DOCUMENT RESUME

ED 383 212 FL 023 018

AUTHOR Hart, Robert S.
TITLE Improved Algorithms for Identifying Spelling and Word Order Errors in Student Responses.
INSTITUTION Illinois Univ., Urbana. Language Learning Lab.
REPORT NO LLL-T-22-94
PUB DATE Dec 94
NOTE 80p.
AVAILABLE FROM Language Learning Laboratory, University of Illinois at Urbana-Champaign, G70 Foreign Languages Building, 707 S. Mathews St., Urbana, IL 61801.
PUB TYPE Guides - Non-Classroom Use (055)

EDRS PRICE MF01/PC04 Plus Postage.
DESCRIPTORS *Algorithms; *Authoring Aids (Programming); Capitalization (Alphabetic); Comparative Analysis; *Computer Assisted Instruction; Diacritical Marking; *Error Analysis (Language); Grammatical Acceptability; Hypermedia; Programming; *Sentence Structure; *Spelling
IDENTIFIERS *Word Order

ABSTRACT

The report describes improved algorithms within a computer program for identifying spelling and word order errors in student responses. A "markup analysis" compares a student's response string to an author-specified model string and generates a graphical error markup that indicates spelling, capitalization, and accent errors, extra or missing words, and out-of-order words. The algorithm determines whether the response was acceptable or not, and computes a string of graphical error marks to be displayed below the student response. Synonyms and ignorable words can be specified and spelling errors, extra words, and word order errors can be accepted at the author's discretion. Spelling analysis is done using a dynamic programming algorithm that produces a least-cost edit trace; word order analysis is implemented using recursive branch and bound search. Improvements on earlier versions of the algorithm give more intuitive markup values. The algorithm is implemented as a HyperTalk XFCN. HyperTalk scripts can provide numerous input parameters that control the details of the matching process, and the algorithm returns a variety of fit measurements that characterize the match. Non-roman linear writing scripts are supported. The report contains detailed information on use of the function and serves as a user manual. Contains two references. (Author/MSE)

Language Learning Laboratory
College of Liberal Arts and Sciences

LLL

Technical Report No. LLL-T-22-94
December 1994

University of Illinois
at Urbana-Champaign

IMPROVED ALGORITHMS FOR IDENTIFYING SPELLING AND WORD ORDER ERRORS IN STUDENT RESPONSES

Robert S. Hart

Technical Report No. LLL-T-22-94
December 1994

IMPROVED ALGORITHMS FOR IDENTIFYING SPELLING AND WORD ORDER ERRORS IN STUDENT RESPONSES

Robert S. Hart

ABSTRACT

This report describes improved algorithms for identifying spelling and word order errors in student responses. A markup analysis compares a student's response string to an author specified model string and generates a graphical error markup that indicates spelling, capitalization and accent errors, extra or missing words, and out-of-order words. The algorithm determines whether the response was acceptable or not, and computes a string of graphical error marks which can be displayed below the students response as error feedback. Synonyms and ignorable words can be specified within the model, and spelling error, extra words and word order errors can be accepted at the author's discresion. Spelling analysis is done using a dynamic programming alogorithm which produces a least-cost edit trace; word order analysis is implemented using recursive branch and bound search. Improvements on earlier versions of the algorithm give a more intuitive markup values. The algoritm is implemented as a HyperTalk XFCN. HyperTalk scripts can provide numerous input parameters that control the details of the matching process, and the algorithm returns a variety of fit measurements that characterize the match. Non-roman linear writing systems are supported. The report contains detailed information on use of the function and serves as a user manual.

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

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

Technical Report No. LLL-T-22-94

IMPROVED ALGORITHMS FOR IDENTIFYING SPELLING AND WORD ORDER ERRORS IN STUDENT RESPONSES

Robert S. Hart

Associate Director, Language Learning Laboratory
Assistant Professor of Humanities

September 1993

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

INTRODUCTION

The program described in this report implements improved versions of the algorithms for analyzing and marking word order errors in student responses first described in Hart (1988). Several minor difficulties in the previous version have been corrected, a number of new features have been added, and the whole utility has been reimplemented as a HyperCard XFCN (external function) to make it is available in a Macintosh environment.

The basic functionality of the Markup XFCN is to accept a model string (in educational applications, usually a "correct answer" specified by the courseware author), a response typed in by the student, and generate a graphic markup that indicates where the response is incorrect. Here is an example:

```
Model:      The very quick brown fox jumped over the big lazy dog

Response:   thevery qick pronw foxx oar the lazy big dög.
Markup:     -  [     \   =  ><    xΔXXX    Δ     «    ~      (1)
```

The markup symbols displayed beneath the (incorrect) response indicate the editorial changes needed to make the response match the model. The symbol "\" indicates one or more omitted letters; "x" an extra letter, and "=" an incorrectly substituted letter. The symbol pair "><" indicates transposition of two letters; "X" means an extra word (one that should not be in the response, or else one that is so badly misspelled that it can't be recognized); "Δ" indicates that one or more words is missing at that point in the response; and "«" means that the word is part of the response, but in the wrong position -- it should be moved leftward to one of the "Δ" symbols. Incorrect capitalization is symbolized by "_" and an accent error by "~".

Spelling analysis is done by a dynamic programming algorithm which generates a markup corresponding to a least-cost editing trace. Editing operations are restricted to omission of a letter, insertion of an extra letter, substitution of one letter for another, or transposition of two adjacent letters. Capitalization and accent errors are also identified and marked. The user may specify the way in which capitalization disagreements will be treated: exact agreement required, upper case required wherever the model has uppercase, or case differences ignored. Run-together words are identified as such if they are adjacent in the model. Some misspelling is tolerated in one or both run-togethers.

Order analysis identifies extra words, missing words, and misplaced words, The user can specify various degrees of tolerance when defining what constitutes a match with the model: spelling errors can be excused; incorrect word order can be excused, and extra words in the response can be excused. The order analysis returns three goodness of fit measures: proportion of matched words, proportion of words in correct order, and average amount of misspelling per matched word.

When specifying the model, the author is allowed to specify one or more words which will be ignored if they occur in the student's response. Such a word or list must be surrounded by angle brackets, <>. A list of "synonyms" (i. e., a set of words any one of which would be correct at a given position in a sentence) must be surrounded by square brackets, [ ].

The basic theory underlying the word order and spelling analysis and markup, which was presented in Hart (1989), has not changed and the reader should consult that document for a detailed exposition of the approach used. This report is restricted to two goals: (a) describing the changes in the algorithms made in version 2.0, and (b) giving a detailed account of the HyperCard interface so that lesson developers can easily incorporate the markup facility into HyperCard stacks. Appendix 1, however, does present a complete listing of this version of the MarkUp XFCN.

CORRECTIONS IN VERSION 2.0 OF THE MARKUP ALGORITHM

In certain cases, version 1.0 of the MarkUp algorithm returned a markup string which, though technically correct, was counter-intuitive. Once example is:

```
Model:      He lives in Chicago

Response:   He lives in in Chicago.
Markup:             ~  XX                                    (2)
```

Here, it would agree better with common sense if the mis-accented "in" were marked as the extra word rather than the adjacent, and perfectly correct, "in":

```
Response:       He lives in in Chicago.
Better Markup:          XX                                   (3)
```

The reason for the poor markup in (2) is simple. In determining the edit distance between pairs of words, version 1.0 of the MarkUp XFCN simply ignored any capitalization or accent errors. Word similarity was determined without any reference to these matters, and since spelling and word order analysis built on similarity information, they also ignored such errors. Only at the very end, when displaying the graphic markup did the algorithm check for such problems. Thus, as far as the word order routine is concerned, these two responses are identical:

Response1: He lives in Chicago (4a)

Response2: He lives in Chicago (4b)

In cases like (2), where there are redundant identical words, the MarkUp XFCN always uses the leftmost possibility.

The fix for this problem was straightforward. Two new weight parameters have been introduced into the program, wcap (the weight of a capitalization error) and waccent (the weight of an accent error). Since capitalization and accent errors are normally perceived as relatively minor problems, we do not want them to have much influence on edit distance, so their default values have been made much smaller than the remaining weights. The current default weights are:

```
wdelete          20
winsert 2        20
wsubstitute      30
wtranspose       20
wcap              1
waccent           1                                          (5)
```

This means that "in' and "in" are now at an edit distance of 1 from one another, rather than 0 as previously. Note also that the "average" distance associated with a single edit operation is now

((wdelete + winsert + wsubstitute + wtranspose)/4) + ((wcap + waccent)/2) (6)

This is appropriate because deletion, insertion, substitution, and transposition are mutually exclusive operations, but a cap or accent error may or may not accompany a substitution.

A related situation where version 1.0 of the MarkUp XFCN fails to perform satisfactorily is illustrated in this example:

```
Model:      the time

Response:   then the time.
Markup:       x XXX                                          (7)
```

Here the problem is not due to accents or capitalization errors, but to the way that response words were paired with model words. When considering candidates for pairings, the procedure ignored the exact magnitude of edit distance between word pairs. Instead, a cutoff criterion was used. If the edit distance exceeded the cutoff value, the pair were considered as a potential match; otherwise, they were considered to be distinct words which could not be paired under any circumstances. Thus, in (7) the response words "then" and "the" are considered to be equally good matches for the model word "the", and "then" is selected because it happens to be leftmost, even though this entails a spelling error.

A third case of inappropriate markup is

```
Model:      seen on a boat in Chicago

Response:   seen in a boat in Chicago
Markup:       Δ   « «    XX                                  (8)
```

In this case the first "Δ" marks the location of the missing words "on a boat", consequently "on" must be inserted at the location of the "Δ", and the words "a boat" are marked with "«" to show that them must be moved leftward from their current location to the location of the "Δ". This will leave the word "in" (the one which has not been marked as extra) adjacent to "Chicago", to make up the phrase "in Chicago". The second "in" is unneeded and is marked as extra. While technically correct, this markup is unintuitive and, if fact, very confusing. Most readers would agree that the student simply substituted "in" for "on" in her response, so the appropriate markup would be

```
Response:   seen in a boat in Chicago
Markup:         XX                                           (9)
```

The strange markup in (8) occurs because of the way the word order analysis operated in version 1.0. Matching proceeded in two steps. First response words were paired with candidate model words in such a way that the number of inversions is minimized; that is, the criterion for doing the matching is to keep the order of words in the response as close as possible to their order in the model. In (8), the candidates were

Response word:	1	2	3	4	5	6	
Model word candidates:	1	5	3	4	5	6	(10)

Notice that model word 5, "in", is the only available candidate for pairing with response word 2 and also response word 5, both of which are "in". (Model word 2, "on", is not a candidate for pairing with anything because its normalized edit distance from every response word exceeds the cutoff threshold.) In the initial phase of matching, response words are paired one at a time, proceeding from left to right, When response position 2 is considered, the only available candidate is model word 5. A response word is not allowed to remain unpaired if there is any candidate that will match it, so the pairing (2, 5) will be created. This means that when we reach response word 5, there is nothing left to pair it with, so it is tagged as an extra word.

To remedy such problems, the word order analysis had a second stage, embodied in the `adjust_solution` procedure, which attempted to improve the quality of the match by taking into account the actual magnitude of edit distance. This procedure looked at certain pairs of matches to see if exchanging the match between pairs would reduce the overall edit distance without increasing the number of inversions. However, attention was restricted to unmatched response words (namely, those matched with the "null" model word), and the algorithm merely checked to see if the overall match could be improved by taking a model word away from some other response word and pairing it with a currently unmatched one. The algorithm was roughly as follows, where M, M' denote words in the model ; R, R', words in the response, and R <-> M indicates a pairing of R with M:

```
FOR each matched response word R DO
    FOR each unmatched response word R' to the right of R DO
        BEGIN
            M := the word paired with R;
            IF edit distance of (M, R') is less than the edit distance of (M, R) AND
            (pairing M with R' leads to no more inversions than pairing M with R)
            THEN
                BEGIN
                    Re-pair M with R';
                    Re-pair null with R
                END
        END                                                          (11)
```

Although this adjustment cleared up many deficiencies in the match, it was not sufficient in general and still allowed markups like (8). To eliminate these shortcomings, version 2.0 of the `adjust_solution` algorithm has been completely rewritten and made much more general. Now all possible pairs of matches, R <-> M and R' <-> M', are considered, and if exchanging the pairing so that R <-> M' and R' <-> M results in a lesser overall edit distance without increasing the number of inversions, or decreases the inversion count without increasing the edit distance, then the solution is modified to incorporate the exchanged match, essentially as indicated in the following algorithm:

```
FOR each response word R DO
    FOR each response word R DO
        BEGIN
            M  := the model word matched with R;
            M' := the model word matched with R';
            oldEditD := edit distance of (M, R) + edit distance of (M', R');
            newEditD := edit distance of (M, R') + edit distance of (M', R);
            oldInvK := number of inversions in original solution;
            newInvK := number of inversion when R <-> M' and R' <-> M;
            IF (newEdit D < oldEdit D AND new InvK <= oldInvK) OR
               (newEditD = oldEditD AND newInvK < oldInvK) THEN
                BEGIN
                    Re-pair R with M';
                    Re-pair R' with M
                END
        END                                                          (12)
```

The revised algorithm considers many more potential exchanges, and thus improves the overall match in situations where the earlier version failed to do so.

A complete listing of version 2.0 of the MarkUp XFCN, including the `adjust_solution` procedure, appears in Appendix 1.

USE OF CHARACTER CATEGORY INFORMATION FOR SPELLING ANALYSIS

Version 1.0 of the MarkUp XFCN specified the phonetic category of each character (whether it was a vowel or consonant), and allowed the user to modify such information, but made no use of it. In version 2.0, phonetic information is utilized during the process of spelling analysis to achieve a more psychologically meaningful measure of edit distance.

The problem is the weight which should be attached to a substitution error, that is, an error which results because some character M in a model word has been replaced by a different character R in the student's

response. Version 1.0 assigned the same weight (namely, the value of the variable wchange which defaults to 30) regardless of the actual identities of M and R. Consequently, "readable" has the same edit distance from "readible" and "reidable" as it does from "readxble", "rezdible", "oedable", and "readaule". Intuitively, however, the latter substitutions are less likely to occur, at least for students who are careful typists but poor spellers -- a description which applies, for example, to many student language learners. The reason is that *spelling* errors usually involve substituting one vowel for another or one consonant for another, but only rarely a consonant for a vowel or visa versa. Of course *typing* errors, being a function of keyboard position and n-gram frequency (Rumelhart & Norman, 1982), do not necessarily show this pattern. We need a way of assigning differential substitution costs to different pairs of character categories.

The problem of determining accurate substitution probabililties for character pairs (or character category pairs) can be solved only by some combination of empirical data and psychological modelling. However, there are situations where even rough estimates will be useful. Consider the case of Japanese writing, which utilizes three basic categories of character: katakana, hirigana, and kanji. In the computer representations now becoming standard, all of these characters are represented as 32-bit character codes, and MarkUp will treat each as if it were a single character. But this psychologically inaccurate. Since the hiragana character represent CV syllables, a beginner will be relatively likely to confuse them with one another. The substitution of a hiragana character for a kanji within a word should be a relatively rare event, and thus have a high edit distance attached to it. Kanji do have internal structure, and thus might be substituted, one for another,. with one another with varying degrees of probability. However, the character code of a Kanji does not reveal its internal structure sufficiently so that MarkUp can determine similarity. Thus, the cost of every kanji-kanji substitution must be the same. In fact, to prevent every word in the model from being identified with every word in the response, such substitutions must be forbidden (infinite cost). We thus need at least two categories of character, hirigana and kanji. Hirigana-hirigana substitution will be permitted at a moderate cost, but kanj-kanji and kanji-hirigana substitution will be excluded. (This example is theoretical, because version 2.0 of MarkUp does not support 32-bit characters.)

Version 2.0 of MarkUp supports the assignment of differential substitution costs by providing five different "phonetic" categories. Actually, a better term would be *character categories*, because they need not actually concern phonetic properties of the character, and because the categories are mutually exclusive and exhaustive -- each character must belong to exactly one category. The categories have the hard-coded names

vowel, consonant, phon3, phon4, phon5

These labels are purely conventional however; the lesson author can redefine the categories in any way she wishes. The default phonetic information assigns a, e, i, o, u, and y to the "vowel" category and all other characters to the "consonant" category; as a result, the three remaining categories "phon3", "phon4" and "phon5" are not used at all.

To make use of these categories, a 5x5 matrix of weights called phon_matrix is created and intialized with these default values:

	Response Char Category				
Model Char Category	vowel	consonant	phon3	phon4	phon5
vowel	30	36	36	36	36
consonant	36	30	36	36	36
phon3	30	36	36	36	36
phon4	30	36	36	36	36
phon6	30	36	36	36	36

Each row M corresponds to a possible category of the model character, and each column R to a possible category of a response character. The cell phon_matrix[M, R] gives the cost of replacing a model character of category M by a response character of category R. For instance phon_matrix[vowel, consonant] is 36,

the cost of subsituting a consonant for a vowel. Given the default partition of characters between vowel and consonant, the matrix entries indexed by phon3, phon4, and phon5 are redundant, because they will never be referred to during the analysis.

The default weights are assigned according to the following rule: intra-category substitutions (vowel-vowel, consonant-consonant, and other cells along the diagonal) are assigned the value of the parameter `wchange`, which has a the default value 30. Inter-category substitutions (off-diagonal cells) are assigned the value `(1.2*wchange)` = 36. The factor 1.2 is arbitrary, but was chosen so that vowel-consonant substitutions would be more expensive than vowel-vowel or consonant-consonant, and yet not so expensive that they would prevent words containing typos from being identified as potential matches. The user can modify both character category assignments and the phon_matrix values, as explained below.

USING THE MARKUP XFCN

The MarkUp XFCN is implemented as a Macintosh code resource, so it must be made available to your stack before you use it. This can be done in several ways: (1) copy it directly into the stack resource fork with the RESCOPY or RESEDIT utilities; (2) copy it into your HOME stack resuorce fork using the same utilities; or (3) execute a START USING STACK command to attach a stack which already contains the MarkUp XFCN as a resource. Copying directly to your own stack is more stable, but also wasteful of space if copies proliferate.

Once the MarkUp XFCN has been made available, it can be called like any other HyperCard function. It enables you to produce a graphic error markup in a HyperCard stack. When you call MarkUP, you must input a model string and a response string. MarkUp will match the two and return a markup string, as well as other information about the quality of the match. You can then display this information to the student, or use it in any other way you chose.

As one step in generating the graphical error markup, the MarkUp XFCN *judges* the response ((i. e., evaluates it for correctness). You can control the amount and nature of error tolerance during this judging process by changing the values of various input parameters to MarkUp. To be specific, you can specify synonyms for various words in the response. You can specify words which should be ignored if they occur in the response. And you can stipulate that the response will be judged correct even if it contains spelling errors, or word order errors, or extra words.

Evaluating and marking up a user's response is a complex activity which can be modified in various ways to reflect various kinds of content and instructional needs. You can control the way in which MarkUp operates by setting input parameters. Thirteen of these parameters can be set by passing values directly to the MarkUp XFCN when it is called. More esoteric aspects of operation can be controlled by putting suitable values into five global HyperCard variables: `theMarkUpWeights`, `theMarkUpSymbols`, `theMarkUpCharInfo` and `theMarkUpDebug`.

The direct return of the Markup XFCN is a markup string. This is simply a sequence of symbols in character string format, like that in (1), which indicates the nature of the mismatches between the model and the student's response. Additional information may be returned in the four HyperCard global variables `theMarkUpReturnValues`, `theMarkUpMaps`, `theMarkUpParamDisplay`, and `theMarkUpDebug`.

CALLING THE MARKUP XFCN

The syntax for calling the MarkUp XFCN allows for up to 14 input parameters, however 12 of these are optional and need not be specified except in special situations. The general form of a MarkUp function call is

```
markUp( model,
        response,
        capFlag,
        extraWordsOk,
        anyOrderOk,
        misspellOk,
        wordMarkUpNeeded,
        runTogetherNeeded,
        adjustNeeded,
        shortCut,
        markUpMapsNeeded,
        parameterDisplayNeeded
        spellingOnlyNeeded
        debugNeeded )                                    (13)
```

The meaning of each of the thirteen input parameter slots, and the range of values acceptable in that slot, is as follows:

model	String or container specifying the correct response.
response	String or container holding the student's response.
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. 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`, an error markup string will be generated and returned. If `False` (the default), 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 compairing *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. Only the "pretty" markup will display properly, but it has incomplete information about the nature of errors present, so the "r" option 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.

The Response and Model strings *must* be specified when you call the MarkUp XFCN. The remaining 12 parameters are optional. If you are satisfied with the default value of a parameter, simply leave that slot empty (of course, a comma must be present to mark the location of the unused slot if other parameter values follow). If the unused parameters are dangling (i. e., come after the last real parameter value), the commas may also be omitted, following the usual HyperCard convention for input parameters.

values follow). If the unused parameters are dangling (i. e., come after the last real parameter value), the commas may also be omitted, following the usual HyperCard convention for input parameters.

IMPORTANT: Each of the 14 input parameters reverts to the default value after each call to MarkUp, so *non-default values must be respecified each time you call MarkUp.*

HyperCard evaluates each parameter before sending it to the MarkUp XFCN, so parameters may be specified by any HyperCard expression, including constants, variables, chunk specifiers, or field specifiers.

SPECIFYING THE MODEL AND RESPONSE PARMETERS

The simplest form of correct answer is a single word or string of words:

```
The quick brown fox jumped over the lazy dog.          (14)
```

The model should not contain any characters which are currently defined as punctuation marks, because such characters are removed from the student's response string before it is judged. If such characters appear in the model string, it will be impossible for the response to match the model.

The square brackets "[ ]" and the angle brackets "< >" have special uses in the model string. Square brackets are used to specify a list of (one or more) *synonymous words*. The words must be separted by one or more *spaces*; ther punctuation is not acceptable. The words do not have to be synonyms in the usual sense; in fact any collection of words can be put into a synonym list. Such a list simply specifies that any member of the list will be acceptable at that point in the model, as in

```
The [quick fast speedy] brown fox jumped over the [lazy stupid] dog     (15)
```

Any word in the list will be acceptable at that position in the response. Thus the model shown will result in the following markups

```
Response:   The quick brown fox jumped over the lazy dog.   OK      (16a)
Markup:     (none)

Response:   The speedy brown fox jumped over the stupid dog.  OK     (16b)
Markup:     (none)

Response    The brown speedy fox jumped the lazy dog over.   NO     (16c)
MarkUp:        Δ      «                  Δ            «
```

Angle brackets specify a list of (one or more) *ignorable words.*

```
<the a> brown fox jumped over [lazy stupid] dog                        (17)
```

There may be several lists of ignorable words, which may appear anywhere in the response, but the effect will always be the same as a single list of ignorables at the front of the model. Any response word which matches any of the ignorable words "well enough" will simply be treated as if it were not present in the response. "Well enough" is defined to permit captialization and accent errors, but no other kinds of spelling errors. Thus, if (17) is used as a model, the following responses will all be judged correct:

```
Response:   A brown fox jumped over the stupid dog.                    (18a)

Response:   The brown fox jumped over the lazy dog.                    (18b)
```

Version 2.0 of the MarkUp XFCN places some limitations on both model and response string:

> The model and response strings must each be 255 characters or less (or 22 characters, if you have selected the SpellingOnly analysis).
>
> No single word in the model or response may be more than 22 characters (punctuation and spaces do not count as part of a word)
>
> Neither model nor response may contain more than 18 words. Each entry in an ignorable word list or synonym list counts as a word.

