculine marked nouns in Indo-Aryan words, but feminine unmarked nouns in Arabic loanwords).

It is not entirely clear from the literature on Urdu grammar (see 2.3) to what extent Arabic and Persian loanwords are prevalent in the language. While there are clearly very many such words, their frequency in usage has not been quantified. One might well hypothesise that they are too infrequent to be concerned with. Therefore, it did not seem worthwhile complicating the analysis algorithm with the Arabic and Persian endings until it could be seen to what degree they were needed. That being the case, the initial form of the suffix analysis algorithm will contain all and only the categories listed above as native to Urdu, with those categories marked (***) as exceptions to be stored in the lexicon. The suffixes that the algorithm looks for (in reverse order, so that *–stān* is represented as "n-ā-t-s") and the Unicode strings that represent them are listed below. Urdutag always matches the longest suffix it can;

[24] In fact, some of the suffixes listed in the "native to Urdu" group are actually derived from either Persian or Arabic. However, in Schmidt's discussion, these suffixes are pointed out as being a particular help in identifying part-of-speech and/or noun gender, and so I retain them in this group whilst putting those that Schmidt discusses solely in the context of loanwords in another group.

therefore, if a word ends in *–pā*, Urdutag will assign the set of tags for "ā-p" (NNMM1N), *not* the set of tags for "ā" (NNMM1N, JJM1N, VVYM1N).

Table 6.2

Unicode string[25]	Suffix	Set of tags to allocate
U+0627, 062a	ā-t	VVTM1N
U+0627, 0646	ā-n	VVNM1N
U+0627, 067e	ā-p	NNMM1N
U+0627, 06cc	ā-Y	NNMF1N, NNMF1O, NNMF1V, VVYM1N
U+0627	ā	NNMM1N, JJM1N, VVYM1N
U+062a, 06cc	t-Y	NNUF1N, NNUF1O, NNUF1V
U+0646, 0627, 062a, 0633	n-ā-t-s	NNUM1N, NNUM1O, NNUM1V, NNUM2N
U+0646, 067e	n-p	NNUM1N, NNUM1O, NNUM1V, NNUM2N
U+0648, 062a, 06cc	V-t-Y	NNUF2V
U+0648, 0646, 0627, 062a, 0633	V-n-ā-t-s	NNUM2V
U+0648, 0646, 067e	V-n-p	NNUM2V
U+0648, 0679, 0648, 0627	V-T-V-ā	NNUF2V
U+0648, 0679, 06c1,	V-T-h-ā	NNUF2V

[25] It should be noted that, as I was writing in C code, it was necessary to enter Unicode strings as hexadecimal codes, character by character, in a similar form to the layout in this table. This was one factor which made programming this part of the analyser troublesome.

0627		
U+0648, 06c1, 0627, 06af	V-h-ā-g	NNUF2V
U+0648	V	NNMM2V, NNMF2V, NNUM2V, NNUF2V, VVST2, VVIT2
U+064b	[tanvīn]	RR
U+0679, 0648, 0627	T-V-ā	NNUF1N, NNUF1O, NNUF1V
U+0679, 06c1, 0627	T-h-ā	NNUF1N, NNUF1O, NNUF1V
U+06ba, 0627, 0648	~-ā-V	JDNM1N[26]
U+06ba, 0627, 06cc	~-ā-Y	NNMF2N
U+06ba, 0648, 0646, 0627, 062a, 0633	~-V-n-ā-t-s	NNUM2O
U+06ba, 0648, 0646, 067e	~-V-n-p	NNUM2O
U+06ba, 0648, 062a, 06cc	~-V-t-Y	NNUF2O
U+06ba, 0648, 0679, 0648, 0627	~-V-T-V-ā	NNUF2O
U+06ba, 0648, 0679, 06c1, 0627	~-V-T-h-ā	NNUF2O

[26] For this category (and the other ordinal number category, ~-Y-V) it would be possible to write a special algorithm to check whether what is left after the suffix is removed is a cardinal number, and to reject the numeral tags if this is not the case. However, since the *–vā~* suffix is non-ambiguous, it was decided to leave this until and unless it became obviously necessary at a later stage. Irregular ordinals (the majority between "first" and "twentieth": see Schmidt 1999: 229) are stored in the lexicon.

U+06ba, 0648, 06c1, 0627, 06af	~-V-h-ā-g	NNUF2O
U+06ba, 0648, 06cc	~-V-Y	NNMF2O
U+06ba, 0648	~-V	NNMM2O, NNUM2O, NNUF2O, VVSM1
U+06ba, 06cc, 0626	~-Y-[hoy]	NNUF2N
U+06ba, 06cc, 062a, 06cc	~-Y-t-Y	NNUF2N
U+06ba, 06cc, 062a	~-Y-t	VVTF2N
U+06ba, 06cc, 0648	~-Y-V	JDNM1O JDNM2N JDNM2O JDNF1N JDNF1O JDNF2N JDNF2O
U+06ba, 06cc, 0679, 0648, 0627	~-Y-T-V-ā	NNUF2N
U+06ba, 06cc, 0679, 06c1, 0627	~-Y-T-h-ā	NNUF2N
U+06ba, 06cc, 06c1, 0627, 06af	~-Y-h-ā-g	NNUF2N
U+06ba, 06cc	~-Y	NNUF2N , VVSM2, VVSV2, VVYF2N
U+06c1, 0627, 06af	h-ā-g	NNUF1N, NNUF1O, NNUF1V
U+06c1, 06cc	h-Y	NNMM1N
U+06c1	h	NNMM1N
U+06cc, 062a	Y-t	VVTF1N, VVTF1O, VVTF2N, VVTF2O
U+06cc, 0646	Y-n	VVNF1, VVNF2
U+06cc	Y	NNMF1N, NNMF1O, NNMF1V, JJF1N, JJF1O, JJF2N, JJF2O, VVYF1N, VVYF1O, VVYF2N, VVYF2O

U+06d2, 0626, 0627	E-[hoy]-ā	NNUF1N, NNUF1O, NNUF1V
U+06d2, 0626	E-[hoy]	VVIA, NNMM1O, NNMM1V, NNMM2N
U+06d2, 062a	E-t	VVTM1O, VVTM2N, VVTM2O
U+06d2, 0646	E-n	VVNM1O, VVNM2
U+06d2, 06cc, 0626	E-Y-[hoy]	VVIA
U+06d2	E	NNMM1O, NNMM1V, NNMM2N, JJM1O, JJM2N, JJM2O, VVYM1O, VVYM2N, VVYM2O, VVST1, VVSV1, RRJ

The default set of tags is as follows (tags for proper nouns, shown in brackets, are not implemented in Urdutag, but would be required in a system designed to tag the U1 or U0 tagsets).

JJU, NNUM1N, NNUM1O, NNUM1V, NNUM2N, NNUF1N, NNUF1O, NNUF1V, RR, VVIT1, VV0 (NPUM1N NPUM1O NPUM1V NPUM2N NPUF1N NPUF1O NPUF1V)

The codes used by Urdutag to indicate that it has changed the tagging of a token are as follows:

Table 6.3

Code	Process by which the tag was assigned
A10	Item found in lexicon
A30	Tags assigned by suffix analysis

A50	Tags assigned by character analysis
A70	Tags assigned by the part of algorithm which spots and separates proclitic *al–*.
A90	Default set of tags assigned to unknown word

It was originally hoped that the algorithm above would provide 100% tagging correctness, albeit with a high ambiguity rate, when used in conjunction with a suitable lexicon (one containing not only the classes of exceptions listed above but also any other "exceptional" words encountered whilst developing the tagger). This did prove to be possible, but only when using an automatic lexicon derived from the data that was being tagged (so that *all* the tokens could be found by lexical lookup, and the problem became one of disambiguation only). Using a more modest lexicon, of around 1,000 words of closed category words plus the most common words in the training data[27], accuracy rates tended to be in the region of 86 to 94%, depending on the text being tagged. This is rather poor.

