

LINGUISTIC TOOLS for a PHONOLOGICAL EXPERT SYSTEM

The Domain of this Project

The concept of a phonological expert system involves much more than what I have attempted to do in this project. It is fair to say that what I have done is created some vital "tools" which a phonological expert system can eventually use. The tools are as follows:

1. Distinctive Features (DF) table inquires.
2. Prefix and Suffix rule combinations of regular forms.
(Harmony rules are excluded for now)

This part of the PHONological Expert system (PHONEX) is not "AI-ish" in the true sense of the word but one can not attempt to build or experiment with PHONEX until the tools are in place. Therefore, I will only briefly discuss how I envision the future structures and concepts of PHONEX focussing most of my attention on the tools which I have created.

What is a PHONOLOGICAL expert system?

PHONEX will be an expert system which will be able to analyze the phonological surface forms of a given language and break them down into rules or exceptions. This data would then be used to assist in the understanding and generation of speech and will eventually enable a system to constantly update and analyze occurring speech. Thus, when the system encounters a new word it can successfully use the current rule system to generate its essential forms (i.e. plurals, person...).

The data which PHONEX will eventually start with will be from two sources:

1. the surface forms stored in a DF table
2. and the surface forms of words stored in FRAMES.

The knowledge representation in PHONEX is yet to be determined. Before the knowledge base is constructed, certain tools must be made. One of these tools is rule generations for prefixes and suffixes. That is what my program is about.

POSSIBLE REGULAR RULE GENERATION FOR PREFIXING AND SUFFIXING

The program which I have written for my independent project involves generating the possible rules for prefixes and suffixes of a given list of surface forms (those which occur naturally). Below is a demonstration of my program. The atom PLURALS is bounded to a list containing lists of a word and its plural.

*PLURALS

((aks akse^n)	(be bez)	(bUS bUSE^z)	(bUK buKs)	(bv^g bv^gz)
(bv^s bv^se^z)	(flf0 flf0s)	(k0t k0ts)	(kaz kaze^z)	(lId lIdz)
(llv llvz)	(l0w l0wz)	(li' li'z)	(pol polz)	(pv^f pv^fs)
(ru3 ru3e^z)	(sr^ sr^z)	(sv^m sv^mz)	(swIN>g swIN>gz)	(t0b t0bz)
(tIp tIps)	(uz uze^z)			

*(POSSIBLE-REGULAR-AS-RULES PLURALS)

(NIL

```
(( (e^z z ((+voi) (-cp) NIL NIL NIL)) ((*Dl* e^)))
  ((z s ((+obs -back -voi)
          (+cont -tnse -cp -cor -lat -nas)
          (-back -cp -lat -loja -nas -str) NIL)) ((*Cl* z (-voi)))))

(( (z e^z ((+obs +cont +cor)
          (+syl -cons -loja -nas) NIL)) ((*Il* e^)))
  ((z s ((+obs -back -voi)
          (+cont -tnse -cp -cor -lat -nas)
          (-back -cp -lat -loja -nas -str) NIL)) ((*Cl* z (-voi)))))

(( (s z ((+voi) (-cp) NIL NIL NIL)) ((*Cl* s (+voi))))
  ((z e^z ((+obs +cont +cor)
          (+syl -cons -loja -nas) NIL)) ((*Il* e^))))

(( (z s ((+obs -back -voi)
          (+cont -tnse -cp -cor -lat -nas)
          (-back -cp -lat -loja -nas -str) NIL)) ((*Cl* z (-voi))))
  ((z e^z ((+obs +cont +cor)
          (+syl -cons -loja -nas) NIL)) ((*Il* e^)))))
```

The rules are grouped into two halves: the first being for the prefixes and the second for the suffixes. In the above example, the NIL represents the prefixes meaning that there are none. The second part of the example includes four groups of rules which each of which are broken up in the following segments:

```
(( (s z      [ (+voi) (-cp) NIL NIL NIL]      [( *Cl* s (+voi)) ] )
  :
phoneme-transaction      stem-environment      action
```

Phoneme-transaction

This shows which group of phonemes are changing to which. In the simple rule above, "s z" means that the following rule changes the phoneme "s" to "z".

Stem-environment

This is the environment which must be met if the rule is to be fired. If the rule is an prefixing rule then the environment is listed in the correct order. If the rule is a suffixing rule then the environment is listed in reverse (from closest to the suffix to farthest).

Action

The action can be one of three forms: change (*Cl*), insert (*Il*), or delete (*Dl*). The number in the mnemonic (such as *Cl*) tells which character slot the action affects. The phoneme following the mnemonic is the phoneme which is affected. In the case of change (as in the previous example), the phoneme is followed by the features which are to be changed.

Thus, the previous example

```
(( (s z      [ (+voi) (-cp) NIL NIL NIL]      [( *Cl* s (+voi)) ] )
```

would be interpreted linguistically as:

"s" in the environment of [-co +voi]

becomes [+voi]

The correct rule in English is:

"s" in the environment of [+voi]

becomes [+voi]

If you look carefully, the correct rule is a generalization of the program generated rule. This generalization will be part of the "true" AI portion of this program which has yet to be written.

THE THEORY BEHIND POSSIBLE RULE GENERATION

The function POSSIBLE-REGULAR-PS-RULES is broken down into five main steps:

1. separation of the prefixes, stems, and suffixes.
2. weeding out the exceptions.
3. converting the stems into distinctive feature environments.
4. generating all possible rules from the pairing of known prefixes or suffixes.
5. generating the smallest possible groupings of the pair-generated rules to cover all cases of the prefixes or suffixes.

The first step involves the separation of the prefix, stem, and suffix. At present, this is accomplished by looking for the biggest common segments of a group of phonetically spelled words. For example:

(GET-PREFIX-STEM-SUFFIX '(buk buks)) returns (NIL buk s)

The second step is the weeding out of exceptional prefixes or suffixes. This is done by simply stating that "x" amount or less of something designates it as being exceptional.

(SEPARATE-EXCEPTIONS '(s z z s e^n s z) 2) returns

((s z) e^n) where "2" is the cut-off point for the exceptions.

The third step is the conversion of the stems into distinctive features environments. This is done by starting from the front (or back depending on whether its prefixing or suffixing) and finding the minimal features from the DF table for all the phonemes in one given position. For example, if we took the last phoneme from a list of phonemes in the example below, the following minimal features would be generated (for English):

(NTH-ENVIRONMENT '(bus bv^s ru3 uz) -1) returns

(+obs +cont +cor)

1. deleting a phoneme
2. inserting a phoneme
3. changing a phoneme

The main functions which make up the fourth step are:

PAIR-POSSIBLE-PS-RULES,
COMPARE-SLOT, COMPARE-LIST-OF-SLOTS,
STAR-COMBINATION, STAR-COMBINATIONS

The fifth and final step involves grouping the pair-generated rules so that they generate all possible prefixes or suffixes. This is done by selecting one of the possible prefixes or suffixes as the "norm" and using the pair-generated rules to generate the prefixes or suffixes which are left. For example, if we pick the ending "s" to be the sound which is the "normal" ending for plurals we would have to pick pair-generated rules such that they generate the two other endings: "z" and "e^z". If we picked "z" to be the "norm" we would have to choose different rules.

The major functions for the fifth step are:

POSSIBLE-RULES and FIND-NEXT-RULE

TESTING THE PROGRAM

To test my program I gave it the only other interesting phonological rule in English: the preterates. The preterates are the past forms of verbs and are formed by adding one of the following three suffixes to a stem: "t", "d", or "e^d". Without knowing the rule, I entered the pairs of words below and let my program generate alternative rule forms for the PRETERATES. Like the PLURALS, it came up with four alternatives.

*PRETERATES

((b@' b@'d)	(bv^z bv^zd)	(c@ c@d)	(dIq dIgz)	(fil filed)
(fli flid)	(flow flowd)	(ga>lf ga>lft)	(glu glud)	(p@k p@kt)
(pes pest)	(pI@ pI@t)	(pIn pInd)	(pUS pUSt)	(pat pate^d)
(pr^ pr^d)	(rU3 rU3d)	(slv slvd)	(s@d s@de^d)	(sid side^d)
(sl@m sl@m d)	(st@b st@b d)	(tlp tlpt)	(w0'k w0'ks)	

*(POSSIBLE-REGULAR-AS-RULES PRETERATES)

(NIL

```
(((((t d ((+voi) NIL NIL NIL)))
)
((d e^d ((+obs -cont -high -tnse +cor)
(+syl -cons -loja -nas -str)
(+obs -high -back -tnse -voi)))
(((*Cl* t (+voi)))

(((d t ((+obs -back -voi)
(-obs +cont -tnse -cp -lab -nas +voi -str)
(-high -tnse -lat -rnd -loja -nas -str) NIL)) ((*Cl* d (-voi))
))
((d e^d ((+obs -cont -high -tnse +cor)
```

```

(+syl -cons -loja -nas -str)
(+obs -high -back -tnse -voi))) ((*l1* e^)))

(((e^d d ((+voi) NIL NIL NIL))
  ((d t ((+obs -back -voi)
    (-obs +cont -tnse -cp -lab -nas +voi -str)
    (-high -tnse -lat -rnd -loja -nas -str) NIL)) ((*Cl* d (-voi))
    )))

(((d e^d ((+obs -cont -high -tnse +cor)
  (+syl -cons -loja -nas -str)
  (+obs -high -back -tnse -voi))) ((*l1* e^)))
  ((d t ((+obs -back -voi)
    (-obs +cont -tnse -cp -lab -nas +voi -str)
    (-high -tnse -lat -rnd -loja -nas -str) NIL)) ((*Cl* d (-voi))
    )))

```

FUNCTIONS FOR REGULAR AFFIXING RULE GENERATION

(COMPARE-SLOT <LIST: list to be compared>)

This function compares two lists of slots and transforms them into either delete, insert, or change. These transformations are symbolized by the following:

DELETE -- *D1* (where "1" is slot number 1)
 INSERT -- *I2* (where "2" is slot number 2)
 CHANGE -- *C3* (where "3" is slot number 3)

INPUT REQUIREMENTS: CAR given-list -- phoneme environment
 CADR " -- first slot list
 CADDR " -- second slot list

The slot lists are generated from the function STAR-COMBINATION.

SAMPLE: (COMPARE-SLOT '((+obs +cont +cor) (*STAR* z) (e^ z)))
 returns ((+obs +cont +cor) ((*I1* e^)))

RETURNS: a LIST-OF-LISTS

(COMPARE-SLOTS <LIST-OF-LISTS: lists to be sent to COMPARE-SLOT>)

This function sends a list of correctly formatted lists to COMPARE-SLOT and returns the ones with the shortest list of actions (actions being delete, insert, or change).

SEE : COMPARE-SLOT

RETURNS: a LIST-OF-LISTS

(ENVIRONMENT-DIRECTION <LIST : word list >
 <+-NUMBER: direction of scan>)

This function scans a list of words left-to-right or right-to-left and returns the list of minimized distinctive features for each position. If the number is negative, the phonemes are scanned right-to-left. If the number is positive, the phonemes are scanned left-to-right. Once the "nth" phonemes of the words are gathered, they are converted to minimal distinctive features.

SEE: NTH-ENVIRONMENT

RETURNS: a LIST-OF-LISTS

(FILL-STAR-SLOTS <LIST : list to be examined >
 <+-NUMBER: length the list needs to be>)

This function returns the possible combinations of a list which needs slots to be added. The "dummy" slots are labeled as "*STAR*" slots.

SAMPLE: (FILL-STAR-SLOTS '(e^ z) 4) returns

((*STAR* *STAR* e^ z) (e^ z *STAR* *STAR*) (*STAR* e^ z *STAR*))

RETURNS: a LIST-OF-LISTS

(FIND-NEXT-RULE <LIST: unique list of phonemes tried>
<LIST: given rule >
<LIST: list of all rule to try >)

This complex recursive function finds the rules which generate the other affixes given a rule as the "norm". This routine is called by POSSIBLE-RULES.

(GET-PREFIX-STEM-SUFFIX <LIST: list of morphemes>)

This function separates a list of morphemes into its prefixes, stem, and suffixes. It does so by simple segment matching.

SAMPLE: (GET-PREFIX-STEM-SUFFIX '(falu fala fala~muS fala~)) returns
(NIL fal (u a a~muS a~))

RETURNS: a LIST

(GROUP-REGULAR-STEMS <LIST: words to be analyzed such as PLURALS>)

This function groups the regular stems by their prefixes or suffixes. The main functions it calls are STEM-EXCEPTIONS and GROUP-STEMS-BY-SEGMENTS.

(GROUP-STEMS-BY-SEGMENTS <LIST: stem list >
<LIST: segment list>
<LIST: word-list >)

This function groups stems by a list of segments; namely prefixes and suffixes. The main function it calls is ENVIRONMENT-DIRECTION.

(PAIR-POSSIBLE-REGULAR-PS-RULES <LIST: words to be examined>)

This function returns the smallest possible rules for describing one phoneme group changing to another. It returns the smallest logical rule combinations using minimal combinatorics.

(POSSIBLE-REGULAR-PS-RULES <LIST: words to be examined such as PLURALS>)

This function returns the smallest possible regular rules for prefixing and suffixing. The first half of the returned list is for the prefixes, the second for the suffixes.

(POSSIBLE-RULES <LIST: rules>)

This function derives the possible rules from a list of pair-generated rules. It assumes the first rule to be true and picks combinations of

the others to derive all possible affixes. It then picks the second rule to be true repeating the process until all rules have been chosen.

(SEPARATE-STEM-EXCEPTIONS <LIST: words to be examined such as PLURALS>)

This function separates the given words into their regular prefixes and suffixes. (SHORTEST-RULE-ACTION-LENGTH <LIST: rules>)

This function returns the length of the shortest rule actions in pair-generated rules.

(SHORTEST-RULE-ACTION <LIST: rules>)

This function returns the rule with the shortest number of rule actions. These actions include delete (*Dl*), insert (*Ii*), and change (*Ci*).

