A Grammar Writing Language for the Machine Translation System Using Transfer Method

Cheol-Jung Kweon, Won-Seok Kang,
Key-Sun Choi and Gil-Chang Kim

A Grammar Writing Language, which has tree to tree transformation as basic operation, is designed and implemented. Grammars which are used in three stages, i.e. Analysis, Transfer and Generation can be expressed by GWL. To easily express grammar rules which contain tree to tree transformation, GWL offer 6 unary operators—Unique, One, Dagger, Optional, Any, Star. By these operators each grammar rule can be simply expressed. And in order to increase productivity of grammar writing and readability of written grammar GWL supports not only general control structure but also abstracted control structure.

1. Introduction

When developing machine translation systems, we must take into account a mechanism of improving and modifying the system as needed. In general, the development process of machine translation systems can be seen as in Figure 1.

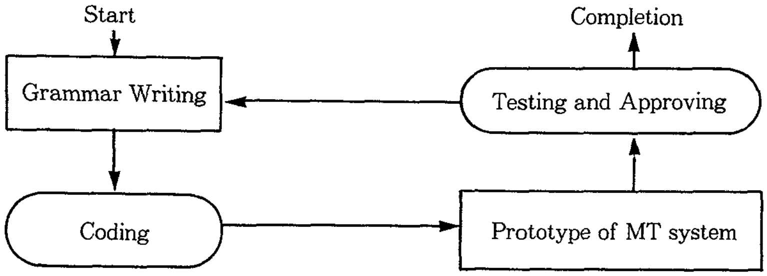


Figure 1. Developing Cycle of Machine Translation System.

In such a developing environment as shown above, we must be able to change or expand the system with no big difficulty. If the system is implemented in conventional programming languages, there exist many problems in modifying the system. In particular, some difficulties may occur in understanding coded grammar and maintaining the consistency of the whole grammar when new fraction is added.

Thus, there is the need at some way to provide the system with better readability and expandability by separating the control routine from grammar rule, which will also allow a good deal of convenience to grammar developers (Boitet et al. (1982)).

When a grammar writing language is to be developed some important issues should not be neglected. Firstly, the syntax and semantics of the language may well be fairly simple since it is intended to aid grammar writers to write and read grammar with much ease. A grammar writing language also must provide ways of handling the syntactic or semantic ambiguities of the grammar. For instance, when the ambiguity arises because of multiple categories of a word in English sentence analysis system, the grammar writing language must be able to handle the case by itself giving no extra burden to grammar writers.

In developing machine translation systems, in general a single person cannot write the whole grammar because of the huge size of the grammar; therefore there should be a mechanism that allows multiple grammar writers to work independently of others. The hierarchical structure of grammar needs to be supported so as to enable the design of the whole grammar in top-down in writing the grammar.

In addition to the points mentioned above a grammar writing language should provide the following functions. (1) A grammar writing language should be able to handle general linguistic phenomena that are common in many languages. (2) A grammar writing language should be able to handle the rules associated with words, or the rules contained in a lexicon. (3) To make it possible to describe the processes of all the three translation stages in a uniform manner, a grammar writing language should be suitable for each of the stages (Nakamura et al. (1984)).

GWL that is to be described is based on the principles discussed above. In section 2 the basic data structure of GWL is described, and in section 3 the syntax, the semantics and the components of GWL is explained. Section 4 describes how GWL supports the handling of ambiguities and access the dictionaries. Section 5 introduces the structure of the implemented grammar writing language system in detail. Section 6 concludes the discussion.

2. Annotated Tree

The process of machine translating a sentence is a successive transformation of a representation of the sentence which added some information (Johnson et al. (1984)).

$$\begin{tikzcd}\text{source text} \arrow[r] & Rp \cdots \arrow[r] & Rq \cdots \arrow[r] & Rr \arrow[r] & \text{target text}\end{tikzcd}$$

Figure 2. Process of MT.

The structure representing a sentence in a machine translation system is the basic data structure of a grammar writing language. Then when the data structures of all the stages—analysis, transfer, generation—can be treated in the same way the grammar writing language can express the sentence structure of each stage of analysis, transfer, and generation in the same structure. GWL takes annotated trees as a basic data structure to represent sentence structures. A node in an annotated tree contains linguistic information and temporary information needed in writing the grammar in a tree frame (Nakamura et al. (1984)).