I therefore analysed the errors made tagging the training data in this way. Almost 30% of these errors were due to FF (about 20%), FU, and FB in the training data – tags which Urdutag cannot assign, since it has no way of knowing what is a foreign word or not. As I noted in an earlier chapter, the manual tagger tended to overuse these tags. However, it is also probably due to the nature of the data (transcripts of UK-based Hindi-Urdu radio programmes) that foreign words are so common. It would therefore seem that unless these words are included in the lexicon, they cannot be tagged correctly. Another cause of errors was typographic errors and typographic variants in the data. Nothing can be done about this, either, other than

[27] See 4.5 for a discussion of the composition of the training dataset.

storing variants in the lexicon[28].

The other major cause of errors was that Urdutag was spotting suffixes where none existed – i.e. giving an inappropriate analysis to a word which appeared to end in an inflectional suffix, which was actually an unalterable part of the root. For example, many unmarked adjectives end in *–ī*; many words that are not imperfective participles end in *–tā*, *–tē* and *–tī*; and so on and so forth. Even the "default set" of tags for words with no suffix was sometimes generating errors, for instance with Persian- or Arabic-derived plural forms, which unlike normal unmarked plurals have no ending in the oblique. So مجاہدین, for example, could easily be NNUM2O – but this tag was not being assigned by default.

Given these problems, I tried two experiments. Firstly, I removed the suffix analysis part of the program altogether, so that anything not in the lexicon or tagged by character analysis got the default set of tags. Accuracy fell by about 0.5% - although it was getting different words wrong. For instance, JJU was almost always correctly tagged by the altered system, whereas before it had frequently been wrongly tagged. To test the value of smaller changes to the algorithm, I added all the unmarked plural noun tags to the default set, to try and handle the Arabic and Persian plurals. It increased accuracy by 0.2%, but also increased ambiguity by 0.36 tags per word. Given that a lexicon that contains more of the problematic words has a much more dramatic effect, the value of this modification is uncertain, and it has not been generally implemented.

[28] It might be possible to employ fuzzy matching algorithms in an attempt to identify variant spellings, as has been explored, for example, by Piao and McEnery (2003) with regard to Early Modern English. Due to the complexity of achieving this goal in Urdu, it lies beyond the scope of this thesis; however I note this as a possibility for future researchers to explore.

Some other small modifications were made to Urdutag, however. For instance, the ending "–[hoy]Y~" as a predictor for NNUF2N was frequently incorrect, so it was removed. Words ending in this string were given the default tags for "–Y~" instead (which include NNUF2N).

6.2.4 The design of the disambiguator program

In this section, I will discuss the Unirule program, the rule-based disambiguator written to function within Unitag. The development and optimisation of the set of rules used to disambiguate tagged Urdu text will be discussed in section 6.4. Here, however, I will restrict my comments to the general operation and formalism of Unirule (although drawing on examples from my experience with Urdu where appropriate).

A rule-based disambiguator requires some kind of formalism in which its rules can be expressed and stored. This formalism will necessarily restrict the type of rules that can be implemented by the system. The formalism devised for Unirule was based on a brief survey of best practice as exemplified by the rule formalisms of Constraint Grammar (see section 5.2.2) and Brill's transformation-based tagger (see section 5.4.2).

6.2.4.1 The formalisms of rules in Constraint Grammar and Brill's tagger

In Constraint Grammar (Karlsson 1995b), rules or "constraints" consist of a domain, a target, an operator, and context conditions. The domain specifies a word-

form to which the constraint applies[29]. The target consists of a set of features (tags or a lemma); any given analysis of a token is affected by the constraint if it has all those features. The operator indicates what operation will be performed on the analysis or analyses that match the target. Possible operations are: 1) discard all analyses except the one that matches the target if the context conditions are fulfilled, but otherwise discard the analysis that matches the target; 2) discard all analyses except the one that matches the target if the context conditions are fulfilled; 3) discard the analysis that matches the target if the context conditions are fulfilled.

The context conditions have the form <polarity, position, set>. Polarity expresses whether the constraint is positive or negative (i.e. NOT). Position defines the location of the word that the condition refers to, relative to the word that is matched by the target. It may be of the form "*x* words left/right", or "*x* or more words left/right"; the CG program can look up to 25 words each way. The set is a set name which must be declared by the user to refer to a set of feature combinations (i.e. a set of items within the tagset). If the word in position has an analysis that is within the set, then the condition is fulfilled. If one constraint has multiple conditions, then all must be satisfied for the operation to be carried out.

Conditions have an optional "careful mode" – when this is on, *all* analyses of a given token must be within the set for a condition referring to that token to be satisfied, thus allowing the writer of the constraint to stipulate whether ambiguously-tagged words can be used as context to disambiguate other words or not. Other features of the Constraint Grammar formalism include relative positions (i.e. when a condition has found something in position x, it looks a further y words along for some

[29] This may be left unspecified, in which case the constraint will apply to any token that matches the target.

other context condition to be fulfilled), a marker to prevent a constraint from operating across clause boundaries, and ways of defining phrases (for instance, noun phrases) so that any of a variety of types of such phrases will be accepted as fulfilling the condition. One final feature of the constraints in this system is a distinction between "safe" constraints (which never reduce the accuracy of the tagging, but only remove incorrect analyses) and heuristic constraints (which may possibly remove correct analyses).

The rules in Brill's transformation-based tagger are of a rather simpler form, as they have a different purpose. Rather than being a formalism allowing a linguist to express some generalisation they are aware of in ways that a computer program can implement, Brill's rules are templates for a program which develops its own rules. Brill's rule templates are all of a like form, to wit, "Change tag a to tag b when" some condition is fulfilled (1995: 553). The condition is a tag which must be found on a specified token to be fulfilled – the previous token (or the token before that), or the next token (or the token after that), or either of the two tokens before, or either of the two tokens after. The condition may also stipulate that two such matches must be made for the condition to be fulfilled. The "lexicalised" version of Brill's tagger allows rules to refer to the forms of nearby tokens as well as the analyses given to those tokens; again combinations of conditions are allowed. In the n-best version of Brill's tagger, the transformations add tags to those already there instead of changing tags. In the unsupervised version of Brill's tagger, the transformations reduce sets of tags to single tags.

In summary, a number of features may be perceived as being shared by the formalisms of Brill's tagger and Constraint Grammar, two approaches which are otherwise quite distinct. All rules consist of some operation to be carried out on the

analyses given to a token if a given set of conditions is fulfilled. One of these conditions refers to the analysis to be changed; this is the "target" in Constraint Grammar, and the "tag a" in Brill's paraphrase "Change tag a to tag b…". The other conditions may refer to the analyses on nearby tokens (ambiguously or unambiguously), or to the wordforms of nearby tokens, or (in the case of Constraint Grammar) to the lemma of a nearby token. There is provision for multiple conditions which incorporate, in some form or another, logical OR and logical AND. The operations that may be performed include imposing a specified analysis (in Brill's tagger), selecting a specified analysis from those currently given to the token in question, or deleting a specified analysis from those currently given to the token in question.

Although to actually use either the CG formalism or Brill's formalism for Unirule would have been computationally unfeasible, these shared features were used as the guiding principles of the Unirule formalism, which is discussed in the following section.

6.2.4.2 The Unirule formalism

When creating Unirule, I aimed to create a formalism which would be as flexible as possible, so as not to exclude the possibility of other researchers using it, whilst at the same time making the rules as transparent as possible to the non-specialist.

