TR-I-0114

Multiple Inheritance in RETIF

Martin EMELE and Rémi ZAJAC

1989.09

Abstract

This report describes the new version of a prototype of rewriting system for typed feature structures. The formalism is based on a lattice theoretic approach due to Ait-Kaci. The semantics of Ait-Kaci treats disjunction and simple inheritance. This semantics has been extended to multiple inheritance, and the interpreter has been reimplemented to treat multiple inheritance.

In section 1 we present the syntax of the formalism; in section 2, the semantics of the formalism; in section3, the user's interface; in section 4, an HPSG grammar for English written in RETIF.

ATR Interpreting Telephony Research Laboratories
ATR 自動翻訳電話研究所

© (株) ATR 自動翻訳電話研究所 1989
© 1989 by ATR Interpreting Telephony Research Laboratories

Multiple Inheritance in RETIF*

Martin Emele and Rémi Zajac

ATR Interpreting Telephony Research Laboratories
Sanpeidani, Inuidani, Seika-cho, Soraku-gun, Kyoto 619-02, Japan

Abstract

We describe the new version of a prototype of a rewriting system for typed feature structures. The formalism is based on a lattice theoretic approach due to [Aït-Kaci 84]. The semantics of Aït-Kaci treats disjunction and simple inheritance. This semantics has been extended to multiple inheritance, and the interpreter has been reimplemented for treating multiple inheritance.

In section 1 we present the syntax of the formalism and the partial order of type symbols derived from a set of definitions. The new interpreter is described in the next section, in particular the rewriting process and the *eval* function which defines the operational semantics of the interpreter. In section 3, we describe the new user's interface. The last section is devoted to the presentation of a linguistic example: an HPSG grammar for English written in RETIF.

Key-Words multiple inheritance, types, sorts, rewriting system, unification-based formalisms, feature structures.

---

* Rewriting system for TYped Feature structures

TABLE

Introduction

1. Definitions

1.1 Disjunction and multiple inheritance
1.2 Embedding the partial ordering in a meet semi-lattice
A. The embedding
B. The meet operation
C. Typed unification

2. Interpretation

2.1 The EVAL function
2.2 The rewriting algorithm
2.3 A detailed example

3. The user's interface

3.1 General commands
reset
read
read-from-string
load
write
write-to-string
write-to-file
syntax
syntax-options
reset-syntax-options
printer-options
reset-printer-options
trace
3.2 Knowledge base related commands
get-kbs
get-kb
in-kb
create-kb

rename-kb
delete-kb
print-kb-names
print-kb
print-label-order
print-suppressed-labels
3.3 Type related commands
eval-type
get-types
get-type
get-internal-type
delete-type
rename-type
print-type-names
print-type
print-internal-type
partial-order
caller-dependency

4. An HPSG grammar for English

Conclusion

References

Appendix 1: a DCG-like grammar

Appendix 2: an HPSG grammar for English

Appendix 3: a sample session

Introduction

In [Emele and Zajac 89], we described a first implementation of a typed feature structure rewriting system which handled disjunction and single type inheritance. The main purpose of this new version is to enhance the expressive power of the formalism by introducing multiple inheritance.

Disjunctions can be used to represent ambiguity. The introduction of conjunctions (i.e. multiple inheritance) enhances modularity: bits of information common to several different types can be factored out and described separately. Furthermore, new types can be defined by combining several different types, much like objects can be defined in object-oriented programming paradigms.

Refering to other unification-based grammar formalisms (see [Shieber 86] for example), multiple inheritance could at first glance look like template mechanisms. There is however a major difference. The type rewriting system uses a type inference mechanism to derive new types dynamically during computation whereas templates are statically expanded at compile time (like macros in more traditional programming languages).

We describe the new version of a prototype of a rewriting system for typed feature structures. The formalism is based on a lattice theoretic approach due to [Aït-Kaci 84]. The semantics of Aït-Kaci treats disjunction and simple inheritance. This semantics has been extended to multiple inheritance, and the interpreter has been reimplemented for treating multiple inheritance.

In section 1 we present the syntax of the formalism and the partial order of type symbols derived from a set of definitions. The new interpreter is described in the next section: in particular, the rewriting process, and the *eval* function which defines the operational semantics of the interpreter. In section 3, we describe the new user's interface. The last section is devoted to the presentation of a linguistic example: an HPSG grammar for English written in RETIF.

1. Definitions

1.1 Disjunction and multiple inheritance

A type can be defined as a disjunction of types or a conjunction of types, or any combination of conjunctions and disjunctions. The symbol for the operator of disjunction is '|' and the symbol for the operator of conjunction is '&'. Syntactically, a type definition is written as:

<type-symbol> = <logical expression of typed feature structures> .

There are two constraints imposed on definitions:

1) Definitions must not lead to a cycle in the type system, otherwise, the type system cannot be ordered. Hence, some simple dependency tests are performed to check possible loops during rewriting.
2) Tags are local to a feature structure and cannot refer to an other feature structure in a logical expression.

A definition A = <exp> states that an object (a feature structure) of type A must verify the constraints expressed by the expression <exp>. Evaluation of a typed feature structure is constraint checking: during the evaluation process, any typed feature structure S_A of type A must verify the constraints stated in the definition of A: for a disjunction, S_A must verify at least one of the constraints stated as disjuncts; for a conjunction, S_A must verify all the constraints stated as conjuncts. If the constraint is a type symbol, S_A inherits from the constraints of the definition of this symbol, if it has one; if the constraint is a feature structure, S_A must be compatible (unifiable) with this feature structure.

The three cubes example

A knowledge base for the three cubes problem is defined as follows.

```
:KB 3cubes

COLOR = GREEN | NON-GREEN.
NON-GREEN = BLUE | OTHERS.

3CUBES = % 3CUBES is-a STACK which-has top, middle and bottom slots. %
         STACK[top:GREEN,
               middle:COLOR,
               bottom:BLUE].
```

```
ON = % ON is either a 3CUBES where top = above and middle = below
        or a 3CUBES with middle = above and bottom = below.  %
      3CUBES &
      (   [top:#x,
           middle:#y,
           above:#x,
           below:#y]
        | [middle:#z,
           bottom:#t,
           above:#z,
           below:#t]).

QUERY := %  A variable that contains a typed feature structure
             "Is there a green cube on top of a non green cube?"  %
           ON[above:GREEN, below:NON-GREEN].
```

1.2 Embedding the partial ordering in a meet semi-Lattice

A. The embedding

The normal form of the definitions

In order to be interpreted, the definitions of the KB are put in Normal Form as follows.

1. A disjunction of typed feature structures is replaced with a disjunction of new symbols, one for each disjunct: $A[k_1:R_1,\dots k_m:R_m] \mid B[l_1:S_1,\dots l_n:S_n]$ is replaced with A' | B' where $A' = A[k_1:R_1,\dots k_m:R_m]$ and $B' = B[l_1:S_1,\dots l_n:S_n]$.

2. A conjunction of typed feature structures is replaced with a conjunction of new symbols, one for each conjunct: $A[k_1:R_1,\dots k_m:R_m]$ & $B[l_1:S_1,\dots l_n:S_n]$ is replaced with A' & B' where $A' = A[k_1:R_1,\dots k_m:R_m]$ and $B' = B[l_1:S_1,\dots l_n:S_n]$.

3. A typed feature structure $A[l_1:S_1,\dots l_n:S_n]$ of type A is replaced with the conjunction A & S', where S' is a new symbol defined as $S' = [l_1:S_1,\dots l_n:S_n]$ (of type T).

4. The resulting logical expression of type symbols is put in a disjunctive normal form using the following simplification laws:

zero and one elements

A | *top* = *top* | A = *top*

A | *bottom* = *bottom* | A = A

A & *top* = *top* & A = A

A & *bottom* = *bottom* & A = *bottom*

idempotency

A | A = A

A & A = A

commutativity

A | B = B | A

A & B = B & A

associativity

(A | B) | C = A | (B | C) = A | B | C

(A & B) & C = A & (B & C) = A & B & C

distributivity

(A | B) & C = A & C | B & C

absorption

(A & B & C) | (A & B) = (A & B)

Using these rules, the set of definitions of the knowledge base 3cubes is replaced with:

```
COLOR = GREEN | NON-GREEN.
NON-GREEN = BLUE | OTHERS.

3CUBES = STACK & S3.
S3 = [top:GREEN,
      middle:COLOR,
      bottom:BLUE].

ON = ON1 | ON2.
ON1 = 3CUBES & S1
ON2 = 3CUBES & S2.
S1 = [top:#x,
      middle:#y,
      above:#x,
      below:#y].
```

```
S2 = [middle:#z,
      bottom:#t,
      above:#z,
      below:#t].
```

The partial order on type symbols extracted from the KB definitions is shown in Figure 1.