From Figure 2, we can see translation process is a sequence of annotated tree transformations. An annotated tree transformation includes a partial change of the tree structure and the change of the information stored in the nodes. To express an annotated tree GWL offer 6 unary operator described in the following section.

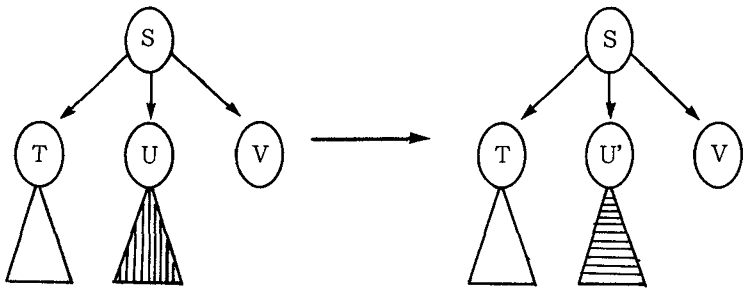


Figure 3. Tree to Tree Transformation.

3. GWL (Grammar Writing Language)

To make grammar writing using a grammar writing language an easier task, one must be able to design in top-down. For example, analyzing an English sentence requires analyzing NP and VP that consititute the sentence. Similary, for NP and VP to be analyzed the subconstituents of NP and VP need to be analyzed. The GWL is composed of three components: GR (GRammar), PG (Part of Grammar), and RR (Rewriting Rule). Through these three components grmmmar writers can design the grammar in top-down.

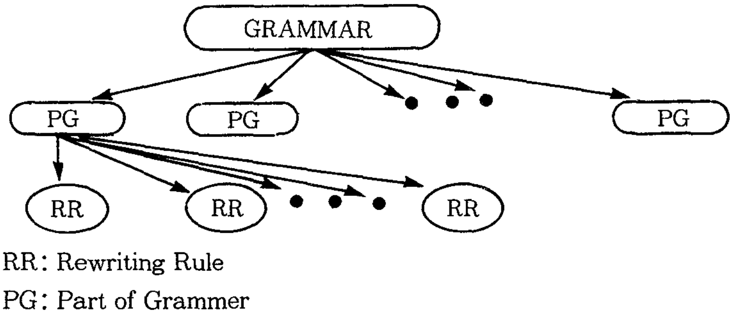


Figure 4. The Hierarchical Structure of Grammar in GWL.

3.1. Tree Transformation in GWL

RR provides with the function for tree to tree transformation (refer to Figure 3). To transform an annotated tree to another RR facilitates an explicit representation of the old and new annotated trees. A RR consists of seven components.

[1] *Instruction* Which describes the application method of the RR has the information on mode and depth. Mode information helps the grammar writer choose the way the RR can be applied. These are six types of modes. Depth indicates the limit of search depth when an annotated tree of a sentence is searched for a pattern to be matched with a partial tree to which a RR is to RR applied. Using depth information, redundant tree search can be avoided.

[2] $Tree_l$ represents a partial tree pattern that a RR is about to process.

The partial tree pattern is represented using *six unary operator*. Unary operand for terminal node is category name (ex, noun, verb, …), and for nonterminal node it is node name (ex, NP, VP, …). The six unary operators are as follows:

~(any) : match with zero or more arbitrary nodes.

?(one) : match with one arbitary nodes.

&(optional) : match with zero or one specific node.

!(unique) : match with one specific node

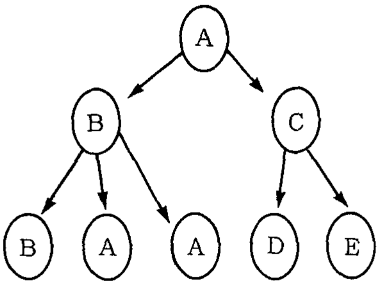
+(dagger) : match with one or more specific node.

*(star) : match with zero or more specific node.

Followings show the combination of the operators that can come together.

```
<tree>     := unique <children>
<children> := <tree> | <tree> <children>
              <cx> | {<cx> any}
<cx>       := {unique | one | dagger | optional | star}
```

As an example there can be many expressions of the pattern that can be matched with tree in Figure 5. The six unary operators allow the construction of any tree structure that grammar writers may want. The partial tree or node matched with an operand will be bound to the operand. Also the operands can be used in *condition* and *action* parts of a RR as variables of tree–type.



