A New Version of the Machine Translation System LMT

M. C. McCORD
IBM Thomas J. Watson Research Center, NY, USA

Abstract

This paper describes a recent revision of the machine translation system LMT in which (*a*) source analysis is based on Slot Grammar, and (*b*) there is a large language-independent portion of the system, a kind of 'X-to-Y translation shell,' making it easier to handle new language pairs. Slot Grammar makes a systematic use of *slots* (essentially syntactic relations) obtained from lexical entries for head words of phrases. No phrase structure rules (augmented or plain) are used. Instead, there are *slot filler rules* and separately stated *ordering rules* for slots. A great deal of the Slot Grammar system is in the shell. This includes most of the treatment of coordination, which uses a method of 'factoring out' unfilled slots from elliptical coordinated phrases. The parser (a bottom-up chart parser) employs a parse evaluation scheme used for pruning away unlikely analyses during parsing as well as for ranking final analyses. The transfer step is designed so that all of the transfer rules except those arising from lexical transfer entries (which are database-like) are in the shell. Syntactic generation uses a system of transformations, written in a formalism involving an extension of Prolog unification. The revision includes a new treatment of transformation rule ordering. LMT is implemented entirely in Prolog.

1. Overview

LMT[1] began as an English–German MT system, (McCord, 1986, 1988*a*: McCord and Wolff, 1988), in which source analysis was based on Modular Logic Grammar (McCord, 1982, 1985, 1987). Recently, LMT has been revised in two ways:

1. Source analysis is now based on Slot Grammar (McCord 1980, 1989).
2. An effort has been made to maximize the language-independent portion of the system, making a kind of 'X-to-Y translation shell', and to prove the viability of the shell by developing prototype versions for several language pairs.

Analysis with a Slot Grammar is based on a systematic use of *slots* (essentially syntactic relations, like subj and obj), obtained from lexical entries for head words of phrases. No phrase structure rules (augmented or plain) are used. Instead, there are *slot filler rules* and separately stated *ordering rules* for slots. There is a high degree of lexicalism in Slot Grammar, making grammars simpler and having also a simplifying effect on the rest of the MT system. Also, the differences between source grammars for various languages are reduced because of the modular treatment of ordering.

The original Slot Grammar system (McCord, 1980) was developed in 1976–8 without consideration of logic programming (and was implemented in Lisp).[2] The Modular Logic Grammar system, used in the earlier design of LMT, represents a *combination* of the original Slot Grammar techniques with the (augmented) phrase structure grammar techniques common in logic programming (Colmerauer, 1978), and employs top-down parsing. In this combined approach, Slot Grammar rules, expressed in terms of phrase structure rules and Prolog clauses, are used systematically for postmodification of open-class words; but elsewhere in the grammar more standard phrase structure rules are used.

Recently (McCord 1989) the original ('pure') Slot Grammar system was redone in a logic programming framework, with several improvements, and the rest of the LMT system was adapted to this method of source analysis. As with the original Slot Grammar system, a bottom-up chart parser is employed. Aside from greater theoretical neatness, the motivation for this shift was to deal better with multiple source languages.[3] This is facilitated because of the characteristics of Slot Grammar mentioned above and because of a good fit with the idea of the shell, as will be explained below.

The overall design of LMT has not changed in the recent revisions in terms of the basic steps of translation. The system is transfer-based and currently deals with only one sentence (or input phrase) at a time. There are five steps (passes) in translating a sentence:

1. Source/transfer lexical processing.
2. Source syntactic analysis (with Slot Grammar).
3. Transfer.
4. Target syntactic generation.
5. Target morphological generation.

Two themes run through the design of these five steps.

One theme is to do as much as possible by syntactic means alone. This shows up most strongly in source analysis. As mentioned, the slots forming the basis of Slot Grammar analysis are syntactic slots; no semantic slots (like agent or patient) are used in the system. On the whole, words are considered to have different *senses* (for source analysis) only when they have different morpho-syntactic features or different slot frames. Very little use is made of semantic features or semantic type-checking in source analysis (although this is done in transfer). Instead, there is a strong use of syntactic heuristics in parsing, to be described below.

The output of source analysis (and input to transfer) is basically a syntax tree. It is a dependency tree, showing the modifiers of each head word as slot/filler pairs in surface order. (Each filler is again a dependency tree.) On the other hand, the analysis tree shows deep grammatical relations through unification of logical variables

Correspondence: Michael C. McCord, IBM Thomas J. Watson Research Center, P. O. Box 704, Yorktown Heights, NY 10598, USA.

associated with slots and their fillers--including remote dependencies, logical relations in passive constructions, and implicit subjects in non-finite verb phrases. In particular, predicate/argument structure for open-class words is shown. In fact, the system can produce logical forms in the language LFL (McCord, 1987) from these syntax trees, and the main step in doing so is to decide on scoping of (generalized) quantifiers ('focalizers').

It was decided to use the syntax trees instead of logical forms for transfer because (*a*) there is useful information in the syntax trees, such as morphosyntactic features and surface order, (*b*) it is hard to decide scoping accurately on a broad scale, and (*c*) the syntax trees do have much of the content of logical form anyway, as indicated above.

The second theme in the five steps of translation is to make as much as possible language-independent, i.e., to put as much as possible in the shell. Every one of the five steps involves modules from the shell. This theme is strongest in source analysis and transfer: Slot Grammar is more special-purpose to natural language than many grammatical formalisms. A lot is packed into the parser module, which, however, is largely independent of specific natural languages and so is part of the shell. For example, the parser module contains most of the treatment of coordination (which is basically a metagrammatical phenomenon), although this treatment depends on some language-specific data given in the Slot Grammar for a specific language.

All of the transfer system except the transfer portion of the lexicon is in the shell. The transfer tree has basically the same shape as the source analysis tree, but contains target-language words and features. It also contains many of the markings necessary for target surface structure, such as the correct target prepositions for prepositional complements.

The shell has facilitated the development of prototype versions for other language pairs, in cooperation with other groups and individuals in IBM, namely: English--Danish and Danish-English (Arendse Bernth and IBM European Language Services), English-French (the KALIPSOS team of the IBM Paris Scientific Center (Fargues *et al.*, 1987), with work especially by Eric Bilange), German-English (Ulrike Schwall, IBM Germany Scientific Center), and English-Spanish (Nelson Correa).

The remaining four sections of the paper deal with the treatment of the first four steps of translation in LMT named above. Target morphological generation has undergone little revision in the new version of LMT, and is largely described in previous references cited.

2. Source/transfer lexical processing

In the current design of LMT, there is only one lexicon for each language pair, the *source/transfer lexicon*. It is indexed by citation forms of source language words, and each entry contains *source elements* and *transfer elements*. The source elements are readily separable (and could form the basis of the lexicon for a different target language), but the transfer elements refer to information in the various source elements for a source word.

