Generation

Eric NYBERG 3rd, Rita McCARDELL, Donna GATES and Sergei NIRENBURG

ABSTRACT: The structure and function of the target-language generation module for KBMT-89 is described. The lexical selection module (which includes thematic-role subcategorization, a meaning distance metric, and syntactic subcategorization) is presented. We also describe the generation mapping rules, and rule interpretation in the generation of f-structures for target language utterances.

KEYWORDS: functional structure (f-structure), generation, lexical selection, mapping rules, subcategorization

1. OVERALL DESIGN

The goal of the generation module of the KBMT-89 system is to produce target language sentences from the interlingua text (ILT) frames that are produced by the analysis module and augmented by the augmentor. There are three main steps in generation:

1. *Lexical Selection.* For each concept in the ILT, the most appropriate lexical item must be selected.

2. *F-Structure Creation.* A syntactic functional structure which determines the grammatical structure of the target utterance must be produced from the ILT frames.

3. *Syntactic Generation.* The syntactic f-structure is processed by the generation grammar to produce a target language sentence.

The design of the generation module combines recent research in the area of lexical selection\(^1\) with a map-and-generate paradigm used in previous translation systems at the Center for Machine Translation. However, the mapping phase (which creates an f-structure from the ILT) has been completely re-designed, and provides a more powerful mapping rule paradigm than was previously implemented for generation. In addition, the latest version of the GENKIT grammar compiler is now used to produce the syntactic generator from the generation grammar.\(^2\)

---

\(^1\)The lexical selection module of the system is a modified version of the lexical selection knowledge source in DIogenes-88 (Nirenburg, et al., 1988).

\(^2\)GENKIT is described in Tomita and Nyberg, 1988.
The control structure of the generator is general and extensible, and will support both further research and more sophisticated use of knowledge in generation. However, the restricted focus of the present application domain makes unnecessary the consideration of text ordering and other trans-sentential phenomena. Furthermore, the relatively detailed granularity of the concept lexicon made the use of collocational restrictions unnecessary during generation3 (cf. Figure 10 in Nirenburg's paper "Knowledge-Based Machine Translation" in Part I of this issue, illustrating the process of lexical selection).

The architecture of the generation module is shown in Figure 9 in the introductory paper by Nirenburg in this issue. The ILT frames are first input to the lexical selection module, which directly stores the appropriate lexical choices in the frames themselves. The ILT frames are then input to the mapping rule interpreter, which fires the mapping rules for the target language and creates an f-structure from the ILT frames. This f-structure is then passed through to the compiled generator, which produces an output sentence in the target language.

The structure of this paper is as follows. In Section 2, we describe the lexical selection module of the system. In Section 3, we describe the mapping rules that are used for f-structure creation. In Section 4, we discuss the process of mapping f-structures into target language strings using the GENKIT generation grammar compiler. Finally, in Section 5 we describe the overall control structure of the generator and how these three modules are integrated into a complete system.

2. THE LEXICAL SELECTION MODULE

While in natural language analysis the main task is robust lexical disambiguation, the major lexicon-related task of a natural language generator is to perform principled selection of target language lexical items, based on lexical, semantic, pragmatic and discourse knowledge available in the input.

In this section, we discuss the types of input meanings that must be realized, along with the use of subcategorization knowledge and a meaning matching metric for lexical selection. Then we present a formal description of the algorithms used for lexical selection in our system.

2.1. Types of Meanings and Types of Realizations

The ILT (as described in the paper "Knowledge Representation Support," Nirenburg and Levin, in Part I of this issue) contains information about the meanings of propositions, as well as non-propositional information about modality, speech acts, focus, discourse cohesion markers and so forth. Propositional meanings are typically realized as open-class items (nouns, verbs, adjectives, adverbs) and sometimes as closed-class items (determiners, prepositions, conjunctions, etc.).

3A description of the use of collocational restrictions in generation can be found in (Nirenburg, et al. 1988)
Non-propositional meanings can be realized with the help of lexical items, word order or syntactic structures.

In this paper, we concentrate on the selection of open-class lexical items. Our working system, the KBMT-89 generator, does a substantial amount of processing of non-propositional meaning as well, in the spirit of Pustejovsky and Nirenburg, 1987. However, we cannot claim a sufficient degree of coverage of that set of phenomena to describe them in detail here. This will become one of the major directions of future research.

The lexical selection module of a natural language generation system should make use of both context-dependent and context-independent information. Thematic role subcategorization is an example of the use of context-dependent information in generation. Lexical selection based on the “distance” between a given input meaning and a generation lexicon entry is an example of a context-independent process, since it does not take into account any constraints on other ILT meanings in selecting the realization for a given ILT component.

The next two sections illustrate how these types of information sources influence the process of lexical selection.

2.2. Context-Dependent Selection: Subcategorization

Lexical selection using information about thematic-role subcategorization is the problem of comparing the thematic roles of various lexical candidates with the roles of the given input role or proposition. Each noun and verb in the generation lexicon is subcategorized for the thematic roles that are either required, optional or forbidden.

