# 言語学習用オーサリングシステムのための「フレージングメカニズム」

立　野　　　彰*

## "Phrasing Mechanism" for Language Learning Authoring System

Akira　　TATENO

富山工業高等専門学校紀要　第30巻　抜刷平成8年3月

# 言語学習用オーサリングシステムのための「フレージングメカニズム」

立 野 彰 *

# "Phrasing Mechanism" for Language Learning Authoring System

Akira TATENO

Abstract

The purpose of this paper is to introduce a mechanism to give syntactical information --- a built-in mechanism which helps an author give his/her students syntactical information on the target language with relatively small labor on the part of an author. The mechanism has been devised as an extension to and implemented for NEC version of 'CALIS' -- Computer Assisted Language Instruction System developed in Humanities Computing Facility at Duke University. The usage and basic idea of the mechanism will be explained with examples of a CALIS script -- a course ware file for CALIS. The source code -- about one thousand lines written in the C language -- are not included here. All the code except one function 'SpellWord' whose original code was implemented in Humanities Computing Facility, is available freely by contacting me,

## Preface

Functions to give appropriate hints responding to students' answers, take up more and more central part in authoring systems for language learning. As operating systems for personal computers get sophisticated and cover more and more fields, such as windowing, pictures, sound, movies, it has become much easier to make user interface, which used to occupy a large part of programmers' work. In "Microsoft Windows" operating system, some programming tools like "Visual Basic", "Delphi", for example, even give teachers enough authoring environment if several functions essential to language learning are provided. Though the "Phrasing Mechanism" has been devised as an extension of a specific authoring system -- NEC version of CALIS for learning English, I hope it might serve computer assisted language learning as one of such functions.

The mechanism is a group of functions to check a typed answer by a student and feed back some hints including syntactical or grammatical information on the basis of correct answers given by an author. It exploits one of the characteristics of the English language namely that words in it are separated by spaces, and therefore does not work on languages, such as Japanese. Still there are many languages whose words can be picked out as letters between spaces, and generally in these languages syntax plays an important part. I believe that the mechanism will do as well in those languages as in English.

This paper composes of two parts. In the following section - Usage of "Phrasing Mechanism" -, the mechanism will be explained through its usage with examples of a CALIS script -- a course ware file for CALIS. In the latter - Data Structure of "Phrasing Mechanism" -, the data structure used in the mechanism will be described so that readers may have clearer understanding of the first section.

As I suppose that most readers' concern is not about its actual details but about what the mechanism can do or not, no source code will be shown. The source code -- about one thousand lines written in the C language -- are available for those who might be interested, except one function 'SpellWord' whose original code was implemented in Humanities Computing Facility.

I sincerely thank Humanities Computing Facility at Duke University for permitting me to transplant their

*一般科目 英語

CALIS to NEC 98 series of computers. Without this work, I never thought of devising this kind of mechanism.

I Usage of "Phrasing Mechanism"

If an application for language learning allowed *only one* correct answer to a question, it would surely be a poor product. Usually authoring systems for language learning like CALIS provides a mechanism to accommodate a list of several correct or incorrect answers and respond to a student according to what he/she types or chooses on the basis of the list. In a CALIS script, for example, the lines

〔 i 〕

```
+You should respect the rights of {others | other people} ; any response hare
+You must respect the rights of {others | other people} ; any response hare
```

admit the following answers 1.-4. typed by a student as true,

1. You should respect the rights of others
2. You must respect the rights of others
3. You should respect the rights of others people
4. You must respect the rights of other people

and respond to the answers 1. 3. and 2. 4., with a comment written by an author after the semicolon ' ; '. The wild card '{ | }'here means the alternative of 'othere' or 'other people'.

Likewise for anticipated incorrect answers, the lines

〔 ii 〕

```
-& other ; any response hare
-& others * & ; any response hare
```

enable any answer ended by the word 'other' or including the combination of 'others' and any words, to be judged as false and each responded by showing any feed-back by the author written after the semicolon. The wild card '&' means any sequence of words in a student's answer and '*' any one word.