Some sample entries in an English-German lexicon (not meant to give the full story for the words involved) are as follows:

```
add  < v(obj.pobj(to))
     < tv(obj.iobj,hinzu:fueg).
eat  < v(obj)
     < tv(subj:~human,obj,fress)
     < tv(obj,ess).
view < v(obj)
     < n(nobj)
     < tv(obj,be+tracht)
     < tn(nobj,ansicht.f.n).
```

These entries are actually Prolog unit clauses for the infix predicate <, which is also used to separate the elements of the entry.

The entry for *add* has a source element (on the first line) showing *add* as a verb with an object slot obj and a *to*-PP complement slot pobj(to). The subject slot subj is added by default to every verb. The next line has a transfer element showing the separable-prefix verb *hinzufuegen* as the target verb (tv), with slots obj and iobj (indirect object) corresponding to the two slots in the source element. The entry for *eat* illustrates the use of semantic types in target selection, and will be discussed in Section 4. The entry for *view* shows this word as both a verb and a noun, with corresponding transfer elements. The English nobj (noun object) slot is filled by an *of*-PP. The corresponding German nobj-complement takes the genitive case or *von* plus dative, depending on the noun phrase. Note that there is target-language morphological information in these transfer elements. For instance, in the *view* entry, the German verb is shown to be an inseparable-prefix verb, and the noun's gender and declension class are specified.

The ingredients of lexical entries most consequential for the rest of the system are the slot lists shown in the entries. The slots are all *complement* slots, and the ones shown in source elements are important input to the Slot Grammar. *Adjunct* slots depend only on the part of speech of the word, and are declared in the grammar itself, as we will see below.

There is a thorough treatment of multiwords in the lexical format (and processing). Multiword entries are allowed for all of the parts of speech. For example, *take care of* can be handled as a multiword indexed under *take*, and the associated morphology permits inflected forms like *took care of*. Also, prepositional object slots can specify 'multiprepositions', allowing forms like *put up with*, and particle slots can specify 'multiparticles', allowing forms like *take X into consideration* (where *into consideration* is the multiparticle). Semantic type conditions can be specified in lexical entries, but the discussion of these is postponed to Section 4 below, because they are used mainly in transfer.

The lexical processing step actually involves two substeps, performed for each word in the input string:

1. Source/transfer morphological analysis,
2. Lexical compiling.

The first substep produces from the input word (possibly inflected or derived), a *derived* lexical analysis for the word, where the elements show morphological structure together with source or transfer elements of the type already discussed. The second substep then translates the derived lexical analysis elements into Prolog clauses

usable by source analysis and transfer. Both substeps involve modules from the shell.

The morphological analyser consists of a general rule interpreter (in the shell) and a collection of language-dependent, low-level morphological rules. Given a derived (or inflected) input word, the analyser first strips off affixes until a base word is found in the lexicon, and then the affixes *operate* on the lexical analysis of the base (acting on both source elements and transfer elements), producing finally a derived analysis for the input word. In the process of affix stripping, a state transition system is used for constraints. For details, see McCord and Wolff 1988.

After the (possibly derived) lexical analysis of a (possibly derived) word Word is obtained, the lexical compiler translates each source element of this analysis into one or more clauses for the predicate

wordframe(Word,Sense,Features,SlotFrame).

Each such clause defines a *word analysis* for Word, consisting of the last three arguments.

The *sense* (Sense) of the word analysis is normally the citation form of Word, but can be a particular sense name (like give1) if such is specified in the source element, or can show morphological structure in the case of a derived word.

The *feature structure* (Features) of the word analysis is a logic term (using standard positional notation, not attribute/value notation) basically giving part-of-speech and inflectional information for Word. The part of speech is the principal functor. But Features can serve as the feature structure for phrases obtained by modifying Word, so that in some cases Features has arguments that have to do with the whole phrasal configuration. As an example, the allowable feature structures for verbs in the English Slot Grammar ESG are

verb (inf(Full))
verb (prespart)
verb (pastpart)
verb (pastparta)
verb (fin(Pers,Num,Tense,Cl))

Here Full is preinf or bare according as the infinitive is premodified by *to* or not. The feature pastpart signals the common past participle with a passivized frame, and pastparta signals the active past participle used with the perfect *have*. The last argument Cl of the finite verb feature (fin) is used to express clause-level features such independent/dependent.

The *slot frame* (SlotFrame) of the word analysis is the list of complement slots in *internal form*. Each internal-form slot is of the form slot(Slot,Ob,X), where Slot is the slot name (like obj, appearing in the external lexical analysis), Ob indicates whether the slot is obligatory or not, and X is the *marker* of the slot. When the slot is filled by a phrase, the slot marker X becomes bound to an identifier associated with the filler phrase, in a way to be described below.

As an example of a word analysis, the following is produced for word *adds* from the v element for add above:

wordframe(adds, add, verb(fin(pers3,sg,pres,*)),
slot(subj,op,X).slot(obj,op,Y).slot(pobj(to),op,Z).nil).

The main part of the treatment of passive verb constructions occurs during the formation of word analyses. A passive form of a verb, such as an English past participle or a Danish finite passive form, leads to one or more word analyses in which the slot frame is altered to show the appropriate passive slots. For more details, see (McCord, 1989), or for an earlier treatment, (McCord, 1982). The internal form of transfer elements is described below in Section 4.

In the case of multiwords, the internal Prolog predicate for source elements is slightly different from the above, showing the boundaries of the multiword in the input word string.

Putting both source elements and transfer elements in the same lexicon is not crucial to the design of LMT; it is mainly a matter of indexing. In fact, ESG is interfaced to the monolingual English UDICT lexicon (Byrd, 1983, 1986, Byrd *et al.* 1986) (which has around 60,000 lemmas), and there is a separate (partially usable) interface to a lexical data base system for the Collins English–German dictionary, being developed by Neff *et al.*, (1988). The UDICT system has its own morphology, representation of features, and storage and access method; the main point of the interface is to produce word analyses in the form described above.

3. Source analysis with Slot Grammar

Analysis with Slot Grammar is word-oriented. Phrase analyses for word strings are built up by beginning with word analyses of the individual words (produced by lexical processing) and growing larger phrases by attaching modifier phrases on the left and the right. Such attachment is controlled completely by slot filling for slots associated (normally) with the original word analyses.

The representation of a phrase analysis is slightly more complex than that of a word analysis because, for instance, it must show modifiers (daughters) of the phrase. But parsing starts by forming for each word analysis an initial phrase analysis having no modifiers and having the word as head.

Specifically, a *phrase* structure is represented by a term

phrase(X,Sense,Features,SlotFrame,Ext,Mods),

where the components are as follows:

The first component X is a logical variable called the *marker* of the phrase. It is used as a kind of identifier of the phrase, and plays a role in slot filling, to be explained below.

