LISP IMPLEMENTATION

Oliver Gajek
Hanno T. Beck
Diane Elder
Greg Whittemore

1. Data Files

The data files serve as input which Pascal and LISP compile into the morphology generator/analyzers. These data files are "nuts and bolts" descriptions of the phonological processes, and of the affixes and word roots of a particular language. The data files are of two general types: sets of finite state automata, and dictionaries. The automata allow a step-by-step analysis and generation of the surface forms of words and represent the phonological and morphological rules of a language. The dictionary supplies two kinds of information: the canonical form of lexical items along with pertinent morphological features (e.g. if the item in question is a suffix or root morpheme) and morphosyntactic information that implicitly determines the order in which morphemes may occur.

It is not mandatory for one to use the same structure for the data file for the two different programming languages; it would actually be more convenient to use a different set-up for LISP, but to maintain compatibility between the two programs, we chose to stay with the Pascal version. The manner in which the data files are read will of course differ for the two languages. Pascal is inferior to LISP in string-handling facilities, so much of its program consists of low-level routines for creating and accessing fairly complicated data structures. Both versions will also have the same output.

The input data files are read in one at a time. A common nomenclature is used for easy retrieval. Automata file names have the general form "anylanguage.AUT" (e.g. ENGLISH.AUT); dictionary file names, "anylanguage.DIC" (e.g. JAPANESE.DIC).
1.1. The Automata Input Files

Each "anylanguage.AUT" file contains a definition of the character forms necessary to guide the programs through the finite state automata (FSAs) for the proper generation and recognition of the allomorphic suffix strings. The initial component of the character definition is the ALPHABET. This is typically a set of regularly used alphabetic orthographic symbols. Other symbols such as the "+" marker and special phonemic characters normally not represented in an alphabetic system will also appear. The size of the ALPHABET varies from language to language and is sometimes quite large.

Following the ALPHABET are three other symbol types: NULL or 0, the "null string"; ANY or "=" the character that can represent any alphabetic symbol; and SUBSETs like "C" or "V", any subgrouping of the ALPHABET. A "0" is used for times when there may be a realized surface character for a non-existing lexical one and vice-versa. The other two aid in the automaton formulation since one would not want to re-list the whole alphabet or even a subset of it for each time any character could appear. "END", a widely used Pascal convention, then marks the character definitions as complete (see sample below).

The automata themselves are matrices matching input (lexical/surface pairs) with states. Each "anylanguage.AUT" may have any number of FSA's (Koskenniemi has 19 for Finnish, a highly inflected language). Within the file, each FSA must be labeled on the first line by a "string" (a series of characters within quotation marks) for easy recognition by the processing program. Also, on the first line, there are two numerals; the first one denotes the "breadth" of the matrix, or the number of lexical/surface character pairs, and the second denotes the "depth", the number of states.

To show lexical/surface pairs, the lexical characters are listed in a row with their counterparts aligned immediately under them in a row. This is equivalent to the idea of underlying and surface characters.

States are then lined up under the appropriate character correspondences to indicate the routing of recognition or generation involved for any particular phonological rule. The column of numbers to the far left (see below) is a list of all the states and their "finality;" the ones followed by ":." are final, the others with "." are not.
The first FSA listed, SURFACE CHARACTERS, is required by the analysis/generation algorithm and serves only to list the trivial lexical/surface pairs, i.e., those that are not affected by a phonological rule. The remaining FSA's then appear and are followed by END to indicate that the list of FSA's is complete. A skeleton of a sample automata file is shown below:

```
ALPHABET
  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 +
  NULL 0
  ANY =
  SUBSET C b c d f g h j k l m n o p q r s t v w y z
  SUBSET V a e i o u
  SUBSET W b d g k m n p
  SUBSET S ... 
  END
```

"Surface Characters" 1 28
  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' =
  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' =
  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