In Unirule, a rule consists of *conditions* and an *action* (the different types of conditions held in the domain, target and conditions of CG rules are treated simply as "conditions" in the Unirule formalism). The conditions are given before the actions,

each on a separate line.

6.2.4.2.1 Actions

The action is the operation which is performed on the "current" token[30] if the conditions are fulfilled. There are four possible actions, *assign*, *select*, *delete* and *deletenot*. An action is specified as follows in the Unirule formalism:

- a select NNMM1N

The initial "a" indicates that this is an action, the following word specifies the type of action that is required, and the tag given at the end is the argument of the action. The types of action are as follows:

assign The token is assigned the tag as given. All other tags are deleted.

select If one of the tags listed for that token matches the tag as given, then all other tags are deleted. If more than one tag listed for that token matches the tag as given, then the first such tag in the list is selected, and all other tags deleted. If no tags match the tag as given, then no action is taken.

delete Any tag matching the tag as given is deleted, unless it is the only tag remaining on the list.

deletenot Any tag *not* matching the tag as given is deleted, unless it is the only

[30] That is, the token that Unirule is currently analysing in the course of sequentially examining every token in the file.

tag remaining on the list.

The option which exists in Constraint Grammar to discard all but the target reading if the context conditions are fulfilled, and discard target reading if it is not, is not implemented in the Unirule formalism. However, the same effect could be achieved with two rules, one covering the positive case, and one the negative.

The actions *select* and *deletenot* are very similar. Indeed, in a best-case scenario, they would be identical. For example, the following two actions are identical if applied to a token which has the tag NNMM1N in its list of tags:

- a select NNMM1N
- a deletenot NNMM1N

However, a difference will become apparent if these actions are applied to a token which *does not* have NNMM1N in its list of tags. In this situation, *select* will take no action at all, whereas *deletenot* will remove all the tags except the last one on the list (in effect, selecting a random tag). Clearly, therefore, *select* is the better choice in this situation.

However, *select* has disadvantages if the given tag includes a wildcard character. Unirule recognises two wildcard characters. The asterisk (*) can represent any single character. So in rules for disambiguating Urdu, NNM*1N can be used to represent either NNMM1N or NNMF1N, since M and F are the only two characters that can appear in this context in the Urdu tagset. The tag given in an action may contain any number of this first type of wildcard character. The second wildcard character is #. This may only appear at the end of a string and represents *any zero or*

more characters up to the next white-space character. Thus, a string with # at the end may be used to specify the start of the tag without saying anything about the end. This feature has been particularly designed to take advantage of hierarchical tagsets[31], so N# can refer to all nouns, NNMM# to all masculine nouns, J# to all adjectives, JD# to all determiner-like adjectives, VH# to any form of the verb *hōnā*, and so on.

Obviously, when wildcard characters are used, it is possible for more than one tag on the list of a particular token to match the tag specified in the action. If *select* is the action used, then in such a circumstance, it would select the first such tag on the list – again, effectively a random selection. By contrast, when used with a tag containing wildcards, *deletenot* will leave intact any tags that match the tag as given – however many that might prove to be. Of course, *deletenot* will still impose a random selection if none of the tags given to that particular token matches the string (as *deletenot* will delete all but the last remaining tag).

6.2.4.2.2 Conditions

The conditions state what the features of the context must be for the action to be performed on the current token. A condition consists of an instruction on what type of comparison to carry out, the range of the comparison, and what is being looked for. For example:

- c ifnextwordis 1 be

The "c" specifies that this is a condition. The following word specifies the

[31] Note, however, that Unirule is still entirely compatible with non-hierarchical tagsets.

comparison type; it is followed by an integer indicating the range, and then by the string which is the basis for that comparison. This example condition will be fulfilled if the first word after the current token is the same as the defined string (in this case, the word "be").

In the current version of Unirule, the range may be up to 25 words in either direction (not counting the current token). I chose this value to make sure that Unirule could handle the long-range constraints used in Constraint Grammar[32]. Unirule has no direct means of specifying ranges in terms of "up to *x* words" or "more than *x* words" as Constraint Grammar and Brill's tagger both have to some degree. However, such rules may be expressed indirectly, by the use of multiple rules (an approach which I personally find more perspicuous). SGML elements (or anything else tagged as NULL) are not counted when calculating a range[33].

There are 18 comparison types. A comparison may look at the current token, the previous, or the next (3 options); it may compare the wordforms, or it may look for a tag that is to be matched unambiguously, or it may look for a tag simply

[32] Note however that because Unirule does not make reference to clause boundaries – an important feature of Constraint Grammar – the upper end of the 25-word range is not likely to be used very much by Unirule. Unfortunately, making reference to clause boundaries necessitates a degree of prior syntactic analysis which Unirule cannot afford to assume in its input.

[33] However, since Unirule's scope is implemented using a window of 25 tokens each way, any SGML elements within this window will reduce the maximum possible range. So if there is an SGML element (tagged NULL) immediately before the "current" token, then a previous-type condition with range 3 will actually look at the 4th word before the current token – but a condition with range 25 could not in such a circumstance be evaluated, and would be treated as *not* fulfilled. This is incidentally the key difference between the negative functions and a simple logical negation of the positive functions – *ifprevwordis 25 XXX* and *ifprevwordisnot 25 XXX* would *both* be treated as *not* fulfilled if the 25th word before the current token was not accessible due to intervening SGML.

occurring in the list of possible tags (3 options). Furthermore each type of comparison has its negative as well (2 options). The full list of comparison types is as follows:

ifthiswordis Fulfilled if the word-form of **this** token[34] matches

ifthistagis Fulfilled if all the tags on **this** token match[35]

ifthistaginc Fulfilled if at least one of the tags on **this** token matches

(these first three, and the corresponding negatives below, have no range)

ifprevwordis x Fulfilled if the word-form of the x^{th} token **before** this token matches

ifprevtagis x Fulfilled if all the tags on the x^{th} token **before** this token match

ifprevtaginc x Fulfilled if at least one of the tags on the x^{th} token **before** this token matches

ifnextwordis x Fulfilled if the word-form of the x^{th} token **after** this token matches

ifnexttagis x Fulfilled if all the tags on the x^{th} token **after** this token match

ifnexttaginc x Fulfilled if at least one of the tags on the x^{th} token **after** this token matches

ifthiswordisnot Fulfilled if the word-form of **this** token **does not** match

ifthistagisnot Fulfilled if at least one of the tags on **this** token **does not** match (i.e. if this token is not unambiguously tagged as…)

ifthistagincnot Fulfilled if all the tags on **this** token **do not** match (i.e. if none of the tags on this token are the same as)

ifprevwordisnot x Fulfilled if the word-form of the x^{th} token **before** this token **does not** match

ifprevtagisnot x Fulfilled if at least one of the tags on the x^{th} token **before** this token **does not** match

ifprevtagincnot x Fulfilled if all the tags on the x^{th} token **before** this token **do not** match

ifnextwordisnot x Fulfilled if the word-form of the x^{th} token **after** this token **does**

[34] That is, the "current" token as previously defined (the one which Unirule is currently analysing).

[35] That is, if this word is unambiguously tagged as – compare *ifthistaginc*, which will accept ambiguously tagged words for its comparison; *ifthistagis* will not do this.

	not match
ifnexttagisnot x	Fulfilled if at least one of the tags on the x^{th} token **after** this token **does not** match
ifnexttagincnot x	Fulfilled if all the tags on the x^{th} token **after** this token **do not** match

Note that in the negative forms of the above conditions, the "all the tags" conditions apply to "ifthistagisnot", etc., and the "at least one of the tags" conditions apply to "ifthistagincnot", etc. This is for reasons of logic. The negation of ifthistagis would be that *if* all the tags match the string given, then the condition is *not* fulfilled. Therefore the corresponding negative condition would be fulfilled *if* at least one of the tags failed to match the string. Conversely the negation of ifthistaginc would be that *if* at least one of the tags matches the string given, then the condition is *not* fulfilled. Therefore the corresponding negative condition would be fulfilled if all the tags failed to match the string.