The next three components Sense, Features, and SlotFrame have the same form as the three components of a word analysis. In the initial phrase structure formed from a word analysis, these three components *are* just the word analysis. In the formation of larger (modified) phrases, they may be changed by unification, and in some grammars they are changed *only* by unification (this is currently the case in ESG); but the grammar rules do allow feature structures to be changed.

The component Ext is a list of *extraposed* slots (in internal form), used for later filling by extraposed

phrases. In the initial phrase for a word analysis, this component is nil.

The last component Mods represents the modifiers (daughters) of the phrase, and is of the form mods(LMods,RMods) where LMods and RMods are the lists of *left modifiers* and *right modifiers*, respectively. Each member of a modifier list (left or right) is of the form Slot:Phrase where Slot is a slot (complement or adjunct), and Phrase is a phrase which fills Slot according to the given Slot Grammar rules. Modifier lists reflect surface order, and a given (adjunct) slot may appear more than once. In the initial phrase formed from a word analysis, Mods is mods(nil,nil).

Markers of phrases and markers of slots are used together with unification to show links of modification in phrase structures. When a slot slot(Slot,Ob,X) is filled, the slot marker X is bound to the term e(X0), where X0 is the marker of the filler phrase—except that when the filler is a PP, X0 is taken to be the marker of the object of the preposition. Whenever a phrase P fills an adjunct slot for a phrase H, the markers of P and H are unified, except when P is a noun phrase or a verb phrase.[4] Such binding of markers is important for lexical control in transfer, as we will see in Section 4, and is also important for building logical forms.

In displaying parse trees, it is convenient to do the following. The recursive problem is to display a pair consisting of a slot and a phrase structure. (For a top-level analysis, the slot is taken to be the symbol top.) First we display the slot and feature structure on a line. Then the left modifiers are recursively displayed, indented. (Recall that each member of a modifier list is a slot/phrase pair.) Then the *sense predication* of the phrase (to be defined) is displayed on a line. Finally, the right modifiers are recursively displayed, indented.

The *sense predication* of a phrase is a term whose principal functor is the (head word) sense, and whose arguments are as follows. The first argument is the marker of the phrase. The remaining arguments are obtained from the markers of the slots in the slot frame, in order. Specifically, for each slot slot(Slot,Ob,X) in the frame, if the slot is filled and X=e(X0), then the argument is X0; otherwise the argument is an unbound variable. In one view of logical form, the initial marker argument in a verb sense predication is like an 'event variable', and for a noun, it is the main variable for the noun sense (corresponding to the entity referred to by the NP), like X in man(X) or brother(X,Y).

We give an example of a parse tree, using this display format. The example will be continued in Sections 4 and 5 below to illustrate transfer and syntactic generation as well. The sentence is:

The user adds new lines to the file.

Recall that the word analysis for *adds* was given in the preceding section. The English grammar ESG and the parser produce the following single parse:

```
top verb(fin(pers3,sg,pres,X2))
  subj noun(cn,sg,nwh)
    ndet det(sg,def)
      the(X3)
      user(X3)
    add(X1,X3,X4,X5)
    obj noun(cn,pl,nwh)
      nadj adj(X9,X10)
        new(X4)
      line(X4)
    pobj(to) prep(to,X7,e(X5))
      to(X6,X5)
      objprep(X8) noun(cn,sg,X7)
        ndet det(sg,def)
          the(X5)
        file(X5)
```

Note, for instance, that the phrase marker X3 for *the user* is identified as the subject of *adds* (the second argument of the *add* sense predication in the display). The phrase marker X4 of *new* is unified with that of *lines*, because nadj is an adjunct slot. The last argument X5 of *add* (associated with its pobj(to) slot) is unified directly with the marker of *the file* since the PP *to the file* is a complement.

A more complex example is the following. For the sentence

Who did the old man try to find and sit with?

a single parse results:

```
top verb(fin(pers3,sg,past,ind:q:wh))
  obj noun(pron(wh),X5,wh)
    who(X2)
  do1(X1,X3,X4)
  subj noun(cn,sg,nwh)
    ndet det(sg,def)
      the(X3)
    nadj adj(X9,X10)
      old(X3)
    man(X3)
  auxcmp(inf(bare)) verb(inf(bare))
    try(X4,X3,X7)
    infcmp verb(inf(full))
      preinf preinf
        preinf(X7)
      lconj verb(inf(full))
        find(X7,X3,X2)
      coord(and,find,sit)
      rconj verb(inf(full))
        sit(X7,X3)
        vprep(3) prep(with,nwh,e(X2))
          with(X7,X2)
```

Interesting linkings of markers: (1) The marker X2 for *who* is unified with the object markers of *find* and *with*. This results from the treatments of extraposition and coordination in the system, discussed below. (2) The marker X3 for *man* is unified with the subject marker of *do1*, as well as the subject markers of *try*, *find*, and *sit*. There is a treatment of implicit subjects in non-finite verb phrases, described in (McCord, 1989).

The remainder of this section is divided into two subsections, *The Slot Grammar formalism* and *The parser module*. In writing a Slot Grammar for a language, one should be generally aware of the representation of phrase structures, because grammar rules can refer to the components of phrases. However, there are special rule

formalisms that make this easy to do in an abbreviated way.

3.1. *The Slot Grammar formalism*

The main ingredients of a Slot Grammar are the following:

1. A declaration of *adjunct slots* for each part of speech.
2. *Slot filler rules.*
3. *Slot ordering rules.*
4. *Obligatory slot rules.*

In addition, there are certain language-specific data (expressed mainly as unit clauses) for treating extraposition, co-ordination, and punctuation. Much of the treatment of these constructions, however, is in the language-independent parser module.

Let us look briefly at the four main ingredients listed above. There are special rule formalisms and rule compilers for rules of types 2, 3, and 4.

Adjunct slots are declared simply by unit clauses

```
adjuncts(POS,Adjuncts)
```

where POS is a part of speech (like verb) and Adjuncts is the list of possible adjunct slots for all words (or phrases) of that part of speech. (The part of speech of a phrase is the principal functor of its feature structure.) Unlike a complement slot, which may be obligatory and can be filled at most once, an adjunct slot is optional and can be filled any number of times.

Slot filler rules are the core of a Slot Grammar, constituting the main rules for modification of one phrase by another. More specifically, given a phrase P, one chooses an *available* slot Slot for P, i.e. either an unfilled (complement) slot in the slot frame or extraposed slot list of P, or an adjunct slot associated with its part of speech. A phrase M adjacent to P will be a *filler* of Slot (and a modifier of P) if there is a filler rule

```
Slot→Body
```

for which Body holds. The condition Body has the same form as the body (antecedent) of a Prolog clause, but it can *contain special* goals which refer to the components of the higher phrase P or the modifier phrase M (which are implicit in the use of the rule).

