

# Chapter 4

## Reference Manual

*Last modified November 22, 1995*

---

### **4.1 Introduction and technical specifications**

### **4.2 Installing PC-KIMMO**

### **4.3 Starting PC-KIMMO**

### **4.4 Interacting with the user interface**

### **4.5 Command reference by function**

### **4.6 Alphabetic list of commands**

### **4.7 File formats**

### **4.8 Trace formats**

### **4.9 Algorithms**

### **4.10 Messages**

---

## **4.1 Introduction and technical specifications**

PC-KIMMO is a new implementation for microcomputers of a program dubbed KIMMO after its inventor Kimmo Koskenniemi. Koskenniemi's two-level model was designed to generate words (see Koskenniemi 1983). Work on PC-KIMMO was begun in 1985, following the specifications of the LISP implementation of Koskenniemi's model described in Karttunen 1983. The aim was to develop a version of the two-level processor that would run on an IBM PC compatible computer and that would include an environment for testing and debugging a linguistic description. The PC-KIMMO program is actually a shell program that serves as an interactive user interface to the primitive PC-KIMMO functions. These functions are available as a source code library that can be included in a program written by the user.

The coding has been done in the C language by David Smith, Stephen McConnel, and Femke Hemels under the direction of Gary Simons and under the auspices of the Summer Institute of Linguistics. Every effort has been made to maintain portability. PC-KIMMO presently runs under four computing platforms:

- IBM PC and compatibles running MS-DOS or PC-DOS version 3.3 or higher and a 386 or higher CPU
- IBM PC and compatibles running Windows 3.0 or higher
- Apple Macintosh system 6.0 or higher
- UNIX System V (SCO UNIX V/386 and A/UX) and 4.2 BSD UNIX

The Windows and Macintosh versions use the same command-line interface as the DOS and UNIX versions, rather than using the graphical user interface one might expect. Also, the DOS and UNIX versions support a few commands which are not available in the Windows and Macintosh versions.

The release diskettes for the PC and Macintosh versions contain the executable PC-KIMMO program, examples of language descriptions, and the source code library for the primitive PC-KIMMO functions. The UNIX diskettes contain the complete source code which the user must compile.

The PC-KIMMO executable program and source code are copyrighted but are made freely available to the general public under the condition that they not be resold or used for commercial purposes.

The English description referred to in this chapter described in chapter 3.

## 4.2 Installing PC-KIMMO

---

### 4.2.1 Installing the MS-DOS version

### 4.2.2 Installing the Windows version

### 4.2.3 Installing the Macintosh version

### 4.2.4 Installing the UNIX version

---

The following instructions describe how to install the various versions of PC-KIMMO.

### 4.2.1 Installing the MS-DOS version of PC-KIMMO

If your computer has floppy disks only, make a working copy of the PC-KIMMO release diskette that came with this book. Store the original in a safe place. Insert your working copy of the PC-KIMMO diskette in drive A of your machine.

If your computer has a hard disk, use the INSTALL.BAT procedure on the PC-KIMMO diskette to install the system on your hard disk. To do this, insert the PC-KIMMO diskette in one of your disk drives. Type A: (or whatever the name of the drive is) in order to log control to that disk. Now type *install* followed by the name of the hard disk on which you want to install PC-KIMMO (for instance, *install C:*). This will create a subdirectory called PCKIMMO on your hard disk and copy the contents of the release diskette (with all its subdirectories) into it.

Whether you are using a floppy or hard disk system, the operating system's PATH variable must be set to include the directory where the PC-KIMMO program is found. The AUTOEXEC.BAT file on your boot disk should contain a path statement that specifies all the disks and directories that contain programs. On a floppy disk system, the path statement should include as a minimum the root directory of drive A, for instance, `PATH=A:\.` On a hard disk system, add `;\C:\PCKIMMO` to the end of the path statement. For the path statement to become effective, you must reboot the computer. (If you want to change the path variable without changing the AUTOEXEC.BAT file and rebooting, enter a path command directly at the operating system prompt.)

In order to use PC-KIMMO's *edit* command, you must set the operating system environment variable EDITOR to the name of your text editing program. This is done by including in the AUTOEXEC.BAT file a line of this form:

`SET EDITOR=filespec` where *filespec* specifies the path and full file name of your editing program. For example, if your editor's file name is EMACS.EXE and is found in the UTIL subdirectory directly under the root directory, include this line: `SET EDITOR=\\UTIL\\EMACS.EXE`

## 4.2.2 Installing the Windows version of PC-KIMMO

Follow the instructions above on installing the MS-DOS version from diskettes. Then run Windows and create a program item for PC-KIMMO.

## 4.2.3 Installing the Macintosh version of PC-KIMMO

Create a new folder named PC-KIMMO (or a name of your choice) on your hard disk and copy into it all the files and folders from the release diskette.

If you get an "Out of memory" error when you try to load a large lexicon, then you need to increase the amount of memory allocated to PC-KIMMO. To do this, click once on its icon or file name to select it, choose Get Info on the File menu, and type a larger number into the memory box. For example, Englex (the description of English described in chapter 3) requires a memory allocation of at least 3000KB.

## 4.2.4 Installing the UNIX version of PC-KIMMO

# 4.3 Starting PC-KIMMO

---

### 4.3.1 Starting the MS-DOS version

### 4.3.2 Starting the Windows version

### 4.3.3 Starting the Macintosh version

### 4.3.4 Starting the UNIX version

### 4.3.5 Starting PC-KIMMO with command line arguments

### 4.3.6 Starting PC-KIMMO with default settings

---

The following instructions describe how to start the various versions of PC-KIMMO.

## 4.3.1 Starting the MS-DOS version of PC-KIMMO

To start the MS-DOS version of PC-KIMMO on a PC, first be sure that DOS is logged onto the drive where PC-KIMMO is located. To change to the subdirectory that contains the English example, enter `cd \english` on a floppy disk system, or `cd \pckimmo\english` on a hard disk system. Now type `pckimmo` (if your PATH variable is not correctly set to include the PC-KIMMO subdirectory, type `..\pckimmo`). When PC-KIMMO has successfully started up, you will see a version message and the PC-KIMMO command line prompt.

## 4.3.2 Starting the Windows version of PC-KIMMO

To start the Windows version of PC-KIMMO on a PC, double-click on the PC-KIMMO program icon.

## 4.3.3 Starting the Macintosh version of PC-KIMMO

To start PC-KIMMO on a Macintosh, double-click on the PC-KIMMO program icon.

## 4.3.4 Starting the UNIX version of PC-KIMMO

## 4.3.5 Starting PC-KIMMO with command line arguments

The MS-DOS and UNIX versions of PC-KIMMO can also be started with optional command line arguments.

The format of the command line is:

```
pckimmo [-r rulefile] [-l lexiconfile] [-g grammarfile] [-t takefile]
```

The options are used as follows:

- The *-r* option specifies a rules file to be loaded. It is equivalent to issuing the *load rules* command from the program prompt.
- The *-l* option specifies a lexicon file to be loaded. It is equivalent to issuing the *load lexicon* command from the program prompt. It must be used with the *-r* option.
- The *-s* option specifies a synthesis-lexicon file to be loaded. It is equivalent to issuing the *load synthesis-lexicon* command from the program prompt. It must be used with the *-r* option.
- The *-g* option specifies a grammar file to be loaded. It is equivalent to issuing the *load grammar* command from the program prompt.
- The *-t* option specifies a "take" file from which PC-KIMMO reads and executes commands. It is equivalent to issuing the *take* command from the program prompt.

## 4.3.6 Starting PC-KIMMO with default settings

On start-up, PC-KIMMO automatically tries to load user-defined settings from a "take" file named PCKIMMO.TAK or PC-KIMMO.TAK. Command line arguments are executed after loading the default "take" file.

# 4.4 Interacting with the user interface

---

### 4.4.1 Entering commands

### 4.4.2 Getting on-line help

### 4.4.3 The MS-DOS interface

### 4.4.4 The Windows interface

### 4.4.5 The Macintosh interface

### 4.4.6 The UNIX interface

---

## 4.4.1 Entering commands

The user interacts with PC-KIMMO by entering commands at the command line prompt, in much the same way that one enters commands at the operating system prompt. Case is ignored for all command keywords.

Keywords can be shortened to any unambiguous form. For instance, *load rules*, *load rul*, *load r*, and *loa r* are all acceptable. Typing just *l* is ambiguous for the commands *load*, *log*, and *list*. However, because *load* is such a frequently used command, it takes special precedence over the other commands beginning with *l*, which means that typing just *l* will execute only the *load* command.

## 4.4.2 Getting on-line help

There are several ways to get on-line help:

- To get a list of the available commands, type *?*.
- To get information on what these commands do, type *help*.
- To get the specific syntax and use for a command, type *help* plus a specific command name.
- To get a list of the keywords that can go with a particular command, type the command name followed by *?*. Note however that if the command does not take a keyword it will be executed; for instance typing *new ?* will execute the *new* command.

## 4.4.3 The MS-DOS interface

Screen scrolling can be halted by pressing *Ctrl-S* (that is, hold down the *Ctrl* (Control) key and press *S*); any key will resume scrolling.

Processing can be interrupted by pressing *Ctrl-C*. Note that this action does not abort PC-KIMMO, but returns it to the program prompt. It is useful for stopping a long screen display (such as a trace) or a file processing command.

Pressing *Ctrl-P* causes screen output to be echoed to the printer. Pressing *Ctrl-P* again stops printer echoing.

## 4.4.4 The Windows interface

The PC-KIMMO window can be moved, resized, and scrolled in the usual way (see your Windows documentation).

Anything in the PC-KIMMO window can be copied and pasted. This is done by using the mouse to highlight material on the screen and then executing the Copy and Paste commands available either on the Edit menu. For example, a command can be copied and pasted after the command line prompt.

Screen scrolling can be halted by pressing *Ctrl-S* (that is, hold down the *Ctrl* (Control) key and press *S*); press *Ctrl-Q* to resume scrolling.

Pressing *Ctrl-C* is equivalent to the *exit* command on the Windows File menu.

## 4.4.5 The Macintosh interface

The PC-KIMMO program window can be moved, resized, and scrolled in the usual way (see your Macintosh documentation).

The File, Edit, Font, and Size menus provide standard Macintosh commands. No PC-KIMMO commands (except quitting the program) can be done using menus. Note that the Font and Size commands affect the entire contents of the window.

Anything in the PC-KIMMO program window can be copied and pasted. This is done by using the mouse to highlight material on the screen and then executing the Copy and Paste commands available either on the Edit menu or with the Command keys (see your Macintosh documentation). For example, a command can be

copied and pasted after the command line prompt.

The command line can be edited using the mouse, the arrow keys, and the *Delete* key.

See section 4.5.18 for information on Macintosh directory paths and the CD command.

To interrupt an operation (such as a long trace display or a file reading/writing function), press *Command-*. (that is, hold down the Command key and press the period key). This is the same as *Ctrl-C* on the PC version.

To exit PC-KIMMO, either press *Command-Q* or choose Quit from the File menu. You can also enter either of the PC-KIMMO commands *exit* or *quit*. (*Ctrl-D* also works.)

## **4.4.6 The UNIX interface**

# **4.5 Command reference by function**

---

### **4.5.1 Get help**

### **4.5.2 Load rules, lexicon, and grammar**

### **4.5.3 Clear rules, lexicon, and grammar**

### **4.5.4 Take commands from a file**

### **4.5.5 List rule names, feasible pairs, or sublexicon names**

### **4.5.6 Set system options**

### **4.5.7 Turn logging on or off**

### **4.5.8 Show system status**

### **4.5.9 Show rule or sublexicon**

### **4.5.10 Generate surface forms from a lexical form**

### **4.5.11 Recognize lexical forms from a surface form**

### **4.5.11A Synthesize surface forms from a morphological form**

### **4.5.12 Compare data from a file**

### **4.5.13 Generate forms from a file**

### **4.5.14 Recognize forms from a file**

### **4.5.14A Synthesize forms from a file**

### **4.5.15 Execute an operating system command**

### **4.5.16 Edit a file**

## 4.5.17 Halt the program

## 4.5.18 Change directory

---

The following subsections document each command, arranged by function, of the PC-KIMMO system. Square brackets in the command line summaries indicate optional elements. The notation  $\{x / y\}$  means either  $x$  or  $y$  (but not both). Command keywords and arguments in boldface are typed literally; for instance, the command summary **set tracing** {**on** | **off**} means to type either *set tracing on* or *set tracing off*. Command arguments in italics are replaced by elements of the specified type; for instance, the command summary **show rule** *number* means to replace *number* with a rule number, such as *show rule 3*.

### 4.5.1 Get help

**?**

Displays a list of command names.

**help** [*command-name*]

Issuing the help command with no argument displays a list of commands with a brief description of their function. Issuing the *help* command with the name of a specific command displays a usage summary for the command.

*command-name* ?

Typing a command name followed by *?*, instead of a keyword, displays a message listing the keywords expected for that command.

### 4.5.2 Load rules, lexicon, and grammar

The *load* command is used to load either rules, a lexicon, or a word grammar from a file.

**load rules** [*filespec*]

The *load rules* loads a set of rules from the file specified on the command line. The *filespec* can contain a path; for example, B:\ENGLISH\ENGLISH.RUL is a fully specified path under MS-DOS (see section 4.5.18 for information on Macintosh directory paths). The default file name extension is .RUL; thus, the command *load rules english* will load the file ENGLISH.RUL. If no file name is given, the default file name RULES.RUL is used. The rules file must be in the format described later in this chapter (see section 4.7.1).

An error in the format of the rules file will cause the program to stop loading the file, erase the rules already loaded, and report an error message with the line number where the error was encountered. Refer to section 4.10.2 on error messages for more details.

The rules file must be loaded before the lexicon and before performing any generation or recognition operations.

The *load rules* command can also be invoked by using the *-r* command line option when starting up PC-KIMMO (see section 4.3.6).

**load lexicon** [*filespec*]

The *load lexicon* command loads a lexicon from the file specified in the command line. The *filespec* can contain a path; for example, B:\ENGLISH\ENGLISH.LEX is a fully specified path under MS-DOS (see section 4.5.18 for information on Macintosh directory paths). The default file name extension is .LEX; thus,

the command *load lexicon english* will load the file ENGLISH.LEX. If no file name is given, the default file name LEXICON.LEX is used. The lexicon file must be in the format described later in this chapter (see section 4.7.2).

An error in the format of the lexicon file will cause the program to stop loading the file, erase the parts of the lexicon already loaded, and report an error message with the line number where the error was encountered. Refer to section 4.10.3 on error messages for more details.

The rules file must be loaded before the lexicon. The lexicon file must be loaded before using the recognizer function. The generator function can be used without loading a lexicon and indeed makes no use of a lexicon.

The *load lexicon* command can also be invoked by using the *-l* command line option when starting up PC-KIMMO (see section 4.3.6).

### **load synthesis-lexicon** [*filespec*]

The *load synthesis-lexicon* command loads a synthesis-lexicon from the file specified in the command line. The *filespec* can contain a path; for example, B:\ENGLISH\ENGLISH.LEX is a fully specified path under MS-DOS (see section 4.5.18 for information on Macintosh directory paths). The default file name extension is .LEX; thus, the command *load synthesis-lexicon english* will load the file ENGLISH.LEX. If no file name is given, the default file name LEXICON.LEX is used. The synthesis-lexicon file must be in the format described later in this chapter (see section 4.7.2).

An error in the format of the synthesis-lexicon file will cause the program to stop loading the file, erase the parts of the synthesis-lexicon already loaded, and report an error message with the line number where the error was encountered. Refer to section 4.10.3 on error messages for more details.

The rules file must be loaded before the synthesis-lexicon. The synthesis-lexicon file must be loaded before using the synthesizer function. The generator function can be used without loading a synthesis-lexicon and indeed makes no use of a synthesis-lexicon.

The *load synthesis-lexicon* command can also be invoked by using the *-l* command line option when starting up PC-KIMMO (see section 4.3.6).

