Design of LMT: A Prolog-Based Machine Translation System

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

LMT (logic-based machine translation) is an experimental English-to-German MT system, being developed in the framework of logic programming. The English analysis uses a logic grammar formalism, Modular Logic Grammar, which allows logic grammars to be more compact, and which has a modular treatment of syntax, lexicon, and semantics. The English grammar is written independently of the task of translation. LMT uses a syntax-to-syntax transfer method for translation, although the English syntactic analysis trees contain some results of semantic choices and show deep grammatical relations. Semantic type checking with Prolog inference is done during analysis and transfer. The transfer algorithm uses logical variables and unification to good advantage; transfer works in a simple left-to-right, top-down way. After transfer, the German syntactic generation component produces a surface structure tree by application of a system of tree transformations. These transformations use an augmentation of Prolog pattern matching. LMT has a single lexicon, containing both source and transfer information, as well as some idiosyncratic target morphological information. There is a compact external format for this lexicon, with a lexical preprocessing system that applies defaults and compiles it into an internal format convenient for the syntactic components. During lexical preprocessing, English morphological analysis can be coupled with rules that synthesize new transfer entries.[1]

1 Introduction

The purpose of this paper is to describe an experimental English-to-German machine translation system, LMT (logic-based machine translation),[2] which has evolved out of previous work by the author on logic grammars.

The translation system is organized in a modular way. The grammar for analysis of the source language (English) is written completely independently of the task of translation. In fact, this grammar produces logical forms that can be used for other applications such as database query systems and knowledge-based systems, and has been used in the systems described in McCord (1982, 1987), Teeple (1985), Bernth (1988), and Dahlgren (1988). The components of LMT dealing specifically with translation do not index into the grammar rules, as, for example, in the LRC system (Bennett and Slocum 1985).

An interesting sort of modularity exists in the English grammar itself, whereby syntax, lexicon, and semantic interpretation closely interact, yet manage to be clearly separated. The lexicon exerts control over syntactic analysis through the use of slot frames in lexical entries and slot filling methods in syntax, as well as through type checking with semantic types taken from lexical entries. Yet the syntax rules look completely syntactic; e.g., no specific semantic types or word senses are referred to. The syntactic analysis trees look like surface structure trees, with annotations showing grammatical relations (including remote relations due to extraposition). The terminal nodes of these trees are **logical terminals** (explained below), which contain word sense predications and can be used in building logical forms as semantic representations of sentences. These logical forms are built by a separate semantic interpretation component which deals with problems of scoping of quantifiers and other modifiers.

Given that the English grammar can produce both syntactic structures and logical forms, an issue in designing LMT was what structures to use as input to transfer. The initial idea was to use the logical forms. The main argument for this was that 1. the logical form analyses express the complete meaning of the source text, and 2. there is no doubt that for perfect translations, one must in general have a complete semantic analysis of the source text (and employ world knowledge to get it). The logical form analyses are expres-

Copyright 1989 by the Association for Computational Linguistics. Permission to copy without fee all or part of this material is granted provided that the copies are not made for direct commercial advantage and the *CL* reference and this copyright notice are included on the first page. To copy otherwise, or to republish, requires a fee and/or specific permission.

0362-613X/ 89 /010033-52$03.00

sions in a **logical form language** (LFL) (McCord 1985a, 1987). Although the formalism for LFL is intended to be language universal, there is actually a different version of LFL for every natural language, because most of the predicates are word senses in the natural language being analyzed. The original scheme, then, for LMT was to analyze English text into English LFL forms, then transfer these to German LFL forms, then generate German text.

This scheme is neat, and may be investigated again later; but for the sake of practicality, the compromise has been to use the syntactic analyses produced by the grammar as the point of transfer. Useful MT systems must generally work with rather large domains, and the trouble with the use of logical forms is that too many decisions must be made and too much world knowledge is needed to produce correct analyses for a large domain. For example, LFL expressions for degree adjectives like "good" are focalizers (McCord 1985a, 1987), where the base argument shows the base of comparison for the adjective. In general, it may be difficult to determine such arguments. In the syntactic structure, arguments of focalizers are not yet determined; but for the purposes of translation, such scoping problems can often (though not always) be ignored. They can often be sidestepped because the same ambiguity exists in the target language. For example, "He is good" can easily translate into *Er ist gut* without deciding "good with respect to what?". Another point is that in the case of languages as close as English and German, it is simply more direct to transfer syntactic structure to syntactic structure. For more discussion of the practicality of a syntactic transfer method, see Bennett and Slocum (1985).

It should be emphasized that the syntactic analysis trees produced by the grammar do contain some of the ingredients of semantic interpretation. As mentioned above, terminal nodes contain word sense predications. Although the arguments of focalizer predications are not yet filled in, the arguments of verb and noun senses (corresponding to complements), are filled in (inasmuch as they can be determined by the syntax of the sentence, plus a few heuristics). Semantic type checking involves Prolog inference and is used for constraining word sense selection, complementation, and adjunct attachment. Also certain preference heuristics, described in Section 2 below, are used for modifier attachment.

Translation of a sentence by LMT proceeds in five steps.

1. Lexical preprocessing;
2. English syntactic analysis;
3. English-to-German transfer;
4. German syntactic generation;
5. German morphological generation.

During Step 1, lexical preprocessing, the words of an input sentence are looked up in the LMT lexicon, in combination with English morphological analysis (both inflectional and derivational). Morphological derivations are used to synthesize new transfer entries. For example, the derivation of "reuseable" from "use" and the existence of a transfer entry *use→ verwenden* allow automatic synthesis of a new transfer entry *reuseable → wieder verwendbar*.

Step 1, and Step 5 as well, are the topics of a companion paper (McCord and Wolff 1988). The present paper deals mainly with the syntactic components of LMT; but enough description of the lexicon is given to make the discussion self-contained.

Step 2, English syntactic analysis, is dealt with in Section 2. Several aspects of the English grammar are described: the Modular Logic Grammar formalism, use of metarules in the grammar, special syntactic techniques, and the methods used for semantic type checking.

Section 3 provides an overview of the LMT lexicon and its relation to the English grammar.

Step 3 is dealt with in Section 4, "The Transfer Component of LMT." The transfer component converts an English syntax tree into the German transfer tree. This is a syntax tree that (normally) has the same shape as the English tree, but has different node labels. Its nonterminal nodes are labeled by feature structures appropriate for German syntax and morphology; and its terminal nodes are (normally) citation forms of German words, together with feature structures that determine the inflections of the words during Step 5.

The transfer algorithm works in a simple way, in one top-down, left-to-right pass, yet manages to get a lot done, making German word choices and essentially producing all required German feature structures (like case markers). This is facilitated by use of logical variables and unification. Lexical transfer information resides in Prolog clauses (in internal representation), used by the transfer algorithm for simultaneous determination of German target words and associated inflectional markings for complements of these target words.

Step 4, German syntactic generation, is described in Section 5. This phase takes the German transfer tree and produces a German surface structure tree by applying a battery of tree transformations in a cycle, as in transformational grammar. The pattern matching used by these transformations is mainly Prolog unification, but there is an augmentation for matching sublists. Transformations are expressed in a special notation involving this augmented pattern matching and are compiled by the system into normal Prolog clauses. The number of transformations used in the system is rather small (currently 44), because the general idea of LMT is to get as much right as possible during the transfer step.

As mentioned above, Step 5, German morphological generation, is described in detail in McCord and Wolff (1988), but some comments are given here in Section 6.

Section 7 briefly describes the status of the system as of November 1988. It is worth noting here that LMT, although fairly large by now, is written entirely in Prolog (except for a few lines of trivial system code). No need has been seen for other methods, even for quick access to large dictionary disk files. The version of Prolog used is VM/Prolog (written by Marc Gillet), running on an IBM mainframe. The features of Prolog (especially logical variables and unification) have been very useful in making LMT easy to write.[3]

2 The English analysis grammar, ModL

The term **grammar** is being used in this paper in the broad sense of a system that associates semantic representations with sentences (and may also associate syntactic analyses). A **modular logic grammar** (MLG) (McCord 1985a, 1987) has a **syntactic component** (with rules written in a certain formalism), and a separate **semantic interpretation component** (using a certain methodology). The English MLG used in LMT is called **ModL,** and has evolved since 1979. Many of the ingredients have been described previously (McCord 1981, 1982, 1985a, 1987), so the background description given here is abbreviated (but fairly self-contained), and the emphasis is on new ingredients.

2.1 The MLG formalism

Metamorphosis grammars (MGs) (Colmerauer 1975, 1978) are like type-0 phrase structure grammars, but with logic terms for grammar symbols, and with unification of grammar symbols used in rewriting instead of equality checks. In an MG in normal form, the left-hand side of each rule is required to start with a nonterminal, followed possibly by terminals. Definite clause grammars (DCGs) (Pereira and Warren 1980), are the special case of normal form MGs corresponding to context-free phrase structure grammars; i.e., the left-hand side of each rule consists of a nonterminal only. In MGs (and DCGs), any of the grammar symbols can have arguments, and these can be used to constrain the grammar as well as to build analysis structures. MGs (in normal form) can be compiled directly into Horn clauses (Colmerauer 1978) (hence run in Prolog for parsing and generation), by adding extra arguments to nonterminals representing difference lists of word strings being analyzed. In MGs, the right-hand side of a rule can also contain ordinary Horn clause goals, which translate into themselves in the compilation to Horn clauses.

The MLG syntactic formalism is an extension of the DCG formalism. The three most important extra ingredients (to be explained in this subsection) are the following:

1. A declaration of **strong nonterminals** preceding the listing of syntax rules;
2. **Logical terminals**;
3. **Shifted nonterminals.**

The second and third ingredients are allowed on the right-hand sides of syntax rules. There are several other minor types of extra ingredients in the MLG formalism, which will be mentioned at the end of the subsection.

The syntax rule compiler of an MLG compiles the syntactic component directly into Prolog (as is common with MGs, so that parsing is top-down), but takes care of analysis structure building, so that the grammar writer does not have to bother with the bookkeeping of adding nonterminal arguments to accomplish this (as in MGs). Also, since systematic structure building is in the hands of the rule compiler, it is easier to write metagrammatical rules.

The MLG rule compiler has two options for structure building. The compiled grammar can operate in a **one-pass mode,** in which LFL representations are built directly during parsing, through interleaved calls to the semantic interpreter, and no syntactic structures are built. Or it can operate in a **two-pass mode,** in which syntactic structures are built during parsing, and these are given to the semantic interpreter in a second pass. The two-pass mode is used for LMT, since we want syntactic analysis trees. In the following discussion, the one-pass mode will be ignored—for details, see McCord (1985a, 1987).

Now let us look at the three distinctive ingredients of MLGs mentioned above. Strong nonterminals represent major syntactic categories, and they are declared by a clause

```
strongnonterminals(NT1.NT2. ....NTn.nil).
```

(The dot operator is used for lists.) Each `NTi` is of the form `A/k` where `A` is an atom, the principal functor of the strong nonterminal being declared, and `k` is a nonnegative integer, less than or equal to the number of arguments of the nonterminal. The first `k` arguments of the nonterminal are called its **feature arguments**; their significance is explained below.[4] A nonterminal not declared strong is called **weak.** A syntax rule whose left-hand side is a strong (weak) nonterminal is called a **strong (weak)** rule.