Subcategorization phenomena are discussed in the paper “Analysis and Generation Grammars” by Gates et al. in Part I of this issue. But another example will be helpful: Verbs like move expect an AGENT and a THEME and optionally a SOURCE and a GOAL. Verbs like appear, on the other hand, occur only with a THEME (never an AGENT) and verbs like be can require a DOMAIN and a RELATION-RANGE, as in The keyboard is the primary input to the system. There are of course many other subcategories of English verbs and nouns.

When a lexical item is selected for a particular ILT frame, the contents of that frame must be checked against subcategorization requirements of the lexical item. For example, if a candidate verb subcategorizes for an AGENT and THEME, the ILT frame must contain both an AGENT slot and a THEME slot for the verb to be selected.

There are two sources of knowledge for subcategorization that must be defined for each language to be generated:

- A list of all possible thematic roles used to subcategorize each category (or part of speech) in the language. This information is defined with the subcat macro and is usually placed at the beginning of the generation lexicon. For example, the thematic roles used for English verb subcategorization in our domain are defined as follows:
A specific subcategorization pattern for each generation lexicon entry, where appropriate. The subcategorization pattern lists the slots required in the ILT frame and which slots are optional. If a thematic role is not mentioned, then it must not appear. For example, consider the generation lexicon entry for the verb *add*:

(make-frame add
  (is-token-of (value *APPEAR*))
  (lexeme (value "add"))
  (syntactic-info (CAT V)
    (FEATURES (valency trans) (root add)))
  (subcategorization-info (req AGENT THEME) (opt GOAL))
  (morphological-info (infl regular))))

The verb *add* requires that an AGENT and THEME be present, and that a GOAL is optional. It also specifies that no roles other than AGENT, THEME and GOAL be present.

Nouns in the generation lexicon contain required and optional thematic roles as well. The lexical units *note* and *message* will serve as illustrations of how thematic-role subcategorization works for nouns:

(make-frame note
  (is-token-of (value *TEXT-GROUP*))
  (convey (value *COMMUNICATIVE-CONTENT) (importance 10))
  (lexeme (value "note"))
  (syntactic-info (CAT N)
    (FEATURES (count yes) (root note)))
  (subcategorization-info (req CONVEY))
  (morphological-info (infl regular))))

(make-frame message
  (is-token-of (value *TEXT-GROUP*))
  (lexeme (value "message"))
  (syntactic-info (CAT N)
    (FEATURES (count yes) (root message)))
  (subcategorization-info (opt CONVEY STRING-IS))
  (morphological-info (infl regular))))
Message can optionally take convey and string-is slots. However, note must have a convey slot filled with the concept *communicative-content. When no convey slot is mentioned within the ILT, then the lexeme message is chosen because its thematic-role slots are optional. When both convey and string-is are present, only message can be chosen. (Note is not subcategorized to take a string-is slot.) For example, notice the difference between the 'IBM PERSONAL COMPUTER BASIC' message and the note.

When only the convey slot is present within the ILT, then either word can be selected. The generation lexicon entry for note, however, constrains the convey slot to be filled with the concept *communicative-content. If this is not the case, then message will be selected by virtue of the meaning matching metric. However, if the slot is filled with a role whose concept is *communicative-content, then note will be selected.

2.3. Context-Independent Selection: The Matcher

Context-independent lexical selection is the task of comparing the meaning of each lexical realization candidate with the input meaning, in order to determine the best lexical choice. Our system matches an ILT frame's meaning pattern slots against the meaning pattern slots of candidate generation lexicon entries. In the absence of a perfect match, the matcher must determine exactly how far the input ILT frame is from a perfect match with the generation lexicon pattern. Mismatches are assigned penalties proportional to how much the two frames differ in meaning.

The ILT frames and the meaning patterns of generation lexicon entries are collections of slots whose fillers are members of domains which are predefined for each slot. Slot fillers can be symbols (e.g., (temperature (value tepid))) or numerals (e.g., (weight (value 141.5))).

The domains (or value sets) from which slot filler values are taken can be unordered, ordered discretely (hereafter, ordered) or ordered continuously (hereafter, continuous). An example of an unordered set would be occupations; an ordered set months-of-the-year; and a continuous set height. Note that the slot fillers of a continuous set can contain only numeric values.

The cardinality of slot fillers in meaning patterns can be single as in (sex (value female)), enumerated as in (subjects (value physics math history)) or range as in (age (value (0 100))).

The matching metric is designed on the basis of the following heuristics:

- If the types of slot fillers or the domains of the filler values in the ILT and generation lexicon meaning pattern differ, then the match is declared inadequate and a maximum distance is assigned.
- The quality of the match is proportional to the size of the intersection of the actual slot fillers in the ILT and generation lexicon frames, which is
then modified by the size of the domain itself.