### **load grammar** [*filespec*]

The *load grammar* command loads a grammar from the file specified in the command line. The *filespec* can contain a path; for example, B:\ENGLISH\ENGLISH.RUL is a fully specified path under MS-DOS (see section 4.5.18 for information on Macintosh directory paths). The default file name extension is .GRM; thus, the command *load grammar english* will load the file ENGLISH.GRM. If no file name is given, the default file name GRAMMAR.GRM is used. The grammar file must be in the format described later in this chapter (see section 4.7.3).

An error in the format of the grammar file will cause the program to stop loading the file, erase the parts of the grammar already loaded, and report an error message with the line number where the error was encountered. Refer to section 4.10.4 on error messages for more details.

The rules and lexicon files must be loaded before the grammar. Use of a grammar file is optional, even when using the recognizer function. The generator function can be used without loading a grammar and indeed makes no use of a grammar file.

The *load grammar* command can also be invoked by using the *-g* command line option when starting up PC-KIMMO (see section 4.3.6).

## **4.5.3 Clear rules, lexicon, and grammar**

**clear**



The *clear* command erases from memory the rules, lexicon, synthesis-lexicon, and grammar currently loaded. Strictly speaking it is not needed, since the *load rules* command erases all existing rules, the *load lexicon* command erases any existing lexicon, the *load synthesis-lexicon* command erases any existing synthesis-lexicon, and the *load grammar* command erases any existing grammar.

## 4.5.4 Take commands from a file

**take** [*filespec*]

The *take* command causes PC-KIMMO to read and execute commands from a file. The *filespec* can contain a path; for example, B:\KIMMO\ENGLISH.TAK is a fully specified path under MS-DOS (see section 4.5.18 for information on Macintosh directory paths). The *take* command recognizes the default file name PCKIMMO.TAK and the default file extension .TAK. The command file can itself issue the *take* command to call another command file down to a depth of three files. That is, the user can specify a command file <file1> that contains the command *take* <file2>, that itself contains the command *take* <file3>. It would be an error for <file3> to contain a *take* command.

On start-up, PC-KIMMO automatically tries to load default settings and commands from a "take" file named PCKIMMO.TAK or PC-KIMMO.TAK.

A command file can also be specified by using the *-t* command line option when starting up PC-KIMMO (see section 4.3.6). Note that a command file cannot submit forms to the special generator and recognizer prompts (see sections 4.5.10 and 4.5.11).

## 4.5.5 List rule names, feasible pairs, or sublexicon names

The *list* command is used to display either rule names, feasible pairs, or sublexicon names.

**list pairs**

The *list pairs* command displays on the screen the set of feasible pairs specified by the set of rules currently turned on.

**list rules**

The *list rules* command displays on the screen the current state of the rules that are loaded. The display consists of each rule by number, an indication of whether the rule is on or off, and the rule name from the header lines of its state table in the rules file.

**list lexicon**

The *list lexicon* command displays on the screen the names of the sublexicons of the lexicon currently in use.

## 4.5.6 Set system options

The *set* command is used to turn on or off various processing options including several related only to using a word grammar. The *save* command saves the current settings to a "take" file.

### 4.5.6.1 General options

**set alignment** {on | off}

The *set alignment* command turns alignment display mode on or off. If alignment mode is *on*, then the results of the recognizer function are displayed on the screen in a vertically aligned format: the first (top line) displays the lexical form of each morpheme; the second line displays sublexicon names; and the third line displays glosses. If alignment mode is *off*, the results are displayed the usual way. The default setting is *off*.

### **set limit {on | off}**

The *set limit* command limits the result of a generation or recognition function to one form. That is, if limit is set *off*, then PC-KIMMO backtracks after finding a correct result so that it can find every possible result. With limit set *on*, after finding one correct result form PC-KIMMO does not backtrack to try to find more results. The default setting is *off*.

### **set rules {on | off} {list-of-numbers | all}**

The *set rules* command allows you to turn selected rules on or off for testing or debugging purposes. When a rule is turned off, it is completely ignored in the recognition or generation of forms. One effect of this is to cause the recalculation of feasible pairs, considering only the rules which remain on. Use the *list pairs* command to see the set of feasible pairs currently in use.

On the command line, you can specify the action *on* or *off* followed by a list of rule numbers or the keyword *all* (in which case all rules are turned on or off). Specific rules are turned on or off by listing their rule numbers (shown by the *list rules* command), each separated by a space.

### **set timing {on | off}**

The *set timing* command uses the computer's system clock to time the execution of generation and recognition operations. It displays the result as the number of seconds the operation lasted. It applies to these commands: *generate*, *recognize*, *file compare generate*, *file compare recognize*, *file compare pairs*, *file generate*, and *file recognize*. The default setting is *off*.

### **set tracing {on | off | level}**

The *set tracing* command allows you turn the tracing mechanism on or off. When tracing is *on*, details of the analysis of a form are displayed on the screen during generation or recognition operations. If logging (see section 4.5.7) is on, the trace will also be written to the log file. Tracing is operative for these commands: *generate*, *recognize*, *file compare generate*, *file compare recognize*, *file compare pairs*, *file generate*, and *file recognize*. The default setting is *off*.

The amount of detail shown in the trace display is set by the tracing level. The *level* argument to the *set tracing* command can range from 0 to 3, where 0 is no tracing at all and 3 is the most detailed level of tracing. Issuing the command *set tracing off* sets tracing to level 0. Issuing the command *set tracing on* sets tracing to level 2. At level 1, no information is given as to which feasible pair is being tried or the condition of the rules (that is, what state each automaton is in). Both the generator and recognizer report each RESULT line, with all NULL symbols being explicitly printed. The recognizer also displays lexicon information; that is, it reports which sublexicon is being entered or backed out of. At level 2, the feasible pairs being tried and the state of each rule (automaton) is displayed. The recognizer displays lexicon information as it does at level 1. At level 3, more detailed information is given on which feasible pairs are being tried and the state of each rule. For more information on the format of the trace display, see section 4.5.8 on trace formats.

### **set verbose {on | off}**

The *set verbose* command affects the amount of information displayed on the screen during a file comparison operation (either *generate*, *recognize*, or *pairs*, see section 4.5.12). If verbose is set *off*, a file comparison operation displays only a dot for each form correctly analyzed, though any exceptional results will cause the complete form and warning messages to be displayed. If verbose is set *on*, a file comparison operation displays the complete contents of the file (minus comments) plus confirmation and warning messages. The default setting is *off*.

### **set warnings {on | off}**

The *set warnings* command turns warning mode on or off. If warning mode is *on*, then any warning messages that occur while loading a file or while processing a form are displayed on the screen. If warning

mode is *off*, then no warning messages are displayed. The default setting is *on*. (See section 4.10 for an explanation of the difference between errors and warnings.)

#### 4.5.6.2 Options related to use of a word grammar

##### **set ambiguities** *number*

The *set ambiguities* command limits the number of analyses produced by the word grammar to the specified *number*. The default setting is 10. Note that this command assumes that a word grammar is loaded (see section 4.5.2) and that the *grammar* option is set to *on* (see section 4.5.6.2).

##### **set failures** {**on** | **off**}

The *set failures* command turns grammar failure mode on or off. When grammar failure mode is *on*, the partial results of forms that fail the word grammar are displayed. A form may fail the word grammar either by failing the feature constraints or by failing the constituent structure rules. In the latter case, a partial tree (bush) will be displayed. When grammar failure mode is *off*, forms that fail the word grammar are filtered out and no results for them are displayed. The default setting is *off*. Note that this command assumes that a word grammar is loaded (see section 4.5.2) and that the *grammar* option is set to *on* (see section 4.5.6.2).

##### **set features** {**top** | **all** | **off**}

##### **set features** {**full** | **flat**}

The command *set features* controls the display of feature structures returned by the recognizer when a word grammar is used. When *features* is set to *top*, the feature structure for only the top node of the tree is displayed. When *features* is set to *all*, the feature structures for all nodes of the tree are displayed. And when *features* is set to *off*, no feature structures are displayed. The default setting is *top*.

When *features* is set to *full*, the feature structures are displayed in a vertical, indented format; when *features* is set to *flat*, the feature structures are displayed as a linear string. The default setting is *full*.

Note that this command assumes that a word grammar is loaded (see section 4.5.2) and that the *grammar* option is set to *on* (see section 4.5.6.2).

##### **set grammar** {**on** | **off**}

The command *set grammar* turns the word grammar on or off. When *grammar* is *on*, then results from the lexicon are passed to the word grammar for parsing. When *grammar* is *off*, then the results from the lexicon are displayed without using the word grammar. The default setting is *off*, but is automatically turned *on* when a grammar is loaded.

##### **set tree** {**full** | **flat** | **indented** | **off**}

The command *set tree* controls the display of the tree structure returned by the recognizer when a word grammar is used. When *tree* is set to *full*, a full branching tree is displayed. When *tree* is set to *flat*, a linear bracketed string is displayed. When *tree* is set to *indented*, a north-west oriented indented tree is displayed. And when *tree* is set to *off*, no tree structure is displayed. The default setting is *full*. Note that this command assumes that a word grammar is loaded (see section 4.5.2) and that the *grammar* option is set to *on* (see section 4.5.6.2).

##### **set trim-empty-features** {**on** | **off**}

The command *set trim-empty-features* controls the display of features that have empty values. When *trim-empty-features* is set to *on*, empty features are not displayed. When *trim-empty-features* is set to *off*, empty features are displayed. The default setting is *on*.

**set unification {on | off}**

The command *set unification* turns feature unification in the word grammar on or off. When *unification* is *on*, any feature constraints used in conjunction with the grammar rules are used as expected. When *unification* is *off*, the feature constraints are ignored and only the constituent structure rules are used. The default setting is *on*. Note that this command assumes that a word grammar is loaded (see section 4.5.2) and that the *grammar* option is set to *on* (see section 4.5.6.2).

### 4.5.6.3 Save settings

**save [*filespec*]**

The *save* command writes the current setting to a "take" file named *filespec*. If *filespec* is not specified, the settings are written to a file named PCKIMMO.TAK in the current directory. On start-up, PC-KIMMO automatically tries to load default settings from PCKIMMO.TAK (or PC-KIMMO.TAK).

## 4.5.7 Turn logging on or off

The *log* and *close* commands are used to turn logging on and off.

**log [*filespec*]**

The *log* command turns the logging mechanism on. When logging is on, the information displayed on the screen during execution of generation or recognition operations is also written to a disk file whose name is specified in the command line. The *filespec* can contain a path; for example, B:\ENGLISH\ENGLISH.LOG is a fully specified path under MS-DOS (see section 4.5.18 for information on Macintosh directory paths). If no file name is given, a log file named PCKIMMO.LOG is written to the default directory. If a file name with no extension is given, a log file with the extension .LOG is written to the default directory. If a *log* command is given when a log file is already open, then the open log file is closed before the new log file is created. Logging records the processing of these commands: *generate*, *recognize*, *file compare generate*, *file compare recognize*, *file compare pairs*, *file generate*, and *file recognize*. Tracing displays are also recorded in a log file.

**close**

The *close* command turns logging off and closes the log file.

## 4.5.8 Show system status

The *status* command is used to display on the screen the status of various system parameters.

**status**

The *status* command displays the names of the rules, lexicon and grammar files currently loaded, the name of the log file (if logging is on), the comment delimiter character, and the status of the processing options controlled by the *set* command. It can also be invoked with the synonyms *show status* or *show*.

## 4.5.9 Show rule or sublexicon

**show rule *rule-number***

The *show rule* command first displays the number, on/off status, and name of the rule (similar to the *list rules* command). If the rule is turned on, it then displays each column header of the state table for that rule with the set of feasible pairs that it specifies. This command is used primarily for debugging purposes.

**show lexicon *sublexicon-name***

The *show lexicon* command displays the contents of a sublexicon. It shows each lexical item, its gloss, and its continuation class. If the continuation class of a lexical entry names an alternation, the alternation is expanded into a list of sublexicon names. Note that this command displays the parts of the lexical entry in the following order (rather than the order in which they appear in the lexicon file): *lexical item*, *gloss*, *continuation class*.

## 4.5.10 Generate surface forms from a lexical form

**generate** [*lexical-form*]

The *generate* command accepts as input a lexical form and returns one or more surface forms. If no lexical form argument is given, PC-KIMMO supplies a special generator prompt where forms can be typed in directly without the *generate* keyword. Entering a blank line at the generator prompt returns the program to the main command line prompt.

## 4.5.11 Recognize lexical forms from a surface form

**recognize** [*surface-form*]

The *recognize* command accepts as input a surface form and returns one or more lexical forms. If no surface form argument is given, PC-KIMMO supplies a special recognizer prompt where forms can be typed in directly without the *recognize* keyword. Entering a blank line at the recognizer prompt returns the program to the main command line prompt.

## 4.5.11A Synthesize surface forms from a morphological form

**recognize** [*morphological-form*]

The *synthesize* command accepts as input a morphological form (a sequence of morpheme glosses separated by spaces) and returns one or more surface forms. If no morphological form argument is given, PC-KIMMO supplies a special synthesizer prompt where forms can be typed in directly without the *synthesize* keyword. Entering a blank line at the synthesizer prompt returns the program to the main command line prompt.

## 4.5.12 Compare data from a file

The *compare* commands compare data prepared by the user to the results of data processed by PC-KIMMO. The data are contained in files whose formats are described in sections 4.7.4, 4.7.5, and 4.7.6.

**[file] compare generate** [*filespec*]

The *compare generate* command reads lexical forms from a file, submits them to the generator for analysis, and compares the resulting surface form(s) with the expected results listed in the file. The *filespec* can contain a path; for example, B:\ENGLISH\ENGLISH.GEN is a fully specified path under MS-DOS (see section 4.5.18 for information on Macintosh directory paths). A generation comparison file has the default extension .GEN and the default file name DATA.GEN. The format of the generation comparison file is described in section 4.7.4. If a word grammar is in use, the *tree* option and the *features* option must both be set *off*.

Results of the comparison are reported according to the setting of the *verbose* option (see the *set verbose* command described in section 4.5.6.1). If *verbose* is set *off*, only exceptions (that is, actual results from the generator that are different from the expected results as specified in the file) are reported. A dot is displayed on the screen as each input (lexical) form is processed. If *verbose* is set *on*, each group of lexical and surface forms in the file is displayed, either with an error message for wrong comparisons or the message OK if the actual and expected results match exactly.

**[file] compare recognize** [*filespec*]

The *compare recognize* command reads surface forms from a file, submits them to the recognizer for analysis, and compares the resulting lexical form(s) with the expected results specified in the file. The *filespec* can contain a path; for example, B:\ENGLISH\ENGLISH.REC is a fully specified path under MS-DOS (see section 4.5.18 for information on Macintosh directory paths). A recognition comparison file has the default extension .REC and the default file name DATA.REC. The format of the recognition comparison file is described in section 4.7.5. If a word grammar is in use, the *tree* option and the *features* option must both be set *off*.

Results of the comparison are reported according to the setting of the *verbose* option (see the *set verbose* command described in section 4.5.6.1). If *verbose* is set *off*, only exceptions (that is, actual results from the recognizer that are different from the expected results as specified in the file) are reported. A dot is displayed on the screen as each input (surface) form is processed. If *verbose* is set *on*, each group of surface and lexical forms in the file is displayed, either with an error message for wrong comparisons or the message OK if the actual and expected results compared identically.

#### **[file] compare pairs [filespec]**

The *compare pairs* command allows lexical:surface pairs of forms listed in the file specified on the command line to be compared in both directions. The *filespec* can contain a path; for example, B:\ENGLISH\ENGLISH.PAI is a fully specified path under MS-DOS (see section 4.5.18 for information on Macintosh directory paths). A pairs comparison file has the default extension .PAI and the default file name DATA.PAI. The format of the pairs comparison file is described in section 4.7.6. If a word grammar is in use, the *tree* option and the *features* option must both be set *off*.

PC-KIMMO considers each pair of forms (a lexical form followed by its surface form). The lexical form is input to the generator to produce one or more surface forms. The surface form listed in the file is compared with the generated surface forms to see if there is a successful match. The surface form listed in the file is then input to the recognizer to produce one or more lexical forms. The lexical form listed in the file is compared with the recognized lexical forms to see if there is a successful match.