The most significant way in which the strong/weak distinction is used by the MLG rule compiler is in automatic analysis structure building. Nodes for the analysis tree get built corresponding to the application of strong rules, but not weak rules. Specifically, when a strong nonterminal

```
A(X1,...,Xn)
```

is expanded in the derivation of a sentence, a tree node of the form

```
syn(A(X1,...,Xk),B,Mods)
```

is built for the analysis tree. The first argument of the `syn` term is the label on the node, consisting of the nonterminal together with its feature arguments. Thus feature arguments are made available in the syntactic description of the sentence, and may be used by other modules—such as transfer in LMT. (The significance of feature arguments for MLG metarules is indicated below.) The second argument B of syn has to do with bracketing of the phrase analyzed by A, and will be explained below. The last argument Mods is the list of daughter nodes.

The second way in which MLGs differ from DCGs is that the right-hand sides of rules can contain **logical terminals.** These are building blocks for analysis structures, just as ordinary terminals are building blocks for the word strings being analyzed. The terminal nodes of syntactic analysis trees are logical terminals. In fact, the terminal node members of Mods in the syn term above are just the logical terminals encountered while expanding the strong nonterminal A, possibly through the application of subordinate weak rules, but not through other applications of strong rules.

Logical terminals are terms of the form Op-LF. Here, LF is a logical form (an LFL expression), usually a word sense predication, like see(X,Y). The term Op, called an **operator,** determines how the logical terminal will combine with other logical terminals during semantic interpretation to build larger logical forms. For a description of the way MLG semantic interpretation works, see McCord (1985a, 1987).

As indicated above, the MLG semantic interpreter is not used in LMT. Because of this, the operator components of logical terminals are not important here; however, the logical form components are used. The arguments of word sense predications show deep relations of words to other parts of the sentence, including remote dependencies, and play a central role in the transfer algorithm, as we will see in Section 3. It should also be noted that the grammar ModL has been shaped strongly by the need to produce logical form analyses.

The last distinctive ingredient of MLGs is the **shift operator,** denoted by %. Its purpose is to allow the production of left-embedded structures while using right-recursive rules (necessary because of the top-down parsing). Before describing the shift operator generally, let us look at an example.

Left-recursive constructions occur in English noun phrases like "my oldest brother's wife's father's car". A noun phrase grammar fragment with shift that handles this is as follows:

```
np ⇒ determiner: np1.
np1 ⇒ premodifiers: noun: np2.
np2 ⇒ apostrophe_ s: np % np1.
np2 ⇒ postmodifiers.
```

Here, np is declared a strong nonterminal and all others are weak. (The colon operator on the right-hand side of MLG rules denotes the usual sequencing.) The occurrence of an apostrophe-s triggers a shift back to the state np1, where we are looking at the premodifiers (say, adjectives) of a noun phrase. In making the transition, though, the provisional syntactic structure being built for the noun phrase is changed: All daughters constructed so far are made the daughters of a new node with label np (the left operand of the shift operator), and this new node is made the initial daughter in the new provisional syntactic structure.

In general, the right-hand side of an MLG syntax rule can contain a **shifted nonterminal,** which is of the form Label%NT, where Label is a term (to be used as a node label), and NT is a weak nonterminal. The idea, in rather procedural terms, is: 1. Take the list of daughters built so far for the active tree node (corresponding to the most recently activated strong rule), and make it the complete daughter list of a new node Mod with label Label; and then 2. proceed with NT, using Mod as the initial daughter of the active tree node.

It should be noted that the syntactic analysis structures built automatically for MLGs differ from derivation trees in three ways: a. Weak rules do not contribute nodes to analysis trees (but strong rules do); b. shifted nonterminals can contribute nodes in the special way just indicated; and c. terminal nodes are logical terminals, not word string terminals.

It was mentioned above that there are several minor types of extra ingredients in the MLG formalism. Five of these will be described here briefly.

1. There is an "escape" to the DCG formalism: A grammar symbol of the form -NT does analysis with a DCG nonterminal NT, defined by its own DCG rules. (DCG rules are written with the symbol →, whereas MLG rules are written with ⇒.) This is useful, for the sake of efficiency, when MLG structure building is not necessary.
2. In order to look at right context, one can refer to the next terminal T (without removing T from the word stream) by using the symbol +-T. (Ordinary references to terminals are indicated by +T.)
3. Also, one can examine the complete right context with a DCG nonterminal NT by use of the symbol -NT.
4. As with DCGs, one can specify a Prolog goal Goal on the right-hand side of a rule. Our notation for this is $Goal. Such goals are executed when the compiled grammar is executed. But there is another type of Prolog goal, denoted by !Goal, which gets executed immediately at compile time. This is convenient, e.g., for specifying feature selection goals whose immediate compilation constrains feature structures through unification—with more efficient execution during parsing. Writing such constraints directly may not be as perspicuous, or as flexible if one wants to change the representation of feature structures.
5. Although syntactic structures are handled automatically by the rule compiler, it is occasionally convenient to be able to refer to them. A symbol NT > Syn , where NT is a strong nonterminal, binds Syn to the syntactic structure of the phrase analyzed by NT (and is otherwise treated like an occurrence of NT)[5]. There is a similar method for referring directly to bracketing symbols (dealt with in the next section).

2.2 METAGRAMMATICAL RULES

There are two grammatical constructions that are so pervasive and cut across ordinary grammatical categories to such an extent, that they invite treatment by metagrammatical rules: **coordination** and **bracketing.** Coordination is construction with "and", "or", "but", etc. Bracketing consists of the enclosure of sentence substrings in paired symbols like parentheses, other types of brackets, dashes, and quotes. Also, in text formatting languages, there are paired symbols used for font changes and other formatting control. LMT is being written to process the source text for the IBM SCRIPT/GML formatting language (as well as ordinary text), so it is important to handle such formatting control symbols. (Note that "bracketing" symbols can be nested [as in this sentence].) Use of metarules allows one to make coordination and bracketing more "invisible" to the parser and translator.

Coordination has been treated metagrammatically in several systems. In the logic programming framework, treatments include those in Dahl and McCord (1983), Sedogbo (1984), and Hirschman (1986). The first of these systems implemented coordination metarules in an **interpreter** for the logic grammar, whereas the last two implement them in a syntax rule compiler. Bracketing with ordinary parentheses is treated in the LRC system (Bennett and Slocum 1985) by reliance on LISP's handling of parentheses.

There is a limited treatment of coordination and bracketing through metarules in the MLG rule compiler. Specifically, the implementation is for coordination and bracketing of complete phrases, where a phrase is a word string analyzed by a strong nonterminal. Any phrase (type) can be coordinated, any number of times, using the usual coordinating conjunctions, commas, and semicolons, as well as the "both-and", "either-or" constructions. Bracketing of a phrase (with nesting to any level) is allowed in contexts where the phrase could occur grammatically anyway (as in this sentence). In addition, appositive parentheticals, as in "I know that man (the one over there)", where a phrase type is repeated in parentheses, are treated by the metarules.

The current restriction to coordination of complete phrases (without identifying gaps) is not quite as severe as it might seem, because 1. there are quite a few phrase types (including verb phrases, verb groups, and noun groups), and for these, all appropriate associations of variables are made; and 2. examples with real gaps often do at least get parsed because of optional constituents (as in "John saw and Mary heard the train", where "John saw" is parsed as a complete phrase because the object of "saw" is optional).

The second argument of the Prolog term syn(Label, B,Mods) representing a syntax tree node is used to accommodate bracketing. The term B is a list of symbols, like quote.paren.nil, each representing a pair of brackets enclosing the phrase represented by the node.

This "factored out" representation of brackets allows the translation component of LMT to handle brackets in a way that is transparent to most of the rules. The result is that if a phrase is bracketed in a certain way in the English source, then the corresponding phrase will automatically be bracketed in the same way in the German translation.

Coordination and bracketing are handled in an integrated way by the rule compiler. For each strong nonterminal, the following is done (a simplified version is given here). For the sake of concreteness, let us say that the nonterminal has name nt and that it has five arguments:

```
nt(F,G,H,I,J)
```

(before compilation), where the first two arguments F and G are declared to be the feature arguments. The existing syntax rules for nt are compiled essentially as in McCord (1985a), but the name given to the head predicate is changed to ntbase, representing the simple (noncoordinated, nonbracketed) form of the phrase. In addition, the metarules create four additional Prolog rules—for the original nt, not for ntbase. The first additional rule is:

```
nt(F,G,H,I,J, syn(Lab,B1,Mods), U,Z) ←
  copylabel(nt(F,G),nt(F1,G1)) &
  bbrackets(B, U,V) &
  preconj(PC,Mods,Mods1, V,W) &
  ntbase(F1,G1,H,I,J, B, Syn0, W,X) &
  ntconj(F1,G1, F,G,H,I,J, PC,
         Syn0,syn(Lab,B1,Mods1), X,Y) &
  ebrackets(B1, Y,Z).
```

In each of these predications besides copylabel, the last two arguments represent difference lists for word strings. The purpose of copylabel is to create a new version of the label nt(F,G) which can differ in some subterms, to allow for differences in the feature structures of the coordinated phrases. (Feature arguments for constituents of coordinated phrases are thus allowed to differ, but the other arguments in repeated calls of ntbase must match.) The procedure bbrackets ("begin-brackets") reads the list B (possibly empty) of brackets from the word list (represented by difference list (U,V).) A possible preconjunction PC (like "both") is gotten by preconj. Then the simple nonterminal ntbase is called. Then ntconj gets the remainder of the coordinated phrase, and ebrackets closes off the brackets.

There are three rules for the continuation ntconj. The first of these (to be given below) gets most types of conjunctions, makes another call to ntbase, and finally calls ntconj recursively. The second allows termination. The third is like the first, but gets other types of conjunctions. Thus, with termination in the middle of these three rules, a **preference**[6] is created for certain types of coordination at the given phrase level. The details of this preference coding will not be given here. The first rule for ntconj is:

```
ntconj(F0,G0, F,G,H,I,J, PC,
      syn(nt(F0,G0),nil,Mods0),Syn, U,X) ←
  optionalcommma(U,T.V) &
  coord(T,PC,a,nt,Op,LF) &
  copylabel(nt(F,G),nt(F1,G1)) &
  ntbase(F1,G1,H,I,J, nil,
         syn(nt(F1,G1),nil,Mods1), V,W) &
  combinelabels(T,nt(F0,G0),nt(F1,G1),nt(F2,G2)) &
  ntconj(F2,G2, F,G,H,I,J, nil,
         syn(nt(F2,G2),*,
                syn(nt(F0,G0),nil,Mods0).
                Op-LF.
                syn(nt(F1,G1),nil,Mods1).nil),
         Syn, W,X).
```

Here coord tests that the terminal T is a coordinating conjunction, allowing preconjunction PC, being of conjunction type (a,nt), and having associated logical terminal Op-LF. Conjunctions used in the first ntconj rule are given conjunction type (a,nt), and those used in the third rule are given type (b,nt). This distinction is related to specific conjunctions by the rules for coord. The procedure combinelabels combines features of conjuncts (this includes the treatment of number for coordinated noun phrases). Finally, ntconj is called recursively to get possible further coordinated phrases.

