ED 383 202	FL 023 001
AUTHOR	Hart, Robert S.
TITLE	Errata: Response Analysis and Error Diagnosis Tools.
INSTITUTION	Illinois Univ., Urbana. Language Learning Lab.
REPORT NO	LLL-TR-T-23-94
PUB DATE	Dec 94
NOTE	114p.
PUB TYPE	Guides - Non-Classroom Use (055)
EDRS PRICE	MF01/PC05 Plus Postage.
DESCRIPTORS	Authoring Aids (Programming); Comparative Analysis; *Computer Software; Data Processing; Discourse Analysis; *Error Analysis (Language); Error Patterns; *Hypermedia; *Item Analysis; Programming
IDENTIFIERS	*ERRATA (Hyper Card)

ABSTRACT

This guide to ERRATA, a set of HyperCard-based tools for response analysis and error diagnosis in language testing, is intended as a user manual and general reference and designed to be used with the software (not included here). It has three parts. The first is a brief survey of computational techniques available for dealing with student test responses, including: editing markup that identifies spelling, capitalization, and accent errors and extra, missing, or out-of-order words; pattern matching for rapid identification of specific grammatical errors, keyword searches, and easy specification of alternate answers; and error-tolerant parsing, which puts error diagnosis under control of a grammar and dictionary of the target language. The second section is a user's manual and tutorial guide, describing ERRATA and offering examples of its use. Section three is a reference manual useful to anyone with unusual analysis requirements or wanting to tailor-make responses analyses. Installation and technical information is also included, and complete program code is appended. (MSE)

Language Learning Laboratory
College of Liberal Arts and Sciences

LLL

Technical Report No. LLL-T-23-94
December 1994

University of Illinois
at Urbana-Champaign

ERRATA: RESPONSE ANALYSIS AND ERROR DIAGNOSIS TOOLS

Robert S. Hart

Technical Report No. LLL-T-23-94
December 1994

ERRATA: RESPONSE ANALYSIS AND ERROR DIAGNOSIS TOOLS

Robert S. Hart

ABSTRACT

ERRATA is a set of HyperCard-based tools for response analysis and error diagnosis. Several analysis capabilities are provided: A markup analysis compares a student's answer to several author specified correct or wrong answers, determines the best matching alternative, and generates a graphical error markup that indicates spelling, capitalization and accent errors, extra or missing words, and out-of-order words. A regular expression matcher allows specification of keywords or other strings which should be present in the response; complex patterns can be specified by combining simple strings with AND, OR, NOT, and WILDCARD operators. A simple number analyzer permits specifications of numerical point values and ranges. These various types of analyses may be combined into sequences. Feedback text and HyperCard code may be attached to each answer specificatcin, to be executed if that specification is satisfied. Utility routines support substitution and other kinds of simple string manipulations. *ERRATA*'s primitive functions can be combined to support tailor-made response analyses as complex as partial parsing. Although not intended as a complete authoring system, *ERRATA* also aids with item presentation by providing tools for retrieving and displaying blocks of styled text, and can be used as a self-contained context for authoring drill activities. This document gives a tutorial intorduction to *ERRATA*, and documents and lists all HyperTalk scripts. It is designed to accompany and serve as a reference for the *ERRATA* software.

KEYWORDS: software tools, HyperCard, HyperTalk, XCMD, XFCN, CAI, CALL, response analysis, error diagnosis, response judging, markup, matching, error feedback, authoring system, spelling, word order

LANGUAGE LEARNING LABORATORY
College of Liberal Arts and Sciences
University of Illinois at Urbana-Champaign

Technical Report No. LLL-T-23-94

ERRATA: RESPONSE ANALYSIS AND ERROR DIAGNOSIS TOOLS

Robert S. Hart

Associate Director, Language Learning Laboratory
Assistant Professor of Humanities

December 1994

Available from: Language Learning Laboratory, University of Illinois at Urbana-Champaign, G70 Foreign Languages Building, 707 S. Mathews St, Urbana, IL 61801 (217)-333-9776

TABLE OF CONTENTS

PREFACE

Although the presentational sophistication of CAI (computer-assisted instruction) and CALL (computer-assisted language learning) has greatly increased in recent years as a result of graphic user interface and multimedia capabilities, most courseware continues to use primitive response analysis. This is unfortunate, because nearly any learning activity can be made more "intelligent" by incorporating careful response analysis and error diagnosis. Part of the problem has been the lack of effective software tools for doing the rather specialized operations required. *ERRATA*, a software package which facilitates response analysis within HyperCard™, was created to fill this need. *ERRATA* was designed with special attention to the needs of CALL but will be generally useful for all kinds of instruction. This document provides both a user's manual and an exhaustive technical description of *ERRATA*. The reader is assumed to have a basic knowledge of HyperCard and HyperTalk™ scripting.

Section 1 is introductory. It briefly surveys a variety of computational techniques available for dealing with students' responses, including editing markup, which identifies spelling, capitalization and accent errors as well as extra, missing, or out-of-order words in a student's response; pattern-matching, which allows the quick identification of specific grammar errors, keyword searches, and easy specification of alternate answers; and error-tolerant parsing, which puts error diagnosis under control of a grammar and dictionary of the target language.

Section 2 is a user's manual. It describes *ERRATA*, a pair of stacks which implement many of these techniques for the HyperCard environment and gives examples of how *ERRATA* can be used. These examples are accompanied by on-line implementations in the *ERRATA EXAMPLES* and readers may find it useful to study the on-line material as they proceed. This section provides all the information necessary for straightforward uses of the analysis tools.

Section 3 is a reference manual for the handlers and global variables used by *ERRATA* It should be useful to anyone who has unusual analysis requirements or wants to utilize the primitive analysis routines to construct tailor-made response analyses. Many of the technical aspects of *ERRATA* are described only in this section, so it should be read throughly by anyone wanting an understanding of all *ERRATA's* features.

Complete program code for all HyperCard handlers and XFCNs is appended.

ERRATA is freeware. The author retains full copyright, but you may use the *ERRATA* software as a component of commercial or non-commercial software, provided that (1) You do not modify the *ERRATA* code in any way; (2) You acknowledge *ERRATA* in any software using it, and when preparing apublished description of work that utilizes such software, and (3) You do not sell the *ERRATA* software by itself.

As freeware, *ERRATA* is offered as is, without any warranty of any kind. However, if you have questions or encounter problems in using the package, please contact

Robert S. Hart, Associate Director
Language Learning Laboratory
University of Illinois at Urbana-Champaign
G-70 Foreign Languages Building
707 S. Mathews Ave.
Urbana, IL 61801

voice 217)-333-9776
fax (217)-244-0190
email hart@ux1.cso.uiuc.edu

ERRATA uses the **findInField()** XFCN fror ·`e very useful freeware stack DARTMOUTH XCMDs, Version 3.4.

An early version of the software and some of the written material was presented at "Response Analysis and Error Diagnosis", a workshop given at the Computers and Applied Linguistics Conference, Ames, IA, in July 1994; I would like to thank partici· ·nts in that workshop for their useful feedback. I also wish to express my gratitude to Ms Toshiko Sakurai for her invaluable cooperation during the development of *ERRATA*.

A BRIEF OVERVIEW OF RESPONSE ANALYSIS

Response analysis is concerned with establishing the properties of constructed responses, that is, some text that the user has typed in. Sometimes these properties are used to form a judgment or evaluation of the response (right/wrong, acceptable/unacceptable, satisfactory/unsatisfactory). However, response analysis need not be limited to establishing correctness. When the response is wrong, the properties of the response can be examined to see what kind of errors are present (error analysis) and give appropriate error feedback. And when the response is correct, it can be examined to see what kinds of capabilities were exercised by the student who constructed it (e.g., what grammar rules were employed).

Computationally, identifying properties is done by hypothesizing characteristics (patterns) which might be present in the response; for example, containing (or not containing) a verb inflection error, containing (or not) an uncapitalized German noun, as containing the root "chat" followed by the root "blanc", etc. The response is then examined to see if these characteristics are actually present. The general term for this process of comparison is matching. Defined as above, response analysis always involves matching. Even advanced parsing techniques can be conceptualized as a kind of matching. A parser, however, does not have a pre-stored set of patterns, but generates them as it goes by using a dictionary, morphology, and a set of grammar rules. Consequently, effective response analysis is largely a matter of devising and applying effective matching tools. Table 1 shows various matching techniques which have been applied within CAI. The leftmost column briefly indicates how the pattern is specified. The middle column suggests how the pattern is matched to the response. The rightmost column cites examples of software which use the technique.

Exact matching usually focuses on suitability and is designed for the case when there is exactly one suitable response. However, it is really too inflexible for judging anything but single letter responses, and even there, a leading or trailing space will cause a mismatch. Nevertheless, its negligible programming effort makes it popular in CAI, often in situations where it is unsuitable. To a large extent, improving response analysis means breaking the addiction to exact matching.

Optimized "closest match" for spelling and word order is appropriate when there is a "canonical" correct answer. Defined in terms of this canonical answer are a set of correct and "slightly wrong" responses. Correct responses differ from the canonical answer in trivial ways (extra spaces or extraneous punctuation). Slightly wrong responses vary from the canonical answer somewhat more, because of capitalization errors, accent errors, spelling errors (insertion, deletion, substitution, or transposition of single letters), or word order errors (extra, missing, or out-of-order words). The degree of difference is determined by a complex matching process. Perfect responses are labelled OK, responses which cannot be matched to the canonical answer are labelled as NO without any markup, and deviations from the canonical answer are represented by means of a set of graphic markup symbols. Since errors are determined in terms of insertion, omission, substitution, and movement of letters and words, the matching process is completely language independent. This is an advantage, since a spelling/syntax matcher will work as well with Swahili or Hebrew as with English. It is also a disadvantage, because information about errors is formulated in terms of edit operations rather than more instructionally meaningful grammar rules.

Table 1

RESPONSE SET	ANALYSIS TECHNIQUE	EXAMPLES
Unique string	Exact match	Ubiquitous
Response must "almost" match a small set of "correct" strings. Spacing can vary Punctuation can vary Capitalization can vary Accentuation can vary Spelling can vary Word order can vary Synonymous words allowed Ignorable words allowed	Optimized "closest match" for spelling/word order, usually with graphic error markup.	TUTOR, TenCORE DASHER
Word or string must be present(key words) Word or string must be absent (NOT) Specific word or string sequences (WILDCARD) Several strings all present at once (AND) Any one of several strings present (OR)	Regular expression pattern matching Exact match of pattern required but pattern may be quite abstract.	PILOT, SNOBOL
Morphology obeys TL rules Syntax obeys TL rules	Morphology analysis Finite state transducers, Syntactic parsing	Finite state transducers, KIMMO, BRIDGE, ALICE, ATHENA

Often it is too restrictive to assume that only the canonical answer is correct. Hence, for the sake of flexibility, the lesson author may be allowed to indicate synonymous words and ignorable words when specifying the canonical answer, and otherwise control what will be counted as a "serious" deviation. In addition, it is often the case that there will be several correct answers dissimilar enough that they cannot be considered as editorial variations of one another. Sometimes, a small set of wrong answers can also be anticipated as likely to occur. The existence of multiple answers requires a generalization analysis which will first pick out the best fit from the set of correct and wrong answers and give editorial markup based on that best fitting answer.

Regular expression pattern matching is a way of determining whether certain combinations of words or strings are present (or absent) in a response. The most familiar case of this is probably keyword matching, where the suitability of an answer is determined by looking to see if certain keywords are present. In native language CAI this approach often works well, because one is after the basic meaning of the response and can assume that there are no grammar errors present or else that they are unimportant. Most instructional designs in CALL will not allow major grammar errors to go unnoted, however, and this means that one cannot base CALL response evaluation solely on keyword matching.

Nevertheless, pattern matching can be a powerful technique for determining the nature of word order, agreement, inflection, and dependency errors. This is most easily illustrated by example. Suppose that a correct answer contains "un grande chat blanc", while the student's response contains "un blanc chat grande".

Finding the root "chat" in the response followed by the root "grand" establishes the existence of an adj/noun order error, and finding "grande" following the root "chat" established an adj/noun gender agreement error. Finding both the roots "blanc" and "grand" present (in either order) indicates that the correct adjective vocabulary was used. Simple pattern matchers typically allow patterns to be built out of literal values, the WILDCARD operator "*" which basically means "precedes", and the logical operators AND, OR, and NOT.

Finally, error analysis can be performed by syntactic parsing, which requires as essential components a set of grammar rules, a morphology analyzer, and a dictionary. For our purposes, it is appropriate to think of a syntactic parser as a matcher for determining whether a response matches any one of a very large set of pattern sentences -- namely, all the grammatical sentences in the target language. Because the set of "correct" answers is pre-specified in this way, the parser does not need a canonical sentence against which to match the response. The grammar rules, dictionary, and morphology cooperate to specify in a rather abstract way the responses which are acceptable. Error-tolerant parsing accepts not only correct sentences of the target language, but also sentences which are deviant in certain specified ways.

If the parser succeeds in matching the student's response to a grammatical sentence, it outputs a description of the sentence in the form of a parse tree, that is, a labelled bracketing of the sentence where each word is assigned a part of speech and each phrase is assigned a phrase structure label. Nowadays, it is usual to have the parser return more information than this: each word or phrase is labelled not only for primary category, but also for features such as person, number, gender, case, tense, mood, etc. An error-tolerant parser will usually be expected to label deviant structures with a description of the kind of rule violation involved.

All these kinds of analyses can be implemented to varying extents using the tools in *ERRATA*; the remainder of this document shows how this can be done.

A QUICK INTRODUCTION TO *ERRATA*

Using *ERRATA* involves two steps: (1) designing a response analysis, and (2) implementing the analysis in HyperTalk. Designing the analysis is a matter of instructional design, sound pedagogical principle, and teaching experience. This can be done on paper. It does not require any HyperTalk programming, although *ERRATA* requires that you use a particular notation when you write the analysis specifications. Implementing the analysis you have developed requires a minimum of HyperTalk programming to attach it to the *ERRATA* software.

DESIGNING A RESPONSE ANALYSIS

First, some useful terminology. An P/R interaction is defined to be a situation where the program prompts the student to type in a response at a particular spot and then analyzes that response. If the response fails to be satisfactory, the student is usually required to continue with other responses until producing one that is sastisfactory. In CALL, a P/R interaction is often called an item (not to be confused with a HyperTalk item). Response judging is the process of analyzing a response in order to get an OK/NO judgment and show appropriate feedback. To do response judging, *ERRATA* requires the following:

1. Analysis specifications: *ERRATA* requires you, for each P/R interaction, to specify some correct answers, some anticipated wrong answers, and some patterns or numbers to be matched ("some" may be zero). The collection of specifications for a single response constitutes the analysis for that response, e. g.,

```
:ANSWER cat
:WRONG dog
:MATCH no (cow)
...
```

If the response is "cat", then the judgment will be OK; if the response is "dog", then the judgment will be "no", and if the string "cow" is present somewhere in the response, then the judgment will be NO also. In addition, any spelling or capitalization errors in "cat" or "dog" will be automatically marked with a graphical markup. In the case of a multi-word :ANSWER or :WRONG specification, missing words and out-of-order words would also be marked. A common kind of analysis is a single correct answer, such as

```
:ANSWER cat
```

2. Feedback: For each :ANSWER, :WRONG, or :MATCH alternative, you may specify actions which should be taken if that answer etc. is matched. Usually the action consists of displaying some feedback text to the student but you might want to do some data collection too. The analysis specifications together with their conditional feedback create an IF...THEN..ELSE IF... structure, as in

```
:ANSWER cat
        This is a feline.
        Other examples: tiger, lynx, lion
:WRONG dog
        This is a canine.
:MATCH no (cow)
        This is a bovine.
...
```

which has this meaning:

> **if** the response was "cat", **then** display "This is a feline. Other examples: tiger, lynx, lion.";
> **else if** the response was "dog" **then** display "This is a canine.";
> **else if** the string "cow" occurs in the response, **then** display "This is a bovine.";
> etc.

You are never required to provide feedback for a specification; it is always optional. You can hav several lines of feedback. You do not have to indent the feedback lines, but if you do (as in the example just above), it will make the data easier to read.

3. Help Information: Besides analysis and feedback specifications, *ERRATA* also allows you to specify various categories of "help" information:

```
:HELP vocab
        To read: lire, lis, lissons, lu (4); Last: dernier; Ten: dix; Page: page (f).
:HELP grammar
        Cardinal numeral adjective precedes adjectives such as dernier, premier, etc.
:ANSWER Elle nous a lu les dix dernières pages.
:MATCH no  (pour_nous)
        No preposition needed with pronoun in this context.
```

You may label the :HELP specifications any way you please; in the example above, the two labels "vocabulary" and "grammar" have been used. All the :HELP specifications are ignored during response judging. It is up to your own program to retrieve :HELP information at the appropriate time (e.g., when the student requests a particular type of help) and display it at an appropriate location. The :HELP specifications may be included along with the response analysis because they are part of the data that define how the program will interact with the student during a particular item. *ERRATA* does provides a function **getHelp(<label>)** which retrieves the information associated with a particular :HELP label. (Text retrieved with **getHelp()** should be displayed using **showText**).

☞ For an example of how to handle student-requested help using **getHelp()**, look at the script of the VOCAB HELP button in *ERRATA EXTRAS #* 8 (Complex Drill Design).

The information associated with a :HELP label does not really have to be help; it can be any kind of text which is relevant to the item. For instance,

```
:HELP prompt
        Translate to French: She read the last ten pages for us.
```

could specify the text used to set up the prompt for a particular item. Of course, as explained, your program will have to retrieve and display this prompt.

4. Puting the analysis specification on-line: An analysis specification must, of course, be typed into field somewhere. The question is, where should that field be located? If your analysis contains no styled text (no sized text, no boldface, underline, or italics, and nothing but the default font), then the field which contains it can be located anywhere in the lesson stack. If your feedback contains styled text, however, the analysis field must be on the same card where the student types in her response. (This requirement is imposed because HyperCard makes it very slow and clumsy to moved styled text from one card to another.) If your analysis text is located on another card or in another stack, you can use the **copyText** handler provided by *ERRATA* to copy it into a field on the card which has the response field.

5. Specifying analyses for multiple items: If a number of responses will be typed into the same response field (as is usually the case with drill activities), then each item must have its own analysis. Type all the

analyses into the same field, in order, separating each from the next by a line starting with "#". Below, for instance, are shown the data for a three-item drill:

```
:HELP prompt
        Name a feline:
:ANSWER panther
:ANSWER tiger
:ANSWER housecat
#
:HELP prompt
        Name a canine:
:ANSWER dog
:ANSWER fox
:WRONG bear
#
:HELP prompt
        Name a bovine:
:ANSWER cow
:ANSWER bison
:WRONG sheep
:WRONG goat
```

ERRATA provides handlers for accessing this kind ot data structure in order to get at the data needed for each item; see below for details.

IMPLEMENTING THE RESPONSE ANALYSIS IN YOUR STACK

First you must attach the *ERRATA* handlers and XFCNs by executing a command of this form

```
start using stack "MyHardDisk:MyFolder:ERRATA
start using stack "MyHardDisk:MyFolder:ERRATA EXTRAS
```

Usually, the **openStack** handler is the best place to do this.

☞ The handlers in *ERRATA EXTRAS* are not needed by the response analysis system; if you are not calling them from your own scripts, you do not need to attach this stack.

You must now distinguish these two cases: (A) There is only one response field, but several responses will be typed into it, each with its a different prompt, as is the case with most drill exercises; (B) There are several response fields on the card, each with a single prompt and answer. *ERRATA* requires a slightly different setup for each case.

(A) <u>When there is only one response field which will be used to receive the response for several items, set up this way</u>:

1. Set up the multiple-item analysis data as shown above.

2. Execute these handlers in the **openCard** handler of the card script:

```
resetErrata
setUpMarkUp  <response field name>
setFeedBackField <feedback field name>
```

If your feedback contains styled text (special fonts, bold, italic, or underline, or sized text) *ERRATA* will display the styling by default. If there isn't any styling or don't want to display it,execute

```
setPreserveFormating False
```

which will substantially speed up display operations. **setUpMarkUp** determines the current response field. When the student presses RETURN in this field, judging will be done automatically and OK or NO, any graphical markup, and any answer-contingent feedback will be displayed. **setFeedBackField** determines where the feedback will be displayed.

3. As part of presenting each new item, execute lines like these

```
setCorrectAnswer <analysis field name>, <starting line number>,
<ending line number>
showText getHelp( "prompt" ), <name of prompt field>
```

The start and ending line numbers tell where the analysis for the current item begins and ends within the field. They will depend on the current item number and the format of your analysis data. If you omit the line numbers, the entire contents of the analysis field will be used.

A call to the XFCN **delimiterTable("linePointers", <analysis field name>, "#")** returns a list of the line numbers which start with a "#" character. If you have separated the analyses for different items using lines which begin with "#", then the analysis for the Nth item will start at line number (item N of list + 1) of the analysis field and end at line number (item N + 1 of list - 1). Get and store this table in a global before beginning the drill. Each time you present a new item, access the proper table entries to get the line number parameters for **setCorrectAnswer**.

☞ For an example of a drill activity of this sort which does preserve styled text, examine *ERRATA EXAMPLES* # 8 (Complex Drill Design).

If you have turned off styled display by executing **setPreserveFormating False** then you can specify the analysis using an expression rather than a field name, for example

```
setCorrectAnswer  item 3 of myVar, 5, 7
```

Since the value returned by a HyperTalk expression is stripped of text styling, an expression such as **myVar** or **item 3 of myVar** cannot be used when styled display is turned on.

☞ For an example of this method of specifying an analysis, see *ERRATA EXAMPLES*# 5 (A Simple Drill Design).

(B) If there are several response fields on the same card, each with its own prompt and answer, set up this way:

1. Execute these handlers in the **openCard** handler of the card script:

```
resetErrata
setFeedBackField <feedback field name>
```

The **resetErrata** handler simply makes sure that, in case *ERRATA* has been used previously, all default values have been returned to normal. If you feedback styled text, or don't want to display the styling, you can speed up display by executing

```
setPreserveFormating False
```

2. In the script of each response field, install an **openField** handler with these lines:

```
on openField
       activateField  <analysis field name>, <start line number>, ¬
           <end line number>
end openField
```

where the starting and ending lines have the same meaning as above. If the line numbers are omitted, the entire contents of the analysis field will be used. If the analysis consists of a single correct answer with no feedback, you may alternatively specify a string instead of a field name

```
activateField  <correct answer string>
```

for example, **activateField "cat"**, which is the same as specifying :ANSWER cat as the entire analysis.

When the student clicks on the field whose script contains this handler, that field becomes the current response field. When the student presses RETURN in that field, response judging is initiated using that field's contents as the response and whatever the **activateField** handler specified as the analysis. OK or NO, any graphical markup, and feedback will be displayed. Clicking on some other field which has an **activateField** handler will make that field the current response field. Thus, the student can move around from one response field to another.

☞ *ERRATA EXAMPLES* # 4 (Multiple Response Fields) shows how this is done.

ERRATA USER MANUAL

The matching tools in *ERRATA* are built on two HyperCard XFCNs (external functions): MARKUP and MATCH.

> The MARKUP XFCN is designed for situations where there is a small set of correct and (optional) incorrect answers. It does two things: it finds out whether an entire response is "close enough", overall, to one of the answers and, if so, it returns information about the editorial changes needed to make the response perfectly correct. This editorial information can be shown to the student as a graphical markup, or used by the program.
>
> The MATCH XFCN is designed to establish whether the response has (or does not have) certain properties. These properties must be expressed in terms of the presence or absence of words or other strings, or sequences of words or strings.

MARKUP and MATCH can be called directly from HyperTalk. But to be really convenient for instructional purposes, they must be combined with programming which accesses data, controls display, and does a variety of other "bookkeeping" tasks. Hence an important part of the overall response analysis system consists of a set of routines which take care of these tasks without requiring the detailed attention of the lesson author. In *ERRATA*, these routines are implemented by HyperTalk handlers and functions, but analogous functions could easily be written in IBM ToolBook™ or in general-purpose languages such as PASCAL or C. These routines have been designed for easy, clear, and consistent interface to the primitive XFCNs. Hence scripting will normally be easier if you do not access the XFCNs directly. Instead you should always access them though the HyperTalk handlers which have been provided. Table 2 represents the general scheme.

Boldfacing indicates the kernel of essential routines required for ordinary response analysis. The core pattern matching handler in *ERRATA* is the **judgeResponse** handler, which combines the capabilities of all the other pattern matching tools, calling on MARKUP, MATCH and other handlers as needed to create a flexible pattern matching capability.

When response analysis is done by **judgeResponse**, you must specify two things: the response that is to be analyzed and the analysis text (the set of answers and other patterns that will be used to analyze the response). Often the instructional design also requires specifying feedback that will be displayed if a particular pattern matches (or fails to match) the response. Sometimes it is useful to specify various types of help which will be shown to the student on request. Although not strictly a part of response analysis, help can be treated much in the same way as feedback, so it is supported by the response analysis system. Finally, you may want special evaluation message to replace the default "OK" and "NO". In brief, these are the things which must be specified to set up a response analysis:

> Analysis Specifications: What patterns to use and where they are stored.
> Response: Where to get it.
> Feedback: What to show if the response matches a pattern, and where to show it.
> Help: What to show when a given type of help is requested
> Evaluation Messages: What to display to indicate a correct/incorrect response.

Table 2

Your Application		***ERRATA***		
		Response Analysis Interface Handlers		Primitive XFCNs
		activateField		
		setUpMarkUp		
		setCorrectAnswer		
		setFeedBackField		
		resetErrata		
		changeMarkUpSymbol		Match XFCN
Your own handlers	<==>	restoreDefaultMarkUpSymbols	<==>	
		setJudgingParams		Markup XFCN
		restoreDefaultJudgingParams		
		restoreMarkUpDefaults		
		judgeResponse		
		markupUsingParams		
		setJudgingHandler		

The most complex step is specifying the patterns which are to be matched, and that is what we will discuss first.

judgeResponse needs to be given an analysis text. This text is a sequence of (one or more) analysis specifications which tells **judgeResponse** how to carry out the analysis that you want done. Each individual analysis specification indicates the type of matching to do, the polarity of the match (whether a successful match indicates a correct or incorrect response), and (optionally) what feedback should be given to the student if the match succeeds. **judgeResponse** recognizes six different types of analysis specification:

:ANSWER	(markup analysis and judgment)
:WRONG	(markup analysis and judgment)
:MATCH	(pattern matching analysis and judgment)
:NUMBER	(number analysis and judgment)
:DO	(custom-made user analysis)

:ANSWER and :WRONG, between them, enable **judgeResponse** to perform a markup analysis which will determine a right/wrong evaluation and mark minor errors with editorial symbols, :MATCH enables a general pattern matching analysis helpful for error diagnosis, and :NUMBER takes care of situations where numbers need to be judged. :DO allows the execution of additional user-written analysis routines. Note that :HELP, which specifies what the user will see when different types of help are requested and allows specification of blocks of styled text, is ignored by **judgeResponse**.