- Each slot in a generation lexicon meaning pattern is rated with respect to its importance for the meaning expressed by the frame. A mismatch on a less important slot will have a smaller influence on the overall score of the match.

- The quality of a match between two frames is a weighted sum of the quality of the matches of the individual frame slots.

For the matching program, it is assumed that the ILT frame contains all the slots found in the generation lexicon meaning pattern. Conversely, the ILT frame can (and regularly will) contain more slots than the generation lexicon frame. This is because an ILT frame corresponds not to a single lexical unit, but rather to an entire phrase (such as a noun phrase, complete with various modifiers). Lexical selection proceeds in two phases: First, the heads of phrases (typically, nouns and verbs) are generated and then the modifiers (typically, adjectives and adverbs, but frequently such recursive elements as prepositional phrases and relative clauses) are selected.

Fillers are compared for each slot present in both the ILT and the generation lexicon entry. If the fillers are equal, then the slots match perfectly, and no penalty is assigned. If the fillers do not match perfectly, a penalty is computed for that slot. The function used to calculate the penalty depends on the type of slot filler (e.g., single, enumerated, or range). Once the penalty is computed, it is weighted according to the importance of that slot to a successful match (i.e., the importance value within the generation lexicon entry), and added to the total penalty for the match.\(^4\)

### 2.4. The Lexical Selection Algorithm

The function generation-lexicon-search searches the generation lexicon and produces an initial set of candidate lexical realizations by obtaining all entries that are instances of the same concept as the ILT role or proposition (i.e., that have the same is-token-of value).\(^5\)

Subcategorization information is processed by the function subcat-match to filter the initial set of generation lexicon entries returned by the function generation lexicon-search to only those whose subcategorization patterns are satisfied by the ILT frame to be generated.

After subcat-match processes any relevant subcategorization information, the function select-best is called to perform context-independent lexical selection based on the quality of the match between the meaning pattern

---

\(^4\)A detailed description of the metric for this and related functions is given in the KBMT-89 Project Report and is to be published elsewhere.

\(^5\)The complete KBMT-89 project report contains full formal descriptions of the lexical selection algorithms. For reasons of space limitation, they are omitted here.
of the ILT role or proposition frame and the weighted meaning patterns of
the generation lexicon entries in the candidate set. A penalty is computed for
each potential candidate, and the candidate with the lowest final penalty is then
returned.

Once the head of the lexical item has been realized (typically, as a noun or a
verb), modifiers may be selected. The function unrealized-slots checks
to see if there exist properties (or slots) in the ILT role or proposition which are
not accounted for by the lexical selection for the head. If all properties have
been accounted for, then lexical processing terminates for that ILT frame. Oth-
erwise, each remaining property will be realized as one lexical unit, i.e., a single
word. The KBMT-89 generator will realize two different types of modifiers —
attribute and relational. Attribute modifiers are derived from such properties as
'color,' 'size,' etc., while relational modifiers are those derived from features
such as 'part-of,' 'made-of,' etc. The modifier type (either attribute or rela-
tion) of each unrealized slot is determined and the appropriate lexical selection
function is triggered. The functions attribute-lexical-selection and
relational-lexical selection use essentially the same set of algo-
rithms, i.e., generation lexicon-search and select-best, for choosing phrase mod-
ifiers as were employed for selecting phrase heads, with some slight modific-
tions.6

3. F-STRUCTURE CREATION

This section describes the rules that are used to map ILT frames into syntactic
f-structures following the lexical selection process. We discuss the formal spec-
ification for the mapping rules, as well as some example rules. We also present
the algorithm for the mapper itself.

3.1. Mapping Rules

Here we describe the Backus-Naur Form (BNF) for mapping rules and then
offer examples to illustrate how the various fields within a rule should be used.

The mapping rules used for generation are described by the following BNF
(square brackets indicate optionality):

\[
\text{<mrule>} := (\text{MAPRULE } \text{<target>} \text{<index>} \text{<mr-type>} \text{<predicate>} \text{<mslot>*})
\]
\[
\text{<target>} := E \mid J
\]
\[
\text{<index>} := \text{<ilt-type>} \mid \text{<lex-class>}
\]
\[
\text{<mr-type>} := \text{:EXCLUSIVE } \mid \text{:ANY}
\]
\[
\text{<ilt-type>} := \text{*CLAUSE } \mid \text{*PROPOSITION } \mid \text{*ROLE } \mid \text{*MOD } \mid \text{etc.}
\]
\[
\text{<lex-class>} := \text{any lexical root (e.g., THROW)}
\]
\[
\quad \text{or lexical class (e.g., CHANGE-OF-LOCATION)}
\]
\[
\text{<predicate>} := \text{LISP expression, evaluated,}
\]