(A! ~CH)
(A![1] (B![1] B&[2] A*[2])
(C! G? E!))
(A! ~CH C!)

Figure 5. An Example of Expressions of a Tree Pattern.

[3] *With–tree* describes the information in the partial tree node a RR tries to transform. The partial tree node selected from the whole annotated tree must satisfy the condition specified in the information described by *With–tree.*

[4] In *Var–def* the variables used in Condition and Action parts are de-

fined. There are two types of variables: tree type and feature type. A partial tree can be bound to a tree type variable while a feature value can be bound to a feature-typed variable.

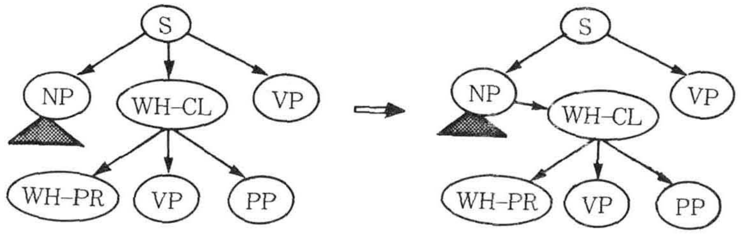
[5] *Condition* part specifies the condition for the action part to be applicable.

[6] In *Action* part functions for the manipulation of partial trees, feature, and feature values as well as dictionary access can be prescribed.

[7] *Tree*$_2$ is the tree to be replaced by an input tree. Figure 6 shows an example of a tree to tree transformation by only *Tree*$_1$ and *Tree*$_2$.

In GWL the methods of applying rules can be selected through mode information. For instance, as there can be many NP's in an English sentence, the NP's that can be expressed in the same pattern can be processed by one rule. GWL does so by defining mode in RR's.

This language which was designed by us has very simple syntax.



```
Clause-RR. Begin
            INST mode : 4;
            (S! (NP!~CH) WH-CL! VP!)
            (S (NP CH WH-CL) VP)
Clause-RR. END
```

Figure 6. An Example of a Tree to Tree Transformation Using RR in GWL.

3.2. Control Structure in GWL

A PG can be any fraction of the grammar that can be grouped to be a grammatical unit. The rules that make up NP's in English can be treated as a unit. In this way grammar writers can get simplified view of the grammar.

```
/* NP<—PRO | [ART] [ADJP] (NOUN)
 ADJP<—PRO | [ADV] (ADJ)          */

  Simple-noun-phrase-PG. begin
       order : 2;
          /* apply each rule once sequentially */
       pro-to-adjp-rr;
       adv-adj-to-adjp-rr;
       art-adjp-noun-to-np-rr;
       pro-to-np-rr;
  Simple-noun-phrase-PG. END
```

In above example *order : 2* means that all the constituents (PG or RR) of PG need to be executed in sequence. Thus in the above PG, the RR that changes possessive pronouns such as *his* and *my* into adjective phrase is applied, then from the result an adjective phrase composed of adverb and adjective will be formed. The rest of the PG will be executed in sequence.

In PG, also, control scheme of the analysis can be described separately from the detailed tree transformations. As components of PG, there are RR and PG. The order and method of the execution of these components are decided by one of the predefined execution orders (abstracted control structure) and the control structures (if branch, while). Through the two control structures an arbirary control structure can be implemented. There are six types of orders that can be selected in PG (see appendix).

3.3. GR (GRammar) in GWL

In general, the sequence of applying groups of grammars is of importance. That is, in analyzing a Korean sentence phrases without embedded sentences is processed first, and then phrases with embedded sentences will be processed followed by the process of the whole sentence. English analysis can be seen in the same manner. NP's and VP's without embedded sentences can be processed before one with embedded clauses.

In GR in the GWL higher level control over the grammar can be expressed. A GR is a sequence of RR's. The RR's will be executed in sequence, and the execution will come to a halt wherever a PG fails.

```
Grammar. Begin
        NP-PG;
        VP-PG;
        CLAUSE-PG;
        SENT-PG;
Grammar. End
```

A simple example of English syntactic analysis is shown above. After NP's in a sentence are processed VP's are analyzed, and clauses will be formed. Then the analysis of a sentence will be completed in SENT-PG.