(STAR-COMBINATION <LIST: rule list 1>
<LIST: rule list 2>)

This function returns the rule lists filled with "*STAR*" slots ready to be converted into deletes, inserts, and changes. If the given rule lists do not have equal lengths, then this function returns all possible slot combinations.

(STAR-COMBINATIONS <LIST-OF-LISTS: list of rule pairs>)

This function sends a list of rule pairs to STAR-COMBINATION to fill in the slots.

DISTINCTIVE FEATURE TOOLS

(CONVERT-PHONEMES-TO-MATRIX <LIST: list of phonemes>)

This function converts the given list of phonemes to a special kind of features matrix. This matrix contains "+", "-", and "0". The third character "0" means that that feature is not specified and does not contribute to the feature distinctions.

SAMPLE: (CONVERT-PHONEMES-TO-MATRIX '(S 3 s z)) returns

(0 0 + + 0 0 0 0 + 0 0 0 0 0 0)

RETURNS: a LIST

(FEATURE-MATRIX <ATOM: feature>)

This function returns the boolean matrix associated with the given feature.

SAMPLE: (FEATURE-MATRIX 'tase) returns

(+ + + + - - - - + + + + - - - - + + + + + + + - - - - -

This trivial function returns the current list of features in the DF table.

SAMPLE: (LIST-FEATURES) returns

(syl cons obs cont high back tnse cp cor lat rnd lab loja
nas voi str)

RETURNS: a LIST

(LIST-PHONEMES <NIL>)

This trivial function returns the current list of phonemes in the DF table.

SAMPLE: (LIST-PHONEMES) returns

(i i' i~ i~/ I I' I~ I~/ u u' u~ u~/ U U' U~ U~/ e e' e~ e~/
o o' o~ o~/ E E' E~ E~/ 0 0' 0~ 0~/ a a' a~ a~/ a> a>' a>~
a>~/ a^ a^' a^~ a^~/ @ @' @~ @~/ e^ v^ e^~ v^~ b p d d^ t t^
g g> g< k k> k< v f z s /6 /0 /3 S x x> m ! m n ! n N N> ! N>
l * l l\ ! l\ /2 r^ ! r^ * r^ j w h ?)

RETURNS: a LIST

(MATRIX-COMPARE <LIST: matrix 1>
<LIST: matrix 2>)

This function takes two special matrices and compares them character by character. A special matrix consists of "+", "-", and "0". If the characters are the same, then the same character is returned. Else, "0" is returned.

SAMPLE: (MATRIX-COMPARE '(+ + + - - - 0 0 0)
'(+ - 0 + - 0 + - 0)) returns

(+ 0 0 0 - 0 0 0)

RETURNS: a LIST

(MINIMIZE-FEATURES <LIST: list of features and their boolean signs>)

This function uses the IMPLIED-TABLE to minimize the given features. Minimizing means that for any set of features, there is a subset of them which "captures" the entire set.

SAMPLE: (MINIMIZE-FEATURES '(+cons +obs -loja -lat)) returns
(+obs)

RETURNS: a LIST

(PHONEME-CHANGE <ATOM: phoneme 1>
<ATOM: phoneme 2>)

This function compares two given phonemes and returns those features

which differ. The feature sign which is returned with each feature is that of the second phoneme (in accordance with the idea of change).

SAMPLE: (PHONEME-CHANGE 'd 'd^) returns (+high +tnse)

RETURNS: a LIST

(PHONEME-DIFFERENCE <ATOM: phoneme 1>
<ATOM: phoneme 2>)

This function returns the feature names (without the signs) of those features which have opposite signs in the given phonemes.

SAMPLE: (PHONEME-DIFFERENCE 'd 'd^) returns (high tnse)

RETURNS: a LIST

(PHONEME-FEATURE-INTERSECTION <ATOM: phoneme >
<LIST: features>)

This function returns the common features between the given features and the phoneme generated features.

SAMPLE: (PHONEME-FEATURE-INTERSECTION 'b '(+cp -rnd +lab)) returns
(-rnd +lab)

RETURNS: a LIST

(PHONEME-FEATURE-INTERSECTIONS <ATOM : phoneme >
<LIST-OF-LISTS: a list of feature lists>)

This function returns the common features between the given features in the the given lists and the phoneme generated features.

SAMPLE: (PHONEME-FEATURE-INTERSECTIONS
'b
'((+cp -rnd +lab) (-rnd -loja +lab))) returns
(-rnd +lab)

RETURNS: a LIST

(PHONEME-IMPLIED-FEATURES <LIST: phonemes>)

This function converts a lis of phonemes into a minimal list of distinctive features.

SAMPLE: (PHONEME-IMPLIED-FEATURES '(S 3 s z)) returns
(+obs +cont +cor)

RETURNS: a LIST

(REWRITE-OF <LIST: new "untransposed" distinctive features table>)

This function rewrites the global variables DFMATRIX and DFTMATRIX using the "untranspose!" format.

(REWRITE-DFT <LIST: new "transposed" distinctive features table>)

This function rewrites the global variables DFTMATRIX and DFMATRIX using the "transposed" format.

(REWRITE-IMPLIED-TABLE <NIL>)

This function rewrites the global variable IMPLIED-TABLE by using the function DF-IMPLIED-FEATURES. Therefore, this process is very time consuming. (That's the whole point of having a table!)

***NOTE: IMPLIED-TABLE is not rewritten when the DF table has been altered in any way. Therefore, when the user is done fiddling with DF table altercations, he/she should invoke this function.

DISTINCTIVE FEATURES EDIT COMMANDS

(DISPLAY-DFMATRIX)

This function displays the current Distinctive Features table.

RETURNS: display of DF table

(ADD-FEATURE-BEFORE <ATOM: feature before >
<LIST: new feature and feature matrix>)

To add a feature to the DF table simply indicate which present feature you wish to place the new feature before and follow it by the complete new feature matrix.

SAMPLE: (ADD-FEATURE-BEFORE 'lab '(loja - - - - + ... - -))

RETURNS: "OK" --- action completed
"===NO-MATCH===" --- feature not matched

(ADD-PHONEME-BEFORE <ATOM: phoneme before >
<LIST: new phoneme and phoneme matrix>)

To add a ptheoneme to the DF table simply indicate which present phoneme you wish to place the new phoneme before and follow it by the complete new phoneme matrix.

SAMPLE: (ADD-PHONEME-BEFORE 'e^~ '(e^ + - - + ... + -))

RETURNS: "OK" --- action completed
"===NO-MATCH===" --- phoneme not matched

(CHANGE-FEATURE-NAME <ATOM: old feature>
<ATOM: new feature>)

This command allows one to change the name of a feature in the DF table.

SAMPLE: (CHANGE-FEATURE-NAME 'tnse 'tens)

RETURNS: "OK" --- action completed
"===NO-MATCH===" --- feature not matched

(CHANGE-PHONEME-NAME <ATOM: old phoneme>
<ATOM: new phoneme>)

This command allows one to change the name of a phoneme in the DF table.

SAMPLE: (CHANGE-FEATURE-NAME 'tnse 'tens)

RETURNS: "OK" --- action completed
"===NO-MATCH===" --- phoneme not matched

(REPLACE-FEATURE-MATRIX <ATOM: feature to replace>
<LIST: new feature matrix>)

To replace an existing feature matrix simply indicate the name of the feature to replace followed by the complete feature matrix.

SAMPLE: (REPLACE-FEATURE-MATRIX 'lab '(lab - - + - ... - -))

RETURNS: "OK" --- action completed
"===NO-MATCH===" --- feature not matched

(REPLACE-PHONEME-MATRIX <ATOM: phoneme to replace>
<LIST: new phoneme matrix>)

To replace an existing phoneme matrix simply indicate the name of the phoneme to replace followed by the complete phoneme matrix.

SAMPLE: (REPLACE-PHONEME-MATRIX 'lab '(lab - - + - ... - -))

RETURNS: "OK" --- action completed
"===NO-MATCH===" --- phoneme not matched

(CHANGE-BOOLEAN <ATOM: phoneme name>
<ATOM: feature name>
<ATOM: "+" or "-" >)

To change an individual boolean character in the DF table, give the phoneme name, feature name, and either a "+" or a "-".

SAMPLE: (CHANGE-BOOLEAN 'b 'rnd '-')

RETURNS: "OK" --- action completed
"===NO-MATCH===" --- phoneme or feature not matched

(FEATURE-ORDER <LIST: list of ALL the features in desired order>)

To re-order the features in the DF table one must list ALL the features in the desired order.

SAMPLE: (FEATURE-ORDER '(syl cons obs ... voi str))

RETURNS: "OK" --- action completed
"===NO-MATCH===" --- feature not matched

(PHONEME-ORDER <LIST: list of ALL the phonemes in desired order>)

To re-order the phonemes in the DF table one must list ALL the phonemes in the desired order.

SAMPLE: (PHONEME-ORDER '(i I u U e ... r^ j w ?))

RETURNS: "OK" --- action completed
"===NO-MATCH===" --- phoneme not matched

DISTINCTIVE FEATURES RELATED FUNCTIONS

(DISPLAY-MATRIX <LIST-OF-LISTS: given matrix>)

This function displays a given matrix which is a list in the following form:

```
[ (matrix-title heading-1 heading-2 ... heading-N)
  (row-title-1 boolean-1 boolean-2 ... boolean-N)
  (row-title-2 boolean-1 boolean-2 ... boolean-N)
  :
  :
  (row-title-M boolean-1 boolean-2 ... boolean-N) ]
```

This function is specifically made to display boolean matrices. The headings are printed out vertically with a special row for one sub-character. The row-title however, is limited to 4 characters.

RETURNS: a display of the matrix

(SPLIT-MATRIX <LIST-OF-LISTS: matrix to be split>
 <NUMBER : split point >)

This function splits up a the DF matrix when the number of boolean characters is greater than the "split point" NUMBER. The headings and row-titles are correctly rewritten in both halves so that they can be immediately displayed.

RETURNS: the two halves of the matrix in a list

(ANY-IN <LIST: are there any elements of this list>
<LIST: in this list? >)

If any element of the first list is in the second, this predicate function returns "T". Else NIL.

SAMPLE: (ANY-IN '(A B C D E) '(F G H D I)) returns

RETURNS: T or NIL

(APUSH <LIST: list to be altered >
<ATOM: atom to be PUSHed onto list>)

This macro is identical to the following:

(SETQ list (APPEND atom list))

RETURNS: a the new LIST

(BREAK-INTO-SEGMENTS <ATOM: word or word segment>)

This function breaks a word or word segment into a list of its possible segments.

SAMPLE: (BREAK-INTO-SEGMENTS 'puxa) returns

(puxa pux uxa pu ux xa p u x a)

RETURNS: a LIST

(COMBINATION-CDR <LIST: list of elements for combination>)

This recursive function returns all the possible arrangements of the elements in the CDR of the given list.

SAMPLE: (COMBINATION-CDR '(A B C D)) returns

((A B C D) (A C D B) (A D B C) (A B D C) (A D C B) (A C B D))

RETURNS: a LIST-F-LISTS

(COMBINATION-ORDER <LIST: list of elements for combination>)

This function uses COMBINATION-CDR to produce all possible arrangements of all elements.

SAMPLE: (COMBINATION-ORDER '(A B C)) returns

((A B C) (B C A) (C A B) (A C B) (C B A) (B A C))

RETURNS: a LIST-OF-LISTS

(COMBINATION-PAIR <LIST: list of elements for combination>)

This function returns all NON-ORDERED combination of PAIRS of the elements.

SAMPLE: (COMBINATION-PAIR '(A B C D)) returns
((A B) (A C) (A D) (B C) (B D) (C D))

RETURNS: a LIST-OF-LISTS

(COMBINE <ATOM: atom to be combined>
<LIST: list of "combinees">)

COMBINE combines the given ATOM with each element of the given LIST.

SAMPLE: (COMBINE 'A '(B C D E)) returns
((A B) (A C) (A D) (A E))

RETURNS: a LIST-OF-LISTS

(COMBINE-APPEND <ATOM: atom to be combined>
<LIST: list of "combinees">)

COMBINE-APPEND combines the given ATOM by APPENDING it with each element of the given LIST.

SAMPLE: (COMBINE 'A '((B C) (D E F) (G H I J))) returns
((A B C) (A D E F) (A G H I J))

RETURNS: a LIST

(COMMON-ELEMENT <LIST: list #1>
<LIST: list #2>)

This function returns all the common elements of the two lists.

SAMPLE: (COMMON-ELEMENT '(A B C) '(B C D E)) returns
(B C)

RETURNS: a LIST

(COMMON-ELEMENTS <LIST-OF-LISTS>)

This function returns the common elements of all the lists in the given list.

SAMPLE: (COMMON-ELEMENTS '((A B C D) (B C D E) (C D E))) returns
(C D)

(COMMON-SEGMENTS <LIST: list of words>)

This function returns the common segments of the given words. Note that this does not return the UNIQUE segments because it is sometimes useful to know the number of occurrences of a given common segment.

SAMPLE: (COMMON-SEGMENTS '(Mississippi sip)) returns

(sip si si ip i s s i s s i p p i)

RETURNS: a LIST

(FIRST-N-ELEMENTS <LIST: list to be looked at>
<+NUMBER: nth element >)

This function can return the nth element either from the front to the back. The number must always be positive.

SAMPLE: (FIRST-N-ELEMENTS '(A B C D E) 3) returns

(A B C)

RETURNS: a LIST

(IN <ATOM: item we are looking for>
<LIST: where we are looking >)

IN is a predicate function which returns "T" if the item is in the list and NIL if it is not.

SAMPLE: (IN 'C '(A B C D E)) returns T

RETURNS: T or NIL

(MAXIMUM-CHARACTER-LENGTH <LIST: list of words>)

This function returns the length of the longest word in the given list.