Results of the comparison are reported according to the setting of the *verbose* option (see the *set verbose* command described in section 4.5.6.1). If *verbose* is set *off*, only exceptions (that is, one of the comparisons failed) are reported. A dot is displayed on the screen as each pair of forms is processed. If *verbose* is set *on*, each pair of lexical and surface forms in the file is displayed, either with an error message for wrong comparisons or the message OK if the forms match exactly.

#### **[file] compare synthesize [filespec]**

The *compare synthesize* command reads morphological forms from a file, submits them to the synthesizer for analysis, and compares the resulting surface form(s) with the expected results listed in the file. The *filespec* can contain a path; for example, B:\ENGLISH\ENGLISH.SYN is a fully specified path under MS-DOS (see section 4.5.18 for information on Macintosh directory paths). A generation comparison file has the default extension .SYN and the default file name DATA.SYN. The format of the synthesis comparison file is described in section 4.7.6A. A morphological form is a sequence of morpheme glosses separated by spaces. Results of the comparison are reported according to the setting of the *verbose* option (see the *set verbose* command described in section 4.5.6.1). If *verbose* is set *off*, only exceptions (that is, actual results from the synthesizer that are different from the expected results as specified in the file) are reported. A dot is displayed on the screen as each input (morphological) form is processed. If *verbose* is set *on*, each group of morphological and surface forms in the file is displayed, either with an error message for wrong comparisons or the message OK if the actual and expected results match exactly.

### **4.5.13 Generate forms from a file**

#### **file generate input-filespec [output-filespec]**

The *file generate* command reads lexical forms from a file, submits them to the generator for analysis, and returns each lexical form followed by the resulting surface form(s). The format of the generation input file is

described in section 4.7.7.

If an *output-filespec* argument is specified, the results are written to that file; otherwise, the results are displayed on the screen. The format of the output file created by this command is identical to a comparison generation file. The *filespec* of either file can contain a path; for example, B:\ENGLISH\ENGLISH.FG is a fully specified path under MS-DOS (see section 4.5.18 for information on Macintosh directory paths). The command does not recognize any default file names or extensions.

The *verbose* option (see the *set verbose* command described in section 4.5.6.1) has no effect on the *file generate* command.

## 4.5.14 Recognize forms from a file

**file recognize** *input-filespec* [*output-filespec*]

The *file recognize* command reads surface forms from a file, submits them to the recognizer for analysis, and returns each surface form followed by the resulting lexical form(s). The format of the recognition input file is described in section 4.7.8. If an *output-filespec* argument is specified, the results are written to that file; otherwise the results are displayed on the screen. The format of the output file created by this command is identical to a comparison recognition file. The *filespec* of either file can contain a path; for example, B:\ENGLISH\ENGLISH.FR is a fully specified path under MS-DOS (see section 4.5.18 for information on Macintosh directory paths). The command does not recognize any default file names or extensions.

The *verbose* option (see the *set verbose* command described in section 4.5.6.1) has no effect on the *file recognize* command.

## 4.5.14A Synthesize forms from a file

**file synthesize** *input-filespec* [*output-filespec*]

The *file synthesize* command reads morphological forms from a file, submits them to the synthesizer for analysis, and returns each morphological form followed by the resulting surface form(s). The format of the synthesis input file is described in section 4.7.8A. A morphological form is a sequence of morpheme glosses separated by spaces.

If an *output-filespec* argument is specified, the results are written to that file; otherwise, the results are displayed on the screen. The *filespec* of either file can contain a path; for example, B:\ENGLISH\ENGLISH.FS is a fully specified path under MS-DOS (see section 4.5.18 for information on Macintosh directory paths). The command does not recognize any default file names or extensions.

The *verbose* option (see the *set verbose* command described in section 4.5.6.1) has no effect on the *file synthesize* command.

## 4.5.15 Execute an operating system command

**system** [*system-command*]

The *system* command allows you to execute an operating system command from within PC-KIMMO. It is available only for the MS-DOS and UNIX versions of PC-KIMMO. For example, on an IBM PC-compatible computer, the command *system dir* will execute the DOS directory command. If no command argument is given, then PC-KIMMO is pushed into the background and a new system command processor shell is started. While you are in the shell, you can execute any commands or programs. To leave the shell and return to PC-KIMMO, type *exit*. On an IBM PC-compatible computer, the *system* command will not work unless a copy of the DOS system file COMMAND.COM is available.

The *system* command has the alias *!* (exclamation point), which does not require a space between it and the

following command. For example, *!dir* performs the DOS directory command.

## 4.5.16 Edit a file

**edit** *filespec*

The *edit* command attempts to edit a file using the editing program specified by the operating system environment variable EDITOR. It is available only for the MS-DOS and UNIX versions of PC-KIMMO. If this environment variable is not defined, then the command will try to use EDLIN (on a DOS machine) or vi (on a UNIX machine) to edit the file. To set the environment variable, include a line such as this in your AUTOEXEC.BAT file: SET EDITOR=*filespec*

where *filespec* specifies the path and full file name of your editing program, for example, \UTIL\EMACS.EXE. You can use the *edit* command, for example, to invoke your text editor and modify the rules or lexicon files. After saving the files and leaving the editor, you must *load* the files again in order for PC-KIMMO to utilize the changes.

## 4.5.17 Halt the program

**exit**

The *exit* command causes PC-KIMMO to exit back to the operating system.

**quit**

The command *quit* is the same as *exit*.

## 4.5.18 Change directory

**cd** *pathname*

PC-KIMMO considers the directory (folder) where the program file resides to be the current or default directory. While running PC-KIMMO, it is often necessary to access files in other directories. The Macintosh version of PC-KIMMO has a special command for changing directories called CD. Its command syntax is CD *pathname*, where *pathname* is a concatenation of volume and directory names. The Macintosh uses the colon as a separator character between directory names in much the same way that MS-DOS uses a \ or backslash character. There are two types of pathnames: full pathnames and partial pathnames. A full pathname starts with the name of the root directory (or volume name). For example, a full pathname to the English directory might look like this:

```
MyDisk:PC-KIMMO:English
```

A partial pathname starts with the name of a directory whose position in the directory tree must be specified relative to the current directory. A partial pathname always starts with a colon (whereas a full pathname never does). For example, if the current directory is MyDisk, the partial pathname to specify the English directory is:

```
:PC-KIMMO:English
```

Thus if we are running PC-KIMMO while in the MyDisk directory and we want to change to the English directory, we issue the command:

```
CD :PC-KIMMO:English
```

Here are some things to remember when using Macintosh pathnames with the CD command.



- Directory names are not case sensitive; if the directory name is "English", you can refer to it as "english" or "ENGLISH".
- A directory name used in a CD command cannot contain a space.
- To change to the directory below the current one, use a partial pathname; for example CD :ENGLISH.
- To change to the directory immediately above the current one, simply issue the command CD :: (two colons). To go up two directories, type CD ::: (three colons), and so on.
- To change to a directory that shares a parent with the current directory, go up to the parent using colons and then specify the directory names to go down to the sibling directory. For example, if you are in the English directory and want to change to the Finnish directory type CD ::FINNISH (which means, go up one directory and then down to the FINNISH directory).
- It is also possible to access files in a directory other than the current one without changing directories. This can be done by specifying the file's pathname when it is used in a command. The pathname follows the same conventions as described above for the CD command. For example, the command LOAD RULES :ENGLISH:ENGLISH.RUL will look for the file ENGLISH.RUL in a directory named ENGLISH immediately below the current directory.

## 4.6 Alphabetic list of commands

---

This section documents each command, arranged alphabetically, of the PC-KIMMO system. Square brackets in the command line summaries indicate optional elements. The notation  $\{x \mid y\}$  means either  $x$  or  $y$  (but not both). Command keywords and arguments in boldface are typed literally; for instance, the command summary **set tracing** {**on** | **off**} means to type either *set tracing on* or *set tracing off*. Command arguments in italics are replaced by elements of the specified type; for instance, the command summary **show rule** *number* means to replace *number* with a rule number, such as *show rule 3*.

**!** [*system-command*]

Executes an operating system command or invoke a new command processor shell (same as *system*).

**?**

Displays a list of command names.

**cd** *pathname*

Changes default subdirectory. This command is available only for the Macintosh version only.

**close**

Turns logging off and closes the log file.

**edit** *filespec*

Edits *filespec* using the editing program specified by the operating system environment variable EDITOR. This command is available only for the MS-DOS and UNIX versions of PC-KIMMO.

**exit**

Exits PC-KIMMO and returns to the operating system.

**[file] compare generate** [*filespec*]

Reads lexical forms from *filespec*, submits them to the generator, and compares the resulting surface form(s) with the expected results listed in *filespec*.

**[file] compare recognize** [*filespec*]

Reads surface forms from *filespec*, submits them to the recognizer, and compares the resulting lexical form(s) with the expected results listed in *filespec*.

**[file] compare synthesize** [*filespec*]

Reads morphological forms (a sequence of morpheme glosses separated by spaces) from *filespec*, submits them to the synthesizer, and compares the resulting surface form(s) with the expected results listed in *filespec*.

**[file] compare pairs** [*filespec*]

Reads pairs of lexical and surface forms from *filespec* and analyzes them to see if the surface form can be generated from the lexical form and the lexical form can be recognized from the surface form.

**file generate** *input-filespec* [*output-filespec*]

Reads a list of lexical forms from *input-filespec*, submits them to the generator, and returns each lexical form followed by the resulting surface form(s).

**file recognize** *input-filespec* [*output-filespec*]

Reads a list of surface forms from *input-filespec*, submits them to the recognizer, and returns each surface form followed by the resulting lexical form(s).

**file synthesize** *input-filespec* [*output-filespec*]

Reads a list of morphological forms (a sequence of morpheme glosses separated by spaces) from *input-filespec*, submits them to the synthesizer, and returns each morphological form followed by the resulting surface form(s).

**generate** [*lexical-form*]

Accepts as input a lexical form and returns one or more surface forms.

**help** [*command-name*]

Without a command name argument, displays a list of commands with a brief explanation of each. With a command name argument, displays a usage summary for the command.

**list lexicon**

Displays on the screen the names of the sublexicons of the lexicon currently in use.

**list pairs**

Displays the set of feasible pairs specified by the set of rules currently turned on.

**list rules**

Displays the current state of the rules that are loaded.

**load grammar** [*filespec*]

Loads grammar from *filespec*.

**load lexicon** [*filespec*]

Loads lexicon from *filespec*.

**load rules** [*filespec*]

Loads rules from *filespec*.

**load synthesis-lexicon** [*filespec*]

Loads synthesis-lexicon from *filespec*.

**log** [*filespec*]

Turns the logging mechanism on.

**clear**

Clears the rules, lexicon, synthesis-lexicon, and grammar currently loaded.

**quit**

Same as *exit*.

**recognize** [*surface-form*]

Accepts as input a surface form and returns one or more lexical forms.

**save** [*filespec*]

Save the current settings to a "take" file named *filespec*.

**set alignment** {on | off}

Turns alignment mode on or off.

**set ambiguities** *number*

Limits the number of analyses produced by the word grammar to *number*.

**set failures** {on | off}

Turns grammar failure mode on or off.

**set features** {top | all | off}

**set features** {full | flat}

Determines how feature structures are displayed.

**set grammar** {on | off}

Turns the loaded word grammar on or off.

**set limit** {on | off}

Limits the result of a generation or recognition function to one form.

**set rules** {**on** | **off**} {*list of numbers* | **all**}

Turns selected rules on or off.

**set timing** {**on** | **off**}

Times the execution of generation and recognition functions and displays the result.

**set tracing** {**on** | **off** | *level*}

Turns the tracing mechanism on or off.

**set tree** {**full** | **flat** | **indented** | **off**}

Determines how tree structures are displayed.

**set trim-empty-features** {**on** | **off**}

Determines how empty features are displayed.

**set unification** {**on** | **off**}

Turns feature unification in the word grammar on or off.

**set verbose** {**on** | **off**}

Determines the amount of information shown on the screen during a file comparison operation.

**set warnings** {**on** | **off**}

Turns warning mode on or off.

**show** [**status**]

Same as *status*.

**show lexicon** *sublexicon-name*

Displays the contents of the named sublexicon. For each lexical entry it shows the lexical form, gloss, and continuation class.

**show rule** *rule-number*

Displays the number, on/off status, and name of the rule (similar to the list rules command). If the rule is turned on, it then displays each column header of the state table for that rule with the set of feasible pairs that it specifies.

**status**

Displays the names of the rules, lexicon and grammar files currently loaded, the name of the log file (if logging is on), the comment delimiter character, and the status of the processing options controlled by the *set* command. Obeys the synonyms *show status* and *show*.

**synthesize** [*morphological-form*]

Accepts as input a morphological form (a sequence of morpheme glosses separated by spaces) and returns one or more surface forms.

**system** [*system-command*]

Executes an operating system command or invokes a new command processor shell. See also **!**. This command is available only for the MS-DOS and UNIX versions of PC-KIMMO.

**take** [*filespec*]

Reads and executes commands from *filespec*.

## 4.7 File formats

---

### 4.7.1 Rules file

### 4.7.2 Lexicon files

### 4.7.3 Grammar file

### 4.7.4 Generation comparison file

### 4.7.5 Recognition comparison file

### 4.7.6 Pairs comparison file

### 4.7.6A Synthesis comparison file

### 4.7.7 Generation file

### 4.7.8 Recognition file

### 4.7.8A Synthesis file

### 4.7.9 Summary of default file names and extensions

**Figure 4.1** Structure of the rules file

**Figure 4.2** A sample rules file

**Figure 4.3** Structure of the main lexicon file

**Figure 4.4** A sample main lexicon file

**Figure 4.5** Structure of a lexical entry

**Figure 4.6** A sample lexical entry

**Figure 4.7** Structure of the grammar file

**Figure 4.8A** A lexical rule example

**Figure 4.8B** Feature structure before application of lexical rule

**Figure 4.8C Feature structure after application of lexical rule**

**Figure 4.9 A sample grammar file**

**Figure 4.10 A sample generation comparison file**

**Figure 4.11 A sample recognition comparison file**

**Figure 4.12 A sample pairs comparison file**

**Figure 4.12A A sample synthesis comparison file**

**Figure 4.13 A sample generation file**

**Figure 4.14 A sample recognition file**

**Figure 4.14A A sample synthesis file**

**Figure 4.15 Default file names and extensions**

---

This section describes the formats for the files that are used as input to PC-KIMMO. In any of the files, comments can be added to any line by preceding the comment with the comment character. This character is normally a semicolon (;), but can be changed with the COMMENT keyword in the rules file. Anything following a comment character (until the end of the line) is considered part of the comment and is ignored by PC-KIMMO.

In the descriptions below, reference to the use of a space character implies any whitespace character (that is, any character treated like a space character). The following control characters when used in a file are whitespace characters: ^I (ASCII 9, tab), ^J (ASCII 10, line feed), ^K (ASCII 11, vertical tab), ^L (ASCII 12, form feed), and ^M (ASCII 13, carriage return).

The control character ^Z (ASCII 26) cannot be used because MS-DOS interprets it as marking the end of a file. Also the control character ^@ (ASCII 0, null) cannot be used.

Examples of each of the following file types are found on the release diskette as part of the English description.

## **4.7.1 Rules file**

The general structure of the rules file is a list of keyword declarations. Figure 4.1 shows the conventional structure of the rules file. Note that the notation  $\{x \mid y\}$  means either  $x$  or  $y$  (but not both). The following specifications apply to the rules file.