The second rule for ntconj (termination) is trivial, and will not be given. The third is essentially like the first, but requires the conjunction type (b,nt) in the call to coord.

Note that some category-specific information for coordination does have to be written, mainly in the rules for copylabel and combinelabels (since these depend on the nonterminal nt). However, default rules exist for these in ModL, so that one does not have to write special rules for all categories. On the whole, the amount of rule writing is greatly reduced by the metarules.

As mentioned above, the rules produced by the metarules were given here in simplified form. The actual, more complex, forms deal with the following three things.

1. A more complete treatment of bracketing and punctuation within coordinated phrases. The above rules allow bracketing only at the beginning and end of a complete, coordinated phrase; therefore extra calls to begin-bracket and end-bracket procedures are needed. Also there are actually *two* termination clauses for ntconj—a clause dealing with appositives introduced by commas, and a simple termination clause.
2. Appositive parentheticals (mentioned above). These are handled by an additional clause for ntconj.
3. A partial tabular parsing facility. The purpose of this facility is to allow parsing (and translation) of inputs that are not complete sentences, while using top-down parsing. The only nonterminal called by the driver of ModL is s, for a complete sentence. It happens that most types of phrases can begin a sentence in the ModL grammar. When s fails but the input is a well-formed phrase of some type, a syntactic structure for the input usually gets built nevertheless during the parse. Thus it is worth saving results of phrase analyses that span the whole input. The rule compiler takes care of this by adding at the end of the main rule for each strong nonterminal (cf. the rule for nt above) a call to a procedure savesyn, which saves the corresponding syntactic structure when the analyzed phrase spans the whole input string. (Saving is done by assertion into the Prolog workspace.) Therefore, when a sentence analysis fails, these saved partial results may be used.

Experiments were made with general tabular parsing (see, e.g., Pereira and Shieber 1987), but it was found that this does not speed up parsing with the particular grammar ModL, especially considering that only the first parse is normally used in LMT.

2.3 SYNTACTIC AND SEMANTIC TECHNIQUES IN MODL

The syntactic component of ModL is basically an extension of that in McCord (1982), which was written as a DCG. In particular, slot filling techniques are used in ModL for handling complementation. However, there are some improvements in the basic techniques, which will be described in this section.

2.3.1 POSTMODIFIERS AND ORDERING CONSTRAINTS

As in the earlier grammar, the analysis of the complements of an open class word (verb, noun, or adjective) is directly controlled by a slot frame which appears in the lexical entry of the open-class word. There is a weak nonterminal postmods, which takes as input the slot frame of the word, chooses slots (nondeterministically, and not necessarily in the order in which they appear in the slot frame), and tries to fill the slots by slot filling rules indexed to specific slot names. The procedure postmods also finds adjunct postmodifiers. Slot fillers (complements) correspond to arguments of the word sense predication for the open-class word, and adjuncts correspond to outer modifiers of it in logical form.

By itself, the free choice of slots and adjuncts for postmodification allows for free ordering of these postmodifiers; but of course the ordering should not be completely free, and some constraints are needed. An improved method of expressing such constraints has been developed for ModL.

The same procedure postmods is used for all three open class categories, but let us illustrate its use for verbs. The following ModL rule (simplified) for the nonterminal predicate gets a verb and its postmodifiers.

```
predicate(Infl,VS,X,C) ⇒
  vc(Infl,VS,Y,Slots):
  voice(Infl,X,Y,Slots,Slots1):
  $theme(X,Slots,Z):
  postmods(vp,nil,Slots1,VS,Z,C).
```

(Recall that the $ sign signals that its operand is a Prolog goal.) Here Infl is the inflectional feature structure of the verb, VS is the verb sense, X is the **marker**[7] for the grammatical subject of the verb, and C is the **modifier context** for predicate (to be explained below).

The nonterminal vc (**verb compound**), which is the only strong nonterminal in this rule, gets the head of the predicate. (The feature arguments of vc are declared to be its first two arguments.) A verb compound normally consists of a single word, but could be a compound like "time share". And of course since vc is a strong nonterminal, coordinated forms are allowed. Verb compounds do not include auxiliary verbs as premodifiers; these are treated as separate, higher verbs with their own complementation. The call to vc determines the marker Y for the logical subject of the verb, and the slot list Slots of the verb.

The procedure voice handles the passive transformation (when the verb analyzed by vc is a passive past participle) as a slot list transformation, and theme computes the marker Z for implicit subjects in complements like "John wants *to leave*", and "John wants Bill *to leave*". For these, see the discussion in McCord (1982).

The first rule for postmods, which gets slot fillers (as opposed to adjuncts), is as follows, slightly simplified (we leave off the treatment of the modifier context argument for now).

```
postmods(Cat,State,Slots,VS,Z) ⇒
  $selectslot(Slot,State,Slots,Slots1):
  filler(Slot,Z):
  postmods(Cat,Slot.State,Slots1,VS,Z).
```

What is of interest here (compared with McCord 1982) is the use of the State argument, whose purpose is to constrain the free ordering of postmodifiers. In the earlier grammar, states were a linearly ordered set of symbols isomorphic to the natural numbers, and the idea was that postmodification by a given slot (or adjunct type) can advance the state to a certain level, or leave it the same, but can never go backwards. The trouble with this (as implemented) was that postmods could try filling a "late state" slot when an obligatory "early state" slot has not been filled yet. (This does not cause any wrong parses, but it is inefficient.)

The cure involves looking not only at the postmodifiers that have already been found, but also at the obligatory slots that are still pending. The state is now just the list of slots and adjunct types that have already been used. (Building up of this list can be seen in the above rule for postmods.)

The procedure selectslot selects a Slot from the current list Slots, with Slots1 as the remainder. In so doing, it looks at the current state as well as the remaining slots to exercise the constraints.

The specific constraints themselves are expressed in the most straightforward way possible—as ordering relationships S1 ≪ S2 , where S1 and S2 are slots or adjunct types. Slots are represented as terms slot(S,Ob,Re,X), where 1. S is the slot name (like obj or iobj), 2. Ob indicates whether the slot is obligatory or optional, 3. Re indicates whether the slot has a real filler, or a virtual filler (because of left extraposition), and 4. X, is the marker for the slot filler. **Adjunct types** are simple symbols (like avcl for adverbial clause), which divide adjuncts into broad types. Specific ordering constraints are:

```
slot(iobj,*,r,*) << slot (obj,*,r,*).
slot (obj,*,r,*) << slot (S,*,r,*) ← S = /iobj.
slot(*,*,r,*) << avcl.
```

The idea of selectslot is then simple. It selects a slot S nondeterministically from the current slot list Slots, leaving remainder Slots1; but it checks that 1. there is no member S1 of State such that S ≪ S1, and 2. there is no obligatory slot S2 in Slots1 such that S2 ≪ S.

The basic idea of factoring out the control of constituent ordering into simple ordering relationships has been used in other systems, for example in the systemic grammar system of Hudson (1971), and more recently in the ID/LP formalism (Gazdar and Pullum 1982).

2.3.2 Preference Attachment

A second improvement in ModL concerns **preference attachment** of postmodifiers in the sense of Wilks, Huang and Fass (1985), and Huang (1984a, 1984b). The problem is simply stated: When we have parsed part of a sentence, as in "John saw the way to send a file . . .", and we see a further phrase "to Bill", then does this attach to "file", "send", "way", or "saw"? I.e., which final phrase of the partial sentence does it modify? If the initial segment were instead "John described the way to create a file . . .", then the answer would be different.

The method of handling this in ModL is basically similar to that in the work of Huang, Wilks, and Fass cited above, but seems slightly simpler and more general, because of the systematic use of postmods in ModL. The implementation involves the modifier context argument (the last argument) of postmods.

It should be mentioned first that the modifier context is used not only for handling preference attachment, but also for left extraposition. The modifier context contains a pair of **topic** terms (T,T1) used as in McCord (1982) to represent a left-extraposed item T, with T1 equal to nil or T according as T is or is not used as a virtual filler (or adjunct) by postmods.

A **modifier context** is a term of the form c(T,T1,Pend), where (T,T1) is a topic pair and Pend is a **pending** stack. The latter is a list whose members are **pending frames**, which are terms of the form Cat.Sense. Slots, giving a phrase category (verb, noun, or adjective), the sense of the head, and the current slot list of the head (some slots may already be used). A pending frame describes what is possible for further modifiers of a given head word (adjunct modification depends on the category Cat and the particular head word (sense) Sense).

Using modifier contexts, an essentially complete version of the slot-filling rule for postmods is:

```
postmods(Cat,State,Slots,VS,Z,c(T,T2,Pend)) ⇒
  $selectslot(Slot,State,Slots,Slots1):
  filler(Slot,Z,c(T,T1,(Cat.VS.Slots1).Pend)):
  postmods(Cat,Slot.State,Slots1,
           VS,Z,c(T1,T2,Pend)).
```

Thus, in the call to filler, the current pending frame is stacked onto the pending stack. A rule for filling, say, an object slot with a noun phrase would pass this larger modifier context argument into the noun phrase, where the higher context is then available.

On a given level for postmods, the most pressing question is how to attach prepositional phrases. Slot filling is always preferred over adjunct modification on a given level. Thus, if the given head word has a prepositional object slot pobj(Prep) matching the given preposition, then only this will be tried.

To decide whether a pp can attach as an adjunct modifier, the pp rule (as soon as it sees the preposition) looks at the pending stack to determine whether there are pending prepositional case slots (pobj) that could take the given preposition, and, if so, the pp aborts. Adjunct attachment of a pp can also be blocked by semantic type requirements made by the preposition on the modified phrase and the object of the preposition (even the combination of these two). A discussion of semantic type checking is given at the end of this subsection. Currently the grammar does not try to compare semantic types for preferences; but this could be done since the pending stack, with all the higher head word senses, is in place.

2.3.3 Noun Compounds

A third improvement in the grammar is the treatment of noun compounds. Noun compounds were treated in a limited way in McCord (1982) by allowing noun premodifiers of the head noun to fill slots in the head noun, as in "mathematics student". In the syntactic structure, these noun premodifiers were all shown on the same level, as daughters of the noun phrase, although the slot filling attachment to the head corresponds logically to a right branching structure. But of course noun compounds in English can exhibit any pattern of attachment, with the patterns corresponding to the ways one can bracket *n* symbols. This is important to capture.

The shift operator allows one to produce all patterns of attachment—left branching, right branching, and all combinations in between—while using right recursive rules. The following small grammar produces all possible bracketings:

```
np → +N.
np → +N: np%np1.
np1 → np.
np1 → np: np%np1.
```

Here, np is a strong nonterminal and np1 is weak. Recall that + N signals that N is a terminal.

In ModL, a somewhat more complicated form of this fragment is used in the noun compound rules. Each subcompound gets a slot list and a marker associated with it, and there is a procedure attach (an extension of that in McCord 1982), which allows one subcompound to attach to another. Adjectives are included in the pot, but the rules for attaching them are of course different. The potential to get any pattern of attachment exists in the rules, but again preferences are implemented. Roughly, the idea is this: As a new noun (or adjective) N is read, if 1. the structure N0 already built has a head that is a noun, and 2. N can attach to N0, then one requires this immediate attachment, building a left branching structure. Otherwise, one continues with right branching and attaches the larger compound to N0. This scheme prefers left branching for a sequence of nouns, if attach allows it, but prefers a right branching structure for a sequence of adjectives followed by a noun.