SAMPLE: (MAXIMUM-CHARACTER-LENGTH '(tudo bem)) returns 4

RETURNS: a NUMBER

(MAXIMUM-SPECIAL-WORD-LENGTH <LIST: list of words>)

This function returns the length of the longest word in the given list but uses the SPECIAL-EXPLODE instead of the regular EXPLODE.

SAMPLE: (MAXIMUM-WORD-LENGTH '(bv^'g bv^'gz)) returns 4

RETURNS: a NUMBER

(MAXLIST <LIST: of numbers>)

MAXLIST simply returns the maximum value of a list of numbers.

SAMPLE: (MAXLIST '(1 4 3 5 2 4 3)) returns 5

RETURNS: a NUMBER

(MEMBERS <LIST: of items to look for>
<LIST: list to be searched >)

This predicate function searches a list for a list of items.

SAMPLE: (MEMBERS '(A B C) '(X Y Z A B C D)) returns T

RETURNS: T or NIL

(MINLIST <LIST: of numbers>)

MINLIST simply returns the minimum value of a list of numbers.

SAMPLE: (MINLIST '(3 4 3 1 2 4 3)) returns 1

RETURNS: a NUMBER

(MOVE-NTH-TO-FRONT <LIST: list of elements >
<+-NUMBER: nth element to be moved>)

This function moves the nth element to the front of the list. If the number is positive, the nth element from the left is moved. If negative, the nth element from the right is moved.

SAMPLE: (MOVE-NTH-TO-FRONT '(A B C D E) 3) returns
(C A B D E)

or (MOVE-NTH-TO-FRONT '(A B C D E) -2) returns
(D A B C E)

RETURNS: a LIST

(MOVE-TO-FRONT <ATOM: element to be moved>
<LIST: list of elements >)

This function moves the given element to the front of the list.

SAMPLE: (MOVE-TO-FRONT 'C '(A B C D E)) returns
(C A B D E)

RETURNS: a LIST

(NTH-ELEMENT <LIST: a list of elements>
<+-NUMBER: nth element >)

This function gets the nth element of a given list. If the number is positive, the nth element from the left is obtained.

If the number is negative, the nth element from the right is obtained.

SAMPLE: (NTH-ELEMENT '(A B C D E) 2) returns B

or (NTH-ELEMENT '(A B C D E) -2) returns D

RETURNS: an element

(NTHCHAR-LIST <LIST: a list of words >
<+-NUMBER: nth character>

NTHCHAR-LIST is NTHCHAR applied to a list of arguments.

SAMPLE: (NTHCHAR-LIST '(como vai voce) 2) returns

(o a o)

or (NTHCHAR-LIST '(como vai voce) -2) returns

(m a c)

RETURNS: a LIST

(NUMBER-OF-OCCURENCES <ATOM: item to be counted >
<LIST: list to be searched>)

This function counts the number of a specified item in a given list.

SAMPLE: (NUMBER-OF-OCCURENCES 'A '(A B D A F A G)) returns 3

RETURNS: a NUMBER

(REMOVE-CHARACTER <ATOM: character to be removed>
<LIST: list of words >)

This function removes a character from a list of words.

SAMPLE: (REMOVE-CHARACTER 'a '(faith and toast)) returns

(fith nd tost)

RETURNS: a LIST

(REMOVE-CHARACTERS <LIST: characters to be removed>
<LIST: list of words >)

This function removes a list of characters from a list of words.

SAMPLE: (REMOVE-CHARACTERS '(a t) '(faith and toast)) returns

(fih nd os)

RETURNS: a LIST

(REMOVE-FIRST-OCCURENCE <ATOM: item to be removed >
<LIST: list to be searched>)

This function removes the first occurrence of a specified item in a given list.

SAMPLE: (REMOVE-FIRST-OCCURENCE 'A '(U I A O A E)) returns
(U I O A E)

RETURNS: a LIST

(REMOVES <LIST: list of elements to be removed>
<LIST: list to be searched >)

This removes all occurrences of the elements in the element list from the given list.

SAMPLE: (REMOVES '(A B) '(A B D A O N B E)) returns (D O N E)

RETURNS: a LIST

(REPLACE <ATOM: old item >
<ATOM: new item >
<LIST: list to be searched>)

To replace an item in a list, give this function the old name of the atom, the new name, and the list in which it is to be changed.

SAMPLE: (REPLACE 'A 'O '(D A N E)) returns (D O N E)

RETURNS: a LIST

(REPLACES <ATOM: old item >
<ATOM: new item >
<LIST-OF-LISTS: lists to be searched>)

To replace an item in a list of lists, give use this function the old name of the atom, the new name, and the list of lists in which it is to be changed.

SAMPLE: (REPLACES 'A 'O '((D A N E) (F A R) (N A W)) returns
((D O N E) (F O R) (N O W))

RETURNS: a LIST OF LISTS

(REPLICATE <ATOM: item to be replicated>
<NUMBER: number of replicas >)

This function creates a list of items which the user specifies.

SAMPLE: (REPLICATE '*STAR* 3) returns (*STAR* *STAR* *STAR*)

RETURNS: a LIST

(REVERSES <LIST-OF-LISTS: list of lists to be reversed>)

This functions reverses a list of lists.

SAMPLE: (REVERSES '((A B C) (D E) (F))) returns ((C B A) (E D) (F))

RETURNS: a LIST-OF-LISTS

(ROTATE-LEFT <LIST: list to be rotated>)

This function rotates the elements in the given list left.

SAMPLE: (ROTATE-LEFT '(A B C D E)) returns (B C D E A)

RETURNS: a LIST

(ROTATE-LIST <LIST: list to be completely rotated>)

This function rotates a list to left until it returns to its original form.

SAMPLE: (ROTATE-LIST '(A B C)) returns ((A B C) (B C A) (C A B))

RETURNS: a LIST-OF-LISTS

(ROTATE-RIGHT <LIST: list to be rotated>)

This function rotates the elements in the given list right.

SAMPLE: (ROTATE-RIGHT '(A B C D E)) returns (B C D E A)

RETURNS: a LIST

(RREMOVE <ATOM: element to be removed>
<BOUND-ATOM: list to be altered >)

This macro is identical to REMOVE except for the fact that the list given is permanently altered.

SAMPLE: (SETQ L '(A B C D))

(RREMOVE 'A L) returns (B C D)

L is now bound to (B C D)

RETURNS: a LIST

(RRETURN <BOUND-ATOM: name of list to be reverse-returned>)

This macro is identical to the RETURN in a PROG except that it reverses the list before it returns it.

SAMPLE: (RRETURN NEW-LIST) is identical to

(RETURN (REVERSE NEW-LIST)) in a PROG

RETURNS: a LIST

(SEPARATE-EXCEPTIONS <LIST: list of elements >
<NUMBER: number which constitutes an exception>)

This function separates the elements of a list into "regular" and "exceptional" elements. The "regular" elements are those elements which occur more than the specified number for an exception.

SAMPLE: (SEPARATE-EXCEPTIONS '(s z s e^z s z z)) returns
((s z) e^z)

RETURNS: a LIST: CAR -- the LIST of "regular" elements
CDR -- the LIST of the "exceptions"

(SHORTEST-CDR-LISTS <LIST-OF-LISTS: given lists to be examined>)

This function returns the lists which have the shortest CDRs.

SAMPLE: (SHORTEST-CDR-LISTS '((A B C) (A D) (A B) (D E F))) returns
((A D) (A B))

RETURNS: a LIST-OF-LISTS

(SHORTEST-LIST-LENGTH <LIST-OF-LISTS: given lists to be examined>)

This function returns the length of the shortest list length of the given lists.

SAMPLE: (SHORTEST-LIST-LENGTH '((A B C) (A B) (A B C D))) returns 2
RETURNS: a NUMBER

(SPECIAL-EXPLODE <ATOM: word to explode>)

This function is used to explode words which are spelled using the phonetic notation in the DF table. This is needed because there is often more than one ASCII character to represent a phoneme.

SAMPLE: (SPECIAL-EXPLODE '*r^a~'p) returns (*r^ a~' p)
RETURNS: a LIST

(SPECIAL-NTHCHAR <ATOM: word to be searched >
<+-NUMBER: ,nth special character to be returned>)

This function is used to get the nth character of a special word. (SPECIAL meaning that it uses SPECIAL-EXPLODE for phonetically spelled words).

SAMPLE: (SPECIAL-NTHCHAR '(*r^a~'p 2) returns a~'

RETURNS: an ATOM

(SPECIAL-NTHCHAR-LIST <LIST: words to be searched >
<+-NUMBER: nth special character to be returned>)

This function is used to get the nth character of a list of special words. (SPECIAL meaning that it uses SPECIAL-EXPLODE for phonetically spelled words).

SAMPLE: (SPECIAL-NTHCHAR '(*r^es da~'p ok) 2) returns (e a~' o)

RETURNS: an LIST

(SPLICE <ATOM: old item >
<LISTM: list to be spliced in place of the old item>
<LIST: list to be searched >)

To splice a list into a list, give this function the old name of the atom, a list ot be spliced, and the list into which it is to be spliced.

SAMPLE: (SPLICE 'A '(O N) '(D A E)) returns (D O N E)

RETURNS: a LIST

(SPLICES <ATOM : old item >
<LIST : list to be spliced in place of the old item>
<LIST-OF-LISTS: list to be searched >)

To splice a list into a list of lists, give this function the old name of the atom, a list ot be spliced, and the list of lists into which it is to be spliced.

SAMPLE: (SPLICE 'A '(O N) '((W A) (T A))) returns ((W O N) (T O N))

RETURNS: a LIST-OF-LISTS

(SPLIT-LIST <LIST : list to be split>
<+-NUMBER: splitting point >)

To split a list in two, simply send this function the list to be split, and the number of the element which is to be the CAR of the second half.

SAMPLE: (SPLIT-LIST '(A B C D E) 3) returns ((A B) (C D E))

RETURNS: a LIST: CAR -- first half
CADR -- second half

(SQUASH <ATOM: front atom to be squashed>
<ATOM: back atom to be squashed >)

This function squashes two atoms into one. If one of the atoms is NIL,

then the other atom returned.

SAMPLE: (SQUASH 'con 'pound) returns "compound"

RETURNS: an ATOM

(SQUASH-LIST <LIST: list of atoms to be squashed>)

This function squashes a list of atoms into one. If one of the atoms is NIL, then the other atoms are squashed.

SAMPLE: (SQUASH '(M iss iss ipp i)) returns "Mississippi"

RETURNS: an ATOM

(STRING-SUBSET <ATOM: subset to be searched for>
<ATOM: word to be searched >)

This function searches a word for a given subset and divides the word into three pieces: segments before the subset, the subset itself, and segments after the segment.

SAMPLE: (STRING-SUBSET 'guard 'irregardless) returns
(irre guard less)

RETURNS: a LIST: 1st element -- segment before
2nd element -- subset being searched for
3rd element -- segment after

(STRIP-ONE-LEVEL <LIST: list to be stripped>)

This function strips one set of parentheses from every element in the list.

SAMPLE: (STRIP-ONE-LEVEL '((A B C) (D (E F)))) returns
(A B C D (E F))

RETURNS: a LIST

(TALLY-ELEMENTS <LIST: list of elements to be tallied>)

This function tallies all the elements in a list and returns the number of occurrences of each element.

SAMPLE: (TALLY-ELEMENTS '(A B C A D A B E D)) returns
((A 3) (B 2) (C 1) (D 2) (E 1))

RETURNS: a LIST

(TRANPOSE <LIST-OF-LISTS: a list of lists to be transposed>)

This function transposes a list of lists much like a matrix is

transposed in math. Another way of thinking about it is reversing the rows and columns of a table. If there are any blanks (or NILs), they are removed.

SAMPLE: (TRANPOSE '((T S S D) (R P E) (A O) (N))) returns
((T R A N) (S P O) (S E) (D))

RETURNS: a LIST-OF-LISTS

(UNIQUE-ELEMENTS <LIST: a list to be searched for unique elements>)

This function returns a list of the unique elements of the given list.