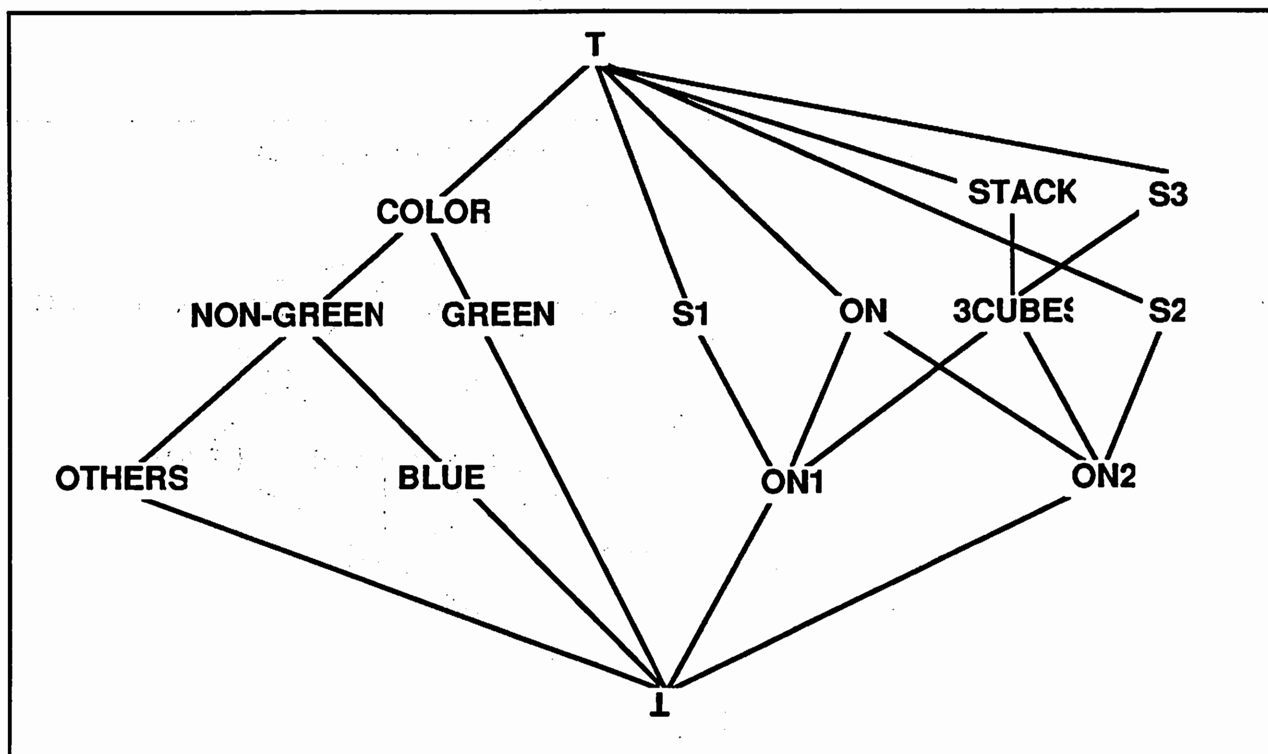

Figure 1: The partial ordering on type symbols extracted from the KB definitions

B . The Meet Operation

The requirement that the partial order extracted from the set of type definitions has to be a lattice is a much too strong restriction on the kind of definitions that could be written in the formalism. But in the previous example there are two maximal elements for the meet of ON and 3CUBES: ON1 and ON2. The solution is to embed the partial ordering P extracted form the KB definitions in the (restricted) power set $2^{(P)}$ (the set of all non-empty finite subsets of pairwise not comparable elements of P).

The meet operation can be defined using the inclusion ordering in $2^{(P)}$: the meet of two elements X and Y of $2^{(P)}$ is then defined as the maximal restriction of the intersection of the sets of subtype symbols for each pair of symbols x and y of X and Y.

If X and Y are elements of $2^{(P)}$, the meet of X and Y is defined as

$$X \wedge Y = \lceil \cup_{x \in X, y \in Y} (I_x \cap I_y) \rceil. \qquad \text{(Note that this construction does not preserve LUBs)}$$

The maximal restriction $\lceil E \rceil$ takes the set of maximal elements of P: when 2 elements are comparable, the smaller is removed: $\lceil E \rceil = \{x \in E \mid y \leq x \Rightarrow x = y\}$. The set of subtypes symbols of x is called the principal ideal of E generated by x : $I_x = \{y \in E \mid y \leq x\}$.

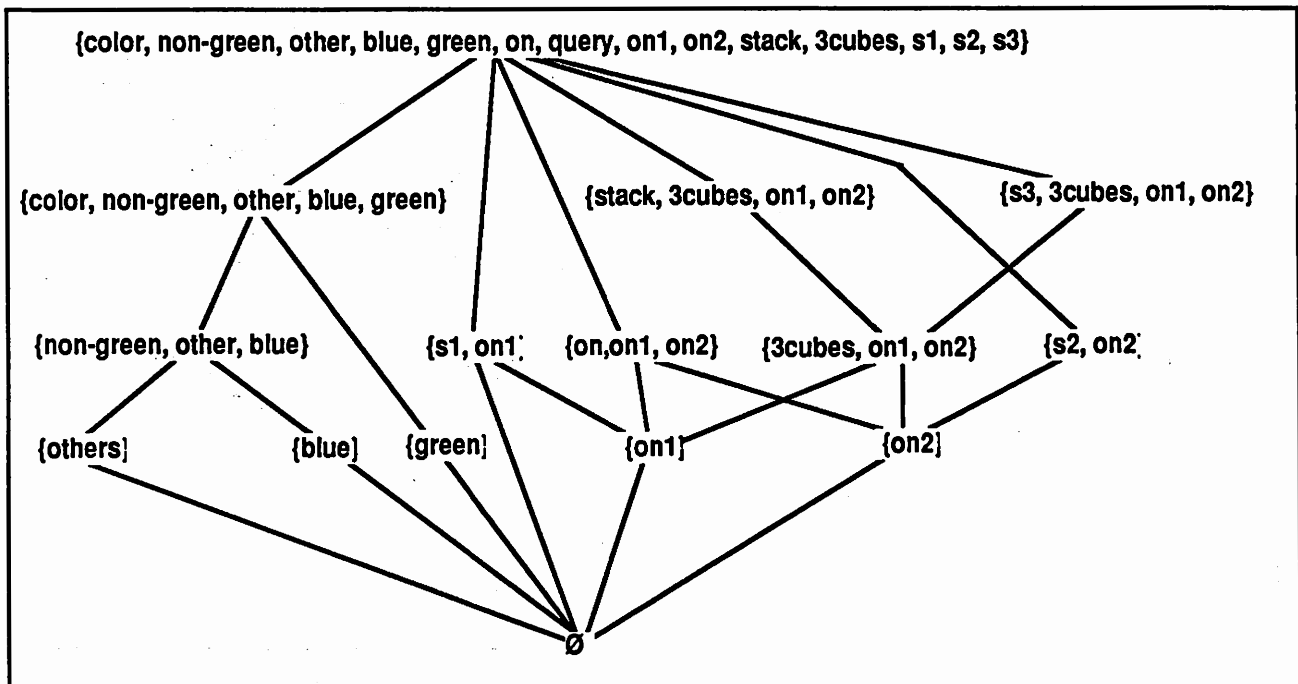

Figure 3: The embedding in the set of all principal ideals preserve the GLBs.

Example: ON ∧ 3CUBES

principal ideals → {ON, ON1, ON2} ∩ {3CUBES, ON1, ON2}

intersection of principal ideals → {ON1, ON2}

maximal restriction → {ON1, ON2}

Implementation note

The efficiency of the meet operation is crucial for the overall performance of the interpreter. The evaluation of the cost of the unification algorithm (which is almost linear with the number of nodes) does not take into account the cost of the meet operation.

If p=|P| the number of symbol in the type system P, then the size of X and Y, elements of $2^{(P)}$ is bound by p. The cost of computing an ideal is the cost of a traversal of the graph which represents the partial order, and is linear with the number of nodes p, and there are at most 2p such traversals (one for each element of X and Y): $2p^2$. The size of an ideal, element of $2^{(P)}$ is bounded by p. The cost of an intersection is then p^2 and there are also at most p^2 intersections: p^4. The cost of each union is also p^2, and there are at most p^2 unions: p^4. The maximal restriction compares each element of the set with each other: p^2 comparison. Each comparison requires a traversal of the graph and costs p. The cost of the maximal restriction is then in p^3. Finally, we get $O(2p^3 + p^4 + p^4 + p^3) = O(p^4)$ for the cost of the meet operation.

During the construction of the partial order out of the KB definitions, all ideals are pre-computed. However, contrary to our previous implementation, the meet between two symbol is not precomputed because most often the meet is now performed between conjunctions of symbols, and we don't precompute all possible conjunctions (nor all possible disjunctions, otherwise the size of the graph representing the order would be in $O(2^P)$). As a consequence, we need to compute each time intersections and maximal restrictions between conjunctions of symbols. But the worst case is of course still the same: in $O(p^4)$.

C. Typed unification

The unification on ordinary feature structures is defined as a meet operation on the set of feature structures partially ordered by the subsumption ordering (see for example [Shieber 86]). This is extended straightforwardly to typed feature structures using the meet operation on the lattice of types to compute the new type associated with the result of the unification of two feature structures.

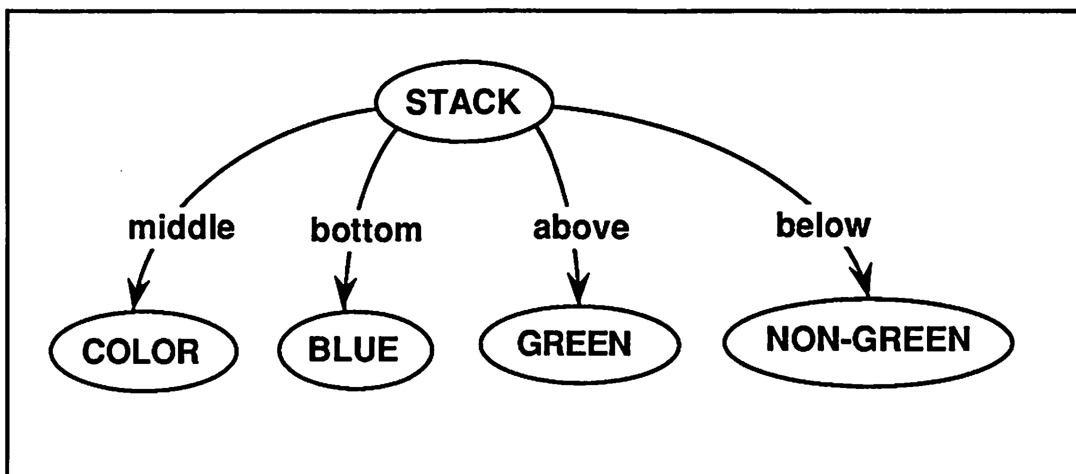

Figure 4: STACK[middle: COLOR, bottom: BLUE, above: GREEN, below: NON-GREEN]

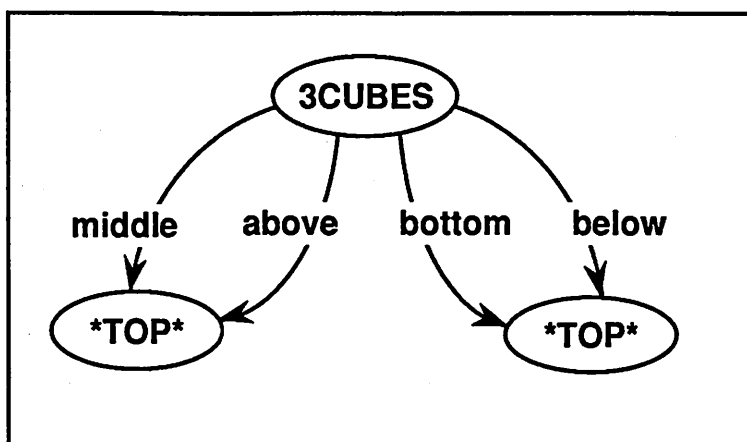

Figure 5: 3CUBES[middle:#z, bottom:#t, above:#z, below:#t]

The unification of the two feature structures above (Fig. 4 and 5) can be done essentially like ordinary unification. The only extension we need is to compute the meet of two type symbols which are associated with the feature structures. When two paths are merged, the new type associated to the subterm under these paths is the meet of the types of the two subterms which are unified (Fig. 6).

merged paths	*associated types*	*new type symbol*
ε	3CUBES & STACK	3CUBES
middle above	GREEN & COLOR	GREEN
bottom below	BLUE & NON-GREEN	BLUE

Figure 6: the meet for merged paths.