---

6Space limitations preclude a full discussion of modifier lexical selection here.
may include valid @ abbreviation (see below)
<mslot> ::= <feature> | <slot> | <mapping>
<feature> ::= (FEATURE <feature-name> [<value>])
<slot> ::= (SLOT <slot-name> [<value>])
<mapping> ::= ([lhs] => <rhs> [post-process])
(lhs) ::= <slot> | <expr>
(rhs) ::= <slot> | <feature>
<expr> ::= (EXPR <expr-form>)
<expr-form> ::= LISP expression, evaluated,
may include valid @ abbreviation (see below)
<value> ::= a LISP expression, unevaluated
<feature-name> ::= a symbol denoting an LF feature
<slot-name> ::= a symbol denoting an ILT or f-structure slot
<post-process> ::= LISP expression, evaluated,
may access !FRAME, !ILTSLOT, !FSSLOT, !SLOTFS
<mclass> ::= (MAPCLASS <target> <lex-class> (<lex-class>*)

3.1.1. Mapping Rule Fields

The <target> field is used to store the rule along with the other rules for the
given target language.\footnote{The symbols 'E' and 'J' are used to identify rules for English and Japanese, respectively.}

The <index> field is used to associate a rule with the entity being mapped. There are two methods of indexing: by ILT type and by lexical class. In the first case, the type of the ILT frame being mapped is used to access rules for mapping; in the second case, the lexical item selected for the head of the phrase for the frame is used.

The <ilt-type> field should be a symbol denoting an ILT frame type (e.g., *ROLE, *PROPOSITION, etc.).

The <lex-class> field should be a symbol denoting either a lexical root (e.g., THROW) or a lexical class (e.g., CHANGE-OF-LOCATION) that the rule should fire on.

The <mr-type> field is used to control the behavior of :EXCLUSIVE and :ANY rules when they are fired.

Only one :EXCLUSIVE rule may be fired on a single ILT frame during mapping. This allows the mapping rule writer to create a set of mutually-exclusive structural mappings for a single type of frame. All of the mappings in an :EXCLUSIVE rule must succeed, or the rule will not fire.

There is no limit to the number of :ANY rules that may fire on a single frame. If any of the mappings in an :ANY rule succeed, they are used, even if not all of the mappings are successful. For this reason it is possible to place all of the :ANY mappings for an ILT frame into a single rule. However, clarity dictates that only conceptually related :ANY mappings should be grouped into the same rule (e.g., mappings that generate PPADJUNCTS).
The `<predicate>` field should be an evaluable LISP expression. When evaluated, the `<predicate>` must return a non-NIL value for the rule to fire. The context present when the rule is fired can be accessed in one of two ways.

First, in the case of *path abbreviation* the `<predicate>` can contain a symbol of the form

`@<start>[.<<field>]]*`

The ‘@’ sign indicates that the rest of the symbol should be treated as a path abbreviation. `<start>` can be either FRAME or LEX, depending on whether the user intends to access the ILT frame or the lexicon frame that has been selected. Each `<field>` should be either a slot name or a facet name. For example, the path `@frame` returns the frame being mapped; the path `@frame.agent` returns the frame filling the AGENT slot of the frame being mapped; the path `@lex.syntactic-info.verb-class` will access the verb class facet of the syntactic information slot of the lexicon frame.

Secondly, with regard to *context variables*, the variables `!frame` and `!lex` are bound to the frame being mapped and its lexicon entry, respectively, and can be accessed freely inside the `<predicate>`. Although the user can use `@frame` and `!frame` interchangeably, the use of `@frame` symbols creates an extra function call where the use of `!frame` does not. As a result, `!frame` and `!lex` should be used for efficiency when no path is required.

The `<mslot>` field can take six basic forms:

1. **Slot Addition**: A slot is added directly to the f-structure.
2. **Feature Addition**: A feature is added directly to the f-structure.
3. **Slot to Slot Mapping**: A slot in the ILT frame is mapped to a slot in the f-structure.
4. **Slot to Feature Mapping**: A slot in the ILT frame is mapped to a feature in the f-structure.
5. **Expr to Slot Mapping**: An evaluated expression provides the value for a slot in the f-structure.
6. **Expr to Feature Mapping**: An evaluated expression provides the value for a feature in the f-structure.

### 3.1.2. Some Sample Mapping Rules

Rather than describing each type of `<mslot>` separately, we present and discuss several rules containing examples of each type of `<mslot>`. Because the distinction between :ANY and :EXCLUSIVE rules is important only when several rules are fired together, we defer discussion of this distinction until subsection 3.2.
Slot Addition:

(maprule e *ROLE :any t
 (slot cat NP))

The target language is e, (English), the rule is for ILT frames that are members of the *ROLE class, and the <predicate> field is t, so there are no restrictions on the rule. The rule contains a single <mslot>, (slot cat NP), which is an example of Slot Addition. If this rule is present, the slot (cat ((root NP))) will be added to each f-structure constructed from a *ROLE frame.

Feature Addition:

(maprule e *ROLE :any t
 (feature cat NP))

Here, the target language is again e, (English), the rule is for ILT frames that are members of the *ROLE class, and the <predicate> field is t, so there are no restrictions on the rule. The rule contains a single <mslot>, (feature cat NP), which is an example of Feature Addition. If this rule is present, the feature (cat NP) will be added to each f-structure constructed from a *ROLE frame.