4. Handling of Ambiguities and Dictionary Access in GWL

There arise various ambiguities from the application of the grammar in all stages of the analysis (syntactic, semantic) of machine translation systems. GWL helps grammar writers handle the ambiguities with great efficiency.

Syntactic ambiguities occur frequently from the words with multiple categories. From morphological analysis only simple ambiguities can be resolved based on lexical information. Even with full utilization of lexical information ambiguity resolution in morphological level is severely limited. Most syntactic ambiguities can be resolved through syntactic and semantic analysis.

Without proper resolution ambiguities during the process the order of ambiguities can be of combinatorial explosion.

GWL aids users to deal with ambiguities avoiding the dreadful complexity of ambiguities by supporting structure sharing and operation sharing. The structure sharing is implemented by *dummy* node. In GWL if two or more trees occur when a RR is applied, the multiple trees are connected under the node *dummy*. When a RR is applied to the children of a dummy node the rule will be applied to all the children unless a special mark is set (mode 5 in a RR). That is by setting the special mark (mode 5) a grammar writer can process each of the children of a dummy node separately.

For example, the word *book* can be categorized noun, verb and adverb.

Thus the sentence *This is a book* can be represented as Figure 7 after morphological analysis.

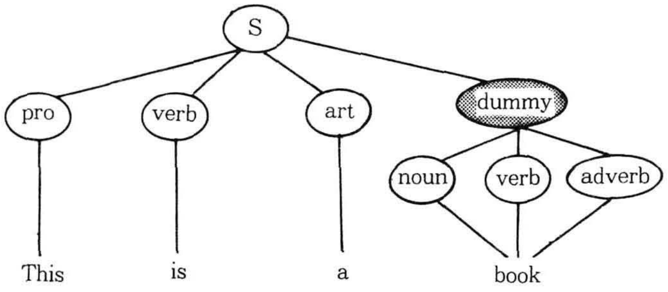


Figure 7. A Representation of Multiple Categories.

Moreover, Figure 8 shows how structure ambiguity can be treated by GWL interpreter. Operation sharing can be implemented naturally by structure sharing.

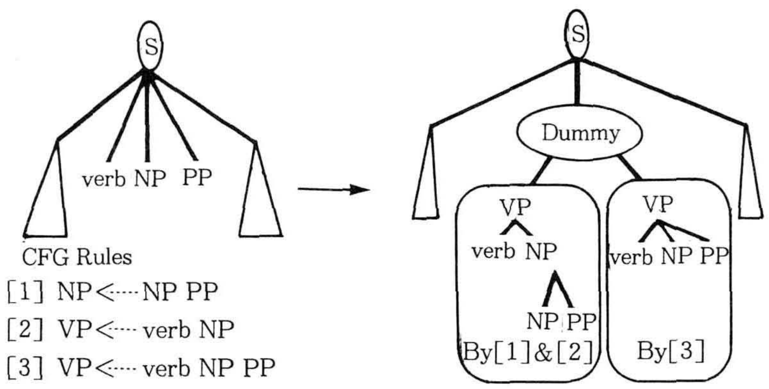


Figure 8. A Representation of Stuctural Ambiguity.

GWL provides two ways for accessing a dictionary to obtain lexical information with which lexical dependent rule can be applied. When a grammar writer wants to get specific lexical information, functions for extracting the feature value over dictionaries can be used. If a grammar writer wants to describe a rule using lexical information at particular words mode 5 of RR can be selected.

5. Implementation of GWL

GWL was implemented by devising translator and interpreter. The grammar written by grammar writers is transformed into s-expression at Lisp through the translator, and the s-expressions can be executed by the interpreter. The interpreter accepts the translated grammar and the result from morphological analyzer to do the analysis. The output of the interpreter is an input to morphological generator. The function of dictionary interface is to access and interpret the information written in the dictionary.

