ED 370 437 FL 022 218

AUTHOR Lee, Kang-Hyuk
TITLE P-KIMMO: A Prolog Implementation of the Two Level Model.
INSTITUTION Illinois Univ., Urbana. Language Learning Lab.
REPORT NO LLL-TR-T-18-91
PUB DATE Mar 91
NOTE 38p.
AVAILABLE FROM Language Learning Laboratory, University of Illinois at Urbana-Champaign, G70 Foreign Languages Building, 707 S. Mathews, Urbana, IL 61801.
PUB TYPE Reports - Descriptive (141)

EDRS PRICE MF01/PC02 Plus Postage.
DESCRIPTORS *Authoring Aids (Programming); *Computational Linguistics; *Computer Software; Microcomputers; *Morphology (Languages); *Structural Analysis (Linguistics)

ABSTRACT

Implementation of a computer-based model for morphological analysis and synthesis of language, entitled P-KIMMO, is discussed. The model was implemented in Quintus Prolog on a Sun Workstation and exported to a Macintosh computer. This model has two levels of morphophonological representation, lexical and surface levels, associated by morphophonological rules that specify legitimate pairs of characters. The description offered here focuses on aspects of implementation only and not underlying theory. Components of the program are described, including structure of the lexicon, use of finite state automata to encode two-level rules, and the recognizer/generator algorithm. This version of the program is then compared and contrasted with a previously implemented version. Finally, procedures for use of the program on the UNIX and Mackintosh computers are outlined, with some screen illustrations. A brief bibliography is included, and a source listing of the P-KIMMO system is appended. (MSE)

Language Learning Laboratory

LLL

TECHNICAL REPORT NO. LLL-T-18-91
MARCH 1991

The School of Humanities

University of Illinois
at Urbana-Champaign

P-KIMMO: A PROLOG IMPLEMENTATION OF THE TWO LEVEL MODEL

KANG-HYUK LEE

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

Technical Report No. LLL-T-18-91

P-KIMMO: A PROLOG IMPLEMENTATION OF THE TWO LEVEL MODEL

Kang-Hyuk Lee

Research Assistant, Language Learning Laboratory
Grac ate Student, Department of Linguistics

March, 1991

Available from: Language Learning Laboratory, University of Illinois at Urbana-Champaign, G70 Foreign Languages Building, 707 S. Mathews, Urbana, Illinois 61801

1. Introduction

This report describes a Prolog implementation of the two-level model for morphological analysis and synthesis developed by Kimmo Koskenniemi (1983). The two level model was originally implemented in Pascal by Koskenniemi himself. Karttunen and his students developed a LISP implementation, which was named "KIMMO" after its originator. Their work was published in "Texas Linguistic Forum" (1983). A Prolog implementation of the formalism was done by Boisen (1988). A comparison between the two Prolog implementations will be given in section 4. Quite recently, a C implementation by Antworth (1990) has been commercially available. Some testing results are also given in section 4.

The two-level system described in this report, which I will call "P-KIMMO" has been implemented in Quintus Prolog on a Sun Workstation, and exported to the Macintosh computer. An additional routine which harnesses P-KIMMO with a menu-driven user interface was written for the Macintosh version (see section. 5.2). The source code is listed in the appendix of this report. It can also be obtained as an ASCII file on a Macintosh formatted 3 1/2" disk. Requests should be sent to:

Kang-Hyuk Lee
Department of Linguistics
University of Illinois
4088 Foreign Languages Building
707 S. Mathews
Urbana, IL 61801

E-mail: klee@lees.cogsci.uiuc.edu

The current version of P-KIMMO has been integrated into the UNICORN natural language processing system (Gerdeman and Hinrichs 1988) as the morphological component. Research is ongoing to empower P-KIMMO to do morphological analysis with an on-line dictionary.

2. The Two-Level Formalism

Since the purpose of this report is to describe the implementational aspects of P-KIMMO, I will not attempt to provide a detailed description of the two-level formalism. Rather, I refer the reader to Koskenniemi (1983) for the full exposition of the formalism. Karttunen (1983) also is a valuable source. The description given in this section is intended for those who are not familiar with the two-level formalism so that they get the flavor of it.

As suggested in its nomenclature, the two-level model has two levels of morphophonological representations: the lexical level and surface level. These two levels are associated by morphophonological rules which specify legitimate pairs of characters, as illustrated in figure 2.1.

```
lexical:   s   p   y   +   s
                   |
           +----------------+
           |two-level rules |
           +----------------+
                   |
surface:   s   p   i   e   s
```

Figure 2.1

The general format of two level rules is given in figure 2.2. CP which stands for "correspondence" refers to a lexical/surface pair. LC and RC refer to the left and right environment, respectively. *OP* is a logical operator which is instantiated as <--> ("if and only if") in many cases.[1] What this operator says is that CP is obligatory in the given context and is possible only in that context. Note that LC and RC also are character pairs.

```
CP   *OP*   LC ___ RC
```

Figure 2.2: The general format of two-level rules

The two-level rule that legitimizes the y/i pairs (or, the y/i alternation in generative-phonological parlance) in figure 2.1 can be put in prose as follows (cited from Karttunen and Wittenburg 1983).

```
Y-replacement: y/i <--> C __ +/= - {i, a}

               After a consonant, lexical y corresponds to i
               when a lexical suffix marker and any pair
               other than i/i or a/a follows; to y elsewhere.
```

Figure 2.3

The capital "C" stands for all the consonant pairs. "+/=" abbreviates the pairs consisting of the suffix marker plus any character.[2] To sum up, two-level rules express *correspondences* between lexical and surface forms. This correspondence relation between two characters is a major departure from traditional generative phonology and characteristic of two-level rules.

3. Components of P-KIMMO

3.1. Lexicon

As in other implementations of two-level morphology, a lexicon is represented in the form of a letter tree in order to gain efficient lexical

[1] In Koskenniemi (1983: sec. 2.3.9), this operator is interpreted as the combination of the two operators, namely, --> and <-- which means "only if" and "if", respectively.

[2] These abbreviatory conventions make the two-level rules of a language and the corresponding finite state automata more compact and easy to read. See section 3.2.

access.[3] For example, an English lexicon that contains the words *be*, *beer*, *believe*, *big*, and *boy* is roughly represented as follows:

```
          e - r
         /
        e - l - i - e - v - e
       /
      b - i - g
       \
        o - y
```

Figure 3.1: The Lexical Tree

The last character of each word in the tree is associated with lexical entries. The *b-e-e-r* branch, for example, carries the entry for *be* at the *e* and the entry for *beer* at the *r*. *b* and the third *e* do not have any lexical specifications because *b* and *bee* are not words in this sample lexicon. In the current version of P-KIMMO, a lexical entry is a list consisting of a continuation class and a feature description. The empty list symbol [] is used for indicating characters devoid of lexical entries. Figure.3.2 shows the actual machine-readable format of the lexical tree in figure 3.1. Notice that the copular verb *be* has multiple entries.

```
[("b", [],
     [("e", [[#, "AUX"], [ivl, ""]],
         [("e", [],
             [("r", [[n, ""]], [])]),
          ("l", [],
             [("i", [],
                 [("e", [],
                     [("v", [],
                         [("e", [[v, ""]], [])])])])])]),
      ("i", [],
         [("g", [[a, ""]], [])]),
      ("o", [],
         [("y", [[n, ""]], [])])])]
```

Figure 3.2:
Pretty-printed list representation of the lexical tree

```
alternation(a, [ca, cs]).
alternation(ivl, [pr, i, ag, ab]).
```

Figure 3.3:
Possible expansions of the continuation classes "a" and "ivl"

The symbols *#*, *ivl*, *n, v, and a* are continuation classes which allow the recognizer to select the possible affixes. For example, the continuation class *a* has the comparative (ca) *+er* and superlative (cs) *+est* as its members, as shown in figure 3.3, which make possible to analyze words like *bigger* and *biggest*. #

[3] Although it is common practice to represent lexicons as lexical trees, it is controversial whether the tree representation is appropriate for modelling human performance. See Forster (1976) for some arguments against "lexical trees" from psycholinguistic perspectives.

indicates termination, so no continuation is permitted. This shows that the continuation of a lexical formative is specified in its lexical entry, and thus how morphotactics is described in the two-level formalism.

Although the tree format increases the efficiency of lexical access, it is very laborious and error-making to encode a lexical tree by hand, since it requires extremely careful arrangements of parentheses and brackets. Indentation for increasing readability would also be a tedious job. The problem would be much more serious if one wanted to build a large lexicon. It is almost impossible for the human eye to trace down the number of parentheses and brackets needed to properly enclose a big lexical tree. This bulkiness of the lexical tree also makes it difficult to augment the lexicon with new words.

As in KIMMO, a lexicon compiler has been added to the P-KIMMO system which automatically builds the corresponding lexical tree from an easy-formatted dictionary (called "EZ-Lexicon"). An EZ-Lexicon consists of Prolog clauses each of which contains a lexical string and information relevant to that lexical item (i.e. continuation class and features). The EZ-Lexicon corresponding to the English lexical tree above is given in figure 3.4.

```
lexicon(root, "be", [[#, "AUX"], [iv1, ""]]).
lexicon(root, "beer", [[n, ""]]).
lexicon(root, "believe", [[v, ""]]).
lexicon(root, "big", [[a, ""]]).
lexicon(root, "boy", [[n, ""]]).
```

Figure 3.4 A Sample EZ-Lexicon of English

The tree-building program reads an EZ-Lexicon file and writes the corresponding lexical tree to a designated output file. The lexical tree is also saved in the pretty-printed format as in figure 3.2.

There is another motivation for which the lexicon compiler has been developed. As mentioned in section 1, an effort is being made to augment P-KIMMO by making it capable of doing morphological analysis with an on-line dictionary. It is very unlikely, however, that on-line dictionaries are organized in such a way that P-KIMMO can make direct use of them. One way to make P-KIMMO run on such a dictionary would be to modify the recognition algorithm so that it could consult the original dictionary format. Above all, this obviously would lead to the loss of efficient lexical access. Since on-line dictionaries are usually huge in size, it is not hard to imagine that the process time would increase significantly. For this reason, the preprocessing of a dictionary--i.e. building the lexical tree from a dictionary--has been chosen to achieve the goal.