An example of a filler rule for the subject slot in English might be

```
subj→f(noun(*,nom,Pers,Num)) &
      hf(verb(fin(Pers,Num,*,*))).
```

Here the special goal f(F) (or hf(F)) requires that F is the feature structure of the filler phrase (or the higher phrase).

Most special goals arise from selector predicates for the phrase data structure. For instance the special goals f(F) and hf(F) arise from the predicate f(P,F) which selects the features F of a phrase P, and which is defined simply by the unit clause

```
f(phrase(*,*,F,*,*,*), F).
```

For each such selector predicate pred(P,...), one gets a pair of special goals pred(...) and hpred(...) referring to the (implicit) filler phrase and higher phrase, respectively.

The rule compiler converts a filler rule to a Prolog clause in which the filler phrase and the higher phrase are mentioned explicitly. The predefined special goals are compiled specially, and all other goals in the body of the rule are compiled to themselves. For details of the rule compiler, see (McCord 1989).

There are two types of slot ordering rules, (1) *head/slot ordering rules*, expressing ordering of slots (their fillers actually) with respect to the head word, and (2) *slot/slot ordering rules*, expressing ordering of slots with respect to other slots.

Head/slot ordering rules are of either of the forms:

```
lslot(Slot) ← Body.
rslot(Slot) ← Body.
```

These rules say respectively that Slot is a *left slot* (or *right slot*), under the conditions of Body. The condition Body (which may be omitted, with the arrow) may, like the body of a filler rule, contain special goals referring to the modifier phrase or the higher phrase. For example, the rule

```
rslot(subj) ← hf(verb(fin(*,*,*,ind:q:*))) &
              hsense(Verb) & finaux(Verb).
```

for English says that subj can be a right slot in a question sentence (feature q) if the verb is a finite auxiliary.

A slot/slot ordering rule is of the form:

```
LSlot≪RSlot ← Body
```

where the Body may be omitted. This means that every filler for LSlot must precede every different filler for RSlot under the conditions of Body. Again, the body can contain special goals referring to the higher phrase or to the filler phrases for LSlot or RSlot.

Examples of slot/slot ordering rules are the following ones, expressing relative ordering of the direct and indirect objects of verbs:

```
iobj≪obj ← lf(noun(*,*,*)).

obj≪iobj ← rf(prep(*,*,*)).
```

Special goal predicates prefixed with l (r) refer to the slot on the left (right) of the operator ≪. These rules say that the indirect object precedes or follows the direct object according as the indirect object is a noun phrase or a prepositional phrase.

A phrase that is to be an allowable top-level analysis must be *satisfied* in the sense that all of its *obligatory* slots are filled. Also, for a phrase to become a modifier of another phrase, all of its obligatory slots, after possible extraposition of one of its slots, must be filled.

A complement slot may be specified in the lexicon to be obligatory, in which case its internal form will be slot(Slot,ob,X) (the second argument is op otherwise). The default is that slots are optional.

One may also specify obligatoriness of slots by general rules in the grammar. Such rules are of either of the forms:

```
obl(Slot) ← Body.
obl(Slot,Slot1) ← Body.
```

The first rule says that Slot is obligatory under the conditions of Body, and the second says that Slot must be filled if Slot1 is filled and Body holds. The body again can contain special goals. Examples:

```
obl(subj) ← hf(verb(fin(*,*,*,*))).
obl(objprep).
obl(obj,iobj) ← f(noun(*,*,*)).
```

The last rule says that the direct object must be filled if the indirect object is filled by a noun phrase.

As indicated above, the treatment of left extraposition is divided between the (language-dependent) grammar and the (language-independent) parser module. There are two ingredients in the grammar dealing with extraposition.

1. One declares that certain slots allow extraposition of other slots *out of* their fillers, by writing certain unit clauses that mention these slots. The parser module takes care of storing extraposed slots in the extraposed slot list component of the phrase structure.
2. The slot on the left-hand side of a filler rule is normally just a slot name, but it can have the following special form, which indicates that the rule is an *extraposed slot filler rule*:

```
ext(Slot,Level).
```

Here Slot is a slot name and Level is ext if Slot is actually extraposed (taken from the extraposed slot list), or norm if Slot is a normal slot (taken simply from the slot frame). The latter case is needed for examples like the relative clause *who Mary saw*, where *who* fills a normal object slot. Extraposed filler rules are currently used in ESG to handle *wh*-phrases and relative pronouns in *wh*-questions and relative clauses.

Most of the treatment of coordination is in the parser module. In a grammar, however, there are two ingredients dealing with coordination. First, there is a specification of coordinating conjunctions as well as associated preconjunctions, like *both* and *either*, given in unit clauses. (These are not listed in the lexicon because of their special nature.) Secondly, there is a specification of the coordination of feature structures through clauses for the predicate

```
coordfeas(Conj,LFeas,RFeas,Feas).
```

This says that when a phrase having features LFeas is conjoined by Conj with a phrase having features RFeas, then the result has features Feas. The default is that all three feature structures are the same, but variations are allowed, for instance in dealing with agreement features in noun phrases.

3.2. *The parser module*

The parser is a bottom-up, left-to-right chart parser. Partial analyses are stored as Prolog unit clauses:

```
result(LB,RB,Eval,State,Phrase)
```

where the arguments are as follows. The last argument is a phrase structure analysing a portion of the input string, and LB and RB are its left and right boundaries, represented as integers. Eval is the *parse evaluation*, which will be discussed below. State is used for recording what kind of modifiers (left, right, or extraposed) Phrase has received. In building up phrases, we choose first to attach normal (non-extraposed) left modifiers, then normal right modifiers, and finally extraposed left modifiers. Specifically, State is 0 if the phrase has no right or extraposed modifiers, but may have left modifiers; State is 1 if the phrase has some right modifiers, but no extraposed modifiers; and State is 2 if the phrase has some extraposed modifiers.

In the following, a *result* (or *partial analysis*) is a quintuple of terms

```
(LB,RB,Eval,State,Phrase)
```

appearing as the argument list of a result clause.

In the left-to-right parsing, when a new word is encountered, the system looks at each word analysis of the word and does the following.[5] The initial phrase for the word analysis (as described at the beginning of Section 3) is constructed, and then the initial result is formed and stored for this phrase and the word's position. This result has state 0, and the initial parse evaluation argument is formed, as described below. The new result is then *combined*, if possible, with every adjacent result to the left, and any combination results are stored and further combined recursively.

After processing all words in the input, the final parses are those satisfied phrases in results that span the whole input.

Combining of two adjacent results to give a new result involves (1) *modifying* the phrase of one by the phrase of the other, (2) computing the new parse evaluation from the old ones, as discussed below, (3) setting the new state according to the meaning of 'state' given above, and (4) setting the new phrase boundaries appropriately. Step (1) is non-deterministic; the modification could be in either direction, and in general one phrase can modify another in more than one way because of multiple choices of available slots.