CALIS also provides a very useful function called 'SpellWord' which checks each word in a student's answer and gives hints of the correction for a misspelled word. Suppose that a student types 'light' in the example above, then the letter 'l' in the word will be shown highlighted to indicate that 'l' should be replaced by another letter and one under-bar '-' is added at the end of it to point out one missing letter . It saves an author a lot of labor because the only thing the author need to do is write 〈SPCK〉 after the semicolon in the example〔 ii 〕. Especially placed below〔 ii 〕,

```
-& ; 〈SPCK〉
```

enables any other incorrect answer by a student than checked in〔 ii 〕to provoke the 'SpellWord' function.

The "Phrasing Mechanism" is an extension to this mechanism provided by CALIS to add to it means to give hints for sentence construction such as words orders and categories of words.

The "Phrasing Mechanism" serves the following task.

a) To choose the most similar correct answer to student's answer from the list of correct answer.

In a CALIS script when an author writes the lines below for the list

```
+I {don't | do not} know what to do with this problem ;
+I {don't | do not} know what I should do with this problem ;
```

the mechanism compares the sentence typed by the student with all the alternatives described by the lines and shows hints on the basis of the most similar correct answer. An author can write as many lines as he/she likes. Also as many alternative words as possible can be put at the place separated by '|' in '{ | | |.....}'. In a special case, an alternative containing nothing in { | }, like "I think { | that} ....", means the possibility of omission, thus the alternative of "I think .... " or "I think that .... ". The '{ | }' notation allows iteration in it, so the author can use '{ | }' in another '{ | }' like "..... {by | in{his | the}} car .....". The number of alternatives '{ | } in a line is not limited.

b) To check a student's answer by smaller parts in a correct answer

In responding to a student's answer it is important to let the student know what part in his/her answer is correct and what part is not. The mechanism enables an author to separate a part -- hereafter I call the part 'phrase' -- by putting it in parentheses '('and')' in a correct answer. If the author writes following lines for the list of correct answers

```
+I ({don't | do not}know) ((what to do) (with this problem)) ;
+I ({don't | do not}know) (what I (should do) (with this problem)) ;
```

and the student types an wrong answer except a phrase above, say "What we don't know to this problem do with', then the mechanism gives hints showing that the phrase "don't know" is correct. A parenthesis '('must have the matching parenthesis')'. As the example shows, an author can use a pair of parentheses within another pair. In order to avoid illegibility through iteration of parentheses, '<'and'>' are available in place of '('and')'. The slight difference of these two separators will be explained later.

c) To accommodate grammatical information in a correct answer

An author can include some information on a word or phrase in a correct answer by using '=' as in the following lines.

```
+<=SI><=V(=Au 2 {don't | do not}know)><what(=Inf to do)<(=Pre with)this problem>> ;
+I ({don't | do not}know) ((Wh=what)<=SI>(=V should do) (with this problem)) ;
```

If a student does not type the phrases with a grammatical category correctly, the mechanism shows the grammatical category in place of actual words. For example, if the student fails to type the phrase "to do", "...... Inf( ^ ^ ) ......" will be shown as a hint indicating that an infinitive which requires two words is needed. What category should be adopted or how to abbreviate it is basically left to an author. The maximum number of the letters for a grammatical category is 3. The mechanism simply ignores the

remaining letters. There are some reserved category words. For example, 'Pre' is a reserved word for a preposition. For reserved words, the mechanism checks student's answer with all the words of the category. If a student types 'on' in place of 'with' in the example above, it will give a hint suggesting that 'on' is not correct but still of the same category.

d) To measure the divergence of a student's answer from the corresponding correct answer

The mechanism checks the word order of a student's answer by every unit of phrases of which the largest is the whole answer itself and evaluate the whole answer of the student. A typical hint screen shown by the mechanism is as follows.

```
DIAGNOSIS :    CONSTRUCTION  33%     CORRECT WORDS  86%
YOUR ANSWER :
I not know how do this problem
CORRECTION :
I V(Au 2 (ˆ not)know) O(w___ Inf(∧do)Pre[ˆ]this problem)
```