The terms "previous" and "next" have been used to refer to direction instead of "left" and "right", because Unirule was designed from the outset to be non-language specific, and therefore it is better to avoid terminology rooted in any particular writing system which is not universally applicable. The conditions in Unirule are rather wordier and more bulky than the comparable conditions in – say – Constraint Grammar. However, I believe that this makes them more perspicuous to the untrained reader and is thus by no means necessarily a drawback.

6.2.4.2.3 Some example rules

A single rule consists of an action, and all its accompanying conditions, which are listed directly before the action (and after the immediately preceding action). If an

action has no conditions, it will be triggered on every token. If the action has more than one condition, then all the conditions must be fulfilled for the action to take effect (logical AND). In the interests of simplicity there is no logical OR, though this effect could easily be achieved by adding another rule with the same action and a different condition.

It is my intention and my belief that one can express in the Unirule formalism any generalisation or constraint that could be expressed in Brill's formalism or the CG formalism, although it has not been possible to find out for certain whether this is indeed the case. One unique feature of Unirule is that it only accepts rules saved in files in Unicode format – this allows conditions like "ifnextwordis" to refer directly to actual Unicode forms.

A Unirule rule file is a single Unicode text file consisting of a string of conditions and actions. Comment lines may be included (beginning in /) and empty lines are passed over. There follows an example of a very short set of rules (actually a toy ruleset for Urdu used in the development of the Unirule software).

```
/ postpositions follow (pro)nouns, therefore delete verb, adjective and adverb tags
c ifnexttagis 1 II#
a delete V#
/ note: this does not account for infinitives, which may precede postpositions
c ifnexttagis 1 II#
a delete J#
c ifnexttagis 1 II#
a delete R#
/ postpositions follow oblique case, therefore delete nominative and vocative
c ifnexttagis 1 II#
a delete N****N
```

```
c ifnexttagis 1 II#
a delete N****V
/ pronouns can be nominative before nē but are oblique before other postpositions
c ifnexttagis 1 II#
c ifnextwordisnot 1 ے
a delete PP**N
```

6.2.4.3 How Unirule works

As has already been mentioned, Unirule requires a set of rules to run, which it loads in from a text file at time of running. For input it requires Unicode text in the Unitag vertical format, analysed ambiguously. The rules are stored in a linked list (if any of the rules is malformed according to the system already described, the reading will abort, and disambiguation will be performed with a truncated rule list). The entire set of rules is applied, in the order they appear in the rule file, to each token in the text file in turn. Any given rule may or may not alter the tagging of a particular token. If it does, then the responsibility tag is changed to that of the Unirule program. This is an R followed by the number of the rule in question – rules are numbered in base 36 (i.e. using all digits and letters) from the start of the rule file to the end.

Thus Unirule produces an output file which is the same as the input file with some of ambiguities removed. While it is ultimately based on a single-pass algorithm, Unirule can be instructed to make a multiple passes of a single file, to allow rules to take advantage of the unambiguous context that the application of other rules has provided[36].

[36] Of course, multiple passes can also be simulated by repeating rules within a rule list. The later copy of the rules will be using more precise information than the earlier copy, and therefore may do better than the first copy of the rule.

Although primarily designed as a disambiguator, Unirule could also work as an improver – for example, if one did not use the ambiguity-tolerant comparison-types, and only used the *assign* action, then Unirule could easily be used to implement an improver.

A call to Unirule from the command line has the following form: *unirule input_filename output_filename rulelist_filename no_passes* . Within Unitag, therefore, the name of the file containing the list of rules and the number of passes must be specified in the instantiation file.

6.3 Creating and optimising the lexicon

In this section, I will firstly discuss *Unilex*, the software which manages lexicons within the Unitag framework; I will secondly discuss the manual lexicons used within the Urdu tagging system, and finally I will describe how the lexicon was optimised for use in tagging.

6.3.1 The Unilex software

The *Unilex* program is an adjunct to the Unitag tagger architecture. It uses the same formats as Unitag, and can be used to acquire and manage the lexicons which programs like Urdutag require to analyse text.

Unilex has several functions. As well as being able to sort lexicons by wordform or by first tag and merge two lexicons together (ensuring that no word is omitted or repeated), it can also derive a lexicon automatically from a tagged file in the Unitag vertical format. Lexicons are stored in a similar format. A line of a lexicon

file looks like this:

i000365 کرنی VVNF1 VVNF2

The serial number begins with “i” (for “item”) and is followed by 6 figures (to distinguish lexicons from tagged files). This is followed by a space, and then the wordform. There is then a horizontal tab character, followed by a list of all possible tags fro that wordform separated by spaces, followed by a carriage return. As with the Unitag format, the lexicon format may contain “probabilities”, in this format:

i000365 کرنی VVNF1/50 VVNF2/50

The same rules apply to probabilities in Unilex and its associated file format as to the Unitag format (see 6.2.1.3 above). Unilex produces these probabilities by counting all the instances of the wordform-tag combinations, and dividing by the total number of occurrences of the wordform. It does not allow probability to reach 100%, so the highest value that can occur is 99.

A key parameter for the function of Unilex which acquires lexicons from text files is the “frequency threshold”. This is an integer, selected by the user, which sets a minimum number of occurrences below which a word will not be included in the lexicon. So if the threshold is set at 3, and the word *kitāb* occurs twice in the file, it will not be included in the lexicon. See 6.3.3.1 and 6.3.3.2 below for a discussion of the importance of this parameter.

In the development of the Urdu tagging system, Unilex was used to create a spoken lexicon, a written lexicon (each emanating from the relevant training data) and to combine the lexicons together in various ways to create the lexicons that Urdutag

actually uses.

6.3.2 Manual lexicons

The lexicon actually used by Urdutag is a combination, compiled using Unilex, of four separate lexicons. The first is the lexicon acquired automatically from the training dataset (see 4.5). The other three were written manually using a Unicode word processor[37]. One contains approximately 30 words which consist of words incorporating clitic forms that must be split off. These are not given tags, but instead markers to indicate that this is what they are; this is part of the system-internal workings of Urdutag and should not be seen as part of the lexicon *per se*.

The second contains, to date, 63 words which are classified as "exceptions". That is to say, these are words which go against the tendencies written into the suffix-analysing algorithm in Urdutag, or disrupt the workings of the tagging system in some other way, and would be tagged incorrectly if not contained in the lexicon. The first words in this category were the ones identified as exceptions during the design of Urdutag (marked with a (***) in 6.2.3.3 above). To this were added all words beginning in *al–* where this did not represent the Arabic definite article (if these were not in the lexicon, Urdutag would split off the *al–* as a clitic). More words were added in order to correct errors made by Urdutag and bring the analyser's accuracy as close as possible to 100%, as discussed above in 6.2.3. A few more were added during the

[37] As well as automatic derivation and manual creation, it is also, as a general rule, possible to obtain a tagging lexicon from a pre-existing electronic lexicon or dictionary. Unfortunately, for Urdu, there do not appear to exist any suitable electronic lexicons. Thus, the lexicon described here was created entirely from scratch.

process of devising the rule list (see below). For example, the adjective *purānā*, "old", had to be added to this exceptions list because of its formal resemblance to an infinitive verb, as did the unmarked noun *pānī*, "water", because of its resemblance to a marked feminine noun or adjective.

