Technical Report

UCAM-CL-TR-155
ISSN 1476-2986

Number 155

UNIVERSITY OF
CAMBRIDGE

Computer Laboratory

Computational morphology of English

S.G. Pulman, G.J. Russell, G.D. Ritchie,
A.W. Black

January 1989

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

© 1989 S.G. Pulman, G.J. Russell, G.D. Ritchie, A.W. Black

Technical reports published by the University of Cambridge
Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/TechReports/

ISSN 1476-2986

Computational Morphology of English

S. G. Pulman, G. J. Russell,
Computer Laboratory,
Cambridge University.

G. D. Ritchie, A. W. Black,
Dept. of Artificial Intelligence,
Edinburgh University.

Introduction

This paper describes an implemented computer program which uses various kinds of linguistic knowledge to analyse existing or novel word forms in terms of their components. Three main types of knowledge are required (for English): knowledge about spelling or phonological changes consequent upon affixation (notice we are only dealing with isolated word forms); knowledge about the syntactic or semantic properties of affixation (i.e. inflexional and derivational morphology), and knowledge about the properties of the stored base forms of words (which in our case are always themselves words, rather than more abstract entities). These three types of information are stored as data files, represented in exactly the form a linguist might employ. These data files are then compiled by the system to produce a run-time program which will analyse arbitrary word forms presented to it in a way consistent with the original linguistic description.

The system can therefore be viewed as falling into two logically distinct parts: a linguistic description, consisting of a set of spelling or phonological rules, word grammar rules and the lexical entries themselves; and a set of mechanisms - a spelling rule automaton driver, and a morpheme level parser - for applying this description during the analysis or generation of word forms.

The current system contains two separate descriptions of English, one based on Generalised Phrase Structure Grammar, (Gazdar, Klein, Pullum and Sag 1985), and a second based on a more general type of unification grammar formalism, adopting a more traditional approach to English morphology.

The linguistic theory implicit in some of the formalisms used is known to face some problems of descriptive adequacy, but it is demonstrably capable of handling most aspects of English morphology, and that of other languages, in a way that is theoretically interesting. Furthermore, it has the merit that the formal properties and intended interpretations of the notations used are well enough understood that all their consequences can be inspected via the computational implementation of them. In contrast, there are reasonable grounds for suspicion that much of current phonological and morphological theory is, as far as their formal apparatus is concerned, playing with 'notation without denotation'; that is to say, the literature contains a wealth of proposed formalisms whose mathematical and computational properties are, to say the least, obscure. This fact entails that the

proponents of theories couched in such notations quite literally do not know what they are saying with them: there is no way that the linguistic consequences of these notations can be systematically explored if their formal properties are not known. Thus although what is described here deals only with written English input, it should not be dismissed as mere 'linguistic engineering', but as, among other things, a possible test-bed for experimentation with alternative theories of morphophonology, where these prove to be specifiable with sufficient precision. The system is flexible enough as it stands to be able to encode in a direct form a range of possible linguistic approaches to derivational and inflexional morphology. Furthermore, it is quite within the spirit of our enterprise that the mechanisms we have provided should be regarded, not as the final word in linguistic formalisms, but as a kind of 'assembly language' into which different, more abstractly stated proposals could be 'compiled'.

1. Spelling Rules

To cope with the various spelling/phonological changes that occur when morphemes are concatenated, we use a variant of the 'two level' formalism originally due to Koskenniemi (1983a,b). (For a short, clear exposition see Gazdar 1985). The basic idea behind this approach is to regard graphological/phonological rules as statements of possible correspondences between underlying and surface forms, in common with most segmental level phonological formalisms. The difference is that in the two level model, no intermediate levels of representation are allowed: thus the formalism is in many ways much more constrained than traditional phonological theories, which typically allow a sequence of rules to successively transform an underlying to a surface representation via many intermediate stages.

Our own description is concerned with graphemic rather than phonemic representations, as we are dealing with printed text. However, this is not an inherent limitation of the formalism, which has been used to implement non-trivial phonological descriptions, notably of Finnish (Koskenniemi, op. cit.)