"Epentheses" 5 7
  c h S y ++ =
  c h S i o e =
  1: 3 1 2 2 1 0 1
  2: 2 2 2 2 4 5 1
  3: 1 2 2 2 1 0 1
  4: 0 0 0 0 0 0 1
  5: 0 0 1 0 0 0 0

"Gemination" 15 25
  etc....... 
  END

1.2. The Dictionary Input Files

The "anylanguage.DIC" files comprise the lexical descriptions of a language. The first portion defines the "morphological alternations," the possible combinations of word classes and suffixes. Items that can be affixed are generally marked by a "/"; "non-continuable" sorts have no marking. For example the class "verb" is shown as "/V" and in English is considered to be
"affixable" by the suffixes such as the present third person singular (PS), the past (PS), progressive (PR), etc. A different continuation class is necessary to block affixation of regular suffixes to irregular forms. Each irregular pattern must have a separate continuation class (see Karttunen and Wittenburg's article). The items that can be suffixed are then parenthetically enclosed with all of the suffixes that can follow (e.g. (/N = P3 PS PP PR etc.)). Items that cannot have affixes are also listed and are indicated as being items that equal themselves (e.g. (Cl = Cl)). A sample is given below.

The second part of the file is a list of individual LEXICONs. Each LEXICON has a detailed description for both the suffix and root morphemes given on the right side of the "=" in ALTERNATIONS. For the suffix LEXICONs, this includes the name of the lexicon (e.g. P3), the canonical string it consists of (e.g. "+s" for the third person singular, or 0 or the null string), the class of morphemes that can be attached, (a "#" indicates that only a word boundary can follow), and other morphological information which is put in a string (i.e. " "). For the Root LEXICON, a large list of basic vocabulary items is given along with word class designations (e.g. "N" for noun) and other relevant morphological information. The END again is placed at the end of the data.

A sample skeleton of an anylanguage.DIC file looks like this:

```
ENGLISH.DIC

ALTERNATIONS
( /N = N
( /V = P3 PS PP PR I AG )
( /A = PA CA CS Ly )
( Cl = Cl)
etc.............

END

LEXICON N 0 Cl "N SG"; +s C2 "N PL"
etc.............
LEXICON Root am # "V PRES SING 1st"
    am # "AUX"
    etc.............

END
```
Lisp Implementation

2. LISP Representation of Automata

The contents of the automata file are stored in LISP as properties of the relevant language. Once a file ENGLISH.AUT, for example, has been read in, the atom ENGLISH will have properties ALPHABET, NULL, ANY, and SUBSETS. The automata are represented in several ways:

SIMPLEAUTOMATA

This is a list of automata, where each automaton is a list of the rows of the automaton as it appears on the input file. This representation is used for editing automata and dumping automata on the screen or on a file in the common file format. The following is an example automaton in its SIMPLEAUTOMATA representation:

```lisp
(("Testaut" X Y)
 (a b =)
 (a 0 =)
 (1: 1 2 3)
 (2: 2 3 1)
 (3: 3 1 2))
```

AUTOMATA

A list of automata, where each automaton is essentially a 90-degree inversion of its SIMPLEAUTOMATA counterpart. Each automaton is a list of lists, where the first list contains the name and height and width specifications. The second list is the finality list, which is a vector that gives the finality (T or NIL) for each state of the automaton. Here we make crucial use of the fact that the states of any automaton are natural numbers from 1 to n. The position of an element in the vector is determined by the statenumber it is associated with. Therefore the finality for state 3, for example, will be the third element of the finality vector. The other elements of the list are the columns of the automaton. Each such list has as its first element a dotted pair consisting of a lexical character and the surface character that it can correspond to. The rest of this list can again be used as a vector, where position corresponds to the state the automaton is previously in, and the state at that position is the new state that the automaton is in after advancing over the particular character correspondence. The following is the example automaton in its AUTOMATA representation:
( "Testaut" X Y )
( T T NIL )
((a . a) 1 2 3 )
((b . 0) 2 3 1 )
((= . =) 3 1 2 )

The AUTOMATA representation is used for compiling both GMACHINE and RMACHINE, and BIGAUTOMATON.