The analysis text can be any length. A minimal analysis might consist of a single :ANSWER specification without any feedback. A complex one might contain dozens of different patterns to be examined and a great deal of feedback text. Analysis text can be specified either by providing the name of the field that contains it, or as an expression (i.e., a literal string, variable, or chunk expression).

The analysis text is a bit like a HyperTalk script, and the specifications are something like HyperTalk commands. However, you must distinguish carefully between the specifications and true HyperTalk commands. Specifications are not executed by HyperCard directly; they are interpreted and executed by the **judgeResponse** handler, which uses them to do matching operations.

So that error analyses can be tailored to the lesson author's needs, each kind of specification has its own syntax, which you must learn before using the analysis tools. The following sections describe this syntax.

SPECIFYING A MARKUP ANALYSIS

:ANSWER and :WRONG specifications instruct **judgeResponse** to do a markup analysis. The syntax for specifying the pattern to be matched is the same for both :ANSWER and :WRONG, as shown in these examples:

> :ANSWER Le garçon voit le chien blanc
>
> :WRONG La jeune fille voit le chat noir
>
> :ANSWER The [quick fast speedy] brown fox jumped over the [lazy lethargic] dog
>
> :WRONG <the a> big vulture flew over [sleeping resting] aardvark

A correct answer is indicted by simply writing the label :ANSWER followed by the actual correct answer. An anticipated wrong answer is indicated by writing the label :WRONG followed by the anticipated wrong answer. You must remember several things about the format of these specifications:

☞ Like most Macintosh names, :ANSWER, :WRONG, and other analysis specification labels are indifferent to case: :ANSWER, :ANSWER, :ANSWER, :ANSWER are all equivalent.

The :ANSWER or :WRONG label must be on the same HyperCard line as the answer string and separated from it by at least one space.

The :ANSWER or :WRONG does not need to begin on the first character of the line. However, the label must have ":" as its first character; ": ANSWER" will not work.

Styling (in particular, font) is ignored by **judgeResponse** when it uses the specifications. So it doesn't matter if :ANSWER or :WRONG strings or :MATCH patterns are styled or not. But you may find cyrillic or Hebrew easier to read if you hand-set the font properly.

The answer string must not contain any punctuation marks, since punctuation marks are ignored when computing a markup. The default set of punctuation marks is

SPACE, RETURN, and . , ; : ( ) [ ] < > ? !

If you need to analyze punctuation, it is possible to respecify which characters are to be considered punctuation marks.

Synonymous words can be indicated by putting the list of words in square brackets []. Any one of the synonyms will be acceptable at that point in the sentence. (There is no way to make a phrase of two or more words synonymous with a single word, however.) Hence, if the :ANSWER specification is

> :ANSWER The [quick fast speedy] brown fox jumped over the [lazy lethargic] dog

then both these responses will be matched and judged correct:

> The quick brown fox jumped over the lethargic dog.
> The fast brown fox jumped over the lazy dog.

Ignorable words are indicated by a list of words in angle brackets <>. All such words are removed from the response before judging, so they must not appear elsewhere in the answer (outside of the ignorable word list). It is most efficient if there is just one list of ignorable words which appears at the front of the answer string. Since ignorable words are ignored everywhere, an answer specification like

```
:ANSWER <the a> big vulture flew over [sleeping resting] aardvark
```

will accept this response:

```
A the big vulture the flew a over resting the aardvark a the the.
```

Hence, some care must be exercised in using ignorable words for CALL purposes.

When the answer to the prompt is unambiguous, it may suffice to use a single :ANSWER specification without feedback as, for example, with the following prompt and :ANSWER specification:

```
Change to past tense: John is doing his homework.
_________________________________
:ANSWER John was doing his homework
```

This is perhaps the most common case. Often, however, several correct responses are possible:

```
Change to a question: John came yesterday.
_________________________________
:ANSWER Who came yesterday
:ANSWER When did John come
:ANSWER Did John come yesterday
```

In other cases, several responses are likely to occur, but some of them are wrong:

```
Change to a yes/no question: John came yesterday.
_________________________________
:ANSWER Did John come yesterday
:ANSWER Was it yesterday that John came
:WRONG Who came yesterday
:WRONG When did John come
```

This last example points up the fact that different responses will require different feedback. Since feedback depends on which answer was matched, it is convenient for the lesson author to put any feedback text just after the :ANSWER or :WRONG specification:

```
Change to a yes/no question: John came yesterday.
_________________________________
:ANSWER Did John come yesterday
        Right, this can be answered "Yes" or "No".
:ANSWER Was it yesterday that John came
        Right, this can be answered "Yes it was."
:WRONG Who came yesterday
        This is not a Yes/No question, since
        the proper reply to it is "John."
:WRONG When did John come
        This is not a Yes/No question, since
        the proper reply to it is "Yesterday."
```

☞ In these examples the feedback lines are indented a few spaces to make the analysis more readable, but this is not necessary. (If indenting is present it will, however, be displayed as part of the feedback.)

Text styling in the feedback, such as the underlining and bold italics in the example, will be preserved in the display unless you execute **setPreserveFormating False**.

Everything after an :ANSWER/:WRONG specification, all the way to the next specification (or the end), serves as feedback to display if the response matches the specification. Feedback is never required. It can always be omitted if it seems instructionally extraneous (an OK or NO message is automatically generated by the markup analysis in any case):

```
Change to a yes/no question:  John came yesterday.
______________________________________
:ANSWER  Did John come yesterday
:ANSWER  Was it yesterday that John came
:WRONG  Who came yesterday
          This is not a Yes/No question, since
          the proper reply to it is "John."
:WRONG  When did John come
          This is not a Yes/No question, since
          the proper reply to it is "Yesterday."
```

When there is a sequence of :ANSWER/:WRONG specifications, as in the examples above, the markup analysis compares the response to each of the :ANSWER/:WRONG specifications. The one which gives the closest fit to the response determines the judgment (OK or NO) and the feedback.

The special tag ":?" indicates an unanticipated response, and can be used with either :ANSWER or :WRONG. If no other :ANSWER or :WRONG has matched the response so far, then :ANSWER :? or :WRONG :? matches immediately with a perfect fit (regardless of what the response actually was), the associated feedback is fetched, and the response analysis terminates. The purpose of ":?" is to specify what to do with a response that the analysis couldn't recognize at all, so it is only meaningful to have one such specification. Use :ANSWER :? if you want an unanticipated response judged OK, or :WRONG :? if you want it judged NO. ":?" specifications should always come last, so that all the other :ANSWER and :WRONG tags have a chance to match first. (This is an exception to the general rule that the relative order of :ANSWER and :WRONG specifications is irrelevant.) Here is an example which will judge NO:

```
Change to a yes/no question:  John came yesterday.
______________________________________
:ANSWER  Did John come yesterday
:ANSWER  Was it yesterday that John came
:WRONG  Who came yesterday
          This is not a Yes/No question, since
          the proper reply to it is "John."
:WRONG  When did John come
          This is not a Yes/No question, since
          the proper reply to it is "Yesterday."
:WRONG  :?
          I can't understand what you typed in.
```

Note that using ":?" will cut off any :MATCH, :NUMBER or :DO specifications which come after, so ":?" should be placed after these too.

Sometimes it is instructionally desirable that feedback and :HELP consist not only of text display, but of other display operations, such as playing sounds or showing pictures or movies, or executing still other HyperTalk commands (e.g., incrementing counters). *ERRATA* provides a simple facility for this. Any feedback line beginning with the double right-arrow "»" is assumed to contain a HyperTalk command, and will be executed rather than displayed. For example:

```
:HELP prompt
        » picture "aardvark"
        What is the animal in this picture?
:ANSWER <a an> aardvark
        » play "okMessage"
        That is correct - an African mammal!
:WRONG <a> penguin
        » play "noMessage"
        » movie "penguin"
        No, here is a movie of some penguins in action.  See the difference?
```

The HyperTalk commands must preceed any text that you want displayed. This provides some capability for multimedia display and response-contingent execution, but the parameters of handlers such as play or picture must be constants -- they cannot be variables.

☞ *ERRATA EXAMPLES* # 11 (Multimedia) uses this feature.

SPECIFYING A PATTERN-MATCHING ANALYSIS

Designing a pattern-matching analysis is usually more complicated than designing a markup analysis. This is because you are describing particular pattern(s) that may be present or absent in the response, and not the whole of a (correct or incorrect) response. Each pattern specification consists of the word :MATCH, a polarity specification, and a pattern descriptor. This is shown by the following example where the prompt asks the student to translate a French noun phrase.

```
Translate to French:  the dog and the cow
_________________________________________
:MATCH ok (_et_)
        This is a conjunctive noun phrase.
:MATCH okStop (le_chien_et_la_vache)
        Right!
:MATCH noStop -(_chien_)
        Wrong word for "dog".
:MATCH noStop -(_vache_)
        Wrong word for "cow".
:MATCH no (_la_chien_)
        "Chien" is masculine.
:MATCH no (_le_vache_)
        "Vache" is feminine.
```

To understand the syntax used to specify patterns, look at the second pattern, "le_chien_et_la_vache". Note that the pattern must always be enclosed in parentheses. The underscore character "_" is used to indicate that a space should be present at that point. It is completely equivalent to write the pattern using spaces rather than underscores:

```
:MATCH okStop (le chien et la vache)
```

but underscores make the pattern more readable. A negation sign "-" preceding a pattern indicates that a match will occur only if the pattern in parentheses is absent in the response. So for instance

```
-(_vache_)
```

is a pattern that will be matched if the word "vache" is not present anywhere in the response string.

The words "okStop", "noStop", "ok" and "no", which immediately follow the :MATCH tag, are polarity specifiers which specify what evaluation should be made if the associated pattern is matched. The specifiers "ok" and "okStop" indicate that the pattern is correct or appropriate, whereas "no" and "noStop" indicate that the pattern is an error or inappropriate.

Normally, **judgeResponse** goes thorough the sequence of pattern specifications one at a time, from first to last. However, if it a pattern with polarity "okStop" or "noStop" is matched, then no more specifications will be processed and the analysis will simply stop at that point. This gives the lesson author some minimal capability to execute analysis specifications conditionally. For instance, in the example sequence above,

```
:MATCH okStop (le_chien_et_la_vache)
        Right!
```

specifies the correct answer. If it is matched, there is no point in continuing to execute the remaining specifications, because they all look for various kinds of errors, and there won't be any. Hence "okStop" is the proper polarity.

As **judgeResponse** goes through the sequence of analysis specifications, it keeps track of which patterns have been matched. If at least one "ok" or "okStop" pattern has been matched, and zero "no" or "noStop" patterns have been matched, then **judgeResponse** concludes that the response as a whole is OK. If, however, one or more "no" or "noStop" patterns have been matched, **judgeResponse** concludes that the response has problems and judges the response as a whole to be wrong, regardless of the number of matches which were ok. If the response is judged to be ok, all of the feedback from the "ok" and "okStop" patterns which were matched is shown; otherwise, all of the feedback from matched "no" and "noStop" patterns is displayed. (This logic for interpreting :MATCHes can be modified with the **matchingIsOk** handler.)

Some additional notations are allowed within pattern descriptors that permit the lesson author to specify more abstract patterns without too much effort. They use these special characters:

-	Negation (NOT)
|	Disjunction (OR)
^	Conjunction (AND)
*	Wildcard (BEFORE...AFTER)
&	Ending
?	Any character

Negation has been illustrated in the examples above. The disjunction (alternate appearance) of several patterns is specified with "|". For the prompt below, the following analysis sequence is appropriate:

```
Name a feline:
_______________________________

:MATCH okStop ((cat) | (lion) | (leopard) | (lynx) | (tiger))
```

For this match to succeed, at least one of "cat" or "lion" or "leopard" or "lynx" or "tiger" must be present in the response. Any number of them may be present, and there may be additional, unmatched material present. Conjunction (simultaneous presence) is specified by "^":

```
List the great lakes:
_______________________________

:MATCH okStop ((Superior) ^ (Huron) ^ (Erie) ^ (Ontario) ^ (Michigan))
```

"Superior" and "Huron" and "Erie" and "Ontario" and "Michigan" must all be present in the response, although they may appear in any order. There may be additionai, unmatched material present also. The NOT, AND and OR operations can be used to build up complex expressions, for example,

```
:MATCH ok (((one) ^ (two)) | ((-(three)) ^ (-(four))))
```

which will match any response in which "one" and "two" occur simultaneously, OR any response in which both "three" and "four" are absent.

☞ The "|" and "^" operators can connect only expressions fully enclosed in parentheses. It is not legal to write (one|two) or (three)^-(four).

The asterisk "*" serves as a "wildcard" character, so that in

```
:MATCH okStop (_ne*pas_)
```

the pattern will be matched just in case both "ne" and "pas" occur in the response, and "ne" occurs to the left of "pas". The "*" stands for any amount of intervening material (possibly none), no matter what it is.

☞ "*" can only appear between strings. It cannot be used between expressions containing OR, AND, or NOT. For example, (((a) | (b)) * c) or (-(a)*(b)) are not a legal pattern descriptors for :MATCH.

The "&" character will match everything up to the next space. This is useful when you want to match the root of a word and don't care what ending the response has. For example, suppose the prompt and correct answer are

Translate this phrase to French: the last ten pages

---

les dix dernières pages

A :MATCH specification which will identify the presence of a word order error which puts the adjective "dernier" before "dix" is

```
:MATCH no (derni&_dix_page&)
```

this specifier will match

derniers dix pages
dernières dix pages
dernier dix page
dernièrs dix pages
dernièr dix page
dernieres dix pages,

etc., so that the word order error will be correctly identified regardless of spelling or inflection errors in the adjective and noun forms. A "?" in a pattern will match one character of the response, no matter what, so

```
:MATCH ok (c?t)
```

will be matched responses containing "cat", "cit," "cét", "cut", "cxt", "cAt", etc. Finally a :MATCH with a polarity but without a pattern will succeed regardless of the response, as in

```
:MATCH okStop (aarvaark)
        Correct!
:MATCH noStop (penguin)
        No, the picture doesn't show a bird!
...
:MATCH noStop
        I don't understand what you typed in.
```

Put this after all other :MATCHs to specify a default "ok" or "no" judgment and feedback if no other :MATCH specification succeeds.

The MATCH XFCN has substantial limitations, most notably the fact that there are no pattern variables, and hence no way to return the material which was matched within a particular context, or to build up very abstract pattern descriptions under program control. Nevertheless, the capabilities of MATCH suffice to form fairly sophisticated error analyses, e.g.:

> Translate to French: She read the last ten pages for us.
>
> ---
>
> :MATCH no (pour_nous)
>   Use indirect object form of PN rather than PREP + PN.
> :MATCH no (_dern& dix page)
>   Numeral adj must precede dernier, etc.
> :MATCH no (pag&_derni)
>   Dernier precedes the noun.
> :MATCH no -(_a_)
>   Passé composé is the appropriate form of past tense.
> :MATCH no -((_a_)^(_lu_))
>   You need a compound verb form.
> :MATCH ok (Elle_)
>   Yes, the subject must be feminine singular.

☞ *ERRATA EXAMPLES* # 9 (Keyword Driven Dialog), # 12 (Multiple choice) and # 15 (Answer/Match Mixed) show various applications of MATCH. Topics # 19 (Simple Parsing), # 20 (Error Statistics) illustrate direct use of the MATCH XFCN. #21 (Response History) shows direct use of the MATCH XFCN within user-customized diagnosis handlers which are :DOne by **judgeResponse**.

SPECIFYING A NUMBER ANALYSIS

The purpose of the :NUMBER analysis is to find out whether a response (which is assumed to be a number but need not be) falls within a given numerical range. Some example of :NUMBER specifications are

```
:NUMBER okStop (3)
:NUMBER ok (15-30)

:NUMBER no (*-0)
:NUMBER noStop (250-*)

:NUMBER ok ¬(15-30)
:NUMBER okStop ¬(15)
```

The first tag following :NUMBER must be a polarity value of "ok", "okStop", "no", or "noStop". It determines how the response will be evaluated if the response fall within the specified number range.

The parentheses surrounding the second tag specify a range of numbers. This may be either a singe number value, in which case the response must exactly equal the value, or a range, in which case the response must fall within the range. An open ended range is indicated by "*", so that (15 - *) means 15 or greater and (* - 20) means 20 or less. A "¬" preceding the parentheses means that the :NUMBER specifier is satisfied if the response does NOT fall within the range. If the response is not a number, the match fails.

For CALL purposes, a facility like this is useful mainly when a student is asked to type in numerical responses to comprehension questions.

☞ :NUMBER has no built-in facility for handling ordinals or the written forms of cardinals, nor does it have any way to specify tolera[illegible] equired precision) relative to a value.

ERRATA EXAMPLES # 9 (Keyword-driven Dialog) and # 14 (Number Analysis) utilize the :NUMBER specifier.

SPECIFYING A CUSTOM-MADE ANALYSIS

The purpose of :DO is to allow the user to execute any handler that she wishes in place of :ANSWER, :WRONG, :MATCH, or :NUMBER. The tag of a :DO specification must contain two space-separated items:

1. A polarity of "ok", "no", "okStop", or "noStop"
2. A HyperTalk message, together with any input parameters needed by that message.

The handler (it cannot be a function) called by the :DO message should return a value of **True** or False using HyperTalk's **return** command. **judgeResponse** will access the returned value via **the result.** If there is no return, the return will be assumed to be **True.**

Like :ANSWER and :MATCH, :DO may have its own contingent material. **judgeResponse** treats :DO as if it were a form of :MATCH. A return of **True** from the :DOne handler is treated as if it were a match; a return of **False** is treated as if it were a non-match. When the return is **True**, the feedback is appended to **okFeedback** if the polarity tag is "ok" or "okStop"; it is appended to **noFeedBack** if the polarity is "no" or "noStop". If the return is **False**, then the feedback is not processed.

The following example uses :DO to implement a modified exact match judging (an expository, not instructional, choice). First we write a handler which performs a modified exact match on a model and a response and returns **True** if the match succeeds:

```
on exactMatch  model, response
       if last char of response = "s" then delete last char of response
       return ( model = response )
end exactMatch
```

Notice that this handler performs a simple-minded de-pluralization of the response before doing the exact match judgment. The response to "What is your favorite kind of pet?" may now be analyzed by an analysis text which :DOes **exactMatch** several times, once for each anticipated answer.

```
:DO okStop exactMatch "cat", theResponse
       Yes, cats are noble creatures.
:DO no exactMatch "dog", theResponse
       Dogs are too servile.
:DO no exactMatch "horse", theResponse
       Horses don't make very good house pets.
```

A more practical example shows how :DO can be used implement flexible judging of numerical expressions:

```
on judgeExpression model, resp
       put subst( resp, "units", empty ) into resp
       return  value( resp ) = model
end judgeExpression
```

Note that the word "units", if present, is removed before evaluation by the **subst()** (string substitution) function which replaces the string "units" by the null string if it is present. If the question is "How long is the hypotenuse of a right triangle with sides of 4 units and 3 units?" then this response analysis:

```
:DO okStop judgeExpression( 5, theResponse )
        Right.  You must have used the Pythagorean theorem.
```

will accept all these responses

```
5
5 units
sqrt( 25 )
sqrt( 16 + 9 ) units
sqrt( 4^2 + 3^2 )
```

or any other arithmetic expression which evaluates to 5 and optionally contains the word "units".

The :DOne handler is called in the context of the responseAnalysis handler, so that any variables which occur within input parameters of the :DOne handler must also be variables which appear within **responseAnalysis.** The most useful of these is the global variable **theResponse**, which contains the current response string. This is the reason that both **exactMatch** and **judgeExpression** above use **theResponse** as the value of the second input parameter.

The :DOne handler message is first sent to the current card and will travel up the HyperTalk message hierarchy in the usual way.

☞ *ERRATA EXAMPLES* # 21 (Response History) shows how :DO can be used to implements some simple error-tolerant parsing machinery.

SPECIFYING HELP

The :HELP specification provides a way to stipulate various kinds of information which would aid the student in successfully providing a response. :HELP information is not really part of the response analysis, and in fact is ignored by **judgeResponse**. It is up to the lesson author to write handlers which will access and display :HELP information, normally at the request of the student.

:HELP specifications have the same syntax as :ANSWER, :WRONG, :MATCH, :NUMBER and :DO, as shown in this example:

Translate to Swedish: Kerstin is going to the movies tonight

:HELP grammar
    Swedish often uses present or future tense where English uses
    present progressive.
:HELP morphology
    att gå is an irregular verb: *gå, går, gick, gått.*
:ANSWER Kerstin går till bio igår kväll.
:ANSWER Kerstin ska gå till bio igår kväll.

The :HELP specification consists of the tag :HELP, followed by a label intended to indicate what type of help text follows. The label can be anything the lesson author likes. Lines following :HELP contain the actual help text, which is ended by the next :ANSWER, :WRONG, :MATCH, :NUMBER, :DO, or :HELP directive, or by the end of the analysis text.

☞ Since :HELP is ignored during the answer judging process, the position of :HELP specifications within the analysis text is irrelevant; they may be interspersed anywhere, but the text will be easier to read if they are collected at the beginning or end.

Because :HELP is ignored during response analysis, *ERRATA* provides a special function **getHelp()** to access help text. It takes one (optional) input parameter, which is the label of the type of help being sought:

```
        getHelp( "grammar" )
        getHelp( "morphology" )
```

getHelp() returns the text associated with the specified type of help. If that type of help cannot be found in the current response analysis text, then it returns **empty**. If the input parameter is not provided or is **empty**, then all the help texts present in the current analysis are concatenated and returned. The output of **getHelp()** will generally be line pointers to the help text rather than the actual text itself; hence, the value returned by **getHelp()** should always be displayed by **feedBack** or **showText** or some user handler which is capable of dereferencing the line numbers.

Here is a simple example of how to make grammar help available to the student on request. Suppose that the current analysis text is that show immediately above, and that there is a card button named "GRAMMAR HELP" which has the following script:

```
        on mouseUp
               feedBack  getHelp( "grammar" )
        end mouseUp
```

When the student clicks on this button, the string

> Swedish often uses present or future tense where English uses
> present progressive.

will appear in the current feedback field.

Notice that all help indicated with the :HELP specification is in effect only while the analysis text which contains it is the current analysis text. Each new item (P/R interaction) will generally bring its own new answers and new help. Help which is general to a whole group of questions or an entire stack must be provided for in some different way.

Actually, a block of text labelled with the :HELP specification does not have to be "real" help text. It can be anything, for instance, a prompt for the item. Suppose, for example, that card field "data" contains this analysis text, which implements a multiple choice item:

```
        :HELP prompt
               According to your reading, what was the main cause of the civil war?
                      a. Slavery
                      b. Economic conflict
                      c. The conflict over Kansas
                      d. Regional cultural differences
        :MATCH okStop (a)
               Right, it dominated even traditional economic differences.
        :MATCH noStop (b)
               It was a contributing cause, but  not the main one.
        :MATCH noStop (c)
               Kansas was a symptom of the slavery conflict.
        :MATCH noStop
               Please type a, b, c or d.
```

Given these data, the following two lines of HyperCard code will display the prompt text in a field named "promptField"

```
setCorrectAnswer the name of card field "data"
showText getHelp( "prompt" ) into card field "promptField"
```

Note that setCorrectAnswer must be executed first, because it specifies the current answer field, and this is the field that **getHelp()** searches to retrieve the text.

☞ *ERRATA EXAMPLES* # 8 (Complex Drill Design) uses :HELP to display a prompt and to specify vocabulary help.

IMPLEMENTING A RESPONSE ANALYSIS

Once you have written the response analysis specifications, implementing the response analysis requires only a few steps:

1. Attach the *ERRATA* stack (**start using stack**)
2. Specify where the analysis text is located (**setCorrectAnswer**)
3. Specify the field the response will come from (**setUpMarkUp**)
4. Specify where feedback text should appear (**setFeedBackField**)
5. Specify where the OK and NO message will come from (**setOkNoField**)

Each of these steps requires a bit more discussion.

1. Attach the *ERRATA* stack. To do this, executing the statement

```
start using stack "ERRATA"
```

If the response analysis stack is not in the HyperCard default path, specify the full pathname of the response analysis stack, for example,

```
start using stack "myHardDrive:myLibraryFolder:ERRATA"
```

This message needs to be sent only once, at the time you first enter your stack, so you can put it into the **openStack** handler. If you need some of the handlers which are in the *ERRATA EXTRAS* stack, then **start using** that stack too.

2. Specify where the analysis text will come from. To do this, send the message **setCorrectAnswer.** The tag may be a literal string if the analysis specification is only one line long and lacks feedback:

```
setCorrectAnswer ":ANSWER Elle a lu les dernières dix pages"

setCorrectAnswer "Elle a lu les dernières dix pages"
```

As shown in the second message,you may omit the :ANSWER label when specifying a single-line correct answer; **setCorrectAnswer** will provide it automatically. Hence, the two messages have identical effects.

If the analysis text occupies more than a single line, it will normally be in a container (field or variable) and the parameter of **setCorrectAnswer** must specify the FULL name of this container. If the analysis text is mixed with other data inside the container, then you must provide the line numbers of where the analysis text begins or ends, e.g.:

```
setCorrectAnswer the name of card field "answers"

setCorrectAnswer the name of field "drillAnswers", 13, 24
```

The latter form can be useful for drill-style activities where the the data for several items may reside together in the same field. If, for example card field "data" contains this text:

> What is the German for "one"?
> :ANSWER ein
> Yes, you got the first question right.
> What is the German for "two"?
> :ANSWER zwei
> You got the *second* question right.
> What is the German for "three"?
> :ANSWER drei
> You completed the third question successfully.

Then the analysis text for item 2 is specified by:

```
setCorrectAnswer  the name of card field "data", 5, 6
```

since lines 5 thru 6 contain the text

> :ANSWER zwei
> You got the *second* question right.

For complex analyses, you should type all of the analysis text ahead of time into a hidden field and extract it (or chunks of it) when needed. If you are using **activateField** (see (3) below), you do not need to use **setCorrectAnswer**, since **activateField** has the same function.