An example of a two level rule is the following, which describes the process by which an additional e is inserted when some nouns or verbs are suffixed with the morpheme +s (notice that the morpheme boundary symbol is here regarded as part of the lexical entry for the affix):

```
Epenthesis
+:e <=> { < s:s h:h > s:s x:x z:z } --- s:s
                    or < c:c h:h > --- s:s
```

The epenthesis rule states that 'e' must be inserted at a morpheme boundary if and only if the boundary has to its left 'sh', 's', 'x', 'z' or 'ch' and to its right 's'. The interpretation of the rule is simple; character pairs (representing 'lexical character:surface character') form the actual change (the relation of a lexical '+' to a surface 'e') in the contexts to the right of the arrow. Braces ({ , }) indicate

disjunction and angled brackets indicate a sequence. Alternative contexts may also be specified using the word 'or'. Thus a paraphrase of the rule might be 'if there is a lexical + preceded by 'sh', 's', 'x', 'z', or 'ch' and followed by 's', then surface 'e' must appear, preceded by 'sh', 's', 'x', 'z', or 'ch' and also followed by 's', and vice versa'.

As well as ⇔ there are also the operators ⇒ (if) and ⇐ (only if). The interpretation of these is:

```
a:b => LeftContext _ RightContext
```

'If you've got a:b, then you must have LC ... RC' - so the pair a:b will never appear in other contexts, but other pairs can appear in this context.

```
a:b <= LeftContext _ RightContext
```

'If you've got LC ... RC, and lexical a, then you must have a:b' - so a:b could appear in other contexts, but no other pairs with lexical a can appear in this context. The ⇔ operator combines these two, so:

```
a:b <=> LeftContext _ RightContext
```

means 'if you've got a:b, you must have LC ... RC, and vice versa'. Most rules use the ⇔ operator, in practice.

Lexical and surface strings of unequal length can be matched by using the null character '0'. A possible rule matching things like lexical 'move+ed' to surface 'moved' contains two occurrences:

```
e:0 <=> C:C --- < +:0 V:V >
```

The context in this rule is not specified solely in terms of concrete character pairs, but rather with the symbols 'C' and 'V', which can be declared as standing for the set of consonants and the set of vowels. This mechanism allows arbitrary (preferably natural) classes of segments to be defined and thenceforward referred to by a single identifier. This rule can be paraphrased 'surface 'e' will match with '0' in the lexicon (i.e. always match) when there is a consonant to its left and a boundary symbol followed by a vowel to its right'.

The computational interpretation of this rule formalism is as follows. Firstly, the list of the base lexical forms of morphemes are (at compile time) made into a character tree (cf Thorne, Bratley and Dewar 1969) of the form:

```
...
sea     N
sell    Vtrans
she     Pro
shell   N
shore   N
...
```

```
etc.

        /o_r_e_*
       / /-l-l-*
....s-h-e-*
     \e-a-*
       \l-l-*
```

where the -* represents the leaves of the tree containing the syntactic or other information stored with the base form. Secondly, the two level rules are compiled into finite state transducers. A finite state transducer is best thought of as a familiar finite state automaton, but which has two symbols on each transition, which are inspected simultaneously. Thus two inputs are being inspected in parallel, and the tranducer keeps them in step. Alternatively, one of these symbols can be regarded as the input, and one the output. (In some cases, this choice may be arbitrary, as the transducer will be reversible.) A move from one state to another can be made only if both symbols are present in the two inputs, or under the alternative construal, if the input symbol is present, then the transition can be made, emitting the output symbol.

The compilation method used, and the resulting transducers, are different from those described by Koskenniemi, and are similar to those in Bear (1986) (although developed independently). Informally (and vastly simplified), what happens in this phase is as follows:

First we find the set of 'feasible pairs' - all pairs of *lexical:surface* characters that occur in the description, pairs of identical *lexical:surface* characters, and any that have been specially declared as feasible pairs. These are used in expanding out the values of declared set identifiers, and effectively constitute the alphabet of the final automaton.

For rules of the form 'pair $\Rightarrow$ LC ... RC' we construct a machine that recognises sequences of the form 'LC pair RC'.

For rules of the form 'pair $\Leftarrow$ LC ... RC' we construct a machine that recognises sequences of the form 'LC *lexical:* $\neg$ *surface* RC' - i.e. the set of sequences with the appropriate contexts and lexical character but the wrong surface character.