**Figure 4.1 Structure of the rules file**

```
COMMENT <character>
ALPHABET <symbol list>
NULL <character>
ANY <character>
BOUNDARY <character>
SUBSET <subset name> <symbol list>
. (more subsets)
.
.
RULE <rule name> <number of states> <number of columns>
    <lexical symbol list>
    <surface symbol list>
<state number>{: | .} <state number list>
```

```

. (more states)
.
.
. (more rules)
.
.
END

```

- Extra spaces, blank lines, and comment lines are ignored.
- Comments may be placed anywhere in the file. All data following a comment character to the end of the line is ignored. (See below on the COMMENT declaration.)
- The set of valid keywords used to form declarations includes COMMENT, ALPHABET, NULL, ANY, BOUNDARY, SUBSET, RULE, and END.
- These declarations are obligatory and can occur only once in a file: ALPHABET, NULL, ANY, BOUNDARY.
- These declarations are optional and can occur one or more times in a file: COMMENT, SUBSET, and RULE.
- The COMMENT declaration sets the comment character used in the rules file, lexicon files, and grammar file. The COMMENT declaration can only be used in the rules file, not in the lexicon or grammar file. The COMMENT declaration is optional. If it is not used, the comment character is set to ; (semicolon) as a default.
- The COMMENT declaration can be used anywhere in the rules file and can be used more than once. That is, different parts of the rules file can use different comment characters. The COMMENT declaration can (and in practice usually does) occur as the first keyword in the rules file, followed by either one or more COMMENT declarations or the ALPHABET declaration.
- Note that if you use the COMMENT declaration to declare the character that is already in use as the comment character, an error will result. For instance, if semicolon is the current comment character, the declaration *COMMENT ;* will result in an error.
- The comment character can no longer be set using a command line option or with a command in the user interface, as was the case in version 1 of PC-KIMMO.
- The ALPHABET declaration must either occur first in the file or follow one or more COMMENT declarations only. The other declarations can appear in any order. The COMMENT, NULL, ANY, BOUNDARY, and SUBSET declarations can even be interspersed among the rules. However, these declarations must appear before any rule that uses them or an error will result.
- The ALPHABET declaration defines the set of symbols used in either lexical or surface representations. The keyword ALPHABET is followed by a *<symbol list>* of all alphabetic symbols. Each symbol must be separated from the others by at least one space. The list can span multiple lines, but ends with the next valid keyword. All alphanumeric characters (such as *a*, *B*, and *2*), symbols (such as \$ and +), and punctuation characters (such as . and ?) are available as alphabet members. The characters in the IBM extended character set (above ASCII 127) are also available. Control characters (below ASCII 32) can also be used, with the exception of whitespace characters (see above), ^Z (end of file), and ^@ (null). The alphabet can contain a maximum of 255 symbols. An alphabetic symbol can also be a multigraph, that is, a sequence of two or more characters. The individual characters composing a multigraph do not necessarily have to also be declared as alphabetic characters. For example, an alphabet could include the characters *s* and *z* and the multigraph *sz%*, but not include % as an alphabetic character. Note that a multigraph cannot also be interpreted as a sequence of the individual characters that comprise it.
- The keyword NULL is followed by a single *<character>* that represents a null (empty, zero) element. The NULL symbol is considered to be an alphabetic character, but cannot also be listed in the

ALPHABET declaration. The NULL symbol declared in the rules file is also used in the lexicon file to represent a null lexical entry.

- The keyword ANY is followed by a single "wildcard" *<character>* that represents a match of any character in the alphabet. The ANY symbol is not considered to be an alphabetic character, though it is used in the column headers of state tables. It cannot be listed in the ALPHABET declaration. It is not used in the lexicon file.
- The keyword BOUNDARY is followed by a single *<character>* character that represents an initial or final word boundary. The BOUNDARY symbol is considered to be an alphabetic character, but cannot also be listed in the ALPHABET declaration. When used in the column header of a state table, it can only appear as the pair *#:#* (where, for instance, *#* has been declared as the BOUNDARY symbol). The BOUNDARY symbol is also used in the lexicon file in the continuation class field of a lexical entry to indicate the end of a word (that is, no continuation class).
- The SUBSET declaration defines set of characters that are referred to in the column headers of rules. The keyword SUBSET is followed by the *<subset name>* and *<symbol list>*. *<subset name>* is a single word (one or more characters) that names the list of characters that follows it. The subset name must be unique (that is, if it is a single character it cannot also be in the alphabet or be any other declared symbol). It can be composed of any characters (except space); that is, it is not limited to the characters declared in the ALPHABET section. It must not be identical to any keyword used in the rules file. The subset name is used in rules to represent all members of the subset of the alphabet that it defines. Note that SUBSET declarations can be interspersed among the rules. This allows subsets to be placed near the rule that uses them if such a style is desired. However, a subset must be declared before a rule that uses it.
- The *<symbol list>* following a *<subset name>* is a list of single symbols, each of which is separated by at least one space. The list can span multiple lines. Each symbol in the list must be a member of the previously defined ALPHABET, with the exception of the NULL symbol, which can appear in a subset list but is not included in the ALPHABET declaration. Neither the ANY symbol nor the BOUNDARY symbol can appear in a subset symbol list.
- The keyword RULE signals that a state table immediately follows.
- *<rule name>* is the name or description of the rule which the state table encodes. It functions as an annotation to the state table and has no effect on the computational operation of the table. It is displayed by the *list rules* and *show rule* commands and is also displayed in traces. The rule name must be surrounded by a pair of identical delimiter characters. Any material can be used between the delimiters of the rule name with the exception of the current comment character and of course the rule name delimiter character of the rule itself. Each rule in the file can use a different pair of delimiters. The rule name must be all on one line, but it does not have to be on the same line as the RULE keyword.
- *<number of states>* is the number of states (rows in the table) that will be defined for this table. The states must begin at 1 and go in sequence through the number defined here (that is, gaps in state numbers are not allowed).
- *<number of columns>* is the number of state transitions (columns in the table) that will be defined for each state.
- *<lexical symbol list>* is a list of elements separated by one or more spaces. Each element represents the lexical half of a lexical:surface correspondence which, when matched, defines a state transition. Each element in the list must be either a member of the alphabet, a subset name, the NULL symbol, the ANY symbol, or the BOUNDARY symbol (in which case the corresponding surface character must also be the BOUNDARY symbol). The list can span multiple lines, but the number of elements in the list must be equal to the number of columns defined for the rule.
- *<surface symbol list>* is a list of elements separated by one or more spaces. Each element represents the surface half of a lexical:surface correspondence which, when matched, defines a state transition.



Each element in the list must be either a member of the alphabet, a subset name, the NULL symbol, the ANY symbol, or the BOUNDARY symbol (in which case the corresponding lexical character must also be the BOUNDARY symbol). The list can span multiple lines, but the number of characters in the list must be equal to the number of columns defined for the rule.

- *<state number>* is the number of the state or row of the table. The first state number must be 1, and subsequent state numbers must follow in numerical sequence without any gaps.
- *{:|.}* is the final or nonfinal state indicator. This should be a colon (:) if the state is a final state and a period (.) if it is a nonfinal state. It must follow the *<state number>* with no intervening space.
- *<state number list>* is a list of state transition numbers for a particular state. Each number must be between 1 and the number of states (inclusive) declared for the table. The list can span multiple lines, but the number of elements in the list must be equal to the number of columns declared for this rule.
- The keyword END follows all other declarations and indicates the end of the rules file. Any material in the file thereafter is ignored by PC-KIMMO. The END keyword is optional; the physical end of the file also terminates the rules file.

Figure 4.2 shows a sample rules file.

**Figure 4.2** A sample rules file

```
ALPHABET
  b c d f g h j k l m n p q r s t v w x y z +      ; + is morpheme boundary
  a e i o u
NULL 0
ANY @
BOUNDARY #
SUBSET C b c d f g h j k l m n p q r s t v w x y z
SUBSET V a e i o u
; more subsets

RULE "Consonant defaults" 1 23
  b c d f g h j k l m n p q r s t v w x y z + @
  b c d f g h j k l m n p q r s t v w x y z 0 @
1: 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

RULE "Vowel defaults" 1 6
  a e i o u @
  a e i o u @
1: 1 1 1 1 1 1

RULE "Voicing s:z <=> V__V" 4 4
  V s s @
  V z @ @
1: 2 0 1 1
2: 2 4 3 1
3: 0 0 1 1
4: 2 0 0 0

; more rules

END
```

## 4.7.2 Lexicon files

A lexicon consists of one main lexicon file plus one or more files of lexical entries. The general structure of the main lexicon file is a list of keyword declarations. The set of valid keywords is ALTERNATION, FEATURES, FIELD CODE, INCLUDE, and END. Figure 4.3 shows the conventional structure of the lexicon file. The following specifications apply to the main lexicon file.

**Figure 4.3** Structure of the main lexicon file

```
ALTERNATION <alternation name> <sublexicon name list>
. (more ALTERNATIONS)
.
.
FEATURES <feature abbreviation list>

FIELDPCODE <lexical item code> U
FIELDPCODE <sublexicon code> L
FIELDPCODE <alternation code> A
FIELDPCODE <features code> F
FIELDPCODE <gloss code> G

INCLUDE <filespec>
. (more INCLUDED files)
.
.
END
```

- Extra spaces, blank lines, and comment lines are ignored.
- The comment character declared in the rules file is operative in the main lexicon file. Comments may be placed anywhere in the file. All data following a comment character to the end of the line is ignored.
- The set of valid keywords used to form declarations includes ALTERNATION, FEATURES, FIELDPCODE, INCLUDE, and END.
- The declarations can appear in any order with the proviso that any alternation name, feature name, or fieldcode used in a lexical entry must be declared before the lexical entry is read. In practice, this means that the INCLUDE declarations should appear last, but the ALTERNATION, FEATURES, and FIELDPCODE declarations can appear in any order.
- The ALTERNATION declaration defines a set of sublexicon names that serve as the continuation class of a lexical item. The ALTERNATION keyword is followed by an *<alternation name>* and a *<sublexicon name list>*. ALTERNATION declarations are optional (but nearly always used in practice) and can occur as many times as needed.
- *<alternation name>* is a name associated with the following *<sublexicon name list>*. It is a word composed of one or more characters, not limited to the ALPHABET characters declared in the rules file. An alternation name can be any word other than a keyword used in the lexicon file. The program does not check to see if an alternation name is actually used in the lexicon file.
- *<sublexicon name list>* is a list of sublexicon names. It can span multiple lines until the next valid keyword is encountered. Each sublexicon name in the list must be used in the sublexicon field of a lexical entry. Although it is not enforced at the time the lexicon file is loaded, an undeclared sublexicon named in a sublexicon name list will cause an error when the recognizer tries to use it.
- The FEATURES keyword followed by a *<feature abbreviation list>*. A *<feature abbreviation list>* is a list of words, each of which is expanded into feature structures by the word grammar.
- The FIELDPCODE declaration is used to define what fieldcode will be used to mark each type of field in a lexical entry. The FIELDPCODE keyword is followed by a *<code>* and one of five possible internal codes: U, L, A, F, or G. There must be five FIELDPCODE declarations, one for each of these internal codes, where U indicates the lexical item field, L indicates the sublexicon field, A indicates the alternation field, F indicates the features field, and G indicates the gloss field.
- The INCLUDE keyword is followed by a *<filespec>* that names a file containing lexical entries to be loaded. An INCLUDED file cannot contain any declarations (such as a FIELDPCODE or an INCLUDE declaration), only lexical entries and comment lines.

- The keyword END follows all other declarations and indicates the end of the main lexicon file. Any material in the file thereafter is ignored by PC-KIMMO. The END keyword is optional; the physical end of the file also terminates the main lexicon file.

Figure 4.4 shows a sample main lexicon file.

**Figure 4.4** A sample main lexicon file

```
ALTERNATION Begin PREF
ALTERNATION Pref N AJ V AV
ALTERNATION Stem SUFFIX

FEATURES sg pl reg irreg

FIELD CODE  lf  U  ;lexical item
FIELD CODE  lx  L  ;sublexicon
FIELD CODE  alt A  ;alternation
FIELD CODE  fea F  ;features
FIELD CODE  gl  G  ;gloss

INCLUDE affix.lex      ;file of affixes
INCLUDE noun.lex       ;file of nouns
INCLUDE verb.lex       ;file of verbs
INCLUDE adjectiv.lex   ;file of adjectives
INCLUDE adverb.lex     ;file of adverbs

END
```

Figure 4.5 shows the structure of a lexical entry. Lexical entries are encoded in "field-oriented standard format." Standard format is an information interchange convention developed by the Summer Institute of Linguistics. It tags the kinds of information in ASCII text files by means of markers which begin with backslash. Field-oriented standard format (FOSF) is a refinement of standard format geared toward representing data which has a database-like record and field structure. The following points provide an informal description of the syntax of FOSF files.

**Figure 4.5** Structure of a lexical entry

```
\<lexical item code> <lexical item>
\<sublexicon code> <sublexicon name>
\<alternation code> {<alternation name> | <BOUNDARY symbol>}
\<features code> <features list>
\<gloss code> <gloss string>
```

- A **field-oriented standard format** (FOSF) file consists of a sequence of records.
- A **record** consists of a sequence of fields.
- A **field** consist of a field marker and a field value.
- A **field marker** consists of a backslash character at the beginning of a line, followed by an alphabetic or numeric character, followed by zero or more printable characters, and terminated by a space, tab, or the end of a line. A field marker without its initial backslash character is termed a **field code**.
- A field marker must begin in the first position of a line. Backslash characters occurring elsewhere in the file are not interpreted as field markers.
- The first field marker of the record is considered the record marker, and thus the same field must occur first in every record of the file.
- Each field marker is separated from the **field value** by one or more spaces, tabs, or newlines. The field value continues up to the next field marker.

- Any line that is empty or contains only whitespace characters is considered a comment line and is ignored. Comment lines may occur between or within fields.
- Fields and lines in an FOSF file can be arbitrarily long.
- There are two basic types of fields in FOSF files: **nonrepeating** and **repeating**. Repeating fields are multiple consecutive occurrences of fields marked by the same marker. Individual fields within a repeating field can be called **subfields**.

The following specifications apply to how FOSF is implemented in PC-KIMMO.

- Lexical entries are encoded as records in a FOSF file.
- Only those fields whose field codes are declared in the main lexicon file are recognized (see above on the FIELDPCODE declaration). All other fields are considered to be extraneous and are ignored.
- The first field of each lexical entry must be the lexical item field. The lexical item field code is assigned to the internal code U by a FIELDPCODE declaration in the main lexicon file.
- Only nonrepeating fields are permitted.
- The comment character declared in the rules file is operative in included files of lexical entries. All data following a comment character to the end of the line is ignored.

A file of lexical entries is loaded by using an INCLUDE declaration in the main lexicon file (see above). An INCLUDED file of lexical entries cannot contain any declarations (such as a FIELDPCODE or an INCLUDE declaration), only lexical entries and comment lines.

The following specifications apply to lexical entries.

- A lexical entry is composed of five fields: lexical item, sublexicon, alternation, features, and gloss. The lexical item, sublexicon, and alternation, fields are obligatory, the features and gloss fields are optional. The first field of the entry must always be the lexical item. The other fields can appear in any order, even differing from one entry to another.
- Although the gloss field is optional, if a lexical entry does not include one, a warning message to that effect will be displayed when the entry is loaded. To suppress this warning message, do the command *set warnings off* (see section 4.5.6.1) before loading the lexicon.
- If an entry has an empty gloss field (that is, the field marker for the gloss field is present but there is no data after it), then the contents of the lexical form field will be also be used as the gloss for that entry.
- A lexical item field consists of a *<lexical item code>* and a *<lexical item>*.
- A *<lexical item code>* is a field code assigned to the internal code L by a FIELDPCODE declaration in the main lexicon file.
- A *<lexical item>* is one or more characters that represent an element (typically a morpheme or word) of the lexicon. Each character (or multigraph) must be in the alphabet defined for the language. The lexical item uses only the lexical subset of the alphabet.
- A sublexicon field consists of a *<sublexicon code>* and a *<sublexicon name>*.
- A *<sublexicon code>* is a field code assigned to the internal code L by a FIELDPCODE declaration in the main lexicon file.
- A *<sublexicon name>* is the name associated with a sublexicon. It is a word composed of one or

more characters, not limited to the alphabetic characters declared in the rules file. Every lexical item must belong to a sublexicon. Every lexicon must include a special sublexicon named INITIAL (that is, there must be at least one lexical entry that belongs to the INITIAL sublexicon).