The third and final manual lexicon contains all words in the closed categories, with the exception of regularly formed ordinal numbers. Holding all non-lexical words in the lexicon avoids the problems one might get if one attempted to acquire the non-lexical words from a text. For example, the word *hai* is probably one of the most common in the Urdu language[38]; it is usually VHHV1 (third person singular present tense form of the auxiliary verb *hōna*). However, it can also be VHHT1 (second person singular). Because of the very restricted usage of the second person singular in Urdu, it would be entirely plausible for *hai*_VHHT1 never to occur in a text or set of texts. In the largest of the spoken texts in the training corpus, which is over 16 thousand words in length, VHHT1 does not occur at all (compared to 526 occurrences of VHHV1 – i.e. roughly one word in every 32). It would therefore be quite possible for Unilex to create lexicons which stated that the only permissible tag for *hai* was VHHV1. But this is not the case; its other use may be rare but it is not non-existent, and assuming otherwise will eventually lead to an error. The manual closed-class lexicon contains both VHHT1 and VHHV1 for *hai* and therefore avoids this pitfall.

[38] The unavailability of fully Perso-Arabic compatible corpus analysis software made it impossible to create a wordlist and quantify this (although new versions of the SARA and WordSmith software forthcoming in the near future will hopefully amend this lack). However, *hai* occurs 1,213 times in the 38,000 word training dataset – clearly this word is very common.

6.3.3 Optimising the lexicon

Given that there are a number of variables involved in the production of a lexicon for tagging, it is reasonable to ask what the optimal values or settings for these variables are. This is the issue that I address in this section. Firstly, I will identify and discuss some aspects of lexicon creation which must be optimised. Then, in the following subsections (6.3.3.2 to 6.3.3.4) I will describe the optimisation of these aspects, before describing (in 6.3.3.5) the optimal lexicon eventually arrived at.

6.3.3.1 Variables in lexicon creation

In this section, by "variable" I do not necessary mean a variable in the strict mathematical sense, but rather some aspect or point in the process which may differ and whose difference might conceivably have an effect on the ultimate output.

Clearly not all aspects of lexicon creation which may have such an effect are possible to manipulate. The manual lexicons, for example, must simply be as good as can be achieved, in the time available, based on the knowledge available. For these latter two variables, "optimised" essentially means "maximised" – and as they have been, there is no further benefit to be had experimenting with them. Therefore, we may treat the closed-category lexicon and the exceptions lexicon as fixed once complete[39], and consider the automatic acquisition as something which needs to be optimised. In that case, what are the variables in the process of acquiring a lexicon automatically which can be manipulated in an effort to find the maximal

[39] Of course, in practice, both manual lexicons, like the majority of aspects of the tagging system, underwent continual, incremental revision and improvement during the development process.

configuration?

The most obvious of these, arising from the nature of the Unilex program, is the threshold. As defined above, the *frequency threshold* is the minimum number of times that a wordform must occur in the source file if it is to be included in the lexicon, and this is specified by the user when Unilex is creating the lexicon.

The advantage of a low threshold is clear: since it includes many wordforms (or every wordform), a lexicon created this way is more capable of giving to any token a tag that has been attested in the training data. While one might therefore expect a low threshold to give the best results, and while this ought indeed to be the case when looking at just the training data, this may not hold for other text. This is because of the order of operations in Urdutag's analysis algorithm. Lexical look-up precedes and pre-empts morphological analysis. It might be possible for a lexicon acquired from training data to exclude a legitimate tag for a given wordform, simply because an instance of the word having that tag did not occur in the training data; in this case, the lexical look-up will perform less well than the suffix-analysis part of Urdutag, which might well have deduced the correct tag for the shape of the word. For example, if the word *sunī~/sunē~*[40], "hear (perfective participle/subjunctive)" occurs only as VVYF2N in the training text, Urdutag would use lexical look-up to determine that every instance of *sunī~/sunē~* must be VVYF2N. But *sunī~/sunē~* may also be VVSV2 or VVSM2, and an instance of *sunī~/sunē~* as a subjunctive verb just does not happen to have occurred in the training text. In this case, Urdutag would have achieved better accuracy by letting the suffix analyser work on the form of the word, to produce the following list of tags: NNUF2N, VVSM2, VVSV2, VVYF2N.

[40] The two words (though not homophonous) are identical in Indo-Perso-Arabic script; see 6.2.3.3 for more on this phenomenon.

This list includes all the correct readings (plus an extraneous noun tag). This is obviously more likely to happen with words that occur relatively infrequently in the source text. Thus, a higher threshold can prevent these kinds of "blocking" entries taking root in the lexicon.

So, in the light of the comprehensiveness given by a low threshold, and the exclusion of partial relationships that might block the suffix analysis given by a high threshold, we are entitled to conclude that the optimum threshold in terms of tagging accuracy is not obvious *a priori*.

The second variable that can be changed in the creation of the lexicon is the source data used to produce it. It would thus have been desirable to compare a lexicon derived from spoken data, a lexicon derived from written data, and a lexicon derived from both (see Smith 1997: 141-142 for a discussion of the differences which may occur between a spoken lexicon and a written lexicon). However it was not possible to do this, because only for spoken Urdu was enough training data available (see 4.5)[41].

The third variable is the variable of combination with the manual lexicons. Although one would naturally assume that adding in the closed-class and exceptions lexicons would improve accuracy, this is again by no means proven. By experimenting with different combinations of the automatic and manual lexicons, this hypothesis can either be confirmed or disproved, in either case to the benefit of the overall standard of tagging produced.

[41] A further potentially valuable step, that of crafting supplementary lexicons for specific text genres, was clearly impossible given the small quantity of available data.

6.3.3.2 Deducing the optimal threshold

To discover the effect of the lexicon threshold, Urdutag was run on the training data and the two test texts using 11 different lexicons, created using automatic lexicons with thresholds of 1 to 10 and, in addition, the manual lexicon on its own[42]. Fig. 6.2 below shows the number of items in each lexicon, and those items' provenance; Fig. 6.3 shows the resulting accuracy[43].

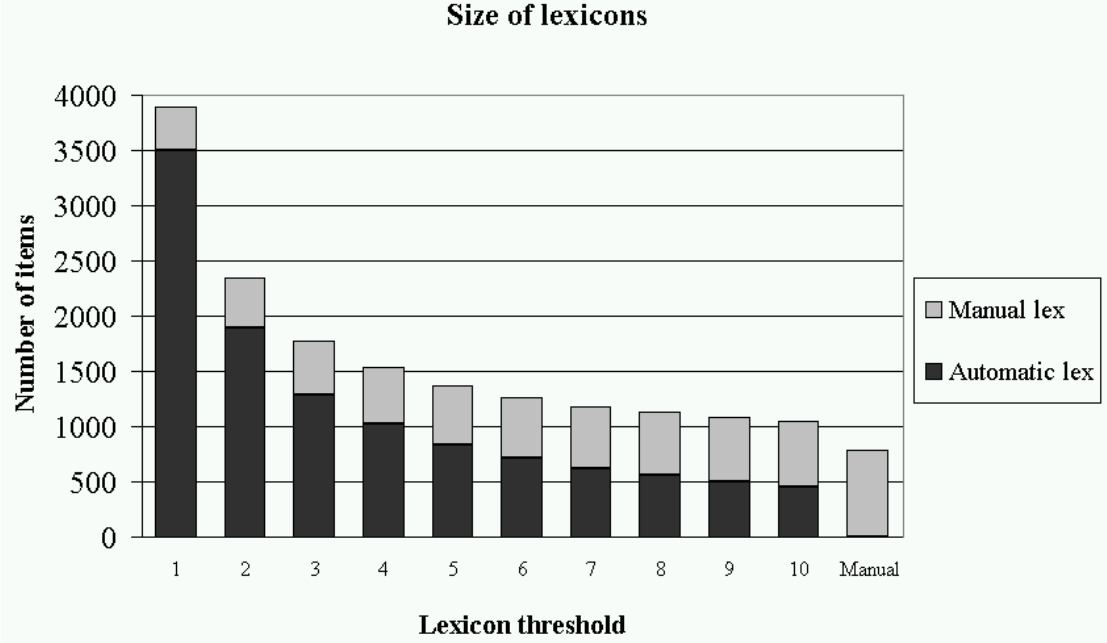

Fig. 6.2

[42] This experiment was performed with an early version of Urdutag. I make the assumption that the optimal lexicon with this early version would remain optimal with an improved analyser and in combination with a disambiguator. This assumption – also made with regard to other tests described in this chapter – could not be tested. This is a problem inherent in this kind of multivariate system. To examine all the different combinations is impossible, so the variables (lexicon, rule list, and so on) must be treated as if they were independent for testing purposes.