Exceeding these limits will cause MarkUp to abort the judging process and return an error string which defines the nature of the error.

These limits are hard-coded in PASCAL as global constants, and can be changed by recompiling the PASCAL source code. They are imposed by the fact that Version 2.0 of MarkUp defines its large data structures as static arrays within PASCAL. Since MarkUp XFCN runs under HyperCard and has to borrow its space from HyperCard, increasing the limits above causes the HyperCard stack to overflow into the heap and immediately terminates HyperCard with system error 28 (stack has moved into application heap).

ADDITIONAL PARAMETERS CONTROLLING THE MARKUP PROCESS

The more technical aspects of the markup analysis can be controlled by changing the values of the five HyperCard global variables `theMarkUpPunctuation`, `theMarkUpSymbols`, `theMarkUpWeights`, `theMarkUpPhonMatrix` and `theMarkUpCharInfo`. You can change the values of these variables by using the HyperCard PUT command, and can inspect their current values by using the HyperCard debugger's SHOW VARIABLE.option. It is unlikely that you will have reason to change these variables, but specialized judging situations sometimes require it.

Each of these globals expects a list of comma-separated items as a value. To change a value simply PUT a new list into the appropriate global variable. *You must always provide the entire list of values, including all the values which you are not changing.*

Each time that the MarkUp XFCN is executed, it examines the values of each of these globals. If the value is empty, then the global is ignored and the default values built into MarkUp (as indicated immediately below) will be used. If the value is non-empty, then the contents of the global will be read into the appropriate PASCAL tables and variables before the markup analysis is begun. Hence, *you may revert to the default values of the parameters at any time by simply PUTing empty into the appropriate global.*

Note that these parameter values are "sticky": once you have PUT a value into a global, it will continue to be effective until you change it, or until you leave HyperCard. *You do not need to reset these values each time you call the MarkUp XFCN.* Of course, it will not hurt anything if you do so, except to slow things down a bit.

IMPORTANT: The information from these global variables is converted into PASCAL strings which cannot be more than 255 characters long. Hence, *never put more than 255 characters of text into these variables.*

Each of the four global variables expects to receive a list with a very exact format, as explained here:

`theMarkUpPunctuation`	This string of characters determines what characters MarkUp will consider to be punctuation marks if they occur in the student's response. Its default value is `( "<>;:()[]<>?!"`

`theMarkUpPunctuation`

This string of characters determines what characters MarkUp will consider to be punctuation marks if they occur in the student's response. Its default value is (`"<>;:()[]<>?!" & space & return`). If you redefine the punctuation set don't forget to include the space and return characters.

`theMarkUpSymbols`

This string is a list of 12 characters which determine the symbols used to display the error markup. The default value of the string is "_ _~XΔ«x\=><[". Each position in the list corresponds to a particular type of error:

1	addcap	"_"	underscore
2	dropcap	"_"	underscore
3	accenterror	"~"	tilde
4	extraword	"X"	capital x
5	missingword	"Δ"	capital delta
6	moveword	"«"	double left arrow
7	extraletter	"x"	lower case x
8	missingletter	"\"	backslash
9	substituteletter	"="	equal sign
10	transposeletter1	">"	left angle bracket
11	transposeletter2	"<"	right angle bracket
12	runonword	"["	left square bracket

The shapes shown are those which display in courier font. If you use a font other than courier, you may need to change some of the characters in this list, selecting appropriate characters from the font that you are actually using.

`theMarkUpWeights`

This is a list of nine comma-separated numbers which determine how spelling errors are computed. The default value of the is "20,20,30,20,1,1,0.67,0.35.0.2" The meaning of each position is

1	winsert	20
2	wdelete	20
3	wchange	30
4	wtranspose	20
5	wcap	1
6	waccent	1
7	cutoff	0.67
8	prop_errors	0.35
9	runon_criterion	0.2

The names in the second column are the PASCAL variable names used internally by the MarkUp XFCN. The first six numbers are the costs or weights attached to (respectively) letter insertion, omission, substitution, transposition, capitalization errors, and accent errors, when matching for spelling errors. The last three numbers have the following meanings:

cutoff: Ratio of word lengths (shorter/longer) must exceed this value, or the edit distance between them will automatically be set to infinity (relevant only when the "shortcut" input parameter is set to True).

prop_errors: Normalized edit distance between two words must be less than this value, or the two words will be considered non-matches

runon_criterion: Maximum edit distance which can exist between the concatenation of two adjacent model words, M and M', and a response word R, if R is to be considered as a candidate match for M and M' run together.

`theMarkUpPhonMatrix` This is a list of 25 (=5x5) comma-separated integer values. The first five values correspond to the first row of the `phon_matrix`; the next five to the second row, and so on. Item number R of row number M specifies the cost of replacing a model character of category M by a response character of category R. I. e., the list of entries is in this order:

(m1 r1) (m1 r2) (m1 r3) (m1 r4) (m1 r5) (m2 r1) (m2 r2) (m2 r3) ....

`theMarkUpCharInfo` Defines character properties such as case, base character, diacritic, and phonetic category. If you are using special character sets or special alphabets or keyboards, you may need to change this information. The values you provide here will be read into various PASCAL arrays internal to the MarkUp XFCN code resource. How to change these tables is described in the next section.

CHANGING THE CHARACTER INFORMATION TABLES

As explained above, the default character information tables may be modified by placing information into the global variable `theMarkUpCharInfo`. However, the information there must be formatted in a precise way before it can be used by MarkUp. Each HyperCard line in `theMarkUpCharInfo` must contain information of one of four types: base character, diacritic, case, or phonetic.

IMPORTANT: Each HyperCard line of `theMarkUpCharInfo` will be placed in a PASCAL string; hence *no line should ever exceed 255 characters*. If you have too much information to fit on one line, use additional lines for the remaining information.

The format for each type of information is as follows:

base character info: b,CHAR,x x x x ...

Here "b" is a switch which informs MarkUp that the following information concerns base character. CHAR is a base character, and x x x x ... stands for a list of characters with diacritics which have char as their base character, e. g.,

b,e,é è ê ë É
b,i,í ì ï î
b,c,ç

Notice that case, like base and diacritic is an character attribute, so all base characters should be specified as lower case characters.

diacritic info:

d,DIACRIT,x x x x ...

Here "d" is a switch which informs MarkUp that the following information concerns diacritics. DIACRIT must be one of the diacritic values: `acute`, `grave`, `circumflex`, `dieresis`, `supero`, `cedilla`, `tilde`, `macro`. x x x x ... stands for a list of characters which have that type of diacritic, e. g.,

d,acute,á é í ó ú
d,grave,à è ì ò ù
d,cedilla,ç Ç
d,dieresis,ä ë ï ö ü

case info:

c,CASE,x x x x ...

Here "c" is a switch which informs MarkUp that the following information specifies character case. CASE must be one of the two case values up_case or down_case, and x x x x ... is l list of characters that have that attribute, e. g.,

c,up_case,A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
c,down_case,a b c d e f g h i j k l m n o p q r s t u v w x y z 0 1 2 3 4 5 6 7 8 9

phonetic info:

p,PHON,x x x x ...

Here "p" is a switch which informs MarkUp that the following information specifies "phonetic" properties. PHON is one of the five values `vowel`, `consonant`, `phon3`, `phon4`, or `phon5`; the list x x x x ... is a list of the characters which have that attribute, e. g.,

p,vowel,a e i o u y
p,consonant,b c d f g h j k l m n p q r s t v w x z

Phonetic information is used to adjust the edit distances assigned for mismatched letters.

Regardless of the type of information, the first two items in each line must be separated by commas. The remaining entries in the line may be run together or separated by one or more spaces for readability (as in these examples). There can be any number of lines, and the different types of information can be mixed in any order.

INFORMATION RETURNED BY MARKUP

The MarkUp XFCN directly returns the graphical markup string, which can simply be displayed beneath the student's response. (Note, however, that unless the markUpNeeded input value was `True`, MarkUp will return an empty string.)

If there was some problem which prevented MarkUp from carrying out judging in the usual way, the judging will be aborted and an error message will be returned in place of the usual string. This error message will give a brief description of the nature of the problem and *will always be prefaced by a single "%" character*. This will be true even if no markup was requested. Hence, HyperCard can look at the first character of the MarkUp XFCN return to see whether the return is an actual markup string or an error message and act accordingly.

The MarkUp XFCN may also return information in four other global HyperCard variables:

`theMarkUpReturnValues`	A list of comma-separated items. Usually, the first item will be "True" (if the response was judged OK), or "False" (if it was judged NO). The remaining items contain additional information about the match. (If `spellingOnlyNeeded` is not `"x"`, however, the return will be different.) *This information is returned every time MarkUp is called.*
`theMarkUpMaps`	If markup maps were requested, they are placed in this variable, a response-to-model map in the first line, and a model-to-response map in the second line. This information is returned only if the input parameter `markUpMapNeeded` is set to `True`.
`theMarkUpParamDisplay`	If a display if judging parameters was requested by setting `parameterDisplayNeeded` one of the non-default options, then the requested information is returned in this variable.
`theMarkUpDebug`	Reports technical information on the operation of Markup. This global is intended for development purposes and is created by MarkUp only if the `debugNeeded` is turned on.

IMPORTANT: If you have not requested the map or parameter display information, then the values of the three global variables `theMarkUpMaps`, `theMarkUpParamDisplay` and `theMarkUpDebug` *are left unchanged*. Specifically, they will *not* be set to `empty`, and the information they contain may be out of date. It is up to your HyperCard script to make sure that any information you read from these globals is up to date. In contrast, the values in `theMarkUpReturnValues` are updated each time MarkUp is called, whether you request it or not, hence they are always current.

The information returned in these global variables can be inspected visually with the HyperTalk debugger facility, or by PUTing a copy into a field. Or it can be read directly by your scripts. Neither you nor your program will be able to make any sense out of the information returned, however, unless you know how it is formatted. The details of formatting are explained in the following paragraphs.

`theMarkUpReturnValues` always returns information about the match between model and response. *The nature of the information, however, depends on the value of the input parameter `spellingOnlyNeeded`.*

If `spellingOnlyNeeded` was `"x"` (the normal case), indicating a standard word-to-word matching, then `theMarkUpReturnValues` returns a comma-separated list of four values: judgedOk, pMatched, pNoninversion, and aveDist. The meaning of these items (in the order they appear in the returned list) is:

judgedOk	This item will be "true" if the response matched the model, or "false" if it did not. A match is defined relative to the current values of tolerance for misspellings, extra words, and word order. This return value can be used to make decisions about feedback and branching after a response has been judged.

However, if something goes wrong during the judging process, so that judging could not be completed in the normal manner, judgedOk will be set to "false". Consequently, you should not use this value to make instructional decisions without also checking the markup string to see if an error occurred.

pMatched — This value, which ranges from 0.0 to 1.0, measures the proportion of words matched. It is computed by dividing the number of matched words by the total number of word types (non-identical words) in the model and response combined, excluding ignorable words. Equivalently, it may be thought of as the cardinality of the intersect of the set of model words and the set of response words, divided by the cardinality of their union.

pNoninversion — This value, which ranges from 0.0 to 1.0, measures the proportion of words which are in correct order by dividing the number of inversions in the solution into the total number of non-ignorable words in the response. Unmatched words (including ignorable words) are excluded when computing this inversion count.

aveDist — This value, which ranges from 0.0 to 1.0, is computed as the average edit distance between model-response words pairs which were actually matched. It provides a measure of how well the model fits the response with respect to spelling.

If the value of `spellingOnlyNeeded` was `"r"` or `"p"` which simply requests a least-cost edit trace for the model and response strings, then `theMarkUpReturnValues` returns a list of two comma-separated items: rawEditDistance, normalizedEditDistance

Remember that the values in theMarkUpReturnValues are returned automatically each time you call MarkUp. You do not need to request this information, and indeed you cannot prevent it from being computed and returned.

`theMarkUpMaps` returns information about how the response words are matched with words in the model. An example will clarify this:

```
Model:      The quick brown fox [jumped leaped] over the lazy dog
Response:   The brown quick fox walked over the big lazy dog.        (19)
```

If a markup map is requested for this model and response, the value returned in `theMarkUpMaps` will be

```
1,3,2,4,0,6,7,0,8,9
1,3,2,4,0,6,7,9,10
1,5,11,17,21,28,33,37,41,46                                          (20)
```

The first line of (20) is a response-to-model map. It will have as many comma-separated items as there are words in the response. The first position of this list corresponds to the first response word, the second position to the second response word, and so on. The number in each position tells which model word is paired with that position. In case no model word is paired with a particular response word, 0 is returned in that position. Thus, in the above example, line 1 indicates that response word 1 goes with model word 1, response word 2 goes with model word 3, response word 3 goes with model word 2, response word 5 is unmatched, and so on.

The second line of (20) is a model-to-response map. It will have as many comma-separated items as there are word positions in the model (a synonym list counts as one word position, and an ignorable word list

does not count at all). The first list position corresponds to the first model word, the second to the second model word, and so on. The value in each position tells which response word is paired with that position in the model. If no response word is paired, this fact is represented by the presence of "0" in that position. In the example above, line 2 indicates that model word 1 is associated with response word 1, model word 2 with response word 2, model word 3 with response word 2, model word 5 is unmatched and so on

The information in the two maps obviously overlaps, and in fact when there are no missing or extra words they give identical information. But because either model or response words may remain unpaired, both maps are needed to completely specify the match.

The third line of (20) contains information about the starting character number of each response word, as computed by MarkUp. For example, the first word occupies characters 1-4 of the response, the second word characters 5-10, and so on. This is different from the HyperCard definition of a word, because MarkUp includes the space or other punctuation mark immediately preceeding a word as belonging to that word, as well as trailing spaces up to the next word. These pointer can be used not only to pull individual "words" out of the response string, but also the markup substring which corresponds to that word out of the markup string. Note, however, that the markup string has an extra leading character (to accomodate a symbol for missing words at the beginning of the response), and possibly an extra trailing character (to accommodate a symbol for missing words or letters at the end).

The global variable `theMarkUpParamDisplay` will return different kinds of information depending on how you set `parameterDisplayNeeded` when MarkUp was called. In every case, there will be at least two lines. The first will consist of the string "MUParamDisplay" immediately followed by a single letter which identifies the type of information. Successive lines contain the actual information. (The following examples show the information which will be returned when all the default values are in effect.)

If you use "v" as an input value of `paramDisplayNeeded`, the return will be of this form,

```
MUParamDisplay v
Markup XFCN 2.0   18 Aug 93, 12:47 PM  -  R. Hart UI/UC Language Learning Laboratory
```

the second line of which identifies the MarkUp XFCN version number and by date and time of compilation.

If you use "b" as an input value of `paramDisplayNeeded`, you will get back information in `theMarkUpParamDisplay` about the base characters corresponding to various characters, in this format:

```
MUParamDisplay b
Ä=a,Å=a,Ç=c,É=e,Ñ=n,Ö=o,Ü=u,á=a,à=a,â=a,ä=a,ã=a,å=a,ç=c,é=e,è=e,ê=e,ë=e,í=i,ì=i,î=i,ï=
i,ñ=n,ó=o,ò=o,ô=o,ö=o,õ=o,ú=u,ù=u,û=u,ü=u,À=a,Ã=a,Õ=o,ÿ=y,Ÿ=y,Â=a,Ê=e,Á=a,Ë=e,È=e,Í=i,
Î=i,Ï=i,Ì=i,Ó=o,Ô=o,Ò=o,Ú=u,Ù=u,Û=u
```

Here each HyperCard item is of the form C=B. This denotes that the base character of C is B. Only those characters which are actually accented appear in this list. If a character is not in the list, this means that is has no accent and thus is its own base character.

If you use "d" as an input value of `paramDisplayNeeded`, then you will get back information in `theMarkUpParamDisplay` about the diacritic of each letter, in a format similar to that used for base character:

```
MUParamDisplay d
Ä=4,Ç=7,É=1,Ñ=8,Ö=4,Ü=4,á=1,à=2,â=3,ä=4,ã=8,ç=7,é=1,è=2,ê=3,ë=4,í=1,ì=2,î=3,ï=4,ñ=8,ó=
1,ò=2,ô=3,ö=4,õ=8,ú=1,ù=2,û=3,ü=4,À=2,Ã=8,Õ=8,ÿ=4,Ÿ=4,Â=3,Ê=3,Á=1,Ë=4,È=2,Í=1,Î=3,Ï=4,
Ì=2,Ó=1,Ô=3,Ò=2,Ú=1,Ù=3,Û=2
```

Each HyperCard item will be of form C=N, where C is a character, and N is an integer between 1 and 14. Each integer represents the value of a diacritic, thus

0	no_accent
1	acute
2	grave
3	circumflex
4	dieresis
5	umlaut
6	supero
7	cedilla
8	tilde
9	subdot
10	superdot
11	subhat
12	superhat
13	subhook
14	macron

This set of diacritics is hard-coded into the PASCAL program for MarkUp and cannot be modified without changing the list of `diacritic_variants` in the PASCAL global `TYPE` declaration.

Characters which are not accented will not appear on the list. If a character is not on the list, this means that it has the default diacritic type 0 = `no_accent`.

If you use "`c`" as the input value of `paramDisplayNeeded`, then you will get back information in `theMarkUpParamDisplay` in this form:

```
MUParamDisplay c
A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S,T,U,V,W,X,Y,Z
```

The second line is a comma-separated list of all the characters which are currently classified as upper-case. Only the upper-case characters are displayed. If a character is not on this list, it is classified as a lower-case character.

If the input value of `paramDisplayNeeded` was "`h`", then information about the phonetic value of the characters will be returned in `theMarkUpParamDisplay`:

```
MUParamDisplay h
 =1, =1, =1, =1, =1,\=1, =1, =1, =1, =1, =1,np=1,para=1, =1, =1, =1, =1, =1, =1, =1,
 =1, =1, =1, =1, =1, =1, =1, =1, =1,--=1,--=1, =1,!=1,"=1,#=1,$=1,%=1,&=1,'=1,(=1,)=1,
*=1,+=1,,=1,h=1,.=1,/=1,0=1,1=1,2=1,3=1,4=1,5=1,6=1,7=1,8=1,9=1,:=1,;=1,<=1,==1,>=1,
?=1,@=1,A=0,B=1,C=1,D=1,E=0,F=1,G=1,H=1,I=0,J=1,K=1,L=1,M=1,N=1,O=0,P=1,Q=1,R=1,S=1,
T=1,U=0,V=1,W=1,X=1,Y=0,Z=1,[=1,\=1,]=1,^=1,_=1,`=1,a=0,b=1,c=1,d=1,e=0,f=1,g=1,h=1,
i=0,j=1,k=1,l=1,m=1,n=1,o=0,p=1,q=1,r=1,s=1,t=1,u=0,v=1,w=1,x=1,y=0,z=1,{=1,|=1,}=1,
~=1, =1,Ä=0,Å=0,Ç=1,É=0,Ñ=1,Ö=0,Ü=0,á=0,à=0,â=0,ä=0,ã=0,å=0,ç=1,é=0,è=0,ê=0,ë=0,í=0,
ì=0,î=0,ï=0,ñ=1,ó=0,ò=0,ô=0,ö=0,õ=0,ú=0,ù=0,û=0,ü=0,†=1,°=1,¢=1,£=1,§=1,•=1,¶=1,ß=1,
®=1,©=1,™=1,´=1,¨=1,≠=1,Æ=0,Ø=0,∞=1,±=1,≤=1,≥=1,¥=1,µ=1,∂=1,∑=1,∏=1,π=1,∫=1,ª=1,º=1,
Ω=1,æ=0,ø=0,¿=1,¡=1,¬=1,√=1,ƒ=1,≈=1,Δ=1,«=1,»=1,…=1, =1,À=0,Ã=0,Õ=0,Œ=0,œ=0,–=1,—=1,
“=1,”=1,‘=1,’=1,÷=1,◊=0,ÿ=0,Ÿ=0,⁄=1,¤=1,‹=1,›=1,fi=1,fl=1,‡=1,·=1,,=1,„=1,‰=1,Â=0,Ê=0,
Á=0,Ë=0,È=0,Í=0,Î=0,Ï=0,Ì=0,Ó=0,Ô=0, =1,Ò=0,Ú=0,Û=0,Ù=0,ı=1,ˆ=1,˜=1,¯=1,˘=1,˙=1,˚=1,
¸=1,˝=1,˛=1,ˇ=1
```

Here the second line consists of a list of comma separated items of form C=P. C is a character and P is an integer value which corresponds the phonetic category of C. The current possible values of P are

0	vowel
1	consonant
2	phon3

3 phon4
4 phon5

The phonetic value of every character is returned, in character-code order. In courier and most other fonts, the first 32 characters are control characters and will display as blank boxes. The 44th character, which is the comma, displays this way

```
..      ,,=1,    ...
```

thus creating a spurious empty item at the 44th position. This must be taken into account when using the HyperCard ITEM chunk designator to parse this information.

If you enter "p" as the value of `paramDisplayNeeded` when you call MarkUp, then you will get back in `theMarkUpParamDisplay` a list of all the characters that count as punctuation. This example displays the default value for the set of punctuation characters:

```
MUParamDisplay p

 !(),.:;<>?[]
```

The list occupies *two* lines because the first character is a return character which displays as a carriage return/line feed. The second character is a space, and the third an exclamation mark.