3. Specify the field the response will come from. There are two ways to do this. When there are several potential response fields on a card and the student is allowed to skip around from one to another of them, use **activateField**. Simply put an **enterField** handler like this into the script of each response field:

```
on enterField
       activateField "This is the correct answer"
end enterField
```

When the student clicks on one of the response fields, it immediately becomes the new current response field. **ActivateField** requires as its parameter a specification of the analysis text -- either a simple string with the correct answer, or a container where the analysis text is stored. The effect of **activateField ans** is simply to send these two messages:

```
setUpMarkUp the target
setCorrectAnswer ans
```

but it is shorter and more efficient to use **activateField**. **ActivateField** also allows you to specify whether the previous field contents should be erased when the field is activated.

If one field serves as the input location for multiple P/R interactions (as might be the case in a drill format), use **setUpMarkUp** to specify the active response field and specify each new correct answer with a new call to **setCorrectAnswer**. Usually it is best to call **setUpMarkUp** when you open the card containing the field:

```
setUpMarkUp "myField"

setUpMarkUp the name of card field "myField"

setUpMarkUp the name of field "myBkgndField"
```

After you send the first or second of these messages, card field "myField" becomes the current response field. It will remain the current response field until it is changed by another execution of **setUpMarkUp** or **activateField**. There is only one current response field at any given time. You must also send **setCorrectAnswer** to specify the analysis text, as shown above. This must be done again for each new item.

Response analysis is done only for the current response field: when the student presses RETURN in the current response field, **judgeResponse** is activated to analyze the contents of that field. If the analysis generates a markup, it is displayed beneath that field.

4. Specify where feedback should be displayed. To do this, send the message **setFeedBackField** with the name of a field on the current card as the tag, as in

```
setFeedBackField the name of bkgnd field "myFeedBackField"

setFeedBackField "myFeedBackField"
```

Either message will make card field "myFeedBackField" the current feedback field, and it will remain so until you change it again. Except for the graphical error markup (which appears in its own specially positioned field), all feedback generated by **judgeResponse** will appear in the current feedback field. To specify the field name you may provide either the full name of a card or a background field using the HyperCard **name** function, as shown in the examples above. Alternatively, you may use a short name; if so, **setFeedBackField** first looks for a card field of that name, and if it fails to find it, for a background field. If you fail to specify a current feedback field, **judgeResponse** will not display any feedback.

5. Specify where the OK and NO evaluation messages are stored. This is optional, and need be done only if you want to replace the default messages ("OK" and "NO") with ones of your own, for instance "Oui" and "Non" or "Да" and "Нет". This can be done by putting the OK message into line 1 of a special (hidden) field, and the NO message into line 2. Then execute **setOkNoFName** with that field name as a parameter. Any styling which you give to the messages will be preserved when they are shown. See ERRATA EXAMPLES # 6 (Cyrillic Font) for an example of this.

USER-SETABLE PARAMETERS

There are certain switches which your scripts can set to control the way that display will be done, specifically, whether styling (font, size, boldface, underline, etc.) will be preserved when feedback and help text are displayed, or whether styling will be stripped off, and only the bare text text displayed. You can turn styled display on and off by executing **setPreserveFormating True** or **setPreserveFormating False**. The default is to preserve styling. However, display will be substantially faster if you show the text without styling, so if the text has no special styling or you do not care whether it shows, disable this feature to improve response time.

You can control where the "OK" and "NO" message will be displayed. By default they are shown both in the feedback field and in the markup field (if there is no graphical markup). You can execute **setOkNoLoc "m"** or **setOkNoLoc "f"** to show it only in the markup or the feedback field, respectively, or **setOkNoLoc empty** to shut off display altogether. **setOkNoLoc "mf"** restores the default.

The appearance of the MARKUP edit symbols can be controlled. The simple way to do this is with **changeMarkUpSymbol**, which allows you to substitute a new symbol for one of the standard markup symbols. If, for instance, you dislike the shape of the "X" extra-word symbol and would prefer to use "?", you can accomplish this with

```
changeMarkUpSymbol  "X", "?"
```

or if you want to suppress the spelling markup for some reason, you can execute

```
changeMarkUpSymbol "x", empty
changeMarkUpSymbol "=", empty
changeMarkUpSymbol "\", empty
changeMarkUpSymbol ">", empty
changeMarkUpSymbol "<", empty
```

This will not affect the judgement in any way, but the student will not see any spelling markup, so you should inform her in some other way when there is a spelling problem. To restore the original set of symbols, execute

```
restoreDefaultMarkUpSymbols
```

If you want a special set of markup symbol shapes, you must design and use your own fixed-width font.

In some instructional situations it is desirable not to ignore punctuation when judging with MARKUP, but mark punctuation errors as if they were spelling errors. You can do this by respecifying the set of punctuation characters, which is stored in **theMarkUpPunctuation.** After executing

```
put  "{}()[]<>" & return & space into theMarkUpPunctuation,
```

for example, the characters ? ! . , ; : will be treated as alphabet letters. Be sure to include space and **return** in the punctuation set, or markup will not work properly. Note that you cannot add or remove individual marks; you must respecify the whole punctuation set. **theMarkUpPunctuation** affects only the MARKUP XFCN. To restore the default value of **theMarkUpPunctuation**, execute

```
restorePunctuation
```

Sometimes it is useful to relax the criteria which MARKUP uses to judge a response OK. This can be done in four ways: allow certain capitalization errors, allow spelling errors, allow extra words, and allow word order errors. These parameters are set using the **setJudgingParams** handler, as in these examples:

```
setJudgingParams "capFlag", "Ignore_case"
setJudgingParams "misspellOk", True, "extraWordsOk", True
setJudgingParams "anyOrderOK", True
```

To reset all these parameters to their default values, use

```
restoreDefaultJudgingParams
```

The **"capFlag"** and **"misspellOk"** parameters are useful if you are concentrating on grammar and do not want to distract the student into attending to peripheral issues; **"anyOrderOk"** is useful when you want the student to type in a list of words, and don't care about the order. **"extraWordsOk"** can be useful for doing keyword judging.

Punctutation, markup symbols, judging parameters, and styled text display are all "sticky" parameters: once you have changed them, they keep their new values as long as HyperCard is running, or until you change them again. To restore all of them to their default values, execute

```
resetErrata
```

It is wise to do this at the beginning of each new activity (usually in the **openCard** script) so that you don't accidentally inherit unexpected values of the parameters from previous activities.

☞ *ERRATA EXAMPLES* # 10 (Sentence Transitions), #16 (Vocabulary Help), and # 17 (Judging Lists) manipulate various judging parameters. Every example in the stack starts out with **resetErrata**.

NON-ROMAN FONTS

If the student is expected to respond in a non-roman writing system, you must explicitly specify the response and markup fonts. The font used for the response field must be fixed-width so that markup characters will align properly beneath response characters. By default, *ERRATA* sets the response font to Courier. But if you put a valid font name into the global variable **responseFont**, then that font will be used instead. (Be sure that the specified font is fixed-width.) If the student is expected to type in Russian, for instance, you will have to change the response font to something like Kurier, a fixed-width Cyrillic font intalled in the ERRATA EXAMPLES stack

```
put "ER Kurier 1252 Normal" into responseFont
```

Whatever font is specified for the response field will be used by default for the markup field also. When the response font is Courier, this guarantees that the standard markup symbols will appear in the markup field. If, however, the response field is something like Kurier, this may not be the case, and you will have to specify the response font explicitly by putting it into the global variable **markupFont**:

```
put "Courier" into markupFont
```

Or, if you have built a fixed-width font with your own custom-made markup symbols, or are using someone else's, you will have to specify it

```
put "MyOwnMarkUpFont" into markupFont
```

The markup characters must be the same (fixed) width as the response characters. To satisfy this constraint, you may have to use two different text sizes. For instance, "Courier" 14 point and "ER Kurier 1252 Normal" 12 point both have a fixed character width of 7 pixels. In this case, set the text sizes of the response and markup fields directly with a set command after executing setUpMarkUp:

```
set textSize of muFName to 14
set textSize of responseField to 12
```

If the non-roman font you are using puts alphabet letters into character slots that are normally punctuation, you will have to change the punctuation set to reflect this. For instance, Kurier users keys ' " \ to store cyrillic characters, so these must be removed from the punctuation set:

```
put " /?!,.;:" & return into theMarkUpPunctuation
```

If the non-roman font you are using has put characters in strange places, e.g., assigning vowels to character codes which are normally assigned to consonants, you should let the MARKUP XFCN know which characters are vowels and which are consonants, and which characters have diacritic marks by putting that information into the global variable **theMarkUpCharInfo**. Since these data are not readable unless displayed in multiple fonts, they should be installed in a field:

```
c,down_case,абвгдеёжзийклмнопрстуфхцчшщъыьэя
c,up_case,АБВГДЕёЖЗИЙКЛМНОПРСТЫФХЦЧШЩЪЫЬЭЮЯ
b,Â,ёЁ
d,diarsis,ёЁ
b,i,йЙ
d,grave,йЙ
```

When initializing the activity, these data should be put into the global variable **theMarkUpCharInfo**, which is where MARKUP looks for such information. This global should be cleared when you are no longer using the Cyrillic font, since it will cause strange markups if applied to roman fonts. See Hart (1994, pp 16-17) for details on the format and meaning of character information data.

Finally, there is the matter of the keyboard. Kurier puts many of the Cyrillic characters on OPTION and SHIFT-OPTION keys, and provides no rational keyboard mapping from English key caps to Cyrillic characters, so it is essential to remap the keyboard. This can be done by manipulating KBD resources with RESEDIT, but a simpler solution is to use *ERRATA*'s MAPKEYS XFCN inside of a **keyDown** handler in each response field to intercept and remap keypresses. Again, the map data should be put in a field so that multiple-font display is possible. You may also want to use this remapping when authoring in response analysis data, in order to get an easier-to-use keyboard. If so, you will need a way to switch back an forth between the Cyrillic and the normal roman mapping.

☞ *ERRATA EXAMPLES* # 6, (Cyrillic Font) shows in detail how to carry out this example.

KEEPING A PERFORMANCE HISTORY

Sometimes it is also instructionally useful to categorize unsuitable responses into different types or classes, an operation known as error analysis. Doing this may be instructionally useful because different errors require different responses from the program. For example, direct error feedback simply describes the error type for the student on the assumption that the student will somehow use this information to improve future performance.

In CALL, suitable responses can also be profitably subcategorized according to the kinds of underlying competences (e.g., grammar rules) required to create them. Describing a correct response in this way gives a (partial) view of the current state of the student's knowledge. The computation needed to identify categories can range from very simple to very complex, depending on how abstract the error categories are. Surface analyses may describe errors in terms of the presence or absence of a word or ending or particular word order. "Deeper" analyses take a view more like that of transformational grammar, which distinguishes between a surface structure and a deep structure. Surface errors are assumed to be accounted for by some sort of underlying linguistic mechanisms. These deeper mechanisms may be conceived of as (deviant) phrase structure or transformational rules in the student's interlanguage or, in a principles and parameters context, as parameter settings.

Such deep mechanisms will not, in general, be directly inferable from a single response, since many hypothetical mechanisms might be compatible with the same observed response. Consequently, it will be necessary to aggregate information from a number of different responses. The various observed surface forms will either confirm or disconfirm the presence of various hypothesized underlying rules, and a sufficiently large body of evidence provided by surface forms will hopefully eliminate all but the "actual" underlying mechanisms. The problem here is to get reliable inference from the observed surface forms to the underlying evidence. One method of building such inferential intelligence into programs is to use techniques, such as pattern-driven inference systems, imported from AI. When developed in a thoroughgoing way, such approaches allow the program to construct a student model which represents the linguistic competence of an individual student in some domain such as syntax or morphology. This is the approach followed by BUGGY (Brown & Burton, 1978) and the tutorial systems descended from it.

In practice, since there is little agreement on the nature of an appropriate underlying model for language acquisition, and since learner performance tends to be rather variable, it is often more feasible to simply keep performance statistics for each of a number of relatively surface categories. Performance statistics constitute a simple kind of student model. Typically, they refer to grammar structures, either standard or deviant, defined in terms of surface features such as

Concordance: Case agreements such as subject/verb agreement, adjective/noun agreement. Morphological case agreements such as stem/ending
Dependencies: Case dependencies, verb/preposition dependencies, adjective/verb dependencies.
Word order: Surface word order specifications

This is the descriptive vocabulary typically used by language textbooks as well as traditional descriptive grammars. Grammar rules expressed in this way may lack in theoretical rigor, but continue to be favored in instructional contexts because they are simple enough to be understood and applied by language learners.

As it happens, properties specifying such rules can often be identified by straightforward matching techniques.

Student models (including performance statistics) can be shown directly to the student or instructor and this may be their most common use in programs that maintain them. Ideally, however, they would serve as input to an adaptive instructional strategy. The general idea of adaptive strategy is to tailor instruction so that it will be appropriate to each individual student. Usually this is a matter of examining the learning history to see which rules have already been thoroughly learned. Presentation of the learned material is then de-emphasized in favor of material that has not yet been mastered.

Computationally, maintaining a learning model requires examining student responses to see whether standard rules have been violated and deviant rules followed. Although this can be done with relatively simple matching tools, a few pitfalls must be noted. First, the model must maintain two numbers for each error category: an error counter, the number of times the error occurred, and an opportunity counter, the number of times that it had the opportunity to occur. The ratio of these two numbers, the error rate, measures the probability that a particular type of error will occur, given that the context allows for it. Error frequencies by themselves are misleading: the fact that a student misused the French subjunctive 0 times might mean perfect mastery, or might simply mean that the subjunctive has not yet been introduced, so that no response has required its use.

Finding out whether a particular type of error had an "opportunity" to occur requires careful thought. For example, consider the following prompt and the two patterns for identifying an adjective order error :

Translate to French: Jean has two black cats. (Correct answer: Jean a deux chats noirs.)

```
:MATCH no (noir&_chat)
        Adjective order error.

:MATCH no -(chat&_noir)
        Adjective order error.
```

If the response is "Jean a deux noirs chats", then both patterns will be matched. But the first pattern looks for presence and the second for absence. There is only one way that the first pattern can succeed. The second can succeed in many more ways, and thus may match too many patterns.

In the first :MATCH above, a specific error is identified as being present. Since "noirs chats" is in the response, the error clearly had the opportunity to occur, since the two constitutent words are present and the student has misordered them. Hence, the opportunity counter for the ADJ-ORDER error can be incremented, and so can the error counter. In this case, there is really only one way that the adjective order can be wrong, so this is a perfectly reasonable approach. However, it must be combined with another pattern, which checks for correct usages of the construction:

```
(noir&_chat)        INCREMENT OPPORTUNITY COUNTER
                    INCREMENT ERROR COUNTER

(chat&_noir)        INCREMENT OPPORTUNITY COUNTER
```

The second :MATCH pattern in the example, -(chat&_noir), however, simply establishes that the specified pattern was not found, but does not tell why -- it might have been that both words were present but that the order was reversed, but it might also have been that one or both expected constitutent words were absent. In fact, this second pattern would match if the response happened to be either of the following

Jean a deux animaux.

Jean a deux chiens blancs.

and thus would attribute an adjective order error to both sentences. But this is clearly wrong, because there is no adjective in the first response, and in the second, the adjective/noun order is perfectly correct.

As a general principle, a rule cannot come into play unless the constituents which enter into it exist. Thus, determining a rule violation is a two-step process

1. Establish that all the constituents involved in the rule are present
2. Determine that the correct construction is absent.

In the example above this can be done by using the two patterns

((_chat)^(_noir))	INCREMENT OPPORTUNITY COUNTER
-(_chat_noir)	INCREMENT ERROR COUNTER

so that the full pattern required to establish an adjective order error is:

:MATCH no ((_chat)^(_noir)^(-(chat_noir)))
Adjective order error.

This will give error feedback under the right conditions. In order to compute error rates, however, we must count both opportunities for the error and actual occurrences. This requires writing some actual HyperTalk code:

```
if  char 1 of match( "((_chat)^(_noir))", response ) = "t" then
         add 1 to baseCount
         if char 1 of match( "(-(chat&_noir)))", response ) then
                   add 1 to errorCount
         end if
end if
```

The two variables **baseCount** and **errorCount** are supposed to hold the number of opportunities and the number of actual errors for the adjective order rule. Notice that <u>this is HyperTalk code, and that the MATCH XFCN is being accessed directly</u>. It is <u>not</u> data which can be processed by the default **judgeResponse** handler. Before this code can be used by *ERRATA*, it must be incorporated into a non-default judging handler.

☞ *ERRATA EXAMPLES* # 19 (Simple Parsing) and # 20 (Error Statistics) exemplify these techniques.

PARTIAL PARSING

The problem with what was developed in the previous section is lack of generality. One will find oneself writing lots of similar patterns like these

:MATCH no ((_chat)^(_noir)^(-(chat_noir)))
Adjective order error.

:MATCH no ((_chien)^(_grand)^(-(chien_grand)))
Adjective order error.

:MATCH no ((_livre)^(_important)^(-(livre_important)))
Adjective order error.

...

for various sentences. This is a lot of work just to check for one error, and it will be even more work if, instead of simply giving feedback, we are keeping counts for a performance history. To make this a practical technique for lesson authoring, more efficient means of error checking are required.

We can begin improvements by noting that, for the purposes of adjective order analysis, the exact words are irrelevant; the important thing is whether two abstract patterns are present, which could be represented something like this:

((N)^(A))	OPPORTUNITY
-(N&_A)	ERROR

where N symbolizes any (French) noun stem and A any (post-positioning) French adjective stem. If we can arrange to replace the words in a phrase by their corresponding parts of speech, then the same two patterns above will suffice for all adjective-noun phrases.

The *ERRATA* function **subst()** performs the needed substitutions:

```
subst( "Les chats noirs",1 , "chat", "N", "noir", "A")

                                    ==>  "les Ns As"
```

You can think of these substitutions as a form of dictionary lookup: **subst()** looks for several words (actually, several roots) in the sentence and, when it finds one, substitutes a "dictionary entry" which, in this case, is simply an indicator of the root's part of speech. The number following the target string tells the maximum number of times each substitution is to be done. Thus, "N" will be substituted for "chat" once and "A" will be substituted for "noir" once within the string "Les chats noirs", yielding the result shown.

Maintaining a performance history involves keeping track of frequency counts. Since we are likely to be checking for many types of errors, it would be wise to set up a data structure to do this. This data structure will comprise the performance history. We will use two global variables, **baseCounts** and **errorCounts** as arrays to store the counters for all error types. Each type of error will be assigned a number, and, so that we don't have to remember which number goes with which error, we will associate a name with each number:

```
global  subjVbAgr, adjNAgr, adjOrd

put 1 into subjVbAgr
put 2 into adjNAgr
put 3 into adjOrder
...
```

Item number i of each array will hold the counter for error number i, so that, for example, item 3 of **baseCounts** holds the opportunity counter for error type 3, **adjOrder**.

Once these matters have been taken care of, we can write a single handler which will do the adjective order analysis. As input, it requires a noun stem and an adjective stem to look for in the current response:

```
on checkAdjOrder  noun, adj

       global  responseField, baseCounts, errorCounts, adjOrder

       put subst((card field responseField), noun, "N", adj, "A") into response

       if  char 1 of match( "((N)^(A))", response ) = "t" then
            add 1 to item adjOrder of baseCounts
            if char 1 of match( "(-(N&_A)))", response ) = "t" then ¬
            add 1 to item adjOrder of errorCounts
       end if

end  checkAdjOrder
```

The **checkAdjOrder** handler implements a partial error-tolerant parsing of the contents of the current response field. It is partial because the whole response is not parsed, just a section identified by the two patterns. This in turn is determined by the presence and location of the two input words. It is also partial because only one rule is applied, the adjective order rule. A full parsing would apply as many rules as necessary to span the full response. In order to fully cover adjective ordering in French, of course, an analogous handler **checkPreAdjOrder** would have to be written to handle adjectives such as "grand" that regularly preceed the nouns they modify. The parsing is error-tolerant because it is able to provide a structural description for a constructions which deviates from standard French.

How can such handlers be incorporated into **judgeResponse**? This is the task that the :DO directive was designed for. Recall that the tag of the :DO directive must be a HyperTalk message (a call to a handler, accompanied by any parameters appropriate for that message). Hence, the analysis data for a series of items might look like this (as in earlier examples, the prompt is stored along with the analysis text as the first line but is not part of the analysis text).

```
:HELP  prompt
  Translate to French:  Jean drives a red Ferari.
:ANSWER Jean conduit un Ferari rouge.
:DO no checkAdjOrder "Ferari", "rouge"
#
:HELP  prompt
  Translate to French:  Jean has two big dogs and a black cat.
:ANSWER Jean a un chien noir.
:DO no checkPreAdjOrder "chien", "grand"
:DO no checkAdjOrder "chat", "noir"
#
:HELP  prompt
  Translate to French:  Jean has a comfortable apartment.
:ANSWER Jean a un appartement confortable.
:DO no checkAdjOrder "appartement", "confortable"
```

The :DO specifications have negative polarity so that the evaluation will be NO unless :ANSWER is satisfied.

Writing specialized parsing for things like subject/verb agreement is more difficult because morphology is involved. The following script, taken directly from *ERRATA EXAMPLES* #21 (Response History), shows a more developed version of this approach which checks for both kinds of adjective order errors as well as subject/verb and adjective/noun agreement.

```
on initializePerformHistory

  -- Reset all performance history counters to 0.

  global baseCounts, errorCounts, errorLabels

  put "SubjVAgr,adjNAgr,adjOrder" into errorLabels
  repeat with i = 1 to number of items in errorLabels
    get item i of errorLabels
    do "global" && it
    do "put i into" && it
    put 0 into item i of baseCounts
    put 0 into item i of errorCounts
  end repeat

end initializePerformHistory


on checkPreAdjOrder  noun, adj

  -- Check whether ADJ precedes NOUN; update ADJORDER counters.

  global  responseField, baseCounts, errorCounts, adjOrder

  put subst( value( responseField ),, noun, "N", adj, "A") into response

  if  char 1 of match( "((N)^(A))", response ) = "t" then
    add 1 to item adjOrder of baseCounts
    if char 1 of match( "(-(A&_N)))", response ) = "t" then
      add 1 to item adjOrder of errorCounts
    end if
  end if

end  checkPreAdjOrder


on checkAdjOrder  noun, adj

  -- Check whether ADJ follows NOUN; update ADJORDER counters.

  global  responseField, baseCounts, errorCounts, adjOrder

  put subst( value( responseField ),, noun, "N", adj, "A") into response

  if  char 1 of match( "((N)^(A))", response ) = "t" then
    add 1 to item adjOrder of baseCounts
    if char 1 of match( "(-(N&_A)))", response ) = "t" then
      add 1 to item adjOrder of errorCounts
    end if
  end if

end  checkAdjOrder
```

```
on checksubjVAgr  subj, verb, class

  -- Finds ending features of SUBJ and VERB and see if their
  -- person & number agree.  Updates performance history counters
  -- for SUBJVAGR. CLASS = verb conjugation = 1 or 2.
  -- If any ending features are unidentifiable, do not score this item.

  global  responseField, baseCounts, errorCounts, subjVAgr

  put value( responseField ) into r
  if (subj is in r) AND (verb is in r) then
    put endingFs( subj, "n" ) into sFs
    put endingFs( verb, "v" & class ) into vFs
    if validFs( sFs, vFs ) then
      add 1 to item subjVAgr of baseCounts
      if NOT fsAgree( sFs, vFs ) then ¬
      add 1 to item subjVAgr of errorCounts
    end if
  end if

end  checksubjVAgr


on checkAdjNAgr  noun, gender, adj

  -- See whether person and number features of NOUN and ADJ agree.
  -- Updates performance history counters for adjNAGR.
  -- If any ending features are unidentifiable, do not score this item.

  global  responseField, baseCounts, errorCounts, adjNAgr

  put value( responseField ) into r
  if ( noun is in r ) AND ( adj is in r ) then
    put endingFs( noun, "n" ) into nFs
    put gender into char 1 of nFs  -- Replace person with gender.
    put endingFs( adj, "a" ) into aFs
    if validFs( nFs, aFs ) then
      add 1 to item adjNAgr of baseCounts
      if NOT fsAgree( nFs, aFs ) then add 1 to item adjNAgr of errorCounts
    end if
  end if

end  checkAdjNAgr


function  endingFs wd, partOfSpeech, class

  -- If whole WD is in response, return feature values of WD.
  -- If WD is present as a stem, return the feature values
  -- which go with WD's ending.  Else return "*FAILED*".
  -- PARTOFSPPEECH is WD's part of speech (N, V1, V2 or A);
  -- CLASS is an optional morphological subclass (not used here).
  global responseField

  if wd = er     hen return empty

  put value( responseField ) into r
  put endingOf( wd, r ) into e
  if e = "*failed*" then   -- WD NOT PRESENT IN RESPONSE.
    return empty
  else if e = wd then      -- FU'L FORM OF WD PRESENT.
    if "n" is in partOfSpeech then -- Pronouns.
      get tableLookUp( wd, ",je 1s,tu 2s,elle 3s,il 3s,on 3s,nous
             1p,vous 2p,elles 3p,ils 3p", "3s" )
    else if "v" is in partOfSpeech then  -- Irreg verb Être.
      get tableLookUp( wd, ",ai 1s,as 2s,a 3s,suis 1s,es 2s,est
             3s,sommes 1p,êtes 2p,sont 3p" )
    else if "a" is in partOfSpeech then -- Irreg adjectives.
      get tableLookUp( wd, ",beau ms,belle fs,belles fp,beaux mp", "ms")
    end if
  else if partOfSpeech = "n" then  --WD IS STEM; DO MORPH ANALYSIS.
    get tableLookUp( e, ",s 3p,al 3s,aux 3p", "3s" )  -- N endings.
  else if partOfSpeech = "v1" then  -- ER verb present indic endings.
     get tableLookUp( e, ",e 1s.3s,es 2s,ons 1p,ez 2p,ent 3p", )
  else if partOfSpeech = "v2" then  -- IR verb present indic endings.
    get tableLookUp( e, ",is 1s.2s,it 3s,issons 1p,issez 2p,issent 3p",)
  else if partOfSpeech = "a" then -- Adjective endings.
    get tableLookUp( e, ",s mp,e fs,es fp", "ms" )
  else get empty
  if it ≠ empty then return it else return "*failed*"

end endingFs


function fsAgree fList1, fList2

  -- FLIST1, FLIST2 are two feature values each of
  -- form Fi or Fi.Fj.Fk...
  -- Return TRUE if two feature values agree, else FALSE.
  -- N. B.:  Lists are order-sensitive.

  return ( fList1 is in fList2 ) OR ( fList2 is in fList1 )

end fsAgree


function  tableLookup wd, table, defaultFs

  -- TABLE is a table of comma-separated [ENDING FEATURES].  E.g.,
  -- ",e 1s.3s,es 2s,ons 1p,ez 2p,ent 3p"
  -- which means that ending "e" is either 1st person singular or 3rd
  -- person singular; "es" is 2nd person singular; "ons" is 1st person
  -- plural, etc.

  get offset(comma & wd & space, table)
  if it > 0
  then return item 1 of word 2 of char it to 999 of table
  else return defaultFs

end  tableLookUp
```

```
      function endingOf wd, r

        -- WD is the root or stem of some word.  If whole WD is present
        -- in string R, return WD.  Else, find WD as stem in R and return
        -- its ending. If no form of WD present, return "*FAILED*".
        -- Default for R is current response.

        global responseField

        if r = empty then put value( responseField ) into r
        get offset(space & wd, space & r)  -- Look for WD as stem
        if char it + length(wd) of r = space then  -- Found exact form WD
          return wd
        else if it > 0 then
          return word 1 of char it + length(wd) to 999 of r  -- Ending of WD.
        else return "*failed*"  -- WD not present.

      end endingOf


      function validFs

        -- Return True if every input param is a valid feature value
        -- (i.e., not "*fail*"), else False.

        repeat with i = 1 to the paramCount
          if param(i) = "*fail*" then return False
        end repeat
        return True

      end validFs


      on showCounters

        -- Compile and display table of all performance history counters.

        global errorLabels, baseCounts, errorCounts

        put "Error Category,  Error Count,  Base Count" & return into display
        repeat with i = 1 to number of items in errorLabels
          put item i of errorLabels && item i of errorCounts && ¬
          item i of baseCounts & return after display
        end repeat
        feedBack display, "append"

end showCounters
```

Here morphological analysis is done by the handler **endingOf(wd)**, which finds the stem **wd** in the current response field and returns any ending the stem may have. The function **endingFs(word, partOfSpeech)** takes a stem and return the <u>features</u> which go with whatever ending **word** has. Some small ending "dictionaries" which associate endings with features are stored as hard-coded lists inside **endingFs()**. The function **fsAgree(fList1, fList2)** checks to see if two feature specifications agree. The handler **checkSubjVAgr subj, verb** takes the root of the subject and the root of the verb and determines if their ending disagree. If so, the proper counters are incremented. The handler **checkAdjNAgr** does the same for a noun and its associated adjective. Both the latter handlers use **validFs()** to check whether the endings attached to the noun, adjective or verb were actually valid French endings. If any of them were not, then, of course, no features could be returned. In that case, the agreement checks are meaningless and will not be performed.