3.2. Rules As Finite State Automata

The most innovative feature of Koskenniemi's model is the use of finite state automata to encode two-level rules.[4] This is why two-level morphology is often referred to as "finite state morphology". The utilization of finite state automata explains why the processor is so efficient. It is well-known that finite state machines are computationally efficient and easy to implement. The finite state transducer behaves in exactly the same way as the ordinary

[4]Precisely speaking, finite state "transducers" in the sense that the input symbols are a pair of characters, rather than a single symbol.

finite automaton except that it reads a pair of input symbols. As shown in figure 3.4, a pair of characters is the input to the transducer, which is currently scanning the "yi" pair.

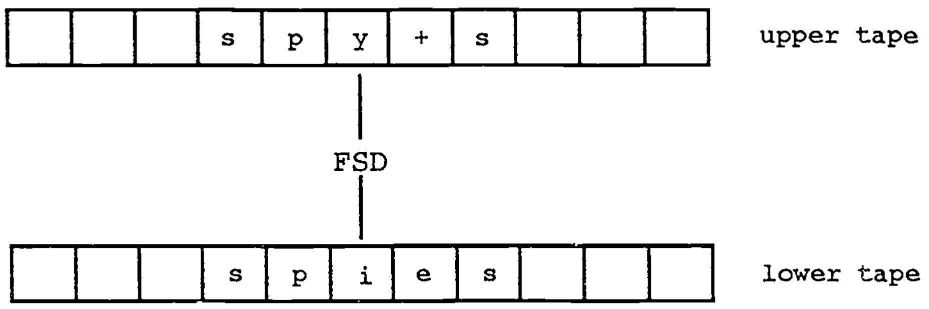


Figure 3.4

The English rule that expresses the y/i alternation can be depicted by a transition network diagram. Figure 3.5 is the graphical representation of the Y-spelling rule for English given in Karttunen and Wittenburg (1983).

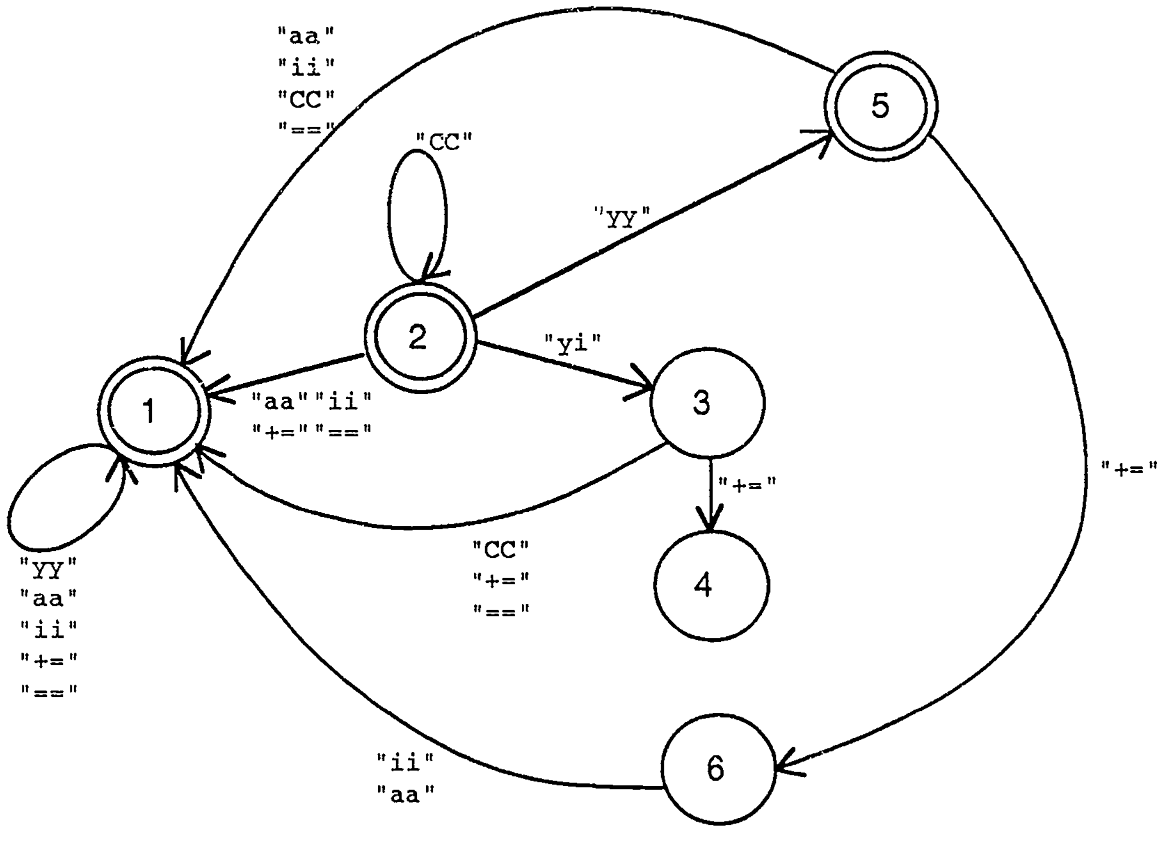


Figure 3.5

On the implementational side, finite state automata are represented as state transition tables. Figure 3.6 shows the tabular form of the Y-spelling rule. The full-fledged machine is presented in the appendix. The internal structure of the automata in P-KIMMO is somewhat different from the one in

KIMMO (i.e. the LISP version). It is also slightly different from Boisen's Pro-KIMMO.

```
[(y_spelling, [true,true,false,false,true,false]),
 ("CC", [2,2,0,1,1,0]),
 ("yy", [1,5,0,1,1,0]),
 ("yi", [0,3,0,0,0,0]),
 ("+=", [1,1,4,1,6,0]),
 ("ii", [1,1,0,0,1,1]),
 ("aa", [1,1,0,0,1,1]),
 ("==", [1,1,0,1,1,0])]
```

Figure 3.6: Y-spelling automaton for English

A transducer is encoded as a list whose member in turn is a LISP-like list consisting of a character pair and a transition list, except that the first member has the rule name (i.e. y_spelling) in the place of a character pair. "true" indicates a final state. The Y-spelling transducer has states 1, 2, and 5 as its final states. 0 means failure. An outgoing state is implicit in the sequential order of a state list, and an incoming state is represented as a member of the transition list. Taking the "yy" pair as an example, if the outgoing state is 2, the incoming state is set to state 5. The pair "CC" is an abbreviation for all the consonant pairs that are not "specified" in the automaton. Thus, the "CC" pair in the Y-spelling automaton stands for all the consonant pairs except for "yy". Theoretically, "CC" includes any possible combination of consonants such as "fv", "zs", "bp", and so on. The abbreviatory conventions are not interpreted that way, however. Their interpretation varies depending on the rules of a language. For example, the schematic pair "CC" includes a pair like "fv", only if it is "specified" in some other rule. The symbol "=" could be thought of as a wildcard which represents any characters. It roughly corresponds to "elsewhere condition" in phonological terms. Taking the Y-spelling rule again as an example, "==" stands for all the pairs other than the pairs specified and subsumed by more "specific" schemata (in this particular case, "+=" and "CC") in the automaton. This "specificity hierarchy" is another crucial factor to interpret two-level rules/automata. Since these notational conventions are very important to understand the two-level formalism, I refer the reader to Koskenniemi (1983) and Karttunen (1983) for detailed description.

As in Karttunen and Wittenburg (1983), each automaton in P-KIMMO has been hand-coded, which, of course, is very tedious. Along the lines of Koskenniemi (1985), a program that compiles automata directly from two-level rules is under development.

3.3. Compiler

Although the finite state automata described in the previous section are machine-readable, they are not the real data structures that the recognition/ generation algorithm of P-KIMMO runs off. In the LISP implementation of Gajek *et al.* (1983), finite state automata are compiled into two data structures called R-MACHINE and G-MACHINE which are used for recognition and generation, respectively. These data structures enable the recognizer and the generator to access finite state automata more efficiently. P-KIMMO basically adopts the same idea, but the data structures created by the Prolog version of the compiler are different from those presented in Gajek *et al.* In P-KIMMO, there is no distinction between R- and G-MACHINE. Rather, a single machine produced by the P-KIMMO compiler, which I call simply MACHINE, serves as both R- and G-MACHINE. The ability to use a single machine for both recognition and generation is due to Prolog's inherent aspect, namely, "reversibility".[5] The availability of a single machine would make the system more compact, since we do not have to maintain two data structures that are structurally identical except that one is accessed by the recognition algorithm and the other by the generation algorithm.

In P-KIMMO, MACHINE can be either asserted in the dynamic database or saved as a file by the compiler. Saving MACHINE as a database file would be a better choice if the user works with a complete set of automata. The recognition/generation algorithm runs much faster with a separate MACHINE file. This is due to the fact that using "pure" data structures generally increases the efficiency of a program. Adding MACHINE to the database would be useful if one still needs to debug the finite state machine he is working on. Otherwise, one has to save MACHINE each time he wants to test it. A fragment of MACHINE for English is given in figure 3.7.

```
machine("aa",[[(1,1)],
              [(1,1),(5,1),(6,1),(7,1)],
              [(1,2),(2,1),(3,1),(5,1),(7,1),(8,1),
               (11,1),(14,1),(15,1)],
              [(1,1),(2,1),(3,1),(4,1),(6,1)],
              [(1,4),(2,1),(4,16),(5,16),(6,16),(7,16),(8,16),(9,16),
               (10,16),(11,16),(12,16),(13,16),(14,16),(15,16),(16,16)],
              [(1,1),(2,1),(5,1),(6,1)]]).
machine("bb",[[(1,1)],
              [(1,1),(5,1),(6,1),(7,1)],
              [(1,5),(2,1),(3,5),(5,1),(7,1),(9,1),
               (11,1),(14,1),(15,1)],
              [(1,1),(2,1),(3,1),(4,1),(6,1)],
              [(1,1),(4,5),(5,16),(6,16),(7,16),(8,16),(9,16),(10,16),
               (11,16),(12,16),(13,16),(14,16),(15,16),(16,16)],
              [(1,2),(2,2),(4,1),(5,1)]]).
```

Figure 3.7

A *machine* clause carries two arguments: a pair of characters (lexical and surface in that order) and a list whose member is again a LISP-like list consisting of the outgoing and incoming state. Each state list of a machine clause specifies the possible states the current character pair could go through. Each machine clause in figure 3.7 has six state lists which correspond to six morphophonological rules of English (see Appendix). An important thing to note is that when the recognizer or generator is invoked, all the six state lists are checked to see if the character pair at hand is licensed by them. If any one of the state lists blocks the pair, then the process will fail. Although all the state lists (i.e. all the rules) are checked when MACHINE receives an input pair, this is done in a serial way (cf. Karttunen 1983, section 4.1). This is why Karttunen (1983) and Gajek *et al.* (1983) mention the merge of separate machines into a single finite state machine (called BIGMACHINE) which makes

[5]This non-deterministic programming technique is also closely related to the recognition/generation algorithm. See section 3.4 below.

the process more efficient. Karttunen (1983) gives an algorithm to merge transducers into a single equivalent transducer. The program merging transducers is also currently under development.