If you enter "w" as the value of `paramDisplayNeeded` when you call MarkUp, then a list of nine comma-separated items will be returned in `theMarkUpParamDisplay`:

```
MUParamDisplay w
20,20,30,20,1,1,0.35,0.67,0.2
```

These nine items represent the values of the following parameters which are used in the spelling analysis (the values shown in this example are the default values):

item 1	winsert	20
item 2	wdelete	20
item 3	wchange	30
item 4	wtranspose	20
item 5	wcap	1
item 6	waccent	1
item 7	cutoff	0.67
item 8	prop_errors	0.35
item 9	runon_criterion	0.2

Entering a value of "f" for `paramDisplayNeeded` will cause `theMarkUpParamDisplay` to contain information on the various judging flags in this format:

```
MUParamDisplay f
false,false,false,true,true,true,true,false,f
```

The second line contains a list of nine comma-separated items which control the way judging is performed

item 1	anyOrderOk
item 2	extraWordsOk
item 3	misspellOk
item 4	wordMarkUpNeeded
item 5	runtogetherNeeded
item 6	adjustNeeded
item 7	shortCut
item 8	markUpMapsNeeded
item 9	paramDisplayNeeded

Finally, if you use `"s"` as an input value of `paramDisplayNeeded`, then `theMarkUpDisplayParams` will contain a 12-character list of all the markup symbols:

```
MUParamDisplay s
+-~XΔ«x\=><[
```

The meaning of a character depends on its position in the list:

char 1	addcap	"_"	underscore
char 2	dropcap	"‾"	underscore
char 3	accenterror	"~"	tilde
char 4	extraword	"X"	capital x
char 5	missingword	"Δ"	capital delta
char 6	moveword	"«"	double leftward arrow
char 7	extraletter	"x"	lower case x
char 8	missingletter	"\"	backslash
char 9	substituteletter	"="	equal sign
char 10	transposeletter1	">"	left angle bracket
char 11	transposeletter2	"<"	right angle bracket
char 12	runonword	"["	left square bracket

If you use "m" is the value of paramDisplayNeeded, theMarkUpParamDisplay will contain the values of the phon_matrix matrix discussed earlier. Since there are 5 possible character categories, phon_matrix is a 5x5 matrix and contains 25 entries. The first entry corresponds to row 1, column 1; the second entry to row 1, column 2; ... the sixth entry to row 2, column 1, and so on. Row M, column R contains the cost of replacing a character of type M by one of type R:

```
MUParamDisplay m
30,36,36,36,36,36,30,36,36,36,36,36,30,36,36,36,36,36,30,36,36,36,36,36,30
```

EXAMPLE 1: MARKING UP A RESPONSE IN HYPERCARD

Here is a simple annotated example of how to use MarkUp to do the answer judging in your own stack. It assumes that the current card has a card field called "prompt" where a question is displayed, and a second card field called "response" where the student will type in a response to the question. A third card field named "markup" must be located below the "response" field. It will be used to display the markup feedback. If card field "markup" does not exist, it can be created and positioned by executing the setUpMarkUp handler, as explained below.

The following handlers should be placed in the *card* script if MarkUp is only needed on one card. If MarkUp will be needed throughout the stack, put these handlers in the *stack* script and change the `openCard` and `closeCard` handlers to `openStack` and `closeStack` handlers.

Before you can use MarkUp, you must attach the stack which contains the MarkUp XFCN. This stack is named "markUp XFCN 2.0" in the software distribution of MarkUp. Besides the code resource which implements MarkUp, the stack contains in its stack script a number of handlers useful for integrating MarkUp into HyperCard programs.

```
on  openCard

      -- If the markUp XFCN stack is located in some other folder, change
      -- the path accordingly.

      start using stack "myFolder:markUp XFCN 2.0"

      -- This enables use of the markUp XFCN.  Parameter value "response" is
      -- the name of the field where the student will type in a response.
      -- It must be a CARD field.

      setUpMarkUp  "response"

      -- Ask a question to elicit a written response.  In a real drill program,
      -- this would be done somewhere other than in OPENSTACK, e. g., in
      -- the handler which presented the next drill item.

      put "Type in the French words for the numbers 1 to 10." into card field
"prompt"

end  openCard


on  closeCard

      -- markUp XFCN uses a lot of space, so disconnect it as soon as it's not
      -- needed.

      stop using stack "myFolder:markUp XFCN 2.0"

end  closeCard
```

```
on  judgeResponse

       -- This handler contains the commands to to the judging and the markup.
       -- It must be called from the response field.

       -- The following globals MUST be declared in any handler that calls
       -- the MarkUp XFCN, because MarkUp may examine their values with callbacks.

       global theMarkupReturnValues, theMarkUpSymbols, theMarkUpPunctuation, ¬
       the MarkUpParameters, theMarkUpCharInfo, theMarkUpParamDisplay, ¬
       theMarkUpMaps, theMarkUpDebug

       -- Erase any previous markup.
       put empty into card field "markUp"

       -- Copy the correct answer string into a variable.  In a real drill program,
       -- this might be done in the handler that present the item.
       put "un deux trois quatre cinque six sept huit neuf dix" into model

       -- Execute the MarkUp XFCN and store markup string which is returned.
       put markUp( model, card field "response" ) into markUpString

       --Use returned values to generate feedback display.
       if ( item 1 of theMarkUpReturnValues = "True" ) then
              -- If resonse was judged correct, no markup required.
              put "OK" into card field "markUp"
       else
              -- If response had errors, display markup string.
              put markUpString into card field "MarkUp"
       end if

end  judgeResponse
```

In addition the following handlers must be placed in the script of the card field where the response is typed (in this example it will be card field "response"):

```
on  returnInField

       judgeResponse
       show card field "markUp"
end  returnInField


on keyDown ch

       put the selectedchunk into s
       hide card field "markUp"
       select s
       pass keyDown

end keyDown
```

This `returnInField` handler causes response judging to begin as soon as the student presses the RETURN key. The `keyDown` handler makes the markup disappear as soon as the student begins to edit

the response. It is important to do this because when the response string changes, the markup becomes invalid (e. g., if the student deletes letters, some markup symbols may no longer be under the right letters).

Executing the `setUpMarkUp` handler changes the default font of the response field to be "Courier" (but does not change any other text properties). This change to a fixed-width font is required so that the markup symbols will be properly alligned beneath the letters of the response. It is not essential to use Courier; any non-proportional font will do, but `setUpMarkUp` must be modified if you want to use some other font. If you use the MarkUp XFCN only to judge the response and do not intend to display the error markup, then you need not execute `setUpMarkUp` at all.

IMPORTANT: `SetMarkUp` installs the "markup" field behind and slightly below the response field. It assumes that *the response field is only a one line deep, and that it has been shaped so that the bottom of the field is immediately beneath the longest descender.* If your response field is not configured this way, the markup characters may be completely hidden by the bottom of the response field.

For expository simplicity, the `judgeResponse` handler above supposes that the judging proceeded normally. In actual courseware, the markup string should always be checked so that error conditions such as too many letter in a word or words in a sentence can be detected and the student informed that the "NO" judgment was due to a special problem. Whenever there is some problem which prevents judging from being completed, *MarkUp returns, instead of the normal markup, a string which begins with the character "%".* The remainder of the string gives a brief description of the problem. To use this feature, modify the code in `judgeResponse` along these lines:

```
...

--Use returned values to generate feedback display.

if  ( char 1 of theMarkUpString = "%" ) then  -- Check for error.
       delete char 1 of theMarkUpString  -- Get rid of the "%" char.
       answer "There was a problem judging you answer:" & return & ¬
       theMarkUpString & return & "Please try again." with "OK"
else
       if I item 1 of theMarkUpReturnValues = "True" ) then
              -- If resonse was judged correct, no markup required.
              put "OK" into card field "markUp"
       else
              -- If response had errors, display markup string.
              put markUpString into card field "MarkUp"
       end if
end if

...
```

EXAMPLE 2: MODIFYING THE MARKUP AND PUNCTUATION LISTS

The following HyperCard program segment illustrates how the details of judging and the appearance of the markup can be manipulated by resetting the global variables `theMarkUpPunctuation` and `theMarkUpSymbols`:

```
...

global theMarkUpSymbols, theMarkUpPunctuation

...

-- Change some of the markup symbols.  New symbols are:
-- "?" = extra word, "^" = make upcase, "~" = make downcase, "°" = extra letter,
-- "˝" = transposed letters, "-" = incorrect letter, "`" = missing letter.

put "^~-?Δ«°`˝"[" into theMarkUpSymbols

-- Change punctuation so that a hyphen will be treated as a word separator, and
-- the final period will be judged.

put ( "-?!,:;()[]" & space & return ) into theMarkUpPunctuation
...
```

These commands can be executed any time before the MarkUp XFCN is called. They changes which they cause will persist until you reset the global variables again. The above modifications in the markup symbol and punctuation lists will result in markups like the following:

Model: `The quick brown fox jumped over the lazy dog.`

Reponse: `the qick prown foxx jumpde the big-Lazy dög over`
Markup: ` ` ` ` ° ˝"Δ ??? ` ~ « ` (21)

The spelling- and case-error symbols have been chosen so that they are smaller and higher in the line than the defaults. This results in a less cluttered spelling markup and clearer visual distinction between the spelling and the word-order symbols. On the other hand, the meaning of the spelling symbols may be somewhat less obvious. Notice that the omitted period at the end of the response is now marked as a missing character, and that "big" and "lazy" are judged as separate words, in keeping with the changes made to `theMarkUpPunctuation`.

The spelling and word-order markups can easily be separated by changing the values of the input parameters misspellOk, anyOrderOk, and extraWordsOk. This permits the two types of errors to be dealt with separately by modifying the judging in example 1 in this way:

```
...

global theMarkUpReturnValues

...

-- Allow word order and extra word errors.  Such errors will not be judged or marked
-- during this call to MarkUp.

put markUp(model, response,,True,True) into spellMarkUp
put item 1 of theMarkUpReturnValues into wordOrderOk

-- Allow spelling errors.  Such errors will not be judged or marked during
-- this call to MarkUp.

put markUp(model, response,True,,) into wordOrderMarkUp
put item 1 of theMarkUpReturnValues into spellingOk

-- If there were any word order or extra word errors, mark them up.
-- Otherwise, mark any spelling errors.  If no errors of either kind are
-- present, judge OK.

If wordOrderOk = "False" then
       put "First, correct your grammar problems" into card field "feedBack"
       put wordOrderMarkUp into card field "markUp"
else if spellingOK = "False" then
       put "No.  Let's look at your spelling errors." into card field "feedBack"
       put wordOrderMarkUp into card field "markUp"
else
       put "OK" into card field "feedBack"
end if

...
```

EXAMPLE 3: JUDGING WITH MULTIPLE RIGHT AND WRONG ANSWERS

A common instructional situation is for a question to have several alternate correct answers. In addition the courseware author may have anticipated several incorrect answers, each of which requires its own specific feedback. Of course, the student's typed response may not exactly match any of the anticipated (correct or incorrect) answers, due to misspellings and other errors. An adequate response analysis requires matching the response against each of the models and determining which one provides the closest match. The MarkUp XFCN returns three numbers which measure goodness of match: pMatched (percent of words matched), pNoninv (percentage of words in correct order), and aveDist (the average edit distance between words in matched pairs). Whenever MarkUp is called, these numbers are returned as items 2, 3, and 4 of the HyperCard global variable `theMarkUpReturnValues`.

To determine best fit, these numbers must be combined to provide a single goodness-of-fit metric. Of course, a response which is judged "OK", and is thus error-free relative to the current settings of the extraWordsOk, wrongOrderOk and misspelledOk flags, should always fit better than any response which is judged "NO"; beyond this, however, the relative contribution of these three factors in providing an intuitive good fit is not clear. Further empirical study is required but has not yet been undertaken. Lacking data, we can as a first approximation suppose that all three factors are weighted equally, so that the metric will be

goodnessOfFit := 1.0, if judgedOk is True
(3*pMatched *(1 - aveDist) + pNoniv)/4, if judgedOK is False
(22)

Here, 1 has been subtracted from aveDist so that values will range from 0.0 (no fit) to 1.0 (perfect fit), as with the other two quantities. The resultant value of goodnessOfFit will vary between 0 and 1.0, although that is not crucial for the application we are developing here.

Supoose now that we have these data for a single question contained in a card field named "item":

```
Complete this sentence:  Ice cream tastes _____ than spinach.

answer  Ice cream tastes better than spinach
              Yes, I agree.
answer  Ice cream tastes worse than spinach
              I don't think so, but if you think so, OK.
wrong   Ice cream tastes gooder than spinach
              "good" has a special comparative form:  "better".
wrong   Ice cream tastes more good than spinach
              "more" is used to form the comparative of multi-syllable adjectives.
wrong   Ice cream tastes more better than spinach
              Use either "more" or "-er" to form the comparative, not both at once!
#
```

These data are formated as follows: the first line is the prompt. Specifications for the correct and wrong answers, and for feedback, follow on the remaining lines. The end of the data is marked by a "#" symbol in column 1. A correct answer must occupy only one line and is indicated by the word "answer" as the first word of the line. Similarly, a wrong answer is indicated by the word "wrong" as the first word. All the lines that come after an answer but before the next answer (here indented for readability) are the feedback which will be shown if the preceeding answer is the best match for the response.

```
on  showPrompt

       -- Copy data into global variable ITEMDATA and display prompt to student.

       global itemData

       put card field "item" into itemData
       put line 1 of itemData into card field "prompt"

end  showPrompt
```

```
on  returnInField

      -- Compair typed resonse to the answer in global var ITEM and find best match.
      -- Display feedback and markup which goes with best matched answer.
      -- If no aswer is matched, display "NO"

      global itemData

      -- Search itemData for best ans.
      put findBestAns( target, itemData ) into bestAns

      put item 1 of bestAns into bestFit

      if  ( bestFit = 0 ) then
            put "NO" into card field "feedBack"    -- No answer matched.
      else
            put item 2 of bestAns into bestLine
            put item 3 of bestAns into bestMarkUp
            put word 1 of line bestLine of itemData into polarity  -- Ans or Wrong?
            if  ( polarity = "answer" ) then
                  put "OK" into card field "feedBack"  -- Matched correct answer.
            else
                  put "NO" into card field "feedBack"   -- Matched wrong answer.
      end if

      repeat with i = bestLine + 1 to number of lines in itemData -- Find feedback.
            if  ( word 1 of line i of itemData is in "answer wrong #" )
            then exit repeat
      end repeat

      put line bestLine + 1 to i - 1 of itemData after card field "feedBack"

end  returnInField
```

```
function  findBestAns response, itemData

       -- Scan through all correct and incorrect answers in ITEMDATA and find
       --  the one which matches RESPONSE best.
       -- ANSDATA must contain answer & feedback data, formated as shown above.
       -- Return is a list of three comma-separated items:
       -- Item 1:     Goodness value of best-matched answer (0 if no match).
       -- Item 2:     Line number within ITEMDATA of best matched answer.
       -- Item 3:     Markup string which goes with best matched answer

       global theMarkupReturnValues, theMarkUpSymbols, theMarkUpPunctuation, ¬
       the MarkUpParameters, theMarkUpCharInfo, theMarkUpParamDisplay, ¬
       theMarkUpMaps, theMarkUpDebug

       put 0 into bestFit
       put empty into bestLine
       put empty into bestMU

       repeat with i = bestLine + 1 to number of lines in itemData
              put line i of itemData into m
              if  ( word 1 of m = "answer" ) or ( word 1 of m = "wrong" ) then
                     delete word 1 of m
                     put markUp( m, response ) into mu
                     put item 1 of theMarkUpReturnValues into match
                     if ( match = "True" ) then
                            put item 2 of theMarkUpReturnValues into pMatched
                            put item 3 of theMarkUpReturnValues into pNonInv
                            put item 4 of theMarkUpReturnValues into aveDist
                            put ( pMatched + pNonInv + 1 - aveDist ) / 3 into ansFit
                            if  ( ansFit > bestFit ) then
                                   put ansFit into bestFit
                                   put i into bestLine
                                   put mu into bestMU
                            end if
                     end if
              end if
       end repeat

       return ( bestFit & "," & bestLine & "," & bestMU )

end  findBestAns
```

The `findBestAns()` function simply searches through the answer data looking for each "answer" or "wrong" specification. Whenever one is found, it is matched against the response. If there is a match and that match improves on the best of the previous matches, the current match is made the best one. Eventually all the answers are examined and the information about the best matched one is returned.

EXAMPLE 4: USING THE MARKUP MAPS TO CONTROL VOCABULARY HELP

When writing foreign language courseware, a simple graphic markup is often not specific enough as error feedback. If, for example, the student is ignorant of certain vocabulary words required by the response, a missing or unidentified word markup may not provide sufficient help. The handler below uses the markup

maps to see which words in the model are not matched by words in the student's response, and given vocabulary help on just those items. As before, we will suppose three card fields named "prompt", "response", and "markUp", and in addition one called "vocHelp", where vocabulary help will be displayed.

```
on  presentItem

       global correctAns, vocList

       put "Translate to French:    She read the last ten page pages for us." ¬
       into card field "prompt"
       put "Elle nous a lu les dix dernières pages." into correctAns

       -- The items in this list correspond to words in the correct answer.
       put "Elle,nous,avoir,lire (irreg),le/la,dix,dernier,page (f)" into vocList

       setUpMarkUp "response"

end  presentItem


on  judgeResponse


       global theMarkupReturnValues, theMarkUpSymbols, theMarkUpPuntuation, ¬
       the MarkUpParameters, theMarkUpCharInfo, theMarkUpParamDisplay, ¬
       theMarkUpMaps, theMarkUpDebug, correctAns, vocList

       -- Request return of markup maps by setting markUpMapsNeeded to True..
       put markUp( correctAns, target,,,,,,,,True ) into markUpString

       --Use returned values to generate usual markup display.
       if I item 1 of theMarkUpReturnValues = "True" ) then
              -- If response was judged correct, no markup required.
              put "OK" into card field "markUp"
       else
              -- If response had errors, display markup string.
              put markUpString into card field "MarkUp"
       end if

       -- Use markup map to generate additional vocabulary help.
       put line 2 of theMarkUpMaps into MtoRMap  -- Model-to-response map
       repeat with i = 1 to numoer of items in MtoRMap
              if  ( item i of MtoRMap = "0" ) then
                     put ( word i of vocList & return ) into card field "vocHelp"
              end if
       end repeat

end judgeResponse


on  returnInField

       -- This handler must be in the script of card field "response"

       judgeResponse

end  returnInField
```

EXAMPLE 5: USING MARKUP MAPS TO JUDGE LISTS

Many questions solicit answers in the form of a list, for example, "Name the five Great Lakes", or perhaps "Name at least three of the five Great Lakes". In such cases, the order in which items are listed is not relevant, only the fact that they are present somewhere in the response. The first case, "Name the five Great Lakes" can be easily provided for by setting anyOrderOk (the 5th parameter slot) to `True` when the MarkUp XFCN is called:

```
get markUp( "Michigan Superior Huron Algonquin Ontario", target,,,True )
```

Missing words, extra words, and misspellings will still be marked appropriately, but any order at all will be accepted. Punctuation, as usual, will be ignored when doing the judging, so the student may use spaces or any other kind of punctuation to separate words. Note, however that anyOrderOk operates on individual words, not phrases, so that a question like "Name the Dakotas" cannot be reliably judged using

```
get markUp("North Dakota South Dakota", target,,,True)
```

because answers like "South Dakota Dakota North" will be judged as correct. There is no way to define or manipulate phrases in version 2.0 of the MarkUp XFCN.

The response to a question like "Name at least three of the 5 Great Lakes" can be handled efficiently with the help of the markup maps, using handlers like these:

```
function  countInstances  list, object

       -- LIST must be a list of comma-separated items.
       -- OBJECT is a value to match to each item (case and accent ignored).
       -- Return is number of times OBJECT occurred as an item in LIST.

       put 0 into count
       repeat with i = 1 to number of items in list
              if  ( item i of list = object ) then add 1 to count
       end repeat

       return count

end  countInstances
```

```
on  judgeResponse

        global theMarkupReturnValues, theMarkUpSymbols, theMarkUpPunctuation, ¬
        the MarkUpParameters, theMarkUpCharInfo, theMarkUpParamDisplay, ¬
        theMarkUpMaps, theMarkUpDebug

        put "Michigan Superior Huron Algonquin Ontario" into correctAns

        -- Set anyOrderOk and markUpMapsNeeded to True.
        put markUp( correctAns, target,,True,,,,,,True ) into markUpString

        put line 1 of theMarkUpMaps into RtoMMap    -- Response-to-model map.
        put line 2 of theMarkUpMaps into MtoRMap    -- Model-to-response map.
        put countInstaces( MtoRMap, "1" ) into numCorrect
        put countInstaces( RtoMMap, "0" ) into NumIncorrect

        if ( numCorrect >= 3 ) and ( numIncorrect = 0 ) then
                put "OK" into card field "feedBack"
        else
                put "NO" into card field "feedBack"
                put markUpString into card field "markUp"
        end if

end judgeResponse
```

The function `countInstaces()` is used here to count both the number of lakes which are matched and the number of response words which are unmatched and thus do not correspond to any lake. If three or more lakes were matched, and there were no incorrect lake names, then the student has successfully answered the question. Otherwise, the markup is shown; this will mark any incorrect lake names as extra words.