Here is the analysis text utilized by this script:

```
:HELP prompt
  Translate to French:  Claude drives a red Ferari.
:ANSWER Claude conduit un Ferari rouge.
:DO  no checkAdjOrder "Ferari", "rouge"
:DO  no checkSubjVAgr "Claude", "condu", 2
:DO  no checkAdjNAgr "Ferari", "m", "rouge"
#
:HELP prompt
  Translate to French:  Claude has two big dogs and a black  cat.
:ANSWER Claude a deux grands chiens et un chat noir.
:DO no checkPreAdjOrder "chien", "grand"
:DO no checkAdjOrder "chat", "noir"
:DO no checkSubjVAgr "Claude", "a"
:DO no checkAdjNAgr "chien", "m", "grand"
:DO no checkAdjNAgr "chat", "m", "noir"
#
:HELP prompt
  Translate to French:  She has a comfortable apartment.
:ANSWER Elle a un appartement confortable.
:DO no checkAdjOrder "appartement", "confortable"
:DO no checkAdjNAgr "appartement", "m", "confortable"
```

☞ This example is implemented as *ERRATA EXAMPLES* # 21 (Response History).

This sketch of a programming approach to partial parsing ignores many complications, among them the possibility of multiple occurrences of the same word in a response and the ambiguous feature values of endings, which is common in languages such as German.

Such an approach to limited parsing is reasonable for CALL activities which involve limited vocabulary and morphology and focus only a few grammar points, but becomes hopelessly inefficient when each response must be tested against many rules. Of course program efficiency can be improved by rewriting HyperTalk functions as XFCNs, organizing all of the words and endings of the target language into a single dictionary for quick lookup, treating morphological juncture phenomena systematically, etc. But the eventual outcome of moving in this direction is designing and implementing a general-purpose parser.

The important point to realize is that, from the perspective of error analysis, classical pattern matching and parsing are not mutually exclusive techniques. They represent two extremes along a continuum of abstaction and generality. The choice of where to locate the design of an error analysis along this continuum will depend very much on the nature of the material and the kind of information that one wishes to obtain from the analysis. Not only does limited parsing implemented by matching take less up-front development than a highly generalized syntactic parsing system. It may be more flexible, allowing the user to diagnose semantic and pragamtic errors not accessible within the framework of a highly generalized parsing.

ERRATA REFERENCE

This section provides fairly complete documentation for the HyperTalk handlers which comprise the response analysis machinery in *ERRATA* and *ERRATA EXTRAS* It is intended mainly for those who want to use more advanced features of the various handlers and functions in writing their own scripts. The first part of the reference describes handler and functions used by *ERRATA* and the second describes the global variables.

The MARKUP XFCN is not documented here; Hart (1989; 1994) give detailed technical information on the internal design of the MARKUP XFCN.

OVERALL STRUCTURE OF *ERRATA*

ERRATA is structured to deal with four fields on any card where it is invoked:

A response field, where the student will type in the response to be analyzed.
An "answer" field, which holds the text of the analysis specifications and feedback.
A feedback field, where all response-contingent, author-specified feedback will be displayed.
An evaluation message field, where alternatives to the "OK" and "NO" messages are stored.

Of these four, the response field is absolutely required, while the others may be absent if the nature of your analysis permits. *ERRATA* does not automatically know the names of these four fields. Your handlers must use the **setUpMarkUp, setCorrectAnswer, setFeedBackField** and **setOkNoField** handlers to specify them.

When **setUpMarkUp** is used to stipulate the name of the response field, *ERRATA* creates a transparent field called "markup" to hold the graphical markup symbols and places it immediately behind the response field. It will be extended one character to the left, to accommodate any "Δ" (missing word) symbols which may preface a line of the response, and will extend about one line below the response field, since markup characters must be positioned beneath the response letters.

The response field will also be converted to "transparent" style so that the markup characters beneath will show through. Consequently, both markup and response fields are transparent, lacking borders to delineate the area for typing into. Therefore, *ERRATA* creates a rectangle-style field, slightly larger than the markup field and positioned just behind it. Its rectangular frame delineates the typing area and, because it is opaque, it "shadows" out any underlying graphics and allows the response characters and markup symbols to show clearly. The name of this "shadow" field is generated by putting an "*" before the name of the response field. Note that a shadow field will be created for each response field, but there will be at most one "markup" field in the foreground and one in the background. These modifications to the card are permanent; *ERRATA* makes no attempt to restore the card to its original condition when interaction with the response field is completed.

"Answer" field is a misnomer; "analysis text" field would be more appropriate since the field can hold not only :ANSWER but :WRONG, :MATCH, :NUMBER, and :DO specifications along with their contingent feedback. If the answer field is present, it will normally be hidden. It need not exist if the analysis is specified using a variable, expression or literal string.

The *ERRATA* handlers fall into five functional categories:

Interface handlers which connect the user's input and display fields to *ERRATA*. (**setMarkUp, setCorrectAnswer, setFeedBackField, setOkNoField, activateField, makeShadowField, defaultKeyHandling**).

Judging parameter handlers, which set parameters which control the details of the analysis process. (**resetErrata, setJudgingHandler, setJudgingParams, changeMarkUpSymbol, restoreDefaultMarkUpSymbols, restoreMarkUpDefaults**).

Text Display handlers, which access and display chunks of styled text. (**getHelp, getFeedBack, showText, feedBack, showMarkUp, okWord, noWord, hiliteMatch, dereferenceText, labelLines, displayLabel, quickCopy, copyText, setPreserveFormating**).

Analysis handlers, which interpret the analysis specifications and provide control structure for the overall analysis process. (**MARKUP XFCN, MATCH XFCN, markUpUsingParams, judgeResponse, responseAnalysis, handleKey**).

Utility handlers, for doing useful string operations such as case change and substitution. (**subst, substResp, upCaseResp, delimiterTable XFCN, STRINGUTILITIES XFCN**).

Straightforward uses of *ERRATA* require the lesson author to manipulate mainly the interface handlers.

A useful CALL package must be prepared to deal with a wide range of writing systems, and this means that multiple text fonts must be supported. All *ERRATA* display handlers can, in fact, preserve text styling, but only at the price of display slowdown and some fairly baroque programming. This is a direct consequence of HyperTalk's continuing failure to provide any quick efficient way to copy styled text or store it in variables: styled text must be copied by the clumsy device of going to the source field and copying it into the clipboard, then going to the destination and pasting the clipboard. Furthermore, ordinary HyperTalk chunk expressions cannot be used to manipulate styled text ranges. Instead, character or line pointers must be maintained and dereferenced, adding greatly to the scripting complexity.

Table 3 summarizes the input parameters/return values of the major *ERRATA* routines. It is not intended to be exhaustive.

Table 3

PARAMETERS AND RETURN VALUES OF MAJOR *ERRATA* ROUTINES

"[]" = direct input parameters and direct return values.
"{}" = display operations or other side effects
Unbracketed items name global variables.

Handler/Function Name	Input values	Output values/Effects
1. CALLED BY USER:		

setUpMarkUp	[field name]
	[keep response]
		muField
		responseField
		{make markup field}
		{make shadow resp. field}

setCorrectAnswer	[field name or expression] [start line number] [stop line number] [keep response]	theMarkUpAnsFName theMarkUpAnsVar theMarkUpAnsL1 theMarkUpAnsL2 theMarkUpAns
activateField	[field name or expression] [start line number] [stop line number] [keep response]	theMarkUpAnsFName theMarkUpAnsVar theMarkUpAnsL1 theMarkUpAnsL2 theMarkUpAns
setJudgingHandler	[handler name]	theJudgingHandler
setFeedBackField	[field name]	theFeedBackFName
setOkNoFName	[field name]	theOkNoFName
setOkNoLoc	[char string of loc indicators]	suppressOkNo
setJudgingParams	[keywords & param values]	capFlag extraWordsOk anyOrderOk misspellOk, etc.
restoreDefaultJudgingParams		capFlag extraWordsOk anyOrderOk misspellOk, etc.
setPreserveFormating	[True or False]	preserveFormating
defaultKeyHandling	[True or False]	userKeyHandling
changeMarkUpSymbol	[pairs of symbols]	theMarkUpSymbols
restoreDefaultMarkUpSymbols		theMarkUpSymbols
restoreMarkUpDefaults		capFlag extraWordsOk anyOrderOk misspellOk, etc. theMarkUpSymbols theMarkUpPunctuation
resetErrata		theFeedBackFName theOkNoFName preserveFormating userKeyHandling theJudgingHandler capFlag extraWordsOk anyOrderOk misspellOk, etc. theMarkUpSymbols theMarkUpPunctuation

2. USED BY ERRATA:

keyDown	[user's keystroke]	{initiates judging, hides feedback, or passes key}
judgeResponse	theResponse muField theMarkUpAnsFName theMarkUpAnsVar theMarkUpAnsL1 theMarkUpAnsL2 theMarkUpAns theBestFitThreshold	the response thcJudgment theMarkUp theFeedBack okFeedBack noFeedBack thePatternFeedBack okCount noCount [bestFit, lineNo of best fit ans, markup for best fit ans]
responseAnalysis()	[model] [response] theBestFitThreshold	the response theJudgment theMarkUp theFeedBack theOkFeedBack noFeedBack thePatternFeedBack okCount noCount [bestFit, lineNo of best fit ans, markup for best fit ans]
markUpUsingParams()	[model] [response]	[markup string] theMarkUpReturnValues theMarkUpMaps
getFeedBack()	[lineNo of best fit Ans] theMarkUpAnsVar theMarkUpAnsL1 theMarkUpAnsL2 theMarkUpAnsFName	theFeedBack
getHelp()	[help spec label] theMarkUpAnsFName theMarkUpAnsVar theMarkUpAnsL1 theMarkUpAnsL2	[txt with line references]
showMarkUp	theMarkUp muField responseField	{format string in mu field}
feedBack	[text with line references] [append] theFeedBackFName	{display text in fdbk field}

showText	[txt with line references] [field name] [append]	{display text in field and execute command lines}
labelLines	[label] [source field] [delimiter char]	[bracketed line range]
showLabel	[label] [source field] [destination field] [delimiter char]	{display block of styled text}
quickCopy	[source chunk type] [source start chunk number] [source end chunk number] [source field name] [destination chunk type] [destination start chunk number] [destination end chunk number] [destination field name]	{copy styled chunk range}
dereferenceText()	[txt with line references]	[dereferenced, destyled text]
okWord()		[text with line references]
noWord()		[text with line references]
hiliteMatch	[match pattern] responseField	{hilite pattern in response}
makeShadowField	[field name]	{clones, resizes, and renames response field}

3. UTILITY ROUTINES

subst()	[text] [number of times to substitute] [pairs of old, new strings]	[substituted text]
substResp()	[number of times to substitute] [pairs of old, new strings] theResponse	theResponse
upCaseResp()	[strip diacritics] theResponse	theResponse
labelLines()	[field name] [label] [delimiter]	[line ptrs to a labeled block of text]
displayLabel	[label] [source field name] [destination field name] [delimiter]	{styled copy of labelled text from source to dest}

CALL STRUCTURES

1. **keyDown** message, sent when the user presses any key:

```
keyDown
       handleKey
              pass keyDown
              judgeResponse
                     ...etc. (see 3 below)
```

2. **setUpMarkUp**, which establishes the current response field and arranges the markup and associated fields:

```
setUpMarkUp
       resolveFName()
       fieldExists()
       makeShadowField
       stringUtilities("strWidth") XFCN
       stringUtilities("fontInfo") XFCN
```

3. **judgeResponse**, the default top-level judging control structure:

```
judgeResponse
       fieldExists()
       responseAnalysis()
              match() XFCN
              judgeNumber
                     stringUtilities("removeChars") XFCN
              send <user :DO Handler>
              getFeedBack()
              markUpUsingParams()
                     markUp() XFCN
              computeAnsFit
              getFeedBack()
              matchingIsOK
              okWd()
              noWd()
       showMarkUp
              stringUtilities("findLineBreak") XFCN
              showText
                     resolveFName()
                     findInField() XFCN
                     findInField() XFCN
                     quickCopy()
       feedBack
              showText
                     resolveFName()
                     findInField() XFCN
                     findInField() XFCN
                     quickCopy()
```

HANDLERS AND FUNCTIONS

The handler or function and its formal input parameters are given in **boldface**. Some examples of the syntax for calling the handler and an explanation of the general function of the handler follow.

When the functioning is complex or the user may need to manipulate the handler extensively, more technical details are provided, including a discussion of variables affected by the handler, defaults for input parameters, etc.

on setJudgingHandler handlerName

```
setJudgingHandler "myVeryOwnJudgingHandler"
setJudgingHandler empty
setJudgingHandler "judgeResponse"
```

Specify that **handlerName** is the name of the handler that will be called to do response analysis when the student presses the RETURN key in a response field.

If no name has been specified, then the default name **"judgeResponse"** is used.

Once you have used **setJudgingHandler** to specify some other name, the analysis package will keep using that name indefinitely, until you change it again with another call to **setJudgingHandler**.

You will never need to use **setJudgingHandler** unless you have written one or more of your own judging handlers to preempt it.

☞ *ERRATA EXAMPLES* # 7 (Response Tracking), # 19 (Simple Parsing) and # 20 (Error Statistics) reset the judging handler. #10 (Transitional Sentences) and various others preempt the default **judgeResponse** handler.

on activateField ans, startLine endLine, keepResp

```
activateField "This is the correct answer"
activateField "Le garçon voit le chien blanc"
activateField "Der hund", "dontErase"
activateField the name of card field "myAnalysis", 15, 35, True
```

activateField should be called from within an **openField** handler in the script of some field on the current card. When the student clicks on that field, the **activateField** handler will be executed. The field which was clicked then becomes the current response field. The text specified by **ans** and **startLine, endLine** becomes the current analysis text, used to judge the response after it has been typed in.

Executing **activateField ans, startLine, endLine, keepResp** is equivalent to executing the two handlers

```
setUpMarkUp  the short name of the target, keepResp
setCorrectAnswer ans, firstLine, lastLine
```

but is more convenient. See the documentation for **setUpMarkUp** and **setCorrectAnswer** for details on how to specify the parameters.

This handler is useful when there are several response fields on the same card, each with its own answer, and the student is allowed to skip around freely from one to another. Call it from within an **openField** handler so that it will be activated when the student moves the text cursor into the field preparatory to typing in a response:

```
on      openField
        activateField "This is the correct answer"
        ...
end     openField
```

Notice, however, that a response field established with **activateField** way can have only one answer (i.e., analysis text). If the student will use the same field to respond to many questions, as in a drill activity, then your scripts will have to manipulate setUpMarkUp and setCorrectAnswer directly.

☞ *ERRATA EXAMPLES* # 4 (Multiple Response Fields) uses this handler.

on setUpMarkUp respField, fieldStyle, keepResp

```
setUpMarkUp "responseField"
setUpMarkUp name of field "myRespField", "noErase"
```

Makes **respField** the current response field. When response judging is initiated, **judgeResponse** takes the response string from this field, and places the markup beneath this field. **respField** can be either the full or short name of a card or background field on the current card.

☞ A call to the function **resolveFName(respField)** is used to determine which field **respField** refers to.

If need be, a field named "markup" will be created. Then the "markup" field will be hidden and moved up or back to the layer just behind **respField.** The default textFont, textSize, and textStyle of the "markup" field will be reset to match those of **respField**. These actions are necessary to assure that the markup symbols will be aligned correctly beneath the student's typing.

The style of field "markup" will be set to **"opaque"** so that the markup field will cover any text or graphics which happen to be behind it. Otherwise the markup symbols would not be legible. The style of the response field will be permanently changed to **"transparent"**, so that the markup symbols behind the response field will appear. Since the boundaries of the transparent response field are no longer visible, **setUpMarkUp** creates a permanent rectangular "shadow" field and places it behind both the response field and markup field. Its name will be the name of the response field with an asterisk prepended. The shadow field provides a uniform, opaque background for the response and markup display. There will be at most one markup field for the background and one for the foreground, but there will be a separate shadow field for each response.

If **keepResp** is omitted or empty, then the current contents of **respField** will be erased when **setUpMarkUp** is executed; otherwise the current contents will be left intact.

on setCorrectAnswer correctAns, startLine, endLine

```
setCorrectAnswer "This is the answer"

put ":WRONG This is a wrong answer" into var
setCorrectAnswer var

put ":ANSWER This is the answer" into card field "answerText"
setCorrectAnswer the name of card field "answerText"

setCorrectAnswer the name of bkgnd field "data", 10, 23
```

BEST COPY AVAILABLE

The name of this handler is a bit misleading. It actually stipulates the entire set of analysis specifications which will be used to do the next response analysis. Often this does consist of nothing but a correct answer, but :WRONG, :MATCH, :DO, and :HELP specifications and associated feedback can be in the analysis text too.

correctAns may be a literal string or expression, a variable which contains the analysis text, or the name of a field which contains the analysis text. In any case, **correctAns** must contain a response analysis text which is acceptable to the current judging handler. If you use the default **judgeResponse** handler, then the analysis text can mix :ANSWER, :WRONG, :MATCH, :NUMBER, :DO and :HELP specifications (including optional feedback) in any order.

If **correctAns** consists of a single line without any special label, **judgeResponse** will interpret it as the tag of an :ANSWER specification which has no feedback. That is,

```
        setCorrectAnswer "The quick brown fox"
```

is the same as

```
        setCorrectAnswer ":ANSWER  The quick brown fox"
```

If correctAns is a field name, it must be the full name of a card or background field on the current card, e.g.,

```
        card field "myField"
        bkgnd field "myField"
```

The most convenient way to get the full field name is to use the HyperCard function **the name**, as in these examples:

```
        setCorrectAnswer the name of card field "myField"

        setCorrectAnswer the name of field "myBkgndField"

        put the name of card field "myField" into analysisField
        setCorrectAnswer analysisField
```

which will cause card field "myField" or background field "myBkgndField" to be used as the source of the analysis specifications. If correctAnswer does not contain a field name, then it is assumed to contain the actual analysis data.

☞ If you have set **preserveFormating** to **True** in order to preserve the font, size and style attributes of feedback text, then **correctAnswer** *must* specify a field name. This is because only fields can support text formatting. Failure to specify a field name in this situation will result in an error dialog.

The two optional input parameters **startLine** and **endLine** allow you to specify that only a certain range of lines in the **correctAnswer** text should be used for analysis. This can be useful for drill activities where a single field contains the response analyses for a number of items, perhaps mixed in with other data. If, for example card field "data" contains this text:

```
What is the German for "one"?
:ANSWER  ein
  Yes, you got the first question right.
What is the German for "two"?
:ANSWER zwei
  You got the second question right.
What is the German for "three"?
:ANSWER  drei
```

You completed the **third** question successfully.

Then the analysis text for item 2 is specified by:

```
setCorrectAnswer  the name of card field "data", 5, 6
```

If no line numbers are specified, then the entire text specified by **correctAnswer** is used.

Calling **setCorrectAnswer** sets five global variables:

theMarkUpAnsFName: Contains the name of the field that contains the analysis text, if **correctAnswer** was specified via a field name. If **correctAnswer** was specified as a literal string or variable, **theMarkUpAnsFName** will be **empty**.

theMarkUpAnsVar: An unformatted copy of text containing the analysis text. If **correctAnswer** was specified as a field name, then **theMarkUpAnsVar** will contain the same text as that field, but without any text formatting. If **correctAnswer** was specified as a variable or literal, then **theMarkUpAnsVar** will contain that value.(This will also be the last line of the text within the field named by **theMarkUpAnsFName**, if there is one).

☞ **theMarkUpAnsVar** may be coextensive with the analysis text, or it may contain the analysis text within it, as a range of lines delimited by **theMarkUpAnsL1** and **theMarkUpAnsL2.**

theMarkUpAnsL1: Number of the first line of the analysis text within theMarkUpAnsVar (This will also be the first line of the text within the field named by **theMarkUpAnsFName**, if there is one).

theMarkUpAnsL2: The number of the last line of the analysis text within theMarkUpAnsVar. (This will also be the last line of the text within the field named by **theMarkUpAnsFName**, if there is one).

theMarkUpAns contains an unformatted copy of the current analysis text, that is, line **theMarkUpAnsL1** to **theMarkUpAnsL2** of **theMarkUpVar.**

on setFeedBackField fName

```
setFeedBackField "myFeedBackField"
```

Specify **fName** as the short or full name of the card or background field where any feedback (other than the standard graphical markup) should appear. Any special feedback messages for OK or :WRONG answers or matched patterns will appear in this field. The function call **resolveFName(fName)** is used to determine which field, if any, **fName** refers to.

Once you have set the feedback field, it will stay unchanged until you reset it with another call to **setFeedBackField.**

on setOkNoFName fName

```
setOkNoFName name of card field "myMessages"
setOkNoFName "russianMessages"
setOkNoFName empty
```

fName specifies the name of a field which holds user-defined messages to replace the standard messages "OK" and "NO". Within field **fName**, the replacement for the "OK" message must be in line 1 of the field, and the replacement for the "NO" message must be in line 2. If you replace either message, you must replace them both. If you want to suppress one of the messages for some reason, leave its line blank. To return to the default messages, execute **setOkNoFName empty.**

The text of the messages may be formatted, and the formatting will be preserved when the messages are displayed.

☞ You must hand-style the text: the default text style of field **fName** is ignored.

Field **fName** must be on the same card as the response and feedback fields. Normally. of course, it will be hidden. The function call **resolveFName(fName)** is used to determine which field, if any, **fName** refers to.

This handler is provided primarily so that the default English messages can be replaced with target-language equivalents such as "Oui" and "Non" or "Да" and "Нет".

☞ *ERRATA EXAMPLES* # 6 (Cyrillic Font) shows how to do this.

on setOkNoLoc locSpecs

(In *ERRATA EXTRAS*)

```
setOkNoLoc "mf"
setOkNoLoc "f"
setOkNoLoc empty
```

Determines where **judgeResponse** will display the current "OK" or "NO" messages. Default is to display the message both in the feedback field and (if there is no markup) in the markup field.

locSpecs must be **"mf"** or **"fm"** or **"m"** or **"f"** or **empty**. If **"m"** is present in **locSpecs**, it indicates that evaluation is to be shown in the markup field. If **"f"** is present, then evaluation will be shown in the feedback field. If both are present, then it will be shown in both places. If **locSpecs** is **empty**, then **judgeResponse** will not show the OK/NO messages.

This handler actually sets the value of **suppressOkNo**, which indicates where markup should not be shown.

☞ *ERRATA EXAMPLES* # 10 (Sentence Transitions) uses this handler to shut off display of evaluation in the markup field.

on defaultKeyHandling tf

```
defaultKeyHandling True
defaultKeyHandling False
```

If **tf** is **True**, the default key handler is enabled; if **tf** is **False**, it is disabled.

This handler is useful if you have written your own **keyDown** key handler. After your **keyDown** handler has handled the special keys such as RETURN, ENTER, etc., you will want to pass the remaining keys on up to the system so that, e.g., editing keys will have the usual effect and alphabetic keys will be inserted into the current response field in the expected way. You will probably not want the keys that you pass to be processed by the default **keyDown** handler in *ERRATA* so you should shut it off by calling **defaultKeyHandling False**. (The default key handler will still receive the keys, but it will pass them straight through without processing them.)

This handler sets the global variable **userKeyHandling** to **False** or **True**.

☞ *ERRATA EXAMPLES* # 12 (Multiple Choice) does its own key handling so that judging will be done after a single keystroke.

on setPreserveFormating tf

```
setPreserveFormating True
setPreserveFormating False
```

tf specifies whether the response analysis system will try to preserve text formatting when displaying feedback and help text. If **tf** is **True**, formatting will be preserved; if **tf** is **False**, formatting will be stripped from the display.