3.4. Recognizer and Generator

The recognition/generation algorithm is the workhorse of P-KIMMO. The reader interested in the algorithm itself is referred to Koskenniemi (1983) and Karttunen (1983), though they are not easy to follow at all. The recognition/generation algorithm of P-KIMMO is close to the one described in Karttunen (1983) in that it adopts the "depth-first" control strategy. In Koskenniemi's (1983) original implementation, the algorithm operates in the "breadth-first" manner. In this section, I will briefly mention a characteristic of the recognition/generation algorithm which distinguishes P-KIMMO from the previous implementations.

The previous implementations of the two-level model usually have two separate routines for recognition and generation. The main difference between the two is that the former is driven by the lexicon, whereas the latter is not. In P-KIMMO, there is no distinction between the recognition and generation algorithm. What this means is that a single algorithm serves for both recognition and generation. A single routine for both recognition and generation is the result of taking advantage of the non-deterministic programming technique of Prolog. In a nutshell, recognition is nothing more than the "reverse" mode of generation, and vice versa. A consequence of using a single algorithm is that the lexicon is also consulted during generation. This prevents the generator from accepting garbage inputs--i.e. non-words or combination of non-words.[6] If the generator is not guided by the lexicon, every garbage input would be accepted as long as it satisfied the morphophonological conditions of the language. In some domains of application, however, it is quite plausible that one wants to use the system to parse non-words. It would be especially useful when one wished to store unknown words to augment the existing lexicon. As a matter of fact, P-KIMMO includes a separate generation routine for these purposes, which makes P-KIMMO more flexible, although it is not listed in the appendices.

4. Evaluation of P-KIMMO

Implemented in Prolog, it is quite natural that P-KIMMO and Boisen's Pro-KIMMO have many things in common. For example, The lexical format of the two systems is strictly identical. The way to encode finite state automata is also very similar. Nevertheless, the two implementations are substantially different at least in one respect. This section briefly discusses the crucial difference which, I believe, renders P-KIMMO superior to Pro-KIMMO.

Although Boisen (1988) alludes to creating new data structures out of finite state automata, he does not spell out what type of data structures his recognition algorithm uses. It seems obvious, however, that his algorithm does not make use of data structures of the kind described in section 3.3. I strongly believe that this is why it took his recognizer more than a minute to

[6]By "non-words", I mean strings that are not listed in the dictionary.

analyze the Japanese word *kattemita*.[7] Surprisingly enough, P-KIMMO consumed only 0.3 second to recognize the same word, which is significantly fast, compared to Pro-KIMMO. Of course, the speed heavily depends on the computer used for the test. The CPU time consumed by P-KIMMO to recognize *kattemita* has been calculated on a Sun Workstation which is quite fast. However, the result of running P-KIMMO on a modest Macintosh SE still proves that Pro-KIMMO is painfully slow. It didn't take more than a couple of seconds to process the same word on an SE. The data structures are not the only factor that slows things down.[8] As a matter of fact, Boisen attributes Pro-KIMMO's inefficiency to possible continuation classes the recognizer has to go through.[9] Given the same complexity caused by continuation classes, however, the unrealistic speed of Pro-KIMMO should be explained otherwise. This is why I believe that everything else being equal, P-KIMMO's superiority over Pro-KIMMO is due to the optimized data structures.[10]

Quite recently, a C implementation of the two level model for personal computers, thus dubbed PC-KIMMO (version 1.0.3) has been made available. Roughly speaking, the system structure of PC-KIMMO is almost identical with P-KIMMO except that it was written in C. Thus, it is not possible to compare the two systems in terms of control strategy, data structures, and so on. I will only mention some timing results from testing the two systems. To test PC-KIMMO, the C source code was compiled on a Unix machine. The timing has been done on a Sun workstation. In the recognition mode, PC-KIMMO is slightly faster (but not always! For some test inputs like *dying* and *spies*, P-KIMMO was faster by 0.00x) than P-KIMMO by 0.0x second or 0.00x (x usually ranges from 1 to 5), while in the generation mode, P-KIMMO (unexpectedly) performs better by the same degree.

Given that P-KIMMO is a little slower (but not significantly) than PC-KIMMO in recognizing words, the question is whether there is a way to improve the recognition speed of P-KIMMO. Since Prolog lacks data types such as arrays in conventional programming languages, the consulting time of state transition tables grows linearly to the size of the tables. This goes against the spirit of the two-level model in which the complexity of rules does not have any significant effect on processing time (cf. Karttunen 1983). A lot of attempts have been made to improve this defect of Prolog by simulating data types such as hash tables (cf. O'Keefe 1990). I believe that further optimization of data structures (e.g. The conversion of state transition lists into Prolog terms would make it possible to pick the desired state transition immediately by its argument position.), which is currently being under study, could improve the performance of P-KIMMO. However, even with somewhat defective data structures, P-KIMMO is efficient enough to compete with rapid systems like PC-KIMMO.

---

[7]As in Boisen (1988), I assume the two-level description of Japanese in Alam (1983).

[8]There are several overheads for the two-level model in general. For example, the traversal of the lexical tree is futile in many cases. Consult Barton et al. (1985) for the general discussion of problems with the two-level formalism.

[9]See section 3.2.3 of Boisen (1988) for Prolog-related problems in implementing two-level morphology.

[10]Since Boisen (1988) lacks the description of the implementational aspects of his system, the comparison given here cannot be considered as empirical results.

5. Using P-KIMMO

5.1 Running P-KIMMO on the UNIX computer

Before running the recognizer and generator, the user needs to convert the finite state automata of a specific language into the suitable data structures (i.e. MACHINE). To compile automata, type "compile.". Then, the user is prompted to enter the file name that contains the finite state automata of the language the user has in mind.

```
|?- compile.

Input filename?
|: 'english.aut'.

Output filename?
|: 'machine.eng'.
```

figure 5.1

To call up the system, all the user has to do is just to type "kimmo." at the prompt (Be sure not to omit the period!). That command will automatically load all the relevant files. Then, the user is again prompted to enter the data files (i.e. the MACHINE and the lexicon) he wants to examine.

```
|?- kimmo.

Which machine?
|: 'machine.eng'.

Which dictionary?
|: 'english.lex'.


Welcome to P-KIMMO !!!

Copyright (C) 1991 by Kang-Hyuk Lee. All rights reserved.
```

Figure 5.2

Now, P-KIMMO is ready to run. Figure 5.3 and 5.4 show the sample inputs and outputs for recognition and generation, respectively.

```
| ?- recognize("dying").

Recognized string: die+ing
Categories:         [root,pr]
Feature(s):          V PROG

RECOGNITION TIME = 0.05 sec.

yes
```

```
| ?- recognize("died").

Recognized string: die+ed
Categories:         [root,ps]
Feature(s):          V PAST
Recognized string: die+ed
Categories:         [root,pp]
Feature(s):          V PAST PRT

RECOGNITION TIME = 0.117 sec.

yes
| ?- recognize("referring").

Recognized string: re`fer+ing
Categories:         [root,pr]
Feature(s):          V PROG

RECOGNITION TIME = 0.1 sec.

yes
```

Figure 5.3

```
| ?- generate("die+ing").

Generated String: dying

GENERATION TIME = 0.183 sec.

yes
| ?- generate("die+ed").

Generated String: died

GENERATION TIME = 0.033 sec.

yes
| ?- generate("re`fer+ing").

Generated String: referring

GENERATION TIME = 0.0669999 sec.

yes
```

Figure 5.4

When the recognizer is invoked, it displays the lexical string of the input, the category names of the elements involved during recognition, features of these categories, and finally the CPU time for recognition. For example, the surface string *dying* is analyzed as combining the root category *die* with the morpheme (i.e. "pr") *ing* ("+" indicates morpheme boundary). "V" and "PROG" stand for "verb" and "progressive", respectively. The generator only returns the surface string of the input and generation time.

As mentioned in the previous section, another mode of P-KIMMO is available which is suitable for the development of a new machine and dictionary. The tracing facility is also being implemented for debugging. This would allow the user to detect errors more easily.

5.2 Running P-KIMMO on the Macintosh computer

Though not a stand-alone application, the Macintosh version of P-KIMMO provides the user with a menu-driven interface. Upon opening the file named "MacKIMMO",[11] the user is given the screen with the Output Window in figure 5.1.

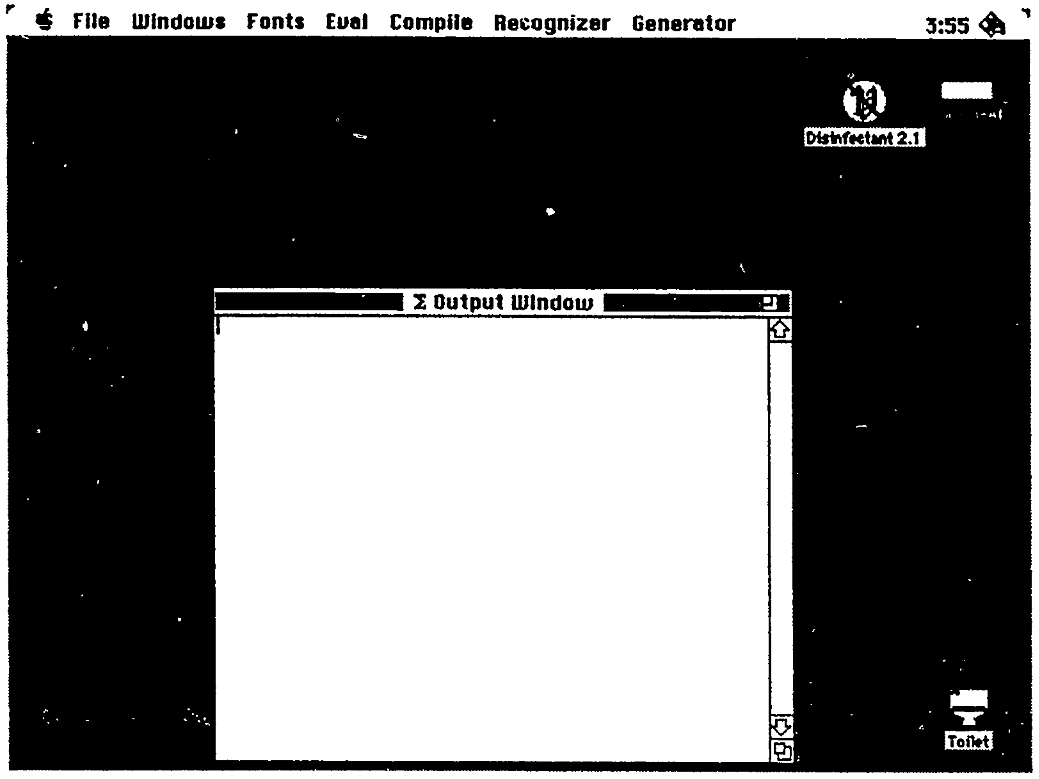


Figure 5.1

The last three menus in the menu bar--"Compile", "Recognizer", and "Generator"--are the ones created by MacKIMMO. The addition of these menus has led to the suppression of other built-in menus. First, click on the "Compile" pull-down menu which currently contains two data types--"Engdata" (for English) and "Nippondata" (for Japanese). Figure 5.2 shows that "Engdata" is selected from the "Compile" menu, which leads MacKIMMO to compile the English automata and load the English lexicon. After the compilation is done, a message will appear in the Output Window, as in figure 5.3.