[43] Accuracy rates are not commensurate with those ultimately attained by the tagging system.

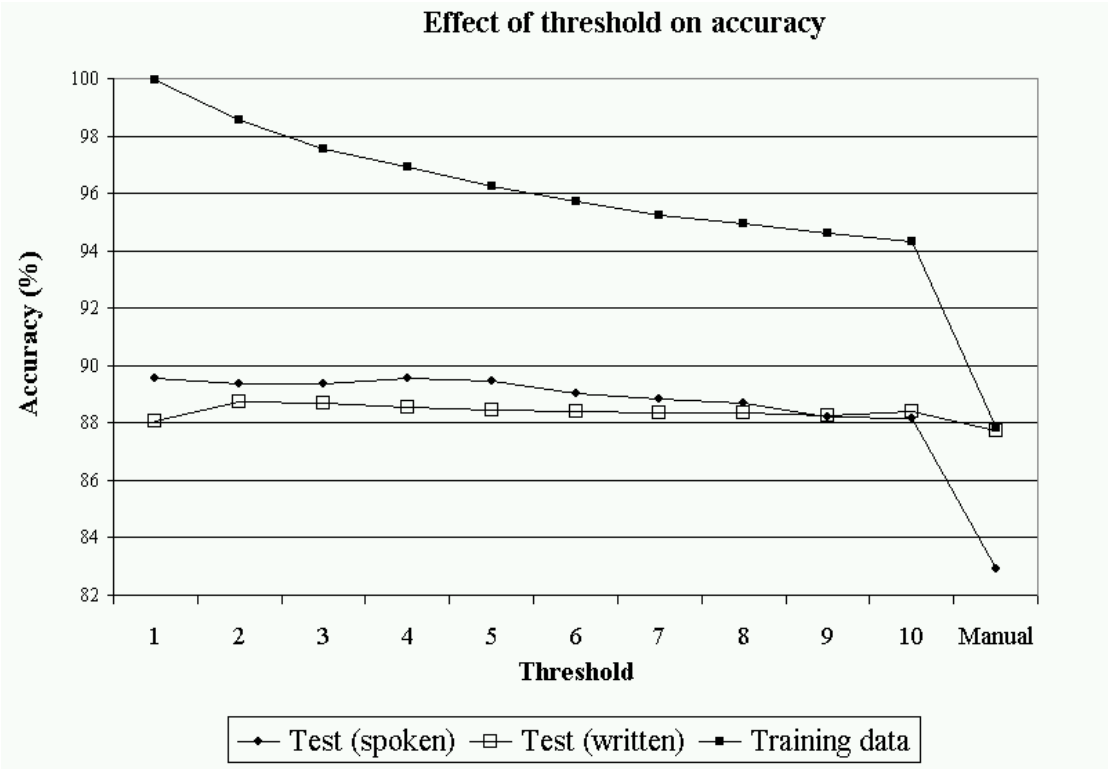

Fig. 6.3

From this experiment, it seemed clear that my initial suspicions were correct. Apart from the clear indication that a lexicon derived from the data being tagged is very useful (thus the much better performance achieved on the training data), several points arise from these results. The manual lexicon alone produces poor results. A threshold 1 lexicon produces excellent results for the training data from which it was derived, but on other data does not necessarily perform better than a lexicon with a higher threshold. The threshold 1 and threshold 4 lexicons do equally well on the spoken test data, and a threshold 2 lexicon does best on the written data.

I then looked in more detail at the first 25 errors[44] made on the written data with a lexicon threshold of 1 that were not made when the threshold was 2. They were without exception caused by process A10 (i.e. lexical lookup) in Urdutag. In each

[44] It was necessary to look at a relatively small number of errors, because sorting through an error report from Comparetag is a very time-intensive task.

case only a single tag was offered, and it almost always represented a category which was (inflectionally speaking) of the same form as another category. These are exactly the categories in which one would expect a suffix analyser to catch what a low-threshold lexicon might miss[45]. Some examples are shown below:

s00013 w015 لگائی	A10 NNMF1N
s00028 w013 چڑھ	A10 VX0
s00031 w039 بولنے	A10 VVNM1O
s00033 w016 علاقوں	A10 NNUM2O
s00040 w010 بولنے	A10 VVNM1O

Within the same section of the error list, there was only one error made by the morphological analyser that the lexical lookup had avoided (tagging *purānē*, "old", JJM2N, as an infinitive verb).

Although it is clear from this that the "blocking" effect described above is real, and problematic for tagging, the experiment does not give sufficient evidence to select an optimum threshold, since the two test texts behaved somewhat differently. I therefore considered another way to get around the blocking "problem". Instead of removing the offending entries from the lexicon by raising the threshold, it seemed logical to attempt to enrich the lexicon in an attempt to add to those entries the tags that they were missing.

[45] There were also some words that were wrongly tagged by both suffix analysis and lexical lookup.

6.3.3.3 Enriching a lexicon

I wrote a program, *Growurdulex*, to enrich an automatically acquired lexicon. It does this by adding tags to the entries in the lexicon on the basis of the tags that are already there. I did not attempt to infer any new entries, as this would be tantamount to lemmatisation, which I had already decided to avoid. Instead, several groups of tags were designated that applied to forms which were morphologically identical. If an entry in the lexicon had any member of a group, then the other members were added if not already there[46]. An example of a group is JJM1O~JJM2N~JJM2O, all of which take the suffix *–ē*.

When the experiment described above was redone with enriched lexicons, the results were as shown in Fig. 6.4.

[46] There were also some non-reciprocal additions, for example if RRJ was present then JJM1O~JJM2N~JJM2O would be added, but not vice versa. Another operation performed by Growurdulex was to standardise the use of vowels in the lexicon to comply with the expectations of Urdutag's lexical lookup (see footnote 19 on page 325).