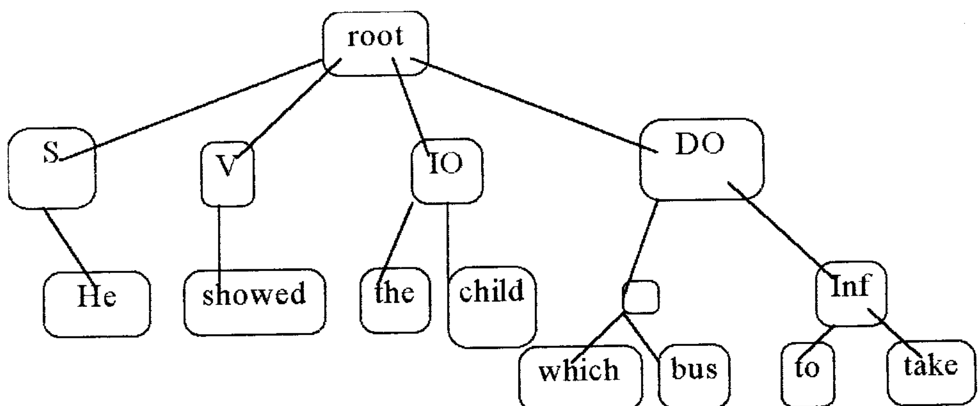
CONSTRUCTION shows the degree of correctness of a student's answer and CORRECT WORDS shows the ratio of correct words to all the words typed by the student. After "YOUR ANSWER : ", the student's answer is shown as follows : if a word can be used without correction, the word will be colored white. If a word should be changed in some way, it will be colored yellow. Unnecessary words will be colored red. In this case the word 'how' is written yellow and in the expression under "CORRECTION : ", it is suggested that the word should be the four-letter word beginning 'w'.

## II Data Structure of "Phrasing Mechanism"

When an author writes as a correct answer, say,

```
+<=S He><=V showed><IO the child><=DO (which bus)(=Inf to take)> ;
```

it is stored in a tree similar to "Directory Structure" for a file system.




Each node which is called 'phrase' in my source code has the following items in it

```
typedef struct Phrase  {
  char Category[LEN_CATEGORY+1];//Containing a category word
  char Healthy;//Flag for the healthiness. If this phrase is correct to some extent,
              //TRUE ==1 else FALSE ==0
  char Check;//The degree of correctness in this phrase 0-100
  char Open;//Flag for whether this phrase is closed or not TRUE or FALSE
  struct Phrase *Parent;//Pointer to parent node. If this is NULL,
                        //this phrase is the root
  struct Phrase **Child ;//Array of the pointers to the child nodes. If this is NULL
                        //this phrase is a leaf (terminal node)
  struct {//Container for a word. Only for a leaf
        char *Target;//pointer to the word in the correct answer
        char *Answer;//pointer to the modified word in the student's answer
           }w;
                          }PHRASE;
```

Phrases are divided into the two groups -- terminals (leaves) and non-terminals. A terminal phrase represents each word in a correct answer. It always contains one word in the correct answer and cannot have child phrases. It contains a category word if an author write '=.... ' immediately after the separators '('or'<'. There is no difference between the two separators when they contain more than two words but the case where there is only one word in them makes a slight difference. In that case, '<=... .....>'is '(=... (........))'. For example, '<=S He>'equals '(=S (He))' which makes a non-terminal parent phrase phrase with its child terminal phrase. A non-terminal phrase never has a word in it and must have its child phrase(s). It may have a category word as in the case of terminal phrases. Non terminal phrase provides a checking unit for its child terminal phrases each of which represents a word in the correct answer. The whole of a correct answer becomes a root phrase. If there are no partial phrases, all the words in it will be non-terminal phrases without a category word.