Currently, attach does not deal with "creative" attachments, where the relationship between the two subcompounds is not mere slot-filling or apposition, but where the combination involves some extraneous relationship, as in "music box" and "work clothes". But an extended version of attach which handles such combinations could still be used in the existing algorithm.

2.3.4 Semantic Type Checking

Semantic type checking is done during parsing with ModL. In earlier versions of the system, semantic type checking was accomplished by Prolog unification of **type trees**, representing types in a hierarchy allowing cross-classification. It appears that in practice this scheme is not flexible and convenient enough; a more general type checking scheme based on Prolog inference has been implemented.

Let us illustrate the new scheme with type checking for noun phrase fillers of verb slots. In the format for a slot mentioned above

```
slot(S,Ob,Re,X)
```

X is the marker for a possible filler of the slot. A marker is of the form

```
Y:Sense:SynFeas & Test.
```

Here, Y is the logical variable associated with the noun phrase filler. During lexical preprocessing, Y is unified with the argument of the verb sense predication corresponding to this slot, or part of this argument.[8] When a filler is found during parsing, Y is also unified with the main logical variable for the noun phrase (normally the first variable in the noun sense predication). The component Sense of the marker is the sense name of the head noun of the filler. (In earlier versions of ModL, this component was a semantic type for the noun sense.) The component SynFeas is a term representing syntactic and morphological features of the noun phrase. Finally, Test is a Prolog goal that is executed after the head noun is found. Normally, Test will contain a variable unified with Sense, so that a test is made on the noun sense.

As a simple example, if a verb requires that its object be animate, then the object slot can have the marker

```
Y:S:SF & isa(S,animate).
```

If the head noun has sense man1, and the clauses

```
isa(man1,human).
isa(S,animate) <- isa(S,human).
```

are in the Prolog workspace, then the test in the marker will succeed.

The lexical preprocessing scheme of LMT allows convenient specification of type requirements on slot fillers (and on other kinds of modifiers) and type statements for word senses. Such type conditions can be given in lexical entries in a compact format that does not explicitly involve isa clauses. This will be described in the next section.

3 The LMT lexicon

Some MT systems have three separate lexicons, for source, transfer, and target; but LMT has only one, unified lexicon, indexed by source language (English) words. The entry for a word contains monolingual English information about the word, as well as all of its transfers. A transfer element can contain monolingual German information about the target word.

For example, a simple entry for the word "view" might be

```
view < v(obj) < n(nobj)
  < gv(acc,be+tracht)
  < gn(gen,ansicht.f.n).
```

Here, < is just an operator that connects the components of the entry. The monolingual English information is on the first line, showing that view is a verb taking an object and is also a noun with a (possible) noun object (appearing in postmodifier form as an of-pp complement). The transfer information is on the second and third lines. This shows that the translation of the verb form is the inseparable-prefix verb *betrachten,* where the German complement corresponding to the English object takes the accusative case. And the translation of the noun form is *Ansicht*, where the noun complement takes the genitive case, and *Ansicht* is a feminine noun (f) of declension class n.

There are two advantages of the unified lexicon design: 1. Lexical look-up is more efficient since only one index system is involved, and 2. it is easier for the person creating the lexicon(s) to look at only one lexicon, seeing all pertinent information about a source language word and its transfers.

It might be argued that it is inefficient to store monolingual target language information in transfer elements, because there is redundancy, e.g., when two noun transfers are German compound nouns with the same head. However, the format for specifying German noun classes and other German morphological information in the LMT lexicon is very compact, so the redundancy does not involve much space or trouble. More will be said below about the specification of German morphological information.

The principle that source language analysis in LMT is independent of the task of translation is not really violated by the unified lexicon, because purely English elements in lexical entries can easily be distinguished (as will be seen from the description below), and the remaining elements can be discarded, if desired, to obtain a monolingual English lexicon for other applications.

Another feature of the LMT lexicon is that the storage format is not the same as the format seen by the syntactic components. Both formats are Prolog clauses, but the lexical preprocessing step of LMT does **lexical compiling** of lexical entries, converting the external storage format into the internal format used by the syntactic components. Lexical compiling is applied not only to entries obtained by direct look-up (for words that are found directly in the lexicon), but also to "derived" entries, obtained by morphological analysis in conjunction with look-up. There are two reasons for doing lexical compiling. One is that it allows for compact, abbreviated forms in the external lexicon based on a system of defaults, whereas the compiled internal form is convenient for efficient syntactic processing. Another reason is that the lexical compiler can produce different internal forms for different applications. In fact, the internal form produced for applications of ModL involving logical forms is different from the form produced for LMT.

Lexical preprocessing is done on a sentence-by-sentence basis. Only the words actually occurring in an input sentence are processed. The internal form clauses produced for these words are deleted from the workspace, once the sentence is translated. Thus the parser sees only lexical clauses relevant to the words of the sentence, and in general the Prolog workspace is not overloaded by the more space-consuming internal-format clauses. Currently, the external lexicon is stored in the Prolog workspace (there being about 1,600 entries), but Prolog procedures for look-up in a large lexicon of the same form stored on disk have been written—along the lines described in McCord (1987).

Now let us look briefly at the external format.[9] A lexical entry consists of an English Word and its Analysis, represented as a Prolog unit clause

```
Word < Analysis.
```

(Here, the predicate for the unit clause is the operator <.) A word analysis consists of subterms called **analysis elements** connected by operators, most commonly the operator <. In the example for view above, there are four analysis elements. In general, analysis elements can be **English elements**, **transfer elements**, or (German) **word list transformations**. English elements will be discussed (briefly) in the current section; transfer elements will be described in the next section, and word list transformations in Section 6.

English analysis elements are of three types:

1. **base analysis elements,**
2. **(irregular) inflectional elements,** and
3. **multiword elements.**

The above example for view has two English base analysis elements, the v (verb) and n (noun) elements. Currently, there are 11 parts of speech allowed in base analysis elements—v (verb), modal, n (noun), propn (proper noun), pron (pronoun), adj (adjective), det (determiner), prep (preposition), subconj (subordinating conjunction), adv (adverb), and qual (qualifier). Let us look in particular at the form of verb elements.

The general, complete format (without use of defaults) for a verb element is

```
v(VSense,VType,SubjType,Slots)
```

Here, VSense is a name for a sense of the index word as a verb, VType is the semantic type of the verb (an inherent feature), SubjType is the semantic type requirement on the (logical) subject, and Slots is the slot list. An example to look at, before seeing more details, is the following simplified v element for the verb "give":

```
v(give1,action,human,obj:concrete.iobj:animate).
```

The semantic type VType of the verb can in general be any conjunction of simple types (represented normally by atoms). The type requirement SubjType on the subject can be an arbitrary Boolean combination of simple types.

The slot list Slots is a list (using the dot operator) of slot names (the final nil in the list is not necessary), where each slot name may have an associated type requirement on its filler. Like the SubjType, a type requirement for a slot can be an arbitrary Boolean combination of simple types.

An abbreviation convention allows one to omit any initial sequence of the arguments of a v element. If the sense name is omitted, it will be taken to be the same as the citation form. Omitting types is equivalent to having no typing conditions. For an intransitive verb with no typing and only one sense, the element could be simply v, with no arguments.

Given a (possibly inflected) verb V and a v element for the base form of V, the lexical compiler translates the v element into a one or more unit clauses for the predicate verb, with argument structure

```
verb(V,Pred,Infl,VSense,XSubj,
     SlotFrame).
```

Before saying what the arguments of verb are in general, we give an example for the inflected verb "gives" produced from the sample v element above:

```
verb(gives, give1(X:XS:XF,Y:YS:YF,Z:ZS:ZF),
 fin(pers3,sg,pres,*), give1,
 X:XS:XF & isa(XS,human),
 slot(obj,op,*,Y:YS:YF & isa(YS,concrete)) .
 slot(iobj,op,*,Z:ZS:ZF &isa(ZS,animate)) . nil) .
```

In general the arguments of verb are as follows: V is the actual verb (possibly a derived or inflected form), and Infl is an allowable inflectional feature structure. (There are as many verb clauses as there are allowable inflectional forms for V. For example, if V is made, then the inflection could be finite past or past participle.) The verb sense VSense becomes the predicate of the verb sense predication Pred, described in the next paragraph. The argument XSubj is the marker for the subject. The slot list in the v element is converted into SlotFrame, consisting of slots in the fuller form slot(S,Ob,Re,Y) described in the preceding section. (There can be optional and obligatory forms of the same slot.)

The verb sense predication Pred has arguments corresponding to the markers for the verb's complements —its subject and its slots—in the order given, but there is an option in the compiler: When ModL is being used to create LFL forms, these arguments will just be the logical variable components of the markers for the complements. But when ModL is used in LMT, the arguments will be the complete markers except for their semantic type tests. (Thus the arguments are of the form Y:Sense:SynFeas, as described in Section 2.3) This ready access to features of complements, by direct representation in the word sense predication, is very useful for transfer in LMT and will be illustrated in the next section.

The lexical compiler handles semantic type conditions by converting them into Prolog goals involving isa. For example, for each component type T of the semantic type VType of a verb (given in a v element), the unit clause isa(VSense,T) is added to the workspace. Thus, in the case of "gives" above, isa(give1,action) is added. Type conditions as isa clauses relating to specific word senses are handled dynamically, but relations between types such as

```
isa(S,animate) <- isa(S,human)
```

are stored permanently. A type requirement for a verb complement (subject or slot list member), being a Boolean combination of simple types T, is converted into a similar Boolean combination, Test, of goals isa(S,T), where S is the sense component of the complement's marker; and Test is made the test component of this marker.

The second kind of English analysis element (mentioned above) is an inflectional element. Eleven kinds of these are allowed (McCord and Wolff 1988). An example of an inflectional element is ven(V), which indicates that the index word is the past participle of the verb V. This appears in

```
become < v(predcmp) < ven(become).
```

where become is shown as the past participle of itself.

The third kind of English analysis element is the multiword element. Multiword elements (existing in transfer also) are used for handling idiomatic phrases in LMT. Multiword forms are allowed for all but three (modal, propn, and qual) of the 11 parts of speech. Their names are like the base analysis element names, but with a initial m. An example of an entry with a multiverb element is the following (simplified) entry for "take":

```
take < v(obj) < mv(=.care.of,obj).
```

The mv element allows a treatment of the phrase "take care of X". Forms based on inflections of the index word, such as "took care of", are handled automatically by the morphological system. Multiword elements have much the same format as single word elements except that sense names cannot be specified, and the first argument is always a multiword pattern (like = .care.of). Lexical preprocessing verifies that the pattern actually matches a sublist of the sentence before compiling the multiword element.

Some kinds of idiomatic phrases are treated through the use of slots in base analysis elements. For example, there is a verb slot ptcl(P) that allows particles, specified by P, for the verb. The particle P might be an ordinary particle like "up" or "back" as in "take up", "take back", or it could be a phrasal particle, like into.consideration, for handling "take into consideration". Note that "into consideration" does behave much like an ordinary particle, since we can say "take X into consideration", as well as "take into consideration X", if X is not too light a noun phrase. A base analysis element for "take" that allows both ordinary particles and the multiparticle "into consideration" is

```
v(obj.ptcl(all | into.consideration)).
```

This shows that "take" is a verb with an object slot and a particle slot. The particle allowed could be any ordinary single-word particle (indicated by all) or (indicated by |) the multiword particle "into consideration".