The merging of common paths and the computation of the meet (that is typed unification) yields the typed feature structure of Figure 7.

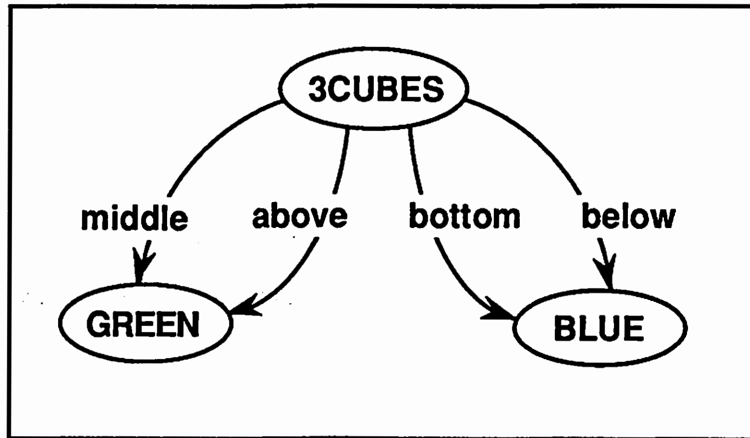

Figure 7: 3CUBES[middle: #z = GREEN, bottom: #t = BLUE, above: #z, below: #t]

2. Interpretation

2.1 The EVAL function

A knowledge base KB consists of a set of definitions $t_i = KB(t_i)$, where t_i is a type symbol and $KB(t_i)$ is a term (a feature structure) $[l_1: t_1, \ldots, l_n: t_n]$, a disjunction of type symbols $t_1 \mid \ldots t_n$, or a conjunction of type symbols $t_1 \;\&\; \ldots t_n$.

We can define a function EVAL : *term* $\rightarrow$ *term*

(1) $\text{EVAL}(t_1 \mid \ldots t_n) = \vee_{i=1 \ldots n} \text{EVAL}(t_i)$

(2) $\text{EVAL}(t_1 \;\&\; \ldots t_n) = \wedge_{i=1 \ldots n} \text{EVAL}(t_i)$

(3) $\text{EVAL}([l_1: t_1, \ldots, l_n: t_n]) = [l_1: \text{EVAL}(t_1), \ldots, l_n: \text{EVAL}(t_n)]$

Equations (1), (2) and (3) define an *operational semantics* which reflects the type-as-set semantics of *terms* in the sense that they compute unions and intersections of sets.

2.2 The rewriting algorithm

Def: A **Symbol Rewriting System** (SRS) on Σ (signature of type symbols) is a system S of n equations $s_i = E_i$, $i = 1, \ldots, n$ where $s_i \in \Sigma$ and E_i is a term: $S = \{ s_i = E_i \}$.

The set of symbols of Σ which have a definition is $E = \{ s_1, \ldots, s_n \}$, the set of *S-expandable symbols*. The set of symbols of Σ which do not have a definition is $N = \Sigma - E$, the set of *non-S-expandable symbols*. In the three cubes example, E = { COLOR, NON-GREEN, 3CUBES, ON, ON1, ON2, S1, S2, S3 } and N = { GREEN, BLUE, PURPLE, OTHERS, STACK }.

Def: A *one step rewriting relation* $t_1 \rightarrow t_2$ is defined *iff* there exists a symbol $s_i \in E$ at some address (path) u in t_1 such that E_i is «substituted» at address u: E_i is unified with the subterm at address u, and the result of unification is inserted at that address. The new term is called t_2:

$t_2 = t_1[E_i/u] = t_1[u: T] \wedge u.E_i$

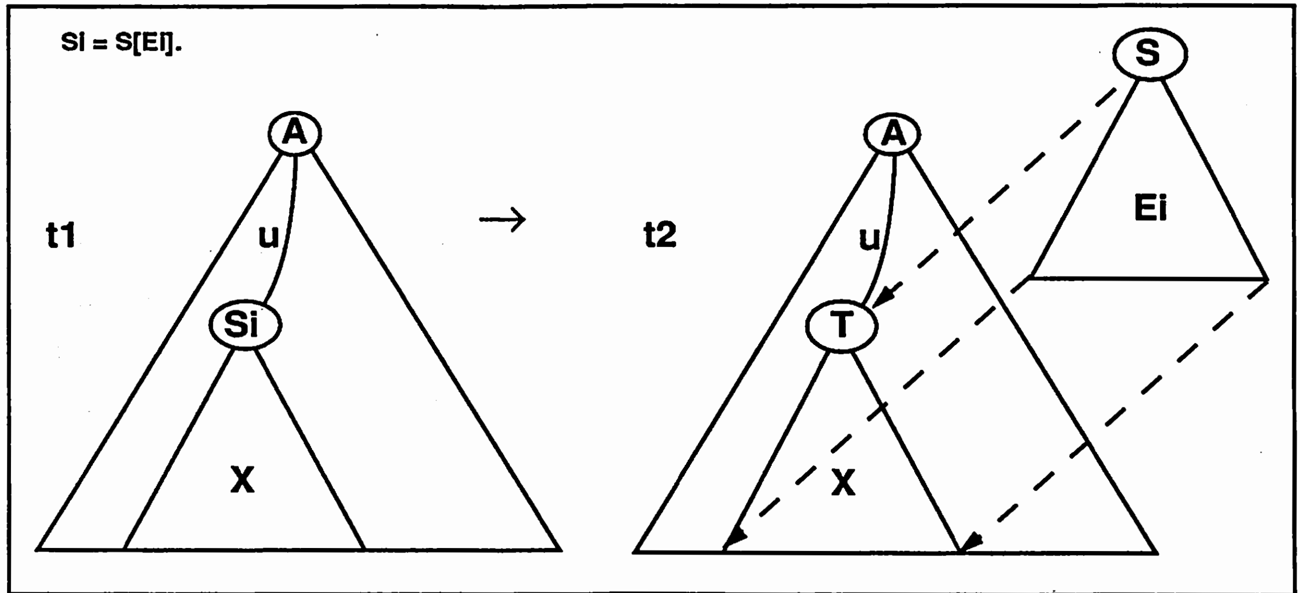

Figure 8: one step rewriting relation.

2.3 A detailed example

Let's take the three cubes KB:

```
COLOR = GREEN | NON-GREEN.   (= E1)

NON-GREEN = BLUE | PURPLE | OTHERS.   (= E2)

3CUBES = STACK  & S3.   (= E3)

S3 = [top:GREEN, middle:COLOR, bottom:BLUE].   (= E4)

ON = ON1 | ON2.   (= E5)

ON1 = 3CUBES & S1.   (= E6)

ON2 = 3CUBES & S2.   (= E7)

S1 = [top:#x, middle:#y, above:#x, below:#y].   (= E8)

S2 = [middle:#z, bottom:#t, above:#z, below:#t].   (= E9)
```

We shall show the behavior of the interpreter on the evaluation of the term ON[above:GREEN, below:NON-GREEN] : is there a green cube on top of a non-green cube? This term is expandable with symbol ON at address ε (at each step, expandable symbols are written in bold face). The set of S-expandable symbols is the set of all left hand side symbols.

t_1 = ON[above:GREEN, below:NON-GREEN]

```
t2 = t1[E4 / ε ]
     = (ON1 | ON2) &
       [above:GREEN, below:NON-GREEN]
```

The rewriting of t_1 leads to the term t_2 which contains a disjunction of type symbols at the root. This disjunction has to be further expanded because ON1 and ON2 belong to the set of S-expandable symbols. The next step expands the disjunction ON1 | ON2 and creates two new terms t_2' and t_2'' with type symbols ON1 and ON2 respectively. These terms are further expanded in two different branches of computation.

Branch 1 of the disjunction: expand ON1.

```
t2' = ON1[above: GREEN, below: NON-GREEN ]

t3 = t2'[E6 / ε ] = 3CUBES & S1 &
                    [above: GREEN , below: NON-GREEN ]
```

Now there are 2 expandable symbols at address ε: 3CUBES and S1. These symbols can be rewritten in any order: 3CUBES for example.

```
t4 = t3[E4 / ε ] = STACK & S3 & S1 &
                  [above: GREEN , below: NON-GREEN ]
```

STACK doesn't have definition and is not expandable, and S3 and S1 are expandable. Their definition is a feature structure which will be unified with the term at address ε.

```
t5 = t4[E5 / ε ] = STACK & S1 &
                  [top: GREEN,
                   middle: COLOR,
                   bottom: BLUE,
                   above: GREEN,
                   below: NON-GREEN]

t6 = t5[E8 / ε ] = STACK
                  [top: #x=GREEN,
                    middle: #y=NON-GREEN,
                   bottom: BLUE,
                   above: #x,
                   below: #y]
```

First solution:

```
t7 = t6[E2/middle] = STACK
                      [top:  #x=GREEN,
                         middle: #y= BLUE | PURPLE | OTHERS,
                         bottom: BLUE,
                         above: #x,
                         below: #y]
```

Branch 2 of the disjunction: expand ON2.

```
t2" = ON2[above: GREEN, below: NON-GREEN ]

t7 = t2"[E7 / ε ] = 3CUBES & S2 &
                     [above:  GREEN,  below:  NON-GREEN  ]

t8 = t7[E7 / ε ] = STACK & S3 & S2 &
                    [above:  GREEN,  below:  NON-GREEN  ]

t9 = t8[E4 / ε ] = STACK & S2 &
                    [top:  GREEN,
                     middle: COLOR,
                     bottom: BLUE,
                     above: GREEN,
                     below: NON-GREEN]
```

Second solution:

```
t10 = t9[E9 / ε ] = STACK
                      [top:  GREEN,
                       middle: #z = GREEN,
                       bottom: #t = BLUE,
                       above: #z,
                       below: #t ]
```

3. The user's interface

We give here a list of commands available for the users. An EMACS mode has been defined for LISP machines. It will automatically be invoked using the pathname extension 'tfs'.