When a phrase M is to modify a phrase P, giving a new phrase P1, the steps are as follows:

1. Choose an available slot Slot for P (as defined in Section 3.1 above).
2. Apply a filler rule (in compiled form) for Slot, M, and P.
3. Check ordering constraints, for normal slot filling (not done for extraposed filling).
4. Apply extraposition from M if possible (not done for extraposed filling).
5. Check that M is satisfied (modulo slots extraposed from it).
6. Bind markers (as described above).

Of course the new phrase P1 is obtained then from P by adding the new modifier M, possible new extraposed slots, and possibly a new feature structure.

As mentioned above, most of the treatment of coordination is in the parser module. The system analyses coordinated phrases of the following form (with some variations):

LM Preconj LC Conj RC RM

where the substrings indicated are as follows. Conj is a coordinating conjunction or a punctuation symbol (like a comma) used in the capacity of a coordinating conjunction. Preconj is an optional associated preconjunction. LC and RC are the *left* and *right conjuncts*, respectively. Each of these conjuncts consists of a single phrase, although it need not be satisfied. LM and RM are the (optional) *left* and *right common modifiers*, respectively (each of these may be represented by several phrases). Examples are:

The man	sees	and	probably hears	the car.
LM	LC	Conj	RC	RM

John sees	and	Mary hears	the car.
LC	Conj	RC	RM

The syntax tree produced by ESG for the last sentence is:

```
top verb(fin(X2,X3,X4,X5))
  lconj verb(fin(pers3,sg,pres,X7))
    subj noun(prop,sg,nwh)
      john(X8)
    see(X1,X8,X6)
  coord(and,see,hear)
  rconj verb(fin(pers3,sg,pres,X10))
    subj noun(prop,sg,nwh)
      mary(X11)
    hear(X1,X11,X6)
  objcmp noun(cn,sg,X9)
    ndet det(sg,def)
      the(X6)
    car(X6)
```

Note that *the car*, with marker X6, is shown as the common object of *sees* and *hears*.

The head of the coordinated phrase is basically the conjunction, but is actually a compound term showing also the heads of the conjuncts. The feature structure of the coordinated phrase is obtained from the feature structures of the conjuncts by using coordfeas, given in the grammar (mentioned above). The two conjunct phrases fill the slots lconj and rconj.

The interesting part of building the coordinated phrase is the definition of its available slots (besides lconj and rconj), which can be filled by the left and right common modifiers. Of course the adjunct slots are determined from the coordinated feature structure, which has been described.

The coordinated slot frame is obtained by a process of 'factoring out' common (or closely similar) unfilled complement slots from the frames of the two conjuncts. This method was outlined in (McCord, 1980) and was implemented in the revised Slot Grammar system (McCord, 1989). In the above example, the objcmp slot (which can be filled by a noun phrase or a *that*-clause) is common to the frames of *see* and *hear* and is factored out, to become a slot for the coordinated phrase.

Before doing the factoring, the extraposed slot list of each conjunct is pooled together with its normal slot frame. This is appropriate because of examples like that of Woods (1973):

John drove his car through and completely demolished a plate glass window.

Here the objprep slot of the preposition *through* is extraposed to the level of *drove* in the left conjunct *drove his car through*. In the 'factoring', this objprep is considered a common slot with the obj slot (not extraposed) for the right conjunct *completely demolished*. The preceding example also illustrates that in the process of factoring out slots, we must be ready to consider (unfilled) slots LSlot and RSlot of the conjuncts to produce a common slot even when LSlot and RSlot are not exactly the same (as in the case of objprep and obj in the example). What we need is a kind of 'g.c.d' of the two slots. We assume a predicate

coordslot(LSlot,RSlot,Slot)

which can produce from LSlot and RSlot the 'g.c.d' slot Slot which is the factored out version.

With this said, we can state when the system succeeds in producing a factored-out frame Frame from the frames (including any extraposed slots) LFrame and RFrame of the left and right conjuncts:

1. Whenever an unfilled slot LSlot of LFrame can be paired by coordslot with an unfilled slot RSlot of RFrame, their markers are unified, and their Ob components are suitably combined. The resulting (factored out) slot is made a member of Frame.
2. Any unfilled slot of LFrame or RFrame that is not paired as in (1) must be optional (where we consider verb subjects obligatory).

We have discussed the main ideas in forming coordinated phrase structures. For more details, including the way coordination fits into the overall workings of the parser, see (McCord, 1989). The main idea of the latter though is just that the *combining* of two results, as described above, can involve coordination of phrases as well as ordinary modification.

The parser module contains a treatment of various kinds of constructions involving tokens that are not words. These include brackets and separators. *Brackets* are paired symbols like parentheses, braces, square brackets, dashes, and quotes that can surround word strings. Brackets also occur very frequently in source text for text formatting languages, for example in font-change commands and special symbols for the other types of brackets just mentioned. *Separators* are symbols like commas and hyphens that can lie between a phrase and one of its modifiers. The use of such tokens has a basically metagrammatical nature, like coordination, so it is reasonable for the treatment to be in the parser module. However, some relevant language-specific data (in the form of unit clauses) are put in the grammar. For details, see (McCord, 1989).

The parser module includes a parse evaluation scheme used for pruning away unlikely analyses during parsing as well as for ranking final analyses. The parse evaluator expresses (weighted) preferences for close attachment.

for complement modification over adjunct modification, and for parallelism in coordination. The results of evaluation for complement preference and close attachment combined (not counting parallelism) have some similarity to the results of the preference methods of (Wilks *et al.*, 1985). Parse space pruning may be turned on or off optionally. When it is on, it is fairly common to get only one parse which is correct modulo attachment of postmodifying adjuncts. When it is off, one gets all the parses allowed by the grammar, but they are ranked as to preference. The parse evaluator is based on a partial order betterthan defined on the parse space, i.e. on the set of all results. If, during parsing, results R1 and R2 are found such that R1 is betterthan R2 (and pruning is turned on), then R2 will be discarded.

In addition, an equivalence relation similarfeas is defined on the parse space, expressing broad similarity of feature structure in results. Results can be related by betterthan only when they are in the same equivalence class, so that parse pruning is done independently in each equivalence class. Within an equivalence class, betterthan is a total order, based on numerical scoring. The component of the numerical scoring function that controls for close attachment is Heidorn's (1982) parsing metric (in its simplest form). The components dealing with complement preference and parallelism dominate over the close attachment component.

The relations betterthan and similarfeas on results are defined basically in terms of the parse evaluation component of a result, mentioned above. This component is a term of the form:

eval(FeatureScheme,Score).