This handler sets the global variable **preserveFormating** to **True** or **False**, which in turn affects the operation of the handlers **getFeedBack()**, **getHelp()**, and **feedBack**.

This switch is useful because the HyperTalk **put** command will not preserve formatting when it copies text from one location to another, and because variables will not store formatted text. In order to retain text formatting (including font, size, and style), text must be copied from the source field onto the clipboard and then pasted from the clipboard into the destination field. The response analysis system provides two handlers, **copyText** and **quickCopy** which perform this service.

When **preserveFormating** is **True** (the default value), text display is handled by **quickCopy**; otherwise it is handled by **put**. However, display is noticeably slower when **quickCopy** is used, so **preserveFormating** should be set to **False** if the feedback and help text lacks any formatting features.

function okWord

```
feedBack okWord()
showText okWord(), the name of card field "myOwnField"
put okWord() into fdbk
```

Returns the current evaluation message for a correct response. The default value is simply "OK", but if a field with user-specified messages has been set using **setOkNoFName**, then line 1 from that field will be returned as an embedded reference of form {1,1,card field "myOkNoField"}. This result must be displayed using **feedBack** or **setText**.

function noWord

```
feedBack noWord()
showText noWord(), the name of card field "myOwnField"
put noWord() into fdbk
```

Returns the current evaluation message for a wrong response. The default value is simply "NO", but if a field with user-specified messages has been set using **setOkNoFName**, then line 2 from that field will be returned as an embedded reference of form {2,2,card field "myOkNoField"}. This result must be displayed using **feedBack** or **setText**.

on changeMarkUpSymbol oldSymbol, newSymbol

```
changeMarkUpSymbol "X", "?"
changeMarkupSymbol "«", space
changeMarkupSymbol "Δ", space
```

Change the default graphical markup symbol **oldSymbol** to be the symbol **newSymbol** instead. Both **oldSymbol** and **newSymbol** must be a single character. The default symbols for various types of errors are:

+	Should be upper case
-	Should be lower case
~	Missing or incorrect accent mark
x	Extra letter
\	Missing letter(s)
*	Incorrect letter
>	Transliterated letter; move right
<	Transliterated letter; move left
[	Run-together words; separate them
X	Extra or unidentifiable word
Δ	Missing word(s) at this location
«	Word out of order: move leftward to one of the "Δ" locations

This handler is useful for changing the way that errors are represented graphically, if you have a special font or believe that other symbols will be better indicators of certain types of errors. You can also suppress the appearance of certain errors markups (e.g., word order or spelling markups) by substituting a space for the default symbol.

on restoreDefaultMarkUpSymbols

```
restoreDefaultMarkUpSymbols
```

Restores all the markup symbols to their original (default) values. This is a convenient way to get back to normal if you have been making a lot of changes in the markup symbols.

on setJudgingParams p1, v1, p2, v2, ... p-i, v-i

```
        setJudgingParams "capFlag", "Authors_caps"

        setJudgingParams "extraWordsOk", True, "anyOrderOk", True
```

Gives new values to some of the standard judging parameters used by the markup XFCN.

Each **p-i** is a string which names a judging parameter. The **v-i** which follows each **p-i** specifies the value which that judging parameter should take, such as true. The value must, of course, be meaningful for the parameter in question. The following table shows each of the **p-i** at the left, and the allowable **v-i** values for that parameter, and their effects, at the right.

"capFlag"	If "**Exact_case**" (the default) then the capitalization in the response must exactly match that in the model or else cap errors will be marked and judgment will be NO. If **"Authors_caps"**, the response must have a capital whenever the model does, but additional capitals in the response are permitted. If **"Ignore_case"** then case is ignored when matching model and response.
"extraWordsOk"	If **True**, judge OK even if extra words are present in the response. If **False** (the default) judge NO if extra words are present.
"anyOrderOk"	If **True**, order of words in the response does not have to match the order of words in the model in order to get an OK judgment. If **False** (the default), judge NO if words are not in the specified order.
"misspellOk"	If **True**, judge OK even if some words are misspelled. If **False** (the default), judge NO if there is any spelling error.
"wordMarkUpNeeded"	If **True** (the default), an error markup string will be generated and returned. If **False**, no string (i.e., a null string) will be returned, only a judgment of OK or NO. If you simply wish an evaluation and don't want to display the graphic markup as error feedback, you can speed things up slightly by setting this parameter to **False**. In that case, your script can use the other information returned by MARKUP to determine what feedback to give the student.
"runtogetherNeeded"	If True (the default), MARKUP will find and mark run-together words. If **False**, run-togethers will not be identified as such, but will be marked as misspelled or unidentified words. Turning off this feature when MARKUP is running slowly will speed things up, but at the cost of degrading the quality of the markup.
"adjustNeeded"	If **True** (the default), MARKUP will try to "improve" the graphical error markup to make it more intuitive. If **False**, this improvement is not done. Do not turn off improvement unless speed is a serious problem, because it significantly degrades the quality of the MARKUP
"shortCut"	If **True** (the default), MARKUP will do a "fast" spelling analysis that will not generate a spelling markup between badly misspelled pairs. If **False**, force a complete spelling analysis for every word. Use False if you need a markup for very badly misspelled words (e.g., when using MARKUP in a spelling lesson). Turning off **shortCut** may slow the program down significantly when model and/or response are long.
"markUpMapsNeeded"	If **True**, MARKUP will generate and return in the HyperCard global variable **theMarkUpMaps** two "maps" showing which model words

are paired with which response words. If **False** (the default), this map will not be returned, and the value of **theMarkUpMaps** remains unchanged.

"parameterDisplayNeeded"	One of the characters **"v"**, **"b"**, **"d"**, **"c"**, **"h"**, **"p"**, **"w"**, **"f"**, **"s"**, or **"m"** or else nothing at all (the default). If one if these characters is present, then information of the requested type will be returned in the HyperCard global variable **theMarkUpParamDisplay**. Otherwise the value of **theMarkUpParamDisplay** remains unchanged. The character that you use as an input parameter determines the kind of information that will be returned:

"v"	VERSION of the MARKUP XFCN which is running
"b"	Table of BASE CHARACTER specifications
"d"	Table of DIACRITIC specifications
"c"	Table of CASE specifications
"h"	Table of PHONETIC CATEGORY specifications
"p"	Table of PUNCTUATION CHARACTER specifications
"w"	Values of the JUDGING WEIGHTS AND THRESHOLDS
"f"	Values of the JUDGING FLAGS
"s"	Values of the MARKUP SYMBOLS
"m"	Values in the PHON_MATRIX

This parameter allows you to copy a judging table into a HyperCard container, where it can be inspected using the SHOW VARIABLES option of the HyperCard debugger. The format in which this information is returned is discussed below.

"spellingOnlyNeeded"	If no value or **"x"** (the default), then the standard spelling and word order analysis is done. If the value is **"r"** or **"p"**, a special, spelling-only analysis will be done: the model and response strings will be immediately submitted verbatim to the spelling analyzer and an edit trace will be generated by comparing *every* character in the two strings, including punctuation, spaces, and return characters. None of the special syntax used to define synonym and ignorable word lists in the model will be recognized. Since there are no word boundaries, no order analysis will be done. The value of **spellingOnlyNeeded** determines the nature of the return:

"p"	Return a "pretty" markup string, suitable for display beneath the response string.
"r"	Return the raw markup string, without prettying it up.
"x"	Do *not* do the special spelling-only analysis; do the normal spelling and word order analysis.

Since the model and response strings are treated as if they were words when spellingOnlyNeeded is "**r**" or "**p**" *neither string can exceed the maximum word length of 20 characters.*

The information returned in the HyperCard global variable **theMarkUpReturnValues** are different for the special analysis, and consists of a raw edit distance and a normalized edit distance.

Returning a raw trace forces a least-cost edit trace string (markup string) to be computed no matter how dissimilar the model and response are, so this option is useful for spelling lessons or other cases where an exact spelling markup is needed even when a response is badly misspelled. The "pretty" markup will display properly, but only the "**r**" option has complete information about the nature of errors present, so it is appropriate if you want to do computations on the markup string.

"debugNeeded"

Setting this parameter to **True** cause technical information about the internal workings of MARKUP to be returned in the HyperCard global **theMarkUpDebug**. Included are the edit distance matrix, values of ignorable words in the model and response, and candidate match sets for each response word. This information is intended only for debugging and development purposes. If **False** (the default), no information is returned.

Since HyperCard allows a handler to accept 16 input parameters, you may specify up to 8 **p-i/v-i** pairs in each call to **setJudgingParams**. If you need to set more than 8 parameters, you can use several calls to **setJudgingParams**. Once you have changed the value of a parameter, it will keep that valueuntil you change it again, or as long as HyperCard continues to run.

☞ The parameter values you specify with **setJudgingParams** are used only if you do the markup by calling the **markUpUsingParams()** function. If you call the MARKUP XFCN directly, these settings are ignored and you must pass values directly to the "XFCN. Hence, you should normally activate the markup through **markUpUsingParams()**.

ERRATA EXAMPLES # 10 (Sentence Transitions), #16 (Vocabulary Help), # 17 (Judging Lists) and # 18 (Spelling/Dictation) modify judging parameters.

on restoreDefaultJudgingParams

```
restoreDefaultJudgingParams
```

Returns all the judging parameters to their default values.

This is a convenient way to get back to the normal (original) configuration of judging parameters if you have been changing them a lot.

on restoreMarkUpDefaults

```
restoreMarkUpDefaults
```

Returns both markup symbols and judging parameters to normal (original, default) state. Calling this handler is equivalent to calling **restoreDefaultMarkUpSymbols, restoreDefaultJudgingParams,** and **restorePunctuation,** but is more convenient.

Use this if you have made a lot of changes in the symbols and parameters and then want to return everything to the default state.

on keyDown ch

Default key handler. If the key **ch** is pressed in the current response field, then it is given to **handleKey** to examine. Otherwise, the key is simply passed on up the message hierarchy.

You program should never need to call this handler except perhaps through a pass statement. It is called automatically whenever the user presses a key.

☞ If you install your own **keyDown** handler at the field, background, card, stack, or Home stack level, you should use **pass** to pass **keyDown** messages on up the judging hierarchy so that, e. g., text editing keys will be operational in the response field.

If you do **pass keyDown**, you will probably want to disable this handler by executing **defaultKeyHandling False**. Otherwise, this default handler may process judging keys like RETURN a second time.

on handleKey ch

Used internally by **keyDown** handler.

This handler processes every key typed into the current response field. It causes a response analysis to be computed and feedback and markup to be displayed when the student presses RETURN. It also causes the graphical markup to disappear when the student begins to type in revisions. Keys other than RETURN are passes on up the message hierachy so that editing operations such as character insertion and deletion will function properly in the response field.

function markupUsingParams model, response

```
get markupUsingParams( "This is an answer", field "response" )
put markupUsingParams( ans, card field "resp" ) into markUpString
```

This function executes the markup XFCN using **model** as the correct answer and **response** as the response string. It is used internally by **judgeAnswer** and **judgeResponse.**

When this function calls the markup XFCN, it passes all of the judging parameter values which you have established using **setJudgingParams.**

Its returns are identical to those of the markup XFCN. The direct return is a graphical markup string. It also returns the evaluation of the response (OK or NO) as the first item of the HyperCard global variable **theMarkUpReturnValues,** and may also cause other requested information such as response maps to be put into other global variables. See Hart (1994). No display is generated.

function getJudgingInfo

(In *ERRATA EXTRAS*)

```
get getJudgingInfo()
```

This function returns a report which displays the values of all of the markup XFCN's internal parameters and judging tables. This gives you a complete picture of the information used by the markup XFCN to do its judging. the markup XFCN is called via **markUpUsingParams()**, so the judging parameters you have specified through HyperTalk globals will be in effect.

For a detailed description of the kind of information returned and its exact format, see Hart (1994).

The information returned by getJudgingInfo() should be displayed in Courier font.

This function is useful mainly for development and debugging purposes, but is also helpful if you are uncertain about which judging parameters are currently in use.

function findBestAns model, response

```
get findBestAns( "This is an answer", field "response" )
put item 1 of findBestAns( ans, resp ) into bestFit
```

Used internally by **judgeAnswer** and **judgeResponse.**

Sets the global variable **theJudgment**.

Direct return is information on the best-fitting answer, formatted as a list of three items: BESTFIT,BESTLINE,BESTMU. These values are identical to those returned by **judgeResponse**; see **judgeResponse** for details.

function getFeedback lineNo

```
feedBack getFeedBack( 12 )
get getFeedBack( 23 )
```

Returns the feedback which belongs to a particular :ANSWER, :WRONG, :MATCH, :NUMBER, :DO, or :HELP specification in the current analysis text. No display is generated.

lineNo is the line number of a specification, relative to the current analysis text in the global variable theMarkUpAns. All the text between that specification and the next one (or the end of the analysis text) is located.

If **preserveFormat** is set to **True**, then the actual text is returned. Otherwise, a line-range reference of form {startLine,endLine} is returned. startLine and endLine are relative to the variable or field which contains the current analysis text (i.e., **theMarkUpAnsVar**). Returning line numbers enables text formatting to be preserved provided that the feedback text is later displayed by the **feedBack** or **showText** handler.

function getHelp type

```
feedBack getHelp( "grammar" )
get getHelp( )
```

Returns the text which belongs to a particular :HELP specifier in the current analysis text. No display is generated.

type is the label for a particular kind of help, or else **empty**.

The current response analysis text in the global **theMarkUpAns** is searched to determine whether a help specification with label **type** is present. If so, the text which belongs to it is identified. If **type** is empty, then all help directives are found, regardless of label, and their texts identified.

If **preserveFormat** is not set to **False**, then the actual text(s) are concatenated, each beginning on a new line, and returned. If it is set to **True**, the return is a series of one or more line-range reference of form

{startLine1,endLine1}
{startLine2,endLine2}
{startLine3,endLine3}
...

where startLine and endLine are relative to the variable or field within which current analysis text is located (**theMarkUpVar**, or the field named by **theMarkUpAnsFName**) which may be larger than the current analysis text in **theMarkUpAns**). Returning line numbers enables text formatting to be preserved when provided that the feedback text is later displayed by the **feedBack** handler. If no help specification(s) of the appropriate type(s) are present, then the return is **empty**.

on hiliteMatch pattern

```
hiliteMatch "(_ne_*_pas_)"
hiliteMatch "(_blan)"
```

pattern is any pattern descriptor.

This handler calls the match XFCN to attempt a match of **pattern** to the contents of the current response field. If it succeeds, and if the pattern descriptor is simple (i.e., one without any AND, OR, or NOT operations), then the matched material is boldfaced within the current response field. If the pattern is not simple, no display will be generated.

Return is identical to that of the match XFCN: the first character of the return is T if the match succeeded, otherwise F. If the pattern is simple (i.e., one without any AND, OR, or NOT operations), then the remaining characters of the return show which response characters are matched by the pattern: "x" indicates a matched character; a space indicates an unmatched character. If the pattern is not simple, then spaces will appear in all character positions. Since **hiliteMatch** is a handler, the return must be accessed by **the result.**

This function is provided so that matched portions of a response can be selectively hilited as part of the feedback given to the student. In this way, attention can be focused on particular parts of the response.

function judgeNumber range, response

```
judgeNumber "(1-10)", card field "input"
judgeNumber "(1-*)", field "response"
judgeNumber "¬(5)", responseVar
```

Checks to see if a response is within a given numerical range. No display is generated.

The numerical value or range is stipulated by **range**, which must be a string containing a valid number specifier, as shown above, where "*" symbolizes infinity (or minus infinity) and "¬" symbolizes NOT.

Response is the student's response string (which need not be a number). <u>All commas are removed from response before judging</u>.

If **response** falls within **range**, then **judgeNumber** returns **True**; if **response** is not a number or falls outside of **range**, the return is **False**.

on judgeResponse

```
judgeResponse
```

This is the default judging handler, used to do response judging unless (1) there is a handler of this name lower in the message hierarchy, e.g., in your stack's field, card, or stack script, or (2) a call to **useJudgingHandler** has specified that a handler with a different name should be used to do the judging. **judgeResponse** is the most general and powerful response analyzing facility provided by *ERRATA*. It allows the lesson author to apply a sequence of :ANSWER, :WRONG, :MATCH, :NUMBER and :DO specifications to the analysis of a response.

The effect of **judgeResponse** is to apply the current analysis text (stipulated by **setCorrectAnswer**) containing various :ANSWER/:WRONG/:MATCH/:NUMBER/:DO specifications to the current response. Any number of specifications of each type (including 0) may be present, and they may be occur in any sequence. The specifications are used to determine a judgment of **True** or **False** (OK or NO) to display a markup (if one of the :ANSWER or :WRONG specifications was satisfied), and to display feedback text associated with satisfied specifications.

The response analysis text must have been previously set by calling the handler **setCorrectAnswer** or **activateField**. The response is taken from the current response field (established by the most recent execution of **setMarkUp** or **activateField**).

judgeResponse processes the specifications in order, but treats :ANSWER and :WRONG specifications differently from :MATCH, :NUMBER and :DO specifications. ANSWER and :WRONG specifications are examined to see which one best fits the current response. :MATCH, :NUMBER and :DO specifications are processed do not enter into this determination of a best fitting answer.

If there is a perfect match to an :ANSWER specification or any (perfect or imperfect) match to a :WRONG specification, the associated feedback message will be put into the current feedback card field, as specified by the most recent call to **setFeedBackField**. If no feedback field name has been specified, then the feedback will not be displayed. Note that feedback from a matched :ANSWER command is shown only if the judgment is OK (the match was perfect, as determined by current values of **capFlag, misspellOk. extraWordsOk,** and **anyOrderOk**).

:MATCH, :NUMBER and :DO specifications are also examined in sequence. Patterns are compared the response string and whenever there is a match, the corresponding feedback is saved for possible later display in the current feedback field established by **setFeedBackField**. If no feedback field name has been specified, the feedback will not be displayed.

If the polarity tag of a matched specifier is "okStop" or "noStop", then all further processing of specifiers ceases immediately. Otherwise, it continues until the last specification is examined. A final judgement value **theJudgment** is then determined in two steps:

1. If :ANSWER/:WRONG specifications have established an **True** or **False** value, then that is used.

2. If no :ANSWER/:WRONG specifications were satisfied, then the results from the :MATCH/:NUMBER/:DO specifications are examined. If all such specifications which were matched has polarity "ok" or "okStop", the final judgment is **True**; otherwise the final judgment is **False**. (This default computation can be overriden by installing a user version of **matchIsOk**.)

Feedback from :MATCH, :NUMBER, and :DO specifications is computed separately The variable **okFeedBack** contains feedback from all such matched specifications which had "ok" or "okStop" polarity. The global variable **noFeedBack** contains feedback from all such matched specifications which had "no" or "noStop" polarity. If the final judgment **theJudgment** is **True**, then **thePatternFeedBack** contains **okFeedBack**; if the judgment was **False** then it contains **noFeedBack**. Finally, **theFeedBack** contains a concatenation of the best answer feedback and **thePatternFeedBack**

Besides displaying feedback, **judgeResponse** returns the results of the analysis in three global variables:

theJudgment: **True** if response was OK, otherwise **False**.

theMarkup: Graphic markup which goes with the best-fitting answer (identical to BESTMU below).

theFeedBack: Feedback generated by the analysis, except for the graphical markup.

In addition **judgeResponse** computes more technical information on the best answer, formatted as a list of three comma-separated items: BESTFIT,BESTLINE,BESTMU. This information is set by the **return** command; however, since **judgeResponse** is a handler rather than a function, it must be accessed through **the result:**

BESTFIT is a number between 0 and 1 indicating how well the best fitting answer actually fit the response, with 0 indicating no fit at all and 1 indicating perfect fit. If none of the specifications fit, it will have the same value as **theBestFitThreshold**.

BESTLINE is the number of the line which contains the best-fitting :ANSWER or :WRONG specification, within the variable or the entire field which contains the answer text., i. e., within **theMarkUpAnsVar** or the field named by **theMarkUpAnsFName**. If none of the :ANSWER or :WRONG specifications fit, BESTLINE will be **empty**.

BESTMU is the markup that goes with the best fit :ANSWER or :WRONG specification (identical to **theMarkup** above). If none of the specifications fit, it will be **empty**.

Answer fit is computed as a function of spelling and word-order errors as follows:

ANSFIT := (3*PMATCHED*(1 - AVEDIST) + PNONINV) /4

where PMATCHED is the proportion of words matched, PNONINV is the proportion of non-inversions in word order, and AVEDIST is the average distance between matched model and response words. (For more detail on the rationale for this formula and significance of these numbers, consult Hart, 1989 and 1994) A value of ANSFIT which exceeds **theBestFitThreshold** is considered to be a fit; anything less than this is a non-fit. (The global variable **theBestFitThreshold** has a default value of .70, which may, however, be changed by the user.) BESTFIT is the maximum ANSFIT taken over all the :ANSWER and :WRONG specifications. This default computation is done by **computeAnsFit** can be preempted by installing your own **computeAnsFit** handler.

☞ *ERRATA EXAMPLES* # 10 (Sentence Transitions) uses the value of BESTLINE to compile and display a list of words missing in the response.

function responseAnalysis model, response

```
get responseAnalysis( "This is an answer", field "response" )
put item 1 of responseAnalysis( ans, resp ) into bestFit
```

Judges the response in **response** against an analysis text in **model**. Identifies the :ANSWER/:WRONG specification which best fits the response. Processes all :MATCH/:NUMBER/:DO specifications and determines appropriate feedback. This best fitting answer is used to arrive at an OK or NO judgment, and also to determine (but *not* display) appropriate graphic markup. All feedback is merged into a single text but is not displayed. No display is generated.

Results of the analysis are returned in three global variables:

theMarkup, the graphic markup which goes with the best fitting answer.

theJudgment: **True**, if the response was suitable; **False** if it was unsuitable.

theFeedBack: All feedback generated by the analysis, except for the markup.

The direct return of **responseAnalysis()** is technical information on the best-fit answer, formatted as a list of three comma-separated items: BESTFIT,BESTLINE,BESTMU. These values are identical to those returned by **judgeResponse**; see **judgeResponse** for details.

This function is used internally by the **judgeResponse** handler. It will also be useful if you want to do the same kind of judging that **judgeResponse** does but want the the timing or format of feedback to be different. In that case, you can call **responseAnalysis**, then use the information in **theMarkUp**, **theFeedBack** and **theJudgment** to do your own display work.

☞ *ERRATA EXAMPLES* # 10 (Sentence Transitions) uses **responseAnalysis()** directly.

on feedBack fdbk, append

```
feedBack "No, try again" & return

put "Subordinate clause requires subjunctive mood." & return ¬
   into var
feedBack var, "append"

feedBack theFeedBack
```

Displays the contents of **fdbk** in the current feedback card field as specified by **setFeedBackField**. If there is no current feedback field specified, then nothing is displayed.

If **append** is omitted or **empty** then **fdbk** replaces the previous field contents; otherwise **fdbk** is appended at the end of the field.

feedBack examines the contents of **fdbk** to see if line range references of form {startLine,endLine} are present (such references are generated by the **getHelp()** and **getFeedBack()** functions). If so, **feedBack** removes each reference and in its place displays lines startLine thru endLine of the variable or field which contains the current analysis text, i.e., of **theMarkUpAnsVar** or the field named by **theMarkUpAnsFName**. If **preserveFormating** is currently set to **True**, then this display is done in such a way that the text format properties of the specified lines are preserved.

Hence, contents of the global variable **theFeedback** (or any other text which might contain such line references) should always be displayed with

```
feedBack theFeedBack
```

so that text formatting can be preserved when this has been requested.

If **feedBack** encounters a reference of form {startLine,endLine,card field "fieldName"}, it gets the lines from the field referred to by the third item rather than from the current response field. The field referred to must be on the current card. This allows you to display formatted text from a variety of locations in the feedback field.

on showText txt, fieldName, append

```
showText "Some text with {30,35} embedded lines", "myField"

put "Here is some help:  {17,45}" into helpTxt
put the name of bkgnd field "helpDisplay" into helpField
showText helpTxt, helpField, "append"
```

Displays the contents of **txt** in the field named by **fieldName**. The **txt** may contain embedded line ranges of the sort generated by **getHelp()** and **getFeedBack()**. **showText** will dereference these line range pointers when it does its display work.

If **append** is omitted or **empty** then **txt** replaces the previous field contents; otherwise **txt** is appended at the end of field **fieldName**.

showText examines the contents of **txt** to see if line range references of form {startLine,endLine} are present (such references are generated by the **getHelp()** and **getFeedBack()** functions). If so, **showText** removes each reference and in its place displays lines startLine thru endLine of the variable or field which contains the current analysis text, i.e., of **theMarkUpAnsVar** or the field named by **theMarkUpAnsFName**.

If **showText** encounters a reference of form {startLine,endLine,card field "fieldName"}, it gets the lines from the field referred to by the third item rather than from the current answer field. The field referred to must be on the current card. This allows you to display styled text from a variety of locations in field **fieldName**.

If **preserveFormating** is currently set to **True**, then **showText** does its display in such a way that the text format properties of the specified lines are preserved.

When **showText** dereferences a range of lines in curly brackets, it examines that text line by line. Any line enclosed begun by a word consisting solely of the double-right-arrow "»" is assumed to contain an executable HyperCard command. **showText** will **send** the contents of such a line to the current card and remove the line from the feedback text. As soon as **showText** encounters a line which does not begin with "»", it stops searching and assumes that the rest of the feedback text is meant for display; hence, commands should always come at the beginning of a block of referenced lines.

Since answer contingent feedback is fetched by **getFeedBack()** in the form of a line range reference, and is eventually displayed by **showText**, this feature provides the user with a capability to execute code conditionally on which analysis specifications are matched. However, note these limitations:

1. Commands are sent one line at a time, so there can be only one command on a line.
2. Any input parameters of the sent message will be evaluated within the context of showText, which defines no variables of its own. Hence, any commands you execute must have constant parameters.
3. Commands are executed only when **txt** is displayed. If you defer or block the display of feedback, then any answer-contingent commands will not be done, whether or not they are display commands

Consequently, this capability will be useful mainly to call simple display commands (such as **play**, **movie** or **picture** for multi-media feedback), or your own answer-contingent handlers which might, for instance, keep track of what kinds of errors student made.

☞ *ERRATA EXAMPLES* # 8 (Complex Drill Design) uses this handler.

function dereferenceText txt

```
dereferenceText( "Here is some help: {12,28,card field "help"} )

put "Information:  {21,33}{52,59}{67,102} into var
get dereferenceText( var )
```

Computes the contents of **txt**, which may contain embedded line ranges of the sort generated by **getHelp()** and **getFeedBack()**. **dereferenceText** expands **txt** by dereferencing all of these embedded line range pointers. The result, which will of course lack any text formatting, is returned directly. No display is generated.