(TFS-help)

Gives a general help. More documentation is available for particular functions using TFS-help with the name of the topic.

(TFS-reset)

Resets the whole TFS universe.

(TFS-read &optional stream)

Reads statements in the stream until EOF.
Default stream: *standard-input*.

(TFS-read-from-string string)

Reads statements from a string.

(TFS-load file-name)

Reads statements from file-name.

(TFS-write object &key

```
        (stream            *standard-output*)
        (mode              *tfs-printer-mode*)
        (minor-mode        *tfs-printer-minor-mode*)
        (macro             *tfs-printer-macro*)
        (comment           *tfs-printer-print-comment*)
        (line-length       *tfs-printer-line-length*)
        (init-indent       *tfs-printer-init-indent*)
        (rel-indent        *tfs-printer-relative-indentation*)
        (newline-p         *tfs-printer-newline-p*)
        (init-tag-counter  *tfs-printer-init-tag-counter*)
        (all-tag-p         *tfs-printer-all-tag-p*)
        (level             *tfs-printer-level*)
        (length            *tfs-printer-length*)
        (label-order       *tfs-printer-label-order*)
        (suppressed-labels *tfs-printer-suppressed-labels*)))
```

```
(TFS-write-to-string object &key
        (mode              *tfs-printer-mode*)
        (minor-mode        *tfs-printer-minor-mode*)
        (macro             *tfs-printer-macro*)
        (comment           *tfs-printer-print-comment*)
        (line-length       *tfs-printer-line-length*)
        (init-indent       *tfs-printer-init-indent*)
        (rel-indent        *tfs-printer-relative-indentation*)
        (newline-p         *tfs-printer-newline-p*)
        (init-tag-counter  *tfs-printer-init-tag-counter*)
        (all-tag-p         *tfs-printer-all-tag-p*)
        (level             *tfs-printer-level*)
        (length            *tfs-printer-length*)
        (label-order       *tfs-printer-label-order*)
        (suppressed-labels *tfs-printer-suppressed-labels*))
```

```
(TFS-write-to-file object pathname &key
        (mode              *tfs-printer-mode*)
        (minor-mode        *tfs-printer-minor-mode*)
        (macro             *tfs-printer-macro*)
        (comment           *tfs-printer-print-comment*)
        (line-length       *tfs-printer-line-length*)
        (init-indent       *tfs-printer-init-indent*)
        (rel-indent        *tfs-printer-relative-indentation*)
        (newline-p         *tfs-printer-newline-p*)
        (init-tag-counter  *tfs-printer-init-tag-counter*)
        (all-tag-p         *tfs-printer-all-tag-p*)
        (level             *tfs-printer-level*)
        (length            *tfs-printer-length*)
        (label-order       *tfs-printer-label-order*)
        (suppressed-labels *tfs-printer-suppressed-labels*))
```

(tfs-printer-options)

Menu for choosing the printer options.

(tfs-reset-printer-options)

Resets the printer options to the initial settings.

(TFS-syntax &optional stream)

Specifications of the TFS syntax
****

COMMENTS: There are 2 kinds of comments.

1. IN-LINE comments begin with the ';' character and end with the end of line. These comments can appear anywhere where a white space is expected in the syntax. These comments will be skipped during reading.

2. SYNTACTIC comments are enclosed with the '%' character. These comments can appear as specified in the BNF syntax specification. They will be attached to the structure produced by the reader.

IDENTIFIERS: Identifiers are case-sensitive.

BNF: The following extensions are used:

[x] denotes the optional element x.
[x]* denotes the free iteration of element x.
[x]+ denotes the iteration of element x.
If there might be a confusion with the symbols of the BNF, symbols of the described syntax are enclosed in quotes: | denotes the alternation in the BNF formalism and '|' denotes the disjunction symbol of the TFS syntax.

<tfs-entity> ::= <knowledge-base> | <with-kb>
| <label-order> | <suppressed-labels>
| <tfs-declaration>
<knowledge-base> ::= :KB <idf> [<comments>]
<with-kb> ::= :WITH-KB <idf> [<comments>]
<label-order> ::= :LABEL-ORDER [<tfs-selector> [, <tfs-selector>]*] .
<suppressed-labels> ::= SUPPRESSED-LABELS
[<tfs-selector> [, <tfs-selector>]*] .
<tfs-declaration> ::= <tfs-type-defintion> | <tfs-variable-declaration>
<tfs-type-defintion> ::= <idf> = <tfs-logical-expression> .
<tfs-variable-declaration> ::= <idf> := <tfs-logical-expression> .
<tfs-logical-expression> ::= [<comments>] <tfs-logical-term> ['|' <tfs-logical-expression>]

<tfs-logical-term> ::= [<comments>] <tfs-logical-complement> [& <tfs-logical-term>]

<tfs-logical-complement> ::= [<comments>] [[<tfs-logical-factor>] \ ] <tfs-logical-factor>
<tfs-logical-factor> ::= [<comments>] (<tfs-logical-expression>)
| [<comments>] <tagged-structure>
<tagged-structure> ::= <tag> [= <typed-structure>]
| <typed-structure>
<typed-structure> ::= <idf> [<structure>]
| <structure>
| <list-structure>
<structure> ::= '[' <selector-structure> [, <selector-structure>]* ']'
<selector-structure> ::= <selector> : <tfs-logical-expression>
<selector> ::= <idf>
<list-structure> ::= < [<tagged-structure>]* >
| < [<tagged-structure>]+ . <tagged-structure>]
<comments> ::= [% <char>* %]*
<key-word> ::= :<idf>
<tag> ::= #<idf>
<idf> ::= <idf-char> [<idf>]
| " <char>* " (Note: the " is doubled in strings)
<idf-char> ::= <letter> | <digit> | _ | - | + | * (Note: kanjis are read as <idf-char>)

(tfs-syntax-options)

Menu for choosing the TFS syntax.
(Note: this feature exists only for compatibility with other systems.)

(tfs-reset-syntax-options)

Resets the TFS syntax to the initial settings.
(Note: this feature exists only for compatibility with other systems.)

(tfs-trace)

Menu for choosing the TFS trace.

(tfs-reset-trace)

Resets the TFS trace.

(eval-type type-name &optional interactive? &rest kb-names)

Evaluate the type <type-name> in the list of KBs <kb-names>.
If <kb-names> is not provided, the type is evaluated in the current KB
otherwise the type is evaluated in the first KB, then
the result is evaluated in the second one, and so on.

The final result is stored in global variable *TFS-result* as a LIST
of internal terms.
If :interactive? is NIL, the evaluation proceeds without interaction.
If :interactive? is T, the interpreter asks what to do for each solution.
If :interactive? is an integer N, the interpreter asks what to do after
the Nth solution.
Default for :interactive?: *ask-for-other-solutions*.
Possible answers for interaction:

N | <carriage-return> == Next solution and continue
P == Proceed without interaction
E | <any other char> == Exit

(get-kbs)

Returns all KBs.

(get-kb &optional kb-name)

Returns the KB <kb-name>.

(create-kb kb-name)

Creates a new (empty) KB <kb-name>.

(in-kb kb-name)

Set the current KB as <kb-name> (contained in the variable *kb*).

(delete-kb kb-name)

Deletes the KB <kb-name>.

(rename-kb old-name new-name)

Renames a KB <old-name> to <new-name>.

(print-kb-names &optional stream)

Print all KB names.
Default stream: *standard-output*.

(print-kb &optional kb-name stream)

Prints all definitions of KB <kb-name>.

Default KB: *kb* (the current KB).

Default stream: *standard-output*.

(print-label-order &optional kb-name stream)

Prints the partial order of labels used for printing.

Default KB: *kb* (the current KB).

Default stream: *standard-output*.

(print-suppressed-labels &optional kb-name stream)

Prints the list of labels which are suppressed during printing.

Default KB: *kb* (the current KB).

Default stream: *standard-output*.

(get-internal-type type-name &optional kb-name)

Returns the internal (compiled) form of the definition of <type-name> in KB <kb-name>.

Default KB: *kb* (the current KB).

(get-type type-name &optional kb-name)

Returns the right hand side of the definition of <type-name> in KB <kb-name>.

Default KB: *kb* (the current KB).

(get-types &optional kb-name)

Returns all right hand side types in KB <kb-name>.

Default KB: *kb* (the current KB).

(delete-type type-name &optional kb-name)

Remove the definition of type <type-name> in KB <kb-name>.

Default KB: *kb* (the current KB).

(print-type-names &optional kb-name stream)

Prints the list of left hand side type names defined in KB <kb-name>.

Default KB: *kb* (the current KB).

Default stream: *standard-output*.

(print-type type-name &optional kb-name stream)

Prints the definition of type <type-name>.
Default KB: *kb* (the current KB).
Default stream: *standard-output*.

(print-internal-type type-name &optional kb-name stream)

Prints the definition of type <type-name>.
Default KB: *kb* (the current KB).
Default stream: *standard-output*.