EXAMPLE 6: A COMPLETE VIEW OF JUDGING PARAMETERS

When using MarkUp to develop new courseware, it is sometimes convenient to collect and view information on all of the judging parameters. This can be easily done with the following HyperCard function:

```
function  fullParamInfo

        global theMarkupReturnValues, theMarkUpSymbols, theMarkUpPunctuation, ¬
        the MarkUpParameters, theMarUpJudgingTables, theMarkUpParamDisplay, ¬
        theMarkUpMaps

        put "nbdchpvfm" into typeList
        put empty into r

        repeat with i = 1 to length( typeList )
                get markUp(,,,,,,,,,,,( char i of typeList ) ) -- Puts info into global.
                put ( theMarkUpParamDisplay & return & return ) after r
        end repeat

        return r

end  fullParamInfo
```

Notice that the MarkUp XFCN is called with null model and response strings; since the object here is not to get a markup, but to retrieve information about the current markup parameters, entering strings to be judged is unnecessary. Only the 12th parameter, which specifies the type of information to be returned, is systematically varied by having the loop step through the list "nbdchpvfm". Each call returns a result which is appended to the temporary variable `r`. Calling this function and putting the return value into a field, e. g.,

```
put fullParamInfo() into card field "parameterDisplay"
```

will yield a formated display like this one:

```
MUParamDisplay n
Markup XFCN  18 Aug 93, 12:47 PM  -  R. Hart UI/UC Language Learning Laboratory

MUParamDisplay b
Ä=a,Å=a,Ç=c,É=e,Ñ=n,Ö=o,Ü=u,á=a,à=a,â=a,ä=a,ã=a,å=a,ç=c,é=e,è=e,ê=e,ë=e,í=i,ì=i,î=i,ï=
i,ñ=n,ó=o,ò=o,ô=o,ö=o,õ=o,ú=u,ù=u,û=u,ü=u,À=a,Ã=a,Õ=o,ÿ=y,Ÿ=y,Â=a,Ê=e,Á=a,Ë=e,È=e,Í=i,
Î=i,Ï=i,Ì=i,Ó=o,Ô=o,Ò=o,Ú=u,Û=u,Ù=u

MUParamDisplay d
Ä=4,Ç=7,É=1,Ñ=8,Ö=4,Ü=4,á=1,à=2,â=3,ä=4,ã=8,ç=7,é=1,è=2,ê=3,ë=4,í=1,ì=2,î=3,ï=4,ñ=8,ó=
1,ò=2,ô=3,ö=4,õ=8,ú=1,ù=2,û=3,ü=4,À=2,Ã=8,Õ=8,ÿ=4,Ÿ=4,Â=3,Ê=3,Á=1,Ë=4,È=2,Í=1,Î=3,Ï=4,
Ì=2,Ó=1,Ô=3,Ò=2,Ú=1,Û=3,Ù=2

MUParamDisplay c
A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S,T,U,V,W,X,Y,Z

MUParamDisplay h
 =1, =1, =1, =1, =1,\=1, =1, =1, =1, =1, =1,=1, =1, =1, =1, =1, =1, =1, =1, =1, =1,
 =1, =1, =1, =1, =1, =1, =1, =1, =1,=1, =1,!=1,"=1,#=1,$=1,%=1,&=1,'=1,(=1,)=1,*=1,
+=1,,=1,=1,.=1,/=1,0=1,1=1,2=1,3=1,4=1,5=1,6=1,7=1,8=1,9=1,:=1,;=1,<=1, = 1,>=1,?=1,@=1
,A=1,B=1,C=1,D=1,E=1,F=1,G=1,H=1,I=1,J=1,K=1,L=1,M=1,N=1,O=1,P=1,Q=1,R=1,S=1,T=1,U=1,
V=1,W=1,X=1,Y=1,Z=1,[=1,\=1,]=1,^=1,_=1,`=1,a=0,b=1,c=1,d=1,e=0,f=1,g=1,h=1,i=0,j=1,
k=1,l=1,m=1,n=1,o=0,p=1,q=1,r=1,s=1,t=1,u=0,v=1,w=1,x=1,y=0,z=1,{=1,|=1,}=1,~=1, =1,
Ä=1,Å=1,Ç=1,É=1,Ñ=1,Ö=1,Ü=1,á=1,à=1,â=1,ä=1,ã=1,å=1,ç=1,é=1,è=1,ê=1,ë=1,í=1,ì=1,î=1,
ï=1,ñ=1,ó=1,ò=1,ô=1,ö=1,õ=1,ú=1,ù=1,û=1,ü=1,†=1,°=1,¢=1,£=1,§=1,•=1,¶=1,ß=1,®=1,©=1,
™=1,´=1,¨=1,≠=1,Æ=1,Ø=1,∞=1,±=1,≤=1,≥=1,¥=1,µ=1,∂=1,Σ=1,∏=1,π=1,∫=1,ª=1,º=1,Ω=1,æ=1,
ø=1,¿=1,¡=1,¬=1,√=1,ƒ=1,≈=1,Δ=1,«=1,»=1,…=1, =1,À=1,Ã=1,Õ=1,Œ=1,œ=1,–=1,—=1,“=1,”=1,
‘=1,’=1,÷=1,◊=1,ÿ=1,Ÿ=1,⁄=1,¤=1,‹=1,›=1,fi=1,fl=1,‡=1,·=1,,=1,„=1,‰=1,Â=1,Ê=1,Á=1,Ë=1,
È=1,Í=1,Î=1,Ï=1,Ì=1,Ó=1,Ô=1,=1,Ò=1,Ú=1,Û=1,Ù=1,ı=1,ˆ=1,˜=1,¯=1,˘=1,˙=1,˚=1,¸=1,˝=1,
˛=1,ˇ=1

MUParamDisplay p

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

MUParamDisplay v
20,20,30,20,1,1,0.35,0.67,0.2

MUParamDisplay f
false,false,false,true,true,true,true,false,f

MUParamDisplay m
---~XΔ«x\=><[

MUParamDisplay t
30,36,36,36,36,36,30,36,36,36,36,36,30,36,36,36,36,36,30,36,36,36,36,36,30
```

Even though the actual parameter information may wrap around several screen lines, it occupies only one HyperCard line. The single exception to this is MUParamDisplay p, the list of punctuation symbols, which will occupy *two* lines whenever the RETURN character is on the list of punctuation symbols.

Usually, one collects parameter information to inspect it visually and verify that the parameter values are as intended. Sometimes, however, the program itself may need to access the parameter values. If so, they can be readily extracted. The fact that each type of information occupies one line simplifies the process of parsing out information from the display. The following HyperCard function, which returns the base character corresponding to a specified input character, will work on either the simple base character or an aggregate of data like that shown just above.

```
function  findCharBase  ch, paramData

       -- PARAMDATA must contain formatted parameter display information.
       -- Return is the base character corresponding to character CH.
       -- If base info is not in the PARAMDATA, return EMPTY.

       repeat with i = 1 to number of lines in paramData
              if  ( line i of paramData = "MUParamDisplay b" ) then
                     put ( line i + 1 of paramData ) into bList  -- Base char info
                     put offset( ch, bList ) into p
                     if  p > 0     -- CH is in the list.
                     then return char p + 2 of bList     -- Every entry has 3 chars.
                     else return ch   -- CH is not in list, so is its own base char.
              end if
       end repeat

       return empty     -- Info on base chars not in data.

end  findCharBase
```

When card field "parameterDisplay" contains the data shown above, calling this function with input parameter "é"

```
get findCharBase( "é", card field "parameterDisplay" )
```

will return "e", which is the base character corresponding to "é".

Accent information can be parsed out in the same way. Information on case, punctuation, and phonetics of any given character can be retrieved similarly. The following function is a predicate which will return `True` if CH is upper-case and `False` if it is lower-case:

```
function  charIsUpperCase  ch, paramData

      -- PARAMDATA must contain formatted parameter display information.
      -- Return is TRUE if CH is upper-case; FALSE if it is lower-case.
      -- If case info is not in PARAMDATA, return EMPTY.

      repeat with i = 1 to number of lines in paramData
            if  ( line i of paramData = "MUParamDisplay c" ) then
                  put ( line i + 1 of paramData ) into cList  -- Case info
                  return ( offset( ch, cList ) > 0 )  -- Check if CH is on cLIST.

            end if
      end repeat

      return empty      -- Info on case not in data.

end  charIsUpperCase
```

Assuming the data shown above, this call

```
get charIsUpperCase( "A", card field "parameterDisplay" )
```

will return `True`.

EXAMPLE 7: USING THE EXACT SPELLING MARKUP

Spelling is a skill which associates with the auditory image of each word a suitable visual image. The irregularities of English spelling do not in general permit the visual (graphic) associate to be fully predicted from the auditory form, so a student must often visually encode the conventional graphical icon which corresponds to each word, paying special attention to those areas of the visual image which are not predicable from phoneme-to-grapheme rules (Simon & Simon, 1973; Simon, 1975). To facilitate visual coding, it is important that the student not see incorrectly spellings, since these may be encoded into long term memory and interfer with the correct coding. To facilitate visual coding it is also useful to focus the student's visual attention on those areas of the word which have not yet been encoded correctly (i. e., which were misspelled). The `correctSpelling` function below returns a display designed to satisfy these two specifications. Incorrect and omitted letters appear as capitals, so that the student can focus attention on those areas of the word; extra letters in the student's response are replaced by "•", which is relatively inconspicuous, but warns the student that her visual image was not correct in this region. (This English example neglects the possibility of capitalization or accent errors.) For example, here are the displays returned by various misspellings of the word "necessary":

Response: `nesessarey`

Display: `neCessar•y`

Response: `neccisary`

Display: `nec•EsSary`

Since the mixture of upper- and lower-case letters is a very unusual image of word spelling, such display string should probably be further processed in HyperCard so that the upper-case letters are changed to lower-case boldface, sized, and perhaps color coded and displayed via 32-bit quickdraw. The missing letter symbol

can be further minimized by hilighting the letters on each side of the omission, then deleting the omitted character to give displays like these:

neCessaIy

neCesSary

Such displays could be useful as feedback in a program that taught spelling by dicating individual words to students (preferably in the context of a full sentence).

```
function  correctSpelling model, response

       global theMarkUpReturnValues

       put markUp(model, response,,,,,,,,,,"r") into markUpString
       put item 1 of theMarkUpReturnValues into judgement

       if judgment = "False" then
              put 1 into m
              put 1 into r
              put empty into w
              repeat with i = 1 to Length( markUpString )
                     put char i of markUpString into c
                     if c = "-" then  -- Chars match.
                            put char r of response after w
                            add 1 to m
                            add 1 to r
                     else if c = "\" then  -- Missing char.
                            put upCase( char m of model ) after w
                            add 1 to m
                     else if c = "x" then  -- Extra char.
                            put "•" after w
                            add 1 to r
                     else if c = "=" then  -- Wrong char.
                            put upCase( char m of model ) after w
                            add 1 to m
                            add 1 to r
                     else if c = ">" then  -- Transposition.
                            put upCase (char m + 1 of model ) after w
                            put upCase (char m of model ) after w
                            add 2 to m
                            add 2 to r
                     else if c = "<" then  -- Already handled at ">".
                     end if
              end repeat
       end if

       return w

end  correctSpelling
```

```
function  upCase c

      -- Returns the upper case version of an alphabet letter C.
      -- If C is not a lower-case alphabet letter, return C unchanged.

      if "a" <= c and ( c <= "z" )
      then return char(  charToNum( c ) - charToNum("a") + charToNum("A") )
      else return c

end   upCase
```

Since individual words are involved, the MarkUp XFCN is called with the rawTrace parameter (slot 13) set to "r". This forces the response to be analyzed as a single string, even if there are leading, trailing, or internal spaces (leading and trailing spaces should be removed by your HyperCard script before the response is submitted to MarkUp). The raw edit trace is needed here to force a markup to be computed even when the misspelling is very bad, and because complete information about the misspelling is required to generate an accurate spelling correction. The markup characters are then processed one at a time. Whereever the response contains a missing or incorrect letter, the uppercase equivalent in the model is substituted, and whereever there is an extra letter in the response, it is replaced by a "•" character. When a character in the model matches the response, then it is shown in lower-case.

TECHNICAL DETAILS AND LIMITATIONS

The most current version of MarkUp (the one documented by this report) is Markup XFCN 3.0 - 19 Dec 94, 8:20PM. To determine the version you have, execute `MarkUp()` without parameters; a version string will be returned. The revision history of MarkUp can be found near the beginning of the PASCAL source file. The MarkUp XFCN is compiled as a THINK PASCAL project containing the following files, in the indicated order:

DRVRRuntime.lib
Interface.lib
HyperXCmd.p
HyperXLib.lib
MarkUpXFCN3.p

It has been tested on a Macintosh Quadra 700, running under the Macintosh Finder 7.1 and 7.5 and HyperCard 2.1 and 2.2 launched with 2 megabytes of memory. It should, however, run under virtually any Macintosh configuration. The user should be aware of the following limitations on the MARKUP XFCN:

Versions through 3.0 will *not* work properly with 16-bit char representations, i. e., with the Macintosh 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 MarkUp code has to run in the limited space provided by the HyperCard stack. 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 type 28 (stack has moved into application heap), 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."

No formal speed testing was done since, even for maximum length sentences, MarkUp returns without discernable delay.

AVAILABILITY

The MarkUp XFCN is freeware. It can ordered on diskette for a handling fee or accessed from FTP. Contact the author for further information. Copyright of MarkUp resides with the author, but you may use as a component of commercial or non-commercial software. If you do so, acknowledgment of the author and the Language Learning Laboratory of the University of Illinois at Urbana-Champaign would be appreciated. As freeware, MarkUp 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

REFERENCES

Hart, R. (1989) Algorithms for the dynamic identification of spelling and word order errors in student responses. Technical Report No. LLL-T-15-89, University of Illinois Language Learning Laboratory, Urbana IL.

Rumelhart, D. & Norman, D. (1982) Simulating a skilled typist: a study of skilled cognitive-motor performance. *Cognitive Science*, 6, 1:36.

Simon, H. & Simon, D. (1973) Alternative uses of phonemic information in spelling. Review of Educational Research, 43, 115-136.

Simon, D. (1975) Spelling: a task analysis. Learning Research and Development Center Technical Report LRDC-1975-3. Pittsburg U., Pittsburg, PA.