Slot to Slot Mapping:

(maprule e *PROPOSITION :exclusive &frame.object
 ((slot agent) => (slot subj))
 ((slot object) => (slot obj)))

Here the <predicate> is &frame.object. The rule will not be fired unless there is an OBJECT present in the ILT frame. In this case, the AGENT and OBJECT slots will be filled with role frames, each of which will have been mapped already. Each role frame will contain a CURRENT-FS slot, which is accessed by the mapper and is the value returned by the left-hand slot in the mapping. This f-structure will be placed inside the slot designated by the left-hand slot in the mapping; hence ‘Slot to Slot Mapping.’ For example, if ((root boy)) was returned from the left-hand slot in the first mapping, then the slot (subj ((root boy))) will be placed into the f-structure.

(maprule e *ROLE :any t
 ((slot reference) => (slot ref)))

This mapping is similar to the previous one, except that the left-hand slot does not resolve to an f-structure. Here, the REFERENCE slot is filled with a non-frame symbolic value (e.g., DEFINITE). This value will be placed inside the slot indicated by the right-hand slot in the mapping. The mapper will place the value inside the form ((root <value>)), so a slot value of DEFINITE
in the ILT frame will add the slot (ref ((root definite))) to the f-structure as a result of this mapping.

Slot to Feature Mapping:

(maprule e *ROLE :any t
  (slot reference) => (feature ref)))

By changing the right-hand side of the mapping to a feature specification, we have an example of Slot to Feature Mapping. No extra structure is added by the mapper as in the case of Slot to Slot Mapping of a symbolic value. This mapping will add the feature (ref definite) to the f-structure, assuming that the value of the REFERENCE slot is DEFINITE.

Expr to Slot Mapping:

(maprule e *PROPOSITION :any t
  (expr (case @frame.clauseid.speechactid.speech-act
    (assertion 'declarative)
    (request-info 'interrogative)
    (request-action 'imperative))
  => (slot mood)))

This rule is an example of Expr to Slot Mapping. The left-hand expression is evaluated, and may contain path abbreviations and/or access the global variables mentioned above. The value returned by evaluating the expression is the value of the left-hand side. Assuming that the SPEECH-ACT slot of the frame attached to the proposition by the CLAUSEID and SPEECHACTID links is filled with ASSERTION, this value will be DECLARATIVE. Since the right-hand side is a slot and this value is symbolic, the slot (mood ((root declarative))) will be added to the f-structure. (If the value returned were a list expression rather than a symbol, it would have been placed directly into the slot without adding a ROOT).

Expr to Feature Mapping:

(maprule e *PROPOSITION :any t
  (expr (case @frame.clauseid.speechactid.speech-act
    (assertion 'declarative)
    (request-info 'interrogative)
    (request-action 'imperative))
  => (feature mood)))

---

8If the value returned by a left-hand slot is a symbol, it is wrapped inside ((root <symbol>)); otherwise it is assumed to be a list containing a root, and nothing special takes place.
In this rule, the left-hand expression is evaluated as in the previous example. Since the right-hand side is a feature specification, the feature (mood declarative) will be added to the f-structure.

The Optional <value> Field:

When a slot or feature specification appears on the right-hand side of an <mslot>, it may contain an explicit value; for example:

(feature mood imperative)

If an explicit value is present in the right-hand side, it is used instead of the value returned by the left-hand side of the mapping.

Post Processing:

Each <mslot> can contain an optional third field: <post-process>. This can be used to modify further the result of the original mapping. The <post-process> field should be filled with an evaluable LISP expression. This expression is evaluated with the following context variables set: !FRAME, !ILTSLOT, !FSSLOT and !SLOTFS. So, !FRAME is the ILT frame being mapped; !ILTSLOT is the name of the slot in the frame that is being mapped (if the left-hand side is an Expr, !ILTSLOT will be bound to EXPR); !FSSLOT is the name of the slot (or feature) being created in the f-structure; and !SLOTFS is bound to the contents of that slot as produced by processing the left side. If a <post-process> form is used, it should produce a valid structure for a full f-structure slot when evaluated, i.e., it should be a list containing a slot name and an f-structure for the slot contents. For example, if !FSSLOT is subj and !SLOTFS is ((root boy)), then the mapping ((slot agent) => (slot subj) (list !fsslot !slothfs)) will produce the slot (subj ((root boy))). To map a slot to another slot and add a feature, for example, a mapping like the following can be used:

((slot agent) => (slot subj)
  (list !fsslot (cons ' (case nom) !slothfs)))

Here the ILT AGENT is mapped to the subj slot, and the feature (case nom) is added to the slot’s f-structure.