For $\Leftrightarrow$ rules we construct two automata like the preceding ones and merge the left context part (which will be identical). For all three types various markings are put on states of the automata to reflect whether they constitute provisional acceptance (i.e. 'ok so far, but we may still need to pass some other tests', rejection (i.e. recognition of an invalid sequence), or successful arrival at a final state.

Finally, transitions are added for all the feasible pairs that are not restricted by any contextual requirements. The latter will all be marked 'provisionally accept', and 'final'.

As an example, the transducer corresponding to the Epenthesis rule above is (roughly):

	e:e	s:s	h:h	x:x	z:z	+:e	+:0	
s1	s2	s2,s3		s3	s3			
s2			s3					
s3						s4	s6	
s4		s5						prov acc
s5								final
s6								reject

In fact, these transducers are also combined and optimised in various ways such that the actual machines used may look rather different. The process of attempting to recognise a word is now one of successively matching characters in the input to the appropriate tape of the transducer, and simultaneously matching the other tape of the transducer to the lexical character tree. We try to make the input match one or more paths through the lexical tree via the transducer. The transducer is restarted at each transition, and so we are in several states simultaneously most of the time (the driving algorithm keeps track of this non-determinism) so that unlike the Koskenniemi version it is not necessary in the transducer itself to put in transitions going back to the initial state (this avoids much complexity in the machines and the compilation process). At each stage in the matching process the following must be true: at least one state we are in must be a provisionally accepting state, and no state must be a rejecting state. When we reach a final state in an automaton it is discarded from the current set of states. If we get to the end with no provisional states pending we have a successful match and a record of the transitions through the lexicon tree will constitute the set of valid segmentations of the input string.

In Koskenniemi's original system, there was a main lexicon, the leaves of which pointed not just to information associated with a morpheme, but also to other distinct sub-lexicons (and so on). Thus at the end of a free morpheme we might have a pointer to the lexicon containing the class of suffixes that could follow it. This has the effect of making all the inflexional and derivational morphology implicit in the arrangement of sublexicons, and carries with it the claim that this type of system is formally strictly finite state, a claim we reject, for English, at least. In our system all morphemes are stored in one tree, and if there is some input still to be consumed we simply start off again at the root of the tree. This process is actually interleaved with the 'word-syntax' phase of analysis (see below), in order to cut down the search space. If the segmentation were not partially guided by these higher level requirements the result would be that this phase of analysis would deliver a set of sequences of morphemes representing the different possible ways a given sequence of input characters can be segmented into morphemes without regard for their syntactic properties. Lack of syntactic constraint can mean that the numbers here get rather large. For example, in our current description there are over 400 ways of segmenting the word 'assassin' into

distinct sequences of valid English morphemes, (e.g. ass-ass-in, as-s-a-s-sin, etc) although only one of these corresponds to a valid word.

One great advantage of the use of transducers for this type of analysis is that they are reversible. Thus our system, as well as being able to segment e.g. a word like 'denied' into 'deny+ed' is also able to generate 'denied' from an input sequence 'deny+ed'.

2. Deficiencies in the two level approach.

There are several areas where the two level approach as it stands is known to be inadequate. One - the finite state nature of derivational morphology - we have already pointed to (in connection with Koskenniemi's original system). Secondly, it is difficult, though not impossible, to represent syllable based prosodic phenomena directly, as the formalism is essentially biased towards a segmental view of phonology. To some extent, this can be got round by employing special 'archiphoneme' or other abstract characters that characters in the lexical forms of words, which trigger various rules, but we have eschewed this method in our own description of English, as being both open to abuse (abstract phonology) and practically undesirable (precluding the use of existing machine readable dictionaries as part of our system).

A practical problem that we have encountered even within the relatively simple domain of English graphemics concerns unexpected rule interactions. Consider the two rules:

```
A-to-B

a:b  <=>      c:c _ d:d
           or c:c _ e:e

F-deletion

f:0  <=>   c:c a:b _
```

The sequence permitted by the latter rule constitutes a context that the A-to-B rule does not mention. Thus the pair a:b will never be allowed here and so the F-deletion rule will fail ever to apply. The only way around this is to add another context to the A-to-B rule to allow for exactly this possibility:

```
...
or  c:c _ f:0
```

It is not clear whether this type of thing should be regarded as stupidity on the part of the rule writer, or a failure of the formalism. In practice, when rules have far more complicated contexts than the artificial ones above, it happens quite often and is very difficult to diagnose. One possible solution would be to include checks

in the compilation algorithm to detect this type of clash and either attempt to resolve them, or issue appropriate warnings. (In the compiler developed at CSLI for the D-Kimmo system, something similar to this has in fact been done. (Lauri Karttunnen, p.c.)). In general, the fact that the rules cannot be looked at in isolation, in the sense that it is the conjunction of them that defines the set of possible transductions, makes debugging of them very difficult.