Idiomatic phrases can also be treated by German word list transformations. These are described in Section 6.

In addition to the unified LMT lexicon we have been describing, there is an auxiliary interface of ModL to the UDICT monolingual English lexicon Byrd (1983, 1984). This contains around 65,000 citation forms, with a morphological rule system to get derived forms of these words. The ModL lexical compiler also can convert UDICT analyses to the internal form required by the ModL grammar.

4 The transfer component of LMT

The transfer component takes an English syntactic analysis tree syn(Lab,B,Mods) and converts it to a German tree syn(GLab,B,GMods) which normally has the same shape. Before discussing the transfer method in general, let us look at an example. The English sentence is "The woman gives a book to the man". The syntactic analysis tree produced by ModL is:

```
s(fin(pers3,sg,pres,ind),give,*,top)
 np(X:woman:*&*)
     detp(X:woman:*&*)
        the(P,Q)
     woman(X:woman:*)
 vp(fin(pers3,sg,pres,ind),give)
    give(X:woman:*,Y:book:*,Z:man:*)
    np(Y:book:*&*)
       detp(Y:book:*&*)
          a(P1,Q1)
       book(Y:book:*)
    ppnp(to,Z:man:*&*)
       np(Z:man:*&*)
          detp(Z:man:*&*)
             the(P2,Q2)
          man(Z:man:*)
```

Each nonterminal node label in the tree consists of the strong nonterminal responsible for the node together with its feature arguments (as indicated in Section 2.1). The feature arguments for the np nodes are just the markers for these noun phrases. For the sake of simplicity in the display of this tree, the syntactic feature structures and semantic tests of np markers are just shown as stars. The terminals in the syntactic analysis tree are actually logical terminals, but we do not display the operator components, since these are not relevant for LMT. Also, we do not display the node labels for noun compounds (nc) and verb compounds (vc) unless these compounds have more than one element.

To get a good idea of the working of the transfer algorithm, let us look at the transfer of the verb "give" in this example, and the effect it has on the rest of the transfer. The terminal in the above tree involving give is a verb sense predication of the form described in the preceding section as the second argument of verb. The most relevant thing to notice in the syntax tree is that the variables X, Y, and Z in the give predication are unified with the logical variables in markers of the corresponding complements of give. Transfer of the give form simultaneously chooses the German target verb and marks features on its (German) complements by binding X, Y, and Z to the proper German cases. The internal form of the transfer element in the lexical entry for "give" might look like the following unit clause.[10]

```
gverb(give(nom:*,acc:*,dat:*),geb).
```

In transfer, the first argument of gverb is matched against the give form in the tree, and we get bindings X = nom, Y = acc, and Z = dat, which determine the cases of the complements. In general, logical variables associated with complements are used to control features on the transfers of those complements. The transfer tree is as follows:

```
vp(ind:s,fin(pers3,sg,pres,ind): X1,nil)
    np(n(cn),nom,sg:pers3-sg-f,X2)
        det(nom,pers3-sg-f,X2)
            d + det(nom,pers3-sg-f,X2)
        frau/l + nc(n(cn),nom,pers3-sg-f,X2)
    vp(ind:vp,fin(pers3,sg,pres,ind): X1,nil)
        geb + vc(ind:vp,fin(pers3-sg-f,pres, ind):X1,nil)
        np(n(cn),acc,sg:pers3-sg-nt,X3)
            det(acc,pers3-sg-nt,X3)
                ein + det(acc,pers3-sg-nt,X3)
            buch/h + nc(n(cn),acc, pers3-sg-nt,X3)
        ppnp(vp(ind:vp,fin(pers3,sg,pres, ind):X1,nil),dat)
            np(n(cn),dat,sg:pers3-sg-m,X4)
                det(dat,pers3-sg-m,X4)
                    d + det(dat,pers3-sg-m,X4)
                mann/h + nc(n(cn),dat,pers3-sg-m,X4)
```

The three noun phrases in this tree have the correct case markings as a result of the above verb transfer, so that we will eventually get *die Frau, ein Buch, and dem Mann.*[11] A transformation (to be discussed below) moves the dative noun phrase, and the eventual translation (after inflection) is *Die Frau gibt dem Mann ein Buch.*

The top-level procedure, transfer, of the transfer component works in a simple, recursive way, and is called in the form

```
transfer(Syn,MLab,GSyn)
```

where MLab is the German node label on the mother of the node Syn being transferred. (In the top-level call, MLab is equal to the symbol top.)

The definition of transfer, somewhat simplified, is:

```
transfer(syn(ELab,B,EMods),MLab,
        syn(GLab,B,GMods)) ←
  tranlabel(ELab,MLab,GLab) &
  tranlist(EMods,GLab,GMods).
transfer(Op-EPred,MLab,GWord + GLab) ←
  tranword(EPred,MLab,GWord,GLab).

tranlist(EMod.EMods,MLab,GMod.GMods) ←
  transfer(EMod,MLab,GMod) &
  tranlist(EMods,MLab,GMods).
tranlist(nil,*,nil).
```

Thus, transfer translates a syn structure (a nonterminal node of a tree) by translating the node label (by a call to tranlabel) and then recursively translating the daughter nodes. Terminal nodes (words) are translated by a call to tranword.

Note that transfer does the transfer in a simple top-down, left-to-right way. The German feature structures (showing case markings, for instance) that get assigned to nodes in the left-to-right processing are often *partially instantiated*, and do not get fully instantiated until controlling words are encountered further to the right. For example, the German feature structure assigned to the subject noun phrase in the above example does not get the case field assigned until the verb is processed. The use of logical variables and unification makes this easier.

The clauses for tranlabel (which transfers node labels) are mainly unit clauses. The basic problem is to transfer an English feature structure to a German feature structure, allowing for differences in a suitable way. For example, the number of an English noun phrase is often the same as the number of the corresponding German noun phrase, but not always. The main tranlabel clause that transfers a noun phrase label is:

```
tranlabel(np(Case:Sense:nf(NType,Num,*,*)&*),
          MLab,
          np(NType,Case,Num,AdjDecl)).
```

The first component, NType, of the German np feature structure (and the first component of the nf ("np features") syntactic feature structure for the English noun phrase) is the **nominal type,** which encodes categorization of the head nominal. Nominal subcategories include common nouns, pronouns, proper nouns, and adjectives. Adjectives are further subcategorized as verbal (verb participles) and nonverbal, and the comparison feature (positive, comparative, superlative) for adjectives is also shown in NType.

The second component, Case, of the German np structure is unified with the first component of the English marker. As indicated above, this gets unified with an actual German case by application of a verb transfer rule.

The third component, Num, of the German np structure encodes number, person, and gender of the German noun phrase. The tranlabel rule above unifies Num with a component of the English nf structure; but, as we will see below, Num is of such a form that 1. its occurrence in the English analysis is independent of German, and 2. the actual number of the German noun phrase can come out different from that of the English.

The last component has to do with adjective declensions (strong vs. weak). This is discussed in McCord and Wolff (1988).

The German feature structure for a noun compound (nc) (including a simple head noun) has a similar form to an np structure.

How is the Num field used to treat differences in number between English and German? This is actually a compound term of the form ANum:CStruct, where ANum is the actual number (sg or pl) of the English noun phrase (which may be a coordinated noun phrase), and CStruct is a term that reflects the coordination structure. For example, for the noun phrase "the man and the woman", the number structure is pl:(N1&N2), where the subphrase "the man" has number structure sg:N1 and "the woman" has number structure sg:N2.

Before transfer, the second components of the number structures of simple noun phrases are just variables, but during transfer these get bound by tranword to structures of the form Pers-Num-Gen showing person, number and gender of the German translations. The person and gender of the simple German noun phrase come directly from the lexical transfer entry for the head noun. The question is how the number of the

German noun phrase (possibly coordinated) is determined. For a simple noun phrase, the default is to unify the German number with the English number, but transfer entries can override this, as in the case of *scissors/Schere*. Given this determination of the German numbers of the simple np components of a coordinated np, the German number of the whole can be determined from the second component of the number field. In the case of coordination with and/*und*, the result will simply be plural in German (as in English). For coordination with or/*oder*, though, German is different from English. In English, the number of the disjunction is the same as that of its last component, whereas in German the disjunction is plural if and only if at least one of its components is plural.

Thus, for the noun phrase "the men or the woman", the English number structure is sg:(N1|N2), where the number of "the men" is pl:N1, and the number of "the woman" is sg:N2. After transfer, this structure for the translation *die Maenner oder die Frau* becomes sg: (pers3-pl-m|pers3-sg-f). The second component of this determines a final number of pl for the translation. On the other hand, the English noun phrase "the knife or the scissors" is plural, but the translation, *das Messer oder die Schere,* has number structure pl: (*-sg-*|*-sg-*) and so is singular.

In the sample transfer tree above, one can see other examples of the transfer of feature structures, for which tranlabel is responsible. In the vp feature structures, the second component is of the form Infl:Infl1, where Infl is the English inflection and Infl1 is to be the final German inflection. The default is for Infl1 to become equal to Infl, but this does not always happen. The English inflection might be overridden by a transformation. For example, LMT translates "The man wants the woman to buy a car" into *Der Mann will, daß die Frau einen Wagen kauft.* The infinitive vp complement of "wants" is transferred to an infinitive German vp, but this and its sister np subject are transformed into a finite clause complement of "will".

The transformation mentioned in the previous paragraph (needed for transforming an np + infinitive-vp structure to a finite clause) is triggered by the lexical transfer element for the controlling verb "want". Specifically, the trigger (or rule switch) is the German "case" corresponding to the last (vp) complement of "want". Any case assigned to a vp complement is unified by tranlabel with the last field of the German vp feature structure (see the sample transfer tree above for examples of such vp structures). Transformations can recognize such cases and be triggered by them.

The procedure

```
tranword(EWord,MLab,GWord,GLab)
```

is the interface to the transfer portion of the lexicon. It takes a terminal EWord representing an English word sense predication dominated by a node with associated German label MLab, and assigns to these the German translation GWord and its associated feature structure GLab. (Often GLab will be taken to be the same as MLab.) The procedure tranword, in looking at the label MLab, can call various more specific transfer procedures, like gverb and gnoun, associated with various parts of speech. Clauses for these are produced by the lexical compiler from transfer elements in the external lexicon. We have already seen a sample clause for gverb.

Lexical transfer elements can be either of **single word** or **multiword** form. Each type of English analysis element (associated with a particular part of speech) has a corresponding type of transfer element, whose name is obtained by prefixing the letter g, except that 1. proper nouns just translate to themselves, 2. modals are subsumed under gv, and 3. qualifiers are subsumed under gadv. Multiword transfer forms exist for all the multiword source forms, and have names of the form mg-part-of-speech (like mgadv).