GMACHINE and RMACHINE

These representations are designed to be make automata access more efficient for the recognition/analysis algorithm. GMACHINE is used for generation. It is a list of lists, one for each possible lexical character. Each such character has associated with it several lists, one for each surface character that it can correspond to, where all correspondences have to be explicitly licensed by an automaton. The surface character leads to a list of transition vectors, one for each automaton. The following is a simple GMACHINE:

( (a (a (1)
   (1 2))
   (0 (1)
    (2 0)) )
 (b (b (1)
    (2 2)) )
  .
  .
)

Here lexical "a" can be either realized as "a" on the surface or it can be deleted, therefore it has two branches associated with it. Lexical "b", on the other hand, can appear on the surface only as "b". Even those trivial correspondences have to be explicitly licensed by an automaton, specifically the "Surface Characters" automaton. In the generation mode the algorithm accesses the GMACHINE representation through the characters in the lexical characters. RMACHINE has the same structure as GMACHINE except that access here is by surface character and it is used by the recognition algorithm. Note that GMACHINE and RMACHINE share most of their structure, so that the representation is efficient as far as storage space is concerned. GMACHINE and RMACHINE are constructed out of the AUTOMATA representation and the set of feasible pairs, where a feasible pair is a correspondence between a lexical character and a surface character that is licensed by at least one automaton. The SUBSET information is used to determine
the state transitions for feasible pairs not explicitly mentioned by a given automaton.

BIGGMACHINE and BIGRMACHINE

These are the representations of the automaton that is the result of merging all automata associated with a language into a single machine. BIGGMACHINE has the same structure as GMACHINE, except for the fact that here there is only one automaton, and BIGRMACHINE, the representation used for recognition, has the same structure as RMACHINE. These representations are used when the system is running in the "fast" mode. During the compilation of the GMACHINE and RMACHINE representations, the system constructs equivalence sets of feasible pairs where all members of an equivalence set behave alike as far as the automata are concerned. These pairs are typically pairs like \((k \cdot k)\), \((p \cdot p)\), and so forth, all of which are normally not specifically mentioned by any automaton other than the first one. In constructing BIGRMACHINE and BIGGMACHINE the system makes use of this information by having all members of an equivalence set (physically) share the transitions computed for its first member. This causes a significant reduction in the amount of storage space necessary for these representations.

3. LISP Representation of Dictionary Material

There are three kinds of dictionary-related information which must be stored in a LISP-useable format; words, suffix forms, and rules which specify which suffixes may attach to which words.

Words are represented in a tripartite manner. Each complete word consists of: (a) an underlying root morpheme (which may differ from the word's surface form somewhat, since it may contain morphophonemes, stress markings and other matter which is filtered out through the operation of finite state automata and thus never appears on the surface), (b) a continuation class, a symbol which describes the word's allowable morphological behavior, and (c) an optional additional piece of information, for example a translation of the word into intensional logic, or if the dictionary file is being constructed for a non-English language, an English translation of the word.

What LISP-compatible form is this collection of information put into? For a human, the most intuitive and accessible way to store these items would probably be just to write down each individual root morpheme, continuation class and translation and be done with
it. Even for LISP, there is nothing technically unworkable about
doing this, but it would be a very inelegant enterprise and would,
in the case of a large dictionary, occupy enormous amounts of
storage space.

Much more efficient use of space can be made by eliminating
some repetitiveness from the lexicon, just as was done by the
creation of continuation classes. Let us consider a simple
example. Suppose we wanted to store, among other things, the root
morphemes ski and skip, which are associated with the continuation
classes N and V respectively. An efficient LISP representation of
these words and their classes would be:

\[(s \text{ NIL} (k \text{ NIL} (i ((N)) (p ((V)))))\]

This list contains a lot of information— the word ski is
associated with label N, the word skip is associated with the
label V, and there are no words s or sk, as we can tell by the
NILs in those positions which would contain their continuation
classes if they had any. Note how the characters s, k and i do
double duty here. In the same way, if we were to decide that sk
really is a word and wanted to add it to the lexicon, we would not
need to add a list (s NIL (k ((X))))), but could simply change the
second NIL in the list above to ((X)).