- Lexical entries belonging to a sublexicon do not have to be listed consecutively in a single file (as was the case for PC-KIMMO version 1); rather, lexical entries in a file can occur in any order, regardless of what sublexicon they belong to. Lexical entries of a sublexicon can even be placed in two or more separate files.
- An alternation field consists of a *<alternation code>* followed by either an *<alternation name>* or the *<BOUNDARY symbol>*.
- An *<alternation name>* is declared in an ALTERNATION declaration in the main lexicon file. The *<BOUNDARY symbol>* is declared in the rules file and indicates the end of all possible continuations in the lexicon.
- A features field consists of a *<features code>* and a *<features list>*.
- A *<features code>* is a field code assigned to the internal code F by a FIELD CODE declaration in the main lexicon file.
- A *<features list>* is a list of feature abbreviations. Each abbreviation is a single word consisting of alphanumeric characters or other characters except ( ) { } [ ] < > = : \$ ! (these are used for special purposes in the grammar file). The character \ should not be used as the first character of an abbreviation because that is how fields are marked in the lexicon file. Upper and lower case letters used in template names are considered different. For example, "PLURAL" is not the same as "Plural" or "plural." Feature abbreviations are expanded into full feature structures by the word grammar (see section 4.7.3).
- A gloss field consists of a *<gloss code>* and a *<gloss string>*.
- A *<gloss code>* is a field code assigned to the internal code G by a FIELD CODE declaration in the main lexicon file.
- A *<gloss string>* is a string of text. Any material can be used in the gloss field with the exception of the comment character.

Figure 4.6 shows a sample lexical entry.

**Figure 4.6** A sample lexical entry

```
\lf `knives
\lx N
\alt Infl
\fea pl irreg
\gl N(`knife)+PL
```

## 4.7.3 Grammar file

The grammar file consists of feature templates, context-free rules, and feature constraints. Figure 4.7 shows the conventional structure of the grammar file.

**Figure 4.7** Structure of the grammar file

```
LET <abbreviation | category> be <feature definition>
. (more feature templates)
.
.
DEFINE <lexical rule name> as <mappings>
```

```

. (more lexical rules)
.
.
PARAMETER <parameter name> is <parameter value>
. (more parameter settings)
.
.
RULE <rule>
  <feature constraint>
  . (more constraints)
  .
  .
. (more rules)
.
.
.
END

```

The following specifications apply generally to the grammar file.

- Blank lines, spaces, and tabs separate elements of the grammar file from one another, but are ignored otherwise.
- The comment character declared in the rules file is operative in the grammar file. Comments may be placed anywhere in the grammar. All data following a comment character to the end of the line is ignored.
- A grammar file minimally consists of one or more context-free rules. Each rule may optionally specify feature constraints.
- A grammar file is divided into fields identified by a small set of keywords.
  1. `Rule` starts a context-free rule with its set of feature constraints. These rules define how words join together to form phrases, clauses, or sentences. The lexicon and grammar are tied together by using the lexical categories as the terminal symbols of the context-free rules and by using the other lexical features in the feature constraints.
  2. `Let` starts a feature template definition. Feature templates are used as macros (abbreviations) in the lexicon. They may also be used to assign default feature structures to the categories.
  3. `Parameter` starts a program parameter definition. These parameters control various aspects of the program.
  4. `Define` starts a lexical rule definition. Lexical rules are used to modify the feature structures of lexical entries.
  5. `End` effectively terminates the file. Anything following this keyword is ignored.

Note that these keywords are not case sensitive: `RULE` is the same as `rule`, and both are the same as `Rule`.

- Each of the fields in the grammar file may optionally end with a period. If there is no period, the next keyword (in an appropriate slot) marks the end of one field and the beginning of the next.

## Rules

The following specifications apply to rules.

A grammar rule has these parts, in the order listed:

1. the keyword `Rule`

2. an optional rule identifier enclosed in braces ({} )
3. the nonterminal symbol to be expanded
4. an arrow (->) or equal sign (=)
5. zero or more terminal or nonterminal symbols, possibly marked for alternation or optionality
6. an optional colon (:)
7. zero or more feature constraints, possibly marked for alternation
8. an optional period (.)

The optional rule identifier (item 2) consists of one or more words enclosed in braces. Its current utility is only as a special form of comment describing the intent of the rule. (Eventually it may be used as a tag for interactively adding and removing rules.) The only limits on the rule identifier are that it not contain the comment character and that it all appears on the same line in the grammar file.

The terminal and nonterminal symbols in the rule have the following characteristics:

- Blank lines, spaces, and tabs separate symbols from one another, but otherwise are ignored.
- Upper and lower case letters used in symbols are considered different. For example, `STEM` is not the same as `Stem`, and neither is the same as `stem`.
- Index numbers are used to distinguish instances of a symbol that is used more than once in a rule. They are added to the end of a symbol following an underscore character (`_`). For example,

```
Stem_1 = Stem_2 SUFFIX
```

- The symbol `X` may be used to stand for any terminal or nonterminal category. For example, this rule says that a `N` expands into a `NStem` plus any category.

```
N = NStem X
```

The symbol `X` can be useful for capturing generalities. Care must be taken, since it can be replaced by anything.

- The characters `(){}[]<>=: /` cannot be used in terminal or nonterminal symbols since they are used for special purposes in the grammar file. The character `_` can be used *only* for attaching an index number to a symbol.
- By default, the left hand symbol of the first rule in the grammar file is the start symbol of the grammar.
- There can be multiple rules for the same symbol, but all rules for a symbol must be contiguous in the file.

The symbols on the right hand side of a context-free rule may be marked or grouped in various ways:

- Parentheses around an element of the expansion (righthand) part of a rule indicate that the element is optional. Parentheses may be placed around multiple elements. This makes an optional group of elements.
- A forward slash (`/`) is used to separate alternative elements of the expansion (righthand) part of a rule.
- Curly braces can be used for grouping elements. For example the following says that a `N` consists of a `NStem` followed by either a `Sing` or a `Plural`:

```
N = NStem {Sing / Plural}
```

- Alternatives are taken to be as long as possible. Thus if the curly braces were omitted from the rule above, as in the rule below, the *Sing* would be treated as part of the alternative containing the *NStem*. It would not be allowed before the *Plural*.

```
N = NStem Sing / Plural
```

- Parentheses group enclosed elements the same as curly braces do. Alternatives and groups delimited by parentheses or curly braces may be nested to any depth.

## Feature structures

The grammar formalism uses a basic element called a *feature structure*. A feature structure consists of a feature name and a value. The notation used for feature structures looks like this:

```
[number: singular]
```

where *number* is the feature name and *singular* is the value, separated by a colon. Feature names and values are single words consisting of alphanumeric characters or other characters except `(){}[]<>=: $!` (these are used for special purposes in the grammar file). Upper and lower case letters used in feature names and values are considered different. For example, "NUMBER" is not the same as "Number" or "number."

A structure containing more than one feature uses square brackets around the entire structure:

```
[number: singular  
 case: nominative]
```

Extra spaces and line breaks are optional.

Feature structures can have either simple values, such as the example above, or complex values, such as this:

```
[agreement: [number: singular  
 case: nominative]]
```

where the value of the *agreement* feature is another feature structure. Feature structures can be infinitely nested in this manner.

Feature can share values. This is not the same thing as two features having identical values. In the first example below, the features *a* and *c* have identical values; but in the second example, they share the same value:

```
[a: [p:q]  
b: [p:q]]  
[a: $1[p:q]  
b: $1]
```

Shared values are indicated by coindexing them with the prefix \$1, \$2, and so on.

Portions of a feature structure can be referred to using the "path" notation. A path is a sequence of feature names (minimally one) enclosed in angled brackets (`<>`). For example, consider this feature structure:

```
[agreement: [number: singular  
 case: nominative]]
```

These are feature paths based on this structure:

```
<number>  
<case>  
<agreement number>  
<agreement case>
```



Paths are used in feature templates and feature constraints, described below. All lexical items used by the grammar are assigned three features: *cat*, *lex*, and *gloss*. These should be treated as reserved names and not used for other purposes.

- The value of the *cat* feature is the name of the sublexicon to which the lexical item belongs, taken from the sublexicon field of the item's lexical entry.
- The value of the *lex* feature is the lexical form of the item, taken from the lexical form field of the item's lexical entry.
- The value of the *gloss* feature is the gloss of the item, taken from the gloss field of the item's lexical entry.

For example, here is a lexical entry for the word *fox*:

```
\lf `fox
\lx N
\alt Stem
\gl N(fox)
```

When this entry is used by the grammar, it is represented as this feature structure:

```
[cat: N
 lex: `fox
 gloss: N(fox)]
```

## Feature constraints

A rule is followed by zero or more *feature constraint*; which refer to symbols used in the rule. The following specifications apply to feature constraints.

A feature constraint has these parts, in the order listed:

1. a feature path that begins with one of the symbols from the context-free rule
2. an equal sign
3. either another path or a value

A feature constraint that refers only to symbols on the right hand side of the rule constrains their co-occurrence. In the following rule and constraint, the value of the Stem's *head pos* feature must unify with the value of the SUFFIX's *from\_pos* feature:

```
Word -> Stem INFL
      <Stem head pos> = <INFL from_pos>
```

If a feature constraint refers to a symbol on the right hand side of the rule, and has an atomic value on its right hand side, then the designated feature must not have a different value. In the following rule and constraint, the *head case* feature for the PRONOUN node of the parse tree must either be originally undefined or equal to NOM:

```
Word -> PRONOUN
      <PRONOUN head case> = NOM
```

(If the *head case* feature of the PRONOUN node was originally undefined, then, after unification succeeds, it will be equal to NOM.)

A feature constraint that refers to the symbol on the left hand side of the rule passes information up the parse tree. In the following rule and constraint, the value of the *head* feature is passed from the INFL node up to the

Word node:

```
Word -> Stem INFL
      <Word head> = <INFL head>
```

PC-KIMMO allows disjunctive feature constraints with its phrase structure rules. Consider these two rules:

```
Stem_1 -> PREFIX Stem_2
        <PREFIX from_pos> = <Stem_2 head pos>
        <PREFIX change_pos> = +
        <Stem_1 head> = <PREFIX head>
```

```
Stem_1 -> PREFIX Stem_2
        <PREFIX from_pos> = <Stem_2 head pos>
        <PREFIX change_pos> = -
        <Stem_1 head> = <Stem_2 head>
```

These rules have the same context-free rule part. They can therefore be collapsed into this single rule , which has a disjunction in its feature constraints:

```
Stem_1 -> PREFIX Stem_2
        <PREFIX from_pos> = <Stem_2 head pos>
        {
        <PREFIX change_pos> = +
        <Stem_1 head> = <PREFIX head>
        /
        <PREFIX change_pos> = -
        <Stem_1 head> = <Stem_2 head>
        }
```

Disjunctive feature constraints may be nested up to eight levels deep.

## Feature templates

The following specifications apply to *feature templates*.

A feature template has these parts, in the order listed:

1. the keyword `Let`
2. the template name
3. the keyword `be`
4. a feature definition
5. an optional period (.)

If the template name is a terminal category (a terminal symbol in one of the context-free rules), the template defines the default features for that category. Otherwise the template name serves as an abbreviation for the associated feature structure. Templates may occur anywhere in the file (interspersed among the rules), but a template must occur before any rule or other template that uses the abbreviation it defines.

Template names are single words consisting of alphanumeric characters or other characters except `(){}[]<>=: $!` (these are used for special purposes in the grammar file). The character `\` should not be used as the first character of a template name because that is how fields are marked in the lexicon file. Upper and lower case letters used in template names are considered different. For example, "PLURAL" is not the same as "Plural" or "plural."

The abbreviations defined by templates are usually used in the feature field of entries in the lexicon file. For example, the lexical entry for the irregular plural form *feet* may have the abbreviation *pl* in its features field.

The grammar file would define this abbreviation with a template like this:

```
Let pl be [number: PL]
```

The path notation may also be used:

```
Let pl be <number> = PL
```

More complicated feature structures may be defined in templates. For example,

```
Let 3sg be [tense:  PRES
            agr:    3SG
            finite:  +
            vform:  S]
```

which is equivalent to:

```
Let 3sg be [<tense>  = PRES
            <agr>    = 3SG
            <finite> = +
            <vform>  = S]
```

In the following example, the abbreviation *irreg* is defined using another abbreviation:

```
Let irreg be <reg> = -
              pl
```

The abbreviation *pl* must be defined previously in the grammar file or an error will result. A subsequent template could also use the abbreviation *irreg* in its definition. In this way, an inheritance hierarchy features may be constructed.

Feature templates permit disjunctive definitions. For example, the lexical entry for the word *deer* may specify the feature abbreviation *sg-pl*. The grammar file would define this as a disjunction of feature structures reflecting the fact that the word can be either singular or plural:

```
Let sg/pl be {[number:SG]
               [number:PL]}
```

This has the effect of creating two entries for *deer*, one with singular number and another with plural. Note that there is no limit to the number of disjunct structures listed between the braces. Also, there is no slash (/) between the elements of the disjunction as there is between the elements of a disjunction in the rules. A shorter version of the above template using the path notation looks like this:

```
Let sg/pl be <number> = {SG PL}
```

Abbreviations can also be used in disjunctions, provided that they have previously been defined:

```
Let sg be <number> = SG
Let pl be <number> = PL
Let sg/pl be {[sg] [pl]}
```

Note the square brackets around the abbreviations *sg* and *pl* without square brackets they would be interpreted as simple values instead.

Feature templates can assign default atomic feature values, indicated by prefixing an exclamation point (!). A default value can be overridden by an explicit feature assignment. This template says that all members of category *N* have singular number as a default value:

```
Let N be <number> = !SG
```

The effect of this template is to make all nouns singular unless they are explicitly marked as plural. For

example, regular nouns such as *book* do not need any feature in their lexical entries to signal that they are singular; but an irregular noun such as *feet* would have a feature abbreviation such as *pl* in its lexical entry. This would be defined in the grammar as `[number: PL]`, and would override the default value for the feature number specified by the template above. If the N template above used `SG` instead of `!SG`, then the word *feet* would fail to parse, since its *number* feature would have an internal conflict between `SG` and `PL`.

## 2.3 Parameter settings

Parameter settings are used to override various default settings assumed in the grammar file. Parameter settings are optional. In the absence of a parameter setting, a default value is used. A parameter setting has these parts, in the order listed:

1. the keyword `Parameter`
2. an optional colon (`:`)
3. one or more keywords identifying the parameter
4. the keyword `is`
5. the parameter value
6. an optional period (`.`)

PC-KIMMO recognizes the following parameters:

- *Start symbol* defines the start symbol of the grammar. For example,

```
Parameter Start symbol is Word
```

declares that the parse goal of the grammar is the nonterminal category `Word`. The default start symbol is the left hand symbol of the first context-free rule in the grammar file.

- *Attribute order* specifies the order in which feature attributes are displayed. For example,

```
Parameter Attribute order is cat head root root_pos
```

declares that the *cat* attribute should be the first one shown in any output from PC-KIMMO and that the other attributes should be shown in the relative order shown, with the *root\_pos* attribute shown last among those listed, but ahead of any attributes that are not listed above. Attributes that are not listed are ordered according to their character code sort order. If the attribute order is not specified, then the category feature *cat* is shown first, with all other attributes sorted according to their character codes.

- *Category feature* defines the label for the category attribute. For example,

```
Parameter Category feature is Categ
```

declares that *Categ* is the name of the category attribute. The default name for this attribute is *cat*

- *Lexical feature* defines the label for the lexical attribute. For example,

```
Parameter Lexical feature is Lex
```

declares that *Lex* is the name of the lexical attribute. The default name for this attribute is *lex*

- *Gloss feature* defines the label for the gloss attribute. For example,

```
Parameter Gloss feature is Gloss
```

declares that *Gloss* is the name of the gloss attribute. The default name for this attribute is *gloss*.

## 2.4 Lexical rules

Lexical rules are used to modify the feature structures of lexical entries. As noted in Shieber 1985, something more powerful than just abbreviations for common feature elements is sometimes needed to represent systematic relationships among the elements of a lexicon. This need is met by lexical rules, which express transformations rather than mere abbreviations.

Lexical rules are similar to feature templates, but are more powerful. While feature templates assign a feature structure to lexical items by means of unification, lexical rules map one feature structure to another, thus transforming it. The name of a lexical rule is included in the features field of lexical entries, similar to feature abbreviations.

A lexical rule has these parts, in the order listed:

1. the keyword `Define`
2. the name of the lexical rule
3. the keyword `as`
4. the rule definition
5. an optional period (.)