[11]MacKIMMO runs on LPA Mac PROLOG 3.0.

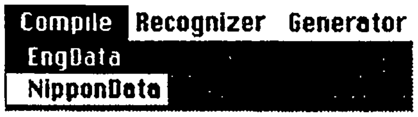
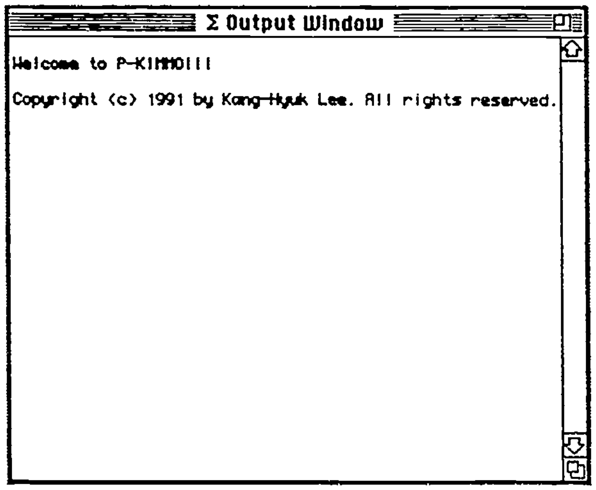


Figure 5.2



```
Welcome to P-KIMMO!!!
Copyright (c) 1991 by Kang-Hyuk Lee. All rights reserved.
```

Figure 5.3

MacKIMMO is now ready to run. To test the recognizer, select "Surface String..." from the "Recognizer" menu (Figure 5.4). Then the user is provided with a dialog box for entering a string. On typing a surface string followed by either the return key or a click on the OK button, the analyzed result is displayed in the Output Window, as illustrated in figure 5.5.

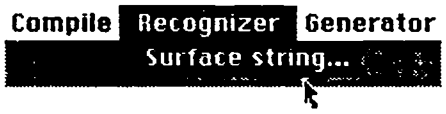
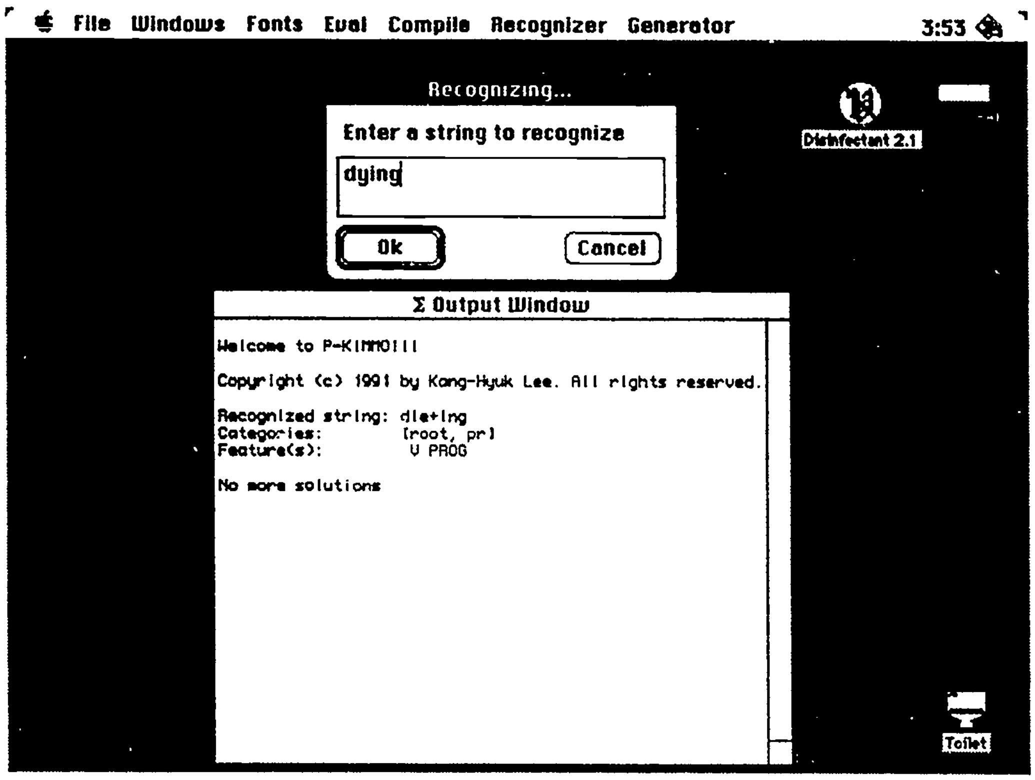


Figure 5.4

16



Enter a string to recognize

dying

Σ Output Window

```
Welcome to P-KIMMO!!!

Copyright (c) 1991 by Kong-Hyuk Lee. All rights reserved.

Recognized string: die+ing
Categories:        [root, pr]
Feature(s):         V PROG

No more solutions
```

Figure 5.5

The same goes for the generator. An illustration is given in figures 5.6 and 5.7. Note that the generated string is displayed above the previously recognized string.

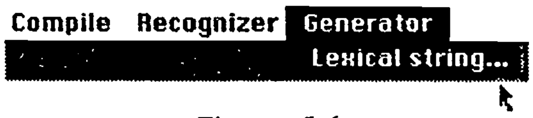
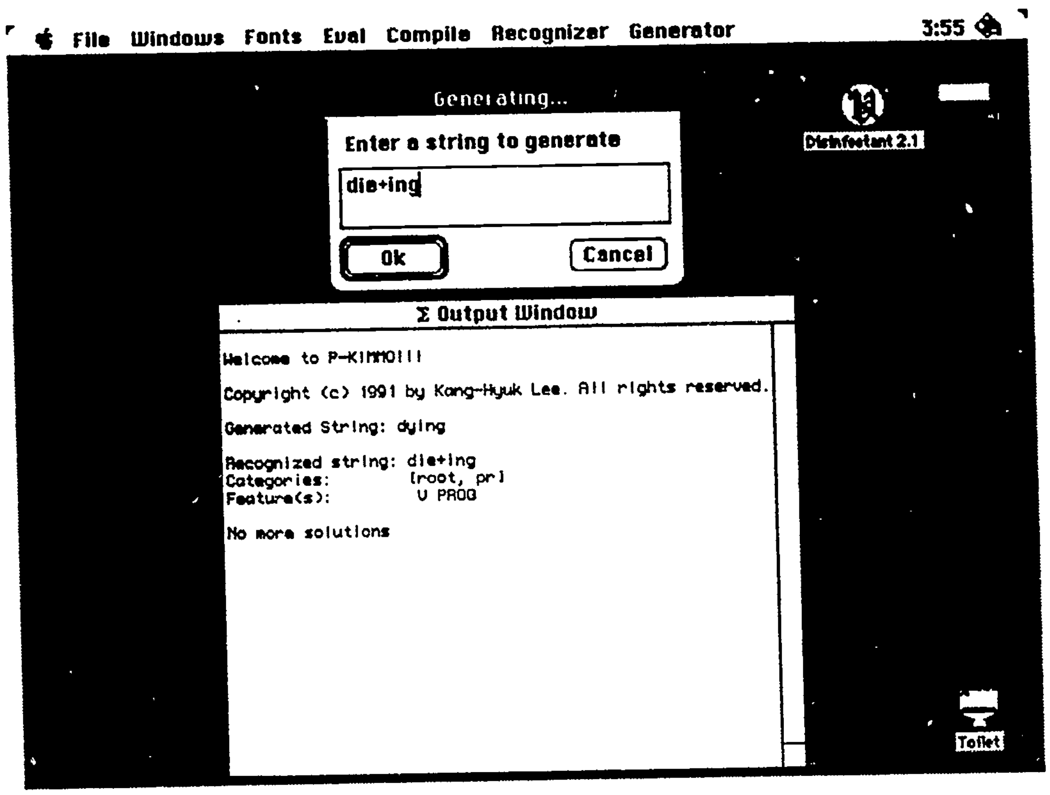


Figure 5.6

17


File Windows Fonts Eval Compile Recognizer Generator
3:55
Generating...
Enter a string to generate
die+ing
Ok
Cancel
Σ Output Window
Disinfectant 2.1
Toilet

```
Welcome to P-KIMMO!!!

Copyright (c) 1991 by Kang-Hyuk Lee. All rights reserved.

Generated String: dying

Recognized string: die+ing
Categories:        [root, pr]
Feature(s):         V PROG

No more solutions
```

Figure 5.7

References

Alam, Yukiko Sasaki (1983) "A two-level morphological analysis of Japanese", *Texas Linguistic Forum*, 22, 229-52.

Antworth, Evan L. (1990) *PC-KIMMO: A Two-level processor for morphological analysis*, Summer Institute of Linguistics:Dallas, Texas.

Barton, Edward, Robert C. Berwick and Eric Sven Ristad (1987) *Computational Complexity and Natural Language*, MIT Press: Cambridge, MA.

Boisen, Sean (1988) "Pro-KIMMO: A prolog implementation of two-level morphology", *Morphology as a Computational Problem*, UCLA Occasional Papers, 7, 31-53.

Forster, K. (1976) "Accessing the mental lexicon", in E.C.T. Walker and R.J.Wales (eds.), *New Approaches to Language Mechanisms*, 257-287, North-Holland, Amsterdam.