Phenomena involving discontinuous morphemes, as in the Semitic languages, are impossible to handle with any degree of elegance in the basic formalism, though a brute force solution is technically available. However, an extension of the basic formalism so as to use transducers with more than two tapes is possible. One tape will represent the consonantal root, a second the vocalic morpheme, and the third the surface form combining the two. The resulting system is no longer two level phonology, strictly speaking, but it is still finite state phonology (Kay 1987).

Another problem area concerns reduplication, which can be handled only clumsily, at best. It remains to be seen whether Kay's suggestions can be used here as well.

The assumption that we can always (as it were) do phonology first, then morphology, is questionable for some languages (Anderson, this volume). If the data and analysis presented by Anderson withstand scrutiny, this is a serious problem for the framework (as for many others), and it is not clear how it could naturally be extended to cope.

Barton, Berwick and Ristad (1987) have pointed out that the two level formalism is NP-hard (i.e. - very informally - we can encode apparently simple problems in the formalism which would take more time than there has ever been to solve). If unrestricted null characters are allowed, the situation is even worse (PSPACE hard). It is not clear what to make of this result, since in practice the descriptions of various languages made within this formalism do not give rise to time or space problems. It is perhaps best taken as an indication that more structure is required in the theory to preclude the possibility of any formulation of the (linguistically) unnatural problems that are used to demonstrate NP properties. Since no other current theory has ever been formalised enough for such results to be forthcoming, comparisons are not possible. However, since many early phonological notations were reminiscent of context-sensitive rules, or explicitly claimed to be so, it may be of interest to know that context-sensitive recognition is also PSPACE hard (op. cit. 1987:64). And a cursory inspection of some of the notations offered as phonological formalisms in recent work does not inspire any optimism that their computational properties (if discoverable) will be particularly tractable - quite the reverse, in fact. Thus two level morphology does not appear to be dismissable solely on that account.

3. Lexical Entries

The actual entries for morphemes are stored in a simple format:

```
(citation-form, phonological-forms(s), syntactic-category,
                                semantic-entry, miscellaneous)
```

An example from the GPSG description of English might be

```
(walk /walk/ ((V +) (N -) (SUBCAT NULL)) WALK NIL)
```

The citation forms can of course be phonological representations if there is an appropriate set of two level rules available. The syntactic category must obey the conventions of GPSG categories (see below), but this is no inconvenience, as a powerful system for defining aliases is available, in effect allowing almost anything sensible to appear as a syntactic category. No restrictions are placed on the semantic field: currently our entries contain the base form for most open class items, and a complex expression of intensional logic for most closed class items. The 'miscellaneous' field is currently the Lisp atom NIL: the intention is that a user can put whatever other information is required here. It might, for example, be the entire content of some existing machine readable dictionary's entry for 'walk'.

These entries can, if desired, be added to, or used to generate new entries, by using two types of 'lexical redundancy' rules, which operate at compile time. 'Completion rules' add feature specifications to existing lexical entries. 'Multiplication rules' create additional entries from existing ones. For example, the entry for 'walk' (which represents the infinitival or base form) is fleshed out with many other features by Completion rules:

```
(walk /walk/ ((V +) (N -) (CAT V) (FIX NOT) (INFL +) (REG +)
              (AGR ((BAR |2|) (V -) (N +) (NFORM NORM))) (PRD -)
              (COMPOUND NOT) (BAR |0|) (AUX -) (INV -) (NEG -)
              (FIN -) (VFORM BSE) (SUBCAT NULL))
     WALK NIL)
```