The *feature scheme* is a term that encodes a certain abstraction from the features involved in the phrase of the result, and the *score* is a real number representing a score for how well the phrase satisfies the preferences described above (a lower score is a better score). Our two relations on results are then defined as follows:

(LB1,RB1,eval(F1,S1),St1,Ph1) similarfeas
(LB2,RB2,eval(F2,S2),St2,Ph2) if and only if
LB1 = LB2, RB1 = RB2, and F1 = F2.

(LB1,RB1,eval(F1,S1),St1,Ph1) betterthan
(LB2,RB2,eval(F2,S2),St2,Ph2) if and only if
LB1 = LB2, RB1 = RB2, F1 = F2, and S1 < S2.

The system actually has different options for defining the feature scheme of a result. The default method is that the feature scheme is the term

head(F,I,J)

where I and J are the boundaries of the head word and F is obtained from the feature structure Feas as follows: In most cases, F is the principal functor of Feas, but if Feas is verb(Infl) then F is the principal functor of Infl. So, when the feature scheme is computed in this default way, pruning is done within classes of results having the same boundaries, the same head word, and (roughly) the same part of speech.

The numerical scoring function is a weighted sum with terms dealing (as indicated above) with close attachment, complement preference, and parallelism. The term controlling for close attachment is, as mentioned, Heidorn's metric. (This term is between 0 and 1.) The term controlling for complement preference is basically just a count of the number of adjunct slots in the phrase structure (on all levels). However, one can declare any adjunct to have another contribution to the score than its count of 1. Thus an adjunct declared to have contribution 0 is valued like a complement. There are actually three (weighted) terms dealing with parallelism in coordination. These measure similarity of feature structures, similarity of slot frames, and similarity of modifier configurations for the two conjuncts. For details, see (McCord, 1989).

4. Transfer

Transfer is facilitated by marker binding, which shows links between words and their complements, as well as links between two phrases in adjunct modification, as described in the preceding section.

The links associated with complementation are especially important. Given a source word W, a filled complement slot of W is of the form slot(Slot,Ob,e(X)), where the logical variable X is the marker of the filler phrase. This link offers communication in two directions, from complement to head word and vice versa:

1. The nature of the complement phrases can affect the choice of the translation of the head word W.
2. A given transfer element can name the target slots associated with the complements of the target head word. These can be different from the corresponding source slots, especially when prepositional complements are involved.

Note that the complements of a word may not be located on the same level as the word itself (they may not be found among the modifier lists of the phrase for which the word is head) because of extraposition and coordination. Nevertheless, marker binding shows such connections.

To make the above communication possible, there is a simple preparatory step for transfer, in which the markers of noun phrases and verb phrases in the source analysis tree become (through unification) *augmented markers*, which are of the form

X:Sense:Features

where X is an unbound variable, Sense is the head word sense of the phrase, and Features is the feature structure. We call X the *augmented marker variable*.

Augmented markers allow communication in direction (1) above, in that a word can easily 'look at' the sense and features of a complement. And communication in direction (2) is possible through binding the augmented marker variable to a target slot.

Communication in both directions is implemented through references to augmented markers in the internal forms of lexical transfer elements. Such internal forms are (possibly conditional) clauses for the predicate

twordframe(POS,SourceWord,Args,TargetWord).

(We give a slightly simplified description, omitting one argument.) POS is the part of speech which the source

word is considered to have, and the source word itself is given in the same form as the Sense argument of a wordframe clause (usually a citation form). The target word is given in citation form, although it can be a compound word. The argument Args is a list of augmented markers, corresponding to the complements of the word.

As an example, suppose we have the English–German entry for *eat* given in Section 2:

```
eat < v(obj) < tv(subj:~human,obj,fress)
            < tv(obj,ess).
```

The idea of the two transfer elements is that *eat* translates into *fressen* if the subject is not marked *human*, else it translates into *essen*. In addition, the target slots are named. Then for an input word *eats* these three lexical analysis elements give rise to the following three internal-form clauses, respectively:

```
wordframe(eats, eat, verb(fin(pers3,sg,pres,*)),
                  slot(subj,op,X).slot(obj,op,Y).nil).
twordframe(verb, eat, (subj:S:*).(obj:*:*).nil, fress)
            ← ~isa(S,human).
twordframe(verb, eat, (subj:*:*).(obj:*:*).nil, ess).
```

Suppose we are translating the sentence *The man eats the apple*, and *man* is marked *human* in its lexical entry. Lexical compiling creates a clause:

```
isa(man,human).
```

Source analysis, and the formation of augmented markers mentioned above, create the bindings

```
X = e(X0:man:noun(...))
Y = e(Y0:apple:noun(...))
```

of the markers of the subj and obj slots of *eats*. The transfer algorithm (as we will see below), in dealing with the transfer of *eats*, calls twordframe with its third argument equal to the list of terms U where e(U) is a marker of a slot of *eats* (in order); i.e. the third argument given to twordframe is

```
(X0:man:noun(...)).(Y0:apple:noun(...)).nil.
```

Application of the first clause for twordframe binds S to man, so the antecedent of this clause fails. But then the second clause succeeds, and this produces the *essen* translation and also creates the bindings X0 = subj and Y0 = obj. These variables are directly available in the phrase structures of *the man* and *the apple*, and in their translations. Hence the translated NPs can be assigned the subj and obj slots and the correct target cases.

In general, transfer elements can indicate any Boolean combination of tests on a complement, and the lexical compiler converts these into a corresponding combination of Prolog goals. Each test in the original combination can be indicated by a simple semantic type (like *human*), which is converted into an isa goal for the sense of the complement, but the test can also involve the feature structure of the complement.

The main, recursive procedure

```
tran(SourceSlotFiller,MotherFeatures,
     TargetSlotFiller)
```

for transfer takes a source slot/filler pair and the target features corresponding to the mother of this pair, and produces a target slot/filler pair. Target phrase structures, instead of being represented as phrase data structures, are represented in a somewhat simpler form:

```
syn(HeadWord,Features,LeftModifiers,
    RightModifiers).
```

The members of the modifier lists are pairs Slot:Filler, where Filler is either another syn structure or a term representing a word. These syn structures are referred to directly in the rules of syntactic generation.

The steps (basically) in the definition of tran are the following four:

1. Find the target slot TSlot. For this, one can refer to the source slot Slot and the augmented marker variable. TSlot is taken to be Slot if Slot is an adjunct; else TSlot is the augmented marker variable if this is bound (by a transfer element); else TSlot is 'zero' if Slot is a PP complement slot; else TSlot is Slot.
2. Find the features of the target phrase. In doing this one can refer to the source features, the source slot, the target slot, and the mother target features.
3. Find the target head word by a procedure

   ```
   tranword(TargetFeas,SourceWord,
            X.SourceFrame,TargetWord).
   ```

   Here the term X is the augmented marker of the source phrase.
4. Recursively call tran on the modifier slot/filler pairs, using the current target features as the middle argument.