Gajek, Oliver, Hanno T. Beck, Diane Elder and Greg Whittemore (1983) "Kimmo Lisp implementation", *Texas Linguistic Forum*, 22, 187-202.

Gerdeman, Dale and Erhard Hinrichs (1988) "UNICORN: A unification parser for attribute-value grammars", *Studies in Linguistic Sciences*, 18(2), 41-86.

Karttunen, Lauri (1983) "KIMMO: A general morphological processor", *Texas Linguistic Forum*, 22, 165-186.

Karttunen, Lauri and Kent Wittenburg (1983) "A two-level morphological analysis of English", *Texas Linguistic Forum*, 22, 217-228.

Koskenniemi, Kimmo (1983) *Two-level morphology: a general computational model for word-form recognition and production*, University of Helsinki: Helsinki.

Koskenniemi, Kimmo (1985) "Compilation of automata from morphological two-level rules", *Papers from the Fifth Scandinavian Conference of Computational Linguistics*, 143-149.

O'Keefe, Richard A. (1990) *The Craft of Prolog*, MIT Press: Cambridge, MA.

APPENDIX

Source Listing of the P-KIMMO system

```
/*********************************************************************
 **
 **           "The Recognition/Generation Algorithm"
 **
 **
 **              Copyright (C) 1991 Kang-Hyuk Lee
 **                   All rights reserved.
 **
 *********************************************************************/

 %%% Unlike previous implementations, there is no algorithmic
 %%% distinction between recognition and generation in the P-KIMMO
 %%% system.  This is the result of taking advantage of non-
 %%% deterministic programming technique inherent in Prolog.


recognize(Surface):-
    statistics(runtime, _Now),
    findall([Lexical, Cat_List, Feature_List],
            transduce(Lexical, Surface, Cat_List, Feature_List),
            Analyses), nl,
    ( Analyses == [] -> (nl, write('No solution Available'), nl)
                  | write_results(Analyses) ),
    statistics(runtime, [_Total, Since]),
    Since1 is Since/1000,
    write('RECOGNITION TIME = '),
    write(Since1),
    write(' sec.'), nl.


%%% Note: Since transduce/4 is not embedded in findall/3, the
%%%       generator doesn't backtrack to see if there are more solutions
%%%       as the recognizer does.  To get all solutions, just put
%%%       transduce/4 into findall/3 as in recognize/1.
%%% -----------------------------------------------------------------
generate(Lexical):-
    statistics(runtime, _Now),
    transduce(Lexical, Surface, Cat_List, Feature_List),
    write('Generated String: '),
    write_output(Surface),
    statistics(runtime, [_Total, Since]),
    Since1 is Since/1000, nl,
    write('GENERATION TIME = '),
    write(Since1),
    write(' sec.'), nl.


is_final([], []).
is_final([Final|Rest], [FinalList|RestFinals]):-
    member(Final, FinalList),
    is_final(Rest, RestFinals).

final([Final|Rest]):-
    finality([FinalList|RestFinals]),
    is_final([Final|Rest], [FinalList|RestFinals]).
```

```
write_results([]).
write_results([[Lexical, Cat_List, Feature_List]|Rest_Cat_Features]):-
    write('Recognized String: '),
    write_output(Lexical), nl,
    write('Categories:        '),
    write(Cat_List), nl,
    write('Feature(s):        '),
    write_feature(Feature_List), nl, nl,
    write_results(Rest_Cat_Features).


write_output([]).
write_output([Char|Rest_Char]):-
    put(Char),
    write_output(Rest_Char).

write_feature([]).
write_feature([Feature|Rest]):-
    write_output(Feature), write(' '),
    write_feature(Rest).


%%% transduce(Lexical_String, Surface_String,
%%%           List_of_Categories, Features)
%%%
%%%      Features: Bundle of features pertaining to categories
%%%                involved in the input string
%%% ---------------------------------------------------------
transduce([Init_Char|Rest_Lex_Char], [Init_Char|Rest_Surf_Char],
          [Cat|Rest_Cat], Features):-
   initials(State),
   lexicon(Cat, [([Init_Char], Cont_Info, Rest_Char_and_Cont_Info)]),
   move_automata(State,
                 [Init_Char|Rest_Lex_Char],
                 [Init_Char|Rest_Surf_Char],
                 [([Init_Char], Cont_Info, Rest_Char_and_Cont_Info)],
                 Rest_Cat,
                 Features).


transduce(State, [], [], [Cont_Info|Rest_Cont_Info], [], [Info|[]]):-
    check_cont_list([Cont_Info|Rest_Cont_Info], [], Info, []),
    final(State).

transduce(State, [Lex_Char], [Surf_Char],
          [Cont_Info|Rest_Cont_Info], [Cat], [Info|Info2]):-
    check_cont_list([Cont_Info|Rest_Cont_Info], Cat, Info,
                    [([Lex_Char], Cont_Info2,Rest_Char_and_Cont_Info)]),
    find_arc([Lex_Char,Surf_Char], State, State2),
    transduce(State2, [], [], Cont_Info2, [], Info2).

transduce(State, [Lex_Char|Rest_Lex_Char], [],
          [Cont_Info|Rest_Cont_Info],
          [Cat|Rest_Cat], [Info|Info2]):-
    check_cont_list([Cont_Info|Rest_Cont_Info], Cat, Info,
                    [([Lex_Char], Cont_Info2, [])]),
    find_arc([Lex_Char,0], State, State2),
    transduce(State2, Rest_Lex_Char, [], Cont_Info2, Rest_Cat, Info2).
```

```
%%% transduce(Current_State, Lexical, Surface,
%%%           Lexicon, Categories, Features)
%%%
%%%      Lexicon: New lexical configuration to be used to process
%%%               the rest of the input string
%%% -----------------------------------------------------------
transduce(State,
          [Lex_Char|Rest_Lex_Char],
          [Surf_Char|Rest_Surf_Char],
          [Cont_Info|Rest_Cont_Info],
          [Cat|Rest_Cat],
          [Info|Info2]):-
    check_cont_list([Cont_Info|Rest_Cont_Info], Cat, Info,
                    [([Lex_Char], Cont_Info2,Rest_Char_and_Cont_Info)]),
    move_automata(State,
                  [Lex_Char|Rest_Lex_Char],
                  [Surf_Char|Rest_Surf_Char],
                  [([Lex_Char], Cont_Info2, Rest_Char_and_Cont_Info)],
                  Rest_Cat,
                  Info2).



%%% move_automata(Current_State, Lexical_String, Surface_String,
%%%               Lexicon, Category, Info).
%%%
%%%     Lexicon: current configuration of the lexicon
%%%     Category: list of categories
%%%     Info: Grammatical information
%%% -----------------------------------------------------------
move_automata(State1,
              [Lex_Char|Rest_Lex_Char],
              [Surf_Char|Rest_Surf_Char],
              Lexicon,
              Cat,
              Info):-
    lexmatch([Lex_Char], Entry,
             [([Lex_Char], Cont_Info, Rest_Char_and_Cont_Info)]),
    find_arc([Lex_Char,Surf_Char], State1, State2),

    % if Lex_Char does not have an entry
    (Cont_Info == []
     % process the next character pair
     -> move_automata(State2,
                      Rest_Lex_Char,
                      Rest_Surf_Char,
                      Rest_Char_and_Cont_Info,
                      Cat,
                      Info)
     % otherwise, i.e. if Lex_Char has an entry
       % either do transduce/6
     ; (transduce(State2, Rest_Lex_Char,
                  Rest_Surf_Char,
                  Cont_Info, Cat, Info);

       % if transduce/6 fails, go on to process the next pair.
       %
```

```
        % Even if a lexical character has an entry, that doesn't
        % necessarily mean that the lexical string scanned by that
        % time consititutes part of the input string.  For example,
        % consider the word "bite".  After scanning "t" which has an
        % entry, the parser will try to match the final character "e"
        % with some character by looking up in another lexical item,
        % which eventually fails.  Therefore, the parser needs to
        % backtrack in order to analyze "bite" as a single word.
         move_automata(State2,
                       Rest_Lex_Char,
                       Rest_Surf_Char,
                       Rest_Char_and_Cont_Info,
                       Cat,
                       Info))).

move_automata(State1,
              [Lex_Char|Rest_Lex_Char],
              [Surf_Char|Rest_Surf_Char],
              Entry,
              Cat,
              Info):-
    lexmatch([Lex_Char], Entry,
             [([Lex_Char], Cont_Info, Rest_Char_and_Cont_Info)]),
    find_arc([Lex_Char,0], State1, State2),  % e.g. "+0", "`0"
    (Cont_Info == []
     -> move_automata(State2,
                      Rest_Lex_Char,
                      [Surf_Char|Rest_Surf_Char],
                      Rest_Char_and_Cont_Info,
                      Cat,
                      Info)
     ; (transduce(State2, Rest_Lex_Char,
                 [Surf_Char|Rest_Surf_Char],
                 Cont_Info, Cat, Info);
        move_automata(State2,
                      Rest_Lex_Char,
                      [Surf_Char|Rest_Surf_Char],
                      Rest_Char_and_Cont_Info,
                      Cat,
                      Info))).


lexmatch(Lex_Char,
         [(Lex_Char,Cont_Info,Rest_Char_and_Cont_Info)|_],
         [(Lex_Char,Cont_Info,Rest_Char_and_Cont_Info)]).
lexmatch(Lex_Char,
         [_|Other_Lex_Char],
         [(Lex_Char,Cont_Info,Rest_Char_and_Cont_Info)]):-
    lexmatch(Lex_Char,
             Other_Lex_Char,
             [(Lex_Char,Cont_Info,Rest_Char_and_Cont_Info)]).
```

```
%%% Find transitions whose arc is labelled this pair
%%%
%%% find_arc(Pair, State_List1, State_List2)
%%%     State_List1: list of outgoing states
%%%     State_List2: list of incoming states
%%% -------------------------------------------
find_arc([Lex_Char,Surf_Char], [State1|Rest1], [State2|Rest2]):-
    machine([Lex_Char,Surf_Char], List_of_Lists),
    all_member([State1|Rest1], [State2|Rest2], List_of_Lists).


%%% all_member(SL1, SL2, LSL): finds a transition rule by rule
%%%      SL1: list of outgoing states
%%%      SL2: list of incoming states
%%%      LSL: list of state lists
%%%           The number of LSL corresponds to that of
%%%           morphological rules.
%%% -----------------------------------------------------
all_member([], [], []).
all_member([State1|Rest1], [State2|Rest2], [List|RestList]):-
    member((State1, State2), List),
    all_member(Rest1, Rest2, RestList).


check_cont_list([[Cont, Info]|_Rest_Cont], Cat, Info, []):-
    (Cont == #).     % end of string
check_cont_list([[Cont, Info]|_Rest_Cont], Cat, Info,
                [([Lex_Char], Cont_Info, Rest_Char_and_Cont_Info)]):-
    lexicon(Cat, [([Lex_Char], Cont_Info, Rest_Char_and_Cont_Info)]),
    check_cont(Cont, Cat).

%%% For a lexical items with mutiple entries
check_cont_list([_Cont|Rest_Cont], Cat, Info,
                [([Lex_Char], Cont_Info, Rest_Char_and_Cont_Info)]):-
    check_cont_list(Rest_Cont, Cat, Info,
                    [([Lex_Char], Cont_Info, Rest_Char_and_Cont_Info)]).

check_cont(Cont, Cat):-
    alternation(Cont, List_of_Alt),
    member(Cat, List_of_Alt).
```

```
/**********************************************************************
 **
 **                       "MACHINE Compiler"
 **
 **
 **                Copyright (C) 1991 Kang-Hyuk Lee
 **                      All rights reserved.
 **
 **
 **********************************************************************/

 %%% The compiler converts automata (i.e transition tables that encode
 %%% morphophonological rules) into different data structures, which
 %%% the recognition-generation routine utilizes.
 %%% The basic idea can be found in Gajek et al.(1983).
 %%% Unlike Gajek et al., this compiler does not generate g-machine and
 %%% r-machine.  The single machine serves as both g- and r-machine.


initial(1).


compile:-
    write('input file name? '),
    read(File1),
    compile(File1),
    write('output file name? '),
    read(File2),
    tell(File2),
    create_lit_pairs(All_Lit_Pairs),
    assert(all_lit_pairs(All_Lit_Pairs)),
    automata([[(RuleName, FinalStateList)|RestAutomaton]|RestAutomata]),
    create_initials([[(RuleName, FinalStateList)
                        |RestAutomaton]|RestAutomata],
                     Initials),
    write(initials(Initials)), write('.'), nl,
    compile_final([[(RuleName, FinalStateList)
                      |RestAutomaton]|RestAutomata],
                   FinalStateList, FinalList, AllFinals),
    write(finality(AllFinals)), write('.'), nl,
    change_automata([[(RuleName, FinalStateList)
                        |RestAutomaton]|RestAutomata],
                     [[(RuleName, FinalStateList)
                        |NewRestAutomaton]|NewRestAutomata]),
    make_machine([[(RuleName, FinalStateList)
                     |NewRestAutomaton]|NewRestAutomata],
                  [FirstPair|RestPairs]),
    assert_automata([FirstPair|RestPairs]),
    told.
```

```
%%% to create all possible character pairs of a given language
%%% They are used to instantiate schematic pairs such as "CC" and "=="
%%% -----------------------------------------------------------------
create_lit_pairs(All_Lit_Pairs):-
    alphabet(Alphabet),
    make_alphabet_pairs(Alphabet, Alphabet_Pairs),
    automata([Automaton|RestAutomata]),
    create_lit_pairs(RestAutomata, Rest_Pairs, Alphabet_Pairs),
    append(Alphabet_Pairs, Rest_Pairs, All_Lit_Pairs).


%%% to produce all the "xx" character pairs from the alphabet of
%%% a language
%%% -----------------------------------------------------------
make_alphabet_pairs([], []).
make_alphabet_pairs([[X]|Y], [[X,X]|Z]):-
    make_alphabet_pairs(Y,Z).

%%% to produce the character pairs that are not the "xx" type but
%%% specified in the automata
%%% ------------------------------------------------------------
create_lit_pairs([], [], Already_Pairs).
create_lit_pairs([[(RN,FSL)|Automaton]|RestAutomata],
                 Conc_Pairs, Alphabet_Pairs):-
    pick_pairs(Automaton, New_Pairs, Alphabet_Pairs),
    append(Alphabet_Pairs, New_Pairs, Already_Pairs),
    create_lit_pairs(RestAutomata, Rest_Pairs, Already_Pairs),
    append(New_Pairs, Rest_Pairs, Conc_Pairs).

pick_pairs([], [], Already_Pairs).
pick_pairs([(Lit_Pair, StateList)|Rest_Pairs],
           NewPairs, Already_Pairs):-
    % if the pair is a member of the list of pairs that was
    % created from the alphabet, skip it.
    member(Lit_Pair, Already_Pairs), !,
    pick_pairs(Rest_Pairs, NewPairs, Already_Pairs).
pick_pairs([([Lex,Surf], StateList)|Rest_Pairs],
           NewPairs, Already_Pairs):-
    % if the pair has either wildcard or a variable,
    % skip it.
    (((alphabet(any, [Lex]);
       alphabet(any, [Surf]));
       abbrev([Lex], Alphabet1));
       abbrev([Surf], Alphabet2)), !,
    pick_pairs(Rest_Pairs, NewPairs, Already_Pairs).
pick_pairs([(Lit_Pair, StateList)|Rest_Pairs],
           [NewPair|RestNewPairs], Already_Pairs):-
    % otherwise, add this pair to the list of character pairs.
    Lit_Pair = NewPair,
    pick_pairs(Rest_Pairs, RestNewPairs, Already_Pairs).


%%% to create the initial list [1,1,1,...] whose length corresponds
%%% to the number of automata
%%% ------------------------------------------------------------
create_initials([], []).
create_initials([Automaton|RestAutomata], [1|Rest]):-
    create_initials(RestAutomata, Rest).
```