SAMPLE: (UNIQUE-ELEMENTS '(U N U N I U Q I U E)) returns
(U N I Q U E)

RETURNS: a LIST

```

(DE ADD-FEATURE-BEFORE (BEFORE NEW-FEATURE)
  (PROG (NEW-MATRIX FEATURE-LIST)
    (COND [(NOT (IN BEFORE (LIST-FEATURES))) (RETURN '===NO-MATCH===)])
    (SETQ NEW-MATRIX (LIST (CAR DFMATRIX)))
    (SETQ FEATURE-LIST (CDR DFMATRIX))
  NEXT-FEATURE
    (COND [(EQUAL BEFORE (CAAR FEATURE-LIST)) (PUSH NEW-MATRIX NEW-FEATURE)])
    (COND [(NULL FEATURE-LIST)
      (REWRITE-DEF (REVERSE NEW-MATRIX))
      (RETURN 'OK)])
    (PUSH NEW-MATRIX (CAR FEATURE-LIST))
    (POP FEATURE-LIST)
    (GO NEXT-FEATURE)))

```

```

(DE ADD-PHONEME-BEFORE (BEFORE NEW-PHONEME)
  (PROG (NEW-MATRIX PHONEME-LIST)
    (COND [(NOT (IN BEFORE (LIST-PHONEMES))) (RETURN '===NO-MATCH===)])
    (SETQ NEW-MATRIX (LIST (CAR DFMATRIX)))
    (SETQ PHONEME-LIST (CDR DFMATRIX))
  NEXT-PHONEME
    (COND [(EQUAL BEFORE (CAAR PHONEME-LIST)) (PUSH NEW-MATRIX NEW-PHONEME)])
    (COND [(NULL PHONEME-LIST)
      (REWRITE-DEF (REVERSE NEW-MATRIX))
      (RETURN 'OK)])
    (PUSH NEW-MATRIX (CAR PHONEME-LIST))
    (POP PHONEME-LIST)
    (GO NEXT-PHONEME)))

```

```

(DE ANY-IN (LIST-1 LIST-2)
  (PROG NIL
  NEXT-ELEMENT
    (COND [(IN (CAR LIST-1) LIST-2) (RETURN T)]
      [(NULL LIST-1) (RETURN NIL)])
    (POP LIST-1)
    (GO NEXT-ELEMENT)))

```

```

(OM APUSH (BODY)
  (LIST 'SETQ (CADR BODY) (LIST 'APPEND (CADDR BODY) (CADR BODY))))

```

```

(DV BOUNDRIES (% + $ =))

```

```

(DE CHANGE-BOOLEAN (PHONEME FEATURE BOOLEAN)
  (PROG (BOOLEAN-MATRIX FEATURE-LIST NEW-MATRIX)
    (SETQ BOOLEAN-MATRIX (PHONEME-MATRIX PHONEME))
    (SETQ FEATURE-LIST (LIST-FEATURES))
  NEXT-BOOLEAN
    (COND [(NULL BOOLEAN-MATRIX)
      (COND [(NULL NEW-MATRIX) (RETURN '===NO-MATCH===)]
        [T (REPLACE-PHONEME-MATRIX
          PHONEME
          (CONS PHONEME (REVERSE NEW-MATRIX)))
          (RETURN 'OK)])]
      [(EQUAL FEATURE (CAR FEATURE-LIST)) (PUSH NEW-MATRIX BOOLEAN)]
      [T (PUSH NEW-MATRIX (CAR BOOLEAN-MATRIX))])
    (POP BOOLEAN-MATRIX)
    (POP FEATURE-LIST)
    (GO NEXT-BOOLEAN)))

```

```

(DE CHANGE-FEATURE-NAME (OLD-FEATURE NEW-FEATURE)

```



```

(COND [(IN OLD-FEATURE (LIST-FEATURES))
      (REPLACE-FEATURE-MATRIX
        OLD-FEATURE
        (CONS NEW-FEATURE (FEATURE-MATRIX OLD-FEATURE)))])
[IT '===NO-MATCH===]])

(DE CHANGE-PHONEME-NAME (OLD-PHONEME NEW-PHONEME)
  (COND [(IN OLD-PHONEME (LIST-PHONEMES))
        (REPLACE-PHONEME-MATRIX
          OLD-PHONEME
          (CONS NEW-PHONEME (PHONEME-MATRIX OLD-PHONEME)))])
  [IT '===NO-MATCH===]])

(DE COMBINATION-CDR (GIVEN-LIST)
  (COND [(NULL (CDDR GIVEN-LIST)) (LIST GIVEN-LIST)]
    [IT (COMBINE-APPEND
          (LIST (CAR GIVEN-LIST))
          (STRIP-ONE-LEVEL
            (MAPCAR '(LAMBDA (A-LIST) (ROTATE-LIST A-LIST))
              (COMBINATION-CDR (CDR GIVEN-LIST))))))])])

(DE COMBINATION-ORDER (GIVEN-LIST)
  (MAPCAR 'CDR (COMBINATION-CDR (CONS '* GIVEN-LIST))))

(DE COMBINATION-PAIR (GIVEN-LIST)
  (PROG (PAIR-LIST NEW-LIST)
    NEXT-GROUP
    (COND [(NULL (CDR GIVEN-LIST)) (RRETURN NEW-LIST)])
    (SETQ PAIR-LIST (CDR GIVEN-LIST))
    (APUSH NEW-LIST (REVERSE (COMBINE (CAR GIVEN-LIST) (CDR GIVEN-LIST))))
    (POP GIVEN-LIST)
    (GO NEXT-GROUP)))

(DE COMBINE (ROOT GIVEN-LIST)
  (MAPCAR '(LAMBDA (ELEMENT) (LIST ROOT ELEMENT)) GIVEN-LIST))

(DE COMBINE-APPEND (ROOT GIVEN-LIST)
  (MAPCAR '(LAMBDA (ELEMENT) (APPEND ROOT ELEMENT)) GIVEN-LIST))

(DE COMMON-ELEMENT (LIST1 LIST2)
  (PROG (NEW-LIST)
    NEXT-ELEMENT
    (COND [(NULL LIST1) (RRETURN NEW-LIST)]
      [(IN (CAR LIST1) LIST2) (PUSH NEW-LIST (CAR LIST1))])
    (POP LIST1)
    (GO NEXT-ELEMENT)))

(DE COMMON-ELEMENTS (GIVEN-LIST)
  (PROG (PREVIOUS-LIST)
    (COND [(NULL GIVEN-LIST) (RETURN NIL)])
    (SETQ PREVIOUS-LIST (CAR GIVEN-LIST))
    NEXT-LIST
    (POP GIVEN-LIST)
    (COND [(NULL GIVEN-LIST) (RRETURN PREVIOUS-LIST)])
    (SETQ PREVIOUS-LIST (COMMON-ELEMENT PREVIOUS-LIST (CAR GIVEN-LIST)))
    (GO NEXT-LIST)))

(DE COMMON-SEGMENTS (WORD-LIST)
  (PROG (SEGMENT-LIST)
    (SETQ SEGMENT-LIST (BREAK-INTO-SEGMENTS (CAR WORD-LIST))))

```

NEXT-WORD

```
(POP WORD-LIST)
(COND [(NULL WORD-LIST) (RETURN SEGMENT-LIST)])
(SETQ SEGMENT-LIST
  (COMMON-ELEMENT
   SEGMENT-LIST
   (BREAK-INTO-SEGMENTS (CAR WORD-LIST))))
(GO NEXT-WORD)))
```

(DE COMPARE-LIST-OF-SLOTS (GIVEN-LIST)

```
(STRIP-ONE-LEVEL
 (STRIP-ONE-LEVEL
  (MAPCAR '(LAMBDA (A-LIST) (COMPARE-SLOTS A-LIST)) GIVEN-LIST))))
```

(DE CONVERT-PHONEMES-TO-MATRIX (PHONEME-LIST)

```
(PROG (NEW-MATRIX NEXT-MATRIX)
  (SETQ NEW-MATRIX (PHONEME-MATRIX (CAR PHONEME-LIST)))
```

NEXT-COMPARE

```
(POP PHONEME-LIST)
(COND [(NULL PHONEME-LIST) (RETURN NEW-MATRIX)])
(SETQ NEXT-MATRIX (PHONEME-MATRIX (CAR PHONEME-LIST)))
(SETQ NEW-MATRIX (MATRIX-COMPARE NEW-MATRIX NEXT-MATRIX))
(GO NEXT-COMPARE)))
```

(DE COMPARE-SLOT (GIVEN-LIST)

```
(PROG (NEW-LIST CAR-1 CAR-2 SLOT-COUNT SLOT-LIST-1 SLOT-LIST-2 ENVIRONMENT)
  (SETQ ENVIRONMENT (CAR GIVEN-LIST))
  (SETQ SLOT-LIST-1 (CADR GIVEN-LIST))
  (SETQ SLOT-LIST-2 (CADDR GIVEN-LIST))
  (SETQ SLOT-COUNT 1.)
```

NEXT-SLOT

```
(SETQ CAR-1 (CAR SLOT-LIST-1))
(SETQ CAR-2 (CAR SLOT-LIST-2))
(COND [(NULL SLOT-LIST-1) (RETURN (LIST NEW-LIST ENVIRONMENT))]
  [(EQUAL CAR-2 '*STAR*)
   (PUSH NEW-LIST
    (LIST (SQUASH-LIST (LIST '*I SLOT-COUNT '*)) CAR-1))]
  [(EQUAL CAR-1 '*STAR*)
   (PUSH NEW-LIST
    (LIST (SQUASH-LIST (LIST '*D SLOT-COUNT '*)) CAR-2))]
  [(NOT (EQUAL CAR-1 CAR-2))
   (PUSH NEW-LIST
    (LIST (SQUASH-LIST (LIST '*C SLOT-COUNT '*))
           CAR-2
           (PHONEME-CHANGE CAR-2 CAR-1))))])
```

```
(INCR SLOT-COUNT)
(POP SLOT-LIST-1)
(POP SLOT-LIST-2)
(GO NEXT-SLOT)))
```

(DE COMPARE-SLOTS (LIST-OF-LISTS)

(WITH (RULES)

```
(SETQ RULES
  (SHORTEST-CDR-LISTS
   (MAPCAR '(LAMBDA (A-LIST) (COMPARE-SLOT A-LIST)) LIST-OF-LISTS)))
(MAPCAR '(LAMBDA (A-LIST)
  (COMBINE (CAR A-LIST) (COMBINATION-ORDER (CADR A-LIST))))
  RULES)))
```

(DE DELETE-FEATURE (FEATURE)

```

(PROG (NEW-MATRIX FEATURE-LIST)
  (SETQ FEATURE-LIST (LIST-FEATURES))
  (COND [(NOT (IN FEATURE FEATURE-LIST)) (RETURN '===NO-MATCH===)])
  (SETQ NEW-MATRIX (LIST (CAR DFMATRIX)))
NEXT-FEATURE
  (COND [(NULL FEATURE-LIST)
    (REWRITE-DF (REVERSE NEW-MATRIX))
    (RETURN 'OK)]
    [(NOT (EQUAL FEATURE (CAR FEATURE-LIST)))
    (PUSH NEW-MATRIX (ASSOC (CAR FEATURE-LIST) DFMATRIX))])
  (POP FEATURE-LIST)
  (GO NEXT-FEATURE)))

```

```

(DE DELETE-PHONEME (PHONEME)
  (PROG (NEW-MATRIX PHONEME-LIST)
    (SETQ PHONEME-LIST (LIST-PHONEMES))
    (COND [(NOT (IN PHONEME PHONEME-LIST)) (RETURN '===NO-MATCH===)])
    (SETQ NEW-MATRIX (LIST (CAR DFMATRIX)))
NEXT-PHONEME
  (COND [(NULL PHONEME-LIST)
    (REWRITE-DF (REVERSE NEW-MATRIX))
    (RETURN 'OK)]
    [(NOT (EQUAL PHONEME (CAR PHONEME-LIST)))
    (PUSH NEW-MATRIX (ASSOC (CAR PHONEME-LIST) DFMATRIX))])
  (POP PHONEME-LIST)
  (GO NEXT-PHONEME)))

```

```

(DV DFMATRIX ((ENGLISH syl cons obs cont high back tnse cp cor lat rnd lab
  loja nas voi str) (i + - - + + - + - - - - - + -)
  (i' + - - + + - + - - - - - + +) (i~ + - - + + - + - - -
  - - - + + -) (i~' + - - + + - + - - - - - + + +)
  (I + - - + + - - - - - - - + -) (I' + - - + + - - - - -
  - - - + +) (I~ + - - + + - - - - - - - + + -) (I~' + - - +
  + - - - - - - + + +) (u + - - + + + + - - - + -)
  (u' + - - + + + + - - - + -) (u~ + - - + + + + - - -
  + - - + + -) (u~' + - - + + + + - - - + - + + +)
  (U + - - + + + - - - - + - - - + -) (U' + - - + + + - - - -
  - - - + +) (U~ + - - + + + - - - - + - + + -) (U~' + - - +
  + - - - - + - - + + +) (e + - - + - - + - - - - - + -)
  (e' + - - + - - + - - - - - + +) (e~ + - - + - - + - - -
  - - - + + -) (e~' + - - + - - + - - - - - + + +)
  (o + - - + - + + - - - + - - - + -) (o' + - - + - + + - - -
  - - - + +) (o~ + - - + - + + - - - + - - + + -) (o~' + - - +
  - + + - - - + - - + + +) (E + - - + - - - - - - - - + -)
  (E' + - - + - - - - - - - - + +) (E~ + - - + - - - - - -
  - - - + + -) (E~' + - - + - - - - - - - - + + +)
  (O + - - + - + + - - - + - - - + -) (O' + - - + - + + - - -
  - - - + +) (O~ + - - + - + + - - - + - - + + -) (O~' + - - +
  - + - - - - + - - + + +) (a + - - + - - - - - - + -)
  (a' + - - + - - - - - - - - + +) (a~ + - - + - - - - - -
  - - - + + -) (a~' + - - + - - - - - - - - + + +)
  (a> + - - + - + - - - - - - + -) (a>' + - - + - + - - - -
  - - - + +) (a>~ + - - + - + - - - - - - + -)
  (a>~' + - - + - + - - - - - - + +) (a^ + - - + - + - - - -
  - + - - - + -) (a^' + - - + - + - - - - - - + +)
  (a^~ + - - + - + - - - - - - + + -) (a^~' + - - + - + - - - -
  - + - - - + +) (@ + - - + - - - - - - - - + -)
  (@' + - - + - - - - - - - - + +) (@~ + - - + - - - - - -
  - - - + + -) (@~' + - - + - - - - - - - - + + +)
  (e^ + - - + - + - - - - - - + -) (v^ + - - + - + - - - -

```



```

- - - - - + + + + - - - - - + + - - + + - - - -
+ + + - - + + - - + - - - - - - - -) (lat - - - - -
- - - - - - - - - - - - - - - - - - - - - - - -
- - - - - - - - - - - + + + + - - - - - - - - -) (rad - - - -
- - - - + + + + + + + + - - - - + + + + - - - - + + + + - -
- - - - - + + + + - - - - - - - - - - - - - - - + - - +
- (lab - - - - - - - - - - - - - - - - - - - - + - -)
- - - - - - - - - - - - - - - - - - - - + + - - - -
- - - - - + + - - - - - - - - + + - - - - - - - -
- - - - -) (loja - - - - - - - - - - - - - - - - - -
- - - - - - - - - - - - - - - - - - - - + + + + - - - -
- - - - - - - - - - - - - - -) (nas - - + + - - + + - - + + - - + +
- - + + - - + + - - + + - - + + - - + + - - + + - - + + - -
+ + - - + + - - - - - - - - - - - - - - - - - - - - + +
+ + + + + - - - - - - - - - - - - - - - - - - - -) (voi + + + + + + + + + +
+ + + + + + + + + + + + + + + + + + + + + + + + + + + +
+ + + + + + + + + + + + + + + + + + + - - + - + - + -
+ - - - + + + + + + + + - + + + + + - + + - -) (str - + - +
- + - + - + - + - + - + - + - + - + - + - + - + - +
- + - + - + - + - + - + - + - + - + - + - + - + - +
- - - - - - - - - - - - - - - - - - - - - - - - -) )))