This list structure, though simple enough to permit LISP to
manipulate it in many ways, is flexible enough to allow for any
number of extensions. Suppose we had a word, say move, which we
wanted to be able to treat morphologically as either taking the
continuation class N or the class V. Rather than entering it into
the lexicon twice, we may represent it as:

\[(m \text{ NIL} (o \text{ NIL} (v \text{ NIL} (e ((N)(V)))))\]

This is why it is useful to represent continuation classes in
LISP as lists of lists, such as ((N)), rather than mere lists,
such as (N); we can preserve a single format for representations
of words, no matter how many continuation classes they are
associated with.

The translation or other piece of information associated with
each word's underlying representation is given in LISP as a
string, a sequence of characters enclosed in double quotation
marks. These strings are juxtaposed to the continuation class
labels for each word. Thus a complete rendering of the word move
might be (m NIL (o NIL (v NIL (e ((N "..."))((V "...")))))].
When there are several words which share initial characters, the LISP representation of them will come to take the form of a multiply-branching tree. In the case of a large lexicon we will actually have as many different trees as there are word-initial characters. To gain a still more uniform treatment, we may take these trees and attach a dummy element (the null string "") to them as an initial character, and so construct a single giant tree of trees, containing all the words in the dictionary but maintaining the original simple structure throughout. Here is an example of a tree which, though still quite unimposing, has more richness than the others examined so far (note that this expression is still a single list, even though it occupies several lines and has been indented at places to show its structure more revealingly):

\[
(\"
\Nil\ (c\ \Nil\ (a\ \Nil\ (r\ ((N\ ""))))
(d\ ((N\ ""))))
(r\ \Nil\ (y\ ((V\ ""))))
(t\ ((N\ ""))))
(u\ \Nil\ (p\ ((N\ ""))))
(k\ \Nil\ (i\ \Nil\ (l\ ((V\ ""))))
(s\ \Nil\ (s\ ((V\ ""))
\Nil\ ""))))
\]

This tree effectively stores all the necessary information for nine words: car N "", card N "", carry V "", cart N "", cat N "", cup N "", kill V "", kiss V "", and kiss N "".

A second part of a complete dictionary file is the decoding of the continuation classes. Each continuation class is an abbreviation for a bundle of suffix forms which a given root morpheme is compatible with. In this way a single symbol can encode the entire allowable morphological behavior of a word. Since many words share parts of their behavior, this procedure allows us to avoid much duplication of information in the lexicon; so now it will only be necessary to spell out the morphological content of these continuation classes once. In a LISP format, this may simple be done by constructing a list of lists, such that the first member of each sublist os a continuation class, and the remaining members of each sublist are the suffix continuations which it abbreviates. (The suffix continuation classes are themselves treated as singleton continuation classes.) So for example, the list ((X A)(Y A B C D)) means that continuation class X stands for the suffix continuation class A, while continuation class Y stands for classes A, B, C and D.
A third necessary part of a dictionary file will consist of expanding out each basic, non-abbreviatory continuation class according to what its underlying form is and what contribution it makes to a word. For these extensions, we may use the exact same format as outlined above for storing a lexicon of words. That is, we may construct a lexicon tree for each continuation class. For each possible terminus of a word, the lexicon might include information strings which describe the grammatical role played by the suffix-modified word; for example, one such lexicon for English is:

"" NIL (+ NIL (I NIL (g ((# "V PROG"))))))

To recap, the overall LISP version of a complete dictionary file will consist of three basic parts. There must be a treelike list containing the underlying form, continuation class and perhaps translation of every root morpheme. There must be a list of lists telling which singleton suffix continuation classes are associated with each abbreviatory continuation class. Finally, there must be miniature lexicon trees for each singleton continuation class, giving the suffix's underlying form and describing its role.