The rule definition consists of one or more mappings. Each mapping has three parts: an output feature path, an assignment operator, and the value assigned, either an input feature path or an atomic value. Every output path begins with the feature name `out` and every input path begins with the feature name `in`. The assignment operator is either an equal sign (=) or an equal sign followed by a "greater than" sign (=>). (These two operators are equivalent in PC-KIMMO, since the implementation treats each lexical rule as an ordered list of assignments rather than using unification for the mappings that have an equal sign operator.) Consider the information shown in figure 4.8A.

**Figure 4.8A** A lexical rule example

```
;lexical item
\lf `mouse
\fea irreg POS_Gloss
\gl `mouse

;feature template
LET irreg be = -

;lexical rule
DEFINE POS_Gloss as
    =
    =
    =
    = .
```

The feature field (`\fea`) of the lexical entry contains two labels: *irreg* is a feature abbreviation and is defined by a feature template (the `LET` statement), while *POS\_Gloss* is the name of a lexical rule which is defined by the `DEFINE` statement.

**Figure 4.8B** Feature structure before application of lexical rule

```
[ cat:    ROOT
  head:   [ agr: [ 3sg:- ]
            number:PL
```

```

        pos:    N
        proper:-
        verbal:- ]
reg:    -
lex:    `mice
gloss:  `mouse ]

```

**Figure 4.8C** Feature structure after application of lexical rule

```

[ cat:    ROOT
  head:   [ agr: [ 3sg:- ]
            number:PL
            pos:    N
            proper:-
            verbal:- ]
  lex:    `mice
  gloss:  N ]

```

When the lexicon entry is loaded, it is initially assigned the feature structure shown in figure 4.8B, which is the unification of the information given in the various fields of the lexicon entry, including the feature abbreviation *pl*. After the complete feature structure has been built, the lexical rule named *POS\_Gloss* is applied, producing the feature structure shown in figure 4.8C. Note that the change in the value of the gloss feature from "*`mouse*" to "*N*" is done by direct mapping, not unification.

There are two important points about using lexical rules. First, the feature structure of a lexical item that has undergone a lexical rule is entirely determined by the mappings in the lexical rule. In the lexical rule in figure 4.8A, the first three mappings (for *cat*, *head*, and *lex*), though they seem redundant, are needed to carry over these feature values from the input feature structure to the output feature structure. Notice that the feature *reg* which is present in the input feature structure in figure 4.8B is absent from the output feature structure in figure 4.8C; this is due to the fact that the lexical rule which applied to the feature structure did not include a mapping for the *reg* feature.

Second, lexical rules apply sequentially in the order in which they are given in the grammar file.

Figure 4.9 shows a sample grammar file.

**Figure 4.9** A sample grammar file

```

;FEATURE TEMPLATES (optional)

;Feature definitions
Let pl be    <head number> = PL
LET v/n be   <from_pos> = V
             <head pos> = N
             <head number> = !SG
LET v\aj be  <from_pos> = AJ
             <head pos> = V

;Category definitions
Let N be     <cat> = ROOT
             <head pos> = N
             <head number> = !SG
Let V be     <cat> = ROOT
             <head pos> = V
Let AJ be    <cat> = ROOT
             <head pos> = AJ

;PARAMETER SETTINGS (optional)

PARAMETER Start symbol is Word

;RULES

```

```

RULE
Word = Stem INFL
    <Stem head pos> = <INFL from_pos>
    <Word head> = <INFL head>

RULE
Stem_1 = PREFIX Stem_2
    <PREFIX from_pos> = <Stem_2 head pos>
    <Stem_1 head> = <PREFIX head>

RULE
Stem_1 = Stem_2 SUFFIX
    <Stem_2 head pos> = <SUFFIX from_pos>
    <Stem_1 head> = <SUFFIX head>

RULE
Stem = ROOT
    <Stem head> = <ROOT head>

```

## 4.7.4 Generation comparison file

The generation comparison file serves as input to the *compare generate* command (see section 4.5.12). It consists of groupings of a lexical form followed by one or more surface forms that are expected to be generated from the lexical form. The following specifications apply to the generation comparison file.

- Each form must be on a separate line.
- Leading spaces are ignored.
- A blank line (or end of file) indicates the end of a grouping. Extra blank lines are ignored.
- The first form in each grouping is the lexical form to be input to the generator. Its gloss does not have to be included, since the generator does not use the lexicon; however, including a gloss with the lexical form does no harm--it is simply ignored.
- Succeeding forms in each grouping are surface forms that are the expected output of the generator.

Figure 4.10 shows a sample generation comparison file.

**Figure 4.10** A sample generation comparison file

```

`trace+ed
traced

`trace+able
traceable

re-+`trace
re-trace
retrace

```

## 4.7.5 Recognition comparison file

The recognition comparison file serves as input to the *compare recognize* command (see section 4.5.12). It consists of groupings of a surface form followed by one or more lexical forms that are expected to be recognized from the surface form. The following specifications apply to the recognition comparison file.

- Each form must be on a separate line.
- Leading spaces are ignored.

- A blank line (or end of file) indicates the end of a grouping. Extra blank lines are ignored.
- The first form in each grouping is the surface form to be input to the recognizer.
- Succeeding forms in each grouping are lexical forms that are the expected output of the recognizer. The gloss of a form follows it on the same line, separated by one or more spaces. The gloss must match exactly (including spaces) the way it is output from the recognizer.

Figure 4.11 shows a sample recognition comparison file.

**Figure 4.11** A sample recognition comparison file

```
traced
`trace+ed      [ V(trace)+PAST ]
`trace+ed      [ V(trace)+PAST.PRTC ]

traceable
`trace+able    [ V(trace)+ADJR ]

retrace
re-+`trace     [ REP+V(trace).INF ]
```

## 4.7.6 Pairs comparison file

The pairs comparison file serves as input to the *compare pairs* command (see section 4.5.12). It consists of pairs of lexical and surface forms; that is, a lexical form followed by exactly one surface form. It is expected that the surface form will be recognized from the lexical form and that the lexical form will be generated from the surface form. Glosses do not have to be included with lexical forms, since the generator does not use the lexicon; however, including a gloss with the lexical form does no harm--it is simply ignored. When recognizing a surface form, the lexicon is used to identify the constituent morphemes and verify that they occur in the correct order, but the gloss part of a lexical entry is not used. The following specifications apply to the pairs comparison file.

- Each form must be on a separate line.
- Leading spaces are ignored.
- A blank line (or end of file) indicates the end of a grouping. Extra blank lines are ignored.
- The first form of a pair is the lexical form, which is input to the generator. It is the expected output on inputting the second (surface) form to the recognizer. The gloss is not included with the lexical form.
- The second form of a pair is the surface form, which is input to the recognizer. It is the expected output on inputting the first (lexical) form to the generator.

Figure 4.12 shows a sample pairs comparison file.

**Figure 4.12** A sample pairs comparison file

```
`trace+ed
traced

`trace+able
traceable

re-+`trace
re-trace

re-+`trace
retrace
```



## 4.7.6A Synthesis comparison file

The synthesis comparison file serves as input to the *compare synthesize* command (see section 4.5.12). It consists of groupings of a morphological form followed by one or more surface forms that are expected to be synthesized from the morphological form. The following specifications apply to the synthesis comparison file.

- Each form must be on a separate line.
- Leading spaces are ignored.
- A blank line (or end of file) indicates the end of a grouping. Extra blank lines are ignored.
- The first form in each grouping is the morphological form to be input to the synthesizer. A morphological form is a sequence of morpheme glosses separated by spaces.
- Succeeding forms in each grouping are surface forms that are the expected output of the generator.

Figure 4.12A shows a sample synthesis comparison file.

**Figure 4.12A** A sample synthesis comparison file

```
`trace +ED
traced

`trace +EN
traced

`trace +AJR25a
traceable

ORD5+ `trace
retrace
```

## 4.7.7 Generation file

The generation file consists of a list of lexical forms. It serves as input to the *file generate* command (see section 4.5.13), which returns a file (or screen display) whose format is identical to the generation comparison file. The following specifications apply to the generation file.

- Each form must be on a separate line.
- Extra white space, blank lines, and comment lines are ignored.
- Each form is assumed to be a lexical form. If a gloss is included, it is ignored.

Figure 4.13 shows a sample generation file.

**Figure 4.13** A sample generation file

```
`cat
`cat+s
`cat+'s
`cat+s+'s
`fox
`fox+s
`fox+'s
`fox+s+'s
```

## 4.7.8 Recognition file

The recognition file consists of a list of surface forms. It serves as input to the *file recognize* command (see section 4.5.14), which returns a file (or screen display) whose format is identical to the recognition comparison file. The following specifications apply to the recognition file.

- Each form must be on a separate line.
- Extra spaces, blank lines, and comment lines are ignored.
- Each form is assumed to be a surface form.

Figure 4.14 shows a sample recognition file.

**Figure 4.14** A sample recognition file

```
cat
cats
cat's
cats'
fox
foxes
fox's
foxes'
```

## 4.7.8A Synthesis file

The synthesis file consists of a list of morphological forms. A morphological form is a sequence of morpheme glosses separated by spaces. A synthesis file serves as input to the *file synthesis* command (see section 4.5.13), which returns a file (or screen display) whose format is identical to the synthesis comparison file. The following specifications apply to the synthesis file.

- Each form must be on a separate line.
- Extra white space, blank lines, and comment lines are ignored.
- Each form is assumed to be a morphological form.

Figure 4.14A shows a sample synthesis file.

**Figure 4.14A** A sample synthesis file

```
`cat
`cat +PL
`cat +GEN
`cat +PL +GEN
`fox
`fox +PL
`fox +GEN
`fox +PL +GEN
```

## 4.7.9 Summary of default file names and extensions

Figure 4.15 summarizes the default file names and extensions assumed by PC-KIMMO. Two entries are given for the different kinds of files. The first is the name PC-KIMMO will assume if no file name at all is given to a command that expects that kind of file. The second entry (with the \*) shows what extension PC-KIMMO will add if a file name without an extension is given.

**Figure 4.15** Default file names and extensions

```
Rules file:                RULES.RUL
                           *.RUL
```

Lexicon file:	LEXICON.LEX *.LEX
Grammar file:	GRAMMAR.GRM *.GRM
Generation comparison file:	DATA.GEN *.GEN
Recognition comparison file:	DATA.REC *.REC
Pairs comparison file:	DATA.PAI *.PAI
Synthesis comparison file:	DATA.SYN *.SYN
Take file:	PCKIMMO.TAK *.TAK
Log file:	PCKIMMO.LOG *.LOG

## 4.8 Trace formats

---

### 4.8.1 Generator trace

### 4.8.2 Recognizer trace

**Figure 4.16 Level 1 generator trace**

**Figure 4.17 Level 2 generator trace**

**Figure 4.18 Level 3 generator trace**

**Figure 4.19 Level 1 recognizer trace**

**Figure 4.20 Level 2 recognizer trace**

**Figure 4.21 Level 3 recognizer trace**

---

This section explains how to read the output of the generator and recognizer traces. Traces are produced by the *set tracing* command described in section 4.5.6.1. The amount of detail shown in the trace display is set by the tracing level. The *level* argument to the *set tracing* command can range from 0 to 3, where 0 is no tracing at all and 3 is the most detailed level of tracing.

### 4.8.1 Generator trace

The purpose of the generator trace is to allow the user to see how a lexical form is processed through multiple recursive calls to the generator. The generator algorithm used to process the form is described in section 4.9.1.

There are three levels of tracing differing in the amount of detail they display: Level 1 gives the least amount of detail, level 2 (the default) gives a moderate amount of detail, and level 3 gives the most detail. Figure 4.16 is a level 1 generator trace of the lexical form *fox+s* (taken from the English example). The only difference from no tracing at all is that the RESULT line is displayed. This line differs from the normal result that is returned because it prints all NULL symbols in the output surface form.

**Figure 4.16** Level 1 generator trace

`fox+s

RESULT = 0fox0es

foxes

Figure 4.17 is from a level 2 generator trace for the form *`fox+s*. To limit the size of the trace, the Geminatio rules (14 and 15) were turned off. Line numbers and column numbers are printed here for reference in the description that follows. Each description refers to an element beginning at the line and column indicated.

**Figure 4.17** Level 2 generator trace

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
1	`fox+s																	
2	0	#:	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	
3	0	`:	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	
4	1	ff	1	1	1	1	1	1	1	2	2	1	1	1	1	1	1	0
5	2	o:o	1	1	1	1	2	2	1	3	3	1	1	1	1	1	1	0f
6	3	x:x	1	1	1	1	1	1	1	7	4	2	1	1	1	1	1	0fo
7	4	+:	1	1	3	3	2	2	1	4	4	1	1	1	1	1	1	0fox
8	5	s:s	1	1	5	5	1	1	2	4	4	1	1	1	1	1	1	0fox0
9	6	#:	1	1	6	2	2	2	3	3	4	1	1	1	1	1	1	0fox0s
10	6-	BLOCKED BY RULE 3: Epenthesis, 0:0 /<= [S ch sh y:i] +:0____s[+:0 #]																
11	5<		1	1	5	5	1	1	2	4	4	1	1	1	1	1	1	0fox0
12	5	s:0	1	1	5	5	1	1	2	4	4	1	1	1	1	1	1	0fox0
13	5-	BLOCKED BY RULE 7: S-deletion, s:0 <=> +:0 (0:e) s +:0 '____																
14	5	0:e	1	1	5	5	1	1	2	4	4	1	1	1	1	1	1	0fox0
15	6	s:s	1	1	1	6	1	1	2	4	4	1	1	1	1	1	1	0fox0e
16	7	#:	1	1	4	7	2	2	3	3	4	1	1	1	1	1	1	0fox0es
17	7		1	1	1	1	1	1	1	3	4	1	1	1	1	1	1	0fox0es
18																		
19	RESULT = 0fox0es																	
20																		
21	6<		1	1	1	6	1	1	2	4	4	1	1	1	1	1	1	0fox0e
22	6	s:0	1	1	1	6	1	1	2	4	4	1	1	1	1	1	1	0fox0e
23	6-	BLOCKED BY RULE 4: Epenthesis, 0:e => [S ch sh y:i] +:0____s[+:0 #]																
24	6	0:e	1	1	1	6	1	1	2	4	4	1	1	1	1	1	1	0fox0e
25	6-	BLOCKED BY RULE 4: Epenthesis, 0:e => [S ch sh y:i] +:0____s[+:0 #]																
26	5<		1	1	5	5	1	1	2	4	4	1	1	1	1	1	1	0fox0
27	4<		1	1	3	3	2	2	1	4	4	1	1	1	1	1	1	0fox
28	4	0:e	1	1	3	3	2	2	1	4	4	1	1	1	1	1	1	0fox
29	4-	BLOCKED BY RULE 4: Epenthesis, 0:e => [S ch sh y:i] +:0____s[+:0 #]																
...																		
39	0<		1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	
40	0	0:e	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	
41	0-	BLOCKED BY RULE 4: Epenthesis, 0:e => [S ch sh y:i] +:0____s[+:0 #]																
42	foxes																	

- Line 1: Input line. Lexical form input to the generator function.

- Line 19: RESULT line. Surface form produced by the generator function. At the point where the input lexical form is empty and each automaton is in a final state, the trace shows that the generator has recorded a result. The generator continues looking for additional results (lines 21 and following).
- Column 1: Level number (all lines except 1, 19, and 42). This represents the level of recursion. Level 0 represents the initial invocation of the generator. Notice that the number coincides with the number of characters in the result string so far.
- Column 1: Backtracking indicator (lines 10, 11, 13). The symbol - indicates that the generator is blocked at that level. The symbol < indicates that the generator is backtracking (that is, returning to a lower level to try another path).
- Column 2: Input pair (lines 2-9, 12, 14-16). This is the lexical:surface pair (from the set of feasible pairs) that is currently being considered by the generator (for example, *f:f* on line 4). The rest of the line shows the results of stepping the automata with the pair as input. The results are indicated by either a new state configuration (for example, line 5) or a BLOCKED BY RULE message (for example, line 10).
- Lines 10, 13: BLOCKED BY RULE message. Indicates that a feasible pair input to the function that steps the automata caused a rule to fail. Gives the number and name of the rule (from the header line of the state table) that failed.
- Columns 3-17: State configuration (lines 2-9, 11-12, 14-17). These are the current states of each of the rules. The leftmost number is the state of rule 1, the second is rule 2, and so on.
- Column 18: Result (lines 4-9, 11-12, 14-17). This is the current value of the result string.
- Lines 21-41: The generator continues to backtrack, looking for other possible paths to a result, until finding no other path it returns to its initial state.