From infinitival forms of verbs are created entries for 1st and 2nd person singular, and plural forms (this is necessary given the GPSG feature system, in which negation and disjunction of feature specifications is not available). Here is the 2nd person singular, to which Completion rules have also applied:

```
(walk /walk/ ((VFORM NOT) (FIN +) (INFL -) (PAST -)
            (AGR ((N +) (V -) (BAR |2|) (NFORM NORM) (CASE NOM)
            (PER |2|) (PLU -))) (V +) (N -) (CAT V) (FIX NOT)
            (REG +) (PRD -) (COMPOUND NOT) (BAR |0|) (AUX -)
            (INV -) (NEG -) (SUBCAT NULL))
     WALK NIL)
```

Those familiar with GPSG will notice that some features are there which do not figure in syntax: e.g. REG and COMPOUND. These are used by the word grammar rules to govern regular inflection and compound formation respectively, and would be stripped out before the entry for a word was handed to a sentence level parser.

The Completion and Multiplication rules mechanism allows concise entries to be written and fleshed out automatically. They are simple and flexible, and it is easy to implement a wide range of theories of lexical redundancy or feature defaults using them.

4. Word Grammar

During the application of the spelling rule transducers, the results of morpheme separation are recorded in a chart (Thompson and Ritchie 1984) and an attempt is made to find a valid parse of the sequences of morphemes according to the Word Grammar of the system. The Word Grammar is a set of rules describing possible combinations of morphemes. The formalism within which these rules are expressed is basically that of a context-free grammar enriched with features. A feature consists of a pair (name, value). A category consists of a set of features. A name is an atom; a value can be a constant, a variable, or a category. A rule consists of a sequence of two or more categories, the first being the mother. Various easily readable notations are provided so that these rules can be entered in a perspicuous form.

Two methods of matching a category in a rule with a category in a lexical item or parse tree are provided (this is an option in compiling the source code of the actual system). The first, known as 'graph unification' is similar to that used in systems such as PATR II (Shieber 1985). To compute the graph unification (U) of two categories:

> if (f v) is in one cat and not in the other, (f v) is in U
> if (f v) is in one cat, and (f v') is in the other,
> then
>
> > if v=v' and both are atomic, (f v) is in U
> > if v is a variable, then (f v') is in U, (where v is consistently replacable by v')
> > if v and v' are categories, and graph unify, then (f V) is in U, where V is the graph unification of v and v'

(A variable is consistently replacable by some value if it has not already been, and is not going to be, replaced by a conflicting value while computing the unification.) The categories will not unify if any features do not fall under this definition.

Graph unification is order independent, in the sense that the ordering of features within a category is irrelevant. It also does not require fully specified categories. Both properties distinguish it from the second type of matching, known as 'term unification'. Here two categories will unify only if they have the same set of features, appearing in the same order, where the only difference between them is that one feature might have a variable as value where the other has a constant (or a different variable). In the latter case the variable becomes bound (if this can be

done consistently, as before). This type of matching is based on directly on the notion of unification familiar from theorem proving and some logic programming languages.

Ultimately, there is no formal difference between these two types of unification, but they differ in the styles of grammar writing they facilitate. Graph unification encourages a style whereby rules and categories can be underspecified so as to apply across a wide range of cases. However, some system of defaults may then be needed in order to rule out unintended matches. For example, if nothing further is said, a category like:

```
((cat det)(wh -))
```

will graph unify with a category like

```
((n +)(bar 2)(num sing)(count +))
```

(there are no conflicting feature values),
to yield

```
((cat det)(wh -)(n +)(bar 2)(num sing)(count +))
```

which may well not be what the grammarian had intended. The two categories would not term unify, however, for obvious reasons.

The system incorporates several mechanisms for declaring default values for features, (as well as those already seen) or default specifications for categories which can be used so as to cause categories like those above not to unify. However, keeping track of the effects of default feature specifications in a large linguistic description can be tricky in practice. Thus under some circumstances it is preferable to use the term unification option, where all categories are fully specified, and debugging is, in general, easier ('what you see is what you get'). There are also some implementational advantages to term unification: for example, we know in advance that two categories of different lengths cannot unify, and for many grammars, we can omit feature names in the internal representations of categories, identifying values instead by their position, with consequent gains in efficiency. Needless to say, each style of unification has proponents who refuse to see any virtue in the other: both camps are catered for by our system.