```

(DE DISPLAY-DFMATRIX NIL

```

(PROG (DFMATRIX-MINUS-TITLE TWO-HALVES BOOLEAN-LIST SPLIT-POINT)
  (SETQ DFMATRIX-MINUS-TITLE (CONS (CDAR DFMATRIX) (CDR DFMATRIX)))
  (TERPRI)

```

```

  (PRINC (CAAR DFMATRIX))
  (COND [(LESSP (LENGTH (LIST-PHONEMES)) 66.)
    (DISPLAY-MATRIX DFMATRIX-MINUS-TITLE)
    (RETURN T)])

```

```

  (SETQ SPLIT-POINT 0.)
  (SETQ BOOLEAN-LIST (FEATURE-MATRIX 'syl))

```

NEXT-BOOLEAN

```

  (COND [(NOT (EQUAL (CAR BOOLEAN-LIST) '-))
    (POP BOOLEAN-LIST)
    (INCR SPLIT-POINT)
    (GO NEXT-BOOLEAN)])
  (SETQ TWO-HALVES (SPLIT-MATRIX DFMATRIX-MINUS-TITLE SPLIT-POINT))
  (DISPLAY-MATRIX (CAR TWO-HALVES))
  (COND [(NOT (NULL (CADR TWO-HALVES))) (DISPLAY-MATRIX (CADR TWO-HALVES))]
  )
  (RETURN T)))

```

(DE DISPLAY-MATRIX (GIVEN-MATRIX)

```

(PROG (FEATURE-LIST BOOLEAN-LIST PHONEME-LIST SPECIAL-COUNT CHARACTER SUB-FLAG
  FIRST-CHARACTER-LIST FLAG-PHONEMES)

```

```

  (SETQ FLAG-PHONEMES (CAR GIVEN-MATRIX))
  (SETQ FEATURE-LIST (CDR GIVEN-MATRIX))
  (COND [(ANY-IN (NTHCHAR-LIST FLAG-PHONEMES 1.) SPECIAL-SUB-CHARACTERS)
    (SETQ FLAG-PHONEMES
      (REMOVE-CHARACTERS SPECIAL-SUB-CHARACTERS FLAG-PHONEMES))
    (SETQ SUB-FLAG T)])
  (SETQ SPECIAL-COUNT (MAXIMUM-CHARACTER-LENGTH FLAG-PHONEMES))

```

NEXT-CHARACTER-LINE

```

  (COND [(ZEROP SPECIAL-COUNT)
    (COND [SUB-FLAG (SETQ FIRST-CHARACTER-LIST
      (NTHCHAR-LIST (CAR GIVEN-MATRIX) 1.))

```

```

                (TERPRI)
                (TAB 6.)
                (GO NEXT-SUB-CHARACTER))
    (( (TERPRI) (GO PRINT-FEATURE))))
    (SETQ PHONEME-LIST FLAG-PHONEMES)
    (TERPRI)
    (TAB 6.)
NEXT-CHARACTER
    (COND [(NULL PHONEME-LIST) (DECR SPECIAL-COUNT) (GO NEXT-CHARACTER-LINE)]
          )
    (SETQ CHARACTER (NTHCHAR (CAR PHONEME-LIST) SPECIAL-COUNT))
    (COND [(NULL CHARACTER) (SPACES 1.)] [T (PRINC CHARACTER)])
    (POP PHONEME-LIST)
    (GO NEXT-CHARACTER)
NEXT-SUB-CHARACTER
    (COND [(NULL FIRST-CHARACTER-LIST) (TERPRI) (GO PRINT-FEATURE)]
          [(IN (CAR FIRST-CHARACTER-LIST) SPECIAL-SUB-CHARACTERS)
            (PRINC (CAR FIRST-CHARACTER-LIST))
            [T (SPACES 1.)]
            (POP FIRST-CHARACTER-LIST)
            (GO NEXT-SUB-CHARACTER)
          ])
PRINT-FEATURE
    (COND [(NULL FEATURE-LIST) (RETURN T)])
    (PRINC (CAR FEATURE-LIST))
    (SETQ BOOLEAN-LIST (CDR FEATURE-LIST))
    (POP FEATURE-LIST)
    (TAB 6.)
NEXT-BOOLEAN
    (COND [(NULL BOOLEAN-LIST) (TERPRI) (GO PRINT-FEATURE)]
          [(PRINC (CAR BOOLEAN-LIST))
            (POP BOOLEAN-LIST)
            (GO NEXT-BOOLEAN))]
    )
(DE ENVIRONMENT-DIRECTION (MORPH-LIST DIRECTION)
  (PROG (NEW-LIST COUNT LIMIT)
    (SETQ LIMIT (ADD1 (MAXIMUM-SPECIAL-WORD-LENGTH MORPH-LIST)))
    (COND [(LESSP DIRECTION 0.) (SETQ LIMIT (MINUS LIMIT)) (SETQ COUNT -1.)]
          [T (SETQ COUNT 1.)])
    NEXT-ENVIRONMENT
    (COND [(EQUAL COUNT LIMIT) (RETURN NEW-LIST)])
    (PUSH NEW-LIST (NTH-ENVIRONMENT MORPH-LIST COUNT))
    (COND [(LESSP DIRECTION 0.) (DECR COUNT)] [T (INCR COUNT)])
    (GO NEXT-ENVIRONMENT)))
(DE FEATURE-DIFFERENCE (MATRIX)
  (PROG (FEATURE-LIST FEATURE-DIFFERENCES)
    (SETQ FEATURE-LIST (LIST-FEATURES))
    NEXT-FEATURE
    (COND [(NULL MATRIX) (RETURN FEATURE-DIFFERENCES)]
          [(EQUAL (CAR MATRIX) '0)
            (PUSH FEATURE-DIFFERENCES (CAR FEATURE-LIST))]
          )
    (POP MATRIX)
    (POP FEATURE-LIST)
    (GO NEXT-FEATURE)))
(DE FEATURE-MATRIX (FEATURE) (CDR (ASSOC FEATURE DFTMATRIX)))
(DE FEATURE-ORDER (FEATURE-LIST)
  (PROG (NEW-MATRIX)
    (SETQ NEW-MATRIX (LIST (CAR DFTMATRIX)))

```

NEXT-FEATURE

```
(COND [(NULL FEATURE-LIST)
      (REWRITE-DEF (REVERSE NEW-MATRIX))
      (RETURN 'OK)]
      [(IN (CAR FEATURE-LIST) (LIST-FEATURES))
      (PUSH NEW-MATRIX (ASSOC (CAR FEATURE-LIST) DEF-MATRIX))
      (POP FEATURE-LIST)
      (GO NEXT-FEATURE)]
      [T (RETURN '===NO-MATCH===)])))
```

(DE FEATURES (MATRIX)

```
(PROG (FEATURE-LIST FEATURES-FOUND)
      (SETQ FEATURE-LIST (LIST-FEATURES))
```

NEXT-FEATURE

```
(COND [(NULL MATRIX) (RETURN FEATURES-FOUND)]
      [(NOT (EQUAL (CAR MATRIX) 'O))
      (PUSH FEATURES-FOUND (SQUASH (CAR MATRIX) (CAR FEATURE-LIST)))]
      (POP MATRIX)
      (POP FEATURE-LIST)
      (GO NEXT-FEATURE)))
```

(DE FILL-STAR-SLOTS (GIVEN-LIST LENGTH-TO-BE)

```
(WITH (LENGTH-OF-LIST LENGTH-DIFFERENCE)
      (SETQ LENGTH-OF-LIST (LENGTH GIVEN-LIST))
      (SETQ LENGTH-DIFFERENCE (DIFFERENCE LENGTH-TO-BE LENGTH-OF-LIST))
      (SPICES '*DUMMY*
              GIVEN-LIST
              (COMBINATION-ORDER
               (CONS '*DUMMY* (REPLICATE '*STAR* LENGTH-DIFFERENCE))))))
```

(DE FIND-NEXT-RULE (UNIQUE-LIST GIVEN-RULE RULE-LIST)

```
(PROG (NEW-LIST RULE SAVE-RULE-LIST)
      (SETQ SAVE-RULE-LIST RULE-LIST)
      (SETQ NEW-LIST (LIST GIVEN-RULE))
```

NEXT-RULE

```
(SETQ RULE (CAR RULE-LIST))
(COND [(NULL RULE-LIST) (RETURN NEW-LIST)]
      [(AND [(IN (CAAR RULE) UNIQUE-LIST)
              (NOT (IN (NTH-ELEMENT (CAR RULE) 2.) UNIQUE-LIST))]
            (PUSH NEW-LIST
                  (COMBINE GIVEN-RULE
                           (FIND-NEXT-RULE
                            (UNIQUE-ELEMENTS
                             (APPEND UNIQUE-LIST
                                     (FIRST-N-ELEMENTS (CAR RULE) 2.))
                             RULE
                             SAVE-RULE-LIST))))))
      (POP RULE-LIST)
      (GO NEXT-RULE)))
```

(DE FIRST-N-ELEMENTS (GIVEN-LIST NUMBER)

```
(PROG (NEW-LIST)
```

NEXT-ELEMENT

```
(COND [(ZEROP NUMBER) (RETURN NEW-LIST)]
      (PUSH NEW-LIST (CAR GIVEN-LIST))
      (POP GIVEN-LIST)
      (DECR NUMBER)
      (GO NEXT-ELEMENT)))
```

(DE FEATURE-IMPLIED-PHONEMES (SIGN-AND-FEATURE)


```

(PROG (PHONEME-LIST NEW-LIST SIGN BOOLEAN-LIST)
  (SETQ SIGN (CAR (EXPLODE SIGN-AND-FEATURE)))
  (SETQ BOOLEAN-LIST
    (FEATURE-MATRIX (READLIST (CDR (EXPLODE SIGN-AND-FEATURE))))))
  (SETQ PHONEME-LIST (LIST-PHONEMES))
NEXT-PHONEME
  (COND [(NULL PHONEME-LIST) (RETURN NEW-LIST)]
    [(EQUAL SIGN (CAR BOOLEAN-LIST))
      (PUSH NEW-LIST (CAR PHONEME-LIST))])
  (POP PHONEME-LIST)
  (POP BOOLEAN-LIST)
  (GO NEXT-PHONEME)))

(DE GET-PREFIX-STEM-SUFFIX (GIVEN-LIST)
  (PROG (PREFIX STEM SUFFIX ENVIRONMENT)
    NEXT-LIST
      (COND [(NULL GIVEN-LIST)
        (RETURN (LIST (REVERSE PREFIX) (REVERSE STEM) (REVERSE SUFFIX)))]
        (SETQ ENVIRONMENT (STEM-ENVIRONMENT (CAR GIVEN-LIST)))
        (APUSH PREFIX (REVERSE (CAR ENVIRONMENT)))
        (PUSH STEM (CADR ENVIRONMENT))
        (APUSH SUFFIX (REVERSE (CAADR ENVIRONMENT)))
        (POP GIVEN-LIST)
        (GO NEXT-LIST)))

(DE GET-BOOLEAN (PHONEME FEATURE)
  (PROG (FEATURE-LIST BOOLEAN BOOLEAN-LIST)
    (SETQ FEATURE-LIST (LIST-FEATURES))
    (SETQ BOOLEAN-LIST (PHONEME-MATRIX PHONEME))
  NEXT-FEATURE
    (SETQ BOOLEAN (CAR BOOLEAN-LIST))
    (COND [(EQUAL FEATURE (CAR FEATURE-LIST)) (RETURN BOOLEAN)])
    (POP BOOLEAN-LIST)
    (POP FEATURE-LIST)
    (GO NEXT-FEATURE)))

(DE GROUP-REGULAR-STEMS (GIVEN-LIST)
  (WITH (EXCEPTION-LIST PREFIXES STEMS SUFFIXES)
    (SETQ EXCEPTION-LIST (SEPARATE-STEM-EXCEPTIONS GIVEN-LIST))
    (SETQ PREFIXES (CDAR EXCEPTION-LIST))
    (SETQ STEMS (CDADR EXCEPTION-LIST))
    (SETQ SUFFIXES (CAADDR EXCEPTION-LIST))
    (LIST (GROUP-STEMS-BY-SEGMENTS STEMS PREFIXES GIVEN-LIST 1.)
      (GROUP-STEMS-BY-SEGMENTS STEMS SUFFIXES GIVEN-LIST -1.))))

(DE GROUP-STEMS-BY-SEGMENTS (STEM-LIST SEGMENT-LIST GIVEN-LIST DIRECTION)
  (PROG (NEW-LIST SCAN-STEM-LIST HOLD-WORDS)
    (SETQ WORD-LIST (STRIP-ONE-LEVEL GIVEN-LIST))
  NEXT-SUFFIX
    (COND [(NULL SEGMENT-LIST) (RETURN NEW-LIST)]
      (SETQ SCAN-STEM-LIST STEM-LIST)
      (SETQ HOLD-WORDS NIL)
    NEXT-STEM
      (COND [(NULL SCAN-STEM-LIST)
        (PUSH NEW-LIST
          (LIST (CAR SEGMENT-LIST)
            (ENVIRONMENT-DIRECTION HOLD-WORDS DIRECTION)))
        (POP SEGMENT-LIST)
        (GO NEXT-SUFFIX)]
        [(IN (SQUASH (CAR SCAN-STEM-LIST) (CAR SEGMENT-LIST)) WORD-LIST)

```



```

(PUSH HOLD-WORDS (CAR SCAN-STEM-LIST)))
(POP SCAN-STEM-LIST)
(GO NEXT-STEM))