APPENDIX 1: LISTING OF MARKUP XFCN

```
{  Spelling and word order markup utility, Version 3.0  }
{  Implemented as HyperCard XFCN in Macintosh THINK PASCAL, Version 4.0.2  }

{  Robert S. Hart   UI/UC Language Learning Laboratory   13 April 1994  }

{  Copyright 1993-4 by Robert S. Hart. }



{  Spelling markup done by dynamic programming algorithm which generates a markup  }
{  corresponding to a least-cost editing trece.  Editing operations are restricted  }
{  to omission of a letter, insertion of an extra letter, substitution of one  }
{  letter for another, or transposition of two adjacent letters.  }

{  Capitalization and accent errors are also identified and marked. The user may  }
{  specify the way in which capitalization disagreements will be treated:  exact  }
{  agreement required, capital required if the model has one, or capitalization  }
{  differences ignored.  }

{  Run-together words are identified as such if they are adjacent in the model.  }
{  Some misspelling is tolerated in one or both run-togethers.  }

{  Order analysis identifies extra words, missing words, and misplaced words.  }
{  The user can specify various degrees of tolerance when defining what constitutes  }
{  a match of the model:  spelling errors can be excused; incorrect word order can  }
{  be excused, and extra words in the response can be excused.  }

{  The order analysis returns three goodness of fit measures:  proportion of  }
{  matched words, proportion of words in correct order, and average amount of  }
{  misspelling per matched word.  }

{  When specifying a correct answer, the author is allowed to specify one or  }
{  more words which will be ignored if they occur in the student's response.  }
{  Such a word or list must be surrounded by angle brackets, < >.  A list of  }
{  "synonyms" (i.e., a set of words any one of which would be correct at a  }
{  given position in a sentence) must be surrounded by square brackets, [ ].  }


{  LIMITATIONS:  }

{  Version 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  }


{  XFCN INPUT PARAMETERS: }

{  Parameter number and description are show in left-hand column.. }
{  Permissible values are indicated in the right-hand column.  Value preceeded }
{  by asterisk is the default value assigned if the parameter is left empty. }

{  1=   model                  <string of 255 chars, max>                  }
{  2 =  response                 <stringof 255 chars , max>                  }
{  3=   cap_flag                 *"exact_case" | "authors_caps" | "ignore_case"  }
{  4=   extraWordsOK           True | *False                               }
{  5=   anyOrderOK             True | *False                               }
{  6=   misspellOK               True | *False                              }
{  7=   wordMarkUpNeeded         *True | False                               }
{  8=   runTogether_needed       *True | False                               }
{  9=   adjust_needed             *True | False                              }
{  10=  shortCut               *True | False                               }
{  11=  markupMapsNeeded       True | *False                               }
{  12=  parameterDisplayNeeded  *<empty> | "x" | "n" | "b" | "d" | "c" | "h" | "p" | "v" | "f " | "t"  }
{  13=  rawTraceNeeded         *<empty> | "x" | "r" | "p"                    }
{  14=  debugNeeded             True | *False                               }

{  These HyperCard globel vars may be used to input additional information:  }

{  theMarkUpPunctuation  }
{  theMarkUpSymbols     }
{  theMerkUpWeights      }
{  theMarkUpPhonMatrix  }
{  theMarkUpCherInfo     }


{  XFCN RETURN VALUES: }

{  (Direct fcn return value):  markup string                                  }

{  Returns in HyperCard globals:                                              }
{  theMarkUpReturnValues :  OK/NO boolean, plus other judging flags                    }
{  theMarkupMaps:            :  markup maps for R-TO-M on line 1, and  M-TO-R  on line 2  (if requested)    }
{  theMarkupParamDisplay:  display of the requested judging table values and parameters   (if requested )  }
```

```
{  theMarkupDebug            :  display of requestad debugging information  }

{  THINK PASCAL Version 4.0.2 project requires the following files , in this order :  }

{  DRVRRuntime.lib          }
{  Interface.lib              }
{  HyperXCmd.p              }
{  HyperXLib.lib              }
{  MarkUpXFCN3.p  (this file)    }


{  REVISON HISTORY FOR 3.0:  }

{  5 April 94 - RSH }
{  Fixed problem in WORD_MARKUP which kept spelling marks from being displayed when NOORDEROK was in effect.  }

{  11 April 94 - RSH  }
{  Rewrote MARK_SENTENCE, which would destroy left substring of markup when a word with final missing letter(s)  }
{  was followed by a missing word carat.  Also rewrote DUP_CHAR for greater efficiency.  }

{  26 May 94 - RSH }
{  Fixed DUP_CHAR so that it returns null string when char count is <= 0.  }
{  Fixed markup display so that a blank moveword won't shadow spelling markup on first char of word.  }
{  Edited SET_DIACRIT() initialization so that Swedish å,Å are assigned "supero" diacrit. }

{  27 May 94 - RSH }
{  Decoupled internal markup symbols from user-specified symbols to prevent confusions when user specifies }
{  weird or ambiguous symbols.  Internal symbol names begin with S  (e. g.  "Sextraletter"). Also created an  }
{  array SYMBOLMAP to map internal chars to user chars.  Transformation to external symbols done in CAPMARK }
{  and SPELLMARKS, and SENTENCEMARKUP.   Also changed internal display logic so that any user symbol set  }
{  equal to "nomark" (a blank space) will be "transparent" -- any symbols it normally shadows will appear properly.  }

{  16 Sept 94 - RSH }
{  Rewrote code which computes NED (normalized edit distance): }
{  Supposedly  0 ≤ NED ≤ 1, but in fact NED > 1 sometimes because normalizing term for RED was based on the "average" }
{  cost of an edit operation.  Now uses maxEditWeight, computed from wchange and the phon_matrix values at time phon_matrix is  }
{  initialized.  Normalization done on basis of max possible cost to convert a string of responses's length into one of model's
length.  }
{  Now 0 ≤ NED ≤ 1 is guaranteed. }

{  Rewrote code which converts NED to SNED (integer scaled normalized edit distance): }
{  Small NEDS became 0 when scaled to SNED integers for the sim[] array, because scale factor was too small, so significant  }
{ digits remained fractional and were lost in truncation.  Made SIM [ ] and other variables that hold scaled NEDs into LONGINT }
{  and increased scale factor to 10000 so that even very small fractions have an integer representation.   Both infinity and  }
{  editWeightScale are now constants and are used everywhere.  MaxCost has been eliminated. }

{  Fixed logic error in SPELLMARKS which failed to reset preemption flag and thus dropped all marks after the first  }
{  preemptive mark. }

{  Edited Phon Category assignments so that upper case vowels are counted as vowels as well as lower case ones.  }

{  1 November 94 - RSH  }
{  Moved large judging matrix MARKS to dynamic memory so that code would not take so much room on HyperCard  }
{  stack.  Replaced by new ptr MARKSP of type LSMATRIXP which is used to point to a new handle.  Handle is disposed }
{  before exit.  }

{  30 November 94 - RSH  }
{  Fixed error which caused the runtogether word analysis to overwrite previous matchings, imposing a new bogus }
{  match on words which already had  imperfect single-word match.  Introduced new sets MMAYBE and RMAYBE to keep }
{  track of M and R positions which have any kind of match, and used it to make sure that already matched M words }
{  are not grabbed by the runtogether analysis.  Now only M words which have no potential match at all can become  }
{  candidates as runtogether word match.  Test case is  }
{  Model:          Yesterd he  }
{  Respononse:  He yesterd  }
{  Markup:         -   ≤          [\  }
{  Where "yesterd" has been interpreted as a runtogether of "Yesterd he"  }
{  "he" is set as ignorable, which leads to a PMATCHED of 4/3  }

{  Also rationalized computation of PMATCHED, which is now #matched words/ total # words in M and in R; i. e., }
{  2*MATCHEDK / [(CARD( M ) - CARD( MIGNORE )) + (CARD(R) - CARD(RIGNORE))]  }

{  20 December 94 - RSH  }
{  Edited set_cap_info and set_phon_info in create_char_tables so that Courier upper-case accented vowels would be correctly  }
{  identified as upper-case and as vowels.  }

{------------------------------------------------------------------------------------------------------------------------}


unit markUpXFCN;

interface

 uses
  HyperXCmd;

 procedure main (paramPtr: XCmdPtr);   { FORWARD }


{------------------------------------------------------------------------------------------------------------------------}

implementation
```

```
procedure main (paramPtr: XCmdPtr);

 const
  versionStr = 'Markup XFCN 3.0  - 19 Dec 94, 8:20PM  - © Robert S. Hart -  UI/UC Language Learning Laboratory';
  lmax = 22;                { max number of letters in a word }
                                { WARNING:  word_max must be larger than or equal to wmax! }
  wmax = 18;                { max # positions in model and words in resp after processing }
  word_max = wmax + 0;      { max # of words in model at input }
  infinity = 9999;
  space = ' ';
  spaces = '                                                              ';
  editWeightScale = 10000;

  {  Internal markup symbols  }

  Saddcap = '+';                { plussign }
  Sdropcap = '-';           {  downarrow  }
  Saccenterr = '~';         {  tilde  }
  Sextrawd = 'X';           {  captial X }
  Smissingwd = 'Δ';         {  capital delta  }
  Smovewd = '«';        {  solid leftward arrowhead  }
  Sextraltr = 'x';          {  small x  }
  Smissingltr = '\';        {  backslash  }
  Ssubstituteltr = '=';     {  equal sign  }
  Stransltr1 = '>';         {  right angle bracket  }
  Stransltr2 = '<';         {  left angle bracket  }
  Srunonwd = '[';           {  left square bracket  }
  Snomark = '_';                {  underscore  }

  letterErrors = [Snomark, Saccenterr, 'u', 'd', 'U', 'D'];  {  Raw markup chars used to indicate case/accent errors or no error on
letter }

 type
  inputwrange = 1..word_max;
  wrange = 1..wmax;
  lrange = -1 .lmax;
  wordstr = string[lmax];
  str80 = string[80];
  wivector = array[wrange] of INTEGER;
  inputwivector = array[inputwrange] of INTEGER;
  wsvector = array[wrange] of wordstr;
  inputwsvector = array[inputwrange] of wordstr;
  diacrit_variants = (no_accant, acute, grave, circumflex, diarsis, umlaut, supero, cedilla, tilde, subdot, superdot, subhat,
superhat, subhook, macron);
  phon_variants = (vowel, consonant, phon3, phon4, phon5);
  cap_flag_type = (exact_case, authors_caps, ignore_case);
  case_variants = (up_case, down_case);

  wordset = set of wrange;
  charrange = 0..255;
  wimatrix = array[wrange, wrange] of INTEGER;
  wlmatrix = array[wrange, wrange] of LONGINT;
  limatrix = array[lrange, lrange] of INTEGER;
  lsmatrix = array[lrange, lrange] of wordstr;
  lsmatrixPtr = ^lsmatrix;
  pmatrix = array[phon_variants, phon_variants] of INTEGER;
  civector = array[CHAR] of INTEGER;
  ccvector = array[CHAR] of CHAR;
  choicelisttype = array[wrange] of wordset;
  solutionrec = record
    seq: str80;
    inversionk: INTEGER;
    firstinv: INTEGER;
   end;
  iset = set of 0..255;

 var
  p: Ptr;
  h, lsmH: handle;
  mw,                                       { vector of model words }
  rw,                                        { vector of response words }
  wordmark: inputwsvector;      { vector of word markups }
  rwxloc,                                   { vector of response word locations }
  m_to_r,                                   { response wd matched to given model wd }
  r_to_m: inputwivector;         { model wd matched to response wd }
  runtogether: wivector;            { index of 2nd run-together model wd }
  pnoninversions,              { proportion non-inverted words }
  pmatched,                                 { proportion words matched }
  cutoff,                                    { spell check applied if length ratio}
                                {    of 2 words falls below this value }
  prop_errors,                 { if edit dist between M and R word}
                                {    exceeds this, then round to infinity }
  runon_criterion: REAL;          { spelling match necessary to consider}
                                {    response word as run on }
  avedist: EXTENDED;           { averege edit distance between matched wds}
                                {    averaged over all matched pairs }
  winsert, wdelete, wchange,    { weights of various spelling errors }
  wtranspose, waccent, wcap, maxEditWeight: INTEGER;
  model, response: string;          { correct ans and student response }
  cap_flag: cap_flag_type;         { tells how to handle wrong cap letters }
```

```
    judgedok,                           { returns Ok or No for response }
    misspellok,                         { judge resp with misspellings Ok }
    extrawordsok,                 { judge resp w extra words Ok }
    anyorderok,                         { judge resp w words out of order Ok }
    runtogether_needed,           { enable/disable runtogether analysis }
    word_markup_needed,          { whether to generate sentence markup }
    adjust_needed,                { whether to adjust for optimal solution }
    markupMapsNeeded,             { whether to return markup maps lists to HyperCard }
    shortcut,                     { whether to shortcut when computing edit distance of very dissimilar words   }
    trace: BOOLEAN;                 { enable/disable tracing output }
    paramDisplayNeeded,          { which data structure info to return to HyperCard }
    rawTrace: CHAR;              { whether edit trace string returned should be "raw" or prettied up for display }

    nomark, addcap, dropcap,       { markup symbols }
    accenterr, ext_wd, missingwd, movewd, extraltr, missingltr, substituteltr, transltr1, transltr2, runonwd: CHAR;
    delim_chars: set of CHAR;     { punctuation symbols }

    mwlg, rwlg: inputwivector;       { lengths of model, resp words }
    mwseq: inputwivector;          { map of model word index to model position index }
    editd: limatrix;                { matrix of normalized edit distances between word substrings }
    marksP: lmatrixPtr;
    a: wlmatrix;
    aseq: wimatrix;
    rignore, mignore, rmatched, mmatched, rmaybe, mmaybe: wordset;
    dr: ciVector;
    symbolMap: ccVector;
    choices: choicelisttype;
    solutionlist: array[wrange] of solutionrec;
    rlg, mlg, plg, mwk, solutionk, edit_dist, matchedk, rightmost, recursionk, solutions_tried: INTEGER;
    mincost: LONGINT;
    runtogetherflag: BOOLEAN;
    time: REAL;

{ Judging tables used to control capitalization/diacritic judging }

    base_char: array[CHAR] of CHAR;
    diacrit_info: array[CHAR] of diacrit_variants;
    case_info: array[CHAR] of case_variants;
    phon_info: array[CHAR] of phon_variants;
    phon_matrix: pmatrix;


{-------------------------------------------------------------------------------}

{     UTILITY PROCEDURES     }

{-------------------------------------------------------------------------------}


{ ------------------------------------------------------------------------------FAIL }

{ Return ERRMSG as the XFCN's return value, and immediately exit the XFCN.   }

  procedure FAIL (errMsg: Str255);

  begin
   if marksP <> nil then
    disposPtr(PTR(marksP));
   paramPtr^.returnValue := PasToZero(paramPtr, errMsg);
   EXIT(Main);  { exit XFCN }
  end;   { FAIL }

{ ------------------------------------------------------------------------ReturnInGlobal }

{ Convenience proc for returning string value VALUE in a HyperCard global var  GLOBALNAME. }

  procedure returnInGlobal (globalName, value: str255);

   var
    h: handle;

  begin
   h := pasToZero(paramPtr, value);
   if h = nil then
    FAIL(concat('%Out of memory for return in global ', globalName))
   else
    begin
     setGlobal(paramPtr, globalName, h);
     disposHandle(h)
    end
  end;   { returnInGlobal }

{ ------------------------------------------------------------------AppendStringToHandle  }

{  Append string S at the end of the information pointed to by HANDLE.  }

  procedure appendStringToHandle (h: HANDLE; s: string);

   var
    r: OSErr;
    errType: string[20];
```

```
begin

 r := ptrAndHand(Ptr(ORD(@s) + 1), h, Length(s));

 if r <> 0 then
  begin
   case r of
    memFullErr:
     errType := 'Memory Full';
    nilHandleErr:
     errType := 'NIL handle';
    memWZErr:
     errType := 'Mem block is free'
   end;
   FAIL(Concat('%AppendStringToHandle error:  ', errType))
  end;

end;   { appendStringToHandle }

{ ------------------------------------------------------------------AppendStringToGlobal }

procedure appendStringToGlobal (gName: str255; s: str255);

 var
  h: HANDLE;
  lg: INTEGER;
  hp: PTR;

begin
 h := getGlobal(paramPtr, gName);
 lg := GetHandleSize(h);
 hp := PTR(ORD(StripAddress(h^)) + lg - 1);   { Ptr to last byte of block. }
 if (lg > 0) & (hp^ = 0) then    { If handle is non-nil and has null char terminator, }
  setHandleSize(h, lg - 1);      { remove null char terminator. }
 appendStringToHandle(h, concat(s, CHR(0)));   { Append string plus null char terminator. }
 setGlobal(paramPtr, gName, h);
 disposeHandle(h);
end;

{ ------------------------------------------------------------------------------------NtoS }

function NtoS (num: INTEGER): str255;

begin
 numToStr(paramPtr, num, NtoS);
end;   { NtoS }

{ ------------------------------------------------------------------------------------LtoS }

function LtoS (lng: LONGINT): str255;

begin
 longToStr(paramPtr, lng, LtoS);
end;   { LtoS }

{ ------------------------------------------------------------------------------------EtoS }

function EtoS (r: REAL): str255;

begin
 extToStr(paramPtr, r, EtoS);
end;   { EtoS }

{ ------------------------------------------------------------------------------------BtoS }

function BtoS (b: BOOLEAN): str255;

begin
 if b then
  BtoS := 'True'
 else
  BtoS := 'False'
end;   { BtoS }

{ ------------------------------------------------------------------------setToString }

function setToString (st: iset): str255;

 var
  i: INTEGER;
  s: str255;

begin
 s := '';
 for i := 1 to 30 do
  if i in st then
   s := concat(s, '1,')
  else
   s := concat(s, '0,');
 setToString := s;
```

```
  end;   { setToString }

{ ------------------------------------------------------------------------------Eq }

{ Convenience function to return case-insensitive equality of two strings }

  function eq (s1, s2: str255): BOOLEAN;

  begin
   eq := stringEqual(paramPtr, s1, s2)
  end;   { eq }

{ ---------------------------------------------------------------------------NthChunk }

{  Return the Nth chunk of string S, where a chunk is a substring lying between  }
{  two DELIM characters (beginning & end of S are implicit delimiters).   }

  function nthChunk (s: str255; n: INTEGER; dchar: CHAR): str255;

   var
    i, p: INTEGER;
    delim: string;

  begin

   delim := dchar;
   for i := 1 to n - 1 do              { remove first n - 1 chunks from string }
    begin
     p := pos(delim, s);
     if p > 0 then
      delete(s, 1, p)
     else                                 { if less than n-1 chunks, return EMPTY }
      begin
       nthChunk := '';
       EXIT(nthChunk)
      end
    end;

   p := pos(delim, s);                    { Nth chunk is now at front of list }
   if p > 0 then
    nthChunk := copy(s, 1, p - 1)
   else
    nthChunk := s;

  end;   { nthChunk }

{ ------------------------------------------------------------------------------Inc  }

  procedure inc (var x: integer);

  begin
   x := x + 1;
  end;   { inc }

{------------------------------------------------------------------------------Dec  }

  procedure dec (var x: integer);

  begin
   x := x - 1;
  end;   { dec }

{------------------------------------------------------------------------------Max  }

  function max (x, y: INTEGER): INTEGER;

  begin
   if x > y then
    max := x
   else
    max := y
  end;   { max }

{------------------------------------------------------------------------------Max  }

  function min (x, y: INTEGER): INTEGER;

  begin
   if x < y then
    min := x
   else
    min := y
  end;   { min }

{---------------------------------------------------------------------------Dup_Char  }

  function dup_char (c: CHAR; lg: INTEGER): string;

   var
    i: INTEGER;
    s: string;
```

```
 begin
  if lg <= 0 then
   begin
    dup_char := '';
    EXIT(dup_char);
   end;
  if (lg > 255) then
   lg := 255;
  s := c;
  for i := 1 to 8 do
   if length(s) < lg then
    if length(s) >= 128 then
     s := Concat(s, Copy(s, 1, lg - length(s)))
    else
     s := Copy(Concat(s, s), 1, lg)
   else
    begin
     dup_char := s;
     exit(dup_char)
    end;
 end;  { dup_char  }

{-------------------------------------------------------------------------------Card  }

{Compute the cardinality of a set of type 'wordset'.}

  function card (setofwords: wordset): INTEGER;

   var
    i, k: INTEGER;

  begin
   k := 0;
   for i := 1 to wmax do
    if (i in setofwords) then
     Inc(k);
   card := k;
  end;  { card }

{------------------------------------------------------------------------------- }

{   DEBUG I/O   }

{------------------------------------------------------------------------------- }

{-------------------------------------------------------------------------ShowSet  }

  procedure showset (s: string; st: iset);

  begin
   appendStringToGlobal('theMarkUpDebug', concat(s, ' =  ', setToString(st), CHR(13)))
  end;

{-----------------------------------------------------------------See_nedit_matrix }

  procedure see_nedit_matrix (s: str255);

   var
    m, r: INTEGER;

  begin
   appendStringToGlobal('theMarkUpDebug', concat(CHR(13), 'A[R,M] :  ', s, CHR(13)));
   for r := 1 to rlg do
    begin
     s := '';
     for m := 1 to mwk do
      s := concat(s, LtoS(a[r, m]), ' ');
     appendStringToGlobal('theMarkUpDebug', concat('R=', NtoS(r), '  ', s, CHR(13)));
    end
  end;   {  see_nedit_matrix  }

{-------------------------------------------------------------------DISABLED I/O  }

{   Following procedures disabled because MAC code resources cannot have standard I/O    }

  procedure pause;

  begin
    {readln;}
  end;

  procedure clrScr;

  begin
  end;


{-------------------------------------------------------------------------------}

{     INITIALIZE ALL STATIC DATA STRUCTURES -- JUDGING TABLES AND PARAMETERS.  }
```

```
{-------------------------------------------------------------------------------}


{-------------------------------------------------------------------Get_char_case  }

  function get_char_case (i: INTEGER): case_variants;

  begin
   if Char(i) in ['A'..'Z'] then
    get_char_case := up_case
   else
    get_char_case := down_case
  end;  { get_char_case  }

{-------------------------------------------------------------------Force_down_case }

  function force_down_case (i: INTEGER): CHAR;

  begin
   if get_char_case(i) = up_case then
    force_down_case := chr((ORD('a') - ORD('A')) + i)
   else
    force_down_case := chr(i);
  end;  {  force_down_case  }

{-------------------------------------------------------------------Set_base_info }

  procedure set_base_char (c: CHAR; s: str80);

   var
    i: INTEGER;

  begin
   for i := 1 to Length(s) do
    if s[i] <> space then
     base_char[s[i]] := c;
  end;  { set_base_char }

{-------------------------------------------------------------------Set_diacrit_info  }

  procedure set_diacrit_info (d: diacrit_variants; s: str80);

   var
    i: INTEGER;

  begin
   for i := 1 to Length(s) do
    if s[i] <> space then
     diacrit_info[s[i]] := d
  end;  {  set_diacrit_ info  }

{-------------------------------------------------------------------Set_cap_info }

  procedure set_cap_info (c: case_variants; s: str80);

   var
    i: INTEGER;

  begin
   for i := 1 to Length(s) do
    if s[i] <> space then
     case_info[s[i]] := c
  end;   { set_cap_info  }

{-------------------------------------------------------------------Set_phon_info  }

  procedure set_phon_info (p: phon_variants; s: str80);

   var
    i: INTEGER;

  begin
   for i := 1 to Length(s) do
    if s[i] <> space then
     phon_info[s[i]] := p
  end;  {  set_phon_ info  }

{-------------------------------------------------------------Create_char_info_tables  }

{   Initialize all character information tables.  These tables provide }
{   descriptive information about each of the 255 characters in the Mac }
{   character set used for the model and response.}

{   Global data structures affected:}
{   base_char           : vector specifying the base (unaccented) char corresponding}
{                               to each char.}
{   diacrit_info        : vector specifying the type of diacritic mark which}
{                               modifies each char}
{   case_info           : vector specifying the case (upper or lower) of each char.}
{   phon_info           : vector specifying whether each char is vowel or consonant.}
```

```
procedure create_char_info_tables;

 var
  i, lineNo: INTEGER;
  c: CHAR;
  s: str80;
  str: Str255;
  h: handle;

begin  { create_char_info_tables  }
 for i := 1 to 255 do
  begin
   c := CHAR(i);
   base_char[c] := CHAR(force_down_case(i));
   case_info[c] := get_char_case(i);
   diacrit_info[c] := no_accent;
   phon_info[c] := consonant
  end; { DO }
{  Replace base_char default value for accented chars.  Chars with accents have   }
{  the unaccented version as base cher.  Uneccented chars have themselves as base  }
{  char (this is the default case).    }

{  NOTE:  These settings assume that the font is COURIER or some compatible font!!  }
{  They may not display properly here in a font other than courier.  }

  set_base_char('a', 'á À ã Ã à À â Â ä Ä Å');
  set_base_char('e', 'é É è È ê Ê ë Ë');
  set_base_char('i', 'í Í ì Ì î Î ï Ï');
  set_base_char('o', 'ó Ó ò Ò ô Ô ö Ö õ Õ');
  set_base_char('u', 'ú Ú ù Ù û Û ü Ü');
  set_base_char('y', 'ÿ Ÿ');
  set_base_char('c', 'ç Ç');
  set_base_char('n', 'ñ Ñ');
{  Enter proper diacritic information for accented chars.  Unaccented chars have   }