5. Feature-Passing Conventions

The system also contains the option of using some 'hard-wired' feature passing conventions along with the unrestricted unification mechanism, (although they are not strictly necessary, as the mechanism of variable instantiation within rules can be used to pass the values of features around within a tree in a completely declarative and semantically well-founded fashion). Those in current use are tied in closely with the Generalised Phrase Structure Grammar treatment of morphology which is the main description of English provided by the system.

There are three conventions built into the system at present, inspired by the work of Selkirk (1982). Notice that the definitions of the feature passing conventions themselves are not under the control of the lexicon-writer, although the features that are affected by the conventions may be controlled, since the conventions act on certain specific features or feature-classes, so the linguist can make use of these conventions by defining certain features to lie within these named classes. The system will then automatically apply the conventions to these features.

All three feature conventions act on what is called within GPSG terminology a 'local tree': a set of one mother node and its immediate daughters. The conventions apply only to binary branching rules. They are written in terms of mother, left daughter, and right daughter. The conventions are:

> The Word-Head Convention: the set of WHead features in the mother should be the same as the WHead features of the right daughter.

In the word parser, this is achieved, roughly speaking, by unifying the WHead features of the right daughter and those of the mother when the daughter is attached. From a linguistic point of view, the WHead features are typically those that will be relevant to sentence-level syntax, and hence those that will be of particular use to a sentence-parser which uses the dictionary. This convention is a straightforward analogue of the simplest case of the Head Feature Convention (Gazdar et al) (1985)). Its effect is to enforce identity of the relevant feature values between mother and the head daughter. Note that in the current system there is no formal definition of 'head' to which the lexicon-writer has access (despite the name given to this convention), since the right daughter always acts in this head-like fashion within our treatment of English morphology. Other analyses may deviate from this pattern, of course; different views of head may be implemented using variables and unification.

Assuming that N, V and Vform are in the set of WHead features, the Word-Head Convention would allow the following trees:

```
((N +) (V -) (PLU +))
        ()
        ((BAR -1) (N +) (V -) (PLU +))
```

and

```
((N -) (V +) (VFORM ING))
        ((N -) (V +))
        ((BAR -1) (N -) (V +) (VFORM ING))
```

but not trees of the form:

```
((N +) (V +) (PLU +))
        ()
        ((BAR -1) (N +) (V -) (PLU +))
```

and

```
((N -) (V +))
        ()
        ((BAR -1) (N -) (V +) (VFORM EN))
```

since one of the trees has a clash in the V value for mother and right daughter, and the other lacks a VFORM marking on the mother to match that on the right daughter.

> The Word-Daughter Convention: if any WDaughter features exist on the right daughter then the WDaughter features on the mother should be the same as the WDaughter features on the right daughter. If no WDaughter features exist on the right daughter then the WDaughter features on the mother should be the same as the WDaughter features on the left daughter.

Again, this is ensured by carrying out unification of the appropriate feature markings during parsing. This convention is designed to capture the fact that the subcategorization class of a word (in English) is not affected by inflectional affixation, although it may be affected by derivation.

Assuming the feature TAKES to be the only WDaughter feature, this convention allows trees such as:

```
((TAKES NP))
        ((V +) (N -))
        ((TAKES NP))
((TAKES NP))
        ((TAKES NP))
        ((VFORM ING))
```

but not

```
((TAKES NP))
        ((V +) (N -))
        ((TAKES VP))
((TAKES NP))
        ((TAKES VP))
        ((VFORM ING))
```

In the first example the right daughter is specified for a TAKES value, and the mother has the same specification; in the second example, the right daughter has no specification for TAKES and so the second part of the WDaughter convention applies. The third example is illegal because the values of TAKES on the right daughter and mother differ, and the fourth is illegal because, under the second part of the convention, the left daughter and mother WDaughter features must be identical when there are no WDaughter features in the right daughter.

The Word-Sister Convention: when one daughter (either left or right) has the feature STEM, the category of the other daughter must be an extension (superset) of the category value of STEM.