3.2. The Mapper and its Algorithm

In this part we describe the top-level function in the mapper, and the algorithm that it implements.
The top-level function that should be called to map a single frame is called MAPGEN. It takes two arguments, a frame and a target specification. The frame argument should be a valid FRAMEKIT frame; otherwise an error is generated. The target language should have some mapping rules associated with it or an error will be signalled. The mapping rules for the indicated target language are fired on the frame, and the resulting f-structure is the value returned by MAPGEN.

The mapping rules associated with the frame are fired in the following way:

1. The CURRENT-CANDIDATE slot in the frame should be filled with a symbol denoting the generation lexicon entry that was selected for the frame (e.g., THROW).

2. Any rules for the target language that are indexed on the selected root are retrieved.

3. Any rules for any lexical class that the selected root inherits from are retrieved.

4. Any rules for the target language that are indexed on the type of the ILT frame (e.g., *ROLE) are retrieved.

5. The rules retrieved are fired in the order given above, one at a time, and the resulting f-structure is returned.

Two distinctions are made between :EXCLUSIVE and :ANY rules:

First, when MAPGEN is first called on the frame, it erases the value of the EXCLUSIVE-MAP? slot in the frame. When an :EXCLUSIVE rule fires successfully, this slot is filled, and no other :EXCLUSIVE rule will be fired. In contrast, an :ANY rule can always fire, no matter what rules have fired previously on the same frame.

Second, when an :EXCLUSIVE rule is fired, all of the <mslot> mappings in the rule must succeed, or the rule will fail and none of its mappings will be invoked. When an :ANY rule is fired, any of the <mslot> mappings can be successful, and will be processed, even if not all of the mappings in the rule are successful.

Mapping rules associated with a single index (lexical class or ILT type) will be fired in the order they were loaded. Hence, the order of definition in the mapping rule file is significant. For :ANY rules, there should be no real effect of ordering, but for :EXCLUSIVE rules (where the first successful rule "locks out" the others) ordering should be done with care.

---

9 This slot is erased entirely when the mapper exits, so it will have no effect on other processes accessing the ILT frame.

10 The :EXCLUSIVE mechanism is provided so that the grammar writer can define mutually exclusive structural mappings.
As described earlier in this document, mapping rules can access the context variables !FRAME and !LEX. The former is bound to the ILT frame being mapped; the latter is bound to the selected root.

4. SYNTACTIC GENERATION: GENKIT

KBMT-89 makes use of a tool called GENKIT (Tomita and Nyberg, 1988) to generate target language strings from the f-structures output by the mapping process; that process is discussed in the previous section. GENKIT is a grammar compiler that compiles source grammars in a unification-based formalism into a set of LISP functions that can be called directly during generation, without reference to the grammar itself.

The generation grammars for English and Japanese are pre-compiled into files containing LISP functions, which are in turn compiled and loaded into the run-time translation system.

The syntactic structures produced by the parser and the syntactic structures accepted as input by the generator are indistinguishable. The latter, as f-structures, capture the constituent structure of the utterance, the lexical entries (roots) of the constituents, and the tense and agreement features of the constituents where appropriate.

As noted in Tomita and Nyberg, 1988, the process of generating a string of words from a syntactic f-structure is basically the reverse of parsing. In the (simplified) grammar rule,\(^{11}\)

\[
(\textsc{dec} \leftrightarrow (\textsc{np} \textsc{vp} ))
\]

a constraint equation places the information from the \textsc{np} inside the subject of the \textsc{dec} during parsing; in generation, the f-structure for a \textsc{dec} will be broken up into its constituent parts, each of which will be generated by further recursive applications of grammar rules. In this case, the embedded f-structure that fills the subject slot of the declarative f-structure will be used as input to all of the rules that can possibly generate an \textsc{np}. The generator follows a top-down, depth-first strategy for applying rules during generation. If the current search path fails, the generator backs up to the next applicable rule. This process continues until a successful generation is found, or until all of the rules are exhausted.

The current implementation of the generator compiler involves creating a set of LISP functions that represents the grammar of the target language. Each function, GG-\textsc{x} (where \textsc{x} is any syntactic category), implements all rewrite rules from the grammar whose left hand symbol is \textsc{x}. When GG-\textsc{x} is called with the f-structure representation of a source-language string, and if that string can

\(^{11}\text{Cf. pp. 5-6 in (Tomita and Nyberg 1988), from which this account is excerpted.}\)
be generated by expansion of the non-terminal <X>, then GG-X returns the 
representative target-language string.

The process of constructing GG- functions consists of reading rewrite rules 
from the file containing the target-language grammar, and adjusting the appro-
priate GG- function after each rule is read. The first time a rewrite rule for 
<X> is read, the basic shell of the GG-X function is created, and the LISP code 
implementing that rule is added as a clause of OR:

```
(defun GG-X (x0)
  (OR  **LISP code for first rewrite rule** ))
```

(The argument passed to GG-X, x0, will be an f-structure.)

This new function is stored in a list with the other GG- functions. Each time 
a rewrite rule for <X> is read from the grammar file, function GG-X is retrieved 
from this list, and the code for the new rule is added as the last argument to the 
OR predicate.