```

```

(DE DF-IMPLIED-FEATURES (SIGN-AND-FEATURE)
(MOVE-TO-FRONT
SIGN-AND-FEATURE
(FEATURES (CONVERT-PHONEMES-TO-MATRIX
(FEATURE-IMPLIED-PHONEMES SIGN-AND-FEATURE))))))

```

```

(DV IMPLIED-TABLE ((+syl -obs +voi) (-syl -loja -str) (+cons -loja -str)
(-cons -obs +cont -cor -lat -lab +voi) (+obs -syl +cons
-cp -lat -loja -nas -str) (-obs) (+cont) (-cont +cons
-lat -loja -str) (+high -cp -lat -lab -loja) (-high)
(+back -cor -lab -loja) (-back -rnd) (+tnse -cp -lat
-lab -loja) (-tnse) (+cp -obs -high -tnse -lat -lab
-loja) (-cp) (+cor +cons -back -rnd -lab -loja -str)
(-cor) (+lat +cons -obs +cont -high -tnse -cp -rnd -lab
-loja -nas -str) (-lat) (+rnd +back -cor -lat -lab -loja)
(-rnd) (+lab +cons -high -back -tnse -cp -cor -lat -rnd
-loja -str) (-lab) (+loja +syl -cons -obs +cont -high
-back -tnse -cp -cor -lat -rnd -lab +voi) (-loja)
(+nas -obs -lat +voi) (-nas) (+voi) (-voi -syl +cons
-loja -nas -str) (+str +syl -cons -obs +cont -cor -lat
-lab +voi) (-str)))

```

```

(DE IN (ITEM ITEM-LIST)
(COND [(NULL (MEMBER ITEM ITEM-LIST)) NIL] [T T]))

```

```

(DE LIST-FEATURES NIL (CDAR DFMATRIX))

```

```

(DE LIST-PHONEMES NIL (CDAR DFMATRIX))

```

```

(DE MATRIX-COMPARE (MATRIX1 MATRIX2)
(PROG (NEW-MATRIX BOOLEAN)
NEXT-ELEMENT
(COND [(NULL MATRIX1) (RETURN NEW-MATRIX)]
[(EQUAL (CAR MATRIX1) (CAR MATRIX2)) (SETQ BOOLEAN (CAR MATRIX1))]
[IT (SETQ BOOLEAN '0)]])
(PUSH NEW-MATRIX BOOLEAN)
(POP MATRIX1)
(POP MATRIX2)
(GO NEXT-ELEMENT)))

```

```

(DE MAXIMUM-CHARACTER-LENGTH (GIVEN-LIST)
(MAXLIST (MAPCAR 'FLATSIZE GIVEN-LIST)))

```

```

(DE MAXIMUM-SPECIAL-WORD-LENGTH (WORD-LIST)
(MAXLIST (MAPCAR '(LAMBDA (WORD) (LENGTH (SPECIAL-EXPLODE WORD))) WORD-LIST)))

```

```

(DE MAXLIST (NUMBER-LIST)
(PROG (MAXIMUM)
(SETQ MAXIMUM (CAR NUMBER-LIST))
NEXT-NUMBER
(COND [(NULL NUMBER-LIST) (RETURN MAXIMUM)]
[(GREATERP (CAR NUMBER-LIST) MAXIMUM)
(SETQ MAXIMUM (CAR NUMBER-LIST))])
(POP NUMBER-LIST)
(GO NEXT-NUMBER)))

```

```

(DE MEMBERS (ITEM-LIST GIVEN-LIST)
  (PROG NIL
    (COND [(NULL ITEM-LIST) (RETURN NIL)])
    NEXT-ITEM
    (COND [(NULL ITEM-LIST) (RETURN T)]
      [(NOT (IN (CAR ITEM-LIST) GIVEN-LIST)) (RETURN NIL)])
    (POP ITEM-LIST)
    (GO NEXT-ITEM)))

(DE MINIMIZE-FEATURES (GIVEN-LIST)
  (PROG (FEATURE-LIST REMOVE-LIST)
    (SETQ FEATURE-LIST GIVEN-LIST)
    NEXT-FEATURE
    (COND [(NULL FEATURE-LIST) (RETURN (REMOVES REMOVE-LIST GIVEN-LIST))])
    (APUSH REMOVE-LIST (CDR (ASSOC (CAR FEATURE-LIST) IMPLIED-TABLE)))
    (POP FEATURE-LIST)
    (GO NEXT-FEATURE)))

(DE MINLIST (NUMBER-LIST)
  (PROG (MINIMUM)
    (SETQ MINIMUM (CAR NUMBER-LIST))
    NEXT-NUMBER
    (COND [(NULL NUMBER-LIST) (RETURN MINIMUM)]
      [(LESSP (CAR NUMBER-LIST) MINIMUM)
        (SETQ MINIMUM (CAR NUMBER-LIST))])
    (POP NUMBER-LIST)
    (GO NEXT-NUMBER)))

(DE MOVE-NTH-TO-FRONT (GIVEN-LIST NUMBER)
  (MOVE-TO-FRONT (NTH-ELEMENT GIVEN-LIST NUMBER) GIVEN-LIST))

(DE MOVE-TO-FRONT (ELEMENT GIVEN-LIST)
  (CONS ELEMENT (REMOVE-FIRST-OCCURENCE ELEMENT GIVEN-LIST)))

(DE NTH-ELEMENT (GIVEN-LIST NUMBER)
  (PROG (COUNT)
    (COND [(LESSP (LENGTH GIVEN-LIST) (ABS NUMBER)) (RETURN NIL)]
      [(LESSP NUMBER 0.)
        (SETQ NUMBER (ADD1 (PLUS NUMBER (LENGTH GIVEN-LIST))))])
    (SETQ COUNT 1.)
    NEXT-ELEMENT
    (COND [(EQUAL COUNT NUMBER) (RETURN (CAR GIVEN-LIST))])
    (INCR COUNT)
    (POP GIVEN-LIST)
    (GO NEXT-ELEMENT)))

(DE NTH-ENVIRONMENT (MORPH-LIST NUMBER)
  (PHONEME-IMPLIED-FEATURES (SPECIAL-NTHCHAR-LIST MORPH-LIST NUMBER)))

(DE NTHCHAR-LIST (GIVEN-LIST NUMBER)
  (CAPCAR (LAMBDA (WORD) (NTHCHAR WORD NUMBER)) GIVEN-LIST))

(DE NUMBER-OF-OCCURRENCES (ITEM GIVEN-LIST)
  (PROG (COUNT)
    (SETQ COUNT 0.)
    NEXT-ITEM
    (COND [(NULL GIVEN-LIST) (RETURN COUNT)]
      [(EQUAL ITEM (CAR GIVEN-LIST)) (INCR COUNT)])
    (POP GIVEN-LIST)
    (GO NEXT-ITEM)))

```

```

(DE PAIR-POSSIBLE-REGULAR-AS-RULES (GIVEN-LIST)
  (WITH (REGULAR-LIST PREFIX PREFIX-RULES SUFFIX SUFFIX-RULES)
    (SETQ REGULAR-LIST (GROUP-REGULAR-STEMS GIVEN-LIST))
    (SETQ PREFIX (CAR REGULAR-LIST))
    (SETQ SUFFIX (CADR REGULAR-LIST))
    (SETQ PREFIX-RULES
      (UNIQUE-ELEMENTS
        (COMPARE-LIST-OF-SLOTS
          (STAR-COMBINATIONS (COMBINATION-PAIR PREFIX))))))
    (SETQ SUFFIX-RULES
      (UNIQUE-ELEMENTS
        (COMPARE-LIST-OF-SLOTS
          (STAR-COMBINATIONS (COMBINATION-PAIR SUFFIX))))))
    (COND [(AND [NULL PREFIX] [NULL SUFFIX]) (LIST NIL NIL)]
      [(NULL PREFIX) (LIST NIL SUFFIX-RULES)]
      [(NULL SUFFIX) (LIST PREFIX-RULES NIL)])))

(DE PHONEME-CHANGE (PHONEME-1 PHONEME-2)
  (MAPCAR '(LAMBDA (FEATURE) (SQUASH (GET-BOOLEAN PHONEME-2 FEATURE) FEATURE))
    (PHONEME-DIFFERENCE PHONEME-1 PHONEME-2)))

(DE PHONEME-DIFFERENCE (PHONEME-1 PHONEME-2)
  (FEATURE-DIFFERENCE (CONVERT-PHONEMES-TO-MATRIX (LIST PHONEME-1 PHONEME-2))))

(DE PHONEME-FEATURE-INTERSECTION (PHONEME FEATURE-LIST)
  (COMMON-ELEMENT (PHONEME-IMPLIED-FEATURES (LIST PHONEME)) FEATURE-LIST))

(DE PHONEME-FEATURE-INTERSECTIONS (PHONEME LIST-OF-FEATURE-LISTS)
  (PROG (NEW-LIST INTERSECTION)
    NEXT-FEATURE-LIST
    (SETQ INTERSECTION
      (PHONEME-FEATURE-INTERSECTION PHONEME (CAR LIST-OF-FEATURE-LISTS)))
    (COND [(NULL INTERSECTION) (RRETURN NEW-LIST)])
    (PUSH NEW-LIST INTERSECTION)
    (POP LIST-OF-FEATURE-LISTS)
    (GO NEXT-FEATURE-LIST)))

(DE PHONEME-IMPLIED-FEATURES (PHONEME-LIST)
  (MINIMIZE-FEATURES (FEATURES (CONVERT-PHONEMES-TO-MATRIX PHONEME-LIST))))

(DE PHONEME-MATRIX (PHONEME) (CDR (ASSOC PHONEME DFMATRIX)))

(DE PHONEME-ORDER (PHONEME-LIST)
  (PROG (NEW-MATRIX)
    (SETQ NEW-MATRIX (LIST (CAR DFMATRIX)))
    NEXT-PHONEME
    (COND [(NULL PHONEME-LIST)
      (REWRITE-DF (REVERSE NEW-MATRIX))
      (RETURN 'OK)]
      [(IN (CAR PHONEME-LIST) (LIST-PHONEMES))
      (PUSH NEW-MATRIX (ASSOC (CAR PHONEME-LIST) DFMATRIX))
      (POP PHONEME-LIST)
      (GO NEXT-PHONEME)]
      [T (RETURN '===NO-MATCH===)])))

(OV PLURALS ((aks akse^n) (bø bøz) (bʊs bʊse^z) (buk buks) (bv^g bv^gz)
  (bv^s bv^se^z) (flfθ flfθs) (kət kəts) (kəz kaze^z)
  (lɪd lɪdz) (lɪv lɪvz) (lɔw lɔwz) (li' li'z) (pɒl pɒlz)
  (pv^f pv^fs) (ru3 ru3e^z) (sr^ sr^z) (sv^m sv^mz) (swɪn>g

```

swin>gz) (t**ab** t**abz**) (t**ip** t**ips**) (uz uze^z)))

```
(DE POSSIBLE-REGULAR-AS-RULES (GIVEN-LIST)
  (WITH (POSSIBLE-LIST PREFIX PREFIX-RULES SUFFIX SUFFIX-RULES)
    (SETQ POSSIBLE-LIST (PAIR-POSSIBLE-REGULAR-AS-RULES GIVEN-LIST))
    (SETQ PREFIX (CAR POSSIBLE-LIST))
    (SETQ SUFFIX (CADR POSSIBLE-LIST))
    (SETQ PREFIX-RULES (POSSIBLE-RULES PREFIX))
    (SETQ SUFFIX-RULES (POSSIBLE-RULES SUFFIX))
    (COND [(AND (NULL PREFIX) (NULL SUFFIX)) (LIST NIL NIL)]
      [(NULL PREFIX) (LIST NIL SUFFIX-RULES)]
      [(NULL SUFFIX) (LIST PREFIX-RULES NIL)])))
```

```
(DE POSSIBLE-RULES (RULE-LIST)
  (PROG (NEW-LIST RULE SAVE-RULE-LIST)
    (SETQ SAVE-RULE-LIST RULE-LIST)
  NEXT-RULE
    (SETQ RULE (CAR RULE-LIST))
    (COND [(NULL RULE-LIST) (RETURN (SHORTEST-RULE-ACTION NEW-LIST))])
    (APUSH NEW-LIST
      (STRIP-ONE-LEVEL
        (CDR (FIND-NEXT-RULE
          (FIRST-N-ELEMENTS (CAR RULE) 2.)
          RULE
          SAVE-RULE-LIST))))
    (POP RULE-LIST)
    (GO NEXT-RULE)))
```