This third convention enables affixes to be subcategorized for the type of stem to which they attach. Notice that this convention is not defined in terms of any feature-classes, but is defined using just one built-in feature (STEM). Hence, the way that the lexicon-writer makes use of this convention is not by declaring the extent of feature-classes (as for the other two conventions), but by adding STEM specifications to the features in morphemes in the lexicon, thereby indicating the combination possibilities for each affix. The following example trees follow the convention

```
()
     ((N -) (V +))
     ((STEM ((N -) (V +))))
()
     ((V +) (N -) (INFL +))
     ((STEM ((N -) (V +) (INFL +))))
```

These various feature-passing conventions allow very general rules to be written for affixation, when used in combination with other mechanisms which can declare valid ranges for variables, and the variable binding mechanism itself. The current description, for example, has a single rule for the attachment of prefixes to stems.

```
( PREFIXES
   [BAR 0, CAT ?maj] ->
         [FIX PRE, CAT NONE],
         [BAR 0, CAT ?maj] )
```

and a single rule for the attachment of suffixes to stems.

```
( SUFFIXES
    [BAR 0, CAT ?maj] ->
          [BAR 0, CAT ?cat],
          [FIX SUF, CAT ?maj] )
```

In both a suitable declaration of possible values for the ?cat and ?maj variables prevents unwanted categories appearing as stems, and the feature conventions, unrestricted unification and variable binding make sure that information is passed up from the daughter categories to the mother in the appropriate way. (A term unification description would have to explicitly list all the possible features for each category). It is instructive to compare the succinctness of the rule with the size of the categories (like those for 'walk' above') to which it will apply.

6. Summary

We have described a system for morphological analysis which seems to us a reasonable compromise between linguistic elegance and practical usefulness. The various grammatical formalisms provided with the system, with the possible exception of the two level rules, do not themselves constitute a linguistic theory in their own right. The correct way to think of them is as part of a linguistically motivated programming language within which quite a wide range of approaches to morphological description (or, indeed, syntax) could be implemented. The feasibility of this has been demonstrated with our GPSG description of English, which is a fairly full treatment of inflexion and derivation, faithful to the principles of GPSG. This system analyses complex words at the rate of three or four per second under optimal conditions (on a 4 mb Sun 3). We also have a much simpler description of English delivering only major category information. This parses words many times faster: between 10 and 20 per second. Thus the system is fast enough to be practically useful in real applications.

Acknowledgement

This work was carried out under SERC/Alvey Grant GR/C/79114.

References

Anderson, S. 1987 paper in this volume

Barton, E., Berwick, R., and Ristad E. (1987) Computational Complexity and Linguistic Theory, Cambridge, Mass: MIT Press

Bear, J. (1986) A morphological recogniser with syntactic and Phonological Rules, Proceedings of 11th International Conference on Computational Linguistics, Bonn, West Germany: 272-276

Gazdar, G. (1985) Finite State Morphology, Linguistics, 23, 597-607

Gazdar, G, Klein, E., Pullum, G. K., and Sag I. A. (1985) Generalized Phrase Structure Grammar , Oxford, Blackwells.

Karttunen, L. (1983) KIMMO - A General Morphological Processor, in Texas Linguistic Forum 22 , 165 - 186) Department of Linguistics, University of Texas, Austin, Texas.

Kay, M. (1987) Nonconcatenative Finite State Morphology, Invited Lecture, Association for Computational Linguistics, European meeting, Copenhagen.

Koskenniemi, K. (1983a) Two-level model for morphological analysis, in Proceedings of the Eighth International Joint Conference on Artificial Intelligence, Karlsruhe , 683 - 685.

Koskenniemi, K. (1983b) Two-level Morphology: a general computational model for word-form recognition and production, Publication No. 11, University of Helsinki, Finland

Koskenniemi, K. (1985) Compilation of Automata from Two-level Rules, talk given at the Workshop on Finite-State Morphology, CSLI, Stanford, July, 1985.

Ritchie, G.D., Pulman, S.G., Black, A.W. and Russell G.J. (forthcoming) A Computational Framework for Lexical Description, Computational Linguistics.

Selkirk, E. (1982) The Syntax of Words, Cambridge, Mass: MIT Press.

Shieber, S. (1985) Criteria for designing computer facilities for linguistic analysis, Linguistics 23, 189-211.

Thompson, H. and G.D. Ritchie (1984) Implementing Natural Language Parsers, in T. O'Shea and M. Eisenstadt (eds.) Artificial Intelligence: Tools, Techniques and Applications, New York: Harper and Row.

Thorne, J.P., P. Bratley, and H. Dewar (1968) The syntactic analysis of English by machine, in D. Michie (ed). Machine Intelligence 3, Edinburgh: Edinburgh University Press.