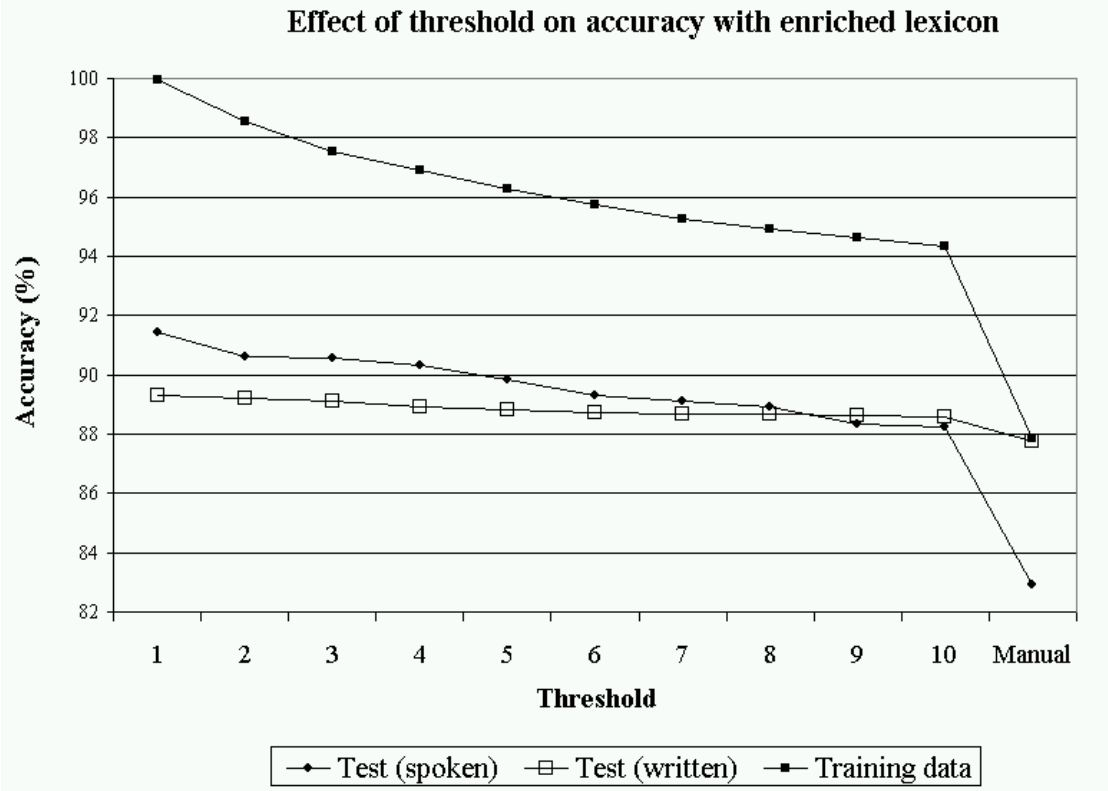

Fig. 6.4

As can be seen, the blocking effect vanishes, so that the threshold 1 lexicon performs best for all three datasets. There is also a noticeable gain in overall best accuracy: 1.9% for the spoken text, 0.6% for the written text. Moreover this comes at a relatively small cost in extra ambiguity (0.28) for the spoken text, and with a *reduction* in ambiguity for the written text[47]. Therefore, the capacity to enrich a lexicon resolves the problem of finding an optimal threshold: it is 1.

6.3.3.4 Different combinations of lexicons

One would anticipate that the addition of the manual lexicon to the automatic

[47] The reduction in ambiguity was possible because the best-performing enriched lexicon had a lower threshold than the best-performing non-enriched lexicon. Increasing the threshold increased the ambiguity in the output in both experiments.

lexicon would be an improvement. In this section, I aim to demonstrate this. Using an automatic lexicon of threshold 1 and the manual "closed class" and "exceptions" lexicons, run on both test texts alone and in combination[48]. The results were as follows:

Table 6.4

Type of Lexicon	Items in lexicon	Accuracy (spoken text)	Accuracy (written text)
Automatic	3536	89.2%	84.3%
Closed class	730	82.7%	87.6%
Exceptions	95	29.5%	32.5%
Automatic + Closed	3867	89.5%	88.1%
Automatic + Exceptions	3562	89.2%	84.3%
Closed + Exceptions	788	82.9%	87.9%
All three	3893	89.6%	88.1%

As anticipated, the best results are obtained by the combination of all three, although for both texts the contribution of the exceptions lexicon is small. This is not unexpected as it is a fraction the size of the closed class lexicon.

6.3.3.5 Summary: the optimal lexicon

The optimal lexicon is an automatically-derived lexicon created using an

[48] All were run in combination with the lexicon containing the Urdutag-internal codes for words with clitics. The training text was not used for this experiment, as with a threshold of 1 it would be tagged with 100% accuracy.

inclusion threshold of 1 (i.e. every word-from in the training text included), enriched with morphologically parallel tags, and combined with the manually-created lists of closed category words and exception words.

There is evidence to suggest that the genre of the text from which the lexicon is derived is critical. Notice that the spoken test text was at all points easier for Urdutag to handle than the written text, since the training data is also spoken.

6.4 Developing a rule list for Urdu

6.4.1 How the rule list was developed

There are two conceivable ways to go about arriving at a list of disambiguation rules. One way begins with linguistic knowledge. The designer uses their knowledge of the language in question to find generalisations about sequences of parts-of-speech which they anticipate will reduce ambiguity. Rules are then devised to encode these generalisations. The other way is based on examination of the data. The designer looks at errors that have been made and ambiguity that remains, and the context in which these occur, in a set of training data that has been subjected to the rules as they stand. In the light of this, the designer creates or modifies the rules to prevent the errors or reduce the ambiguity.

The latter method is that recommended for use by Voutilainen (1999c: 226) in the Constraint Grammar framework. It is also, in effect, employed in Brill's transformation-based error-driven learning methodology (although in this case, the rules are learnt automatically)[49].

[49] See 5.2.2 on Constraint Grammar and 5.4.2 on Brill's methodology.

However, in my case, an initial period was devoted to working on rules based on linguistic knowledge – in particular using my model of Urdu grammar, Schmidt (1999), as a source of potential rules. It is desirable to reduce the level of ambiguity to a somewhat more manageable level before attempting to analyse it, by implementing the obvious rules which arise from the syntactic structure of Urdu. Furthermore, this course of action allowed some work to be done on the rule list before the full set of training data was available. The "acid test" of the rules, however acquired, was their performance rather than any notion of their linguistic accuracy or descriptiveness.

Following this stage, existing rules were corrected, and further rules developed, by examining output from Unirule for errors and remaining ambiguity[50].

However the rule was developed, it was necessary to move from a generalisation about the surface structure of the language to one, or in most cases more than one, rules in the Unirule formalism. This process is illustrated in the following section. All rules – including those derived from Schmidt's description – remained permanently open to being edited or deleted from the list in the light of experience with actual data, throughout the development of the rule list.

6.4.2 The nature of the rules

I began with a set of 105 rules based on my knowledge of the structure of Urdu derived from studying Schmidt (1999). In each case, a principle of Urdu grammar was translated into one or, in most cases, several rules in the Unirule formalism. These principles were as follows (numbers in brackets indicate the number

[50] This stage of the project – and the creation of the rule list in general –was greatly hampered by the lack of a Unicode text processor able to handle large text files and display their contents correctly.

of rules related to that principle):

- Postpositions follow nouns and verb infinitives (4)
- Nouns and pronouns take the oblique case before postpositions (6)
- The particle *hī* is clitic after the oblique case of plural personal pronouns, but not the nominative (2)
- In the phrase *āp kā / kē / kī*, *āp* is reflexive, not honorific (1)
- In a compound postposition consisting of *kē* plus adverb, *kē* is IIM1O (1)
- Words with adjective-like inflection (i.e. marked adjectives, the marked postposition *kā*, marked determiner-like adjectives such as ordinal numerals, the adjectival particle *sā* , and possessive pronouns) agree with a following noun for case, gender and number (78)
- The auxiliary *rahā* is preceded by a verb in the root form (1)
- General auxiliary verbs and the verbal postposition *kē* always follow a verb in the root form (3)
- The future auxiliary follows a subjunctive verb (1)
- An infinitive before *cāhiē~* (VC2) is not singular (1)
- *kyā* at the start of a clause is a question marker rather than an interrogative pronoun[51] (2)
- The principle that Unirule should always delete an F* tag, especially FU, if another analysis was available was also adopted (5)

As an example of how these principles translate into rules, let us take the principle that nouns and pronoun are oblique before postpositions. It was first

[51] This principle turned out to be unreliable in practice and was removed at a later stage.

necessary to consider what kind of operation should be performed. The rule could be stated as "if the next word is unambiguously tagged as a postposition, remove any tags indicating a nominative or vocative noun". This translates into the Unirule formalism as:

c ifnexttagis 1 II#

a delete N****N

c ifnexttagis 1 II#

a delete N****V

This only covers nouns. Pronouns are more complicated. Personal pronouns are nominative, not oblique, before the postposition *nē*. Furthermore, pronoun tags vary in length, so the same wildcard template will not fit all pronoun tags, as is the case for nouns. The rules for pronouns are as follows:

c ifnexttagis 1 II#

c ifnextwordisnot 1 نے

a delete PP**N

c ifnexttagis 1 II

c ifnextwordis 1 نے

a delete PP**O

c ifnexttagis 1 II#

a delete P**N

c ifnexttagis 1 II#

a delete PNN

Two rules handle the personal pronouns (one for if the next word is *nē*, and one for if it is not); another deals with the pronouns of the Y-V-K-J set; and the last handles the indefinite pronoun.