There is one other tracing message not exemplified in the above display. This is the END OF INPUT message. It indicates that the end of the input form has been reached but the generator function has failed on the rule specified because it was not in a final state. For example,

```
END OF INPUT, FAILED RULE 4: Palatalization
```

would indicate that when the end of the input form was reached, rule 4 was not left in a final state.

**Figure 4.18** Level 3 generator trace

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
1	`fox+s																	
2	0	#:#	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	
3	0	`0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	
4	1	f:f	1	1	1	1	1	1	1	2	2	1	1	1	1	1	1	0
5	2	o:o	1	1	1	1	2	2	1	3	3	1	1	1	1	1	1	Of
6	3	x:x	1	1	1	1	1	1	1	7	4	2	1	1	1	1	1	Ofo
7	4	+:0	1	1	3	3	2	2	1	4	4	1	1	1	1	1	1	Ofox
8	5	s:s	1	1	5	5	1	1	2	4	4	1	1	1	1	1	1	Ofox0
9	6	#:#	1	1	6	2	2	2	3	3	4	1	1	1	1	1	1	Ofox0s
10	6-		1	1	0	?	?	?	?	?	?	?	?	?	?	?	?	Ofox0s
11		BLOCKED BY RULE 3: Epenthesis, 0:0 /<= [S ch shly:i] +:0 ____s[:0 #]																
12	5<		1	1	5	5	1	1	2	4	4	1	1	1	1	1	1	Ofox0
13	5	s:0	1	1	5	5	1	1	2	4	4	1	1	1	1	1	1	Ofox0
14	5-		1	1	1	1	1	1	0	?	?	?	?	?	?	?	?	Ofox0
15		BLOCKED BY RULE 7: S-deletion, s:0 <=> +:0 (0:e) s+:0 ' ____																
16	5	0:e	1	1	5	5	1	1	2	4	4	1	1	1	1	1	1	Ofox0
17	6	s:s	1	1	1	6	1	1	2	4	4	1	1	1	1	1	1	Ofox0e
18	7	#:#	1	1	4	7	2	2	3	3	4	1	1	1	1	1	1	Ofox0es
19	7		1	1	1	1	1	1	1	3	4	1	1	1	1	1	1	Ofox0es
20																		
21		RESULT = Ofox0es																
22																		
23		foxes																

Figure 4.18 is part of a level 3 trace for the same form. The level 3 trace differs from the level 2 trace in how it displays rule failures that block the generator. Compare line 10 in the level 2 trace with lines 10 and 11 of the level 3 trace. The level 3 trace explicitly shows what state the automata are in after stepping them. In line 10 of the level 3 trace we can see that the proposed input pair puts rule 3 in state 0, which means that it fails. Notice that the rest of the state array is filled with question marks. This is because if one rule fails the whole configuration fails, so the rest of the rules are not even tried. (This shows that even though conceptually the automata operate in parallel they must still be stepped one at a time).

## 4.8.2 Recognizer trace

The purpose of the recognizer trace is to allow the user to see how a surface form is processed through multiple recursive calls to the recognizer. The recognizer algorithm used to process the form is described in section 4.9.1.

There are three levels of tracing differing in the amount of detail they display: level 1 gives the least amount of detail, level 2 (the default) gives a moderate amount of detail, and level 3 gives the most detail.

**Figure 4.19** Level 1 recognizer trace

```

foxes
  ENTERING LEXICON INITIAL
  ENTERING LEXICON N_ROOT
  ENTERING LEXICON NUMBER
  ENTERING LEXICON GENITIVE
  ENTERING LEXICON End

  RESULT = `fox+0s [ N(fox)+ PL ]

  BACKING UP FROM LEXICON End TO LEXICON GENITIVE
  BACKING UP FROM LEXICON GENITIVE TO LEXICON NUMBER
  ENTERING LEXICON GENITIVE
  ENTERING LEXICON End
  BACKING UP FROM LEXICON End TO LEXICON GENITIVE
  BACKING UP FROM LEXICON GENITIVE TO LEXICON NUMBER
  BACKING UP FROM LEXICON NUMBER TO LEXICON N_ROOT
  BACKING UP FROM LEXICON N_ROOT TO LEXICON INITIAL
  ENTERING LEXICON ADJ_PREFIX

...
  BACKING UP FROM LEXICON V_ROOT_NEG TO LEXICON V_PREFIX
  BACKING UP FROM LEXICON V_PREFIX TO LEXICON INITIAL
`fox+s [ N(fox)+ PL ]

```

Figure 4.19 is a level 1 recognizer trace of the surface form *foxes* (taken from the English example). Like the level 1 generator trace, the level 1 recognizer trace displays the RESULT line but does not show the feasible pairs as they are tried or the states of the rules. However, it does display a record of how the recognizer moves through the lexicon, either with an ENTERING } or a BACKING UP message.

Figure 4.20 is from a level 2 recognizer trace of the form *foxes*. To limit the size of the trace, the Gemination rules (14 and 15) were turned off. Line numbers and column numbers are printed here for reference in the description that follows. Each description refers to an element beginning at the line and column indicated.

**Figure 4.20** Level 2 recognizer trace

```

1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18
1 foxes
2 0  #:# 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
3 ENTERING LEXICON INITIAL
4 ACCEPTING NULL ENTRY
5 ENTERING LEXICON N_ROOT
6 0 `:0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 [
7 1 s:0 1 1 1 1 1 1 1 2 2 1 1 1 1 1 1 `[
8 1- BLOCKED BY RULE 7: S-deletion, s:0 <=> +:0 (0:e) s+:0 '____
9 1 ff 1 1 1 1 1 1 1 2 2 1 1 1 1 1 1 `[
10 2 o:o 1 1 1 1 2 2 1 3 3 1 1 1 1 1 1 `f[
11 3 x:x 1 1 1 1 1 1 1 7 4 2 1 1 1 1 1 1 `fo[
12 ENTERING LEXICON NUMBER
13 4 +:0 1 1 3 3 2 2 1 4 4 1 1 1 1 1 1 `fox [ N(fox)
14 5 s:0 1 1 5 5 1 1 2 4 4 1 1 1 1 1 1 `fox+ [ N(fox)
15 5- BLOCKED BY RULE 7: S-deletion, s:0 <=> +:0 (0:e) s+:0 '____
16 5 0:e 1 1 5 5 1 1 2 4 4 1 1 1 1 1 1 `fox+ [ N(fox)
17 6 s:0 1 1 1 6 1 1 2 4 4 1 1 1 1 1 1 `fox+0 [ N(fox)
18 6- BLOCKED BY RULE 4: Epenthesis, 0:e => [Slch!shly:i] +:0 ____s[+:0!#]
19 6 s:s 1 1 1 6 1 1 2 4 4 1 1 1 1 1 1 `fox+0 [ N(fox)
20 ENTERING LEXICON GENITIVE
21 7 +:0 1 1 4 7 2 2 3 3 4 1 1 1 1 1 1 `fox+0s [ N(fox)+PL
22 8- BLOCKED IN LEXICON GENITIVE, INPUT =
23 7< 1 1 4 7 2 2 3 3 4 1 1 1 1 1 1 `fox+0s [ N(fox)+PL
24 ACCEPTING NULL ENTRY
25 ENTERING LEXICON End
26 ACCEPTING NULL ENTRY
27 7 #:# 1 1 4 7 2 2 3 3 4 1 1 1 1 1 1 `fox+0s [ N(fox)+PL
28 7 1 1 1 1 1 1 1 3 4 1 1 1 1 1 1 `fox+0s [ N(fox)+PL
29
30 RESULT = `fox+0s [ N(fox)+PL ]
...
108 BACKING UP FROM LEXICON V_ROOT_NEG TO LEXICON V_PREFIX
109 0< 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 [
110 BACKING UP FROM LEXICON V_PREFIX TO LEXICON INITIAL
111 0< 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
112 `fox+s [ N(fox)+PL ]

```

- Line 1: Input line. Surface form input to the recognizer function.
- Line 30: RESULT line. At the point where there are no lexicons in the continuation class of an entry, the input surface form is empty, and each automaton is in a final state, the trace shows that the recognizer has recorded a result. The recognizer continues looking for additional results (lines 32 and following).
- Column 1: Level number (lines 2, 6-11, 13-19, 21-23, 27-28). This represents the level of recursion. Level 0 represents the initial invocation of the recognizer. Notice that the number coincides with the number of characters in the result string so far.
- Column 1: Backtracking indicator (lines 8, 15, 18, 22-23). The symbol - indicates that the recognizer is blocked at that level. The symbol < indicates that the recognizer is backtracking (that is, returning to



a lower level to try another path).

- Column 2: Input pair (lines 2, 6-7, 9-11, and so on). This is the lexical:surface pair (from the set of feasible pairs) that is currently being considered by the recognizer (for example, *f:f* on line 9). The results of stepping the automata with the pair as input are indicated by either a new state configuration (for example, line 10) or a BLOCKED BY RULE message (for example, line 15).
- Lines 3, 5, 12, 20, 25: ENTERING LEXICON message. This is the name of the sublexicon that the recognizer is about to search.
- Lines 4, 24, 26: ACCEPTING NULL ENTRY message. message} Indicates that a null lexical entry (that is, an entry whose lexical item is the NULL symbol) has been accepted.
- Line 22: BLOCKED IN LEXICON message. Indicates that no lexical entry could be found in the current lexicon that continues with the input pair under consideration. The remaining part of the input form is displayed on the line (in line 22 it happens that nothing is left of the input form).
- Lines 108, 110: BACKING UP message. Indicates that there were no further sublexicons left in the continuation class, so the recognizer must back up to the previous lexicon branch.
- Lines 8, 15, 18: BLOCKED BY RULE message. Indicates that a feasible pair input to the function that steps the automata caused a rule to fail. Gives the number and name of the rule (from the header line of the state table) that failed.
- Columns 3-17: State configuration (lines 2, 6-7, 9-11, and so on). These are the current states of each of the rules. The leftmost number is the state of rule 1, the second is rule 2, and so on.
- Column 18: Result (lines 6-7 and so on). This is the current value of the result string.
- Lines 108-111: The recognizer continues to backtrack, looking for other possible paths to a result, until finding no other path it returns to its initial state.

The END OF INPUT message may also occur in a recognizer trace. See section 4.8.1 on the generator trace for an explanation of it.

**Figure 4.21** Level 3 recognizer trace

```

1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18
1 foxes
2 0  #:# 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
3      ENTERING LEXICON INITIAL
4 0- -0 LEXICAL CHARACTER NOT MATCHED
5 0- `0 LEXICAL CHARACTER NOT MATCHED
6 0- +0 LEXICAL CHARACTER NOT MATCHED
7 0- s0 LEXICAL CHARACTER NOT MATCHED
8 0- e0 LEXICAL CHARACTER NOT MATCHED
9 0- ff LEXICAL CHARACTER NOT MATCHED
10      ACCEPTING NULL ENTRY
11      ENTERING LEXICON N_ROOT
12 0- -0 LEXICAL CHARACTER NOT MATCHED
13 0  `0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 [
14 1- -0 LEXICAL CHARACTER NOT MATCHED
15 1- `0 LEXICAL CHARACTER NOT MATCHED
16 1- +0 LEXICAL CHARACTER NOT MATCHED
17 1  s0 1 1 1 1 1 1 1 2 2 1 1 1 1 1 1 ` [
18 1- 1 1 1 1 1 1 0 ? ? ? ? ? ? ? ? ` [
19      BLOCKED BY RULE 7: S-deletion, s:0 <=> +:0 (0:e) s+:0 '____
...
75      ACCEPTING NULL ENTRY
76 7  #:# 1 1 4 7 2 2 3 3 4 1 1 1 1 1 1 `fox+0s [ N(fox)+PL
77 7 1 1 1 1 1 1 1 3 4 1 1 1 1 1 1 `fox+0s [ N(fox)+PL
78
79 RESULT = `fox+0s [ N(fox)+PL ]
80
81 `fox+s [ N(fox)+PL ]

```

Figure 4.21 is part of a level 3 trace for the same form. Like level 3 of the generator trace, level 3 of the recognizer trace explicitly shows the state array when a rule fails. Compare line 8 of the level 2 trace with lines 18 and 19 of the level 3 trace. In addition, the level 3 recognizer trace shows pairs that are weeded out by the lexicon even before they are tried with the rules. Compare lines 3-4 of the level 2 trace with lines 3-10 of the level 3 trace. In lines 4-9 the level 3 trace shows explicitly several pairs that are tried but immediately fail. Since the recognizer is at the beginning of the input form, the only possible feasible pairs to try are those whose surface character is 0 (the NULL symbol) or *f* (the first character of the input form *foxes*). Rather than trying each of these pairs with the rules, the recognizer first looks to see if the lexical character of each pair matches any lexical character available in the sublexicon it is currently searching. In each case the match fails, indicated by the message LEXICAL CHARACTER NOT MATCHED. After trying all the pairs, the lexicon accepts the null entry and enters a new sublexicon. This exhaustive process takes place at each point in the recognition process where the recognizer is trying a new pair.

## 4.9 Algorithms

---

### 4.9.1 The Generator

### 4.9.2 The Recognizer

#### Figure 4.22 A lexical letter tree

---

The algorithms used by PC-KIMMO to generate and recognize are based on descriptions in Karttunen 1983. These algorithms pertain only to the rules and lexical components, not the word grammar component.

## 4.9.1 The Generator

The generator function recursively computes surface forms from a lexical form using a set of two-level rules expressed as finite state automata. The generator function does not make use of the lexicon. This means that it will accept input forms that are not found in the lexicon or that even violate the lexicon's constraints on morpheme order, and will still apply the phonological rules to them. To produce a surface form from a lexical form, the generator processes the input form one character at a time, left to right. For each lexical character, it tries every surface character that has been declared as corresponding to it in a feasible pair sanctioned by the description. The generator function has these inputs:

**Lexical form:**

Initially the input form, this string contains whatever is left to process. As the function is recursively called, this string gets shorter as the result string gets longer.

**Result:**

Initially empty, this string contains the results of the generator up to the point of the current function call.

**Rules:**

This is the set of active finite state automata defined for this language.

**Configuration:**

This is an array representing the current state of all rules (automata). Initially, all states are set to 1.

The generator function also uses a list of **feasible pairs** sanctioned by the set of rules; these are all the lexical:surface pairs of alphabetic characters that appear as column headers in the state tables. The **input pair** is a feasible pair selected by the generator as a possible next lexical:surface pair in the process of computing a **surface form** that corresponds to the given **lexical form**. Each time the generator is called it iteratively goes through the list of feasible pairs, selecting one as the input pair.

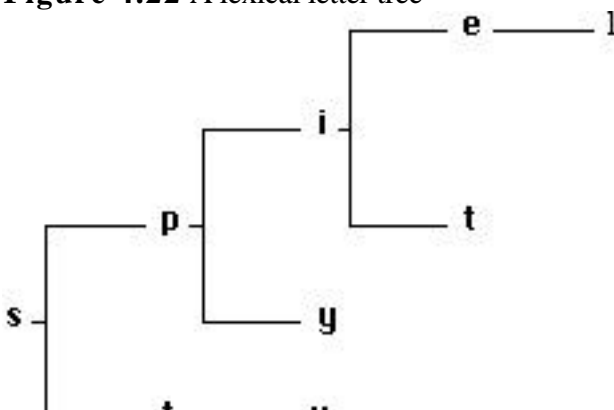
The generator algorithm works as follows:

1. If the **lexical form** is empty (that is, there are no more characters in it to process), do the following steps:
  1. If any of the state tables contains a word boundary column header, step the automata using an input pair consisting of the BOUNDARY symbol as both the lexical and surface character. If this fails, then the **result** is rejected and the function returns to the previous level.
  2. Check that the **configuration** array contains a valid final state for each of the **rules**. If so, then the **result** is accepted and added to the output list. Otherwise, it is rejected. In either case, the function returns to the previous level.