4. The Analysis and Generation Routines

The following is a discussion of our specific implementation. A discussion of the general algorithm can be found in Lauri Karttunen's chapter in this volume. The main difference is that we do not actually have to explicitly construct "tasks" that get stored away for later execution as Koskenniemi does, but that we make full use here of LISP's capabilities for recursive programming, avoiding any elaborate bookkeeping. The search procedure is therefore depth-first, and not breadth-first as in the PASCAL implementation. The main source for computational cost is the constant creation of new automata configurations, i.e., the collection of the states the automata are currently in. Here the most efficient solution turned out to be the creation of a discrimination net that contains every configuration encountered so far. The function that advances the automata over pairs of lexical and surface characters keeps track of the contents of this discrimination net and consequently never has to duplicate configurations that have already come up.

The following discussion of the analysis/generation procedures as well as the actual functions listed in the appendix constitute
the mere core of the algorithm. The procedures handling the extensive tracing facilities and the options for using merged automata have been left out for the sake of perspicuity.

4.1. The Generation Routine

When starting the generation routine, the KIMMO program calls the function TESTGEN with the current language as its argument. This function asks the user for lexical forms that are to be generated. TESTGEN calls the function GENFORM repeatedly on those lexical forms. GENFORM calls the SETUP initialization function which establishes the initial automata configuration (all machines in state 1) and selects the correct automata representation, in this case the GMACHINE. GENFORM then calls the main recursive function, GENFORM1, with the lexical form in the argument REMAIN, the present automata states in CONF, and STRPOINTER, a pointer to an array-like data structure that keeps track of the output characters, i.e., the surface characters. GENFORM1 removes the first character from REMAIN, iterating through the character's possible surface realizations calling GENCHAR with each.

GENCHAR is the function that does the real work of the generation process. Its argument CHOICE consists of a list containing the matching surface character and a list of the automata transitions associated with the particular lexical-surface pair. The function MOVEAUT advances the the automata in their present configuration over the pair, and, if no automaton blocks, the current surface character is added to the output array.

If REMAIN is not NIL at this point, GENFORM1 is called recursively to test the following character. However, if REMAIN is NIL, the function FINALSTATE checks whether every automaton is in a final state, in which case the result is returned, otherwise the search branch is terminated.

4.2. The Recognition Routine

When operating as a morphological recognizer, KIMMO calls the function TESTREC with the current language as its argument. This function calls RECFORM on the lexical form supplied by the user. RECFORM calls the SETUP function for initialization, where the correct machine, in this case RMACHINE, is selected and an initial configuration is constructed. RECFORM next finds the correct STARTSET, i.e., the names of the lexica which initially guide the
recognition process. In most cases this will be the Root lexicon, and possibly a prefix lexicon. Since the recognition routine is driven by lexical entries, the function RECFORM1, which is the main recursive function, is then called successively with each member of the STARTSET. RECFORM1 has the same arguments as GENFORM1 above, plus an additional one, LEXBRANCH, which marks the current position in the lexicon, which is represented as a letter tree. Because RECFORM1 operates recursively, it has to consider all possible options every time it is called. First it tests whether or not the LEXBRANCH is at the end of a morpheme, i.e., whether there is some lexical information at this point. If so, the function LEXENTRY is called on each entry. Regardless of the success of this step, RECFORM1 then calls RECCHAR (which operates in the same way that GENCHAR above does) first with 0 (the null character) and then with the next character of REMAIN. Both steps are necessary because the recognizer has to check continually if an underlying character has been deleted in the surface string.