27

```
%%% to produce the final states of each automaton
%%% ----------------------------------------
compile_final([], [], [], []).
compile_final([[(RuleName, FinalStateList)|RestAutomaton]|RestAutomata],
              FinalStateList, [State1|RestFinal], [Finals|RestFinals]):-
    initial(State1),
    compile_finality(FinalStateList,[State1|RestFinal], Finals),
    compile_final(RestAutomata, FinalStateList2,
                  Pos_Num_List, RestFinals).

compile_finality([State], [FinalState], [X]):-
    State == true,
    X = FinalState.
compile_finality([State], [FinalState], []):-
    State == false.
compile_finality([State|RestState],
                 [FinalState1|[FinalState2|RestFinal]],
                 [X|Y]):-
    FinalState2 is FinalState1 + 1,
    State == true,
    X = FinalState1,
    compile_finality(RestState,[FinalState2|RestFinal],Y).
compile_finality([State|RestState],
                 [FinalState1|[FinalState2|RestFinal]],
                 X):-
    FinalState2 is FinalState1 + 1,
    State == false,
    compile_finality(RestState,[FinalState2|RestFinal],X).



%%% Each automaton is processed by the recursive call of
%%% change_automata/2.  All the schematic pairs such as "==" are
%%% instantiated by change_automaton/2 which calls the relevant
%%% clause (elsewhere/3 or replace_variable/3) depending on the
%%% pair to be processed.
%%% ---------------------------------------------------------
change_automata([], []).
change_automata([[(RN, FSL)|RestAutomaton]|RestAutomata],
                [[(RN, FLS)|SortedAutomaton]|NewRestAutomata]):-
    change_automaton(RestAutomaton, NewRestAutomaton,
                     [(RN, FSL)|RestAutomaton]),
    mergesort(NewRestAutomaton, SortedAutomaton),
    change_automata(RestAutomata, NewRestAutomata).


%%% to produces the final data structures (i.e. MACHINE) that the
%%% recognition/generation algorithm runs off.
%%% The description of MACHINE is given in section 3.3.
%%% ---------------------------------------------------------
make_machine([[(RN, FSL)]|Rest_RN_And_FSL], []).
make_machine([[(RN, FSL)|[(Lit_Pair, X)|Rest]]|RestAutomata],
             [(Lit_Pair, [X|Y])|RestPairs]):-
    make_machine1([[(RN, FSL)|[(Lit_Pair, X)|Rest]]|RestAutomata],
                  (Lit_Pair, [X|Y])),
    remove_pair([[(RN, FSL)|[(Lit_Pair, X)|Rest]]|RestAutomata],
                [[(RN, FSL)|Rest]|NewRestAutomata]),
    make_machine([[(RN, FSL)|Rest]|NewRestAutomata], RestPairs).
```

```
make_machine1([], (Lit_Pair, [])).
make_machine1([[(RN, FSL)|[(Lit_Pair, X)|Rest]]|RestAutomata],
              (Lit_Pair, [X|Y])):-
    make_machine1(RestAutomata, (Lit_Pair, Y)).

remove_pair([],[]).
remove_pair([[(RN, FSL)|[(Lit_Pair, X)|Rest]]|RestAutomata],
          [[(RN, FSL)|Rest]|NewRestAutomata]):-
    remove_pair(RestAutomata, NewRestAutomata).

assert_automata([]).
assert_automata([(Lit_Pair, StateList)|RestPairs]):-
    write(machine(Lit_Pair, StateList)), write('.'), nl,
    assert_automata(RestPairs).


change_automaton([], [], Automaton).
change_automaton([("==", StateList)],
                 [(Else_Lit_Pair, Arcs)|RestArcs], Automaton):-
    elsewhere(("==", StateList), Automaton,
              [(Else_Lit_Pair, StateList)|RestPairs]),
    else_list_arcs([(Else_Lit_Pair, StateList)|RestPairs],
                   [(Else_Lit_Pair, Arcs)|RestArcs]),
    change_automaton([], [], Automaton).
change_automaton([(Var_Pair, StateList)|Rest], All, Automaton1):-
    replace_variable((Var_Pair, StateList), Automaton1,
                     [(Replaced_Pair, StateList)|RestPairs]),
    else_list_arcs([(Replaced_Pair, StateList)|RestPairs],
                   [(Replaced_Pair, Arcs)|RestArcs]),

    % to update the automaton to prevent the Elsewhere condition
    % from being applied to the instantiated literal pairs
    delete_pair(Automaton1, (Var_Pair, StateList), Automaton2),
    append(Automaton2, [(Replaced_Pair, StateList)|RestPairs],
           Automaton3),

    change_automaton(Rest, Rest_Lit_Pairs, Automaton3),
    conc([(Replaced_Pair, Arcs)|RestArcs], Rest_Lit_Pairs, All).

change_automaton([(Lit_Pair, [State2])|Rest],
                 [(Lit_Pair, [(State1,State2)])|RestArcs], Automaton):-
    initial(State1),
    change_automaton(Rest, RestArcs, Automaton).

change_automaton([(Lit_Pair, [State2|[State4|RestStates]])|Rest],
                 [(Lit_Pair, Arcs)|NewRest], Automaton):-
    initial(State1),
    list_arcs(State1, [State2|[State4|RestStates]], Arcs),
    change_automaton(Rest, NewRest, Automaton).


else_list_arcs([], []).
else_list_arcs([(Else_Lit_Pair, StateList)|Rest],
               [(Else_Lit_Pair, Arcs)|RestArcs]):-
    initial(State1),
    list_arcs(State1, StateList, Arcs),
    else_list_arcs(Rest, RestArcs).
```

```
%%% As described in section 3.2, an outgoing state is implicit in
%%% the sequential order of a state list.  This is what "State2 is
%%% State1 + 1" in list_acrs/3 is all about.
%%% ----------------------------------------------------------------
list_arcs(State1, [State2], [Arc]):-
    State2 =\= 0,
    Arc = (State1,State2).
list_arcs(State1, [State2], []):-
    State2 == 0.
list_arcs(State1, [State2|[State4|RestStates]], [Arc|RestArcs]):-
    State3 is State1 + 1,
    State2 =\= 0,
    Arc = (State1,State2),
    list_arcs(State3, [State4|RestStates], RestArcs).
list_arcs(State1, [State2|[State4|RestStates]], Arcs):-
    State3 is State1 + 1,
    % the incoming state is 0 (i.e. failure),
    % don't add this state pair to MACHINE.
    State2 == 0,
    list_arcs(State3, [State4|RestStates], Arcs).



%%% Instantiate the "==" pair
%%% -------------------------
elsewhere((Lit_Pair, StateList), [(RN,FSL)|Automaton],
    [(Else_Lit_Pair, StateList)|RestPairs]):-
    all_lit_pairs([X|Y]),
    assert(statelist(StateList)),
    remove_spec([X|Y], Automaton,
                [(Else_Lit_Pair, StateList)|RestPairs]),
    retract(statelist(StateList)).

%%% Replace a variable pair by the corresponding pairs
%%% e.g. "VV", "CC", "SS", etc.
%%% ------------------------------------------------
replace_variable(([Var,Var], StateList), Automaton,
                 [(Spec_Pair, StateList)|RestPairs]):-
    abbrev([Var], Alphabet),
    make_pairs(Alphabet, Alphabet_Pairs),
    assert(statelist(StateList)),
    remove_spec(Alphabet_Pairs, Automaton,
                [(Spec_Pair, StateList)|RestPairs]),
    retract(statelist(StateList)).

%%% e.g. "V=", "C=", etc.
%%% --------------------
replace_variable(([Var,Wild], StateList), Automaton,
                 [(Spec_Pair, StateList)|RestPairs]):-
    abbrev([Var], Alphabet),
    [Wild] == "=",
    all_lit_pairs([X|Y]),
    make_else_pairs2(Alphabet, [X|Y], Else_Pairs),
    assert(statelist(StateList)),
    remove_spec(Else_Pairs, Automaton,
                [(Spec_Pair, StateList)|RestPairs]),
    retract(statelist(StateList)).
```

```
%%% Constant-Any pair
%%% e.g. "+=", "t=", etc
%%% --------------------
replace_variable(([Cons,Var], StateList), Automaton,
                 [(Spec_Pair, StateList)|RestPairs]):-
    [Var] == "=",
    all_lit_pairs([X|Y]),
    make_else_pairs(Cons, [X|Y], Else_Pairs),
    assert(statelist(StateList)),
    remove_spec(Else_Pairs, Automaton,
                [(Spec_Pair, StateList)|RestPairs]),
    retract(statelist(StateList)).

%%% Variable-Constant pair
%%% ---------------------
replace_variable(([Var,Cons], StateList), Automaton,
                 [(Spec_Pair, StateList)|RestPairs]):-
    abbrev([Var], Alphabet),
    make_pairs(Alphabet, Alphabet_Pairs),
    make_else_pairs1(Cons, Alphabet_Pairs, Else_Pairs),
    assert(statelist(StateList)),
    remove_spec(Else_Pairs, Automaton,
                [(Spec_Pair, StateList)|RestPairs]),
    retract(statelist(StateList)).

%%% Constant-Variable pair
%%% e.g. "+C"
%%% ---------------------
replace_variable(([Cons,Var], StateList), Automaton,
                 [(Spec_Pair, StateList)|RestPairs]):-
    abbrev([Var], Alphabet),
    make_pairs(Alphabet, Alphabet_Pairs),
    make_else_pairs(Cons, Alphabet_Pairs, Else_Pairs),
    assert(statelist(StateList)),
    remove_spec(Else_Pairs, Automaton,
    [(Spec_Pair, StateList)|RestPairs]),
    retract(statelist(StateList)).


make_pairs([], []).
make_pairs([[X]|Y], [[X,X]|Z]):-
    make_pairs(Y, Z).

make_else_pairs(A, [], []).
make_else_pairs(A, [[X,Y]|Z], [Pair|RestPairs]):-
    A == X, !,     % in case "Lex" is a variable
    [X,Y] = Pair,
    make_else_pairs(A, Z, RestPairs).
make_else_pairs(A, [[X,Y]|Z], Pairs):-
    make_else_pairs(A, Z, Pairs).

make_else_pairs1(A, [], []).
make_else_pairs1(A, [[X,Y]|Z], [Pair|RestPairs]):-
    A == Y, !,     % in case "Surf" is a variable
    [X,Y] = Pair,
    make_else_pairs1(A, Z, RestPairs).
make_else_pairs1(A, [[X,Y]|Z], Pairs):-
    make_else_pairs1(A, Z, Pairs).
```

```
make_else_pairs2([], All, []).
make_else_pairs2([[A]|B], [X|Y], Else_Pairs):-
    make_else_pairs(A, [X|Y], Else_Pairs1),
    make_else_pairs2(B, [X|Y], Else_Pairs2),
    conc(Else_Pairs1, Else_Pairs2, Else_Pairs).


% The termination condition has been complicated a bit
% due to the non-correspondence of else pairs to all pairs
remove_spec([], Automaton, []).
remove_spec([X], Automaton, []):-
    is_specified(X, Automaton).
remove_spec([X], Automaton, [(Else_Lit_Pair, ElseStateList)]):-
    statelist(ElseStateList),
    X = Else_Lit_Pair,
    remove_spec([], Automaton, []).

% If the literal pair is specified in the automaton, do nothing.
% Otherwise, add the pair to the new automaton
remove_spec([X|Y], Automaton, ElsePairs):-
    is_specified(X, Automaton), !,
    remove_spec(Y, Automaton, ElsePairs).
remove_spec([X|Y], Automaton,
            [(Else_Lit_Pair, ElseStateList)|Rest]):-
    statelist(ElseStateList),
    X = Else_Lit_Pair,
    remove_spec(Y, Automaton, Rest).


% Think of it as the member/2 predicate
is_specified(Lit_Pair, [(Lit_Pair, StateList)|RestAutomaton]).
is_specified(X, [(Lit_Pair, StateList)|RestAutomaton]):-
    is_specified(X, RestAutomaton).
```