(partial-order &key kb-name type-names stream orientation)

Prints the partial order of KB <kb-name> starting with the list of type names <type-names>.
Default KB: *kb* (the current KB).
Default stream: *standard-output*.
Default type-names: TOP
Default orientation: :horizontal

(partial-order &key kb-name type-names stream orientation)

Prints the partial order of KB <kb-name> starting with the list of type names <type-names>.
Default KB: *kb* (the current KB).
Default stream: *standard-output*.
Default type-names: NIL (means all lhs symbols).
Default orientation: :horizontal

4. An HPSG Grammar for English

The following grammar used in this system for generating English is based on a version of Head-driven Phrase Structure Grammar (HPSG) as described in [Pollard & Sag 87]. This formalism is perfectly suited to the generation task, because it eliminates category-specific phrase-structure rules in favor of very partially specified combinational schemata that constrain the relation of constituency among the linguistic objects. Further specifications of a token linguistic object arise from the lexical entries which contain phonological, syntactic, and semantic information. Together with language-specific and language-universal principles of well-formedness, the final result is obtained by unifying the information from all these various sources. Yet the grammar itself is purely declarative in the sense that it characterizes what constraints are brought to bear during generation independent of the order the constraints are applied. In addition, the grammar itself is unbiased as to the kind of processing task at hand: it is reversible in the sense that it is neutral with respect to analysis versus generation. Viewing the grammar as nothing else but a set of constraints enhances the possibility of the grammar functioning as a filter for the planning phase of the generation. More specifically, for the translation task at hand, the grammar will filter out ungrammatical choices from the possibilities of expressing the content of an utterance by various surface forms. Only those which are compatible with the grammar will finally survive and will be generated. The actual production of the surface string is done by unifying the input description with the combinational rules of the grammar.

In the following part we would like to present a simplified version of the grammar and the morphological rules. The kind of rules which are used can easily be generalized so that they separate the immediate dominance relationships (ID) and the linear precedence constraints (LP) as described in [Pollard & Sag 87]. The only difference is in the way of combining the phonological values of the daughters in order to get the phonological value of the whole phrase. The simple functional application of *append* will be replaced by a function *order-constitutents* which obeys the independent stated language-specific principles of constituent ordering.

For expository purposes the treatment of lexical entries, and especially the morphological description of the individual part of speeches has been simplified. It is beyond the focus of this report to describe in detail the lexical hierarchy of types as suggested in the HPSG-framework. By using typed feature structures as the basic representational device, such an incorporation of the lexical hierarchy could be reformulated straightforwardly in the type system by assigning an individual type definition for each element of the lexical type hierarchy.

The minimal approach which has been adopted in order to factor out information about words which is not idiosyncratic to a specific lexical item led to a schema like (1). It simply states that a MINOR lexical category will be rewritten as a LEXICAL-SIGN, neglecting the fact that every noun, adjective, or verb belongs to an inflectional paradigm which actually produces the phonological surface

description according to the value of certain kinds of features (like number, gender, tense, etc.). Instead, it just creates an initial list of the phonological string which is identical to the lexem itself as marked by the identical tag in the feature description of minor).

(1)

```
MINOR = % Lexical head: relates lexem with the phonological string %
        LEXICAL-SIGN
        [phon: <#string>,
         syn:  [head: [lexem: #string]]] .
```

There is a fundamental division of all linguistic objects or signs into either a LEXICAL-SIGN (respectively, MINOR lexical category) or a PHRASAL-SIGN (respectively, MAJOR category). Thus, every sign or linguistic object which is not an instance of a MINOR lexical category belongs to one of the MAJOR categories which can be described by a specific grammar rule.

(2)

```
SIGN = PHRASAL-SIGN | LEXICAL-SIGN .
```

So far we need only two grammar rules which are stated as a disjunction of type symbols in (2), namely CH_CO_FP and HC*_CO_FP, together with a conjunction of general feature principles like the Head Feature Principle (HFP) and the Subcategorization Principle which they inherit from.

(3)

```
MAJOR = % Grammar Rules %
         PHRASAL-SIGN & HEAD_FP & SUBCAT_FP &
          (% Subject Predicate Rule %
          CH_CO_FP | % Head Complements Rule %
                     HC*_CO_FP) .
```

MAJOR categories will be rewritten to PHRASAL-SIGN. PHRASAL-SIGNS differ from LEXICAL-SIGNS in having a daughter attribute that gives information about the (lexical or phrasal) signs which are the immediate constituents of the sign in question. This attribute encodes the kind of information about constituency that is contained in conventional constituent-structure tree diagrams. In addition, the various daughters are distinguished according to what kinds of information they contribute to the sign as a whole. Thus daughters are classified as *heads* and *complements* respectively as in standard X-bar theory.

The 'Complement Head Constituent Order Feature Principle', given in (4) below, corresponds to the rules usually expressed in the form S → NP VP, NP → Det Noun. This rule simply states that one of the possibilities for a PHRASAL-SIGN is to be a saturated (i.e. [subcat: <>]) sign which has as constituents a single complement daughter (i.e.[comp-dtrs: <>]) and a head daughter, which in turn is constrained to be of type PHRASAL-SIGN rather than LEXICAL-SIGN.

(4)

```
CH_CO_FP = % Complement Head Constituent Order Feature Principle %
             [phon:         #phon,
              dtrs:         [head-dtr:  PHRASAL-SIGN
                                        [phon: #head-phon],
                             comp-dtrs: <[phon: #comp-phon]>],
              patch_phon: APPEND
                            [front: #comp-phon,
                             back:  #head-phon,
                             whole: #phon]] .
```

The 'Head Complements Constituent Order Feature Principle', as given in (5) below, describes how an unsaturated phrasal sign can be constructed by combining any number of complements (including no complement at all) with the head-daugther. In order to do so, one additional type MAP is defined as a disjunction that describes the recursive way of mapping over the individual complements and combining the corresponding phonological values. The result of this recursion is the concatenation of the phonological values of all the complements. The phonological value of the whole phrase can than be computed by concatenating the head phonology value with the complement phonology value.

(5)

```
HC*_CO_FP = % Head Complements Constituent Order Feature Principle %
             [phon:         #phon,
              dtrs:         MAP
                            [head-dtr:  LEXICAL-SIGN
                                        [phon: #head-phon],
                             comp_phon: #comp-phon],
              patch_phon: APPEND
                            [front: #head-phon,
                             back:  #comp-phon,
                             whole: #phon]] .
```

The recursion bottoms out after there are no more complements left (i.e. [comp-dtrs <>]). Thus the complement phonology value is empty as well.

The second disjunct of MAP combines one complement at a time by consuming the first element of the complement-daughters' list and applying the same rule recursively to the rest of the complement-daughters' list. The new phonological value is than derived by concatenating the phonological value of the first complement with the phonological value of all the remaining complements whose value itself is obtained as the result of the recursive application of this rule.

(6)

```
MAP = % Empty comp-dtrs list %
       TREE
       [comp-dtrs: <>,
        comp_phon: <>] |
        % Nonempty comp-dtrs list %
       TREE
       [comp-dtrs:       <[phon: #first-comp-phon] . #rest-comp>,
        patch_comp_phon: APPEND
                         [front: #first-comp-phon,
                          back:  #rest-comp-phon,
                          whole: #comp-phon],
        comp_phon:       #comp-phon,
        map_patch:       MAP
                         [comp-dtrs: #rest-comp,
                          comp_phon: #rest-comp-phon]] .
```

The 'Head Feature Principle' ensures that the head features of the head-daughter are always shared with their phrasal projections.

(7)

```
HEAD_FP = % Head Feature Principle %
          [syn:  [head: #head],
           dtrs: [head-dtr: [syn: [head: #head]]]] .
```

The 'Subcat Feature Principle' states that in any phrasal sign the subcat list of the head-daughter is the concatenation of the list of complement daughters and the subcat list of the mother.

(8)

```
SUBCAT_FP = % Subcat Feature Principle %
            [syn:              [subcat: #rest-subcat],
             dtrs:             [head-dtr:  [syn: [subcat: #subcat]],
                                comp-dtrs: #comps],
             patch_subcatfp: APPEND
                               [front: #comps,
                                back:  #rest-subcat,
                                whole: #subcat]] .
```

Conclusion

We have defined a type as set semantics for multiple inheritance and gave one possible implementation of it by combining a partial ordering of type symbols and the associated meet operation. Furthermore, we have shown that this formalism can be used in a very natural way for describing complex linguistics objects as a combination of simpler objects.

References

Hassan AIT-KACI, 1984, *A Lattice Theoretic Approach to Computation Based on a Calculus of Partially Ordered Type Structures*, Ph.D. Thesis, University of Pennsylvania.

Hassan AIT-KACI, 1986, An Algebraic Semantics Approach to the Effective Resolution of Type Equations, *Theoretical Computer Science* 45, pp 293-351.

Hassan AIT-KACI and Roger NASR, 1986, LOGIN: a Logic Programming Language with Built-in Inheritance, *J. of Logic Programming*, 3, pp 185-215.

Martin EMELE, 1988, *A Typed Feature Structure Unification-based Approach to Generation*, Proceedings of the WGNLC of the IECE, Oiso University, Japan.

Martin EMELE and Rémi ZAJAC, 1989, *RETIF: A Rewriting System for Typed Feature Structures*. ATR Technical Report TR-I-0071.

Andreas EISELE and Jochen DÖRRE, 1988, Unification of Disjunctive Feature Descriptions, *Proc. of the 26h Annual Meeting of the ACL*, 7-10 June, Buffalo, pp 286-294.

Pierre ISABELLE and Eliot MACKLOVITCH, 1986, Transfer and MT Modularity, *Proceedings of COLING-86*, Bonn.

Ron KAPLAN and J. BRESNAN, 1982, Lexical Functional Grammar, a Formal System for Grammatical Representation. In J. Bresnan (ed.), *The Mental Representation of Grammatical Relations*, The MIT Press, 1982, pp 173-381.

Robert T. KASPER and William C. ROUNDS, 1986, A Logical Semantics for Feature Structures, *Proceedings of the 24th Annual Meeting of the ACL*, 10-13 June, Columbia University, New-York, pp 257-266.

Robert T. KASPER, 1987, A Unification Method for Disjunctive Feature Descriptions, *Proceedings of the 25th Annual Meeting of the ACL*, 6-9 June, Stanford University, pp 235-242.

Robert T. KASPER, 1988, Conditional Descriptions in Functional Unification Grammar, *Proceedings of the 26h Annual Meeting of the ACL*, 7-10 June, Buffalo, pp 233-240.

Martin KAY, 1984, Functional Unification Grammar: a Formalism for Machine Translation, *Proceedings of COLING-84*, Stanford.