LEXENTRY calls the function LEXMATCH on the continuation lexica that are specified in the entry. LEXMATCH first checks whether the continuation class is "#", the word boundary character, and, if there is no REMAIN and all automata are in final states, the word and its features are returned as the final result of the analysis. If there is no word boundary, the recursive process RECFORM1 is evoked again with the remainder of the input string and the new lexicon.
Lisp Implementation

A. Functions

(TESTGEN
 [LAMBDA (LANGUAGE)
  (PROG (LINE RES)
     (printout T "Please type one or more lexical forms" T ":")
     (while (SETQ LINE (LINEREAD T))
       do [MAPC LINE
            (FUNCTION (LAMBDA (WRD)
                        (COND
                         [(SETQ RES (GENFORM WRD LANGUAGE))
                          (MAPC RES (FUNCTION (LAMBDA (R)
                                     (printout T 4 R T)
                                     (T (printout T 4 WRD "?" T)
                                     (PRIN1 ":" T))
                                     (RETURN LANGUAGE))]
        (GENFORM
 [LAMBDA (WRD LANGUAGE)
  (PROG (INITCONFIG MACHINE AUTOMATA (STRPOINTER 1)
         (DIR (QUOTE GEN)))
        (COND
         ((SETUP)
          (RETURN (GENFORM1 (DUNPACK WRD SCRATCHLIST)
                         INITCONFIG STRPOINTER))
        (GENFORM1
 [LAMBDA (REMAIN CONFIG STRPOINTER)
  (PROG (CH)
     (SETQ CH (pop REMAIN))
     (RETURN (MAPCONC (CDR (FASSOC CH MACHINE))
                      (FUNCTION GENCHAR))
        (GENCHAR
 [LAMBDA (CHOICE)
  (PROG (NEWCONFIG RES (STRPNT STRPOINTER))
   (COND
    ((SETQ NEWCONFIG (MOVEAUT CONFIG (CDR CHOICE)))
     [COND
      ((NEQ (CAR CHOICE)
       0)
       (PUTSTR (CAR CHOICE)
        STRPNT)
      (SETQ STRPNT (ADD1 STRPNT]
    (RETURN (COND
             (REMAIN (GENFORM1 REMAIN NEWCONFIG STRPNT))
             ((FINALSTATE NEWCONFIG AUTOMATA)
              (SETQ RES (LIST (CONCAT (GETSTR STRPNT)))))
(TESTREC
  [LAMBDA (LANGUAGE)
    (PROG (LINE RES)
      (printout T "Please type one or more surface forms" T ":")
      [while (SETQ LINE (LINEREAD T))
        do (MAPC LINE
          (FUNCTION (LAMBDA (WRD))
            (COND
              [(SETQ RES (RECFORM WRD LANGUAGE))
                (MAPC RES (FUNCTION (LAMBDA (R))
                  (printout T 4 (CAR R)
                    ,
                    (CDR R)
                    T]
                )
              (T (printout T 4 WRD "?" T)))
            ]
          )
        )
      )
    )
  )
(RECFORM
  [LAMBDA (WORD LANGUAGE)
    (PROG (ALTERNATIONS AUTOMATA INITCONFIG MACHINE STARTSET
      (FEATURES "")
      (LEXICON (QUOTE Root))
      (STRPOINTER 1)
      (DIR (QUOTE REC))
      (REMAIN (DUNPACK WORD SCRATCHLST)))
    )
    (COND
      [(SETUP)
        (SETQ STARTSET (GETP LANGUAGE (QUOTE STARTSET)))
        (RETURN (MAPCONC STARTSET (FUNCTION (LAMBDA (LEXICON)
          (RECFORM1 REMAIN INITCONFIG
            (GETP LANGUAGE LEXICON)
            STRPOINTER)))
        )
      ]
    )
  )
(RECFORM1
  [LAMBDA (REMAIN CONFIG LEXBRANCH STRPOINTER)
    (NCONC [MAPCONC (CDR LEXBRANCH)
      (FUNCTION (LAMBDA (ENTRY)
        (LEXENTRY (CAR ENTRY)
          (MERGEFEATS FEATURES (CDR ENTRY)
        )
      )
    )
  ]
    (MAPCONC (CDDR LEXBRANCH)
      (FUNCTION (LAMBDA (NEWBRANCH)
        (NCONC (RECCHAR 0 REMAIN)
          (RECCHAR (CAR REMAIN)
            (CDR REMAIN))))
      )
  )
(RECCCHAR
 [(LAMBDA (CH NEWREMAIN)
    (PROG (CHOICE NEWCONFIG)
       (RETURN (COND
                  ([AND (SETQ CHOICE (FASSOC (CAR NEWBRANCH)
                     (CDR (FASSOC CH MACHINE)
                     (SETQ NEWCONFIG (MOVEAUT CONFIG (CDR CHOICE)
                     (PUTSTR (CAR NEWBRANCH)
                     STRPOINTER)
                     (RECFORM1 NEWREMAIN NEWCONFIG NEWBRANCH
                     (ADD1 STRPOINTER)]))