As in the case of English analysis elements, there is a system of abbreviations and defaults for the external forms of lexical transfer elements. Let us illustrate the situation for verbs (in single word form). The full form of a gv element (external form for gverb clauses) is

```
gv(VSense,SubjCase,CompCases,Target).
```

The first argument is the verb sense. Its default is the index word. The second argument is the German case for the German complement corresponding to the English logical subject (which is usually, but not always, the German logical subject). Its default is nom (nominative). The third argument is the list of cases for the other complements (given in order corresponding to the slots of the v element for the same sense of the index word and having the same number of complements). If this argument is omitted, the verb should be intransitive. The last argument is the German target verb.

The cases appearing in the second and third arguments of gv can have associated semantic type requirements (arbitrary Boolean combinations) on the corresponding complements. An example illustrating this is the following external form entry for "eat", used to translate "eat" into *essen* or *fressen,* according as the subject is human or nonhuman.

```
eat < v(obj)
    < gv(nom:human,acc,ess)
    < gv(nom:(animate&-human),acc,fress).
```

Normally, a gv element is compiled into a (possibly conditional) clause for gverb, where the clause head has the form

```
gverb(Pred,Target).
```

Here, Pred is an English verb sense predication of the same form described for the second argument of verb in the preceding section. The logical variable components of the arguments of Pred are bound (in the gverb clause) to the German cases appearing in the gv element (in the order given). Any semantic type requirements attached to these cases are converted into Prolog goals that are combinations of isa tests on the sense variable of the associated marker, and these goals are put on the condition side of the gverb clause. For example, the gv elements for "eat" above are compiled into the following gverb clauses:

```
gverb(eat(nom:S:*,acc:*),ess) ←
  isa(S,human).
gverb(eat(nom:S:*,acc:*),fress) ←
  isa(S,animate) & ¬isa(S,human).
```

The German case symbols that can appear in transfer entries include not only the standard four cases (nom, acc, dat, and gen), but also prepositional case symbols (for pp complements of German verbs), which are of the form pc(Prep,Case). This form signifies that the specific preposition Prep appears, followed by a noun phrase with case Case. The Case component of pc can be omitted when a default case is to be used. For example, an entry for "search (something) for (something)" could be

```
search < v(obj.pobj(for))
       < gv(acc.pc(nach),durch+such).
```

The gverb clause compiled for this gv element is the unit clause:

```
gverb(search(nom:*,acc:*,pc(nach):*),durch+such).
```

There are also special genitive cases that allow for the variation in *ein Stück des weißen Papiers/ein Stück weißes Papier* ("a piece of the white paper"/"a piece of white paper"). In the first phrase the complement of *Stück* is a real genitive, but in the second phrase the complement takes the same case as Stück itself.

One allowance that has to be made is that the subject of the English verb may not correspond to the subject of the German verb. This occurs with the translation of "like" into *gefallen,* where we can translate "I like the car" into *Mir gefaellt der Wagen.* An internal-form transfer entry for the verb "like" is

```
gverb(like(dat:*,nom:X),ge + fall,*:X).
```

The extra argument of gverb is the marker (minus test) of the German subject. In such instances, tranword must make sure that the German verb (if finite) agrees with the actual German subject.

Care is taken in the tranword rules involving gverb to handle auxiliary verbs correctly. One problem is to get the correct case marking on the German subject and the correct inflection on the highest auxiliary, even though the English subject may not correspond to the German subject of the main verb.

In particular, care with case marking must be taken in the translation of passives. In a German passive, the grammatical subject may correspond to a direct object in the active form, but it may not correspond to an indirect object (as it may in English). Thus, LMT translates "The car was given to the man" into *Der Wagen wurde dem Mann gegeben,* but translates "The man was given a car" into *Dem Mann wurde ein Wagen gegeben* (where *ein Wagen* is the grammatical subject). Currently, LMT translates the English passive only by the use of *werden.* The use of *sein* and active forms will be tackled eventually.

In the translation of the perfect "have", the *haben/sein* distinction is made by feature markings on the English verb complement of "have". It could be argued that this is an exception to the principle that the English grammar is written independently of the task of translation, but the distinction made by the required features is largely semantic.

How does LMT treat situations in which there is not a word-for-word correspondence in translations? Of course transformations can add, delete, or rearrange words, and examples of these will be discussed in the next section. But, in keeping with the principle of getting as much right as possible during transfer, various methods are used in transfer, too.

One method is that the result arguments of lexical transfer elements can contain compound terms that represent word sequences. The most general form is W1#W2, which represents W1 followed by W2 (with a separating blank). This form is used, for example, in cases where the verb translation is a reflexive verb. The verb "refer to" translates to *sich beziehen auf,* and the external form of its transfer element is:

```
gv(pc(auf,acc),sich#be+zieh).
```

The lexical compiler converts this to the internal form:

```
gverb(refer(nom:F,pc(auf,acc):*),
      refl(*:F)#be+zieh).
```

The term refl(X) representing the reflexive contains the feature structure of the German subject, so that it may be realized as the correct form of the reflexive pronoun by the German morphological component.

A special compound form shows up in the representation of separable prefix verbs, which are of the form Prefix:Verb. (As exhibited above for *beziehen, in*separable prefix verbs are given in the form Prefix+Verb.) A separable prefix can become a separate word through a transformation that recognizes the special form and moves the prefix appropriately. The separable prefix verb device P:V can often be used also to specify transfers where the target is an adverb-verb combination, when the adverb behaves transformationally like a separable prefix.

One could say that the translation of noun compounds involves a many-to-one correspondence, since noun compounds are given as a group of words in English, but often as a single word in German. The procedure tranlabel is responsible for marking the nc feature structure of each noun premodifier of a noun. The case feature of such a noun premodifier is marked by a special case symbol comb (combining form), which signals that the noun is part of a noun compound and will be given a special form by the German morphological component.

The most important means for handling noncompositional translation in LMT is through the use of multiword elements in the lexicon. As indicated in the preceding section, all but three of the eleven parts of speech allow multiword forms, in transfer as well as in source analysis elements. As an example, to translate "take care of" into *sich kümmern um,* the relevant portion of the external form entry for "take" could be

```
take < mv(=.care.of,obj)
      > mgv(pc(um),sich#kümmer).
```

The connecting operator > tells the lexical compiler to process its right operand only if its left operand has "succeeded" (i.e., the mv pattern has matched).

The patterns in multiword elements can contain variables. For example, the phrase "at least N", where N is a number (like "5" or "five"), can be considered a multi-determiner. This can be translated into *mindestens M,* where M is the German form of N, by the multiword elements in an entry for "least" (showing only the relevant part of the entry).

```
least < mdet(at.=.N)-enum(N)
      > mgdet(mindestens#M)-gnum(N,M).
```

Prolog goals associated with multiword elements are indicated by attaching them with the operator -. The lexical compiler handles such goals in source elements differently from goals in transfer elements. A goal for a source element, like enum(N) in the example, is treated as a test that is executed at lexical preprocessing time after the multiword pattern successfully matches part of the sentence. If this test does not succeed, the multiword element is not compiled. A goal for a transfer element, like gnum(N,M) (which associates the English number N to the German number M), is added by the lexical compiler to the right-hand side of the clause compiled for the multiword transfer element, and thus it is not executed until transfer time. (In the above example, this postponement is not necessary; but in general it is necessary because the transfer may depend on ingredients from the rest of the sentence that are not known until a complete parse is obtained.)

Non-compositional translation can also be handled by the German word list transformations allowed in lexical entries. This facility will be described in Section 6. For more details on lexical aspects of transfer, see McCord and Wolff (1988).

5 German syntactic generation

As indicated in the Introduction, the purpose of this component is to generate a German surface structure tree from the transfer tree, and this is accomplished by applying a system of transformations. Before the transformations operate, however, a minor bookkeeping step is carried out, having to do with bracketing. Recall that syntax trees (both English and German) are represented so far as terms

```
syn(Label,B,Modifiers)
```

where B is the bracket list for the node. Transformations operate on terms similar to this, but it is convenient if they do not have to deal with bracket lists explicitly. Therefore, tree terms in the above form are converted (by the bookkeeping step in question) to tree terms in the simpler form

```
syn(Label,Modifiers1)
```

where Modifiers1 is obtained by surrounding Modifiers appropriately with the terminal pairs corresponding to the bracket symbols in the list B. The main procedure for syntactic generation,

```
generate(Syn,Syn1),
```

applies transformations to a syn tree Syn (in the simpler form), producing another syn tree Syn1. This works recursively on the tree. At each level, generate is first applied recursively to the modifiers, giving a tree with a new modifier list. Then the transformations are applied in a loop: Each time through the loop, the first applicable transformation is used (hence the order of transformations matters). The loop terminates when no transformation is applicable. Thus the definition can be given as:

```
generate(syn(Lab,Mods),syn(Lab2,Mods2)) ← /&
   genlist(Mods,Mods1) &
   alltransforms(syn(Lab,Mods1),syn(Lab2,Mods2)).
generate(Mod,Mod).
genlist(Mod:Mods,Mod1:Mods1) ←
   generate(Mod,Mod1) &
   genlist(Mods,Mods1).
genlist(nil,nil).
alltransforms(Syn,Syn2) ←
   transform(Trans,Syn,Syn1) &/&
   alltransforms(Syn1,Syn2).
alltransforms(Syn,Syn).
```

(The symbol / denotes the cut in VM/Prolog.)

Note that in this scheme transformations on a given level are applied after the recursive generation of daughter nodes. The alternative schemes of applying them all before recursive generation, or applying some designated transformations before and others after recursive generation, were tried in earlier versions of LMT. But these other schemes led to problems with a workable ordering of the list of transformations. Also, experiments were made with an additional system of transformations, applied to English trees prior to transfer, but it was found that this complication is unnecessary.

Specific transformations are given by rules for transform. Its first argument is just a name for the transformation, like verbfinal, which is used in tracing (in a fuller definition of generate).

One could write rules for transform directly, but in LMT transformational rules are written in a slightly more convenient notation and then compiled into rules for transform. The main purpose of the alternative format is to provide an augmentation of the pattern matching of Prolog in which specially marked terms can match sublists of lists. Specifically, when a term of the form %X is written syntactically as a member of a list, then X matches (unifies with) any sublist in that position. This can be used both in analyzing and constructing lists. As an example, the expression a.b.%X.c.d.nil matches the list a.b.u.v.w.c.d. nil and unifies X with u.v.w.nil. (Similar conventions have been used in many pattern matchers dealing with lists.) In this implementation, such extended list expressions can be embedded arbitrarily in Prolog terms.

The form for a transformation given to the rule compiler is:

```
Name --
A ===> B
← Condition.
```

Here, A and B are arbitrary Prolog terms, containing possible extended list expressions. The Condition is a Prolog goal, and it can be omitted if desired. This rule is compiled into a transform rule of the form

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

Here, the original pseudo-term A involving % elements 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.

As an example, a simplified version of the German dative transformation is

```
dative --
  syn(vp, %LMods.Obj.IObj.RMods)
  ===>
  syn(vp, %LMods.IObj.Obj.RMods)
  ← case(Obj,acc) & case(IObj,dat).
```

This is compiled into the transform rule:

```
transform(dative, syn(vp,Mods), syn(vp,Mods1)) ←
  conc(LMods, Obj.IObj.RMods, Mods) &
  case(Obj,acc) & case(IObj,dat) &
  conc(LMods, IObj.Obj.RMods, Mods1).
```

For efficiency, the Condition is inserted between ASplit and BSplit, because Condition normally contains constraints whose arguments become known immediately after execution of ASplit.