These 105 rules were applied to data analysed using Urdutag and the optimal lexicon (see above). Before the application of the rules, accuracy was 100% and ambiguity 2.89; afterwards, they were 97.8% and 2.55, respectively. This performance was then improved by adding additional rules using the training data, and by using multiple passes as described below, to reach 99.0% and 1.73.

Some of these rules were very general – for example inspection of the data revealed that a participle which was followed by a verb is nominative rather than oblique, a principle which (in the manner described above) led to four rules that removed a large amount of ambiguity. Other rules were dataset-specific – for example, rules to select FB tags for the phrase *bī bī sī*, "BBC", quite common in transcripts from the BBC Asian Network. Others were specific to speech rather than writing (e.g. a rule which selects vocative tags if a noun is followed by the greeting *alsalām 'alaikum*). However, the presence of such rules should not cause errors in texts that lack these features (as the rules will simply not be triggered). So far as it has been possible to verify this in practice using the written test data, this has proven to be the case.

Ultimately a total of 274 rules were written. Some of these disambiguate only one or two tokens each; others disambiguate hundreds of tokens.

6.4.3 The remaining ambiguity

The remaining ambiguity was very difficult to remove without causing large numbers of errors. Some of this was down to categories which have identical forms – such as the various tags for feminine adjectives. JJF1N, JJF1O, JJF2N and JJF2O are impossible to tell apart from their form. If it is not followed by a noun whose case and

number features are unambiguous, then disambiguating the case and number features of a feminine adjective is more or less impossible. Another problem was disambiguating words that could be adjectives or adverbs (i.e. JJU/RR or JJM1O/RRJ). I was unable to find any way based on context to tell the difference between these two. There were also individual problematic words. For instance سو (*sau*, "seven", or *sō*, a multiple homonym) was extremely difficult to disambiguate. Virtually all instances of this word retain the tags JDNU[52], CC and RR.

6.4.4 What can be learnt from the rules?

The original rules were based on statements made by Schmidt (1999); subsequent rules were developed with continued use of Schmidt's descriptions as a reference model. It is therefore not to be expected that any great insight should emerge from the writing of the grammatical rules that was not already there in Schmidt's description. However, some minor points of Urdu grammar did emerge that were not present in the original model.

For instance, the rules based on the principle of a verb root followed by a general auxiliary verb (VV0 followed by VX#) originally caused a large number of errors. Examination of these errors showed that it was possible for there to be an intervening word of one of these categories: XH, XT, RM, RMN. This structure (verb root – particle/adverb – general auxiliary verb) is not outlined explicitly in Schmidt's model. But the necessity to amend the rules to allow for this structure brought it to my attention.

[52] Any word homonymous with a numeral is difficult to fully disambiguate since the numerals may occur in a great variety of syntactic settings.

No claim is made that the above details represent an original discovery about Urdu grammar. They were, however, unknown to me until the process of rule writing brought them to light. It would therefore seem clear that the process of developing a rule list can in and of itself be linguistically informative.

6.4.5 The order of the rules

The rules are stored in the rule list in an order designed for ease of editing; the rules relating to different parts of speech are grouped together. In Brill's system, rules are ordered according to their individual effectiveness (1995: 552). However, I made no effort to do this with the Urdu disambiguation rules. This was mainly because the amount of training data was not sufficient for differences in the effectiveness of the rules to be measured with any certainty. However, in Unirule, ordering is not necessary, because all rules are applied on each pass through the file, as opposed to Brill's system, where one rule is applied at a time.

It would also be theoretically possible to order the rules according to their collective effectiveness, i.e. to compare the performance of lists with the rules in different sequences and select the best order. One could easily imagine a situation where a rule low down in the list alters the annotation of a token in such a way that would cause a rule high in the list to operate, too late for that rule to come into effect. But to find an optimal ordering for 274 rules would involve the analysis of more possible sequences than could practicably be assessed. Therefore, I exploited the ability of Unirule to perform more than one pass of each text. This ought to compensate for sub-optimal ordering of the rules (since context altered by a late rule can trigger the early rule on a subsequent pass). This is discussed in the following

section.

That said, there were some rules which very clearly had to be in a particular relative order. For example, there are rules which delete all F* tags; these plainly conflict with rules such as the one which selects FB FB FB for the phrase *bī bī sī*. In these cases it is necessary for the rule with more specific conditions to occur earlier. The rules deleting F* tags, for instance, were placed at the end of the file, to give other rules (such as those for *bī bī sī*) every chance of selecting an F* tag, since the F*-deleting rules will allow F* to remain if all other tags have been eliminated.

6.4.6 The number of cycles of disambiguation

Unirule allows the user to specify how many cycles of disambiguation a file should be put through. On second and subsequent passes through a file, rules may be triggered which did not apply on earlier passes, the earlier phases having created the conditions necessary for the word to apply.

For example, one Urdu rule selects oblique tags for nouns on the basis of a following postposition. But if the following word is not tagged unambiguously as a postposition, the rule cannot operate – even if another rule will subsequently disambiguate the postposition.

There are thus benefits to making multiple passes over the file. However, there is also a risk that multiple passes would decrease the accuracy of the tagging in unpredictable ways, since the rules were devised and tested using a single pass of the training corpus.

To establish what the optimal number of passes through the file might be, I compared the results obtained on all three datasets using a number of passes varying

from one to ten[53]. The results are given below.

Table 6.5

Unirule passes	Training data		Spoken test data		Written test data	
	Accuracy	Ambiguity	Accuracy	Ambiguity	Accuracy	Ambiguity
0	100.0	2.58	92.5	3.11	89.9	3.97
1	99.6	2.12	91.6	2.66	89.4	3.50
2	99.3	2.06	91.4	2.58	89.0	3.33
3	99.3	2.06	91.4	2.58	89.0	3.33
4	99.3	2.06	91.4	2.58	89.0	3.33

There were no noticeable changes in accuracy or ambiguity after the second pass. The third pass caused reductions in both too small to show up in figures of the given degree of precision. There were no changes at all after the third pass.

This is encouraging, because if errors on one pass were causing more errors on later passes, one might expect performance to degrade more sharply with each pass. The exact opposite is the case. It is also gratifying that maximum results are obtained with a relatively small number of passes, since the computer time required for a pass is not negligible. On my system (see footnote 4 on page 306), a pass of the training data took slightly more than a minute.

On the basis of these results, I felt able to add to the rule list a set of rules that would necessarily only be activated on the second pass of a file, because they required some word occurring afterwards to be disambiguated. I also adopted a policy of running Unirule for whatever number of passes might be required to reach the final rate of accuracy/ambiguity.

[53] This was done at a stage when 155 rules had been developed.

6.5 Concluding remarks

In this chapter, I have given an overview of the software and formalisms used to build the Urdu tagger. I have also discussed two main processes in developing the tagger, the creation of a lexicon and the creation of a list of rules for use in disambiguation. The net result is a working system capable of tagging running Urdu text.

This completes my description of the tagging system for Urdu. In the following chapter, I will discuss the conclusions which may be drawn from the experiments outlined here, and from this entire thesis.