(LEXMATCH
 [(LAMBDA (LEXICON)
    (COND
     ([EQ LEXICON (QUOTE #)]
      (COND
       ([AND (NULL REMAIN)
        (FINALSTATE CONFIG)]
        (LIST (CONS (CONCAT (GETSTR STRPOINTER))
        FEATURES]
     (T (RECFORM1 REMAIN CONFIG (GETP LANGUAGE LEXICON)
        STRPOINTER)]))

(LEXENTRY
 [(LAMBDA (CONTCLASS FEATURES)
    (PROG [[(LEXLST (CDR (FASSOC CONTCLASS ALTERNATIONS)
      (RETURN (COND
     ([LISTP LEXLST)
      (MAPCONC LEXLST (FUNCTION LEXMATCH)))
     ([LEXMATCH CONTCLASS]])

(MOVEAUT
 [(LAMBDA (CFIG TRANSNS)
    (PROG (NEWSTATE NEWCONFIG)
       (COND
        ([EVERY CFIG
          (FUNCTION (LAMBDA (STATE)
           (COND
            ([NOT (ZEROP (SETQ NEWSTATE
             (CAR (PNTH (pop TRANSNS) STATE)
             (COND
            (NEWCONFIG (NCONC1 NEWCONFIG NEWSTATE))
            (T (SETQ NEWCONFIG (LIST NEWSTATE)
             (RETURN NEWCONFIG)]))

(Lisp Implementation)
(MERGEFEATS
   [LAMBDA (STEMFEATS SUFFEATS)
     (COND
      ((ZEROP (NCHARS SUFFEATS))
        STEMFEATS)
      ((ZEROP (NCHARS STEMFEATS))
        SUFFEATS)
      ((CONCAT STEMFEATS " " SUFFEATS))
     )]

(FINALSTATE
   [LAMBDA (CFG)
     (PROG ((AUTS AUTOMATA)
        AUT)
       (RETURN (EVERY CFG (FUNCTION (LAMBDA (STATE)
             (SETQ AUT (pop AUTS))
             (CAR (FNTH (CADR AUT)
               STATE)))
        ))
     )]

(SETUP
   [LAMBDA NIL
     (SETQ SCRATCHSTR (CONCAT " " " "))
     (SETQ ALTERNATIONS (GETP LANGUAGE (QUOTE ALTERNATIONS)))
     (SETQ AUTOMATA (GETP LANGUAGE (QUOTE AUTOMATA)))
     (SETQ INITCONFIG (PUT LANGUAGE (QUOTE INITCONFIG)
       (to (LENGTH AUTOMATA) collect 1)))
     (SETQ MACHINE (GETP LANGUAGE (SELECTQ DIR
       (GEN (QUOTE GMACHINE))
       (REC (QUOTE RMACHINE))
       NIL)))
   )]

(PUTSTR
   [LAMBDA (STR PNT)
     (RPLSTRING SCRATCHSTR PNT STR)])

(GETSTR
   [LAMBDA (PNT)
     (OR (SUBSTRING SCRATCHSTR 1 (SUB1 PNT)) "")])