(DV PRETERATES ((b**e** b**e**^d) (b**v**^z b**v**^zd) (c**o** c**o**d) (d**lg** d**lg**z) (f**il** f**ild**)
 (f**li** f**lid**) (f**low** f**low**d) (g**a**>lf g**a**>lft) (g**lu** g**lud**)
 (p**ek** p**ekt**) (p**es** p**est**) (p**l**o p**lot**) (p**ln** p**lnd**) (p**us** p**ust**)
 (p**at** p**ate**^d) (p**r**^ p**r**^d) (r**U**3 r**U**3d) (s**lv** s**lvd**) (s**o**d s**ode**^d)
 (s**id** s**ide**^d) (s**l**em s**l**emd) (s**t**eb s**t**ebd) (t**ip** t**ipt**)
 (w**o**^k w**o**^ks)))

```
(DE REMOVE-CHARACTER (CHARACTER WORD-LIST)
  (REMOVE NIL
    (MAPCAR '(LAMBDA (WORD)
      (SQUASH-LIST (REMOVE CHARACTER (EXPLODE WORD)))))
    WORD-LIST)))
```

```
(DE REMOVE-CHARACTERS (CHARACTERS WORD-LIST)
  (REMOVE NIL
    (MAPCAR '(LAMBDA (WORD)
      (SQUASH-LIST (REMOVES CHARACTERS (EXPLODE WORD)))))
    WORD-LIST)))
```

```
(DE REMOVE-FIRST-OCCURENCE (ITEM GIVEN-LIST)
  (PROG (LEFT)
  NEXT-ELEMENT
    (COND [(EQUAL ITEM (CAR GIVEN-LIST))
      (RETURN (APPEND (REVERSE LEFT) (CDR GIVEN-LIST)))]
    (PUSH LEFT (CAR GIVEN-LIST))
    (POP GIVEN-LIST)
    (GO NEXT-ELEMENT)))
```

```
(DE REMOVES (ITEM-LIST GIVEN-LIST)
  (PROG NIL
  NEXT-ITEM
    (COND [(NULL ITEM-LIST) (RETURN GIVEN-LIST)]
```

```
(SETQ GIVEN-LIST (REMOVE (CAR ITEM-LIST) GIVEN-LIST))
(POP ITEM-LIST)
(GO NEXT-ITEM)))
```

```
(DE REPLACE (OLD-ITEM NEW-ITEM GIVEN-LIST)
  (PROG (NEW-LIST)
    NEXT-ITEM
      (COND [(NULL GIVEN-LIST) (RETURN NEW-LIST)]
        [(EQUAL (CAR GIVEN-LIST) OLD-ITEM) (PUSH NEW-LIST NEW-ITEM)]
        [T (PUSH NEW-LIST (CAR GIVEN-LIST))])
      (POP GIVEN-LIST)
      (GO NEXT-ITEM)))
```

```
(DE REPLACE-FEATURE-MATRIX (FEATURE FEATURE-MATRIX)
  (PROG (NEW-MATRIX OLD-MATRIX)
    (SETQ NEW-MATRIX (LIST (CAR DFTMATRIX)))
    (SETQ OLD-MATRIX (CDR DFTMATRIX))
    NEXT-FEATURE
      (COND [(NULL OLD-MATRIX)
        (COND [(NULL NEW-MATRIX) (RETURN '===NO-MATCH===)]
          [T (REWRITE-DFT (REVERSE NEW-MATRIX)) (RETURN 'OK)])]
        [(EQUAL FEATURE (CAAR OLD-MATRIX))
          (PUSH NEW-MATRIX FEATURE-MATRIX)]
        [T (PUSH NEW-MATRIX (CAR OLD-MATRIX))])
      (POP OLD-MATRIX)
      (GO NEXT-FEATURE)))
```

```
(DE REPLACE-PHONEME-MATRIX (PHONEME PHONEME-MATRIX)
  (PROG (NEW-MATRIX OLD-MATRIX)
    (SETQ NEW-MATRIX (LIST (CAR DFMATRIX)))
    (SETQ OLD-MATRIX (CDR DFMATRIX))
    NEXT-PHONEME
      (COND [(NULL OLD-MATRIX)
        (COND [(NULL NEW-MATRIX) (RETURN '===NO-MATCH===)]
          (REWRITE-DF (REVERSE NEW-MATRIX))
          (RETURN 'OK)]
        [(EQUAL PHONEME (CAAR OLD-MATRIX))
          (PUSH NEW-MATRIX PHONEME-MATRIX)]
        [T (PUSH NEW-MATRIX (CAR OLD-MATRIX))])
      (POP OLD-MATRIX)
      (GO NEXT-PHONEME)))
```

```
(DE REPLACES (OLD-ITEM NEW-ITEM LIST-OF-LISTS)
  (MAPCAR '(LAMBDA (A-LIST) (REPLACE OLD-ITEM NEW-ITEM A-LIST))
    LIST-OF-LISTS))
```

```
(DE REPLICATE (ITEM NUMBER)
  (PROG (NEW-LIST COUNT)
    (SETQ COUNT 0.)
    NEXT-ITEM
      (COND [(EQUAL COUNT NUMBER) (RETURN NEW-LIST)]
        (INCR COUNT)
        (PUSH NEW-LIST ITEM)
        (GO NEXT-ITEM)))
```

```
(DE REVERSES (LIST-OF-LISTS) (MAPCAR 'REVERSE LIST-OF-LISTS))
```

```
(DE REWRITE-DF (GIVEN-MATRIX)
  (SETQ DFMATRIX GIVEN-MATRIX)
  (SETQ DFTMATRIX (TRANPOSE DFMATRIX)))
```

```
(DE REWRITE-DFT (GIVEN-MATRIX)
  (SETQ DFMATRIX GIVEN-MATRIX)
  (SETQ DFMATRIX (TRANPOSE DFMATRIX)))
```

```
(DE REWRITE-IMPLIED-TABLE NIL
  (PROG (FEATURE-LIST NEW-TABLE)
    (SETQ FEATURE-LIST (LIST-FEATURES))
  NEXT-FEATURE
    (COND [(NULL FEATURE-LIST)
      (SETQ IMPLIED-TABLE (REVERSE NEW-TABLE))
      (RETURN 'OK)])
    (APUSH NEW-TABLE
      (LIST (DF-IMPLIED-FEATURES (SQUASH '- (CAR FEATURE-LIST)))
        (DF-IMPLIED-FEATURES (SQUASH '+ (CAR FEATURE-LIST)))))
    (POP FEATURE-LIST)
    (GO NEXT-FEATURE)))
```

```
(DE ROTATE-LEFT (GIVEN-LIST)
  (APPEND (CDR GIVEN-LIST) (LIST (CAR GIVEN-LIST))))
```

```
(DE ROTATE-LIST (GIVEN-LIST)
  (PROG (NEW-LIST)
    (PUSH NEW-LIST GIVEN-LIST)
  NEXT-ROTATE
    (PUSH NEW-LIST (ROTATE-LEFT (CAR NEW-LIST)))
    (COND [(EQUAL (CAR NEW-LIST) GIVEN-LIST) (RRETURN (CDR NEW-LIST))])
    (GO NEXT-ROTATE)))
```

```
(DE ROTATE-RIGHT (GIVEN-LIST)
  (MOVE-TO-FRONT (CAR (LAST GIVEN-LIST)) GIVEN-LIST))
```

```
(DM RREMOVE (BODY)
  (LIST 'SETQ (CADDR BODY) (LIST 'REMOVE (CADR BODY) (CADDR BODY))))
```

```
(DM RRETURN (BODY) (LIST 'RETURN (LIST 'REVERSE (CADR BODY))))
```

```
(DE BREAK-INTO-SEGMENTS (SEGMENT)
  (PROG (SEGMENT-LENGTH COUNT EXPLODED-SEGMENT COUNT CURRENT-LENGTH
    CURRENT-SEGMENT COMBINATION-LIST)
    (SETQ EXPLODED-SEGMENT (SPECIAL-EXPLODE SEGMENT))
    (SETQ SEGMENT-LENGTH (LENGTH EXPLODED-SEGMENT))
    (SETQ COUNT 0.)
  NEXT-LENGTH
    (COND [(EQUAL COUNT SEGMENT-LENGTH) (RRETURN COMBINATION-LIST)])
    (SETQ CURRENT-LENGTH (ADD1 COUNT))
    (SETQ CURRENT-SEGMENT EXPLODED-SEGMENT)
  NEXT-SEGMENT
    (COND [(ZEROP CURRENT-LENGTH) (SETQ COUNT (ADD1 COUNT)) (GO NEXT-LENGTH)]
      )
    (PUSH COMBINATION-LIST
      (SQUASH-LIST (FIRST-N-ELEMENTS
        CURRENT-SEGMENT
        (DIFFERENCE SEGMENT-LENGTH COUNT)))))
    (POP CURRENT-SEGMENT)
    (SETQ CURRENT-LENGTH (SUB1 CURRENT-LENGTH))
    (GO NEXT-SEGMENT)))
```

```
(DE SEPARATE-EXCEPTIONS (GIVEN-LIST LIMIT)
  (PROG (TALLY-LIST EXCEPTION-LIST REGULAR-LIST)
```

```

(COND [(NULL GIVEN-LIST) (RETURN NIL)])
(SETQ TALLY-LIST (TALLY-ELEMENTS GIVEN-LIST))
NEXT-TALLY
(COND [(NULL TALLY-LIST)
      (RETURN (CONS (REVERSE REGULAR-LIST) (REVERSE EXCEPTION-LIST)))]
      [(GREATERP (CADAR TALLY-LIST) LIMIT)
       (PUSH REGULAR-LIST (CAAR TALLY-LIST))]
      [(PUSH EXCEPTION-LIST (CAAR TALLY-LIST))])
(POP TALLY-LIST)
(GO NEXT-TALLY)))

(DE SHORTEST-CDR-LISTS (LIST-OF-LISTS)
  (PROG (NEW-LIST SMALLEST-LENGTH)
    (SETQ SMALLEST-LENGTH
      (SHORTEST-LIST-LENGTH (MAPCAR 'CADR LIST-OF-LISTS)))
    NEXT-LIST
    (COND [(NULL LIST-OF-LISTS) (RRETURN NEW-LIST)]
          [(EQUAL (LENGTH (CADAR LIST-OF-LISTS)) SMALLEST-LENGTH)
           (PUSH NEW-LIST (CAR LIST-OF-LISTS))])
    (POP LIST-OF-LISTS)
    (GO NEXT-LIST)))

(DE SHORTEST-LIST-LENGTH (LIST-OF-LISTS)
  (MINLIST (MAPCAR 'LENGTH LIST-OF-LISTS)))

(DE SHORTEST-RULE-ACTION-LENGTH (GIVEN-LIST)
  (MINLIST (MAPCAR '(LAMBDA (RULE-LIST)
                     (MAX (LENGTH (CADAR RULE-LIST)) (LENGTH (CADADR RULE-LIST)))
                     )
                GIVEN-LIST)))

(DE SHORTEST-RULE-ACTION (RULE-LIST)
  (PROG (SMALLEST RULE NEW-LIST)
    (SETQ SMALLEST (SHORTEST-RULE-ACTION-LENGTH RULE-LIST))
    NEXT-RULE-SET
    (SETQ RULE (CAR RULE-LIST))
    (COND [(NULL RULE-LIST) (RRETURN NEW-LIST)]
          [(AND (EQUAL (LENGTH (CADAR RULE)) SMALLEST)
                 (EQUAL (LENGTH (CADADR RULE)) SMALLEST))
           (PUSH NEW-LIST RULE)])
    (POP RULE-LIST)
    (GO NEXT-RULE-SET)))

(DE SPECIAL-EXPLODE (WORD)
  (PROG (NEW-LIST HOLD-LIST CURRENT NEXT EXPLODED-WORD)
    (SETQ EXPLODED-WORD (EXPLODE WORD))
    NEXT-CHARACTER
    (SETQ CURRENT (CAR EXPLODED-WORD))
    (SETQ NEXT (CADR EXPLODED-WORD))
    (PUSH HOLD-LIST CURRENT)
    (COND [(IN CURRENT SPECIAL-SUB-CHARACTERS)
           (GO POP-EXPLODED-WORD)]
          [(OR [(IN NEXT SPECIAL-SUB-CHARACTERS)
                 (NOT (IN NEXT SPECIAL-SUPER-CHARACTERS))]
               [NULL NEXT])
           (PUSH NEW-LIST (SQUASH-LIST (REVERSE HOLD-LIST))
               (COND [(NULL NEXT) (RRETURN NEW-LIST)]
                     (SETQ HOLD-LIST NIL)))]
    POP-EXPLODED-WORD
    (POP EXPLODED-WORD))

```


(GO NEXT-CHARACTER)))

(DE SPECIAL-NTHCHAR (WORD NUMBER)
(NTH-ELEMENT (SPECIAL-EXPLODE WORD) NUMBER))

(DE SPECIAL-NTHCHAR-LIST (WORD-LIST NUMBER)
(PROG (NEW-LIST)
NEXT-WORD
(COND [(NULL WORD-LIST) (RETURN NEW-LIST)])
(PUSH NEW-LIST (SPECIAL-NTHCHAR (CAR WORD-LIST) NUMBER))
(POP WORD-LIST)
(GO NEXT-WORD)))

(DV SPECIAL-SUB-CHARACTERS (! * //))

(DV SPECIAL-SUPER-CHARACTERS (< > ~ ^ //))

(DE SPLICE (OLD-ITEM NEW-ITEM GIVEN-LIST)
(PROG (NEW-LIST)
NEXT-ITEM
(COND [(NULL GIVEN-LIST) (RETURN NEW-LIST)]
[(EQUAL (CAR GIVEN-LIST) OLD-ITEM)
(APUSH NEW-LIST (REVERSE NEW-ITEM))]
[T (PUSH NEW-LIST (CAR GIVEN-LIST))])
(POP GIVEN-LIST)
(GO NEXT-ITEM)))

(DE SPLICES (OLD-ITEM NEW-ITEM LIST-OF-LISTS)
(MAPCAR '(LAMBDA (A-LIST) (SPLICE OLD-ITEM NEW-ITEM A-LIST))
LIST-OF-LISTS))

(DE SPLIT-LIST (GIVEN-LIST CUTTING-POINT)
(PROG (FIRST-HALF COUNT)
(SETQ COUNT 0.)
NEXT-ELEMENT
(COND [(OR [NULL GIVEN-LIST] [EQUAL CUTTING-POINT COUNT])
(RETURN (LIST (REVERSE FIRST-HALF) GIVEN-LIST))])
(INCR COUNT)
(PUSH FIRST-HALF (CAR GIVEN-LIST))
(POP GIVEN-LIST)
(GO NEXT-ELEMENT)))

(DE SPLIT-MATRIX (GIVEN-MATRIX LIMIT)
(PROG (FIRST-HALF SECOND-HALF SPLIT-BOOLEAN)
(SETQ SPLIT-BOOLEAN (SPLIT-LIST (CAR GIVEN-MATRIX) LIMIT))
(COND [(NULL (CADR SPLIT-BOOLEAN)) (RETURN (LIST GIVEN-MATRIX NIL))])
(SETQ FIRST-HALF (LIST (CAR SPLIT-BOOLEAN)))
(SETQ SECOND-HALF (CDR SPLIT-BOOLEAN))
(POP GIVEN-MATRIX)
NEXT-BOOLEAN
(COND [(NULL GIVEN-MATRIX)
(RETURN (LIST (REVERSE FIRST-HALF) (REVERSE SECOND-HALF))])]
(SETQ SPLIT-BOOLEAN (SPLIT-LIST (CAR GIVEN-MATRIX) (ADD1 LIMIT))]
(PUSH FIRST-HALF (CAR SPLIT-BOOLEAN))
(PUSH SECOND-HALF (CONS (CAR SPLIT-BOOLEAN) (CADR SPLIT-BOOLEAN))]
(POP GIVEN-MATRIX)
(GO NEXT-BOOLEAN)))