{  "no_accent" as their diacritic.  Accented chars are assigned the proper accent   }
{  mark.  These settings assume COURIER font, and may not display properly in  }
{  another font.  }
  set_diacrit_info(acute, 'á Á é É í Í ó Ó ú Ú');
  set_diacrit_info(grave, 'à À è È ì Ì ò Ò ù Ù');
  set_diacrit_info(circumflex, 'â Â ê Ê î Î ô Ô û Û');
  set_diacrit_info(diarsis, 'ä Ä ë Ë ï Ï ö Ö ü Ü ÿ Ÿ');
  set_diacrit_info(supero, 'å Å');
  set_diacrit_info(cedilla, 'ç Ç');
  set_diacrit_info(tilde, 'ã Ñ ñ Ã õ Õ');
  set_diacrit_info(macron, '');  { IBM PC had some macron chars }

{  Enter case info for upper case accented letters.  This supplements the default   }
{  assignment of "upper_case" to A..Z. - COURIER font.  }
  set_cap_info(up_case, 'Á É Í Ó Ú Ä Ë Ï Ö Ü Ÿ À È Ì Ò Ù Â Ê Î Ô Û Å Æ Œ Ø Ñ Ç');

{  Set phon info for vowels.  This overrides the default setting of "consonant".  }
{  Specify both upper and lower case, separately for chars with diacrits.  - COURIER font.  }
  set_phon_info(vowel, 'a e i o u y A E I O U Y å æ œ ø');
  set_phon_info(vowel, 'á é í ó ú ä ë ï ö ü ÿ á è ì ò ù å ö ä ë ï ö ü');
  set_phon_info(vowel, 'Á É Í Ó Ú Ä Ë Ï Ö Ü Ÿ À È Ì Ò Ù Â Ö Ä Ê Î Ô Û Å Æ Œ Ø');

{ Set cap info for accented chars  - COURIER font  }
 end; { create_char_info_tables }

{-------------------------------------------------------------------OverrideCharInfoTables  }

{ Take a line of char info specs and install them in the proper char info table.  }

 procedure overrideCharInfoTables (specs: Str255);

  var
   i, n: INTEGER;
   switch: char;
   d: diacrit_variants;
   c: case_variants;
   p, q: phon_variants;
   s, ch: Str255;

  begin
   switch := specs[1];  { Specifies type of info.  }
   delete(specs, 1, 2); {leading char and following comma}
   i := pos(',', specs);
   if i = 0 then
    FAIL(Concat('%Missing comma after switch in judging table line:  ', specs));
   ch := Copy(specs, 1, i - 1);   { Specifies base char or variant value for following list.  }
   if ch = '' then
    FAIL('%Missing base char or variant specifier.');
   delete(specs, 1, i);
   case switch of
    'b':   { base char }
     set_base_char(ch, s);
    'd':    {  diacritic information  }
     begin
      if eq(ch, 'acute') then
       d := acute
```

```
      else if eq(ch, 'grave') then
       d := grave
      else if eq(ch, 'circumflex') then
       d := circumflex
      else if eq(ch, 'diarsis') then
       d := diarsis
      else if eq(ch, 'supero') then
       d := superc
      else if eq(ch, 'cedille') then
       d := cedilla
      else if eq(ch, 'tilide') then
       d := tilde
      else if eq(ch, 'macron') then
       d := macron
      else
       FAIL(Concat('%Bad diacritic variant value: ', ch));
      set_diacrit_info(d, s);
     end;
    'c':   { capitalization information  }
     begin
      if eq(ch, 'up_case') then
       c := up_case
      else if eq(ch, 'down_case') then
       c := down_case
      else
       FAIL(Concat('%Bad cap variant value: ', ch, '   ', specs));
      set_cap_info(c, s);
     end;
    'p':   { phon information }
     begin
      if eq(ch, 'vowel') then
       p := vowel
      else if eq(ch, 'consonant') then
       p := consonant
      else if eq(ch, 'phon3') then
       p := phon3
      else if eq(ch, 'phon4') then
       p := phon4
      else if eq(ch, 'phon5') then
       p := phon5
      else
       FAIL(Concat('%Bad phon variant value: ', ch));
      set_phon_info(p, s);
     end;
    otherwise
     FAIL(concat('%Bad judging table switch:  ', switch));
   end;   {  CASE switch OF  }

  end;   {  overrideCharInfoTables  }

{-------------------------------------------------------------------------Init_markup  }

{  Set values for program paramaters and data structures.  }
{  First look to see if values have been provided in these 5 global variables:   }
{       theMarkupPunctuation    }
{       theMarkupSymbols      }
{       theMarkupWeights      }
{       theMarkUpCharInfo      }
{       theMarkUpPhonMatrix    }
{  If so, process those values to set the data; otherwise use default values.  }

  procedure init_markup;

{---------------------SetDelimiters  }

{   Specify characters which will serve as punctuation in modal and response.   }
{   If there are data in the global variable 'theMarkupPunctuation', use them.   }
{   Otherwise, set values below as default values.   }
{   Chr(13) is MAC/HyperCard RETURN char, which starts new line.   }

   procedure setDelimiters;

    var
     h: handle;
     s: str255;
     i: INTEGER;

   begin

    h := getGlobal(paramPtr, 'theMarkupPunctuation');
    zeroToPas(paramPtr, h^, s);
    disposHandle(h);

    if s = '' then
     delim_chars := [' ', '.', ',', ';', ':', '(', ')', '[', ']', '<', '>', '?', '!', Chr(13)]
    else
     begin
      delim_chars := [];
      for i := 1 to Length(s) do
       delim_chars := delim_chars + [s[i]];
     end
```

```
  end;  { setDelimiters }

{-----------------------------------SetSymbols  }

{   Specify characters which will serve as punctuation in model and response.   }
{  If there are data in the global var 'theMarkupSymbols', use them.   }
{   Otherwise, set default values below.   }

  procedure setSymbols;

   var
    h: handle;
    i: INTEGER;
    Ss, s: str255;

  begin

   h := getGlobal(paramPtr, 'theMarkupSymbols');
   zeroToPas(paramPtr, h^, s);
   disposHandle(h);

   if s = '' then
    begin
     addcap := '+';          { uparrow  }
     dropcap := '-';         { downarrow  }
     accenterr := '~';       { tilda  }
     extrawd := 'X';         { capital X }
     missingwd := 'Δ';       { capital delta  }
     movewd := '«';         { solid leftward arrowhead  }
     extraltr := 'x';        { small x  }
     missingltr := '\';      { backslash   }
     substituteltr := '=';    { equal sign   }
     trsnsltr1 := '>';       { right angle bracket   }
     transltr2 := '<';       { left angle bracket  }
     runonwd := '[';         { left square bracket  }
    end
   else
    begin
     addcap := s[1];
     dropcap := s[2];
     accenterr := s[3];
     extrawd := s[4];
     missingwd := s[5];
     movewd := s[6];
     extraltr := s[7];
     missingltr := s[8];
     substituteltr := s[9];
     transltr1 := s[10];
     trsnsltr2 := s[11];
     runonwd := s[12];
    end;

   s := concat(addcap, dropcap, accenterr, extrawd, missingwd, movewd, extraltr, missingltr, substituteltr, transltr1, transltr2,
runonwd);
   Ss := concat(Saddcap, Sdropcap, Saccenterr, Sextrawd, Smissingwd, Smovewd, Sextraltr, Smissingltr, Ssubstituteltr, Stransltr1,
Stransltr2, Srunonwd);
   for i := 1 to Length(Ss) do
    symbolMap[Ss[i]] := s[i];
   nomark := space;
   symbolMap[Snomark] := nomark;

  end;  { setSymbols }

{-------------------------Set_judging_tables  }

  procedura set_judging_tables;

   var
    h: handle;
    p: ptr;
    s: str255;

  begin

{  First, initialize all judging tables with default values.  }

   create_char_info_tables;

{  Override default values with user-specified values in THEMARKUPCHARINFO.  }

   h := getGlobal(paramPtr, 'theMarkupCharInfo');
   p := h^;
   disposHandle(h);

   while True do
    begin
     while p^ = 13 do                { If at CR, move to next char }
      p := PTR(ORD(p) + 1);          { If line is empty, skip over it.}
     if p^ = 0 then                   { If at end of string, exit. }
      LEAVE;
```

```
      returnToPas(paramPtr, p, s);        { If real line, get it  }
      overrideCharInfoTables(s);          { and install its values.  }
      scanToReturn(paramPtr, p);          { Move to the next CR }
     end;

   end;   { set_judging_tables  }

{--------------------------Set_phon_matrix  }

{   Specify characters which will serve as punctuation in model and response.   }
{  If there are data in the global var 'theMarkupSymbols', use them.   }
{   Otherwise, set default values below.   }

   procedure set_phon_matrix;

    var
     h: handle;
     s: str255;
     p, q: phon_variants;
     i, w: INTEGER;

   begin

    h := getGlobal(paramPtr, 'theMarkUpPhonMatrix');
    zeroToPas(paramPtr, h^, s);
    disposHandle(h);

    if s = '' then
{  Put default values into the substitution weigh matrix, pmatrix.  }
{  For a given cell PHON_MATRIX[M, R], M is the phonetic category of a model }
{  cherecter MC and R is the phonetic category of a response character RC.  }
{  The integer value in the cell is the weight attached to substituting RC for MC. }
{  The default values below equal to WCHANGE if MC and RC are in the same category; }
{  if they are in different categories, the cost of a sustitution is 1.2 times WCHANGE.  }
     begin
      for p := vowel to phon5 do
       for q := vowel to phon5 do
       if p = q then
       phon_matrix[p, q] := wchange
       else
       phon_matrix[p, q] := TRUNC(1.2 * wchange);
      maxEditWeight := TRUNC(1.2 * wchange);
     end
    else
{ If the HyperCard globsl THEMARKUPPHONMATRIX is not empty, read values from it.  }
     begin
      maxEditWeight := wchenge;
      i := 1;
      for p := vowel to phon5 do
       for q := vowel to phon5 do
       begin
       w := strToNum(paramPtr, nthChunk(s, i, ','));
       phon_matrix[p, q] := w;
       maxEditWeight := max(maxEditWeight, w);
       inc(i);
       end
     end;

   end;  { set_phon_matrix }

{--------------------------SetWeights  }

{   Specify weights and thresholds which control spelling analysis.   }
{   Values must be contained in the global var THEMARKUPWEIGHTS, and   }
{   must appear as comma-separated items, in this order:   }
{   winsert, wdelete, wchange, wtranspose, cutoff, prop_errors, runon_criterion   }
{   If any of these items is EMPTY, a default value will be used.  If  THEMARKUPWEIGHTS  }
{   does not exist or is empty, all default values will be used . }

   procedure setWeights;

    var
     h: handle;
     v, s: str255;

   begin

{   These are the default weights assigned to the various edit operations, chosen so   }
{   that the cost of a change is less that that of a deletion followed by an insertion. }
{   Also, the cost of a change, or of a deletion/insertion sequence is greater than  }
{   that of a transposition.  The "standard" distances of 2,2,3,2 have been multiplied   }
{   by 10 so that accent and cap errors can be scored at a lower value. }
    winsert := 20;
    wdelete := 20;
    wchange := 30;
    wtranspose := 20;
    waccent := 1;
    wcap := 1;

 {    Ratio of word lengths must be nearer than this to 1 or the edit distance between   }
 {   them will be automatically set to infinity (used only when 'shortcut' is TRUE).   }
```

```
  cutoff := 0.67;

{   This peremeter controls the proportion of spelling edits which can occur when   }
{   attempting to match two words before the two words will be considered non-matches.   }
  prop_errors := 0.35;

{   This is the max normelized edit distance which can exist between 2 model words m + mm   }
{   and a response word r before r can be considered to be m and mm run together.   }
  runon_criterion := 0.2;

{   Override the defaults with values in global variable THEMARKUPWEIGHTS.   }
  h := getGlobal(paramPtr, 'theMarkupWeights');
  zeroToPas(paramPtr, h^, s);
  disposHandle(h);

  if s <> '' then
   begin

    v := nthChunk(s, 1, ',');
    if v <> '' then
     winsert := strToNum(paramPtr, v);

    v := nthChunk(s, 2, ',');
    if v <> '' then
     wdelete := strToNum(paramPtr, v);

    v := nthChunk(s, 3, ',');
    if v <> '' then
     wchange := strToNum(paramPtr, v);

    v := nthChunk(s, 4, ',');
    if v <> '' then
     wtranspose := strToNum(paramPtr, v);

    v := nthChunk(s, 5, ',');
    if v <> '' then
     wcap := strToNum(paramPtr, v);

    v := nthChunk(s, 6, ',');
    if v <> '' then
     weccent := strToNum(paramPtr, v);

    v := nthChunk(s, 7, ',');
    if v <> '' then
     cutoff := strToExt(paramPtr, v);

    v := nthChunk(s, 8, ',');
    if v <> '' then
     prop_errors := strToExt(paramPtr, v);

    v := nthChunk(s, 9, ',');
    if v <> '' then
     runon_criterion := strToExt(paramPtr, v);

   end;

 end;  { setWeights }

{-------------------------------------------}

 begin  {  init_markup  }

{   Specify cap, eccent, vowel, and base-char properties of chars.   }
{   First set defaults, then look for values in global vars.   }
  set_judging_tables;

{   Specify which input chars will serves as 'punctuation' (word delimiters).   }
  setDelimiters;

{   Specify which symbols to use for markup display.   }
  setSymbols;

{   Set numerical weights and thresholds which control judging process.   }
  setWeights;

{  Set values in PHON_MATRIX, which determines substitution cost for various  }
{  combinations of character categories.   Also computes maxEditWeight. }
  set_phon_matrix;

 end;  { init_markup }

{-------------------------------------------------------------------------------------------}

{     INPUT PARSING    }

{-------------------------------------------------------------------------------------------}

{-------------------------------------------------------------Segment_string  }
```

```
{    Process the model (correct answer) string and the (student's) response strings    }
{   and puts them into an internal format suitable for further processing.    }
{   String is segmented into words and the total number of words, as well as the    }
{   length of each word, is recorded.  While breaking out individual words, all    }
{   extraneous characters -- extra spaces and punctuation -- are discarded.    }

{   If string is a model, the special syntax of ingorable words and synonyms is    }
{   interpreted, and a list (in set format) of ignorable word positions is built,    }
{   as well as information on which words are synonyms and which sequential    }
{   position each words occupies in the sentence.  All the synonyms in a group    }
{   share the same sequential position.    }

{   Special syntax for correct answer:  ignorable words are placed within angle    }
{   brackets, and synonymous words within square brackets, eg:    }

{    The quick < brown > fox [ jumped leaped ] over the < lazy > dog    }

{   Input vars:    }
{   c           : string to be processed (correct answer or response)    }
{   ismodel      : True if string is a correct answer; false if it is student response.    }

{   Return vars:    }
{   wk         : number of words    }
{   w           : vector of words (strings)    }
{   wlg         : parallel vector of word lengths (char counts)    }
{   waux       : If string is model,    }
{              position number of each word (all the synonyms in a list share the    }
{              same postion number), or,    }
{            if string is response,    }
{              column location of leftmost letter of each word (entire response    }
{              is assumed to be on a single screen line. )    }
{   mignore     : list (in set format) of word sequence numbers to ignore    }

  procedure segment_string (var wk: INTEGER; var c: string; var w: inputwsvector; var wlg, waux: inputwivector; ismodel: BOOLEAN);

   var
    i, p, position, lg: INTEGER;
    x: wordstr;
    syn_list, ignore_flag: BOOLEAN;

  begin

   c := concat(c, ' ');
   lg := Length(c);
   wk := 0;
   position := 1;
   syn_list := False;        {    Turn on when processing synonym list.    }
   ignore_flag := False;  {    Turn on when processing ignorable word list.    }
   if ismodel then
    mignore := [];
{    Take succesaive chars from string to build next word.  If word has become too    }
{   long, set error flag and exit.    }
   i := 1;
   while i <= lg do
    begin
     x := '';
     p := i;
     while (i <= lg) and not (c[i] in delim_chars) do
      begin
       x := concat(x, c[i]);
       Inc(i)
      end;
{    If word not null, then update word count, word vector, word length vector.    }
     if x <> '' then
      begin
{    If sentence would heve more than the allowable number of words, or if the    }
{    current word is too long, set error flag and exit.    }
       if (wk + 1) > word_max then
       FAIL('%Too many words in input.');
       if Length(x) > lmax then
       FAIL(concat('%Input word too long:  ', x));
       Inc(wk);
       w[wk] := x;
       wlg[wk] := Length(x);
{    If string is model, also update synonym list, ignorable word list, and map    }
{    of model word numbers to model positions.    }
       if ismodel then
       begin
       if ignore_flag then
       mignore := mignore + [position];
       waux[wk] := position;
       end
       else
       waux[wk] := p;
       if not syn_list then
       Inc(position);
      end;
{    Process delimiters trailing at end of word, including ignorable and synonym list    }
{   delimiters.  If one of the latter is encountered, set or clear the appropriate    }
{   flags.    }
     while (i <= lg) and (c[i] in delim_chars) do
```

```
      begin
       if ismodel then
       case c[i] of
       '[':
       syn_list := True;
       ']':
       begin
       syn_list := False;
       Inc(position);
       end;
       '<':
       ignore_flag := True;
       '>':
       ignore_flag := False;
       end; {CASE}
       Inc(i);
      end; { WHILE ( i <= lg ) AND ( c[ i ] IN delim_chars ) }
    end;  { WHILE i <= lg ) }
{   Clean-up code for end-of-string condition.  If number of response words less 1,   }
{   or number of positions in model, exceeds the dimension of the (square) sim   }
{   matrix, the set error flag and exit. Otherwise store the number of positions in   }
{   the model, if string is model, or the column number of the first character   }
{   beyond the end of the response ( used later for markup).    }
   case ismodel of
    True:
     if position > wmax then
      FAIL('%Too many word positions in model.')
     else
      plg := position - 1;
    False:
     if (wk + 1) > wmax then
      FAIL('%Too many words in input.')
     else
      waux[wk + 1] := Length(c) + 1;
   end; {CASE}

  end;  { segment_string }

{-------------------------------------------------------------------------SegmentModel  }

  procedure segmentModel;

   var
    i: INTEGER;

  begin
   for i := 1 to word_max do
    mw[i] := '*';
   segment_string(mwk, model, mw, mwlg, mwseq, True);
  end;  { segmentModel }

{-------------------------------------------------------------------------SegmentResponse  }

  procedure segmentResponse;

   var
    i: INTEGER;

  begin
   for i := 1 to word_max do
    rw[i] := '*';
   segment_string(rlg, response, rw, rwlg, rwxloc, False);
  end;  { segmentResponse }

{-------------------------------------------------------------------------SetModel  }

  procedure setModel (s: string);

  begin
   model := s;
   segmentModel;
  end;  { setModel }

{-------------------------------------------------------------------------SetResponse  }

  procedure setResponse (s: string);

  begin
   response := s;
   segmentResponse;
  end;  { setResponse }


{-------------------------------------------------------------------------}

{     SPELLING ANALYSIS   }

{-------------------------------------------------------------------------}


{-------------------------------------------------------------------------Init_spelling  }
```

```
{   Initialize all matrix data structures used by the dynamic programming algorithm   }
{   which generates a "nearest match" misspelling markup.  These data structures   }
{   are all global.  The (global) vars affected are:   }
{   editd          : matrix of (minimal) edit distances   }
{   marks         : parallel matrix of (minimal) markup corresponding to each edit distance.   }

  procedure init_spelling (var marks: lsmatrix);

   var
    i: INTEGER;

  begin
   editd[0, 0] := 0;
   marks[0, 0] := '';
   for i := -1 to lmax do
    begin
     marks[i, -1] := '';
     marks[-1, i] := '';
     editd[i, -1] := infinity;
     editd[-1, i] := infinity;
    end;
   for i := 1 to lmax do
    begin
     editd[i, 0] := editd[i - 1, 0] + wdelete;
     editd[0, i] := editd[0, i - 1] + winsert;
     marks[i, 0] := concat(marks[i - 1, 0], extraltr);
     marks[0, i] := missingltr;
    end;
  end;  { init_spelling }

{-------------------------------------------------------------------CapMark  }

{    Return System  markup char for capitalization and/or accent error.  Character M,  }
{    assumed to be from the model, is compaired to character R, assumed to  }
{    be in the student's response.  If R has both a cap error and an accant  }
{    error, the cap error takes precedence.  }

  function capMark (m, r: CHAR): CHAR;

   var
    mcase, rcase: case_variants;
    mark: CHAR;

  begin
   mcase := case_info[m];
   rcase := case_info[r];
   mark := Snomark;      { Default is no mark }
   if (cap_flag <> ignore_case) and (mcase <> rcase) then
    case cap_flag of
     exact_case:
      if rcase = down_case then
       mark := Saddcap
      else
       mark := Sdropcap;
     authors_caps:
      if mcase = up_case then
       mark := Saddcap;
    end; {CASE}

{ If case is ignored or ok, or user's case mark is blank, then check to see if accents match.  }
   if (mark = Snomark) & (diacrit_info[m] <> diacrit_info[r]) than
    mark := Saccenterr;

   capMark := mark;

  end;  { capMark }

{-------------------------------------------------------------------AccentError  }

{   Compair the two chars M and R.  Assign score WCAP for a cap error, WACCENT for  }
{   an accent error, and return the total score.   C returns the type of error(s):  Snomark for none,   }
{   'u' and 'd' for case error only; '~' for accent error only; 'U' and 'D' for both case and accent error.  }
{   The datailed info is useful only for return to user when raw markup string is raquested;   }
{   CAPMARK regenerates it during word markup.   }

  function accentError (m, r: CHAR; var c: CHAR): INTEGER;

   var
    errK: INTEGER;

  begin

{   See if cases match.  }
   if (case_info[m] = case_info[r]) | (cap_flag = ignore_case) | ((cap_flag = authors_caps) & (case_info[r] = up_case)) then
    begin
     errK := 0;
     c := Snomark   {  No case err }
    end
   else
    begin
```

```
    errK := wcap;
    if case_info[m] = up_case then
     c := 'u'          { Case error }
    else
     c := 'd';         { Case error }
   end;

{  Also check to see if accents match.  }
  if diacrit_info[m] = diacrit_info[r] then
   accentError := errK
  else
   begin
    accentError := errK + waccent;
    if c = Snomark then
     c := Saccenterr   { Accent err only }
    else if c = 'u' then
     c := 'U'
    else
     c := 'D'    { Case and accent err }
   end;

 end;  { accentError }

{-------------------------------------------------------------------------SpellMarks  }

{   Converts the "raw" spelling markup returned by the least-distance algorithm to   }
{   a markup suitable for display:   (a) When two letters match, checks case/accent  }
{   and generates the case/accent markup, if any; (b) reduces a sequence of omission marks,   }
{   "\", to a single omission mark; (c) supresses any markup of letter following an   }
{   omission, since the omission marker "\" occupies the space beneath the next   }
{   character following the omission; (d) substitutes a blank space for "-" as an   }
{   indicator of properly matched letters.   }

 function spellMarks (var marks: wordstr; var m, r: wordstr): wordstr;

  const
   preemptives = [Smissingltr, Srunonwd];

  var
   i, j, k: INTEGER;
   mc, usermark: CHAR;
   preempted: BOOLEAN;
   markup: wordstr;

 begin

  i := 0;
  j := 0;
  preempted := False;
  markup := '';

  for k := 1 to Length(marks) do
   begin

    Inc(i);
    Inc(j);
    mc := marks[k];

    if mc in letterErrors then    { case or accent or no error }
     mc := capmark(m[i], r[j])  { system error char }
    else if (mc = Sextraltr) then
     Dec(i)
    else if (mc in preemptives) then
     Dec(j);

{ If the user symbol for missing wd or runtogether is space, do not preempt spelling mark.  }
    usermark := symbolMap[mc];
    if not preempted then
     if (mc in preemptives) then
      if (usermark = nomark) then
       preempted := False
      else
       begin
       preempted := True;
       markup := concat(markup, usermark);
       end
      else
       begin
       preempted := False;
       markup := concat(markup, usermark);
       end
    else
{ If preemption is on, skip current markup char, but turn off preemption to accept successive ones. }
     preempted := False;

   end; {FOR}

  spellmarks := markup

 end; { spellMarks }
```

```
{-------------------------------------------------------------------------------Nedit_dist }

{   Computes normalized minimal spelling edit distance between two strings R and M and   }
{   (optionally) the markup string which corresponds to that edit distance.   }
{   Input vars:   }
{   r               : Single word from response string   }
{   m               : Single word from model (correct answer)   }
{   markflag      :TRUE if markup corresponding to edit distance is to be returned;   }
{                   FALSE if no markup string needed.   }
{   shortcut      :TRUE if words of too different length will be given distance infinity; }
{                  FALSE if exact distance must be computed.  }
{   Return vars:   }
{   markup        : Markup string (if requested).   }
{   nedit_dist    : Normalized edit distance ( between 1 and 0 ).   }
{   edit_dist       : (GLOBAL var) : weighted, unnormalized edit distance.   }

  function nedit_dist (var r, m, markup: wordstr; marks: lsmatrix; markflag, shortcut: BOOLEAN): REAL;

   var
    i, j, flag, x, x2, x3, x4, d, ml, rl, db, il, jl: INTEGER;
    ratio: REAL;
    c, lastc, mc, rc, mk: CHAR;
    t: wordstr;

  begin

   runtogetherflag := False;

{  If doing standard order analysis, handle some special cases.  }
{  If raw trace was requested , go directly to produce minimal trace. }

   if rawTrace = 'x' then
    begin
{   If the two words match exactly, return edit distance of 0.   }
     if m = r then
      begin
       nedit_dist := 0;
       edit_dist := 0;
       markup := '';
       EXIT(nedit_dist);
      end;
     ml := Length(m);
     rl := Length(r);
{   If word lengths vary too much, and shortcut flag is set, skip further   }
{   analysis and return infinite edit distance.   }
     if shortcut then
      begin
       if ml < rl then
       ratio := ml / rl
       else
       ratio := rl / ml;
       if ratio < cutoff then
       begin
       nedit_dist := infinity;
       edit_dist := infinity;
       markup := '';
       EXIT(nedit_dist)