All of the rules for steps (1), (2) and (4), and the top-level rules for step (3), are in the shell. There are some non-trivial things to do in step (2) (feature transfer), getting features in a form suitable for generation and managing the communication up and down the tree through the 'mother features' argument. The language-independent top level of word transfer (tranword) is also non-trivial, involving rules for handling passives, subject verb agreement, etc.

The basis of the procedure tranword is of course to call the language-specific lexical transfer predicate twordframe described above (this is done for all but exceptional tokens).

In the preceding section, the source analysis tree for the sentence

The user adds new lines to the file.

was displayed. Recall also the English-German entry for *add*, discussed in Section 2:

```
add   < v(obj.pobj(to))
      < tv(obj.iobj,hinzu:fueg).
```

In the English-German version of LMT, the transfer step produces the following tree:

```
top verb(ind:top,fin(pers3-sg,pres,X2):X3,nil)
  subj noun(cn,nom,pers3-sg-X1:pers3-sg-m,X4)
    ndet det(nom,pers3-sg-m,X4)
      d
```

```
  benutzer/a
hinzu:fueg
obj noun(cn,acc,pers3-pl-X5:pers3-pl-f,X6:a)
  nadj adj(X7,acc,pers3-pl-f,X6:a)
    neu
  zeile/l
zero prep(zero,nil)
  0
  iobj noun(cn,dat,pers3-sg-X8:pers3-sg-f,X9)
    ndet det(dat,pers3-sg-f,X9)
      d
    datei/m
```

Because of the basic compositionality of the transfer algorithm, this tree is isomorphic to the source tree. However, it has the appropriate target language words in citation form (the German nouns are marked with their declension classes), and the correct target features. Notice that the correct target slots are assigned to the complements of the target verb, as dictated by the tv element above, such as the iobj slot for der *Datei* (the translation of *to the file*).

Restructuring the tree to get an appropriate German surface tree is the job of the syntactic generation step, which is discussed next.

5. Syntactic generation

Syntactic generation applies a system of tree transformations to the transfer tree in order to get a target tree in the correct target surface form. (Actually, transformations apply to slot/filler pairs.) The algorithm for applying the transformations, as well as some supporting procedures for writing transformations, are language-independent, but the transformations themselves can depend both on the source and target languages. However, it is possible to share transformations for different language pairs.

There is a special formalism which allows one to write transformations using an extension of Prolog unification involving sublist variables. A rule compiler converts transformation rules to Prolog clauses (McCord 1986, 1988*a*).

The external form of a transformation is:

```
Name –
  A — →B
  ← Condition.
```

Here Name is the name of the transformation, and A and B are slot/tree pairs, where the tree is a syn structure, as described in the preceding section. The modifier lists in the syn structure can contain sublist variables, which are represented in the form %X. The Condition is a Prolog goal, and it can be omitted if desired.

A transformation rule is compiled into a Prolog clause of the form:

```
transform(Name,A1,B1)←
  ASplit & Condition & BSplit.
```

Here the original term A involving per cent variables has been re-expressed as an ordinary term A1 and a conjunction ASplit of calls to conc, which concatenates lists. Similarly, B is re-expressed as B1 and BSplit. For efficiency, the Condition is inserted between ASplit and BSplit, because Condition normally contains constraints whose arguments become known immediately after execution of ASplit.

As an example, in English-German translation, a simplified version of a dative movement transformation can be written as follows:

```
dative–
S:syn(V,verb,LM,%RM1.(obj:Obj).(iobj:IObj).RM2)
— →
S:syn(V,verb,LM,%RM1.(iobj:IObj).(obj:Obj).RM2).
```

This rule interchanges the direct and indirect objects. It is compiled into the transform rule:

```
transform(dative, S:syn(V,verb,LM,RM0),
  S:syn(V,verb,LM,RM4)) ←
  conc(RM3, (obj:Obj).(iobj:IObj).RM2, RM0) &
  conc(RM3, (iobj:IObj).(obj:Obj).RM2, RM4).
```

Note that conc is used both for splitting apart lists (nondeterministically) and for putting lists together.

Transformations are specified in two ordered lists. The members of the lists are the transformation *names*, appearing as first arguments of transform. The first list, of *b-transforms*, consists of transformations that can apply only to non-coordinated phrases ('b' suggests 'basic'). The second list, of *c-transforms*, consists of transformations that can apply to any phrases (possibly coordinated) ('c' suggests 'coordinated'). Generally, there are many more b-transforms than c-transforms.

The algorithm for restructuring a slot/filler pair S:P by the application of transformations is as follows:

1. Recursively restructure all the modifiers of P, and let P1 be the phrase resulting from P by replacing its modifiers with their restructured versions.
2. If P1 is not coordinated, run through the b-transform list, applying each transformation if possible, only once (by calling transform), starting with S:P1 as input and getting final result S2:P2.
3. Run through the b-transform list, applying each transformation if possible, only once, starting with S2:P2 as input. The final result is the restructured version of S:P.

For more discussion of transformations, see (McCord, 1986, 1988*a*). The basic techniques are similar to those in this earlier version of LMT, but new ingredients are (1) the separation of transformations into b-transforms and c-transforms, and (2) the treatment of ordering. In the earlier version, Prolog clause ordering was used, and a given transformation could apply more than once on a given level. Also in the new version it is quite convenient to be able to refer to slots as well as features.

Let us finish by illustrating what happens with the example sentence *The user adds new lines to the file* discussed in the last two sections. The changes needed in the transfer tree (displayed in the preceding section) are (1) to get rid of the zero remnant of the source preposition (this is to be ignored because the PP is a complement), (2) to interchange the obj and iobj complements, and (3) to move the separable prefix *hinzu* to the end.

These changes are performed, respectively, by three

transformations pzero, dative, and sepprefix. These are in the b-transform list, ordered as just indicated. The transformation pzero is trivial, replacing the zero PP by its one right modifier. This prepares the way for dative, already described above, which interchanges the obj and iobj modifiers.

Finally, sepprefix operates. This is defined by:

```
sepprefix–
S:syn(Pre:Verb,verb(ind:T,I,M),LMods,RMods)
— →
S:syn(Verb,verb(ind:T,I,M),LMods,
  %RMods1.Pre.RMods2)
  ← vfsplit (RMods,RMods1,RMods2).
```

The reference to ind requires that the verb phrase be independent. The separable prefix Pre is moved *toward* the end, but may not go all the way. The purpose of vfsplit is to place Pre so that it does not hop over final 'heavy' modifiers (see discussion in (McCord 1986, 1988*a*)). It is important in general that sepprefix is ordered after dative, because if the indirect object has a heavy final modifier, this should *not* impede the movement of the prefix toward the end.