Kiyoshi KOGURE, Hitoshi IIDA, Kei YOSHIMOTO, Hiroyuki MAEDA, Masako KUME, and Susumo KATO, 1988, A Method of Analyzing Japanese Speech Act Types, *2nd International Conference on Theoretical and Methodological Issues in Machine Translation of Natural Languages*, June 12-14, Pittsburgh.

Ikuo KUDO and Hirosato NOMURA, 1986, Lexical-Functional Transfer: A Transfer Framework in a Machine Translation System based on LFG, *Proceedings of COLING-86*, Bonn, 112-114.

Masako KUME, Gayle K. SATO, and Kei YOSHIMOTO, 1989, A Descriptive Framework for Translating Speaker's Meaning - Towards a Dialogue Translation System between Japanese and English, *4th Conference of ACL-Europe*, Manchester.

Kuniaki MUKAI, 1988, Partially Specified Terms in Logic Programming for Linguistic Analysis, *Proc. of the International Conference on Fifth Generation Computer Systems*, Nov.28-Dec.2, Tokyo, pp 479-488.

Carl POLLARD and Ivan A. SAG, 1987, *Information-based Syntax and Semantics*, CSLI, Lecture Notes Number 13.

Stuart M. SHIEBER, 1986, *An Introduction to Unification-based Approaches to Grammar*, CSLI, Lecture Notes Number 4.

Gert SMOLKA, 1988, *A Feature Logic with Subsorts*, LILOG-REPORT 33, IBM Deutschland GmbH, Stuttgart.

Kei YOSHIMOTO, Kiyoshi KOGURE, 1988, *Japanese Sentence Analysis by means of Phrase Structure Grammar*, ATR Technical Report TR-I-0049.

David A. WROBLEWSKI, 1987, Nondestructive Graph Unification, *Proc. of the 6th National Conference on AI, AAAI-87*, July 13-17, Seattle, pp 582-587.

Rémi ZAJAC, 1989, A Transfer Model Using a TFS Rewriting System with Inheritance, *27th Annual Meeting of the ACL*, 26-29 June, Vancouver.

Appendix 1 : a DCG-like grammar

```
:KB G3

% A KBL implementation of the second sample grammar of [Shieber 86].
  Uses a DCG like mechanism do establish the correspondance between
  the feature description and strings. As the mechanism is most
  simple, no left recursive rules are allowed.
  The string is specified in the 'phon' feature
%


:LABEL-ORDER phon, lex, cat, head, subcat, first, rest,
             form, agreement, person, number, gender, np, vp, v, comp,
             whole, front, back, patch, patch0, first, rest.


:SUPPRESSED-LABELS patch, patch0.

LIST = <> | CONS.

APPEND = [front:<>, back:#a=LIST, whole:#a] |
         [front:<#x . #y>,
          back:#z=LIST,
          whole: <#x . #u>,
          patch: APPEND[front:#y, back:#z, whole:#u]] .


X = % List of non-terminal symbols of the grammar. %
    S | VP | NP | V.


S = % S -> NP VP %
     [phon:   #phon,
      head:   #head=[form: finite],
      subcat: <>,
      np:     #subj=NP[phon: #np-phon=CONS],
      vp:     VP[phon:   #vp-phon=CONS,
                 head:   #head,
                 subcat: <#subj>],
      patch: APPEND[front:#np-phon, back:#vp-phon, whole:#phon]].

VP = VP0 | VP1.

VP0 = % trivial verb phrase VP -> V %
      [phon:   #phon=CONS,
       head:   #head,
       subcat: #subcat=<T>,
       vp:     V[phon:   #phon,
                 head:   #head,
                 subcat: #subcat]].
```

```
VP1 = % 1 complement VP -> V NP . %
      % Not VP -> VP NP %
      % because in DCG style grammars left-recursive rules are not allowed. %
     [phon:   #phon,
      head:   #head,
      subcat: #subcat-rest,
      vp:     V[phon:   #vp-phon=CONS,
                 head:   #head,
                 subcat: #subcat=<#obj=NP[phon: #np-phon=CONS] . #subcat-rest>],
      np:     #obj,
      patch0:APPEND[front:<#obj>, back:#subcat-rest, whole:#subcat],
      patch:APPEND[front:#vp-phon, back:#np-phon, whole:#phon]].


;; The LEXICON
;; -----------


Verb = verb[cat: v].
Noun = noun[cat: np].


LEX = NP | V.


NP = UTHER | CORNWALL | KNIGHTS .
V = SLEEP | SLEEPS | STORM | STORMS | STORMED .


;; The nouns
;; ---------

UTHER = Noun
         [phon: <Uther>,
          head: [agreement: [gender: masculine,
                              person: third,
                              number: singular]]].

CORNWALL = Noun
            [phon: <Cornwall>,
             head: [agreement: [gender: masculine,
                                 person: third,
                                 number: singular]]].

KNIGHTS = Noun
           [phon: <knights>,
            head: [agreement: [gender: masculine,
                                person: third,
                                number: plural]]].
```

```
;; The verbs
;; ---------



SLEEPS = Verb
         [phon:   <sleeps>,
          head:   [form: finite],
          subcat: <[cat: np,
                    head:[agreement:[person:third,
                                     number:singular]]]>].

SLEEP  = Verb
         [phon:   <sleep>,
          head:   [form: finite],
          subcat: <[cat: np,
                    head:[agreement:[number:plural]]]>]
       |
         Verb
         [phon:   <sleep>,
          head:   [form: nonfinite],
          subcat: <>] .


STORMS = Verb
         [phon:   <storms>,
          head:   [form: finite],
          subcat: <[cat: np]
                   [cat: np,
                    head: [agreement:[number:singular,
                                      person: third]]]>].
STORMED = Verb
         [phon:   <stormed>,
          head:   [form: finite],
          subcat: <[cat: np]
                   [cat: np]>].

STORM  = Verb
         [phon:   <storm>,
          head:   [form: finite],
          subcat: <[cat: np]
                   [cat: np,
                    head:[agreement:[number:plural]]]>]
       |
         Verb
         [phon:   <storm>,
          head:   [form: nonfinite],
          subcat: <[cat: np]>] .


;; EXAMPLES
;; --------


A1 := % Analysis %
      S[phon: <Uther sleeps>].

G1 := % Generation %
      S[np:UTHER, vp:[vp:SLEEPS]].
```

```
A2  := % Analysis %
       S[phon: <knights sleep>].

G2  := % Generation %
       S[np:KNIGHTS, vp:[vp:SLEEP]].

A3  := % non finite verb form %
       S[phon:< Uther sleep >].

G3  := % 'sleep' doesn't have object %
       S[phon: <Uther sleeps Cornwall>].


A4  := % Analysis %
       S[phon: <Uther storms Cornwall>].

G4  := % Generation %
       S[np:UTHER, vp:[vp:STORMS, np: CORNWALL]].

A5  := % Analysis %
       S[phon: <Uther stormed Cornwall>].

G5  := % Generation %
       S[np:UTHER, vp:[vp:STORMED, np: CORNWALL]].

A6  := % non finite verb form %
       S[phon: <Uther storm Cornwall>].

A7  := % 'storm' should have an object %
       S[phon: <Uther storms>].


A8  := % Analysis %
       S[phon: <knights storm Cornwall>].

G8  := % Generation %
       S[np:KNIGHTS, vp:[vp:STORM, np: CORNWALL]].

A9  := % Analysis %
       S[phon: <knights stormed Cornwall>].

G9  := % Generation %
       S[np:KNIGHTS, vp:[vp:STORMED, np: CORNWALL]].

A10  := % Number agreement %
        S[phon: <knights storms Cornwall>].

A11  := % Subcategorisation %
        S[phon: <Uther stormed knights storm Cornwall>].
```