☞ *ERRATA EXAMPLES* # 10 (Sentence Transitions) uses this handler. This function is provided so that your handlers can examine and possibly interpret HyperCard code within answer-contingent feedback text.

function labelLines F, label, d

(In *ERRATA EXTRAS*)

```
get labelLines( "helpText", "invertedWordOrder" )

put labelLines( name of card field "grammarRef", "verbs" ) into v
```

Return pointers to a block of styled text in field named **F**. If **F** is not a full field name, it is expanded by **resolveFName()**. The block retrieved is the first block delimited by a line beginning with **label**, which should begin with delimiter character **d**, and is delimited at the end by the next line beginning with character **d**. If label does not begin with **d**, then **d** is prepended to **label**. The default for **d** is **#**.

The return is in the form of line range in curley brackets, {startLine,endLine} which can be displayed by **displayLabel**.

on displayLabel label, S, D, d

(In *ERRATA EXTRAS*)

```
displayLabel ":Subjunctive", "grammarHelpField", "feedBack", "$"

displayLabel "3rd Conj", name of card field "grammar", ¬
      name of field "helpDisplay"
```

Append a block of styled text from source field named **S** at the end of destination field named **D**. The two fields must be on the current card. If **S** and **D** are not full field names, they are expanded by **resolveFName()**. The block is the first block delimited by a line beginning with **label**, which should begin with delimiter character **d**, and is delimited at the end by the next line beginning with character **d**. If **label** does not begin with **d**, then **d** is prepended to **label**. The default for **d** is #.

☞ *ERRATA EXAMPLES* # 8 (Complex Drill Design) uses the **displayLabel()** in a handler which will display a list of named grammar help topics from a field of help text.

function resolveFName fName

```
put resolveFName( "myField" ) into fullFieldName

put the name of field "myBgField" into bfName
get resolveFName( bfName )
```

Finds the true, full field name of **fName**, which should name a field on the current card. "Full field name" means the name in the form it is returned by HyperTalk's **the name** function, i.e., card field "xxx" or bgnd field "xxx". If **fName** is already a full field name or is **empty**, then **fName** is returned unmodified. If **fName** is a short name, then the **resolveFName()** looks first for a card field, and, if there is none, for a background field of that name and returns the expanded name. If the current card contains no field named **fName**, then an error dialog is given. No display is generated.

function subst count, txt, old1, new1, old2, new2, ... old6, new6

```
get subst( "Hello, world!",1 , "Hello", "Goodbye" )
                                   ==>   "Goodbye world!"

get subst( "Il a des chats noirs.:,1 , "noir", "A", "chat", "N" )
                                   ==>   "Il a des Ns As."

get subst( "the boy and the girl:,1 , "the", "DET")
                                   ==>   "DET boy and the girl."

get subst( "the boy saw a girl.",-1 , "a ", empty, "the ", empty )
                                   ==>   "boy saw girl."

get subst( "Have a happy, happy, happy day!", 2, "happy", "nice")
                               ==> "Have a nice, nice, happy day!"
```

The "==>" indicates the value which will be in it after each call.

Returns the result of replacing the first (i.e., leftmost) **count** occurrences in **txt** of **old1** by **new1**, then the first count **occurrences** of **old2** by **new2**, etc. The inputs **txt** and at least the pair **old1, new1** must be specified. Up to six pairs of substitutions can be specified.

If **count** is not specified or is negative, then all instances of **old-i** will be replaced. If any of the **old-i** are **empty**, then no substitutions will be done. The search for each new instance of **old-i** starts right after the most recent replacement, so replacement cannot be become non-terminating.

This function is provided mainly so that literal strings can be converted to more abstract patterns, as illustrated in the examples above. It calls **stringUtilities("mSubst", txt, count, old1, new1 ... old6, new6)** to do its work.

☞ *ERRATA EXAMPLES* # 10 (Sentence Transitions) and # 19, 20, and 21 (parsing designs) utilize **subst()** and **substResp.**

on substResp count, old1, new1, old2, new2, ... old6, new6

```
substResp 1 , "Hello", "Goodbye"
substResp , "the", empty
```

A convenience handler which replaces the first (i.e., leftmost) **count** occurrences in of **old1** in theResponse by **new1**, then the first **count occurrences** of **old2** by **new2**, etc. **txt** and at least the pair **old1, new1** must be specified. Up to six pairs of substitutions can be specified.

If **count** is not specified or negative, then all occurrences of the string **old-i** are replaced by **new-i.**

theResponse is modified by this handler. It calls **subst()** to do its work. There is no display generated and no return.

on upCaseResp stripDiacrits

```
upCaseResp
upCaseResp True
```

A convenience handler which converts all the lower-case characters in **theResponse** to upper case equivalents. If **stripDiacrits** is **True**, it also removes any diacritic marks from the characters.

theResponse is modified by this handler. It calls **stringUtilities("strUpper", theResponse, stripDiacrits)** to do its work. There is no display generated and no return.

on computeAnsFit muReturn

Used internally by **judgeResponse.**

muReturn contains the information returned by the MARKUP XFCN in **theMarkUpReturnValues**, consisting of the judgment; PMATCHED, the proportion of matched words; AVEDIST, the average normalized edit distance between pairs of matched model and response words; and PNONINV, the proportion of non-inverted word sequences in the response.

computeAnsFit computes and returns a value between 0 and 1, inclusive, which reflects how well the model string and the response string fit one another overall. The formula used by compute ANSFIT to get this value is:

ANSFIT := (3*PMATCHED*(1 - AVEDIST) + PNONINV) / 4

Roughly speaking, this means that vocabulary being present and recognizable is given three times the weight of correct word order in determining how well the response fits a model. This guarantees that a long sentence which has all the required words present will not be rejected as a fit just because they are out of order.

computeAnsFit returns its value with a **return** command. **judgeResponse** calls **computeAnsFit** each time :ANSWER or :WRONG calls the MARKUP XFCN; since **computeAnsFit** is a handler, not a function, **judgeResponse** reads the return value using **the result.**

The metric given above works reasonably well in general, but you may need to modify it for some special-purpose analyses. For instance, if your feedback concerns word order, then you will not want an :ANSWER or :WRONG specification to be satisfied unless all of the words required are actually present. In this case, you may want to modify the computation of ANSFIT like this:

```
ANSFIT      :=     0,                                  IF PMATCHED < 1
                   ( 3*( 1 - AVEDIST ) + PNONINV ) / 4,  OTHERWISE
```

This excludes a fit unless all the words required are present, and guarantees that anything you say about word order will refer to words which are actually present in the student's response. (Of course, you would not want to do this if the feedback concerns missing words.)

☞ *ERRATA EXAMPLES* # 8 (Complex Drill Design) replaces the default **computeAnsFit** handler with one like this.

In order to activate **computeAnsFit**, **judgeResponse** executes

```
send "computeAnsFit theMarkUpReturnValues" to this card
```

Hence if you put your own handler named **computeAnswerFit** in the card, background, or stack script of your stack, it will preempt the default version in the *ERRATA* stack. Remember that your handler must return a number between 0 and 1, inclusive.

on matchingIsOk

Used internally by **judgeResponse.**

Uses the number of satisfied and unsatisfied :MATCHes to compute an evaluation of the response as OK or NO relative to :MATCH-type specifiers. The default form is

```
( okCount > 0 ) AND ( noCount = 0 )
```

where **okCount** is the number of satisfied :MATCH statements having an "ok" or "okStop" polarity and **noCount** is the number which have a "no" or "noStop" polarity. This means that each positive polarity match is sufficient to identify a correct answer and that each negative polarity match is sufficient to identify a wrong answer. This form of combination is appropriate for the usual kinds of error analysis.

☞ It is an error in your analysis specifications if the same string can satisfy both a positive and a negative polarity match.

This handler returns a value of **True** (response OK) or **False** (response wrong) in **the result.** This handler is used by **responseAnalysis** (via **judgeResponse**), which calls it by sending the message "matchingIsOk" to the current card.

You can substitute a different evaluation computation for the default one by installing your own **matchingIsOK** handler in you card or stack script. This handler should have no input parameters, should consult the global variables **okCount** and **noCount** and/or whatever other globals are appropriate, and return **True** or **False**.

☞ *ERRATA EXAMPLES* # 9 (Keyword-driven dialog) installs a revised version of **matchingIsOk** more suitable for keyword matching..

on copyText chnk1, p1, p2, cb1, f1, c1, s1, chnk2, q1, q2, cb2, f2, c2, s2

```
copyText "line", 15, 25, "cardfield", "myField", "myCard", ¬
    "myHD:myFolder:myStack", "char", 230, 235, "bkgndfield", ¬
    "bField", "aCard",  "aVol:aFolder:aStack"

copyText "item", 1, 10, "c", "fOne", "cOne", "sOne", ¬
   "item", 3,, "b", "fTwo", "cTwo", "sTwo" --Insert before item 3.

copyText "line", 1, 10, "c", "fOne", "cOne", "sOne", ¬
   "char", 0,, "b", "fTwo", "cTwo", "sTwo"  --Insert at front.

copyText "item", 1, 10, "c", "fOne", "cOne", "sOne", ¬
   "char", 99999,, "b", "fTwo", "cTwo", "sTwo"  --Append at end.
```

General facility for copying text from one field to another while preserving all the text styling properties (font, size, boldfacing, etc.). The fields can be absolutely anywhere, but the source and destination stacks must be modifiable. **copyText** works by going to the source field, copying the specified range of chunks into the clipboard, then going to the destination field and copying the clipboard into the specified range of chunks. After this is done, it returns to the orignal card from which **copyText** was called. Because its generality makes it slow, **copyText** may take several seconds to copy text. The current selection will be lost.

chnk1:	Type of chunks: **"line"**, **"item"**, **"word"** or **"char"**.
p1:	Number of first chunk in source range. If **empty**, set to 1.
p2:	Number of last chunk in source range. If **empty**, set to 99999.
cb1:	Type of source field, either "card field" or "background field" (actually, only the first character "c" or "b" need be specified).
f1:	Short name of source field.
c1	Short name of source card. If **empty**, use current card.
s1	Name of source stack (including path information, if source stack is not in the default path). If **empty**, use current stack.
chnk2:	Type of destination chunks: **"line"**, **"item"**, **"word"** or **"char"**.
q1:	Number of first destination chunk in range. If **empty**, set to 1.
q2:	Number of last destination chunk in range. Set to **q1 - 1** if omitted or **empty**, to allow for insertion just before destination chunk number **q2**.
cb2:	Type of destination field, either "card field" or "background field" (actually, only the first character "c" or "b" need be specified).
f2:	Short name of destination field,
c2	Short name of destination card If **empty**, use current card.
s2	Name of destination stack (including path information if destination stack is not in the default path). If **empty**, use current stack.

To insert just before chunk q1, leave **q2** empty. To insert at the front of destination stack, specify **chnk2** = **"char"** and **q1** = 0. To append at end of destination field, specify **chnk2** = **"char"** and **q1** = 99999 or some other sufficiently large value. To copy the whole source text specify **chnk1** = **"char"**, **p1** = 1, **p2** = 99999, **chnk2** = **"char"**, **q1** = 1, **q2** = 99999.

While **copyText** is going from one location to another, **lockMessages** is set to **True** so that **openStack, openCard, closeStack, closeCard,** messages will not be sent. All stack and field properties are restored after copying is completed.

on quickCopy chnk1, p1, p2, f1, chnk2, q1, q2, f2,

```
quickCopy "line", 15, 25, the name of card field "fromField", ¬
    "char", 50,, the name of bkgnd field "toField"
```

Facility for copying text from one field to another while preserving all the text format properties (font, size, boldfacing, etc.). Both the source field and destination field must be on the current card. The destination field must not be a shared-text background field. Because of its limitations, **quickCopy** works faster than **copyText**. **quickCopy** copies the specified range of source field chunks into the clipboard, then copies the clipboard into the specified range of destination field chunks.

chnk1:	Type of chunks: **"line"**, **"item"**, **"word"** or **"char"**..
p1:	Number of first chunk in source range. If **empty**, set to 1.
p2:	Number of last chunk in source range. If **empty** set to 99999.
f1:	Full name of source field, as given by HyperCard function **the name**. Note that this includes specification of whether the field is a card or bkgnd field.
chnk2:	Type of destination chunks: "line", "item", "word" or "char".
q1:	Number of first destination chunk in range. If **empty**, set to 1.
q2:	Number of last destination chunk in range. Set to **q1 - 1** if omitted or empty, to allow for insertion just before destination chunk number **q2**.
f2:	Full name of destination field, as given by HyperCard function **the name**. Note that this includes specification of whether the field is a card or bkgnd field.

To insert at front of the destination field, specify **chnk2** = **"char"** and **q1** = 0. To append at end of the destination field, specify **chnk2** = **"char"** and **q1** = 99999 or some other sufficiently large value. To copy the whole source text specify **chnk1** = **"char"**, **p1** = 1, **p2** = 99999, **chnk2** = **"char"**, **q1** = 1, **q2** = 99999.

function delimiterTable tableType, txt, delimiter

(XFCN)

```
get delimiterTable( "charPointers", card field "myField", "#" )
put delimiterTable( "linePointers", myVar, "##" ) into list
```

Compiles a comma-separated list of delimiter locations--information which is useful in setting up drill materials for presentation.

tableType is the type of pointer required. if it is set to **"charPointers"**, then each entry in the list will be the character position of one of the delimiter characters in txt. Following HyperCard delimiter conventions, the **txt** is assumed to lack leading and trailing delimiters (i.e., the "delimiters" are actually separators), but a "virtual" delimiter just before the text and just after the text are added to make it easier when using the pointer list to retrieve the range of text (characters or lines) between two delimiters.

If **tableType** is **"linePointers"**, then the list will contain the line numbers of delimiter lines within txt. A delimiter line is one which has **delimiter** as its first character. In keeping for HyperCard conventions for delimiters, is assumed that the beginning and end of **txt** will lack delimiter lines, so "virtual" delimiter lines are added at those positions. These virtual delimiters are reflected in the first and last list entries, as with the character pointer table.

delimiter is any character which delimits blocks of texts, such as the items of a drill.

The Nth block of characters or lines (i.e., text between the Nth and N+1st delimiter) is specified by

```
        char ( item N of list + 1 ) to ( item N of list - 1 ) of txt
```

or

```
        line ( item N of list + 1 ) to ( item N of list - 1 ) of txt
```

☞ If txt already has delimiters at the beginning and end, you should remove the first and last entries of the returned list.

ERRATA EXAMPLES # 8 (Complex Drill Design) uses **delimiterTable** and several handlers of its own to manage a multi-item drill with styled text.

function mapKeys str, keyMap, inverse, delimiter

(XFCN)

```
        mapKeys( "abcd", "a1 b2 d4" )  ==> 12c4
        mapKeys( "x123y", "A1 B2 C3 D4", "inverse" )  ==> xABCy
        mapKeys( "abcd", empty )  ==> abcd
        mapKeys( "c", "aX bY cZ" ) ==> z
        mapKeys( "1234", "1a 3b 2b 4d" )  ==> abbd
        mapKeys( "a b c", "aA/bB/cC/ *",, "/" ) ==> A*B*C*
```

str is a string of 253 or less characters. **keyMap** is a string of form "aA bB cC.... nN", which means that all instances of "a" in **str** should be replaced by "A", all instances of "b" should be replaced by "B", etc. If a charcter is not in the set a, b, c, ... n, then it is not replaced within **str**. If inverse is **"inverse"**, then the mapping goes in the opposite direction. If a char appears several times in a, b, c, .. n, then the leftmost substitution will be done. One use of this function is to remap the keyboard by substituting and forwarding keypresses within a **keyDown** handler. If the second parameter is absent, a default map is used appropriate for a keyboard for the Hebrew Tamar font.

Character pairs must be separated by one or more **delimiter** characters (the default **delimiter** is space. The **delimiter** character cannot appear in any character pair. If you need to map the space character, you may specify as the value of **delimiter** some other character that is used to separate character pairs in **keyMap**. When **delimiter** is omitted, the space character is used.

function stringUtilities operation, p1, p2, ...

(XFCN)

stringUtilities() provides a number of operations useful for matching and formatting strings. **operation** is the type of operation to be done. The number and meaning of the remaining input parameters **p1, p2,...** depend on which **operation** is specified:

stringUtilities("reverse", str)

```
stringUtilities( "reverse", "Hello World!" )
                        ==> "!dlroW olleH"
```

Returns the reverse of **str**, which must be 255 or less characters long.

stringUtilities("removeChars", str, charToRemove, removeExtraSpaces)

```
stringUtilities( "removeChars", " Hello ,.. World!? ", ",.!?", True )
        ==> "Hello world"
```

Removes all instances of every character in the string **charsToRemove** from str, which must be 255 or less characters long. If **removeExtraSpaces** is **True**, then any extra spaces, including leading and trailing spaces, will be removed also. Return is an altered copy of **str**.

stringUtilities("findString", txt, pattern, N, offset, compType, ignoreCase, ignoreDiacrits)

```
stringUtilities( "findString", "abcABCabc", "ab", 2, 0, True, False )
        ==> 7   -- Char position.
```

Searches rightward through the text in **txt** to find the Nth instance of **pattern**, beginning at the first char past **offset** (numbering of chars starts at the left with 1). Return is the character position in **txt** where the first character of **pattern** was found. If offset is not given, seach starts at character 1.

compType controls the way in which characters are compared: 0 = strict ASCII identity; 1 = strict international (diacritics and secondary order taken into account); and 2 = weak international (diacritics and "secondary ordering" ignored). Default is 0. For the ASCII identity method only, if **ignoreCase** and/or **ignoreDiacritics** are **True**, then case and/or diacritic marks will be ignored when matching characters. For the other **compType** values, these parameters are ignored.

stringUtilities("findStringL", txt, pattern, N, offset, compType, ignoreCase, ignoreDiacrits)

```
stringUtilities( "findStringL", "abcABCabc", "ab", 2, 0, False, False )
        ==> 4    -- Char position.
```

Searches leftward through the text in **txt** to find the Nth instance of **pattern**, beginning at the first char past **offset** (numbering of chars starts at the left with 1). Return is the character position in **txt** where the first character of **pattern** was found. Character position refers to standard numbering, beginning with 1 at left end of **txt**. If **offset** is not given, seach starts at rightmost character of **txt**.

compType controls the way in which characters are compared: 0 = strict ASCII identity; 1 = strict international (diacritics and secondary order taken into account); and 2 = weak international (diacritics and "secondary ordering" ignored). Default is 0. For ASCII identity method only, if **ignoreCase** and/or **ignoreDiacritics** are **True**, then case and/or diacritic marks will be ignored when matching characters. For the other **compType** values, these parameters are ignored.

stringUtilities("fontInfo", fontName, fontSize)

```
stringUtilities( "fontInfo", "Courier", 12 )
              ==> 10,2,0,11  -- Ascent, descent, leading, maxwidth.
```

Returns information about the font named fontName, in size **fontSize**. Return is a list of 4 comma-separated items: ASCENT,DESCENT,LEADING,MAXWIDTH, which have the usual Macintosh Font Manager definitions (see Inside Macintosh, IV, 27-45).

stringUtilities("strWidth", txt, fontName, fontSize, style)

```
stringUtilities( "strWidth", "abc", "Courier", 12, "plain" )
              ==> 21  -- Width in pixels.

stringUtilities( "strWidth", "abc", "Courier", 12, "bold,underline,italic" )
              ==> 24  -- Width in pixels.
```

Returns the width in pixels of the text in **txt**, when it is presented in font **fontName**, at pointsize **fontSize**, in style **style**.

stringUtilities("findLineBreak", s, displayWidth, txtFont, txtSize, txtStyle)

```
stringUtilities( "findLineBreak", "aa bb cc dd", 20, "Courier", 12, "plain" )
              ==> 21  -- Width in pixels.
```

Finds first linebreak in string s at the same place the Mac system finds it.

s	A string of monostyle, monopscript text (255 chars or less).
displayWidth	Width of display space, in pixels (usually the width of a display field).
txtFont	Font in which the text will be displayed
txtSize	Text size in which the text will be displayed
txtStyle	A string with comma-separated list of style values (plain, italic, bold, etc.) in which the text will be displayed.

Break characters are the characters after which a line break is legal. **"findLineBreak"** uses the set which appear to be default for the Macintosh system's linebreak algorithm for latin scripts: " -+/*&|\<>=≠". Note that **space** and hyphen are in this set.

Return is a comma-separated list of three positive integers: S1,LASTDISPLAYED,S2

S1:	Points to the first char of a run of 1 of more break characters that ends the display line.
LASTDISPLAYED:	Points to the last char which there is actually room to display on the line (may be last character before the run of breaks, or one of the break characters).
S2:	Points to the first character past the run of breaks that ends the display line.

When doing display, a run of break characters is always appended to the preceeding non-break characters; hence break charactes can never begin any line but the first Here is an example text string with the possible locations of S1 and S2 marked:

```
AAAAAA     BBBBBB-CCCCC     DDDDDD
      |    |     .||   |   |      |
      7   12   18 19  24   28     34
     s1   s2   s1 s2  s1   s2     s1,s2
```

To find out how much of this text string will fit into a single line of a field 40 pixels wide when displayed in plain 12-point Courier font, call

```
stringUtilities("findLineBreak", "AAAAAA      BBBBBB-CCCCC     DDDDDD", ¬
      40, "Courier", 10, "plain" )
      --> 7,6,12
```

Hence, the first displayed line will be char 1 to 6 of the text. Chars 1 - 11 should be consumed, so the remaining text to format will begin with character 12.

☞ Used internally by the **showMarkUp** handler. This XFCN has somewhat the same functionality as the STYLEDLINEBREAK function discussed in Inside Macintosh: Text, 5-24, which also explains break characters, etc.

GLOBAL VARIABLES

The range of possible values for each global is given just below its name in **boldface**. Some description of the use of the value, the default value, and the ways in which the variable can normally change value (including the handlers which set the variable) follow.

For globals which effect communication with the MARKUP XFCN, no attempt is made to give an exhaustive description of the format or significance of the data they hold; the user should consult Hart (1994) where full details are provided.

adjustNeeded

True, False

MARKUP XFCN parameter. The quality of the word order markup is increased if this is set to True, but MARKUP will take a bit longer. The default is **True**.

Set by **setMarkUpJudgingParams.**

anyOrderOk

True, False

MARKUP XFCN parameter which determines whether word order errors will cause a response to be judged wrong. The default is **False**.

Set by **setMarkUpJudgingParams.**

capFlag

"Exact_case", "Autbors_caps", "Ignore_case"

MARKUP XFCN parameter which determines whether capitalization errors will cause a response to be judged wrong. If **"Exact_case"**(the default), any deviation in case will cause a wrong judgment. If **"Ignore_case"**, then any differences in case are ignored and will not lead to a wrong judgment. If

"Authors_caps", then the student must have upper case everywhere that the author has it; in other positions, case is ignored.

Set by **setMarkUpJudgingParams.**

debugNeeded

True, False

MARKUP XFCN parameter. This parameter is useful for development purposes, since, if set to **True**, it allows the XFCN PASCAL code to return debugging information to HyperCard in the global variable **theMarkUpDebug.** This is needed because the usual PASCAL debugging tools can't be used within XFCNs. Default is **False.**

Set by **setMarkUpJudgingParams.**

extraWordsOk

True, False

MARKUP XFCN parameter which determines whether extra words not specified in the :ANSWER specification will cause a response to be judged wrong. Default is **False,** so that extra words cause a NO judgment.

Set by **setMarkUpJudgingParams.**

markupFont

Font name

Contains name of font which will be used as default font for the markup field.

Set directly by **put.**

☞ *ERRATA EXAMPLES* # 6 (Cyrillic Font) resets this variable.

markUpMapsNeeded

True, False

MARKUP XFCN parameter which determines whether MARKUP XFCN will compute and return response-model pairings of words as additional information in **theMarkUpMaps.**

Set by **setMarkUpJudgingParams.**

misspellOk

True, False

MARKUP XFCN parameter which determines whether spelling errors will cause a response to be judged wrong. If **True**, then spelling errors will not cause a response to be judged wrong. Default is **False**.

Set by **setMarkUpJudgingParams.**

muField

Full name of a HyperCard card or background field

Contains full name of the markup field, where the graphical markup is displayed.

Set by **setUpMarkUp**.

noCount

Integer ≥ 0

Contains number of :MATCH, :NUMBER, or :DO specifications with a "no" or "noStop" polarity which have been satisfied during the current analysis.

Set by **judgeResponse** (via **responseAnalysis**), **judgeMatch** (via **matchMR**)

noFeedBack

Styled text

Contains a concatenation of the feedback text from all the :MATCH, :NUMBER or :DO specifications which have been satisfied during the current response analysis and which have a negative polarity ("no" or "noStop").

Set by **judgeResponse** (via **responseAnalysis** via **matchMR**), **judgeMatch** (via **matchMR**)

okCount

Integer ≥ 0

Contains number of :MATCH, :NUMBER, or :DO specifications with an "ok" or "okStop" polarity which have been satisfied during the current analysis.

Set by **judgeResponse** (via **responseAnalysis**), **judgeMatch** (via **matchMR**)

okFeedBack

Styled text

Contains a concatenation of the feedback from all the :MATCH, :NUMBER or :DO specifications which have been satisfied during the current response analysis and which have a positive polarity ("ok" or "okStop").

Set by **judgeResponse** (via **responseAnalysis**), **judgeMatch** (via **matchMR**)

parameterDisplayNeeded

empty, "v", "b", "d", "c", "h", "p", "w", "f", "s", "m"

MARKUP XFCN parameter which determines whether the MARKUP XFCN will return information about one of the judging parameters in **theMarkUpParamDisplay**. Default is **empty** (no information returned). For meaning of the remaining parameters see Hart (1994).

Set by **setMarkUpJudgingParams.**

preserveFormating

True, False, empty

Determines whether **getHelp()** and **getFeedBack()** and **feedBack** will try to preserve the text formatting (font, style, size) of contingent feedback text. Default is **empty** (same as False).

Set by **setPreserveFormating.**

responseField

Name of a HyperCard card or background field

Contains name of the current response field. **judgeResponse** will give an error message if you attempt to do a response analysis without setting this variable. **resolveFName()** is used to determine the full field name.

Set by **activateField** and **setUpMarkUp.**

responseFont

Font name

Contains name of font which will be used as default font for current response field.