The transformation relclause is defined as follows (we give here somewhat simplified versions of the actual transformations).

```
relclause --
  syn(vp(dep(rel(Case,Type,PNG)),I,M), Mods)
  ===>
  syn(vp(dep(rel),I,M), (',' +punc).
                           (drel +pro(Case,PNG)).
                           %Mods.
                           (',' +punc).nil).
```

Here, PNG is the person-number-gender structure for the noun phrase modified by the relative clause. (The vp arguments I and M are as described in the preceding section.) The relclause transformation is responsible for adding a relative pronoun and surrounding commas to the relative clause. In the English analysis tree, no relative pronoun is explicitly shown. (Note that in certain cases it can be omitted in the English sentence.) Thus, "The man I saw is my brother" translates into *Der Mann, den ich sah, ist mein Bruder*. There are variants of this relative clause transformation dealing with cases like "The book to which I referred is old", which translates to *Das Buch, auf das ich mich bezog, ist alt*. LMT also gives exactly this same translation for each of the English sentences: "The book which I referred to is old", "The book that I referred to is old", and "The book I referred to is old".

There is a similar transformation compclause, which adds the word *daß* and commas to a finite complement clause. Thus, "Hans knows Peter is my brother" translates into *Hans weiß, daß Peter mein Bruder ist.*

The example just given illustrates the need to move the verb to the end of a dependent clause. This is done by the transformation verbfinal, defined as follows:

```
verbfinal --
  syn(vp(dep(T),I,M), %Mods1.Verb.Mod.Mods2)
  ===>
  syn(vp(dep(T),I,M), %Mods1.Mod.Verb.Mods2)
    ← synlabel(Verb,vc(*,*,*)) &
        ¬clausal(Mod) & ¬allpunc(Mod.Mods2).
```

The idea is simply that the verb Verb hops over the modifier Mod to its right, provided that Mod is not clausal (to be explained) and provided that the remaining modifiers (including Mod) do not consist solely of punctuation. For example, in the translation of the noun phrase "the man that gave the woman the book", the verb *gab* moves all the way to the end of the relative clause, producing: *der Mann, der der Frau das Buch gab*. Note that verbfinal may apply several times, until the verb has moved as far as it can go. (It is possible to be a bit more efficient by writing an auxiliary procedure to perform the movement.)

The point in not hopping over clausal elements in verbfinal is illustrated with the translation of the noun phrase "the man that told me that Hans bought a car", which is *der Mann, der mir sagte, daß Hans einen Wagen kaufte*. Here, *sagte* hops over *mir*, but not over the *daß* clause. Roughly, clausal elements are phrases whose heads are verbs.

But an interesting situation for verbfinal arises when there is a clausal element that is on the right end of the tree but is not a sister of the verb being moved. This occurs in the translation of the noun phrase "the man that gave the woman the book I referred to". Here *gab* should not move past the final relative clause, and the result should be: *der Mann, der der Frau das Buch gab, auf das ich mich bezog*. The final clausal element could actually be embedded several levels. To handle this, there is a transformation clauseraise, ordered before verbfinal, which raises such final clauses. Its definition is simply:

```
clauseraise --
   syn(Lab, %Mods.syn(Lab1,%Mods1.Syn.nil).nil)
   ===>
   syn(Lab, %Mods.syn(Lab1,Mods1).Syn.nil)
     ← clausal(Syn).
```

This may operate through several levels before the result of the raising is pertinent to verbfinal.

The transformation verbfinal also handles (without extra effort) the word ordering in auxiliary verb constructions, because of the treatment of auxiliary verbs as higher verbs. Thus, the sentence "Hans will have bought the car" is structured as Hans [will [have [bought the car]]]. Before transformations, the translation will be structured as *Hans [wird [haben [gekauft den Wagen]]]*. The verb phrases headed by *haben* and *gekauft* are dependent, so verbfinal operates on them to give: *Hans [wird [[den Wagen gekauft] haben]]*. (The phrases hopped over are not cases of clausal elements.)

Right movement of separable verb prefixes in independent clauses is similar to right movement of the verb in dependent clauses. This is handled by two transformations, one to separate the prefix, the other to move it. In this case, too, the moved item does not hop over clausal elements. An example for the separable prefix verb *aufbereiten* ("edit") is in the LMT translation of the sentence "Hans edited the file that he had created", which is *Hans bereitete die Datei auf, die er erstellt hatte*. In dependent clauses, the separable prefix stays with the verb, although inflection has to treat it specially for past participles and *zu* infinitive forms.

Ordering of transformations is important in that clauseraise must be ordered before verbfinal and the separable prefix transformations. In turn, it is important to order the dative transformation before all of these. If the dative noun phrase to be moved contains a final clausal element, then this element should not be a barrier to rightward movement of a verb or separable prefix. If dative operates first, the final clausal element will go with the whole dative noun phrase, and will not have a chance to be raised by clauseraise, which only sees clausal elements on the extreme right of the clause in which it operates. Thus, for the sentence "Hans knew that Peter had given a book to the woman he saw", the translation is *Hans wußte, daß Peter der Frau, die er sah, ein Buch gegeben hatte*.

Another example of a transformation is verbsecond, which operates in independent clauses.

```
verbsecond --
   syn(vp(ind(s),I,M), Mod.%Mods1.
         syn(vp(ind(vp),I1,M1),%Mods2.Verb. Mods3).
         Mods4)
   ===>
   syn(vp(ind(s),I,M), Mod.Verb.%Mods1.
         syn(vp(ind(vp),I1,M1),%Mods2.Mods3).Mods4)
     ← Mods1 = /nil & synlabel(Verb,vc(*,*,*)).
```

As the name indicates, this moves the verb so that it is the second modifier of the independent clause. As a result, the sentence "Probably the file was created by Hans" translates into *Wahrscheinlich wurde die Datei von Hans erstellt*, where *wurde* is moved into second position by verbsecond.

An interesting example of a transformation is subcl, which adds pronouns in examples like

"The man wants the woman to speak with Hans before buying the car."

Der Mann will, daß die Frau mit Hans spricht, bevor sie den Wagen kauft

Here, the translation of the subordinate participial clause "before buying the car" is the finite clause *bevor sie den Wagen kauft* (before she buys the car), where the pronoun *sie* (she), referring to the subject of the matrix clause, is added. The English analysis shows a variable in the analysis of the participial clause which is unified with one in the subject of the matrix clause. This variable serves as the link to transmit the appropriate person-number-gender to the subordinate clause in the transfer tree. The transformation subcl can then easily add the correct pronoun.

A final example of a transformation is possessive, which deals with left-branching possessive noun phrase constructions, as in "my oldest brother's wife's father's car". The possessive transformation is responsible for converting such structures into a sort of right branching mirror image, where extra definite articles are added: *der Wagen des Vaters der Frau meines ältesten Bruders*. The definition of possessive, without its condition, is as follows:

```
possessive --
   syn(NPLab, PossNP.NC.Mods)
   ===>
   syn(NPLab, Det.NC.PossNP.Mods).
```

The condition (not shown) tests that the components of the pattern are what their names suggest, assigns the genitive case to the possessive noun phrase PossNP , and creates a definite article Det that agrees with the whole noun phrase.

6 German morphological generation

The task of this component is to take the output tree from the syntactic generation component and to produce the character string representing the final German translation. There are three substeps for accomplishing this.

The first and most substantial step is the application of morphological procedures, mainly inflectional, to the individual nodes of the tree, which are of the form Base+Features, producing another tree whose terminals are inflected German words. The main procedure for this step, gmorph, takes such a Base+Features structure and produces the required inflected word. It accomplishes this mainly by dispatching the problem to various procedures like gverbf (German verb form) associated with different parts of speech, as determined appropriately from Features. Before calling these procedures, though, gmorph performs several "tidying-up" operations, such as simplifying tense and case structures and handling compound words (through recursive calls).

Details will not be given here for the inflectional procedures for the various parts of speech, but it is worth saying a bit about the noun declension system in LMT, since the most idiosyncratic part of German inflectional morphology is the system for nouns. It was mentioned in Section 3 that German noun class information is exhibited along with target nouns in gn transfer elements of the lexicon, in a compact format. The transfer component marks this information in the transfer tree (see the example tree in Section 4), where it can be read off by the noun inflection procedures. In general, a gn target is of the form

```
Noun.Gender.DeclensionClass.
```

For example, the transfer of brother is *bruder.m.b.* The declension class is usually specified by a single letter. In the case of *Bruder,* for example, the class b dictates that the plural base is formed by umlauting the noun, and there is a general rule for finding where to place the umlaut. In general, the declension class has implicit within it the method of getting the plural base, the combining form (used in forming noun compounds), and the complete declension pattern. In examples where the plural base or combining form is very irregularly formed, the declension class may be given as a letter together with the required morpheme, as in datum.nt.x.daten.

Susanne Wolff has worked out a system of 20 noun classes under the preceding scheme, together with the morphological rules for getting declensions for each class. There are also some rules that compute the gender/class for nouns with certain common endings, so that for these nouns the gender/class can be omitted in the transfer entry.

The second substep of German morphological generation is the application of German word list transformations. The tree output of the first substep (whose terminals are inflected German words) is converted to a linear list of words by this second substep. This is done recursively. On each level, all the daughter nodes are converted to word lists and these are concatenated, producing a tentative word list Words for the node. But then an attempt is made to apply a word list transformation to Words, by calling a procedure

```
gphrase(Label,Words,Words1)
```

where Label is the (principal functor of the) label on the current node. If this succeeds, the desired word list for the node is Words1; otherwise it is Words.

Currently, gphrase rules are used mainly for handling German contractions. For example, there is a rule

```
gphrase(pp, an.dem.U, am.U).
```

But these rules can also be used to handle noncompositional translations. For example, "for example" can translate compositionally into *für Beispiel,* and then a gphrase rule can convert this to *zum Beispiel.* As mentioned in Section 3, word list transformations can be specified in the lexicon. There is a slightly shorter format (like gph(pp,an.dem,am)) which is compiled into gphrase clauses by the lexical compiler.

It seems better on the whole, however, to treat noncompositional translation by means of multiword elements in the lexicon, since these involve both source and transfer elements. It is useful to involve source elements, because in many cases the source phrase is idiomatic in itself, and the parser is helped by having an English multiword analysis.

The last substep, a rather simple one conceptually, is to convert the German word list for the whole sentence into a simple character string. This involves mainly the treatment of punctuation, blanks, capitalization, and text formatting symbols. For a more detailed description of German morphological generation, see McCord and Wolff (1988).

7 Status of the system

LMT handles all of the examples and constructions given above, and many other types of constructions not illustrated for lack of space. Testing and vocabulary development have been done with the IBM CMS Editor (XEDIT) reference manual, as well as with a collection of sentences made up by ourselves and others to illustrate key grammatical constructions and problems of English-German translation. Every effort has been made to keep the rules of the system general. As with most MT systems, it is assumed that there will be some postediting of the output.

In a test on a 500-sentence corpus from an initial part of the XEDIT manual, LMT was able to translate 95% of the sentences in an "understandable"[12] way, with an average processing time on an IBM 3081 of 364 milliseconds per sentence (19.5 msec. per word), using VM/Prolog as an interpreter.