(DE SQUASH (ATOM1 ATOM2)
(COND [(NULL ATOM1) ATOM2]


```

((NULL ATOM2) ATOM1)
(LET (READLIST (APPEND (EXPLODE ATOM1) (EXPLODE ATOM2))) 1))

```

```

(DEF SQUASH-LIST (GIVEN-LIST)

```

```

  (PROG (NEW-LIST)

```

```

    NEXT-ATOM

```

```

      (COND [(NULL GIVEN-LIST) (RETURN NEW-LIST)])

```

```

      (SETQ NEW-LIST (SQUASH NEW-LIST (CAR GIVEN-LIST)))

```

```

      (POP GIVEN-LIST)

```

```

      (GO NEXT-ATOM)))

```

```

(DEF STAR-COMBINATION (GIVEN-LIST-1 GIVEN-LIST-2)

```

```

  (WITH (LENGTH-1 LENGTH-2 MORPH-1 MORPH-2 ENVIRONMENT-1 ENVIRONMENT-2)

```

```

    (SETQ MORPH-1 (SPECIAL-EXPLODE (CAR GIVEN-LIST-1)))

```

```

    (SETQ MORPH-2 (SPECIAL-EXPLODE (CAR GIVEN-LIST-2)))

```

```

    (SETQ ENVIRONMENT-1

```

```

      (LIST (CAR GIVEN-LIST-1) (CAR GIVEN-LIST-2) (CADR GIVEN-LIST-2)))

```

```

    (SETQ ENVIRONMENT-2

```

```

      (LIST (CAR GIVEN-LIST-2) (CAR GIVEN-LIST-1) (CADR GIVEN-LIST-1)))

```

```

    (SETQ LENGTH-1 (LENGTH MORPH-1))

```

```

    (SETQ LENGTH-2 (LENGTH MORPH-2))

```

```

    (COND [(EQUAL LENGTH-1 LENGTH-2)

```

```

      (LIST (LIST ENVIRONMENT-2 MORPH-1 MORPH-2)

```

```

        (LIST ENVIRONMENT-1 MORPH-2 MORPH-1)))

```

```

    [(LESSP LENGTH-1 LENGTH-2)

```

```

      (STRIP-ONE-LEVEL

```

```

        (MAPCAR '(LAMBDA (STAR-LIST)

```

```

          (LIST (LIST ENVIRONMENT-2 STAR-LIST MORPH-2)

```

```

            (LIST ENVIRONMENT-1 MORPH-2 STAR-LIST)))

```

```

        (FILL-STAR-SLOTS MORPH-1 LENGTH-2)))

```

```

    ]T (STRIP-ONE-LEVEL

```

```

      (MAPCAR '(LAMBDA (STAR-LIST)

```

```

        (LIST (LIST ENVIRONMENT-1 MORPH-1 STAR-LIST)

```

```

          (LIST ENVIRONMENT-2 STAR-LIST MORPH-1)))

```

```

        (FILL-STAR-SLOTS MORPH-2 LENGTH-1)))

```

```

(DEF STAR-COMBINATIONS (GIVEN-LIST)

```

```

  (MAPCAR '(LAMBDA (A-LIST) (STAR-COMBINATION (CAR A-LIST) (CADR A-LIST)))
    GIVEN-LIST))

```

```

(DEF STEM-ENVIRONMENT (WORD-LIST)

```

```

  (PROG (PREFIX STEM SUFFIX SPLIT-WORD)

```

```

    (SETQ STEM (CAR (COMMON-SEGMENTS WORD-LIST)))

```

```

    NEXT-WORD

```

```

      (COND [(NULL WORD-LIST)

```

```

        (RETURN (LIST (REVERSE (REMOVE NIL (UNIQUE-ELEMENTS PREFIX)))

```

```

          STEM

```

```

          (REVERSE (REMOVE NIL (UNIQUE-ELEMENTS SUFFIX))))))

```

```

      (SETQ SPLIT-WORD (STRING-SUBSET STEM (CAR WORD-LIST)))

```

```

      (PUSH PREFIX (CAR SPLIT-WORD))

```

```

      (PUSH SUFFIX (CADR SPLIT-WORD))

```

```

      (POP WORD-LIST)

```

```

      (GO NEXT-WORD)))

```

```

(DEF SEPARATE-STEM-EXCEPTIONS (GIVEN-LIST)

```

```

  (PROG (PREFIX-STEM-SUFFIX NEW-LIST)

```

```

    (SETQ PREFIX-STEM-SUFFIX (GET-PREFIX-STEM-SUFFIX GIVEN-LIST))

```

```

    NEXT-GROUP

```

```

      (COND [(NULL PREFIX-STEM-SUFFIX) (RETURN NEW-LIST)])

```

```

      (PUSH NEW-LIST (SEPARATE-EXCEPTIONS (CAR PREFIX-STEM-SUFFIX) 2.))

```

```
(POP PREFIX-STEM-SUFFIX)
(GO NEXT-GROUP)))
```

```
(DE STRING-SUBSET (ITEM STRING)
  (PROG (ITEM-LIST STRING-LIST LEFT MATCH-STRING)
    (SETQ ITEM-LIST (SPECIAL-EXPLODE ITEM))
    (SETQ STRING-LIST (SPECIAL-EXPLODE STRING))
  NEXT-STRING-LETTER
    (COND [(NULL STRING-LIST) (RETURN NIL)]
      [(NOT (EQUAL (CAR ITEM-LIST) (CAR STRING-LIST))) (GO PUSH-POP)])
    (SETQ MATCH-STRING STRING-LIST)
  NEXT-ITEM-LETTER
    (POP ITEM-LIST)
    (POP MATCH-STRING)
    (COND [(NULL ITEM-LIST)
      (RETURN (LIST (SQUASH-LIST (REVERSE LEFT))
        ITEM
        (SQUASH-LIST MATCH-STRING)))]
      [(NOT (EQUAL (CAR ITEM-LIST) (CAR MATCH-STRING)))]
      (SETQ ITEM-LIST (SPECIAL-EXPLODE ITEM))
      (GO PUSH-POP)))
    (GO NEXT-ITEM-LETTER)
  PUSH-POP
    (PUSH LEFT (CAR STRING-LIST))
    (POP STRING-LIST)
    (GO NEXT-STRING-LETTER)))
```

```
(DE STRIP-ONE-LEVEL (GIVEN-LIST)
  (PROG (NEW-LIST)
  NEXT-ELEMENT
    (COND [(NULL GIVEN-LIST) (RRETURN NEW-LIST)])
    (APUSH NEW-LIST (REVERSE (CAR GIVEN-LIST)))
    (POP GIVEN-LIST)
    (GO NEXT-ELEMENT)))
```

```
(DE TALLY-ELEMENTS (GIVEN-LIST)
  (PROG (UNIQUE-LIST TALLY-LIST)
    (SETQ UNIQUE-LIST (UNIQUE-ELEMENTS GIVEN-LIST))
  NEXT-UNIQUE-ELEMENT
    (COND [(NULL UNIQUE-LIST) (RRETURN TALLY-LIST)])
    (PUSH TALLY-LIST
      (LIST (CAR UNIQUE-LIST)
        (NUMBER-OF-OCCURRENCES (CAR UNIQUE-LIST) GIVEN-LIST)))
    (POP UNIQUE-LIST)
    (GO NEXT-UNIQUE-ELEMENT)))
```

```
(DE TRANSPOSE (LIST-OF-LISTS)
  (PROG (NEW-LIST)
  NEXT-ELEMENT
    (COND [(NULL LIST-OF-LISTS) (RRETURN NEW-LIST)])
    (PUSH NEW-LIST (REMOVE NIL (MAPCAR 'CAR LIST-OF-LISTS)))
    (SETQ LIST-OF-LISTS (REMOVE NIL (MAPCAR 'CDR LIST-OF-LISTS)))
    (GO NEXT-ELEMENT)))
```

```
(DE UNIQUE-ELEMENTS (GIVEN-LIST)
  (PROG (NEW-LIST ELEMENT-LIST)
  NEXT-WORD
    (COND [(NULL (CAR GIVEN-LIST)) (RETURN (REVERSE NEW-LIST)))]
    (SETQ ELEMENT-LIST GIVEN-LIST)
    (POP GIVEN-LIST))
```

NEXT-ELEMENT

```
((COND ((NULL ELEMENT-LIST) (GO NEXT-WORD))  
      ((NULL (MEMBER (CAR ELEMENT-LIST) NEW-LIST))  
        (PUSH NEW-LIST (CAR ELEMENT-LIST))))  
      (POP ELEMENT-LIST)  
      (GO NEXT-ELEMENT)))
```

(NOCOMPILE (DEFV INDEP1 (ADD-FEATURE-BEFORE ADD-PHONEME-BEFORE ANY-IN APUSH BOUN^Y
DRIES CHANGE-BOOLEAN CHANGE-FEATURE-NAME CHANGE-PHONEME-NAME COMBINATION-CDR COM^Y
BINATION-ORDER COMBINATION-PAIR COMBINE COMBINE-APPEND COMMON-ELEMENT COMMON-ELE^Y
MENTS COMMON-SEGMENTS COMPARE-LIST-OF-SLOTS CONVERT-PHONEMES-TO-MATRIX COMPARE-S^Y
LOT COMPARE-SLOTS DELETE-FEATURE DELETE-PHONEME DEMATRIX DFMATRIX DISPLAY-DEMAT^Y
RIX DISPLAY-MATRIX ENVIRONMENT-DIRECTION FEATURE-DIFFERENCE FEATURE-MATRIX FEATU^Y
RE-ORDER FEATURES FILL-STAR-SLOTS FIND-NEXT-RULE FIRST-N-ELEMENTS FEATURE-IMPLIE^Y
D-PHONEMES GET-PREFIX-STEM-SUFFIX GET-BOOLEAN GROUP-REGULAR-STEMS GROUP-STEMS-BY^Y
-SEGMENTS DF-IMPLIED-FEATURES IMPLIED-TABLE IN LIST-FEATURES LIST-PHONEMES MATRIX-CDR
MPARE MAXIMUM-CHARACTER-LENGTH MAXIMUM-SPECIAL-WORD-LENGTH MAXLIST MEMBERS MINIM^Y
IZE-FEATURES MINLIST MOVE-NTH-TO-FRONT MOVE-TO-FRONT NTH-ELEMENT NTH-ENVIRONMENT^Y
NTHCHAR-LIST NUMBER-OF-OCCURRENCES PAIR-POSSIBLE-REGULAR-AS-RULES PHONEME-CHANG^Y
E PHONEME-DIFFERENCE PHONEME-FEATURE-INTERSECTION PHONEME-FEATURE-INTERSECTIONS ^Y
PHONEME-IMPLIED-FEATURES PHONEME-MATRIX PHONEME-ORDER PLURALS POSSIBLE-REGULAR-A^Y
S-RULES POSSIBLE-RULES PRETERATES REMOVE-CHARACTER REMOVE-CHARACTERS REMOVE-FIRS^Y
T-OCCURENCE REMOVES REPLACE REPLACE-FEATURE-MATRIX REPLACE-PHONEME-MATRIX REPLAC^Y
ES REPLICATE REVERSES REWRITE-DF REWRITE-DFT REWRITE-IMPLIED-TABLE ROTATE-LEFT R^Y
OTATE-LIST ROTATE-RIGHT RREMOVE RRETURN BREAK-INTO-SEGMENTS SEPARATE-EXCEPTIONS^Y
SHORTEST-CDR-LISTS SHORTEST-LIST-LENGTH SHORTEST-RULE-ACTION-LENGTH SHORTEST-RULE-A
CIAL-EXPLODE SPECIAL-NTHCHAR SPECIAL-NTHCHAR-LIST SPECIAL-SUB-CHARACTERS SPECIAL^Y
-SUPER-CHARACTERS SPLICE SPLICES SPLIT-LIST SPLIT-MATRIX SQUASH SQUASH-LIST STAR^Y
-COMBINATION STAR-COMBINATIONS STEM-ENVIRONMENT SEPARATE-STEM-EXCEPTIONS STRING-SUBS
RIP-ONE-LEVEL TALLY-ELEMENTS TRANSPOSE UNIQUE-ELEMENTS)))

END Job PHONEX Req #261 for CIS78804E.DEHILSTER Date 2-Jun-83 15:1
0:23 Monitor: Ohio State University IRCC/CIS DEC-2060. **E

*** L P T S P L Run Log ***

15:00:56 LPDAT LPTSPL version 104A(2652)-1 Ohio State University IRCC/CIS DEC-2
15:00:56 LPDAT Job PHONEX sequence #1133 on Printer 1 at 2-Jun-83 15:00:56
15:01:02 LPMSG Starting File PS:<CIS78804E.DEHILSTER>PHONEX.MSS.99
15:02:54 LPMSG Finished File PS:<CIS78804E.DEHILSTER>PHONEX.MSS.99
15:02:54 LPMSG Starting File PS:<CIS78804E.DEHILSTER>PHONFUN.MSS.98
15:09:33 LPMSG Finished File PS:<CIS78804E.DEHILSTER>PHONFUN.MSS.98
15:09:34 LPMSG Starting File PS:<CIS78804E.DEHILSTER>PHONT00L.LSP.1
15:16:23 LPMSG Finished File PS:<CIS78804E.DEHILSTER>PHONT00L.LSP.1
15:16:23 LPEND Summary: 46 Pages of Output
15:16:23 LPEND 19 Disk Pages Read
15:16:23 LPEND 17.434 Seconds CPU Time Used