Appendix 2 : an HPSG grammar for English

```
:KB E-GEN-LEX

:LABEL-ORDER  phon, relation, agent, recipient, object, source, manner,
              syn, head, subcat, dtrs, head-dtr, comp-dtrs, subcat,
              front, back, whole, first, rest,
              patch, patch_phon, patch_comp_phon, patch_subcatfp, patch_map.

DETP = % Determiner phrase %
       E-A | E-THE.

;; Determiner:

E-A   = MINOR[syn: CATEGORY[head: [lexem: "a"]]].
E-THE = MINOR[syn: CATEGORY[head: [lexem: "the"]]].


;; Pronouns:

E-SPEAKER = MINOR[syn: CATEGORY[head:   [lexem: "I"],
                                subcat: <>]].

E-HEARER  = MINOR[syn: CATEGORY[head:  [lexem: "you"],
                                subcat: <>]].

;; Nouns:

E-REGISTRATION-FORM = MINOR[syn: CATEGORY[head:   [lexem:  "registration-form"],
                                          subcat: <LEXICAL-SIGN>]].

;; Verbs:

E-SLEEP = MINOR[syn: CATEGORY[head:   [lexem: "sleep"],
                              subcat: <PHRASAL-SIGN>]].

E-LOVE  = MINOR[syn: CATEGORY[head:   [lexem: "love"],
                              subcat: <PHRASAL-SIGN PHRASAL-SIGN>]].

E-SEND = MINOR[syn: CATEGORY[head: [lexem: "send"],
                             subcat: <PHRASAL-SIGN PHRASAL-SIGN PHRASAL-SIGN>]].

E-CAN = MINOR[syn: CATEGORY[head: [lexem: "could"],
                            subcat: <PHRASAL-SIGN PHRASAL-SIGN>]].


;; Pronoun instances:

E-SPEAKER-1 := PRONOUN[relation: E-SPEAKER].

E-HEARER-1  := PRONOUN[relation: E-HEARER].

;; NP instances:

E-REGISTRATION-FORM-1 := NP[relation: E-REGISTRATION-FORM,
                            spec:     DETP].
```

```
;; intransitive VP instances:

E-SLEEP-VP-1 := E-VP-INTRANS[relation: E-SLEEP].

E-SLEEP-1 := E-INTRANS[relation: E-SLEEP,
                       agent:    E-SPEAKER-1].


;; transitive VP instances:

E-LOVE-VP-1 := E-VP-TRANS[relation: E-LOVE,
                          object:   E-HEARER-1].

E-LOVE-1 := E-TRANS[relation: E-LOVE,
                    agent:    E-SPEAKER-1,
                    object:   E-HEARER-1].

;; ditransitive VP instances:

E-SEND-VP-1 := E-VP-DITRANS[relation:  E-SEND,
                            recipient: E-HEARER-1,
                            object:    E-REGISTRATION-FORM-1].


E-SEND-1 := E-DITRANS[relation:  E-SEND,
                      agent:     E-SPEAKER-1,
                      recipient: E-HEARER-1,
                      object:    E-REGISTRATION-FORM-1].
```

```
:KB E-GEN-TEMPL

:LABEL-ORDER  phon, relation, agent, recipient, object, source, manner,
              syn, head, subcat, dtrs, head-dtr, comp-dtrs, subcat,
              front, back, whole, first, rest,
              patch, patch_phon, patch_comp_phon, patch_subcatfp, patch_map.


PRONOUN =  % Pronouns define fully saturated simple NPs %
          MAJOR
          [relation: #Pronoun,
           dtrs:     TREE[head-dtr: #Pronoun]].

NP =  % abstracted NP head will be filled with relation slot, %
      % determiner with spec slot %
     MAJOR
     [relation: #Noun,
      spec:     #Det,
      dtrs:     TREE
                [head-dtr:  MAJOR[dtrs: TREE[head-dtr:#Noun]],
                 comp-dtrs: <#Det>]].


E-PROP = E-VP-INTRANS | E-VP-TRANS | E-VP-DITRANS .

E-DITRANS =  % partial tree for ditransitive clause %
             MAJOR
             [relation:  #Pred,
              agent:     #Subj,
              recipient: #Obj2,
              object:    #Obj,
              dtrs:      TREE[head-dtr:  E-VP-DITRANS[relation:  #Pred,
                                                      recipient: #Obj2,
                                                      object:    #Obj],
                              comp-dtrs: <#Subj>]].


E-VP-DITRANS =  % partial tree for ditransitive vp with relation, recipient, and
object %
             MAJOR
             [relation:  #Pred,
              recipient: #Obj2,
              object:    #Obj,
              dtrs:      TREE[head-dtr:  #Pred,
                              comp-dtrs: <#Obj2 #Obj>]].

E-VP-TRANS =  % partial tree for transitive vp with relation and object %
             MAJOR
             [relation:  #Pred,
              object:    #Obj,
              dtrs:      TREE[head-dtr:  #Pred,
                              comp-dtrs: <#Obj>]].
```

```
E-TRANS      =  % partial tree for transitive clause %
                MAJOR
                [relation:   #Pred,
                 agent:      #Subj,
                 object:     #Obj,
                 dtrs:       TREE[head-dtr:   E-VP-TRANS[relation: #Pred,
                                                         object:   #Obj],
                                  comp-dtrs: <#Subj>]].

E-VP-INTRANS =  % partial tree for intransitive vp with relation %
                MAJOR
                [relation:   #Pred,
                 dtrs:       TREE[head-dtr:   #Pred,
                                  comp-dtrs: <>]].

E-INTRANS = % partial tree for intransitive clauses %
               MAJOR
               [relation:   #Pred,
                agent:      #Subj,
                dtrs:       TREE[head-dtr:   E-VP-INTRANS[relation: #Pred],
                                 comp-dtrs: <#Subj>]].

CLAUSE = MAJOR
         [relation:   #Pred,
          agent:      #Subj,
          recipient:  #Obj2,
          object:     #Obj,
          dtrs:       TREE
                      [head-dtr:   % VP (Head) %
                                   VP-DITRANS
                                   [relation:   #Pred,
                                    recipient:  #Obj2,
                                    object:     #Obj],
                       comp-dtrs: <#Subj>]].

E-ASK-MODALITY = MAJOR
                 [relation:  #Pred,
                  agent:     #Subj,
                  object:    #Obj,
                  dtrs:      TREE
                             [head-dtr:  #Pred,
                              comp-dtrs: <#Subj #Obj>]].
```

```
:KB E-GEN


:LABEL-ORDER phon, relation, agent, recipient, object, source, manner, syn, head,
             subcat, dtrs, head-dtr, comp-dtrs,
             front, back, whole, first, rest, patch, patch_phon, patch_comp_phon,
             patch_subcatfp, patch_map.

:SUPPRESSED-LABELS patch, patch_phon, patch_comp_phon, patch_subcatfp, patch_map.



APPEND = % Append relation %
         [front: <>,
          back:  #x,
          whole: #x] |
         [front: <#first . #rest>,
          back:  #list,
          whole: <#first . #whole>,
          patch: APPEND
                 [front: #rest,
                  back:  #list,
                  whole: #whole]] .

LIST = % List encodings %
       <> |
       <T . LIST> .



SIGN = PHRASAL-SIGN | LEXICAL-SIGN .



MINOR = % Lexical head: relates lexem with the phonological string %
        LEXICAL-SIGN
        [phon: <#string>,
         syn:  [head: [lexem: #string]]] .



MAJOR = % Grammar Rules %
        PHRASAL-SIGN & HEAD_FP & SUBCAT_FP &
        (% Subject Predicate Rule %
         CH_CO_FP | % Head Complements Rule %
                    HC*_CO_FP) .



HEAD_FP = % Head Feature Principle %
          [syn:  [head: #head],
           dtrs: [head-dtr: [syn: [head: #head]]]] .
```

```
SUBCAT_FP = % Subcat Feature Principle %
            [syn:             [subcat: #rest-subcat],
             dtrs:            [head-dtr:   [syn: [subcat: #subcat]],
                               comp-dtrs: #comps],
             patch_subcatfp: APPEND
                              [front: #comps,
                               back:  #rest-subcat,
                               whole: #subcat]] .


CH_CO_FP = % Complement Head Constituent Order Feature Principle %
           [phon:         #phon,
            dtrs:         [head-dtr:   PHRASAL-SIGN
                                       [phon: #head-phon],
                           comp-dtrs: <[phon: #comp-phon]>],
            patch_phon: APPEND
                          [front: #comp-phon,
                           back:  #head-phon,
                           whole: #phon]] .


HC*_CO_FP = % Head Complements Constituent Order Feature Principle %
            [phon:         #phon,
             dtrs:         MAP
                           [head-dtr:   LEXICAL-SIGN
                                        [phon: #head-phon],
                            comp_phon: #comp-phon],
             patch_phon: APPEND
                           [front: #head-phon,
                            back:  #comp-phon,
                            whole: #phon]] .

MAP = % Empty comp-dtrs list %
      TREE
      [comp-dtrs: <>,
       comp_phon: <>] |
      % Nonempty comp-dtrs list %
      TREE
      [comp-dtrs:        <[phon: #first-comp-phon] .
                          #rest-comp>,
       patch_comp_phon: APPEND
                         [front: #first-comp-phon,
                          back:  #rest-comp-phon,
                          whole: #comp-phon],
       comp_phon:        #comp-phon,
       map_patch:        MAP
                         [comp-dtrs: #rest-comp,
                          comp_phon: #rest-comp-phon]] .
```