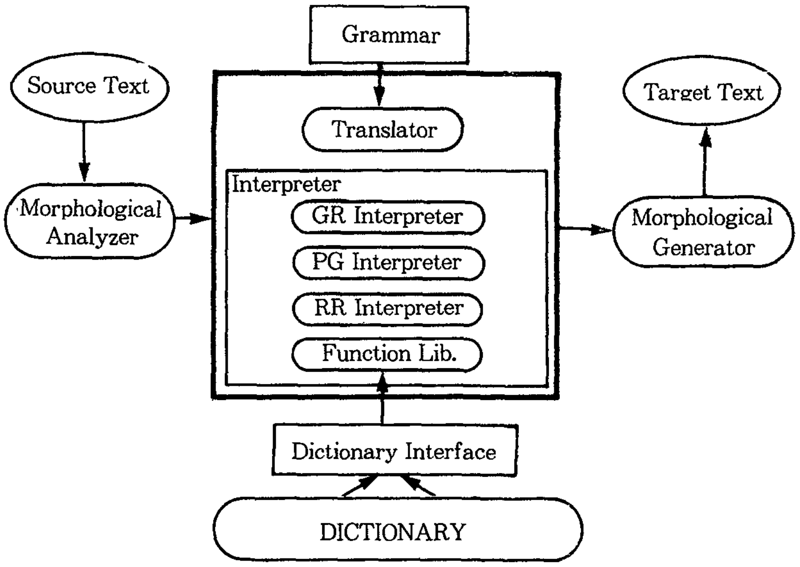


Figure 9. System Configuration of Implemented GWL.

The system was implemented on UNIX environment, and translator was built using Lex and Yacc. The interpreter was coded in KCL (Kyoto Common Lisp).

6. Conclusion

This paper discussed on the design and implementation of the grammar writing language (GWL) based on tree to tree transformation. Grammars for each stage of a machine translation system can be expressed in GWL. GWL allows grammar writers to design grammars in a top-down manner.

Also, to enhance the readability, tree to tree transformation which is a basic operation is explicitly expressed in a RR. GWL facilitates structure sharing and operation sharing to help grammar writers handle ambiguities.

References

Boitet, Ch. et al. (1982) 'Implementation and Conventional Environment of ARIANE 78. 4,' *Proc. COLING '82.*

Johnson, R. L. et al. (1984) 'The Design of the Kernel Architecture for the EUROTRA Software,' *COLING '84*, pp. 226–235.

Muraki, L. (1986) 'VENUS: Two-Phase Machine Translation System,' *Future Generation Computer System* 2, pp. 117–119.

Nakamura, Jun-ichi, Jun-ichi Tsujii, Makoto Nagao (1984) 'Grammar Writing System (GRADE) of Mu-Machine Translation Project and its Characteristics,' *COLING '84.*

Appendix: Orders of PG

Constituents of PG: $R_1, R_2, \cdots, R_n$
Succeed (R_i)=T or F
Return (T or F): Return value of PG

```
Order 0: execute $R_1, \cdots\cdots R_i$ s.t
         Succeed $(R_m)$=F(m=1, ⋯, i–1) and
         Succeed $(R_i)$=T.
         Return (T) if i<n+1
         Return (F) otherwise.
Order 1: execute $R_1, \cdots\cdots, R_i$ s.t
         Succeed $(R_m)$=T(m=1, ⋯, i–1) and
         Succeed $(R_i)$=F.
         Return (T) if i>1
         Return (F) otherwise.
Order 2: execute $R_1, \cdots\cdots, R_n$
         Return (T) if $l_i$ Succeed $(R_i)$=T(0<i<n+1)
         Return (F) otherwise.
Order 3: execute $R_{1,1}\cdots, R_{i,1}, R_{n,2}\cdots,$
         $R_{1,m}, \cdots, R_{k,m}$ s. t
         Succeed $(R_{p,1})$=F(0<p, 1<i, 1)
         Succeed $(R_{i,1})$=T(i, 1<n+1)
```

```
          Succeed ($R_{q,2}$)=F($0<q$, $2<j$, 2)
          Succeed ($R_{j,2}$)=T(j, $2<n+1$)
           :
           :
          Succeed ($R_{r,m}$)=F($0<r$, $m<k$, m)
          Succeed ($R_{j,2}$)=T(k, $m=n$).
          Return (T) if $m>1$
          Return (F) otherwise.
Order 4:  execute $R_1$, …, $R_n$
          Return (T) if $1V_i$ Succeed ($R_i$)=T($0<i<n+1$)
          Return (F) otherwise.
Order 5:  $R_1$, …, $R_n$.
          Return (Succeed ($R_i$)) ($0<i<n+1$)
```

Center for Artificial Intelligence Research (CAIR)
Computer Science Department, KAIST
Korea