The first few sentences and their LMT translation (with no postediting) are as follows:

XEDIT subcommands and macros follow the same rules and conventions. For purposes of this discussion, "subcommand" refers to both XEDIT subcommands and XEDIT macros. The general format of XEDIT subcommands is: (fig.) At least one blank must separate the subcommand name and the operands, unless the operand is a number or a special character. For example, NEXT8 and NEXT 8 are equivalent. At least one blank must be used to separate each operand in the command line unless otherwise indicated. The maximum length of an XEDIT subcommand issued from an EXEC procedure or from an XEDIT macro is 256 characters.

XEDIT Unterbefehle und Makros folgen den gleichen Regeln und Konventionen. Zum Zweck dieser Diskussion bezieht sich "Unterbefehl" sowohl auf XEDIT Unterbefehle als auch auf XEDIT Makros. Das allgemeine Format von XEDIT Unterbefehlen ist: (fig.) Mindestens ein Leerzeichen muß den Unterbefehls-Namen und die Operanden abtrennen, es sei denn der Operand ist eine Zahl oder ein spezielles Zeichen. Zum Beispiel sind NEXT8 und NEXT 8 äquivalent. Mindestens ein Leerzeichen muß verwendet werden, jeden Operanden in der Befehls-Zeile abzutrennen, wenn nicht anderweitig angezeigt. Die maximale Länge eines XEDIT Unterbefehls, der von einer EXEC Prozedur oder von einem XEDIT Makro ausgegeben wird, ist 256 Zeichen.

As for the size of LMT, there are now about 3,500 Prolog clauses, not including the lexicon. This includes the MLG and DCG grammar rules as clauses, and there are about 270 of these. After metarules have operated, the total number of grammar rules is about 450. The lexicon currently contains about 1,600 entries; this includes most of the vocabulary for the XEDIT Reference Manual. As mentioned earlier, ModL is interfaced to the UDICT monolingual English lexicon (Byrd 1983, 1984), with around 65,000 citation forms. Also an interface of LMT to a lexical data base (Neff, Byrd, Rizk 1988) for the Collins English-German Dictionary has been partially developed by Susanne Wolff and the author. Currently, this interface, given an English word, just takes the first German translation provided, for each part of speech. For nouns, the required LMT German noun classes are obtained from a data base worked out by Wolff from the inflectional information on the German-English side of Collins.

In recent work (McCord 1988), LMT has been expanded to deal with target languages besides German. This expansion has been made easier by the development of a large subsystem LMTX of LMT which is essentially target language independent and can be thought of as an "English-to-X translation shell." Development of the shell has involved improvements and generalizations in the modules of LMT, but most of the methods and organization are as described in the current paper.

The shell LMTX includes: 1. the English grammar ModL; 2. most of the source/transfer morphology system and lexical processing system; 3. the transfer algorithm and rule system, except for low level, lexical transfer entries; 4. the syntactic generation algorithm; 5. target independent procedures dealing with morphological generation, and 6. many utility procedures. For a given target language, the only target specific modules are a. the source/transfer (unified) lexicon; b. the set of transformations for syntactic generation, and c. the target morphological system. As an example of the size of the shell, for the English-German version of LMT the shell contains approximately 80% of the rules (not counting the lexicon).

Using the shell, prototype versions of LMT have been started up for several target languages, in cooperation with other groups and individuals: French—in cooperation with the KALIPSOS group of the IBM Paris Scientific Center (Fargues et al. 1987), with work especially by Eric Bilange; Danish—with Arendse Bernth and in cooperation with IBM European Language Services; Spanish—with Nelson Correa; and Portuguese—with Paula Newman's group at the IBM Los Angeles Scientific Center.

References

Bennett, W. S. and Slocum, J. 1985 "The LRC Machine Translation System," *Computational Linguistics* 11: 111–121.

Bernth, A. 1988 "LODUS—A Logic-oriented Discourse Understanding System," Research Report RC 13676, IBM Research Division, Yorktown Heights, NY.

Byrd, R. J. 1983 "Word Formation in Natural Language Processing Systems," In *Proceedings of the 8th International Joint Conference on Artificial Intelligence*, Karlsruhe: 704–706.

Byrd, R. J. 1984 "The Ultimate Dictionary Users' Guide," IBM Research Internal Report.

Colmerauer, A. 1971 "Les systèmes-Q: un formalisme pour analyser et synthétiser des phrases sur ordinateur," Groupe TAUM, Université de Montréal, Québec, Canada.

Colmerauer, A. et al. 1971 "TAUM–71," Groupe TAUM, Université de Montréal, Québec, Canada.

Colmerauer, A. 1975 "Les grammaires de métamorphose," Internal Report, Groupe d'Intelligence Artificielle, Université d'Aix-Marseille, France.

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

Dahl, V., and McCord, M. C. 1983 "Treating Coordination in Logic Grammars," *American Journal of Computational Linguistics* 9: 69–91.

Dahlgren, K. 1988 *Naive Semantics for Natural Language Understanding*, Kluwer Academic Publishers, Norwell, MA.

Fargues, J.; Bérard-Dugourd, A.; Landau, M. C.; Nogier, J. F.; Catach, L. 1987 "KALIPSOS Project: Conceptual Semantics and Linguistics," In *Proceedings of the Conference on Artificial Intelligence and Natural Language Technology*, IBM European Language Services, Copenhagen, Denmark.

Gazdar, G., and Pullum, G. K. 1982 "Generalized Phrase Structure Grammar: A Theoretical Synopsis," Indiana University Linguistics Club, Bloomington, IN.

Hirschman, L. 1986 "Conjunction in Meta-restriction Grammar," *The Journal of Logic Programming* 3: 299–328.

Huang, X-M. 1984a "The Generation of Chinese Sentences from the Semantic Representations of English Sentences," In *Proceedings of the International Conference on Machine Translation*, Cranfield, England.

Huang, X-M. 1984b "Dealing with Conjunctions in a Machine Translation Environment," In *Proceedings of the Joint Association for Computational Linguistics and Conference on Computational Linguistics Meeting 1984*: 243–246, Stanford, CA.

Huang, X-M 1985 "Machine Translation in SDCG Formalism," in Nirenburg (1985, these references):135–144.

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

Isabelle, P. and Bourbeau, L. 1985 "TAUM-AVIATION: Its Technical Features and Some Experimental Results," *Computational Linguistics* 11: 18–27.

Kittredge, R.; Bourbeau, L.; and Isabelle, P. 1973 "Design and Implementation of a French Transfer Grammar," *Conference on Computational Linguistics Meeting 1976*, Ottawa, Canada.

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

McCord, M. C. 1981 "Focalizers, the Scoping Problem, and Semantic Interpretation Rules in Logic Grammars," Technical Report, University of Kentucky, Lexington, KY. Appeared in *Logic Programming and Its Applications*, M. van Caneghem and D. H. D. Warren (Eds.), Ablex, 1986.

McCord, M. C. 1982 "Using Slots and Modifiers in Logic Grammars for Natural Language," *Artificial Intelligence* 18: 327–367.

McCord, M. C. 1984 "Semantic Interpretation for the EPISTLE System," In *Proceedings of the Second International Logic Programming Conference*, Uppsala, Sweden: 65–76.

McCord, M. C. 1985a "Modular Logic Grammars," In *Proceedings of the 23rd Annual Meeting of the Association for Computational Linguistics*, Chicago, IL: 104–117.

McCord, M. C. 1985b "LMT: A Prolog-based Machine Translation System" (extended abstract), in Nirenburg (1985, these references): 179–182.

McCord, M. C. 1986 "Design of a Prolog-based Machine Translation System," In *Proceedings of the Third International Logic Programming Conference*, Springer-Verlag, Berlin, W. Germany: 350–374.

McCord, M. C. 1987 "Natural Language Processing in Prolog," in Walker et al. (1987, these references).

McCord, M. C. 1988 "A Multi-target Machine Translation System," In *Proceedings of the International Conference on Fifth Generation Computer Systems 1988*, Tokyo, Japan: 1141–1149.

McCord, M. C. 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.

Neff, M. S.; Byrd, R. J.; and Rizk, O. A. 1988 "Creating and Querying Lexical Data Bases," In *Proceedings of the Second Conference on Applied Natural Language Processing*, Austin, TX.

Nirenburg, S. 1985, (ed.) *Proceedings of the Conference on Theoretical and Methodological Issues in Machine Translation of Natural Languages*, Colgate University, Hamilton, NY.

Pereira, F. C. N. and Shieber, S. M. 1987 *Prolog and Natural Language Analysis*, CSLI Lecture Notes, no. 10, Menlo Park, CA.

Pereira, F. C. N. and Warren, D. H. D. 1980 "Definite Clause Grammars for Language Analysis—A Survey of the Formalism and a Comparison with Transition Networks," *Artificial Intelligence* 13: 231–278.

Sedogbo, C. 1984 "A Meta Grammar for Handling Coordination in Logic Grammars," In *Proceedings of the Conference on Natural Language Understanding and Logic Programming*, Rennes, France: 137–149.

Teeple, D. 1985 "Reasoning in Embedded Contexts," Research Report RC 11539, IBM Research Division, Yorktown Heights, NY.

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, MA.

Wilks, Y.; Huang, X-M.; and Fass, D. 1985 "Syntax, Preference and Right-Attachment," In *Proceedings of the 9th International Joint Conference on Artificial Intelligence*, Los Angeles, CA.

Wolff, Susanne 1983 *Lexical Entries and Word-Formation*, (Ph.D. dissertation), New York University.

Notes

1. This paper is a revision of a paper (invited presentation) that appeared in the *Proceedings of the Third International Logic Programming Conference*, London, July 1986, published by Springer-Verlag, *Lecture Notes in Computer Science*. The current version reflects recent improvements in LMT.
2. The description "Logic-programming-based" is slightly more accurate.
3. There is an interesting historical connection between machine translation and Prolog. Prior to the development of Prolog, Alain Colmerauer worked in the period 1967–70 on a machine translation project, the TAUM project—Traduction Automatique Université de Montréal (Colmerauer et al. 1971, Kittredge, Bourbeau and Isabelle 1973, Isabelle and Bourbeau 1985). In connection with this project, Colmerauer developed a grammar language, **Q-systems** (Colmerauer 1971), which had some of the features of logic grammars and Prolog. In his subsequent work on the development of Prolog, natural language applications formed a major motivation for Colmerauer.
4. In earlier work on MLGs, only the first argument of a strong nonterminal was used as a feature argument, so that the /k specification was not used in strong nonterminal declarations.
5. Currently, there is only one occurrence of this device in ModL, which could be probably be avoided without too much trouble.
6. The ordering of parses is significant in ModL. Normally translation is done only for the first parse obtained.
7. Markers are logical variables together with semantic and syntactic feature information. The exact format used currently in ModL is described at the end of this subsection.
8. As described in the next section, the lexical preprocessor can optionally make this verb sense argument simply Y or make it the whole term Y:Sense:SynFeas, depending on the application of ModL.
9. For more details, see McCord and Wolff (1988).
10. External forms for transfer elements are discussed below.
11. The symbols l and h attached to the nouns are their declension classes. More details are given in Section 6.
12. This means that a native German speaker can read the translation (without seeing the source) and understand it in the same sense as the source.