To summarize: For every non-terminal category in the generation grammar, 
GENKIT creates a corresponding LISP function to generate structures of that 
category. For each rule that expands a particular category, a piece of code is 
generated to implement the constraints and assignments of the rule.

Let us consider an example in greater detail: In the case of verbs, GENKIT 
creates a function GG-V by building code for all the rules of type

\[<V> \rightarrow .... \]

GG-V will contain a block of code corresponding to each grammar rule.

Consider the following rules for verbs:

\[<V> \rightarrow (p u l l)\]
\[
  ((x0 root) =c pull) 
  (*OR* ((x0 form) =c finite) 
    ((x0 tense) =c present) 
    (*OR* ((x0 number) =c plural) 
      ((x0 person) =c (*OR* 3 2 l))) 
    ((x0 number) =c singular) 
    ((x0 person) =c (*OR* 2 l)))) 
  ((x0 form) =c inf))) 
  ((x0 valency) =c trans))
\]

\[<V> \rightarrow (p u l l s)\]
\[
  ((x0 root) =c pull) 
  ((x0 form) =c finite) 
  ((x0 tense) =c present) 
  ((x0 number) = singular) 
  ((x0 person) = 3) 
  ((x0 valency) =c trans)))
\]

\[<V> \rightarrow (p u l l e d)\]
\[
  ((x0 root) =c pull)
\]
(*OR* (((x0 form) =c pastpart)
   ((x0 passive) =c +)
   ((x0 valency) =c intrans))

   (*OR* (((x0 form) =c finite)
   ((x0 tense) =c past))
   (((x0 form) =c pastpart)
   ((x0 passive) =c -))
   ((x0 valency) =c trans))))

(<V> --> (pulling)
   (((x0 root) =c pull)
   ((x0 form) =c prespart)
   ((x0 valency) =c trans))

These rules are then compiled into the LISP function by GENKIT:

(DEFUN GG-V (X0)
  (IF (EQUAL *TRACE-RULES* 'ALL) (GENTRACE "~a called with ~a"
    'GG-V X0)
    (GENTRACE "~a called" 'GG-V))
  (LET (RESULT)
    (SETO *TRACE-RULE-INDENT* (+ *TRACE-RULE-INDENT* *TRACE-RULE-INDENT-FACTOR*))
    (SETO RESULT
      (OR
       <code for first rule>
       <code for second rule>
       <code for third rule>
       <code for fourth rule>))
    (SETO *TRACE-RULE-INDENT* (~ *TRACE-RULE-INDENT* *TRACE-RULE-INDENT-FACTOR*))
    (GENTRACE "~a returns ~a" 'GG-V RESULT)
    RESULT))

The code for a single rule is provided here:

(LET (((X (LIST (LIST (LIST 'X0 X0)))))
  (AND (P=CA (X0 ROOT) PULL)
    (SETO X
      (APPEND
       (LET ((X X))
       (AND (P=CA (X0 FORM) FINITE)
         (P=CA (X0 TENSE) PRESENT)
         (SETO X
           (APPEND
            (LET ((X X))
            (AND (P=CA (X0 NUMBER) PLURAL)
              (P=CA (X0 PERSON) (*OR* 3 2 1)))
            (LET ((X X))
            (AND (P=CA (X0 NUMBER) SINGULAR)
              (P=CA (X0 PERSON) (*OR* 2 1))))))))
    (LET ((X X)))))
(AND (P=CA (XO FORM) INF))))
(P=CA (XO VALENCE) TRANS)
(OR-DOLIST (FS X)
 (LET ((RESULT-STRING ""))
  (AND
    (SETQ RESULT-STRING
      (CONCATENATE 'STRING RESULT-STRING ""
        (SYMBOL-NAME 'P))
    (SETQ RESULT-STRING
      (CONCATENATE 'STRING RESULT-STRING ""
        (SYMBOL-NAME 'U))
    (SETQ RESULT-STRING
      (CONCATENATE 'STRING RESULT-STRING ""
        (SYMBOL-NAME 'L))
    (SETQ RESULT-STRING
      (CONCATENATE 'STRING RESULT-STRING ""
        (SYMBOL-NAME 'L)))
    (IF (EQUAL RESULT-STRING ""
      NIL RESULT-STRING)))))))

5. CONTROL

5.1. Top-Level Generation Algorithm

The top-level function in the generation module is REALIZE-NODE-TOP-DOWN. It accepts two arguments: an ILT frame to be realized in the target language and the expected syntactic category of the target phrase. Lexical selection proceeds, assigning the best possible choice of the appropriate category to the frame. Then any modifiers of the frame (including both embedded roles and attributes) are realized recursively. Subsequently, an f-structure is constructed by applying the generation mapping rules for the given ILT type. This f-structure is returned as the result of REALIZE-NODE-TOP-DOWN. This f-structure is then passed to the GENERATOR function, created during compilation of the target GENKIT grammar, which produces a target language string from the f-structure. The algorithm for REALIZE-NODE-TOP-DOWN is shown here:

Procedure REALIZE-NODE-TOP-DOWN (Frame, Cat):
  Bind (CandidateSet, OrderedCandidateSet, CurrentCandidate, FStructure);
  CandidateSet = generation lexicon-search (Frame);
  CandidateSet = FilterCat (CandidateSet, Cat);
  CandidateSet = FilterSubcat (Frame, CandidateSet);
  OrderedCandidateSet = SelectBestWord (Frame, CandidateSet);

12The expected syntactic category is determined by the following heuristics. If the frame is a role then it will be realized as a noun or an adjective, in that order. If the frame is a proposition, it will be realized as a verb. If the frame is a modifier, it will be realized as an adverb if it is embedded in a proposition, or as an adjective otherwise.
CandidateSet = SkimBestPenalties (OrderedCandidateSet);
CurrentCandidate = FindBestEntry (CandidateSet);
InvokeDemon (*realize-modifiers*, Frame);
FStructure = InvokeDemon (*build-f-structure*, Frame);
Return (FStructure).

The following sub-modules are called during the generation process:

1. **GL-Search.** This function finds any generation lexicon entries that might be used to realize the input frame.

2. **FilterCat.** This function removes any entries from the candidate set that do not match the Cat argument.

3. **FilterSubcat.** This function removes any entries from the candidate set whose subcategorization pattern does not match that of the ILT frame to be realized.

4. **SelectBestWord.** This function assigns a penalty to each candidate depending on how well it matches the meaning of the ILT frame to be realized, and orders the candidate set to correspond to the penalties assigned, that is, in increasing order.

5. **SkimBestPenalties.** This function finds all entries that share the lowest penalty.

6. **FindBestEntry.** This function decides which of the remaining entries to choose, in the event that Cat contains more than one possible category for realization. It makes use of heuristic rules such as *When a role may be realized as either a noun or an adjective, always try to use a noun first.*

7. **InvokeDemon.** This function searches for generation demons stored in the generation control knowledge base and fires them in order to process modifiers and build f-structures for frames once a lexical entry has been selected.

### 5.2. Control Knowledge

The process of realizing nodes in the ILT proceeds is described above for every type of ILT frame (CLAUSE, PROPOSITION, ROLE, etc.). In order to support specific types of processing for specific types of ILT frames, a separate knowledge base is used to describe precisely which steps should be taken in realization of each type. Hence, the top-level algorithm mentions general steps to be taken, and the knowledge base describes the specific tasks that embody each general step with respect to a particular ILT type. For example, to complete the general task of realizing any embedded modifiers for a ROLE frame, the generator must realize any embedded role frames and any attributitional modifiers. As a result, the generation knowledge frame for ROLE contains two REALIZE-MODIFIERS generation demons: REALIZE-ROLES and REALIZE-MODS; thus:
(defun define-generation-frames ()

(make-frame role
 (is-a (value (common ilt-type)))
 (map-expansion (:mod (common (get-values !frame 'modifier-slots))))
 (generation-demons
  (realize-modifiers (common (realize-roles !frame)
      (realize-mods !frame))
  (build-f-structure (common (build-fs !frame '*ROLE))))))

(make-frame modifier
 (is-a (value (common ilt-type)))
 (generation-demons
  (realize-modifiers (common)) ; no modifiers of modifiers!
  (build-f-structure (common (build-fs !frame '*MOD))))))

(make-frame proposition
 (is-a (value (common ilt-type)))
 (map-expansion (:mod (common (get-values !frame 'modifier-slots))))
 (generation-demons
  (realize-modifiers (common (realize-roles !frame)
      (realize-mods !frame))
  (build-f-structure (common (build-fs !frame '*'PROPOSITION))))))

(make-frame clause
 (is-a (value (common ilt-type)))
 (generation-demons
  (realize-modifiers (common (realize-clause-prop !frame)))
  (build-f-structure (common (build-fs !frame '*CLAUSE))))))

There are two demons that must be defined for each ILT type: REALIZE-
MODIFIERS and BUILD-F-STRUCTURE. These must be defined so that the
top-level algorithm can fire the appropriate demons when INVOKE-DEMON is
called. In the present implementation, the description of the demons is sim-
ple and straightforward. There are three LISP functions that serve as demons:
REALIZE-
ROLES, REALIZE-MODS, and BUILD-FS. REALIZE-ROLES and REALIZE-
MODS are general, in that they can process any kind of ILT frame; BUILD-FS
must be called with the particular type of mapping rule to fire when mapping
the frame.

The ILT frames to be processed are linked to the generation knowledge
frames via the ILT-TYPE slot, which should be present in each ILT frame and
have a value from the set (CLAUSE, PROPOSITION, ROLE, CLAUSE). When an ILT frame is processed, the particular demons to be fired are inherited
through the value of ILT-TYPE.
REFERENCES