32

```
/*********************************************************************
 **
 **                         The English Lexicon
 **
 *********************************************************************/

alternation(in, [c1]).
alternation(v, [p3, ps, pp, pr, i, ag, ab]).
alternation(iv1, [pr, i, ag, ab]).
alternation(iv2, [p3, pr, i, ag, ab]).
alternation(a, [pa, ca, cs ,ly]).
alternation(#, []).
alternation(Cat, [Cat]).

lexicon(n, [0], [[c1, "N SG"]]).
lexicon(n, "+s", [[c2, "N PL"]]).
lexicon(mn, [0], [[c1, "MASS N"]]).
lexicon(c1, [0], [[#, ""]]).
lexicon(c1, "'s", [[#, "GEN"]]).
lexicon(c2, [0], [[#, ""]]).
lexicon(c2, "'", [[#, "GEN"]]).
lexicon(p3, "+s", [[#, "V PRES SG 3RD"]]).
lexicon(ip3, [0], [[#, "V PRES 3RD SING"]]).
lexicon(ps, "+ed", [[#, "V PAST"]]).
lexicon(ips, [0], [[#, "V PAST"]]).
lexicon(pp, "+ed", [[#, "V PAST PRT"]]).
lexicon(ipp, [0], [[#, "V PAST PRT"]]).
lexicon(pr, "+ing", [[#, "V PROG"]]).
lexicon(i, [0], [[#, "V"]]).
lexicon(ip1, [0], [[#, "V PRES SING, 1ST"]]).
lexicon(ag, "+er", [[n, "AG"]]).
lexicon(pa, [0], [[#, "A"]]).
lexicon(ca, "+er", [[#, "A COMP"]]).
lexicon(cs, "+est", [[#, "A SUP"]]).
lexicon(ly, "ly", [[#, "ADV"]]).
lexicon(ab, "+able", [[#, "VERB ABL"]]).

lexicon(root, "are", [[#, "V PRES SING 2ND"]]).
lexicon(root, "at", [[#, "PREP"]]).
lexicon(root, "a`ttack", [[n, ""], [v, ""]]).
lexicon(root, "be", [[#, "AUX"], [iv1, ""]]).
lexicon(root, "beer", [[n, ""]]).
lexicon(root, "believe", [[v, ""]]).
lexicon(root, "big", [[a, ""]]).
lexicon(root, "bit", [[ips, ""]]).
lexicon(root, "bite", [[iv2, ""]]).
lexicon(root, "bitten", [[ipp, ""]]).
lexicon(root, "boo", [[v, ""]]).
lexicon(root, "boy", [[n, ""]]).
lexicon(root, "cacti", [[in, "N PL"]]).
lexicon(root, "cactus", [[in, "N SG"]]).
lexicon(root, "cat", [[n, ""]]).
lexicon(root, "church", [[n, ""]]).
lexicon(root, "cool", [[a, ""]]).
lexicon(root, "day", [[n, ""]]).
lexicon(root, "did", [[ips, ""]]).
lexicon(root, "die", [[v, ""]]).
lexicon(root, "do", [[iv1, ""]]).
lexicon(root, "does", [[ip3, ""]]).
lexicon(root, "done", [[ipp, ""]]).
lexicon(root, "fox", [[n, ""]]).
lexicon(root, "go", [[iv1, ""]]).
lexicon(root, "goes", [[ip3, ""]]).
lexicon(root, "gone", [[ipp, ""]]).
lexicon(root, "grouch", [[n, ""]]).
lexicon(root, "had", [[#, "AUX"], [ips, ""]]).
lexicon(root, "has", [[#, "AUX"], [ip3, ""]]).
lexicon(root, "have", [[#, "AUX"], [iv1, ""]]).
lexicon(root, "ice", [[mn, ""]]).
lexicon(root, "industry", [[n, ""]]).
lexicon(root, "is", [[ip3, ""]]).
lexicon(root, "kill", [[v, ""]]).
lexicon(root, "kiss", [[n, ""],[v, ""]]).
lexicon(root, "mice", [[in, "N PL"]]).
lexicon(root, "milk", [[mn, ""]]).
lexicon(root, "mouse", [[in, "N SG"]]).
lexicon(root, "move", [[v, ""]]).
lexicon(root, "oc`cur", [[v, ""]]).
lexicon(root, "race", [[v, ""],[n, ""]]).
lexicon(root, "rally", [[n, ""]]).
lexicon(root, "refer`ee", [[v, ""],[n, ""]]).
lexicon(root, "re`fer", [[v, ""]]).
lexicon(root, "ski", [[n, ""]]).
lexicon(root, "sleep", [[iv2, ""]]).
lexicon(root, "slept", [[ipp, ""], [ips, ""]]).
lexicon(root, "spy", [[n, ""], [v, ""]]).
lexicon(root, "tie", [[v, ""]]).
lexicon(root, "tiptoe", [[v, ""]]).
lexicon(root, "toe", [[n, ""]]).
lexicon(root, "travel", [[n, ""], [v, ""]]).
lexicon(root, "try", [[v, ""]]).
lexicon(root, "un", [[root, "NEG"]]).
lexicon(root, "under`stand", [[iv2, ""]]).
lexicon(root, "understood", [[ipp, ""], [ips, ""]]).
lexicon(root, "undid", [[ips, ""]]).
lexicon(root, "undo", [[iv1, ""]]).
lexicon(root, "undoes", [[ip3, ""]]).
lexicon(root, "undone", [[ipp, ""]]).
lexicon(root, "untie", [[v, ""]]).
lexicon(root, "went", [[ipp, ""]]).
```