Set directly by **put.**

☞ *ERRATA EXAMPLES* # 6 (Cyrillic Font) manipulates this variable.

runTogetherNeeded

True, False

MARKUP XFCN parameter which determines whether run-together word errors will be identified. If **runTogetherNeeded is False**, run-together words will simply be marked as unidentified words, but MARKUP will run faster. Default is **True.**

Set by **setMarkUpJudgingParams.**

shortCut

True, False

MARKUP XFCN parameter which determines how edit distance between words is computed. **True** causes the edit distance of dissimilar words to be set to infinity; **False** forces a computation of the actual edit distance in every case. If **shortCut is False**, MARKUP will run more slowly. Default is **True**.

Set by **setMarkUpJudgingParams.**

spellingOnlyNeeded

"x", "r", "p", empty

MARKUP XFCN parameter which determines whether input strings to the MARKUP XFCN will be treated as sequences of words (i.e., sentences) or as sequences of characters (i.e., words). If **"x"** or **empty**, then input parameters are treated as sentences. If **"r"** or **"p"** then the inputs are treated as words, and all characters, including spaces and punctuation, are treated as alphabetic. Hence, the edit distance between the two exact strings is computed. **"r"** returns the raw graphic markup resulting from this process; **"p"** returns a prettied up form suitable for display. The default is **"x"**, which is the same as **empty**.

This parameter affects the return values in **theMarkUpReturnValues.**

Set by **setMarkUpJudgingParams.**

☞ *ERRATA EXAMPLES* # 18 (Spelling/Dictation) sets this variable to "r" in order to get back the raw markup needed for a custom-made spelling markup display.

suppressOkNo

"m", "f", "mf", "fm", empty

Used by judgeResponse to determine where to display the current OK/NO messages. The default value of empty causes display in both the feedback field and the markup field (if there is no graphical markup). If "m" is present in suppressOkNo, then display in the markup field is suppressed. If "f" is present, then display in the feedback field is supressed. If the value is "mf" or "fm", then the messages are not displayed. If the value is empty, then display is done in both places.

☞ *ERRATA EXAMPLES* # 10 (Sentence Transitions) resets this variable.

theBestFitThreshold

A decimal number between 0 and 1.0

Determines the interpretation of the "answer fit" computed between a model and response. If answer fit is less than **theBestFitThreshold**, then the model and response are considered NOT to match at all; otherwise they are considered to match (although perhaps imperfectly). The default value is 0.70.

Use **put** to set this variable.

theFeedBack

Text with embedded line references

Contains the feedback generated by the most recent response analysis.

Set by **judgeResponse** (via **responseAnalysis**), **judgeMatch** (via **matchMR**), and **judgeAnswer**.

theFeedBackFName

Full name of a HyperCard card or background field, empty

Contains full name of the current feedback field (e. g., card field "myField" or bkgnd field "myField"). The **feedBack** handler uses this as the destination for the display it generates. If **theFeedBackFName** is empty, no feedback display will be generated by **feedBack**.

Set by **setFeedBackField**.

theJudgingHandler

Name of a HyperCard Handler

Contains name of the current judging handler. The default value is "**judgeResponse**".

Set by **setJudgingHandler**.

theJudgment

True, False

Contains the evaluation of the current response as computed by the current analysis.

Set by **judgeResponse** (via **responseAnalysis**), **judgeMatch** (via **matchMR**), and **judgeAnswer** (via **findBestAns**).

theMarkUp

A string of graphical markup characters

Contains graphical markup corresponding to the best fit :ANSWER or :WRONG specification which was matched during the most recent response analysis. If there was no :ANSWER or :WRONG specification, or if none was matched, then **theMarkUp is empty**.

If something went wrong inside the MARKUP XFCN (usually too many letters in a word or too many words in a response or an :ANSWER or :WRONG specification), then **theMarkUp** will contain, instead of a markup string, an error message whose first character is "%". The remainder of the message will indicate the problem. See Hart (1994) for details.

Set by **judgeResponse** (via **responseAnalysis** via MARKUP XFCN), and **judgeAnswer** (via **findBestAns** via MARKUP XFCN).

theMarkUpAns

A response analysis text

Contains (unformatted) text which will be used as analysis text for the current response analysis. The text contains analysis specifications (:ANSWER/:WRONG/:MATCH/ :NUMBER/:DO/:HELP) with optional feedback for each.

Set by **setCorrectAnswer**, **activateField**.

theMarkUpAnsFName

The full name of a HyperCard card field or background field, empty

Contains the name of a field of data which either is coextensive with the current analysis text or contains the current analysis text somewhere within it. If the data field was specified by a literal value or variable rather than a field name, then **theMarkUpAnsFName** will we **empty**.

Set by **setCorrectAnswer**, **activateField**.

theMarkUpAnsL1

Integer > 0

Contains the number of the first line of the analysis text within the data field or variable which contains the analysis text. The name of the field will be in **theMarkUpAnsFName**, if there is such a field; the variable, which always exists, is **theMarkUpAnsVar**. If the field exists, it will have the same contents as the variable, but with text formatting.

Set by **setCorrectAnswer** and **activateField.**

theMarkUpAnsL2

Integer > 0

Contains the number of the last line of the analysis text within the data field or variable which contains the analysis text. The name of the field will be in **theMarkUpAnsFName**, if there is such a field; the variable, which always exists, is **theMarkUpAnsVar**. If the field exists, it will have the same contents as the variable, but with text styling.

Set by **setCorrectAnswer** and **activateField.**

theMarkUpAnsVar

Text

Contains a copy of the data field or variable which contains the analysis text. The name of the field will be in **theMarkUpAnsFName**, if there is such a field;. If the field exists, it will have the same contents as the variable, but with text formatting.

Set by **setCorrectAnswer** and **activateField.**

theMarkUpCharInfo

Formatted data, empty

Contains data determining the way the MARKUP XFCN will interpret different characters during judging. If **empty** (the initial value), default judging parameter values are used. See Hart (1994) for details.

Set directly by **put.**

theMarkUpDebug

Formatted data

The MARKUP XFCN puts debugging information into this variable, if such information has been requested. See Hart (1994) for details.

Set by **markUpUsingParams()** (via MARKUP XFCN).

theMarkUpMaps

Formatted data

The MARKUP XFCN puts information about the way in which model words and response words are paired, as well as information about the locations of response words, into this variable, if such information has been requested. Line 1 gives the number of the model word which corresponds to each response word, with 0 indicating no correspondent. Line 2 gives the number of the response word which corresponds to each model word, with 0 indicating no correspondent. Line 3 gives the starting character number of each response word withing the unmodified response string. See Hart (1994) for details.

Set by **markUpUsingParams()**, via MARKUP XFCN.

theMarkUpParamDisplay

Formatted data

The MARKUP XFCN puts information about the current judging parameters into this variable, if such information has been requested. See Hart (1994) for details.

Set by **markUpUsingParams()**, via MARKUP XFCN.

theMarkUpParameters

Formatted data, empty

The MARKUP XFCN uses the information in this global to control the way in which the markup analysis, particularly the spelling analysis is done. If **empty** (the initial value), default judging parameter values are used. See Hart (1994) for details.

Set directly by **put**.

theMarkUpPunctuation

A string of characters, empty

The string in this variable specifies the characters which the MARKUP XFCN will consider to be punctuation. These characters will be removed from the student's response before it is judged. If you want to judge some of the punctuation marks, you will need to modify this string so that those marks are removed from the set of punctuation marks.

If **theMarkUpPunctuation** is **empty** (the initial value), then a default value of ".,:;<>?!{}()[]<>?" **& return & space** will be used for the punctuation.

To change this variable, **put** a new string directly into **theMarkUpPunctuation**. The initial, standard, set of punctuation marks is restored by **restorePunctation** and also by **restoreMarkUpDefaults**.

theMarkUpReturnValues

A list of 4 comma-separated items

Contains values which the MARKUP XFCN computes during the process of marking up a response.

Set by **markUpUsingParams(model, response)** (via MARKUP XFCN). The nature of the values returned depends on the setting of the MARKUP XFCN input parameter **spellingOnlyNeeded**.

If the value of **spellingOnlyNeeded** was **"x"** (the default case), then the return is a list of 4 comma-separated values JUDGMENT,PMATCHED,PNONINV,AVEDIST:

JUDGMENT: **True**, if the **model** matched the **response**, otherwise **False**.

PMATCHED: Proportion of words which were matched (between 0.0 and 1.0)

PNONINV Proportion of word order non-inversions (between 0.0 and 1.0)

AVEDIST: Averaged normalized edit distance (proportion of misspelling) between matched words. (between 0.0 and 1.0)

The response analysis package uses these values to compute the fit between the model and the response.

If the value of **spellingOnlyNeeded** was **"r"** or **"p"**, then **theMarkUpReturnValues** contains a list of two comma-separated items, RAWEDITDIST, NORMALIZEDEDITDIST:

RAWEDITDIST: A positive integer representing the minimum total cost of edit operations needed to transform response into model.

NORMALIZEDEDITDIST: A number between 0.0 and 1.0. It is equal to RAWEDITDIST divided by a normalizing factor and represents the similarity of the two strings.

Set by **markUpUsingParams()** (via MARKUP XFCN).

theMarkUpSymbols

A string of characters, empty

The string in this variable specifies the characters which the graphical markup uses to indicate various kinds of errors. If it is empty (the initial value), then a default value of "+-~XΔ«x\=><[" is used for the symbols. The significance of the character at the various positions is:

+	addcap
-	dropcap
~	accenterr
X	extrawd
Δ	missingwd
«	movewd
x	extrltr
\	missingltr
=	substituteltr
>	transltr1
<	transltr2
[	runonwd

Specific default characters can be changed to others by executing **changeMarkUpSymbol**. The default values will be restored by **restoreMarkUpSymbols** or **restoreMarkUpDefaults**.

thePatternFeedBack

Text with embedded line references

A concatenation of feedback generated by :MATCH, :NUMBER and :DO specifications which were satisfied during the most recent response analysis.

If the **theJudgment** was **True**, then only the positive polarity ("ok" or "okStop") specifications will contribute; if the **theJudgment** was **False**, then only the negative polarity specifications ("no" or "noStop") will contribute.

Set by **judgeResponse** (via **responseAnalysis**) and **judgeMatch** (via **matchMR**)

theResponse

Text

Contains a (possibly modified) copy of the response which the student typed into the current response field. This is the string which **judgeResponse** uses to make its judgments.

The user may want to operate on this string to, for example, reduce all the characters to a uniform case suc as uppercase, remove extra spaces or unwanted characters such as punctuation marks, strip off diacritics, or do arbitrary string substitutions using **subst()**. Since MARKUP deals with spaces, case variations, etc., automatically, these operations are useful mainly to get a regularized response to use with the MATCH XFCN.

☞ If you remove or insert characters into **theResponse**, it will no longer match what is displayed in the response field, and any markup or match hiliting done using theResponse as data will not display correctly. Hence, :DOs which perform string substitutions should be placed after all :ANSWER and :WRONG specifications and before all :MATCH , :NUMBER and other :DO specifications.

Set by **judgeResponse** (via **responseAnalysis**) and **judgeMatch** (via **matchMR**).

userKeyHandling

True, False, empty

Determines whether the **keyDown** handler in the *ERRATA* stack will be active or not. If **userKeyHandling** is **False** or **empty**, the default handler will be active; if **True** it will not be. This variable should normally be **True** only if the user has written a **keyDown** handler of his own

When the default handler when it is shut off, keyDown messages are passed up the response hierarchy in the normal way so that, e. g., RETURN will not cause judging to take place in the response field but will simply insert an end-of-line character.

To change the value of this variable, execute **defaultKeyHandling**.

wordMarkUpNeeded

True, False, empty

MARKUP XFCN parameter which determines whether word order errors will cause a response to be judged wrong. **empty** is equivalent to **True**. The default value is **True**.

Set by **setMarkUpJudgingParams.**

INSTALLATION AND TECHNICAL INFORMATION

The *ERRATA* software is in a folder named ERRATA which has this file structure:

ERRATA	HyperCard stack
ERRATA EXTRAS	HyperCard stack
ERRATA EXAMPLES	HyperCard stack
XCMDS	folder
delimiterTableXFCN	folder
delimiterTable.π	THINK PASCAL project file
delimiterTable.p	THINK PASCAL source code
delimiterTable.o	THINK PASCAL object code
mapkeysXFCN	folder
mapKeys.π	THINK PASCAL project file
mapKeys.p	THINK PASCAL source code
mapKeys.o	THINK PASCAL object code
stringUtilitiesXFCN	folder
stringUtilitiesXFCN.π	THINK PASCAL project file
stringUtilitiesXFCN.p	THINK PASCAL source code
stringUtilitiesXFCN.o	THINK PASCAL object code
markupXFCN	folder
markupXFCN.π	THINK PASCAL project file
markupXFCN.p	THINK PASCAL source code
markupXFCN.o	THINK PASCAL object code
matchR	folder
matchR.π	THINK PASCAL project file
matchR.p	THINK PASCAL source code
matchR.o	THINK PASCAL object code

Installation of *ERRATA* is trivial: simply copy the *ERRATA* folder to any convenient location on your hard disk. You can even use *ERRATA* directly from diskette, although it will be slow. Or you can merge the *ERRATA* stacks and folders into some other folder. It is best to keep all *ERRATA* materials in the same folder, however.

ERRATA requires the following hardware and software:

A Macintosh machine, Quadra-class or better.
Macintosh Finder (System) 7.1 or better
HyperCard 2.1 or better

ERRATA was developed on a Quadra 700 using this software configuration. It should run on other Macintosh machines with less clock speed, but the user may experience appreciable slowdown, particularly when doing display, since the handlers which do copying of styled text are quite computation intensive. (Setting the *ERRATA* parameter **preserveFormatting** to False will minimize this effect.) The primitive MARKUP, MATCH, STRINGUTILITY and DELIMITERTABLE XFCNs should be adequately fast on any Macintosh.

All XFCN resources are installed in the *ERRATA* stack. Each XFCN is structured as a THINK PASCAL™ 4.01 project. The PASCAL source code, the object code, and the THINK PASCAL project file for each XFCN is distributed in the folder named XFCNs. The XCMD file is not needed by *ERRATA*, but is provided for your information. (To work with the projects, you will have to rebuild them substituting valid pathnames for your machine.)

The compiled MARKUP XFCN project occupies a bit more than 42700 bytes of space; the MARKUP XFCN itself occupies about 22474 bytes. This is near the limit of the allowed size for HyperCard code resources. XFCNs borrow their space from the HyperCard stack, so if MARKUP is run in recursive or other deeply embedded contexts, there may not be sufficient stack space. Running the MARKUP XFCN in such a situation will cause the stack to overflow into the heap and will most likely cause a hard system crash, if not immediately, then soon thereafter, or at latest during exit from HyperCard. To guard against this, the **markUpUsingParms()** function checks to make sure that there are at least 28500 bytes free on the HyperCard stack; if not, MARKUP is not run and an error dialog appears. (Since MARKUP's word-order-error algorithm is recursive, space requried is somewhat sensitive to the number of words in model and response, but 28500 should be sufficient to run with the maximum of 18 words.) If you call the primitive MARKUP XFCN, you should first use the HyperTalk function **the stackSpace** to assure that this much stack space is available.

To conserve stack space, some large MARKUP array structures have been put into dynamic memory. The Mac Toolbox functions NEWPTR() and DISPOSPTR() are used to allocate and deallocate this memory, which amounts to about 24K of space. If this much heap memory is not available, MARKUP aborts and returns the error message "%Couldn't get matrix memory."

The STRINGUTILITY XFCN is also large and is not guarded in any way, so care should be taken to check stack space before calling it. It does not allocate heap memory and has no recursive routines.

The user should be aware of the following limitations on the MARKUP XFCN:

> Versions through 3.0 will not work properly with the Macintosh 16-bit char representation, i.e., with the language extensions.
> Maximum number of letters in a single word: 22
> Maximum number of words in model (including synonyms but excluding ignorables): 18
> Maximum number of words in response: 18
> Maximum number of characters in model: 255
> Maximum number of characters in response: 255

These limitations are not intrinsic to the MARKUP algorithm, but are imposed by the fact that the entire MARKUP function has to run in the limited space provided by the HyperCard stack. The MATCH XFCN has the following limitations:

> Maximum characters in pattern: 255
> Maximum characters in response: 255

To determine which version of *ERRATA* you have, inspect the first line of the stack script of the *ERRATA* or *ERRATA EXTRAS* stacks. To determine the version of the MARKUP, MATCH, STRINGUTILITIES or DELIMITERTABLE XFCNs, execute the function without input parameters from the message window, e.g.: **Markup()**. A version string will be returned.

REFERENCES

Brown, J. & Burton, R. (1978) Diagostic models for procedural bugs in basic mathematical skills. *Cognitive Science*, 2, 155-191.

Hart, Robert S. (1989) *Algorithms for the Dynamic Identification of Spelling and Word Order Errors in Student Responses*, Technical Report No. LLL-T-15-89. Urbana, IL: Language Learning Laboratory, University of Illinois at Urbana-Champaign.

Hart, Robert S. (1994) *Improved algorithms for Identifying Spelling and Word Order Errors in Student Responses*. Technical Report No. LLL-T-15-94. Urbana, IL: Language Learning Laboratory, University of Illinois at Urbana-Champaign.