The result of the operation of restructuring is the tree:

```
top verb(ind:top,fin(pers3-sg,pres,X2):X3,top)
  subj noun(cn,nom,pers3-sg-X1:pers3-sg-m,X4)
    ndet det(nom,pers3-sg-X1:pers3-sg-m,X4)
      d
    benutzer/a
  fueg
  iobj noun(cn,dat,pers3-sg-X5:pers3-sg-f,X6)
    ndet det(dat,pers3-sg-f,X6)
      d
    datei/m
  obj noun(cn,acc,pers3-pl-X7:pers3-pl-f,X8:a)
    nadj adj(X9,acc,pers3-pl-f,X8:a)
      neu
    zeile/l
  hinzu
```

Then the morphological generation step produces the final result:

Der Benutzer fuegt der Datei neue Zeilen hinzu.

Notes

1. The abbreviation is for 'Logic-based (or Logic-programming-based) Machine Translation.' The system is written entirely in Prolog.
2. Background influence on this work consisted mainly of the following two lines of work: (*a*) Systemic Grammar, especially the work of Richard Hudson (1971, 1976) (see also McCord, 1975, 1977), and (*b*) the work of George Heidorn (1972, 1975) based on the augmented phrase structure grammar formalism now called PLNLP. It is interesting that Hudson's 1971 system specified linear ordering relationships (in terms of grammatical relations) independently of immediate dominance relationships. Hudson's systems were not formulated in a computational framework. Heidorn's system had the similarity with Slot Grammar of being dependency-oriented and using grammatical relations, but the augmented phrase structure grammar basis makes for several differences. In the current scene, Slot Grammar is probably closest in spirit to grammatical systems based on unification of feature structures, such as FUG, LFG, and HPSG (*see* (Shieber, 1986), for an overview, also cf. Dependency Unification Grammar (Hellwig, 1986, 1988)); but there are differences that will become evident in the following. Also, the original 1976–8 Slot Grammar work was done independently of any work of this type.
3. Execution speed was not particularly a consideration. The efficiency of the English Modular Logic Grammar (with top-down parsing through Prolog execution) and the efficiency of the English Slot Grammar (with bottom-up chart parser) seem to be about the same.
4. The unification of markers in these cases is inappropriate for the needs of LMT and for other reasons. Details will not be given here.
5. We describe here the normal parsing process, neglecting at first the treatment of constructions like coordination.

6. References

Byrd, R. J. (1983). 'Word Formation in Natural Language Processing Systems', *Proceedings of IJCAI-VIII*, 704–6.

—— (1986). 'Dictionary Systems for Office Practice', IBM Research Report RC 11872, T. J. Watson Research Center, Yorktown Heights, New York.

—— J. L. Klavans, M. Aronoff, and F. Anshen (1986). 'Computer Methods for Morphological Analysis,' *Proceedings of the Association for Computational Linguistics*, 120–7.

Colmerauer, A. (1978). 'Metamorphosis Grammars,' in L. Bolc (ed.), *Natural Language Communication with Computers*. Springer-Verlag.

Fargues, J., Bérard-Dugourd, A., Landau, M. C., Nogier, J. F., and Catach, L. (1987). 'KALIPSOS Project: Conceptual Semantics and Linguistics,' *Proc. of the Conf. on Artificial Intelligence and Natural Language Technology*, IBM European Language Services, Copenhagen.

Heidorn, G. E. (1972). 'Natural Language Inputs to a Simulation Programming System', Technical Report NPS-55HD72101A, Naval Postgraduate School, Monterey, California.

—— (1975). 'Augmented Phrase Structure Grammars', In B. L. Nash-Webber and R. C. Schank (eds.), *Theoretical Issues in Natural Language Processing*, 2–5, Association for Computational Linguistics.

—— (1982). 'Experience with an Easily Computed Metric for Ranking Alternative Parses', *Proceedings of Annual ACL Meeting, 1982*, 82–4.

Hellwig, P. (1986). 'Dependency Unification Grammar,' *Proceedings of the 11th International Conference on Computational Linguistics, 1986*. Bonn, 195–8.

—— (1988). 'Chart Parsing According to the Slot and Filler Principle', *Proceedings of the 12th International Conference on Computational Linguistics, 1988*, 242–44. Budapest.

Hudson, R. A. (1971). *English Complex Sentences*, North-Holland, Amsterdam.

—— (1976). *Arguments for a Non-Transformational Grammar*, University of Chicago Press, Chicago.

McCord, M. C. (1975). 'On the Form of a Systemic Grammar', *Journal of Linguistics*, **11**, 195–212.

—— (1977). 'Procedural Systemic Grammars,' *International Journal of Man-Machine Studies*, **9**, 255–86.

—— (1980). 'Slot Grammars', *Computational Linguistics*, **6**, 31–43.

—— (1982). 'Using Slots and Modifiers in Logic Grammars for Natural Language', *Artificial Intelligence* **18**, 327–67.

—— (1985). 'Modular Logic Grammars', *Proceedings 23rd*

Annual Meeting of the Association for Computational Linguistics, 104–17, Chicago.

—— (1986). 'Design of a Prolog-based Machine Translation System,' *Proceedings of the Third International Logic Programming Conference*, 350–74, Springer-Verlag, Berlin.

—— (1987). 'Natural Language Processing in Prolog', in Walker *et al.* (1987).

—— (1988*a*). 'Design of LMT: A Prolog-based Machine Translation System', *Computational Linguistics*, **15**, 33–52.

—— (1988*b*). 'A Multi-Target Machine Translation System', *Proceedings of the International Conference on Fifth Generation Computer Systems 1988*,1141–9, Institute for New Generation Computer Technology, Tokyo, Japan.

—— (1989). 'A New Version of Slot Grammar', Research Report RC 14506, IBM Research Division, Yorktown Heights, NY 10598.

—— and Wolff, S. (1988). 'The Lexicon and Morphology for LMT, a Prolog-based MT system', Research Report RC 13403, IBM Research Division, Yorktown Heights, NY 10598.

Neff, M. S., Byrd, R. J., and Rizk, O. A. (1988). 'Creating and Querying Lexical Data Bases', *Proceedings of the Second Conference on Applied Natural Language Processing*. Austin, Texas.

Shieber, S. M. (1986). *An Introduction to Unification-Based Approaches to Grammar*, CSLI Lecture Notes No. 4, Center for the Study of Language and Information, Stanford, CA.

Walker, A. (ed.), McCord, M., Sowa, J. F., and Wilson, W. G. (1987) *Knowledge Systems and Prolog: A Logical Approach to Expert Systems and Natural Language Processing*. Addison-Wesley, Reading, Mass.

Wilks, Y., Huang, X-M., and Fass, D. (1985). 'Syntax, Preference and Right-Attachment', *Proceeding 9th International Joint Conference on Artificial Intelligence*, 779–84. Los Angeles.

Woods, W. A. (1973). 'An Experimental Parsing System for Transition Network Grammars', in R. Rustin (ed.), *Natural Language Processing*, 111–54. Algorithmics Press, New York.