When the "Phrasing Mechanism" receives the student's answer, it tries to establish one by one correspondence between words in the correct answer and in the student's answer. The correspondence is made first by the similarity of the form of each word. If an author uses a reserved category word, like '<=Pre with>', every word of the category gets some marks of similarity. So in this case if a student's answer contains the word 'in', it will have a larger probability to match the word 'with'. If an author wants this function to work but doesn't want to make a category word displayed, he/she can write the first letter of the word in lower case. All the category words with the first lower-case letter are not displayed in hints. All the reserved category words are so called 'functional words' which can be enumerated. The list of reserved category words will be shown at the end of this paper.

Secondly the mechanism adjusts the correspondence by checking a word order in each non-terminal phrase. After the correspondence has been finally established, it evaluates the correct answer in terms of similarity to the student's answer both in vocabularies and word order. It repeats this for all the trees of the anticipated correct answer. When the most similar correct answer has been selected, it stores in the tree of the selected correct answer, information about what part of the phrase should be displayed as hints or not. The description of the detailed algorithm for these checking will be too long to be mentioned here. What I have learned after many trials is that there is no best algorithm for it. Each has its own merit and demerit. The most important and difficult problem is when to display and hide words and category words, responding to a variety of student's answer. Improvement will be only possible through its application to English classes.

In the latest version of the mechanism, making a phrase in a correct answer functions as follows. When a student types the phrase correctly, he/she will see the actual words in the phrase at the right place, wherever he/she may type it in his/her answer so that he/she may guess what words should be in other part. If a student types a wrong phrase and the phrase has a category word, he/she will see the category word and '^' representing a word, like '......... M(Pre [^] a^ ^) .... 'or '........ S (^ ^b_y) .. ... 'at the right place of the phrase. Whether the actual words are displayed or not depends on the degree of the completeness of the phrase typed. To make more phrases means to make the answer easier. How many and what phrases should be made is left to an author.

I am an English teacher and if there is something about the mechanism that I can be proud of, it is the fact that I need the mechanism for my class and to some extent it satisfies me.

```
//List of Reserved Category Word
//********Pronoun********
static char *Pro1[] = {"I", "you", "he", "she", "we", "they", "it", NULL};
static char *Pro2[] = {"me", "you", "him", "her", "us", "them", "it", NULL};
static char *Pro3[] = {"my", "your", "his", "her", "our", "their", "its", NULL};
static char *Pro4[] = {"mine", "yours", "his", "hers", "ours", "theirs", "its", NULL};
static char *Pro5[] = {"myself", "yourself", "himself", "herself", "ourselves", "themselves", "itself",
                       "yourselves", NULL};
//********Auxiary Verb*********
static char *Aul1[] = {"will", "won't", "shall", "can", "can't", "cannot", "may", "must", "mustn't",
                       "would", "wouldn't", "should", "shouldn't", "could", "couldn't", "might",
                       NULL};
static char *Aul2[] = {"do", "don't", "does", "doesn't", "did", "didn't", NULL};
static char *Aul3[] = {"be", "is", "am", "are", "was", "were", "been", "have", "has", "had", NULL};
//********Conjunctions*********
static char *Con [] = {"when", "while", "as", "till", "until", "before", "after", "since", "once",
                       "because", "if", "though", "although", "lest", "whether", "that", NULL};
//********Prepostions*********
static char *Pre [] = {"at", "in", "on", "into", "over", "above", "under", "below", "up", "down",
                       "around", "round", "about", "by", "beside", "between", "among", "before",
                       "behind", "to", "for", "toward", "towards", "along", "across", "through",
                       "beyond", "from", "of", "till", "until", "with", "without", "within", "after",
                       "since", "during", "against", "off", "except", "but", "as", NULL};
//********Wh************
static char *Wh [] = {"who", "whom", "whose", "why", "when", "what", "which", "where", "how",
                      NULL};
//********Articles***********
static char *Art [] = {"a", "an", "the", NULL};
//*********************************
//********Relative Pronouns and Adverbs***********
static char *Rel [] = {"which", "who", "whom", "whose", "that", "where", "when", "what", "why",
                       "how", NULL};
//*********************************
//********Verb be***********
static char *Be  [] = {"be", "is", "am", "are", "was", "were", "been", "being", NULL};
```