34

```
/*************************************************************************
 **
 **                        The English Automata
 **
 *************************************************************************/

 %%% This file contains the finite state automata which encode six
 %%% morphophonological rules in English, as described in Karttunen and
 %%% Wittenburg (1983).


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","'",[0]]).
alphabet(any, "=").
abbrev("V",["a","e","i","o","u"]).
abbrev("C",["b","c","d","f","g","h","j","k","l","m","n",
            "p","q","r","s","t","v","w","x","y","z"]).
abbrev("S",["s","x","z"]).


automata([[(surface, [true]),
           ([0,0], [1]),
           ("aa", [1]),
           ("bb", [1]),
           ("cc", [1]),
           ("dd", [1]),
           ("ee", [1]),
           ("ff", [1]),
           ("gg", [1]),
           ("hh", [1]),
           ("ii", [1]),
           ("jj", [1]),
           ("kk", [1]),
           ("ll", [1]),
           ("mm", [1]),
           ("nn", [1]),
           ("oo", [1]),
           ("pp", [1]),
           ("qq", [1]),
           ("rr", [1]),
           ("ss", [1]),
           ("tt", [1]),
           ("uu", [1]),
           ("vv", [1]),
           ("ww", [1]),
           ("xx", [1]),
           ("yy", [1]),
           ("zz", [1]),
           ("''", [1]),
           ("==", [1])],

          [(i_spelling, [true,false,false,false,true,true,false]),
           ("iy", [2,0,0,0,1,0,0]),
           ([101,0], [1,3,0,0,1,1,0]),
           ([43,0], [1,0,4,0,1,7,0]),
           ("ii", [5,0,0,1,1,0,0]),
           ("ee", [1,0,0,0,6,0,0]),
           ("==", [1,0,0,0,1,1,1])],
[(elision, [true,true,true,false,true,false,false,false,false,
            false,true,false,false,true,true]),
 ("VV", [2,1,1,0,1,0,1,1,0,0,1,0,0,1,1]),
 ("ii", [2,1,1,0,1,0,1,1,0,0,1,0,1,1,0]),
 ("ee", [3,5,5,0,1,0,0,1,0,1,14,0,1,1,0]),
 ([101,0], [4,6,6,0,4,0,0,0,0,0,12,0,0,0,0]),
 ([43,0], [1,1,9,8,7,10,0,0,0,0,1,13,0,15,1]),
 ("gg", [11,11,11,0,11,0,1,0,1,0,1,0,0,11,11]),
 ("cc", [11,11,11,0,11,0,1,0,1,0,1,0,0,11,11]),
 ("bb", [5,1,5,0,1,0,1,0,1,0,1,0,0,1,1]),
 ("==", [1,1,1,0,1,0,1,0,1,0,1,0,0,1,1])],

[(epenthesis, [true,true,true,true,false,true]),
 ("cc", [2,2,2,2,0,1]),
 ("hh", [1,3,1,3,0,1]),
 ("ss", [4,3,3,3,1,0]),
 ("SS", [3,3,3,3,0,1]),
 ("yi", [3,3,3,3,0,1]),
 ("+e", [0,0,5,5,0,1]),
 ([43,0], [1,1,6,6,0,1]),
 ("==", [1,1,1,1,0,1])],

[(gemination, [true,false,true,true,true,true,true,true,true,
               true,true,true,true,true,true,true]),
 ("VV", [4,1,0,16,16,16,16,16,16,16,16,16,16,16,16,16]),
 ("bb", [1,0,0,5,16,16,16,16,16,16,16,16,16,16,16,16]),
 ("dd", [1,0,0,6,16,16,16,16,16,16,16,16,16,16,16,16]),
 ("ff", [1,0,0,7,16,16,16,16,16,16,16,16,16,16,16,16]),
 ("gg", [1,0,0,8,16,16,16,16,16,16,16,16,16,16,16,16]),
 ("ll", [1,0,0,9,16,16,16,16,16,16,16,16,16,16,16,16]),
 ("mm", [1,0,0,10,16,16,16,16,16,16,16,16,16,16,16,16]),
 ("nn", [1,0,0,11,16,16,16,16,16,16,16,16,16,16,16,16]),
 ("pp", [1,0,0,12,16,16,16,16,16,16,16,16,16,16,16,16]),
 ("rr", [1,0,0,13,16,16,16,16,16,16,16,16,16,16,16,16]),
 ("ss", [1,0,1,14,16,16,16,16,16,16,16,16,16,16,16,16]),
 ("tt", [1,0,0,15,16,16,16,16,16,16,16,16,16,16,16,16]),
 ("+b", [0,0,0,0,2,0,0,0,0,0,0,0,0,0,0,0]),
 ("+d", [0,0,0,0,0,2,0,0,0,0,0,0,0,0,0,0]),
 ("+f", [0,0,0,0,0,0,2,0,0,0,0,0,0,0,0,0]),
 ("+g", [0,0,0,0,0,0,0,2,0,0,0,0,0,0,0,0]),
 ("+l", [0,0,0,0,0,0,0,0,2,0,0,0,0,0,0,0]),
 ("+m", [0,0,0,0,0,0,0,0,0,2,0,0,0,0,0,0]),
 ("+n", [0,0,0,0,0,0,0,0,0,0,2,0,0,0,0,0]),
 ("+p", [0,0,0,0,0,0,0,0,0,0,0,2,0,0,0,0]),
 ("+r", [0,0,0,0,0,0,0,0,0,0,0,0,2,0,0,0]),
 ("+s", [0,0,0,0,0,0,0,0,0,0,0,0,0,2,0,0]),
 ("+t", [0,0,0,0,0,0,0,0,0,0,0,0,0,0,2,0]),
 ([43,0], [1,0,0,1,3,3,3,3,3,3,3,3,3,3,3,16]),
 ([96,0], [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1]),
 ("==", [1,0,0,16,1,1,1,1,1,1,1,1,1,1,1,16])],
[(y_spelling, [true,true,false,false,true,false]),
 ("CC", [2,2,0,1,1,0]),
 ("yy", [1,5,0,1,1,0]),
 ("yi", [0,3,0,0,0,0]),
 ("+=", [1,1,4,1,6,0]),
 ("ii", [1,1,0,0,1,1]),
 ("aa", [1,1,0,0,1,1]),
 ("==", [1,1,0,1,1,0])]]).
```

```
/*********************************************************************
 **
 **                            Utilities
 **
 *********************************************************************/

 %%% This file contains some utility functions used by the compiler.


delete_pair([], _, []):- !.
delete_pair([Kill|Tail], Kill, Rest):-
    delete_pair(Tail, Kill, Rest).
delete_pair([Head|Tail], Kill, [Head|Rest]):-
    delete_pair(Tail, Kill, Rest).


%%% This sorting program is basically the same as in Shieber and
%%% Pereirra (1986) except that a few minor changes have been added
%%% to the "merge" clauses.
%%%
mergesort([], []).
mergesort([A], [A]).
mergesort([A,B|Rest], Sorted):-
    split([A,B|Rest], L1, L2),
    mergesort(L1, SortedL1),
    mergesort(L2, SortedL2),
    merge(SortedL1, SortedL2, Sorted).

split([], [], []).
split([A], [A], []).
split([A,B|Rest], [A|RestA], [B|RestB]):-
    split(Rest, RestA, RestB).

merge(A, [], A).
merge([], B, B).
merge([([Lex1,Surf1],S1)|RestAs], [([Lex2,Surf2],S2)|RestBs],
      [([Lex1,Surf1],S1)|Merged]):-
    Lex1+Surf1 < Lex2+Surf2,
    merge(RestAs, [([Lex2,Surf2],S2)|RestBs], Merged).
merge([([Lex1,Surf1],S1)|RestAs], [([Lex2,Surf2],S2)|RestBs],
      [([Lex2,Surf2],S2)|Merged]):-
    Lex2+Surf2 < Lex1+Surf1,
    merge([([Lex1,Surf1],S1)|RestAs], RestBs, Merged).
merge([([Lex1,Surf1],S1)|RestAs], [([Lex2,Surf2],S2)|RestBs],
      [([Lex1,Surf1],S1)|Merged]):-
    Lex2+Surf2 =:= Lex1+Surf1,
    Lex1 < Lex2,
    merge(RestAs, [([Lex2,Surf2],S2)|RestBs], Merged).
merge([([Lex1,Surf1],S1)|RestAs], [([Lex2,Surf2],S2)|RestBs],
      [([Lex2,Surf2],S2)|Merged]):-
    Lex2+Surf2 =:= Lex1+Surf1,
    Lex2 < Lex1,
    merge([([Lex1,Surf1],S1)|RestAs], RestBs, Merged).
```