APPENDIX 1: LISTING OF *ERRATA* HANDLERS

```
-- ERRATA Version 1.0

-------------------------

-- SETUP HANDLERS.

-------------------------


on setJudgingHandler handlerName

  -- Specify the name of the handler which will do the answer judging.

  global theJudgingHandler

  put handlerName into theJudgingHandler

end setJudgingHandler


on activateField C, L1, L2, keepResp

  -- Convenience handler which combines effects of SETMARKUP and SETCCRRECTANS.
  -- When executed in an OPENFIELD handler, prepares the field
  -- to accept and mark up a response using C, L1, L2 as the correct answer.
  -- C, L1, L2      = Correct answer or analysis text
  -- KEEPRESP       = If TRUE, current contents of RFIELD are left alone, else erased.

  setCorrectAnswer C, L1, L2
  setUpMarkUp name of the target, keepResp

end activateField


on setUpMarkUp rField, keepResp

  -- Create (if necessary), format, and position a "markup" field
  -- so that the graphic markup will display properly beneath RESPFIELD.
  -- RFIELD     = Name of the field where student types in response.
  -- KEEPRESP  = If TRUE, current contents of RFIELD are left alone, else erased.

  global responseField, shadowFName, responseFont, muField, markupFont

  put resolveFName( rField ) into rField
```

```
if fieldExists( rField ) then

 lock screen
 put rField into responseField
 put word 1 to 2 of rField into cbf
 put quote & "markup" & quote into muS
 put cbf && muS into muField

 makeShadowField rField  -- Makes rField transparent.
 put the result into shadowFName

  -- If markup field doesn't exist, create it.
 if NOT fieldExists( muField ) then
   set editBkgnd to ( cbf = "card field" )
   domenu "New Field"
   set editBkgnd to False
    do "set name of last" && cbf && "to" && muS
 end if

  -- Move markup field just behind response field.
 show muField
 select muField
 repeat until ( number of muField = number of rField - 1 )
   if ( number of muField > number of rField ) then doMenu "Send Farther" else doMenu "Bring
Closer"
 end repeat
 choose browse tool
 hide muField
 set lockText of muField to True
 put empty into the name of muField

  --  Use fixed-width font and make sure that font characteristics
  --  are identical so marks will line up with chars of response string.
 if responseFont = empty then put "Courier" into responseFont
 if markupFont = empty then put "Courier" into markupFont

  set style of rField to "transparent" -- So markup will show thru.
 set textFont of rField to responseFont

 put textHeight of rField into tH
 put textFont of rField into tFont
 put textSize of rField into tSize
  put textStyle of rField into tStyle

 set style of muField to "opaque"
 set textFont of muField to markupFont
 set textSize of muField to tSize
  set textStyle of muField to tStyle
 set textHeight of muField to tH
 set fixedLineHeight of muField to fixedLineHeight of rField
 set wideMargins of muField to wideMargins of rField
```

```
    -- Extend markup field 1 standard char width to left.
    put rect of rField into r
     subtract stringUtilities("strWidth", "a", tFont, tSize, tStyle ) from item 1 of r
    set rect of muField to r

    -- Lower mu field the height of a letter.
     get stringUtilities("fontInfo", tFont, tSize, tStyle)
    put item 1 of it + item 2 of it into fontHeight  -- Ascent + descent
    get rect of muField
    add fontHeight to item 2 of it
    add fontHeight + 3 to item 4 of it
    set rect of muField to it

    -- Expand shadow 1 pt beyond mu field.
    put "," into c
    get item 1 of it - 1 & c & item 2 of it - fontHeight - 1 & c & item 3 of it + 1 & c & item 4 of
it + 1
    set rect of shadowFName to it
   show shadowFName

    if ( keepResp ≠ True ) then put empty into the name of rField

    unlock screen
     select text of rField

  else ERROR( "Can't find response field:" && rField )

end setUpMarkUp


on setCorrectAnswer C, L1, L2

  -- Specify a new correct answer. If PRESERVEFORMATING = TRUE then C must be the name
  -- of a field. L1, L1 specify a line range within C ( 1 to end if omitted).
  -- Sets THEMARKUPANSFNAME, THEMARKUPANSL1, THEMARKUPANSL2,
  -- THEMARKUPANSVAR, THEMARKUPANS

     global theMarkUpAns, theMarkUpAnsL1, theMarkUpAnsL2, theMarkUpAnsVar,
theMarkUpAnsFName, preserveFormating
```

```
  if word 1 to 2 of C is in "card field bkgnd field" then
    if value( "there is a" && C ) then
     put C into theMarkUpAnsFName
      put value( C ) into theMarkUpAnsVar
    else error( "Can't find ans field." )
  else
    put empty into theMarkUpAnsFName
    put C into theMarkUpAnsVar
    if preserveFormating = True then error( "Ans FIELD NAME needed to preserve formating." )
  end if
  if L1 = empty then put 1 into theMarkUpAnsL1 else put L1 into theMarkUpAnsL1
  if L2 = empty then put number of lines in theMarkUpAnsVar into theMarkUpAnsL2 else put L2
into theMarkUpAnsL2
   put line theMarkUpAnsL1 to theMarkUpAnsL2 of theMarkUpAnsVar into theMarkUpAns --
Deformated ans text.

end  setCorrectAnswer


on setFeedBackField fName

  -- Specify name of field where any special feedback will be put.
  -- (Will be shown and hidden along with markup).

 global theFeedBackFName

  get resolveFName( fName )
  if fieldExists( it ) OR it = empty then
     if ( there is a field theFeedBackFName ) AND ( theFeedBackFName ≠ it ) then hide
theFeedBackFName
    put it into theFeedBackFName
   else error( "Can't find feedback field:" && it )

end setFeedBackField


on setOkNoField F

 global theOkNoFName

  put resolveFName( F ) into theOkNoFName

end setOkNoField


function okWd

 global theOkNoFName

  if theOkNoFName = empty then return "OK" else return "{1,1," & theOkNoFName & "}"

end okWd
```

```
function noWd

 global theOkNoFName

  if theOkNoFName = empty then return "NO" else return "{2,2," & theOkNoFName & "}"

end noWd


on defaultKeyHandling tf

  -- If TF = True, disable default key handler, else enable it.

  global userKeyHandling

   if ( tf = False ) OR ( tf = "off" ) then put True into userKeyHandling else put empty into
userKeyHandling

end defaultKeyHandling


on  setPreserveFormating  tf

  -- If TF = True, preserve text format of feedBack.

  global preserveFormating

  put ( tf = True ) OR ( tf = "on" ) into preserveFormating

end  setPreserveFormating


on  restorePunctuation

  global  theMarkUpPunctuation

  put   ( ".,:;<>?!{}()<>?" & return & space )  into theMarkUpPunctuation

end  restorePunctuation


on changeMarkUpSymbol oldSymbol, newSymbol

  global theMarkupSymbols

   get  offset(  oldSymbol,  "+-~XΔ«x\=><[" )
  if it ≠ 0
  then put newSymbol into char it of theMarkUpSymbols
  else error( "Bad default markup symbol." )

end changeMarkUpSymbol
```

```
on restoreDefaultMarkUpSymbols

 global theMarkupSymbols

  put "+-~XΔ«x\=><[" into theMarkupSymbols

end restoreDefaultMarkUpSymbols


on setJudgingParams

 -- Up to 8 pairs of parameters, each pair consisting of

 --    1. String which names one of the judging parameters (global var name)
 --    2. A value to assign to that parameter

 -- E.g.:  setJudgingParms "capFlag", "Authors_Caps", "anyOrderOK", True

 -- Each parameter named is set equal to the following value.  It keeps this
 -- value until it is changed again!  The order of the pairs is irrelevant.

 global capFlag, extraWordsOk, anyOrderOk, misspellOk, ¬
 wordMarkUpNeeded, runTogetherNeeded, adjustNeeded, shortCut, ¬
 markUpMapsNeeded, parameterDisplayNeeded, spellingOnlyNeeded, debugNeeded

  put ",capFlag,extraWordsOk,anyOrderOk,misspellOk," & ¬
  "wordMarkUpNeeded,runTogetherNeeded,adjustNeeded,shortCut," & ¬
 "markUpMapsNeeded,parameterDisplayNeeded,spellingOnlyNeeded,debugNeeded," ¬
 into paramNames

 put -1 into i
 repeat
   add 2 to i
   if ( i <= the paramCount - 1 ) then
     put param( i ) into pName
     put param( i+1 ) into pVal
     if ( comma & pName & comma is not in paramNames ) then
       error( "Bad judging param name:" && pName )
       exit setJudgingParams
      else do "put" && pVal && "into" && pName
   else exit repeat
 end repeat

end setJudgingParams
```

```
on restoreDefaultJudgingParams

  -- Set global vars with default values of input params.
  -- Other permissible values for params shown in comments.

  global capFlag, extraWordsOk, anyOrderOk, misspellOk, ¬
  wordMarkUpNeeded, runTogetherNeeded, adjustNeeded, shortCut, ¬
  markUpMapsNeeded, parameterDisplayNeeded, spellingOnlyNeeded, debugNeeded

  put "Exact_case" into capFlag          --"Authors_caps", "Ignore_case"
  put False into extraWordsOK           -- True
  put False into anyOrderOK             -- True
  put False into misspellOK             -- True
  put True into wordMarkUpNeeded         -- False
  put True into runTogetherNeeded        -- False
  put True into adjustNeeded             -- False
  put True into shortCut                 -- False
  put False into markUpMapsNeeded        -- True
  put "x" into parameterDisplayNeeded    -- "m", ... etc.
  put "x" into spellingOnlyNeeded        -- "r", "p"
  put False into debugNeeded             -- True

end restoreDefaultJudgingParams


on resetErrata

  global theJudgingHandler, theOkNoFName, theFeedBackFName, preserveFormating

  put "judgeResponse" into theJudgingHandler
  put empty into theOkNoFName
  put empty into theFeedBackFName
  put True into preserveFormating

end resetErrata


on restoreMarkUpDefaults

  restoreDefaultMarkUpSymbols
  restoreDefaultJudgingParams
  restorePunctuation

end restoreMarkUpDefaults

-----------------------------------------------------------------

-- INTERFACE USER INPUT WITH JUDGING/MARKUP.

-----------------------------------------------------------------
```

```
on keyDown ch

  -- Default key handler.  If key was pressed in current response field, use it for
  -- judging; otherwise, pass it on. If USERKEYHANDLING = TRUE, simply pass key thru.

  [illegible]l responseField, userKeyHandling

  if [illegible]rKeyHandling ≠ True ) AND ( name of the target = responseField ) then handleKey ch
else pass keyDown

end keyDown


on handleKey ch

  -- Handles keys typed into the response field so that response is
  -- judged, and the markup field is shown, when RETURN is pressed.
  -- Other keys make the markup field dissappear, to prevent copying.

  global theMarkUpAns, muField, theFeedBackFName, theJudgingHandler

  put the selectedChunk into saveSC
  if ch = return then
    if ( theJudgingHandler = empty ) then put "judgeResponse" into theJudgingHandler
    send theJudgingHandler to the target
   show muField
   if ( theFeedBackFName ≠ empty ) then show theFeedBackFName
    select saveSC
  else
   hide muField
   if ( theFeedBackFName ≠ empty ) then hide theFeedBackFName
    select saveSC
    send "keyDown ch"
  end if

end handleKey


-----------------------------------------------------------------

-- DO MARKUP USING GLOBAL JUDGING PARAMS.

-----------------------------------------------------------------


function markupUsingParams model, response

  -- Do markup using global vars as input parameters.  Return markup string directly.
  --The judgment and other values are returned in global var THEMARKUPRETURNVALUES.

  global capFlag, extraWordsOk, anyOrderOk, misspellOk, ¬
  wordMarkUpNeeded, runTogetherNeeded, adjustNeeded, shortCut, ¬
  markUpMapsNeeded, parameterDisplayNeeded, spellingOnlyNeeded, debugNeeded
  put empty into ıneMarkUpDebug

  if the stackSpace > 28500 then
    return markup(model, response, ¬
    capFlag, extraWordsOk, anyOrderOk, misspellOk, ¬
    wordMarkUpNeeded, runTogetherNeeded, adjustNeeded, shortCut, ¬
   markUpMapsNeeded, parameterDisplayNeeded, spellingOnlyNeeded, ¬
   debugNeeded )
  else ERROR( "Stack too small for MarkUP XFCN" )

end markupUsingParams


------------------------------------

-- HILITE MATCHED MATERIAL IN RESPONSE.

------------------------------------


on hiliteMatch r, fName

  -- R return from a call to the match XFCN:  match( Model, Response ).
  -- The contents of field FNAME (which should be identical to Response) are hilited using M.

  put value( fName ) into response
  put response into the name of fName -- Remove text formating.

  delete char 1 of r  -- T or F.
  if r ≠ empty then
    put empty into list
    put r & space into r
    put length(r) into rL
   put space into lastC
    repeat with i = 1 to rL
      put char i of r into c
      if lastC ≠ "x" AND c = "x" then
        put i & "," after list
      else if lastC = "x" AND c ≠ "x" then
        put i - 1 & "," after list
     end if
     put c into lastC
   end repeat
    delete last char of list -- remove trailing ","
    put 1 into i
    repeat until i >= number of items in list
      put item i of list into startC
      put item i + 1 of list into endC
      do "set textStyle of char startC to endC of" && fName && "to bold"
      add 2 to i
    end repeat
  end if

  return m

end hiliteMatch


function getHelp type

  -- Return help text labelled as TYPE for current analysis text; if TYPE is empty, return all
help texts.

  global theMarkUpAns

  put ":" into dd
  put dd & "help" into h
  put empty into help
  repeat with i = 1 to number of lines in theMarkUpAns
    get line i of theMarkUpAns
    if ( word 1 of it = h ) then
      if ( ( word 2 of it = type ) OR ( type = empty ) ) then
        put getFeedBack( i ) & return after help
      end if
    end if
  end repeat
  return help

end getHelp


function getFeedback lineNo, cmds

  -- Look at current answer and get line range between LINENO and the next SPEC,
  --exclusive. If PRESERVEFOMATING is True, return the line range, in format
   -- "{startLine,stopLine}", else return the text.
  -- StartLine and StopLine are relative to THEMARKUPANSVAR.

  global preserveFormating, theMarkUpAns, theMarkUpAnsL1
  if theMarkUpAnsL1 ≠ empty then
    put ":" into dd
    put theMarkUpAnsL1 - 1 into L  -- lineNo is relative to L.
    if ( lineNo is a number ) then
      add 1 to lineNo
      put 1 + number of lines in theMarkUpAns into r
      repeat with i = lineNo to r
        get word 1 of line i of theMarkUpAns
        if ( char 1 of word 1 of it = dd ) then
          put i into r
          exit repeat
        end if
      end repeat
      subtract 1 from r
      if lineNo <= r then
        if ( preserveFormating = True )
        then return "{" & L + lineNo & "," & L + r & "}"
        else return line lineNo to r of theMarkUpAns
      else return empty
    else return empty
  else return empty

end getFeedBack


-----------------------------------------------------------------

-- NUMBER JUDGING.

-----------------------------------------------------------------


function judgeNumber range, response

  -- Return TRUE if number RESPONSE is within RANGE, else FALSE.
  -- RANGE is an expression of form "(50-75)" or "(50)".
  -- Use "*" to indicate open ended range, e.g., "(*-100)"
  -- or "(100-*)".  Use "¬" to reverse polarity of judgment,
  -- e.g., "¬(100-150)".  Extra spaces in range expression are ok.

  repeat with i = 1 to ( number of chars in range )
    if char i of range ≠ space then
      put ( char i of range = "¬" ) into negFlag
    end if
  end repeat

  repeat with i = 1 to ( number of chars in range )
    if char i of range is in "¬-(,)" then put " " into char i of range
  end repeat

  put stringUtilities(" removeChars", response, ",") into response
  put word 1 of range into n1
  put word 2 of range into n2
  if ( n2 = empty ) then
    if ( n1 is not a number ) then return False
    put ( response = n1 ) into j
    if negFlag then return NOT j else return J
  else
    if NOT ((n2 is a number ) OR ( n2 = "*" )) then return False
    if NOT ((n1 is a number ) OR ( n1 = "*" )) then return False
    put (( n1 = "*" ) OR ( n1 <= response )) ¬
    AND (( n2 = "*" ) OR ( response <= n2 )) into j
    if negFlag then return NOT j else return j
  end if

end judgeNumber


-----------------------------------------------------------------

-- JUDGE COMBINATION OF PATTERN MATCHING AND ANSWER/WRONG WITH MARKUP.

-----------------------------------------------------------------


on judgeResponse

  -- Default judging handler.  Judge response from RESPONSEFIELD using analysis specifications
  -- from global THEMARKUPANS.  Display markup and feedback.  Info on best ans returned via
  -- THE RESULT.THEJUDGMENT, THEFEEDBACK, THEMARKUP also set.

  global responseField, theMarkUpAns, theMarkUp, theFeedBack, muField

  if fieldExists( responseField ) then
    get responseAnalysis( theMarkUpAns, value( responseField ) ) -- Do response analysis.
    if char 1 of it = "%" then return empty
    showMarkUp
    feedBack theFeedBack
     if ( there is a card field "bestFit" ) then put item 1 of it into card field "bestFit" -- Show
bestfit.
    return it -- BestAns info.
  else error( "Can't find response field: " & responseField )

end judgeResponse
```

```
function responseAnalysis  model, response

 -- Go through a list of ANSWER/WRONG/MATCH/NUMBER specifications
 -- (with optional feedback) Find the best ANSWER/WRONG.  Also match all patterns.
 -- Direct return is BESTFIT,BESTLINE,BESTMU.  Also sets THEJUDGMENT, THEMARKUP,
 -- THEFEEDBACK, OKCOUNT, NOCOUNT, OKFEEDBACK, NOFEEDBACK, THEPATTERNFEEDBACK.

 global theMarkupReturnValues, theBestFitThreshold, okCount, noCount, ¬
 okFeedBack, noFeedBack, thePatternFeedBack, theJudgment, theFeedBack, ¬
 theMarkUp, theResponse

 put "." into d
 put ":" into dd
 put response into theResponse
 if NOT ( char 1 of word 1 of model = dd ) then put dd & "answer " before model
 -- if ( last line of model ≠ "#" ) then put return & "#" after model
 if ( theBestFitThreshold = empty ) then put .7 into theBestFitThreshold
 put theBestFitThreshold into bestFit
 put empty into bestLine
 put empty into bestMU
 put empty into thePatternFeedBack
 put empty into okFeedBack
 put empty into noFeedBack
 put False into pos
 put 0 into okCount
 put 0 into noCount

 repeat with i = 1 to number of lines in model
   put line i of model into m
   if char 1 of word 1 of m ≠ dd then next repeat
   put ( char 2 to 99 of word 1 of m ) into cmd
   if ( cmd is in "match.number.do." ) then    -- MATCH, NUMBER
     put word 2 of m into polarity
     delete word 1 to 2 of m
     if m = empty then
       get True
     else if cmd = "match" then
       get match( m, theResponse )
     else if cmd = "number" then
       get judgeNumber( m, theResponse )
     else --DO
       send m to this card
       if the result = empty then get True else get the result
     end if
```

```
      if char 1 of it = "t" then
        put ( d & polarity & d ) into p
        get getFeedBack( i, model )
          if ( it ≠ empty ) then get "•" && it & return
        if p is in ".ok.okStop." then
          add 1 to okCount
          put it after okFeedBack
        else if p is in ".no.noStop." then
          add 1 to noCount
          put it after noFeedBack
        end if
        if p is in ".okStop.noStop." then exit repeat
      end if
    else if ( cmd is in "answer.wrong" ) then  -- ANSWER, WRONG
      if ":?" is in word 2 of m then
        if bestFit <= theBestFitThreshold then
          put ( cmd = "answer" ) into pos
          put 1 into bestFit
          put i into bestLine
          put empty into bestMU
          exit repeat
        end if
      end if
      put markupUsingParams( word 2 to 999 of m, theResponse ) into mu
      if char 1 of mu = "%" then  -- XFCN error.
        ERROR( mu )
        put False into theJudgment
        put empty into theFeedBack
        put mu into theMarkUp
        return mu
      else
        get theMarkUpReturnValues
        send "computeAnsFit it" to this card
        put the result into ansFit
        if ( ansFit > bestFit ) then
          put ( cmd = "answer" ) into pos
          put ansFit into bestFit
          put i into bestLine
          put mu into bestMU
        end if
      end if
    end if
  end repeat

  put ( bestFit = 1 ) AND pos into ansOK
  send "matchingIsOK" to this card
  put the result into matchOK
  put ansOK OR (( bestLine = empty ) AND matchOK ) into theJudgment
  if ( bestFit < 1 ) AND pos then put empty into fbk
  else if ( bestLine ≠ empty ) then put getFeedBack( bestLine, model ) & return into fbk
  else put empty into fbk
  if ( theJudgment AND matchOK ) then put okFeedBack into thePatternFeedBack else put
noFeedBack into thePatternFeedBack
```

```
  if theJudgment then put okWd() into jdg else put noWd() into jdg
  put jdg & return & fbk & thePatternFeedBack into theFeedBack
  put bestMU into theMarkUp
  if theJudgment OR ( theMarkUp = empty ) then put jdg into theMarkUp

  return ( bestFit & comma & bestLine & comma & bestMU )

end responseAnalysis


on feedBack fdbk, addit

  -- Show FDBK in current feedback field.  If ADDIT ≠ empty, append, else replace.

  global theFeedBackFName

  showText fdbk, theFeedBackFName, addit

end feedBack


on showText txt, F, addit, cmds

  -- Show TXT in field named by F.  If ADDIT ≠ empty, then append it.  Dereference &
  -- copyText line ranges of form {STARTL,STOPL} STARTL, STOPL are relative
  -- to full exercise text in THEMARKUPANSVAR and field THEMARKUPANSFNAME.

  global theMarkUpAnsFName, preserveFormating

  put resolveFName( F) into F
  if fieldExists( F ) then
    lock screen
    if ( addit = empty ) then put empty into the name of F
    if preserveFormating = True then
      put 0 into e
      repeat
        put item 1 of findInField( txt, "{", True, e ) into s
        if s = 0 then
          put char e + 1 to 99999 of txt after the name of F
          exit repeat
        else
          put char e + 1 to s - 1 of txt after the name of F
          put item 1 of findInField( txt, "}", True, s ) into e
          put char s + 1 to e - 1 of txt into emb
          put item 1 of emb into startLine
          put item 2 of emb into endLine
          put item 3 of emb into Sr
          if Sr = empty then put theMarkUpAnsFName into Sr
          put value( Sr ) into tx
          if "»" is in tx then
            repeat with j = startLine to endLine
          get line j of tx
           if word 1 of it = "»" then
             if cmds ≠ False then send word 2 to 9999 of it to this card
           add 1 to startLine
          else exit repeat
        end repeat
      end if
         quickCopy "line", startLine, endLine, Sr, "char", 99999,, F
     end if
   end repeat
  else put txt after the name of F -- No formating.
 show F
  unlock screen
end if

end showText


on showMarkUp

  -- Break lines of markup to parallel line breaks in response and display result.

  global responseField, muField, theMarkUp

  put short name of muField into mF
  put theMarkUp into marks
  put value( responseField ) into r
  if wideMargins of responseField then get 18 else get 10
  put width of responseField - it into w
  put textFont of responseField into tFont
  put textSize of responseField into tSize
  put textStyle of responseField into tStyle
  if "{" is not in theMarkUp then
    repeat with i = 1 to 99999
      if r ≠ empty then
                get stringUtilities( "findLineBreak", r, w, tFont, tSize, tStyle ) --
Firstbreak,lastdisplayed,lastbreak + 1
        put min( item 3 of it - 1, item 2 of it ) into endChar
        if i = 1 then put 1 into s else put 0 into s
        put char 1 to endChar + s of marks into line i of card field mF
        delete char 1 to ( item 3 of it - 1 ) of r
        delete char 1 to ( item 3 of it - 1 + s ) of marks
     else
        put marks after card field mF
        exit repeat
     end if
   end repeat
  else  showText theMarkUp, muField

end showMarkUp
```

```
function fieldExists F

  -- FNAME is name of field, e.g., card field "myField"; bkgnd field "yourField"

  if ( F = empty ) then return False else return value( "there is a" && F )

end fieldExists


on quickCopy chnk1, p1, p2, f1, chnk2, q1, q2, f2

  -- Fields F1 and F2 must be on current card.  F2 must not have shared text.

  if q2 = empty then put q1 - 1 into q2
  lock screen
  put visible of f1 into saveV1
  show f1
  get "select" && chnk1 && "p1 to p2 of" && f1
  do it
  put the selection into S1
  put ( the selection = empty ) into void
  if NCT void then doMenu "Copy Text"  -- Copy disabled if null selection.
  set visible of f1 to saveV1
  put lockText of f2 into saveLT2
  put visible of f2 into saveV2
  show f2
  set lockText of f2 to False  -- Must be unlocked to paste into.
  do "select" && chnk2 && "q1 to q2 of" && f2
  put the selectedChunk into SC2
  put the selection into S2
  if NOT void then doMenu "Paste Text" else put empty into the selectedChunk
  set lockText of f2 to saveLT2
  set visible of f2 to saveV2
  unlock screen

end quickCopy


on error str

  -- Show error msg STR then drop into debug mode.

  answer str
  debug checkpoint

end error
```

```
function resolveFName F

  if ( word 2 of F = "field" ) then return F
  else if there is a card field F then return name of card field F
  else if there is a field F then return name of field F
  else if ( F ≠ empty ) then error( "Can't find field named: " & F )
  return empty

end resolveFName


function nthItem t, n, d

  -- Return Nth item of T delimited by D.

  put the itemDelimiter into s
  set itemDelimiter to d
  get item n of t
  set itemDelimiter to s
  return it

end nthItem


function itemCount t, d

  put the itemDelimiter into s
  set itemDelimiter to d
  get number of items in t
  set itemDelimiter to s
  return it

end itemCount


on computeAnsFit muR

  -- ansFit := ( 3*pMatched*(1 - aveDist) + pNonInv ) / 4
  return (3*(item 2 of muR)*(1 - item 4 of muR) + (item 3 of muR))/4

end computeAnsFit


on matchingIsOK

  global okCount, noCount

  return ( noCount = 0 ) AND ( okCount > 0 )

end matchingIsOK
```

```
on makeShadowField F

  -- Create a field identical to F at the location of F, then make F transparent.
  -- Expand shadow field to exceed F's markup field by 1 dot.

  if F ≠ empty then
    lock screen
    put word 1 to 2 of F into cdbg
    put the editBkgnd into saveEB
    if cdbg = "bkgnd field" then set editBkgnd to True
    put cdbg && quote & "*" & short name of F & quote into shadowF
    if value( "there is not a" && shadowF )  then
      put visible of F into saveV
      show F
      select F
      doMenu "Copy Field"
      doMenu "Paste Field"
      do "set name of last" && cdbg && "to" && word 3 of shadowF
      set visible of F to saveV
    end if
    do "select" && shadowF
    repeat until number of shadowF = number of F - 1
      if number of shadowF > number of F
      then doMenu "Send Farther" else doMenu "Bring Closer"
    end repeat
    choose browse tool
    set editBkgnd to saveEB
    if style of F ≠ "shadow" then set style of shadowF to "rectangle"
    set style of F to "transparent"
    set script of shadowF to empty
    set lockText of shadowF to True
    unlock screen
  else ERROR( "Can't find response field." & F )
  return shadowF

end makeShadowField


function subst str, k, o1, n1, o2, n2, o3, n3, o4, n4, o5, n5, o6, n6

  -- Substitute N-i for O-i, K times for each pair, in RESP; return result.

  if k = empty then put -1 into k
  return stringUtilities("mSubst",str, k, o1, n1, o2, n2, o3, n3, o4, n4, o5, n5, o6, n6)

end subst


on substResp  k, o1, n1, o2, n2, o3, n3, o4, n4, o5, n5, o6, n6

  -- Give params to SUBST() to apply to THERESPONSE.

  global theResponse
  put subst( theResponse, k, o1, n1, o2, n2, o3, n3, o4, n4, o5, n5, o6, n6 ) into theResponse
  return False

end substResp


on upCaseResp stripDiacrits

  -- Convert THERESPONSE to all upper case; if STRIPDIACRITS Is TRUE, remove diacrits also.

 global theResponse

  put stringUtilities( "uprString" theResponse, stripDiacrits ≠ True ) into theResponse

end upCaseResp


function labelLines F, label, d

  -- Return line numbers in field F bracketed between LABEL and the next following
  -- label-type string, exclusive.  A label string begins with char D and begins its line.

  if d = empty then put "#" into d
  if char 1 of label ≠ d then put d & label into label
  put value( resolveFName(F) ) into txt
  get findInField( txt, label, False, 0)
  put item 1 of it into p
  if p > 0 then put item 2 of it + 1 into l1 else return empty
  get findInField( txt, d, True, p )
  if item 1 of it > 0
  then return l1 & "," & item 2 of it - 1
  else return l1 & "," & number of lines in txt

end labelLines


on displayLabel label, S, D, d

  -- Copy to field D the block of text started by LABEL with delim char D in field S.

  get labelLines( S, label, d )
  if it ≠ empty then
    put return after the name of D
    quickCopy "line", item 1 of it, item 2 of it, resolveFName( S ), ¬
    "char", 99999,, resolveFName( D )
  end if

end displayLabel
```

```
-----------------------------------------------------------------

-- ERRATA EXTRAS  Version 1.0

-----------------------------------------------------------------


on cursorIntoResponseField

  -- Place cursor at end of response field text.

  global responseFielc

  select after text of responseField

end cursorIntoResponseField


-----------------------------------------------------------------

-- COPY STYLED TEXT FROM FIELD TO FIELD (ACROSS STACKS)

-----------------------------------------------------------------


on copyText chnk1, p1, p2, cb1, f1, c1, s1, chnk2, q1, q2, cb2, f2, c2, s2

  -- Copy any chunk from any field in any stack into any other field in any stack, PRESERVING
TEXT FOMRAT. E.g.,
   -- COPYTEXT "line", 1, 10, "cd", "aField", "aCard", "aStack", "char", 50, 50, "bkgnd",
"bField", "bCard", "bStack"

  If p1 = empty then put 1 into p1
  if q1 = empty then put 1 into q1
  if p2 = empty then put 99999 into p2  -- If no P2, take P1 - end.
  if q2 = empty then  -- If no Q2, locate cursor at Q1.
   put q1 into q2
   add 1 to q1
  end if

  put quote into q
  put the short name of this stack into s  -- Expand stack, card & field names & chnk ptrs.
  if s1 = empty then put s into s1  -- Defaults are this card, this stack.
  if s2 = empty then put s into s2
  put "stack" && q & s1 & q into s1
  put "stack" && q & s2 & q into s2
  get the short name of this card
  if c1 = empty then put it into c1
  if c2 = empty then put it into c2
  put "card" && q & c1 & q into c1
  put "card" && q & c2 & q into c2
```

```
 put "card field" && q & f1 & q into f1
 put "card field" && q & f2 & q into f2
 if char 1 of cb1 ≠ "c" then put "bkgnd" into word 1 of f1
 if char 1 of cb2 ≠ "c" then put "bkgnd" into word 1 of f2
 put the long name of this card into cs
 put ( c2 && "of" && s2 ) into cs2
 put ( c1 && "of" && s1 ) into cs1
 put "Bad COPYFIELD stack spec." into se

 if value( "there is a" && s1 ) AND value( "there is a" && s2 ) then  -- Do the copying.
   put "Bad COPYFIELD field spec." into fe
   put "COPYFIELD can't GO card, stack." into nogo
   put the lockMessages into saveLM  -- Disable openStack, etc.
   set lockMessages to True
   lock screen
   if ( cs ≠ cs1 ) then GO cs1
   if ( the result = empty ) then
     if value( "there is a" && f1 ) then
       put visible of f1 into saveV1
       show f1   -- Must be visible to select.
        do "select" && chnk1 && "p1 to p2 of" && f1
       put ( the selection = empty ) into void
       if NOT void then doMenu "Copy Text"  -- Copy disabled if null selection.
       set visible of f1 to saveV1
     else error( fe )
   else error( nogo )
   if ( cs1 ≠ cs2 ) AND ( the result = empty ) then GO cs2
   if ( f2 ≠ empty ) then  -- If no f2 specified, just leave on clipboard, no error.
     if ( the result = empty ) then
       if value( "there is a" && f2 ) then
         put visible of f2 into saveV2
         put lockText of f2 into saveLT2
        put the editBkgnd into saveBG
           if sharedText of f2 = True then set editBkgnd to True  -- Must be in bkgnd to edit
sharable text.
         show f2  -- Must be visible to select.
         set lockText of f2 to False  -- Must be unlocked to paste into.
          do "select" && chnk2 && "q1 to q2 of" && f2
        if NOT void then doMenu "Paste Text" else put empty into the selectedChunk
          set visible of f2 to saveV2  -- Restore state of field.
        set editBkgnd to saveBG
         set lockText of f2 to saveLT2
       else error( fe )
    end if
   else  error( nogo )
  GO cs
   set lockMessages to saveLM
   if ( the result ≠ empty ) then error( nogo )
   unlock screen
  else error( se )

end copyText
```

```
on deReferenceText txt

  -- Return unformated dereferenced copy of txt with line references.

  global theMarkUpVar

  if ( addIt = empty ) then put empty into r
  put 0 into e
  repeat
    put item 1 of findInField( txt, "{", True, e ) into s
    if s = 0 then
      put char e + 1 to 99999 of txt after e
      exit repeat
    else
      put char e + 1 to s - 1 of txt after r
      put item 1 of findInField( txt, "}", True, s ) into e
      get char s + 1 to e - 1 of txt
      put line item 1 of it to item 2 of theMarkUpVar into r
    end if
  end repeat
  return r

end deReferenceText


function labelLines F, label, d

  -- Return line numbers in field F bracketed between LABEL and the
  -- following LABEL, exclusive.  LABEL begins with ":" and occupies its own line.

  if d = empty then put "#" into d
  if char 1 of label ≠ d then put d & label into label
  put value( resolveFName(F) ) into txt
  get findInField( txt, label, False, 0)
  put item 1 of it into p
  if p > 0 then put item 2 of it + 1 into l1 else return empty
  get findInField( txt, d, True, p )
  if item 1 of it > 0
  then return l1 & "," & item 2 of it - 1
  else return l1 & "," & number of lines in txt

end labelLines


on displayLabel label, S, D, d

  get labelLines( S, label, d )
  if it ≠ empty then
    put return after the name of D
    quickCopy "line", item 1 of it, item 2 of it, resolveFName( S ), ¬
    "char", 99999,, resolveFName( D )
  end if

end displayLabel
```

```
----------------------------------------------------------------

-- GET INFO ABOUT PARAMS AND TABLES INTERNAL TO MARKUP XCMD.

----------------------------------------------------------------


function getJudgingInfo

  -- Return a report of all Markup's current internal parameter values.

  global theMarkupParamDisplay, parameterDisplayNeeded

  put empty into r
  put "vbdchpwfsm" into typeList
  put parameterDisplayNeeded into saveDN
  repeat with i = 1 to length( typeList )
    put (char i of typeList) into parameterDisplayNeeded
    get markUpUsingParams( empty, empty )
    put ( theMarkUpParamDisplay & return & return ) after r
  end repeat
  put saveDN into parameterDisplayNeeded
  return char 1 to length(r) - 2 of r

end getJudgingInfo


function inBrackets txt

  put offset("{", txt ) into p1
  put offset( "}", txt ) into p2
  if p1 > 0 AND p2 > 0 then return char p1+1 to p2-1 of txt else return empty

end inBrackets
```