       end;
      end; { IF shortcut }
    end
   else     { rawTrace = 'r' or 'p'.  }
    begin
     markup := '';
     ml := Length(m);
     rl := Length(r);
    end;

{   Otherwise, compute the edit distance between the two words using dynamic  }
{   programming algorithm expressing recursive relation between left substring  }
{   distances.  This is a form of exhaustive search, implemented here by iteration  }
{  rather than true recursion . }

{   Initialize temporary memory array.   }
{  dr[ch] will store location in resp where char ch last appeared.  }
   for i := 1 to 255 do
    dr[chr(i)] := 0;

{   Main loops to fill matrix of substring edit distances.   }
   for i := 1 to rl do
    begin
     db := 0;

     for j := 1 to ml do
      begin

       mc := m[j];
       rc := r[i];
       il := dr[base_char[mc]];  {last occurence of mc in resp}
       jl := db;                          {last matched char in model}

{   Check for identity or substitution of end chars in each string.   }
```

```
      if mc = rc then
      begin
      d := 0;          {dist between chars mc and rc}
      db := j;         {model position of last matched char}
      mk := Smomark;
      end
      else if base_char[mc] = base_char[rc] then
      begin
      d := accentError(mc, rc, mk);  { return d and mk, is in _~udUd }
      db := j;
      end
      else
      begin
      d := phon_matrix[phon_info[mc], phon_info[rc]]; { subst dist for type mc and type rc}
      mk := Ssubstituteltr;
      end;

{   Find cost of matching via omission, insertion, substitution, and transposition.  }
      x := editd[i - 1, j - 1] + d;
      x2 := editd[i - 1, j] + winsert;
      x3 := editd[i, j - 1] + wdelete;
      x4 := editd[i1 - 1, j1 - 1] + (i - i1 - 1) * wdelete + wtranspose + (j - j1 - 1) * winsert;

{   Select the match which yields least cost.  Start by assuming omission (x1).    }
      flag := 1;
      if x2 < x then
      begin
      x := x2;
      flag := 2
      end;
      if x3 < x then
      begin
      x := x3;
      flag := 3
      end;
      if x4 < x then
      begin
      x := x4;
      flag := 4
      end;
      editd[i, j] := x;

{   If markup return is requested, generate markup string for this char pair.  }

{   When marking an omission, use special mark for omission of space, which indicates   }
{   run together words.    }
      if (flag = 3) and (m[j] = space) then
      runtogetherflag := True;

      if markflag then
      case flag of
      1:
      marks[i, j] := concat(marks[i - 1, j - 1], mk);
      2:
      marks[i, j] := concat(marks[i - 1, j], Sextraltr);
      3:
      if m[j] = space then
      marks[i, j] := concat(marks[i, j - 1], Srunonwd)
      else
      marks[i, j] := concat(marks[i, j - 1], Smissingltr);
      4:
      marks[i, j] := concat(marks[i1 - 1, j1 - 1], Stransltr1, dup_char('*', i - i1 - 1), Stransltr2);
      end;   {CASE flag OF}

     end; { FOR j  DO}

    dr[base_char[r[i]]] := i;

   end; { FOR i  DO}

{   Minimum weighted, unnormalized edit distance is now in lower right entry of editd matrix.  }
{  This number includes weights due to accent and case errors.  Save it in global var edit_dist.  }
   edit_dist := editd[rl, ml];

{   Get & return normalized edit distance, nedit_dist, by dividing maximum total cost of converting  }
{   a string of length rl to one of length ml.  }
   nedit_dist := edit_dist / (maxEditWeight * min(ml, rl) + wdelete * (max(ml, rl) - min(ml, rl)));

{  Return markup string.  Will be null if none was generated in loop above.  }

   if rawTrace = 'r' then
    markup := marks[rl, ml]                              {  Return raw edit markup.  }
   else
    markup := spellmarks(marks[rl, ml], m, r);   {  Return "pretty" markup suitable for display.  }

  end; { nedit_dist }

{-------------------------------------------------------------------------------Edit_trace}

{  Top-level control to generate a least-cost edit trace of the strings MODEL and RESPONSE.  }
{  Form of trace (raw or pretty) is controlled by global var RAWTRACE values 'r' or 'p'.  }
```

```
{  Edit trace string is put into direct HyperCard XFCN return.  Indirect return is put into  }
{  the HyperCard global var THEMARKUPRETURNVALUES:  two comma separated items,  }
{  the first the raw edit distence, and the second  the normelized edit distance.  }

  procedure edit_trece (model, response: str255);

   ver
    ned: REAL;
    c: CHAR;
    s1, s2: atr255;
    ms, rs, marks: wordstr;

  begin
   if (length(model) > lmax) | (length(response) > lmax) then
    FAIL('%Input string length exceeds max word length.')
   else
    begin
     c := ',';
     ms := copy(model, 1, 25);
     rs := copy(response, 1, 25);
     init_spelling(marksP^);

  {  Return markup string in MARKS and weighted unnormelized edit dist in EDIT_DIST.  }
     ned := nedit_dist(rs, ms, marks, marksP^, True, False);

  {  Convert so markup string is direct return; NEDIT_DIST and EDITDIST are returned in  }
  {  THEMARKUPRETURNVALUES.  }
     extToStr(paramPtr, ned, s1);
     numToStr(paramPtr, edit_dist, s2);
     s1 := concat(s1, c, s2);
     paramPtr^.ReturnValue := pasToZero(paramPtr, marks);
     setGlobal(paramPtr, 'theMarkUpReturnValues', pasToZero(paramPtr, s1));
    end;
  end;   { edit_trace }


{------------------------------------------------------------------------------ }

{     WORD ORDER ANALYSIS    }

{------------------------------------------------------------------------------ }


{-------------------------------------------------------------------Fill_editd_matrix  }

{   Get least edit distance between each pair of words (M, R) where M is taken from   }
{   the correct answer and R comes from the student's response.  The procedure   }
{   takes synonyms into account, so that the synonym M with minimal edit distance   }
{   from a given R is used to determine edit distance.   }

{   The normelized edit distance for each pair is computed.  If this distance is   }
{   greater than a certain criterion, prop_errors, then the two words are considered   }
{   to be different and an "infinite" distance assigned.  If prop_errors is less   }
{   than the criterion, then the actual distance is assigned, scaled to make it an   }
{   integer between 1 and editWeightScale.   }

  procedure fill_editd_matrix (var mw, rw: inputwsvector; var mwseq: inputwivector; var sim: wlmatrix; var simseq: wimatrix);

   var
    m, mp, r, p: INTEGER;
    d: LONGINT;
    ned: REAL;
    spellmarks: wordstr;

  begin

{   Create a list (set) of response words which exactly match ignorable model words.  }
   rignore := [];
   for m := 1 to mwk do
    if m in mignore then
     for r := 1 to rlg do
      begin
       sim[r, mwseq[m]] := infinity;
       if mw[m] = rw[r] then
        rignore := rignore + [r]
      end;

{   Look at each response word in turn.  }
   for r := 1 to rlg do

{  If the response word is ignorable word, it cannot match any model position.  }
    if r in rignore then
     for mp := 1 to mwk do  {  Exclude all model positions.  }
      sim[r, mp] := infinity
    else
     for m := 1 to mwk do
      begin
{   Using m as an index for POSITION, initialize distance between each response   }
{   word and model position to the default of "infinity".  Since mlg will always   }
{   exceed the number of positions, this will initialize all positions, and some   }
{   cells beyond that.   }
```

```
      sim[r, m] := infinity;
      simseq[r, m] := -1;

{   Get the edit distance between the current response and model words.  The   }
{   spellmarks are not actually returned here, but a parameter is required at that   }
{   position. }
      ned := nedit_dist(rw[r], mw[m], spellmarks, marksP^, False, shortcut);

{ If the normalized edit distance exceeds a criterion value, then round   }
{   it to infinity; otherwise, convert it to an integer between 1 and editWeightScale.  Shortcut   }
{   may be used -- words differing too much in length assigned infinite distance.   }
      if ned <= prop_errors then
      d := Trunc(editWeightScale * ned)
      else
      d := infinity;

{   Get the position in the model which the current model word corresponds to.   }
{   (several synonyms may share the same position, but note that p will always be   }
{   less than m, hence a cell referenced by p will already have been initialized.   }
      p := mwseq[m];

{  If response word is like ignorable model except for cap and accent errors, then }
{  ignore it and make it unmatchable with every model position.  MP is model position index.  }
      if (m in mignore) then
      if (edit_dist < winsert) then
      begin
      rignore := rignore + [r];
      for mp := 1 to mwk do
      sim[r, mp] := infinity;
      end
      else
                                    {  If R isn't close to ignorable M, leave distance infinite.  }
{   Otherwise, if the current distance is smaller than what is already entered in this cell,   }
{   then replace the cell contents with the new distance, and keep track of the model   }
{   word number used to fill this model position.  This means that when there are several   }
{   synonyms occupying a single position in the pattern, that the model word ultimately   }
{   used will always be the one which is closest to the response word, and the distance   }
{   entered in the cell will always be the minimum possible for the given synonym   }
{   list.  This is essential for proper handling of synonyms.  The actual model word   }
{   matched must be remembered so that an appropriate spelling markup can be   }
{   generated later.   }
      else if d < sim[r, p] then
      begin
      sim[r, p] := d;
      simseq[r, p] := m;
      end;

     end;  { FOR  m := 1 TO mwk;   ELSE;  FOR r := 1 TO rlg }

{   The matrix has the dimensions rlg X plg.  Initializations of other cells are bogus.   }
   if trace then
    see_nedit_matrix('End of FILL_EDITD_MATRIX');

  end; { fill_editd_matrix }

{-------------------------------------------------------------List_possible_matches  }

{   Generate a list of possible matches, in set format, for each response word   }
{   position.  }

  procedure list_possible_matches;

   var
    r, m: INTEGER;

  begin
   rmatched := [];
   mmatched := [];
   rmaybe := [];
   mmaybe := [];
   for r := 1 to rlg do
    begin
     choices[r] := [];
     runtogether[r] := -1;
     for m := 1 to plg do
      if s[r, m] < infinity then
       begin
       choices[r] := choices[r] + [m];
       rmaybe := rmaybe + [r];
       mmaybe := mmaybe + [m];
       if a[r, m] = 0 then
       begin
       mmatched := mmatched + [m];
       rmatched := rmatched + [r];
       end;
       end;
    end;
   if trace then
    begin
     showset('rmatched', rmatched);
     showset('mmatched', mmatched);
```

```
      showset('rignore', rignore);
      showset('mignore', mignore);
      for r := 1 to rlg do
       showset(concat('choices R= ', NtoS(r)), choices[r]);
     end;
  end;  { list_possible_matches }

{-------------------------------------------------------------Find_runtogether  }

{   This procedure called after the initial pass at spell matching has been done.   }
{   For each unmatched word in r, all adjacent UNMATCHED pairs of positions, m, mm,   }
{   in the model are examined in turn to see if r matches the concatenation of m   }
{   with mm (possibly allowing for some misspelling). This means that run-together   }
{   words are identified as such only if they appear in the exact order specified   }
{   by the model. When forming the pairs m, mm, the following complications   }
{   must be taken into account:   }

{   1. Ignorable words in the model must be left out of consideration when   }
{   determining adjacency, so that m and mm will be considered adjacent if   }
{   separated by nothing but ignorable words. E.g., in  a < b c > d, a and d are   }
{   adjacent.   }

{   2. Synonym lists must receive special treatment. Suppose two adjacent   }
{   synonym lists [ a b ] [ d e f ], with each of them unmatched (i.e., no r   }
{   has matched any synonym at either position). Then r must be compared to every   }
{   member of the cartesian product of the two lists: ad, ae, af, bd, be, bf.   }
{   Likewise, for a [ b c d ], r must be compaired to ab, ac, and ad.   }

  procedure find_runtogether;

   var
    r, m, mm, xm, xmm, p, pp: INTEGER;
    unmatchedm, unmatchedr: wordset;

   label
    1;

{---------------------------Next_position  }

{   Seach through list of model words and return index of first word right of 'start'   }
{   which (a) is not an ignorable word, and (b) is not a synonym of the word at   }
{   'start'. If the word found is part of a synonym list, it will always be the first   }
{   member of that list. If no word of this sort can be found beyond 'start', return 0.   }

   function next_position (startingmw: INTEGER): INTEGER;

    var
     i, lastp: INTEGER;

   begin
    next_position := -1;
    if startingmw = -1 then
     EXIT(next_position);
    lastp := mwseq[startingmw];
    for i := Succ(startingmw) to mwk do
     if (mwseq[i] <> lastp) and (mwseq[i] in unmatchedm) then
      begin
       next_position := i;
       EXIT(next_position);
      end;
   end;  { next_position }

{--------------------------Try_to_split_rw  }

{   Compare response word r to the run-together string consisting of model words   }
{   m and mm. If spelling analysis yields an edit distance of less than split_criterion,   }
{   then (a) match r with the word at mm; (b) mark the "run on" portion of r which   }
{   corresponds to the word at mm as ignorable, so that it will not be marked up   }
{   as a missing word; (c) remove both p and pp from the list of unmatched positions   }
{   so that they will not be matched to some other r; (d) add p to the list of choices   }
{   available to assigning to r during the order analysis. No markup and no short-   }
{   cut used when computing edit distance.   }

   function try_to_split_rw: BOOLEAN;

    var
     d: REAL;
     dummy, runtogetherword: wordstr;

   begin
    try_to_split_rw := False;
    runtogetherword := concat(mw[xm], space, mw[xmm]);
    d := nedit_dist(rw[r], runtogetherword, dummy, marksP^, False, False);
    if runtogetherflag and ((edit_dist = wdelete) or (d <= runon_criterion)) then
     begin
      try_to_split_rw := True;
      a[r, p] := Trunc(editWeightScale * d);
      aseq[r, p] := xm;
      runtogether[r] := xmm;
      choices[r] := choices[r] + [p];
      mignore := mignore + [pp];
```

```
      unmatchedm := unmatchedm - [p, pp];
      unmatchedr := unmatchedr - [r];
     end;
   end;  { try_to_split_rw }

{--------------------------------}

  begin  { find_runtogether }

{   Get set of unmatched m positions (not matched and not ignorable). Also set of   }
{   unmatched r words.   }
   unmatchedm := [1..plg] - mmaybe - mignore;
   unmatchedr := [1..rlg] - rmaybe - rignore;

   if trace then
    begin
     showset('unmatchedm :', unmatchedm);
     showset('unmatchedr :', unmatchedr);
    end;

   for r := 1 to rlg do
    begin
     runtogether[r] := 0;
     if r in unmatchedr then
      begin
{   Get pairs of adjacent model positions, p, pp.  In determining adjacency,   }
{   ignorable words are neglected, and a synonym list counts as one position.   }
{   If the next position is a single word, then 'next_position' returns the index   }
{   number of that word; if it is a synonym list, then it returns the index number   }
{   of the first word in that list.  When there are no further such pairs, either   }
{   m or mm will be returned as 0.   }
       m := next_position(0);
       mm := next_position(m);
       if mm = -1 then
       EXIT(find_runtogether);
       while mm <> -1 do
       begin
{   Get the position numbers of the words.   }
       p := mwseq[m];
       pp := mwseq[mm];
{   Verify that both positions are still unmatched.  If not, they cannot be   }
{   matched against the possibly run-together r, and we must move on to the next   }
{   pair of m, mm.   }
       if [p, pp] <= unmatchedm then
{  This double loop takes care of cases where either m or mm, or both, head a   }
{   synonym list.  In this case, r must be tast d against all combinations of words   }
{   w, ww, where w is drawn from the synonym list headed by m, and ww is drawn from   }
{   from the list headed by mm. In the common case where neither m nor mm heads a   }
{   synonym list, each loop executes once only. The loops operate by starting at   }
{   m (mm) and advancing rightword one word at a time to the end of the synonym list,   }
{   signaled when the position number associated with the current word changes.   }
       begin
       xm := m;
       while p = mwseq[xm] do
       begin
       xmm := mm;
       while pp = mwseq[xmm] do
       begin
{   Here rw[ r ] is tested for a match with the run-together string mw[ xm ] + mw[ xmm ] .   }
{   If the match succeeds, then exits the m, mm loops and start work on the next r.   }
       if try_to_split_rw then
       goto 1;
       Inc(xmm);
       end;
       Inc(xm);
       end;
       end; { IF [ p, pp ] <= unmatchedm }
{   Move right to next pair of adjacent model positions.   }
       m := next_position(m);
       mm := next_position(m);
       end; { WHILE ( m > 0 ) AND ( mm > 0 ) }
      end; { IF r IN unmatchedr }
1:
    end; { FOR r := 1 TO rlg DO }
{   Update list of matched response words and model positions.   }
   mmatched := [1..plg] - mignore - unmatchedm;
   rmatched := [1..rlg] - rignore - unmatchedr;

  end; { find_runtogether }

{-------------------------------------------------------------------Search_sequences  }

{  Core procedure of the order analysis; actually produces an optimal matching of }
{  model and response words.  Optimal means that the match returned is at least as }
{  good as any other match which could be generated.  If A and B are two matchings, }
{  A is defined to be better than B if (i) A has less inversions than B, or, if  }
{  A and B have the same number of inversions, (ii) the final inversion is as }
{  far to the right as possible.  Consider this example, where a, b, c ... symbolize }
{  full words: }

{   Model word:                a   b   c   a   b   c          }
```

```
{   Position number:            1   2   3   4   5   6          }

{   Response word:              b   c   a   b   c   a          }
{   Position number:            1   2   3   4   5   6          }

{ For each response word, several model words, at different positions in the}
{ model, may match:}

{   Response position:          1   2   3   4   5   6          }
{   ------------------------------------------------}
{   Matching models words:      2   3   4   2   3   4          }
{                               5   6   1   5   6   1          }

{  A matching is genereted by choosing one of the numbers (i.e., model words) }
{  in each column, subject to the restriction that no number be chosen twice. }
{  Possible matchings are indicated in the table below.  Of course, a model and }
{  response word can only be matched if they are sufficiently similar (ideelly, }
{  identical).}

{   Response word position:     1   2   3   4   5   6          }
{   ------------------------------------------------ }
{   One possible matching:      2   3   4   5   6   1          }
{   A second matching:          5   6   1   2   3   4          }
{   A third matching:           2   6   1   5   3   4          }
{                                       ...                    }

{  An inversion occurs whenever two successive numbers invert their natural order; e.g.,}
{  the first matching has one inversion, at  6 - 1.  The third has two:  at}
{  6 - 1 and at 5 - 3.}

{  The algorithm generates all possible matchings in a depth-first manner, }
{  moving forward in the sequence of response-word positions by recursive }
{  descent, counting the number of inversions along the path as it goes, and }
{  keeping track of the position of the rightmost inversion.  When a matching is }
{  complete, the algorithm checks to see if it is at least as good as the matchea  }
{  generated so far and, if so, saves it in a list of solutions.  }

{  In the worst case, where there were N response words and N model words, all  }
{  identical, there would be N candidata words to fill each response position, }
{  and hence N! paths to check, leading to a near-exponential algorithm.  In fact, }
{  the algorithm turns out to be fairly efficient, for several reasons: }

{  1.  In actuality, even for fairly pathological cases such as cyclic and near- }
{  cyclic patterns, there are relatively few choices for matching at each response }
{  position.}

{  2.  The algorithm does extensive tree pruning.  As soon as it becomes clear that }
{  a path cannot be optimal (because the number of inversions has exceedad the }
{  minimum so far found), work is immediately terminated on that path and all }
{  subpaths.  This drastically reduces the search space.  In practice, it appears }
{  that search time is roughly quadratic. }

{  Input parameters: }
{  remaining_choices    }
{           A list (in set format) of all the model words which have }
{           not yat been matched, and thus are still available for }
{           matching. }
{  solution      A record containing a description of this matching as so }
{           far developed, including sequence of model word numbers, }
{           inversion count, and position of rightmost invarsion. }
{  lastchosen    Position of last model word chosen.  When no model word }
{           matches a response, an arbitrary value of '0' is assignad }
{           to tha solution sequence, but 'lastchosen' retains its }
{           prior velue in this case, so that the inversion count will}
{           not be made against "extra" words.}
{ p         Response word position at which matching should take place.}

{(Global) data structures affected:}
{solutionlist    A list of optimal solutions.  each solution is a record}
{           containing the actual matching ( the sequence into which}
{           the model words must be rearranged to match the responsa),}
{           the number of inversions, and the position of the rightmost inversion.}
{solutionk       Number of records on the solution list (starts with #1).}

  procedura search_sequences (remaining_choices: wordset; solution: solutionrec; lastchosen: BYTE; p: INTEGER);

   var
    chosen: wrange;
    xsol: solutionrec;
    available_choices: wordset;
    i: INTEGER;

{------------------Choose_next  }

{   Chooses the first (leftmost) of a list of words whose positions are represented   }
{   in set format.  }

   function choose_next (wset: wordset): wrange;

    var
     i: INTEGER;
```

```
  begin
   i := 0;
   repeat
    Inc(i)
   until (i in wset);
   choose_next := i
  end;  { choose_text }

{-----------------Save_solution  }

{ Pushes a solution onto the solution list (a stack).  }

  procedure save_solution (sol: solutionrec);

  begin
   if solutionk < wmax then
    Inc(solutionk);
   solutionlist[solutionk] := sol
  end;

{----------------- Trace_solution  }

  procedure tracesolution;

   var
    i: INTEGER;

  begin
     {Write('' : 2 * p);}
     {for i := 1 to Length(solution.seq) do}
     {Write(Ord(solution.seq[i]) : 2);}
     {Write(' invk=', solution.inversionk : 2, ' firstinv=', solution.firstinv : 2, ' p=', p : 2);}
     {Writeln;}
  end;  { tracesolution }

{-------------------------}

 begin  {  search_sequences  }

  if trace then
   begin
    tracesolution;
    pause;
   end;

{  This is the termination clause.  If we have run out of response words to  }
{  match, this path is complete.  Check the cost ( number of inversions ) in this  }
{  matching, and if it is more efficient than those currently on the stack,  }
{  then clear the stack and start it over with this solution; otherwise,  }
{  simply add the solution to those already present and exit, returning to earlier  }
{  recursions.  }
  Inc(recursionk);
  if (p > rlg) then
   begin
    if (solution.inversionk < mincost) or ((solution.inversionk = mincost) and (solution.firstinv > rightmost)) then
     begin
      mincost := solution.inversionk;
      rightmost := solution.firstinv;
      solutionk := 0;
      save_solution(solution);
     end;
    EXIT(search_sequences);
   end;

{  Otherwise, we are still generating a path by recursion.  Make a working copy  }
{  of the solution, so that the partial solution state will be preserved upon  }
{  return.  Find out what matching choices are available for this response word  }
{  by restricting the choices at this position to those not used at previous  }
{  positions.  If no choices are available, match this respose word to the dummy  }
{  model word # 0 (this means treating the response word at this position as an  }
{  extra word), and recurse to match the next position rightward.  }
  xsol := solution;
  available_choices := (remaining_choices * choices[p]);
  if available_choices = [] then
   begin
    xsol.seq := concat(solution.seq, Chr(0));
    search_sequences( remaining_choices, xsol, lastchosen, p + 1);
   end
  else
{  If one or more words are available for match, choose each of them in turn  }
{  (this is done iteratively using the WHILE loop).  Use this choice to extend the  }
{  matching, updating the word sequence, the number of inversions, and the position  }
{  of the rightmost inversion.  Notice that 'available_choices', which records the  }
{  choices not yet tried at this loop will shrink each time though the loop, while  }
{  'remaining_choices', which records words not yet entered into the match, will  }
{  stay unchanged.  }
   while available_choices <> [] do
    begin
     xsol := solution;
     chosen := choose_next(available_choices);
```

```
      available_choices := available_choices - [chosen];
      xsol.seq := concat(solution.seq, Chr(chosen));
      if chosen < lastchosen then
       begin
       Inc(xsol.inversionk);
       if xsol.firstinv = 0 then
       xsol.firstinv := p;
       end;
{ If the matching as so far developed is as good or better than any solution}
{so far found, then continue this matching by recursive descent to the next}
{response word position.  Otherwise, abandon this matching, and do not try to}
{extend any matches from this point in the search tree (tree pruning).  Simply}
{continue the loop, choosing another matching possibility at this position,}
{if any remain.  Note that before it is passed to the next level, the set of}