Appendix 3: Sample session

```
>(print-type e-speaker-1 e-gen-lex)
E-SPEAKER-1 = PRONOUN[relation: E-SPEAKER] .

>(eval-type e-speaker-1 t e-gen-lex e-gen-templ e-gen)

<E-SPEAKER-1> E-GEN-LEX(1):

PRONOUN[relation: MINOR[syn: CATEGORY
                             [head:    [lexem: "I"],
                              subcat: <>]]]

 Next, Proceed, Exit ? n

<E-SPEAKER-1> E-GEN-LEX(1).E-GEN-TEMPL(1):

MAJOR
[relation: #1=MINOR[syn: CATEGORY
                         [head:    [lexem: "I"],
                          subcat: <>]],
 dtrs:      TREE[head-dtr: #1]]

 Next, Proceed, Exit ? n

<E-SPEAKER-1> E-GEN-LEX(1).E-GEN-TEMPL(1).E-GEN(1):

PHRASAL-SIGN
[phon:      <#1="I" . #4=<>>,
 relation: #5=LEXICAL-SIGN
            [phon: <#1>,
             syn:  CATEGORY
                   [head:    #3=[lexem: #1],
                    subcat: #2=<>]],
 syn:       [head:    #3,
             subcat: #2],
 dtrs:      TREE
            [head-dtr:   #5,
             comp-dtrs: <>,
             comp_phon: #4]]


>(print-type e-registration-form-1 e-gen-lex)
E-REGISTRATION-FORM-1 = NP
                        [relation: E-REGISTRATION-FORM,
                         spec:      DETP] .

> (eval-type e-registration-form-1 t e-gen-lex e-gen-templ e-gen)

<E-REGISTRATION-FORM-1> E-GEN-LEX(1):

NP
[relation: MINOR[syn: CATEGORY
                      [head:    [lexem: "registration-form"],
                       subcat: <LEXICAL-SIGN>]],
 spec:      MINOR[syn: CATEGORY[head: [lexem: "a"]]]]

 Next, Proceed, Exit ? n
```

```
<E-REGISTRATION-FORM-1> E-GEN-LEX(1).E-GEN-TEMPL(1):

MAJOR
[relation: #2=MINOR[syn: CATEGORY
                         [head:    [lexem: "registration-form"],
                          subcat: <LEXICAL-SIGN>]],
 dtrs:      TREE
            [head-dtr:  MAJOR[dtrs: TREE[head-dtr: #2]],
             comp-dtrs: <#1=MINOR[syn: CATEGORY[head: [lexem: "a"]]]>],
 spec:      #1]

 Next, Proceed, Exit ? n

<E-REGISTRATION-FORM-1> E-GEN-LEX(1).E-GEN-TEMPL(1).E-GEN(1):

PHRASAL-SIGN
[phon:      <#2="a" . #9=<#1="registration-form" . #7=<>>>,
 relation: #4=LEXICAL-SIGN
            [phon: <#1>,
             syn:  CATEGORY
                   [head:   #5=[lexem: #1],
                    subcat: #6=<#3=LEXICAL-SIGN
                               [phon: <#2>,
                                syn:  CATEGORY[head: [lexem: #2]]] .
                               #8=<>>]],
 syn:       [head:   #5,
             subcat: #8],
 dtrs:      TREE
            [head-dtr:  PHRASAL-SIGN
                        [phon: #9,
                         syn:  [head:   #5,
                                subcat: #6],
                         dtrs: TREE
                               [head-dtr:  #4,
                                comp-dtrs: <>,
                                comp_phon: #7]],
             comp-dtrs: <#3>],
 spec:      #3]

 Next, Proceed, Exit ? n

<E-REGISTRATION-FORM-1> E-GEN-LEX(2):

NP
[relation: MINOR[syn: CATEGORY
                      [head:    [lexem: "registration-form"],
                       subcat: <LEXICAL-SIGN>]],
 spec:     MINOR[syn: CATEGORY[head: [lexem: "the"]]]]


>(print-type e-sleep-1 e-gen-lex)
E-SLEEP-1 = E-INTRANS
            [relation: E-SLEEP,
             agent:    E-SPEAKER-1] .

>(eval-type e-sleep-1 t e-gen-lex e-gen-templ e-gen)

<E-SLEEP-1> E-GEN-LEX(1):
```

```
E-INTRANS
[relation: MINOR[syn: CATEGORY
                      [head:    [lexem: "sleep"],
                       subcat: <PHRASAL-SIGN>]],
 agent:    PRONOUN[relation: MINOR[syn: CATEGORY
                                        [head:    [lexem: "I"],
                                         subcat: <>]]]]

 Next, Proceed, Exit ? n

<E-SLEEP-1> E-GEN-LEX(1).E-GEN-TEMPL(1):

MAJOR
[relation: #2=MINOR[syn: CATEGORY
                         [head:    [lexem: "sleep"],
                          subcat: <PHRASAL-SIGN>]],
 agent:    #3=MAJOR
           [relation: #1=MINOR[syn: CATEGORY
                                    [head:    [lexem: "I"],
                                     subcat: <>]],
            dtrs:     TREE[head-dtr: #1]],
 dtrs:     TREE
           [head-dtr:  MAJOR
                       [relation: #2,
                        dtrs:     TREE
                                  [head-dtr:  #2,
                                   comp-dtrs: <>]],
            comp-dtrs: <#3>]]

 Next, Proceed, Exit ? n

<E-SLEEP-1> E-GEN-LEX(1).E-GEN-TEMPL(1).E-GEN(1):

PHRASAL-SIGN
[phon:     <#1="I" . #13=<#7="sleep" . #11=<>>>,
 relation: #8=LEXICAL-SIGN
           [phon: <#7>,
            syn:  CATEGORY
                  [head:    #9=[lexem: #7],
                   subcat: #10=<#6=PHRASAL-SIGN
                                [phon:     <#1 . #5=<>>,
                                 relation: #4=LEXICAL-SIGN
                                           [phon: <#1>,
                                            syn:  CATEGORY
                                                  [head:    #3=[lexem: #1],
                                                   subcat: #2=<>]],
                                 syn:      [head:   #3,
                                            subcat: #2],
                                 dtrs:     TREE
                                           [head-dtr:  #4,
                                            comp-dtrs: <>,
                                            comp_phon: #5]] .
                                #12=<>>]],
 agent:    #6,
 syn:      [head:   #9,
            subcat: #12],
 dtrs:     TREE
           [head-dtr:  PHRASAL-SIGN
                       [phon:     #13,
```

```
                    relation: #8,
                    syn:      [head:    #9,
                               subcat: #10],
                    dtrs:     TREE
                              [head-dtr:  #8,
                               comp-dtrs: <>,
                               comp_phon: #11]],
            comp-dtrs: <#6>]]



>(print-type e-love-1 e-gen-lex)
E-LOVE-1 = E-TRANS
           [relation: E-LOVE,
            agent:    E-SPEAKER-1,
            object:   E-HEARER-1] .

>(eval-type e-love-1 t e-gen-lex e-gen-templ e-gen)

<E-LOVE-1> E-GEN-LEX(1):

E-TRANS
[relation: MINOR[syn: CATEGORY
                      [head:   [lexem: "love"],
                       subcat: <PHRASAL-SIGN PHRASAL-SIGN>]],
 agent:    PRONOUN[relation: MINOR[syn: CATEGORY
                                        [head:   [lexem: "I"],
                                         subcat: <>]]],
 object:   PRONOUN[relation: MINOR[syn: CATEGORY
                                        [head:   [lexem: "you"],
                                         subcat: <>]]]]

 Next, Proceed, Exit ? n

<E-LOVE-1> E-GEN-LEX(1).E-GEN-TEMPL(1):

MAJOR
[relation: #4=MINOR[syn: CATEGORY
                         [head:   [lexem: "love"],
                          subcat: <PHRASAL-SIGN PHRASAL-SIGN>]],
 agent:    #5=MAJOR
           [relation: #1=MINOR[syn: CATEGORY
                                    [head:   [lexem: "I"],
                                     subcat: <>]],
            dtrs:     TREE[head-dtr: #1]],
 object:   #3=MAJOR
           [relation: #2=MINOR[syn: CATEGORY
                                    [head:   [lexem: "you"],
                                     subcat: <>]],
            dtrs:     TREE[head-dtr: #2]],
 dtrs:     TREE
           [head-dtr:  MAJOR
                       [relation: #4,
                        object:   #3,
                        dtrs:     TREE
                                  [head-dtr:  #4,
                                   comp-dtrs: <#3>]],
            comp-dtrs: <#5>]]
```

```
 Next, Proceed, Exit ? n

<E-LOVE-1> E-GEN-LEX(1).E-GEN-TEMPL(1).E-GEN(1):

PHRASAL-SIGN
[phon:      <#2="I" . #7=<#1="love" . #20=<#13="you" . #11=<>>>>,
 relation: #21=LEXICAL-SIGN
           [phon: <#1>,
            syn:  CATEGORY
                  [head:    #10=[lexem: #1],
                   subcat: <#18=PHRASAL-SIGN
                            [phon:      <#13 . #17=<>>,
                             relation: #16=LEXICAL-SIGN
                                       [phon: <#13>,
                                        syn:  CATEGORY
                                              [head:    #14=[lexem: #13],
                                               subcat: #15=<>]],
                             syn:      [head:   #14,
                                        subcat: #15],
                             dtrs:     TREE
                                       [head-dtr:  #16,
                                        comp-dtrs: <>,
                                        comp_phon: #17]] .
                            #19=<#8=PHRASAL-SIGN
                                 [phon:      <#2 . #3=<>>,
                                  relation: #4=LEXICAL-SIGN
                                            [phon: <#2>,
                                             syn:  CATEGORY
                                                   [head:    #5=[lexem: #2],
                                                    subcat: #6=<>]],
                                  syn:      [head:   #5,
                                             subcat: #6],
                                  dtrs:     TREE
                                            [head-dtr:  #4,
                                             comp-dtrs: <>,
                                             comp_phon: #3]] .
                                 #9=<>>>]],
 agent:    #8,
 object:   #18,
 syn:      [head:   #10,
            subcat: #9],
 dtrs:     TREE
           [head-dtr:  PHRASAL-SIGN
                       [phon:     #7,
                        relation: #21,
                        object:   #18,
                        syn:      [head:   #10,
                                   subcat: #19],
                        dtrs:     TREE
                                  [head-dtr:  #21,
                                   comp-dtrs: <#18 .
                                              #12=<>>,
                                   map_patch: TREE
                                              [comp-dtrs: #12,
                                               comp_phon: #11],
                                   comp_phon: #20]],
            comp-dtrs: <#8>]]
```

```
>(print-type e-send-1 e-gen-lex)
E-SEND-1 = E-DITRANS
           [relation:  E-SEND,
            agent:     E-SPEAKER-1,
            recipient: E-HEARER-1,
            object:    E-REGISTRATION-FORM-1] .

>(eval-type e-send-1 t e-gen-lex e-gen-templ e-gen)

<E-SEND-1> E-GEN-LEX(1):

E-DITRANS
[relation:  MINOR[syn: CATEGORY
                       [head:   [lexem: "send"],
                        subcat: <PHRASAL-SIGN PHRASAL-SIGN PHRASAL-SIGN>]],
 agent:     PRONOUN[relation: MINOR[syn: CATEGORY
                                         [head:   [lexem: "I"],
                                          subcat: <>]]],
 recipient: PRONOUN[relation: MINOR[syn: CATEGORY
                                         [head:   [lexem: "you"],
                                          subcat: <>]]],
 object:    NP
            [relation: MINOR[syn: CATEGORY
                                  [head:   [lexem: "registration-form"],
                                   subcat: <LEXICAL-SIGN>]],
             spec:     MINOR[syn: CATEGORY[head: [lexem: "a"]]]]]

 Next, Proceed, Exit ? n
```

```
<E-SEND-1> E-GEN-LEX(1).E-GEN-TEMPL(1):

MAJOR
[relation:   #7=MINOR[syn: CATEGORY
                           [head:    [lexem: "send"],
                            subcat: <PHRASAL-SIGN PHRASAL-SIGN PHRASAL-SIGN>]],
 agent:      #8=MAJOR
             [relation: #1=MINOR[syn: CATEGORY
                                      [head:    [lexem: "I"],
                                       subcat: <>]],
              dtrs:     TREE[head-dtr: #1]],
 recipient: #6=MAJOR
             [relation: #4=MINOR[syn: CATEGORY
                                      [head:    [lexem: "you"],
                                       subcat: <>]],
              dtrs:     TREE[head-dtr: #4]],
 object:     #5=MAJOR
             [relation: #3=MINOR[syn: CATEGORY
                                      [head:    [lexem: "registration-form"],
                                       subcat: <LEXICAL-SIGN>]],
              dtrs:     TREE
                        [head-dtr:  MAJOR[dtrs: TREE[head-dtr: #3]],
                              comp-dtrs: <#2=MINOR[syn: CATEGORY[head:   [lexem:
"a"]]]>],
              spec:     #2],
 dtrs:       TREE
             [head-dtr:  MAJOR
                         [relation:  #7,
                          recipient: #6,
                          object:    #5,
                          dtrs:      TREE
                                     [head-dtr:  #7,
                                      comp-dtrs: <#6
                                                  #5>]],
              comp-dtrs: <#8>]]

 Next, Proceed, Exit ? n

<E-SEND-1> E-GEN-LEX(1).E-GEN-TEMPL(1).E-GEN(1):

PHRASAL-SIGN
[phon: <"I" "send" "you" "a" "registration-form">]
```

-o-o-o-o-o-o-o-o-o-