{'remaining_choices' must have the current choice removed.}
{N.B. - Still greater pruning efficiency could be obtained by retaining a}
{solution only as long as it remained strictly better than the current solutions}
{( inversionk < mincost ). This is not done here so that the secondary criterion}
{of rightmost final inversions position can be applied. }
      if (xsol.inversionk < mincost) or ((xsol.inversionk = mincost) and (xsol.firstinv > rightmost)) then
       search_sequences(remaining_choices - [chosen], xsol, chosen, p + 1);
     end; { WHILE }

{ If all the possibilities at this position have been used, return to the}
{previous level of recursion and work on further possibilities there, generating}
{new branches in the search tree. }

  end;  { search_sequences }

{-----------------------------------------------------------------Adjust_solution  }

{  The strictly left-to-right recursion of the matching algorithm unfortunately  }
{  leads to situations like this one:  }

{   Model :              the time        }
{   Response:            time  then  the    }
{   Markup generated:    ^       <   x  XXX    }
{   More intuitive markup:  ^       XXXX <      }

{  This comes about because the matching algorithm does not consider variations in  }
{  edit distance when matching words -- it only knows that a pair is or is not a  }
{  permissible match.  Hence a misspelled word is just as good a candidate as  }
{  a perfect match.  When several response words match a given model word, the  }
{  leftmost is always selected in preference to the 'redundant' rightward versions,  }
{  even if the rightward versions are better spelled, and hence intuitively better  }
{  matches.  E.g., in the response above, 'then' is always selected to match 'the'  }
{  in the model, leading to a counter-intuitive markup.}

{  This procedure scans the solution, looking for cases where a rightward word  }
{  would consititute a better fit than the current assignment, and adjusts the  }
{  solution accordingly, producing, e.g., the improved, 'intuitive' markup shown  }
{  above.  Inversion count may be affected in cases like this:  }

{  Model:            the   time        }
{  Response:          then   time   the    }
{  Markup:                x   XXX      }
{  'Improved' markup:  XXXX^          <    }

{  and it is not clear that this really represents an improvement.  Hence, the  }
{  inversion count of each adjusted solution is checked, and if the number of  }
{  inversions is increased, the proposed adjustment is not accepted.  }

  procedure adjust_solution (var sol: solutionrec);

   var
    r, ri, m, mi, ulim, llim, invComp: INTEGER;
    dll, dl2, d21, d22, edl, ed2: LONGINT;
    rsave: CHAR;
    s: str80;
    strgl, strg2, strg: str255;
    validwords: wordset;

{-----------------CompInvCount  }

{  Compares the number of inversions in an old solution and a new solution, and  }
{  returns -1 if the new solution has strictly less inversions than the old one,  }
{  0 if the number of inversion is the same, or 1 if the new solution has more  }
{  The old solution is specified in the input parameters by a ptr into the  }
{  sequence field of a solution record, typecast into a byte array.  The proposed  }
{  new solution is specified as a possible inversion, where the M belonging to r  }
{  is to be exchanged with the M belonging to ri.  }

   function compInvCount (s: str80; r, ri: INTEGER): INTEGER;

    var
     xs: str80;
     i, k, xk, c, lasti, lastxi: INTEGER;

   begin
{  Build a sequence list for the proposed new solution.  }
    xs := s;
```

```
    xs[r] := s[ri];
    xs[ri] := s[r];
{  Initialize inversions counters and bookkeeping for counting loop.  }
    k := 0;
    xk := 0;
    lasti := 0;
    lastxi := 0;
{  Count invesions in both the old and the proposed new solution in parallel.  }
    for i := 1 to rlg do
     begin
      c := Ord(xs[i]);
      if (c <> 0) then
       begin
       if (c < lastxi) then
       Inc(xk);
       lastxi := c;
       end;
      c := Ord(s[i]);
      if (c <> 0) then
       begin
       if (c < lasti) then
       Inc(k);
       lasti := c;
       end;
     end;
{  Return boolean which tells whether the proposed solution is at least as good  }
{  in terms of number of inversions.  }
    if (xk < k) then
     compInvCount := -1    {new sol has less inversions}
    else if (xk = k) then
     compInvCount := 0      {new & old have same number of inversions}
    else
     compInvCount := 1;     {old sol has less inversions}

   end;  { compInvCount }

{------------------------  }

  begin   { adjust_solution  }

{  Build a list, in set format, of all unmatched response positions. }
   s := sol.seq;
   validwords := [1..rlg] - rignore;

{  Look in turn at each valid response word R, and the model word it is matched  }
{  to, M.  Look for another valid response word say R' matched to M' such that exchanging the  }
{  match, so that R is matched to M' and R' is matched to M would improve the overall solution,  }
{  (because it either (a) causes no more inversions and improves total edit distance for sentence) OR,  }
{  (b) causes less inversions, and does not increase total edit distance.  Whenever such R, R' can be found,  }
{  then exchange the match so R goes with M' and R' goes with M. }
   for r := 1 to rlg do
    if (r in validwords) then
     for ri := 1 to rlg do
      if (ri in validwords) then
       begin
       m := Ord(s[r]);
       mi := Ord(s[ri]);
       if m = 0 then
       begin
       d11 := infinity;
       d21 := infinity;
       end
       else
       begin
       d11 := a[r, m];
       d21 := a[ri, m];
       end;
       if mi = 0 then
       begin
       d12 := infinity;
       d22 := infinity;
       end
       else
       begin
       d12 := a[r, mi];
       d22 := a[ri, mi];
       end;
       ed1 := d11 + d22;
       ed2 := d12 + d21;
       invComp := compInvCount(s, r, ri);
       if ((ed2 < ed1) & (invComp <= 0)) | ((ed2 = ed1) & (invComp < 0)) then
       begin                                  { Exchange assignments so R <-> M' and R' <-> M. }
       rsave := s[ri];
       s[ri] := s[r];
       s[r] := rsave;
       end
       end;
   sol.seq := s;

  end;  { adjust_solution }
```

```
{-------------------------------------------------------------Find_best_order }

{  Has overall control of the order analysis.  Accepts a matrix of (normalized)  }
{  edit distances as input, and returns a single optimal matching as a solution,  }
{  in the form of a mapping of model to response words.  The mapping is represented  }
{  in the vectors r_to_m and (with index variable inverted) m_to_r.  }

  procedure find_best_order;

   var
    i: INTEGER;
    totDist: LONGINT;
    r, m, p: 0..wmax;
    sol: solutionrec;
    rightmost: INTEGER;
    s: str80;

  begin

{  Initialize variables used by order matching algorithm, and call the algorithm  }
{  at top level of recursion.  Set of positions initially available consists of all  }
{  positions in the model.  }
   sol.seq := '';
   sol.inversionk := 0;
   sol.firstinv := 0;
   mincost := infinity;
   rightmost := 0;
   solutionk := 0;
   solutions_tried := 0;
   recursionk := 0;
   search_sequences([1..plg], sol, 0, 1);

{  Adjust solution to improve match with respect to spelling accuracy.  }
   if adjust_needed then
    adjust_solution(solutionlist[1]);

{  Build a mapping of response-to-model and model-to-response words.  These  }
{  will be used to generate the sentence markup.  Unmatched words are assigned to  }
{  dummy word #0.  }
   s := solutionlist[1].seq;
   matchedk := 0;
   avedist := 0.0;
   totDist := 0;
   mmatched := [];
   rmatched := [];
   for i := 1 to wmax do
    begin
     m_to_r[i] := 0;
     r_to_m[i] := 0
    end;
   for r := 1 to rlg do
    begin
     m := Ord(s[r]);
     r_to_m[r] := m;
     if m > 0 then
      begin
       rmatched := rmatched + [r];
       mmatched := mmatched + [m];
       m_to_r[m] := r;
       totDist := totDist + a[r, m]; { total scaled normalized edit dist }
       Inc(matchedk);
      end;
    end;
   if matchedk > 0 then
    avedist := totDist / (LONGINT(editWeightScale) * LONGINT(matchedk));

{  Compute proportion of matched words and proportion of non-inversions.  These  }
{  values are used to decide whether a response should be judged as a match (ok)  }
{  or a non-match (no) to the model pattern.  }
   i := card([1..rlg] - rignore) + card([1..plg] - mignore);
   if i > 0 then
    pmatched := (2 * matchedk) / LONGINT(i)
   else
    pmatched := 1;
   if matchedk > 0 then
    pnoninversions := 1 - (solutionlist[1].inversionk / matchedk)
   else
    pnoninversions := 1;

  end;  { find_best_order }

{-----------------------------------------------------------------Word_markup  }

{   Generate proper markup for each word in sentence.  Order markup symbols   }
{   (missing word, extra word, displaced word) are generated here.  If a   }
{   misspelled word is also out of order, a (non-blank) order mark preempts any spelling   }
{   mark which might be on the first character of the word.   }

{   (Global) Return vars:   }
{   wordmark    : vector of response word markups, spelling and order markup combined.   }
```

```
procedure word_markup;

 var
  r, m, p, lastfound: INTEGER;
  c: string[1];
  d: REAL;
  s, mword: wordstr;
  notmissing: wordset;

begin
{   Get the set of model positions which will NOT cause a missing word word markup   }
{   if absent at the expected place in the response.  If anyorderok is false,   }
{   this will consist of ignorable words.  If anyorderok is true, it will be   }
{   ignorable words plus those which appear somewhere in the sentence (i.e., are   }
{   actually matched), but are out of order.   }
  if anyorderok then
   notmissing := mignore + mmatched
  else
   notmissing := mignore;
  lastfound := 0;
  for r := 1 to rlg do
{   For each word in the response, retrieve the matched position, m, in the model,   }
{   and the corresponding word, p (if there were synonym lists in the model, then   }
{   probably m <> p.   }
   begin
    m := r_to_m[r];
    p := aseq[r, m];
{   If nothing matched to response word, and it is not an ignorable word,   }
{  and extra words are not permitted, then generate extra word markup.   }
    if r in rignore then
     wordmark[r] := ''
    else if m = 0 then
     if extrawordsok then
      wordmark[r] := ''
     else
      wordmark[r] := dup_char(extrawd, Length(rw[r]))
{   Otherwise, generate spelling markup.   }
    else
     begin
{  If current word is a runtogether, restore space before generating spellmarks.  }
      if runtogether[r] > 0 then
       mword := concat(mw[p], space, mw[runtogether[r]])
      else
       mword := mw[p];
{  If misspelling is OK, make spelling markup blank, else generate it.  }
      if misspellOK then
       s := dup_char(nomark, length(mword))
      else                                  { Return full markup in S.  }
       d := nedit_dist(rw[r], mword, s, marksP^, True, False);
{   If the model position just matched skips ahead (right) of the last model   }
{   position by more than 1, then -- unless every model word in between the two   }
{   positions is an ignorable word or unmarkable because it is merely out of order   }
{   and order is not being marked -- some model words were left out at this point   }
{   in the response, so generate  a missing word markup to go just before this   }
{   response word. }
      if (m > lastfound + 1) and not ([lastfound + 1..m - 1] <= notmissing) then
       c := missingwd
      else
       c := '';
{   If the model position matched appears in the model at a position to the left   }
{   of the last model position matched, then the response ordering inverts the model   }
{   ordering at this point.  Mark the matched response word as needing to be moved   }
{   leftward.  Do not do this, however, if anyorderok is in effect.  Preempt the   }
{   first character of the spelling markup to show the moveword symbol, unless it's a space.   }
      if (m < lastfound) then
       if (not anyorderok) & (movewd <> nomark) then
       wordmark[r] := concat(c, movewd, Copy(s, 2, Length(s) - 1))
       else
       wordmark[r] := concat(c, s)
       else
       begin
       wordmark[r] := concat(c, s);
       lastfound := m
       end;
     end;
    end; { FOR }
{   If final model position matched was not the rightmost position of model,   }
{   then some model words were left out at the end the response; mark missing   }
{   words at end of response.   }
   if not ([lastfound + 1..plg] <= mignore) then
    wordmark[rlg + 1] := missingwd
   else
    wordmark[rlg + 1] := '';
 end;  { word_markup }

{-------------------------------------------------------------------------Mark_sentence  }

{  Prepare a markup string which can be displayed  beneath words of student's response.   }
{  Task of this routine is to make sure each markup will be positioned beneath appropriate   }
{  word and letters.  }
```

```
{  Input vars:   }
{  wordmark     : (Global) vector of markup strings, one for each response word   }

  function mark_sentence: string;

   var
    r, p: INTEGER;
    m, outstr: string;

  begin

{  First char of markup should plot in position preceeding first char of response  }
{  to provide place for e leading carat when initial words ere missing.   }

{  2 extra chars to provide for leading and trailing carats.  }
   outstr := dup_char(nomark, 2 + length(response));
   for r := 1 to rlg + 1 do
{  Ignorable word chars are appended to preceding real word as if punctuation.  }
    if not (r in rignore) then
     begin
      m := wordmark[r];
{ Move ahead 1 char if no missing wd mark (remember extra char at front) }
      if (m[1] = missingwd) then
       p := rwxloc[r]
      else
       p := rwxloc[r] + 1;
{  Replace blanks et word location with markup string for the word.  }
      delete(outstr, p, length(m));
      insert(m, outstr, p);
     end;
   mark_sentence := outstr;

  end; { mark_sentence }

{-------------------------------------------------------------------CheckOrder }

{   Top-level sentence checking procedure which executes the major sub-procedures   }
{   needed to create the sentence markup.   }

  procedure checkorder;

   var
    i: INTEGER;

  begin

{   Initialize borders of spelling edit distance matrix -- this is never   }
{   cleared, so could be done in markup XFCN initialization if matrix were static.    }
   init_spelling(marksP^);

{   Build matrix of normalized edit distances between all ( M, R ) word pairs.    }
   fill_editd_matrix(mw, rw, mwseq, a, aseq);

{   Build sets that record which model and response words have matches.  For each   }
{   r word, build choices[ r ], a set of all possible matches for r.    }
   list_possible_matches;

{   Try to extend matching by looking for run-together words among those so far   }
{   unmatched.   }
   if runtogether_needed and (not anyorderok) then
    find_runtogether;

{   Apply exhaustive search algorithm to find a matching which minimizes number  }
{   of inversions.    }
   find_best_order;

{   Generate strings for order and spelling markup.   }
   if word_markup_needed then
    word_markup;

  end; { checkorder }

{-------------------------------------------------------------------Compare  }

{   Control structure for (default) full spelling and word order analysis.  }
{   Internalizes the two strings 'model' and 'response' to prepare them for use   }
{   in the order checking algorithm, then runs that algorithm.   }

{   Output data structure (global):   }
{   wordmark       : A vector of markup strings, 'wordmark', with one entry   }
{                          for each word of the response.   }

  function compare (model, response: string): string;

  begin

   judgedok := False;
   setmodel(model);
   setresponse(response);
   checkorder;
```

```
   judgedok := ((pmatched = 1.0) or (extrawordsok and (mmatched = ([1..plg] - mignore)))) and ((pnoninversions = 1.0) or anyorderok)
and ((avedist = 0.0) or misspellok);
   compare := mark_sentence;

  end;  { compare }


{-------------------------------------------------------------------------------------------------------}

{  FORMAT OUTPUT TO HYPERCARD  }

{-------------------------------------------------------------------------------------------------------}


{-----------------------------------------------------------------------FormatMarkUpMaps  }

{    Re-format the R_TO_M and M_TO_R maps into a string suitable for return    }
{    to HyperCard.  The maps are returned as lists of comma-separated integers,   }
{    R_TO_M on line 1, and M_TO_R on line 2, of a two-line string.   }

  function formatMarkupMaps: Str255;

   var
    i: INTEGER;
    s, t: Str255;

  begin

   s := '';
   for i := 1 to rlg do
    begin
     numToStr(paramPtr, r_to_m[i], t);
     s := Concat(s, t, ',')
    end;
   s[length(s)] := chr(13);  {  substitute newline for dangling comma  }

   for i := 1 to plg do
    begin
     numToStr(paramPtr, m_to_r[i], t);
     s := Concat(s, t, ',')
    end;
   s[length(s)] := chr(13);  {  substitute newline for dangling comma  }

   for i := 1 to rlg do
    begin
     numToStr(paramPtr, rwxloc[i], t);
     s := Concat(s, t, ',')
    end;

   formatMarkupMaps := omit(s, length(s), 1);     {  remove dangling comma  }

  end;   {  formatMarkupMaps  }

{-----------------------------------------------------------------------FormatParamDisplay  }

{ Access info requested by PARAMDISPLAYNEEDED and put it into HyperCard global THEMARKUPPARAMDISPLAY. }

  procedure formatParamDisplay (switch: CHAR);

   var
    i: INTEGER;
    p, q: phon_variants;
    s: Str255;
    c, ch: CHAR;
    h: handle;
    hp: PTR;

  begin

   c := ',';
   s := '';
   h := NewHandle(0);
   appendStringToHandle(h, Concat('MUParamDisplay ', switch, chr(13)));

   case switch of
    'v':
     appendStringToHandle(h, versionStr);
    'b':
     for i := 1 to 255 do
      begin
       ch := chr(i);
       if case_info[ch] <> down_case then
       appendStringToHandle(h, Concat(s, ch, '=', base_char[ch], ','));
      end;
    'd':
     for i := 1 to 255 do
      if diacrit_info[chr(i)] <> no_accent then
       appendStringToHandle(h, Concat(s, CHR(i), '=', NtoS(ORD(diacrit_info[chr(i)])), c));
    'c':
     for i := 1 to 255 do
      if case_info[chr(i)] <> down_case then
```

```
      appendStringToHandle(h, Concat(s, CHR(i), c));
    'h':
     for i := 1 to 255 do
      appendStringToHandle(h, Concat(s, CHR(i), '=', NtoS(ORD(phon_info[chr(i)])), c));
    'p':
     for i := 1 to 255 do
      if chr(i) in delim_chars then
       appendStringToHandle(h, Concat(s, CHR(i)));
    'w':
     begin
      appendStringToHandle(h, Concat(NtoS(winsert), c, NtoS(wdelete), c, NtoS(wchange), c, NtoS(wtranspose), c, NtoS(wcap), c,
NtoS(waccent), c));
      appendStringToHandle(h, Concat(EtoS(prop_errors), c, EtoS(cutoff), c, EtoS(runon_criterion)));
     end;
    'f':
     begin
      case ORD(cap_flag) of
       1:
       s := 'exact_case';
       2:
       s := 'authors_caps';
       3:
       s := 'ignore_case'
      end;   { CASE }
      appendStringToHandle(h, Concat(s, c, BtoS(anyOrderOk), c, BtoS(extraWordsOk), c, BtoS(misspellOk), c));
      appendStringToHandle(h, Concat(BtoS(word_markup_needed), c, BtoS(runtogether_needed), c, BtoS(adjust_needed), c, BtoS(shortcut),
c));
      appendStringToHandle(h, Concat(paramDisplayNeeded, c, rawTrace, c, BtoS(trace)));
     end;
    's':
     begin
      appendStringToHandle(h, Concat(addcap, dropcap, accenterr, extrawd, missingwd, movewd, extraltr, missingltr, substituteltr,
transltr1, transltr2, runonwd));
     end;
    'm':
     for p := vowel to phon5 do
      for q := vowel to phon5 do
       appendStringToHandle(h, concat(s, NtoS(phon_matrix[p, q]), c));
    otherwise
     FAIL('%Invalid info type.  Use:  Version, Base, Diacrit, Cap, Punct, pHon, Weights, Flags, markupSymbols, phon_Matrix');
   end;   {  CASE  switch  }

{ Add null char terminator for HyperCard string  }
   i := GetHandleSize(h);
   hp := PTR(ORD(h^) + i - 1);   {  Ptr to last byte of block.  }
   if (i > 0) & (hp^ = ORD(',')) then
    hp^ := 0                        { Replace trailing comma with null char terminator. }
   else
    appendStringToHandle(h, CHR(0));   {  Append null char terminator  }

{  Assign handle to global.  }
   setGlobal(paramPtr, 'theMarkupParamDisplay', h);
   disposeHandle(h);

  end;    {  formatParamDisplay  }


{-------------------------------------------------------------------------------}

{   TOP-LEVEL CONTROL STRUCTURE FOR MARKUP XFCN.   }

{-------------------------------------------------------------------------------}


{-------------------------------------------------------------------------MarkUp}

{  Top-level controlling procedure which unpacks HyperCard parameter values,  }
{  generates markup and sets up values for return to HyperCard.   }

  procedure markUp (paramPtr: XCmdPtr);

   const
    c = ',';

   var
    h: handle;
    p: PTR;

{  ------------------------- GetStringParam }

{   Converts the PARMNUM-th XFCN input parameter to a string.   }

   function getStringParam (parmNum: INTEGER): Str255;

    var
     s: Str255;

   begin

    if (paramPtr^.paramCount < parmNum) then
     getStringParam := ''
```

```
    else
     begin
      ZeroToPas(paramPtr, paramPtr^.params[parmNum]^, s);
      getStringParam := s;
     end

   end;  { getStringParam }

{  ------------------------ GetCharParam }

{   Converts the PARMNUM-th XFCN input parameter to an integer.   }

   function getCharParam (parmNum: INTEGER; default: CHAR): CHAR;

    var
     s: Str255;

   begin

    if (paramPtr^.paramCount < parmNum) then
     getCharParam := default
    else
     begin
      ZeroToPas(paramPtr, paramPtr^.params[parmNum]^, s);
      if length(s) < 1 then
       getCharParam := default
      else
       getCharParam := s[1];
     end

   end;  { getCharParam }

{  ------------------------GetBooleanParam }

{   Converts the PARMNUM-th XFCN input parameter to a boolean value.  }
{   If the parameter is empty, then DEFAULT is assigned as the value.  }

   function getBooleanParam (parmNum: INTEGER; default: BOOLEAN): BOOLEAN;

    var
     s: Str255;

   begin

    s := getStringParam(parmNum);
    if s = '' then
     getBooleanParam := default
    else
     begin
      getBooleanParam := strToBool(paramPtr, s);
      if paramPtr^.result <> noErr then
       FAIL(concat('%Bad boolean input param value', s))
     end;

   end;  { getBooleanParam }

{  ---------------------------GetCapParam }

{   Converts the PARMNUM-th XFCN input parameter to a cap variant value.  }
{   If the parameter is empty, then DEFAULT is used.  }

   function getCapParam (paramNum: INTEGER; default: cap_flag_type): cap_flag_type;

    var
     s: Str255;

   begin

    s := getStringParam(paramNum);
    if s = '' then
     getCapParam := default
    else if eq(s, 'exact_case') then
     getCapParam := exact_case
    else if eq(s, 'authors_caps') then
     getCapParam := authors_caps
    else if eq(s, 'ignore_case') then
     getCapParam := ignore_case
    else
     FAIL(concat('%Bad cap_flag input param value:  ', s))

   end;   { getCapParam }

{  ------------------------------ }

  begin     { markUp }

   p := nil;     { Cuz FAIL operates on P.  }

{   Check input parameter syntax.   }

   if (paramPtr^.paramCount = 0) then
```

```
    FAIL(versionStr);

   if (paramPtr^.paramCount < 2) then
    FAIL('%MODEL and RESPONSE parameters required');

{   Clear debug global.  }
   returnInGlobal('theMarkUpDebug', '');

{   Get memory from Mac heap.  }
   p := NewPtr(sizeOf(lsmatrix));
   if p = nil then
    FAIL('%Couldn''t get matrix memory.')
   else
    marksP := LSMATRIXPTR(p);

{    Unpack the input parameters and format them as Pascal variables.   }
{    Default settings are used if no parameter or empty parameter value.    }

   model := getStringParam(1);
   response := getStringParam(2);
   cap_flag := getCapParam(3, exact_case);
   extraWordsOK := getBooleanParam(4, FALSE);
   anyOrderOk := getBooleanParam(5, FALSE);
   misspellOK := getBooleanParam(6, FALSE);
   word_markup_needed := getBooleanParam(7, TRUE);
   runtogether_needed := getBooleanParam(8, TRUE);
   adjust_needed := getBooleanParam(9, TRUE);
   shortCut := getBooleanParam(10, TRUE);
   markupMapsNeeded := getBooleanParam(11, FALSE);
   paramDisplayNeeded := getCharParam(12, 'x');
   rawTrace := getCharParam(13, 'x');
   trace := getBooleanParam(14, FALSE);

{    Initializes all static data structures, including the char info tables, punct table,  }
{    markup symbol table, phon_matrix, weights and threshold values.  }
   init_markup;

{   Format and return markup parameter display via global variable 'theMarkupParamDisplay'.   }
   if paramDisplayNeeded <> 'x' then
    formatParamDisplay(paramDisplayNeeded);

{   Do all the markup work here; return the markup symbol string as the value of the XFCN.  }
   if rawTrace = 'x' then
{  If requested full (defsult) spelling and word order analysis.  }
    begin

{   Compute markup string as direct return.  }
     paramPtr^.ReturnValue := PasToZero(paramPtr, compare(model, response));

{    Format and return judging information via global variable "theMarkUpReturnValues".   }
     returnInGlobal('theMarkUpReturnValues', concat(BtoS(judgedOk), c, EtoS(pMatched), c, EtoS(pNonInversions), c, EtoS(aveDist)));

  {   Format and return matching map information via global variable 'theMarkupMaps'   }
     if markupMapsNeeded then
      begin
       h := pasToZero(paramPtr, formatMarkupMaps);
       if h = nil then
       FAIL('%Out of memory while formating markup maps.')
       else
       begin
       setGlobal(paramPtr, 'theMarkupMaps', h);
       disposHandle(h)
       end
      end
    end
   else
{  If pure least-edit-trace analysis on input strings was requested, }
{  then generate edit distances and edit trace on raw input strings.  }
    begin
     edit_trace(model, response);
    end;

{  Get rid of dynamic memory.  }
   disposPtr(p);

  {   Don't pass the MARKUP message up HyperCard's inheritance structure.   }
   paramPtr^.PassFlag := FALSE;

  end;   { markUp }


{-------------------------------------------------------------------------------}

{  MAIN  }

{-------------------------------------------------------------------------------}


 begin   {  main  }

  markup(paramPtr);
```

```
end;    { main }
end.    { unit markupXFCN }
```