Otherwise, if the lexical form is not empty (that is, it contains more characters to process), do steps 2 and 3.

2. For each input pair containing the first character in the **lexical form** as the lexical character, do the following steps:
  1. Step the automata using the input pair and the input **configuration** array, producing a new **configuration**.
  2. If this succeeds, recursively call the generator function with these inputs:

**Lexical form:**



Besides applying the phonological rules and identifying morphemes, the recognizer also must enforce morpheme order constraints. The PC-KIMMO lexicon is divided into classes of lexical items that behave alike with respect to order constraints. These lexical classes are called *sublexicons*. The entry for each lexical item specifies the name of the sublexicon that can follow it. This following sublexicon is called a *continuation class*. Lexical items that occur only at the end of a word have no continuation class, indicated by the BOUNDARY symbol.

The names of the sublexicons that make up the entire lexicon are used as nodes at the head of branches of the letter tree. The piece of a letter tree shown in Figure 4.22 may actually be under a branch node called Noun. When the recognizer successfully finds a lexical item in the letter tree, it looks at its specified continuation class and jumps to the branch of the lexicon it names.

It is often the case that at a given point in a word, more than one continuation is possible. Sets of alternative continuing sublexicons are called *alternations*. Thus the continuation class field of a lexical entry may contain the name of an alternation that specifies a list of the sublexicons that can follow it.

When the recognizer successfully recognizes a lexical item (word or morpheme), it reads its gloss from its lexical entry and appends it to the **gloss** string being built up for the entire word.

The recognizer function has these inputs:

**Surface form:**

Initially the input form, this string contains whatever is left to process. As the function is recursively called, this string gets shorter as the result string gets longer.

**Result:**

Initially empty, this string contains the results of the recognizer up to the point of the current function call.

**Gloss:**

Initially empty, this string contains glosses for the lexical items contained in the **result** string.

**Rules:**

This is the set of active finite state automata defined for this language.

**Configuration:**

This is an array representing the current state of all rules (automata). Initially, all states are set to 1.

**Lexicon:**

Initially, this is the entire lexicon defined for the language. During the process of recognition it is restricted to a branch of the lexicon.

Like the generator, the recognizer function uses a list of **feasible pairs** sanctioned by the set of rules; these are all the lexical:surface pairs of alphabetic characters that appear as column headers in the state tables. The **input pair** is a feasible pair selected by the recognizer as a possible next lexical:surface pair in the process of computing a **lexical form** that corresponds to the given **surface form**. Each time the recognizer is called it iteratively goes through the list of feasible pairs, selecting one as the input pair. When a complete lexical item has been recognized, the lexicon is at a **terminal node** of the letter tree. Terminal nodes have glosses and continuation classes attached to them. The recognizer algorithm is initialized as though it has successfully recognized a lexical item and the lexicon is at a terminal node pointing to a continuation class consisting of the INITIAL sublexicon. It then proceeds as follows:

1. If the input **lexicon** is at a terminal node, then for each sublexicon in the continuation class of that item, recursively call the recognizer function with these inputs:

**Surface form:**

This string contains whatever is left to process.

**Result:**

This string contains the results of the recognizer up to the point of the current function call.

**Gloss:**

This is the input gloss string with the gloss of the current lexical entry appended.

**Rules:**

This is the input set of rules.

**Configuration:**

This is the input configuration.

**Lexicon:**

This is the current continuation sublexicon.

If the continuation class of the lexical entry is empty (that is, the lexical item can only be followed by word boundary) and the input **surface form** is empty, do the following steps:

1. If any of the state tables contains a word boundary column header, step the automata using an input pair consisting of the BOUNDARY symbol as both the lexical and surface character. If this fails, then the **result** is rejected and the function returns to the previous level.
2. Check that the **configuration** array contains a valid final state for each of the **rules**. If so, then the **result** is accepted, the gloss of the lexical entry is appended to the **gloss**, and both the **result** and the **gloss** are added to the output list. Otherwise, the **result** is rejected. In either case, the function returns to the previous level.

If the continuation class of the lexical entry is empty but the **surface form** is not empty, the **result** is rejected and the function returns to the previous level.

2. For each input pair that has the head of a branch in the lexicon as the lexical character and the first character of the **surface form** as the surface character, do the following steps:
  1. Step the automata using the input pair and the input **configuration** array to produce a new **configuration**.
  2. If this succeeds, recursively call the recognizer function with these inputs:

**Surface form:**

This is the input surface form with the first character removed.

**Result:**

This is the input result string with the lexical character from the current input pair appended.

**Gloss:**

This is the input gloss string.

**Rules:**

This is the input set of rules.

**Configuration:**

This is the state array produced by stepping the automata.

**Lexicon:**

This is the branch of the lexicon corresponding to the lexical character from the current input pair.

For each input pair that has the head of a branch in the lexicon as the lexical character and has the NULL symbol as the surface character, do the following steps:

1. Step the automata using the input pair and the input **configuration** array to produce a new **configuration**.
2. If this succeeds, recursively call the recognizer function with these inputs:

**Surface form:**

This is the input surface form.

**Result:**

This is the input result string with the lexical character from the current input pair appended.

**Gloss:**

This is the input gloss string.

**Rules:**

This is the input set of rules.

**Configuration:**

This is the state array produced by stepping the automata.

**Lexicon:**

This is the branch of the lexicon corresponding to the lexical character from the current input pair.

If the NULL symbol is the head of a branch of the lexicon (that is, a null lexical entry), recursively call the recognizer function with these inputs:

**Surface form:**

This is the input surface form.

**Result:**

This is the input result string.

**Gloss:**

This is the input gloss string.

**Rules:**

This is the input set of rules.

**Configuration:**

This is the input state array.

**Lexicon:**

This is the branch of the lexicon which has the NULL symbol as its head.

## 4.10 Messages

---

### 4.10.1 Messages related to reading and parsing commands

### 4.10.2 Messages related to reading the rules file

### 4.10.3 Messages related to reading the lexicon file

### 4.10.4 Messages related to reading the grammar file

## 4.10.5 Messages related to recognizing or generating a form

## 4.10.6 Messages related to memory

---

This section lists the various error and warning messages you may encounter. They are listed in numerical sequence and are generally grouped according to the type of error or warning. A warning means that the operation in progress has successfully completed, but an anomalous condition may have resulted. An error means that the operation in progress could not be successfully completed and was therefore prematurely terminated. Only in the case of a memory error is the PC-KIMMO program aborted and control returned to the operating system. Note that in the following error messages the words printed in italics are not literal but are cover terms for a set of items of the type suggested by the term. For instance, when the error message "Missing keyword in *command-name* command" actually appears on the computer screen, the term *command-name* will be replaced by a specific command name, such as *load* or *set*.

### 4.10.1 Messages related to reading and parsing commands

WARNING 100 Input line too long - ignoring after first *number* characters

ERROR 101 Ambiguous command: *command-name*

*command-name* did not specify a unique command. Type more of the command name to insure that it is not ambiguous.

ERROR 102 Invalid command: *command-name*

*command-name* is not a valid command. Type ? or *help* for a list of valid commands.

ERROR 103 Missing keyword in *command-name* command

Expected a keyword to be used with the command. Type the command name followed by ? for a list of valid keywords.

ERROR 104 Missing argument in *command-name* command

Expected an argument to complete the command. Type *help* followed by the command name for an explanation of what arguments the command needs.

ERROR 105 Ambiguous keyword in *command-name* command: *keyword*

*keyword* did not specify a unique keyword. Type more of the keyword to insure that it is not ambiguous.

ERROR 106 Invalid keyword in *command-name* command: *keyword*

*keyword* is not a valid keyword. Type the command name followed by ? for a list of valid keywords for that command.

ERROR 107 Invalid argument in *command-name* command: *argument*

*argument* was not valid for the command. Type *help* followed by the command name for an explanation of what arguments the command needs.

ERROR 108 Missing input file argument in *command-name* command

Expected a file name with the command.



ERROR 109 Cannot open input file *filename* in *command-name* command

Cannot find the file *filename*. Check to see if the file is in the current directory or the path you specified in the command. The command may also be expecting a different default file name or extension.

ERROR 110 Cannot open output file *filename* in *command-name* command

Check to see if the file is in the current directory or in the path you specified in the command. The command may also be expecting a different default file name or extension.

ERROR 111 Must load rules before loading lexicon

The rules file must be loaded before the lexicon in order to verify the lexical forms in the lexicon against the alphabet defined in the rules file.

ERROR 112 TAKE files nested too deeply

TAKE files can only be nested three deep.

ERROR 113 TAKE file aborted due to invalid command: *command-name*

*command-name* is not a valid command. Type ? or *help* for a list of valid commands.

ERROR 114 No log file was open

Result of issuing the CLOSE command when no log file has been opened.

WARNING 115 Closing the existing log file *filename*

Occurs when the LOG command is issued when a log file is already open.

ERROR 116 Missing file name for EDIT command

EDIT command must specify a file to be edited.

ERROR 117 Missing pathname for CD command

ERROR 118 Bad pathname for CD command

ERROR 119 No grammar loaded

## 4.10.2 Messages related to reading the rules file

ERROR 200 Rules file could not be opened: *filename*

Check to see if the file is in the current directory or in the path you specified in the command. The command may also be expecting a different default file name or extension.

ERROR 201 Unexpected end of rules file: *filename*

The rules file is incomplete. Check to see if the last table in the file has fewer states than expected.

ERROR 202 Expected ALPHABET keyword

The first declaration in a rules file must be the ALPHABET declaration.

ERROR 203 Alphabet contains no members

The ALPHABET keyword does not have any characters listed after it.

WARNING 204 Too many characters in the alphabet

The alphabet can contain a maximum of 255 characters.

WARNING 205 Character is already in the alphabet: *character*

A character has been repeated in the ALPHABET declaration.

ERROR 206 No value given for NULL keyword

A single character must appear after the NULL keyword.

ERROR 207 Value given for NULL symbol was already declared as alphabetic: *character*

The character specified for NULL may not also be declared in the ALPHABET.

ERROR 208 The NULL symbol has already been defined

There is more than one NULL declaration.

ERROR 209 Value given for NULL symbol was already declared for ANY

ERROR 210 Value given for NULL symbol was already declared for BOUNDARY

ERROR 211 No value given for ANY keyword

A single character must appear after the ANY keyword.

ERROR 212 Value given for ANY symbol was already declared as alphabetic: *character*

The character specified for ANY may not also be declared in the ALPHABET.

ERROR 213 The ANY symbol has already been defined

There is more than one ANY declaration.

ERROR 214 Value given for ANY symbol was already declared NULL

ERROR 215 Value given for ANY symbol was already declared for BOUNDARY

ERROR 216 No value given for BOUNDARY keyword

A single character must appear after the BOUNDARY keyword.

ERROR 217 Value given for BOUNDARY symbol was already declared as alphabetic: *character*

The character specified for BOUNDARY may not also be declared in the ALPHABET.

ERROR 218 The BOUNDARY symbol has already been defined

There is more than one BOUNDARY declaration.

ERROR 219 Value given for BOUNDARY symbol was already declared for NULL

ERROR 220 Value given for BOUNDARY symbol was already declared for ANY

ERROR 221 Subset name not given

Occurs if there is a SUBSET keyword with nothing after it until the next keyword.

ERROR 222 Subset name *subset-name* is not unique

A subset name, if it is a single character, cannot be the same as one of the characters specified in the ALPHABET, NULL, ANY, or BOUNDARY declarations. If the subset name is more than one character, then it is a duplicate of another subset name already declared.

ERROR 223 Subset *subset-name* contains no members

ERROR 224 Subset *subset-name* contains a nonalphabetic character: *character*

All characters used in subsets must be listed in the ALPHABET declaration, with the exception of the NULL symbol, which can appear in a subset but is not included in the ALPHABET list.

WARNING 225 Subset *subset-name* already contains *character*

A character has been repeated.

ERROR 226 Invalid keyword: *keyword*

The only valid keywords in a rules file are COMMENT, ALPHABET, NULL, ANY, BOUNDARY, SUBSET, RULE, and END.

WARNING 227 ANY symbol not defined

Are you sure the rules do not use an ANY symbol?

WARNING 228 NULL symbol not defined

Are you sure the rules do not use a NULL symbol?

WARNING 229 BOUNDARY symbol not defined

The BOUNDARY declaration is obligatory. Even if the BOUNDARY symbol is not used in the rules file, it must be used in the lexicon file.

WARNING 230 Missing closing delimiter for the name of a rule: *rule-name*

The first nonspace character after the RULE keyword is the opening delimiter of the rule name. A matching delimiter (identical character) was not found in the same line; thus PC-KIMMO will use everything up to the end of the line as the rule name. This is because the rule name must be contained in one line.

ERROR 231 Invalid number of rows: *number*

Must be a number greater than zero.

ERROR 232 Invalid number of columns: *number*

Must be a number greater than zero.

ERROR 233 Invalid state number: *number*

State (row) numbers must start with 1 and ascend consecutively.

ERROR 234 Expected final (:) or nonfinal (.) state indicator: *character*

A state (row) number must be followed by colon (:) or period (.) with no intervening space.

ERROR 235 State table entry out of range: *number*

*number* must not be greater than the specified number of states for the table.

ERROR 236 Lexical character not in alphabet: *character*

A character in a table's lexical character list is not a member of the alphabet declared earlier in the rules file.

ERROR 237 Surface character not in alphabet: *character*

A character in a table's surface character list is not a member of the alphabet declared earlier in the rules file.

ERROR 238 Nonnumeric character in state table: *character*

Expected a numeric state table entry but found a nonnumeric character.

ERROR 239 Rule number *number*, column *number* pairs a BOUNDARY symbol with something else: *column-header*

Occurs if a column header consists of a BOUNDARY symbol is paired with anything but another BOUNDARY symbol; only *#: #* is allowed.

WARNING 240 No feasible pairs for this set of rules

Either there are no rules in the file or the rules contain only subset correspondences. In the latter case, simple rules listing all the default correspondences are needed.

WARNING 241 RULE *number (rule-name)* - *char.char* specified by both columns *number (char.char)* and *number (char.char)*

There is an overlap between two columns of the state table. Issue a *show rule* command for the rule causing the warning and examine the set of pairs specified by each column header.

WARNING 242 RULE *number (rule-name)* - *char.char* not specified by any column

The entire set of feasible pairs must be specified by each table. The table is probably missing an ANY:ANY column.

ERROR 243 Rule number *number*, column *number* pairs two NULL symbols: *column-header*

NULL:NULL is not a legal column header, since it cannot be a feasible pair.

ERROR 244 No value given for COMMENT keyword

ERROR 245 Value given for COMMENT symbol was already declared as alphabetic: *character*

ERROR 246 Value given for COMMENT symbol was already declared for NULL

ERROR 247 Value given for COMMENT symbol was already declared for BOUNDARY

ERROR 248 Value given for COMMENT symbol was already declared for ANY

ERROR 249 Invalid keyword for twolc rules files: *keyword*

ERROR 250 SUBSETs not allowed in twolc rules file

ERROR 251 Missing opening delimiter for the name of a rule

ERROR 252 No rules defined: *filename*

ERROR 253 Invalid column number *number* in alignment for correspondence pair *char:char*

ERROR 254 Pair occurs in alignment twice: *char:char*

ERROR 255 Too few columns in alignment for correspondence pair *char:char*

ERROR 256 Too many columns in alignment for correspondence pair *char:char*

ERROR 257 RULE *number (name)* - no feasible pairs specified for column *number*

### 4.10.3 Messages related to reading the lexicon file

ERROR 300 Lexicon file could not be opened: *filename*

Check to see if the file is in the current directory or in the path you specified in the command.  
The command may also be expecting a different default file name or extension.

ERROR 301 No data in lexicon file *filename*

ERROR 302 Missing alternation name

The ALTERNATION keyword must be followed by an alternation name.

WARNING 303 Empty alternation definition: *alternation-name*

An ALTERNATION keyword was found with no following alternation name or list of lexicon names.

WARNING 304 Adding to existing alternation: *alternation-name*

ERROR 305 No lexicon sections in lexicon file *filename*

A lexicon file must contain sublexicons.

ERROR 306 Missing lexicon name

The keyword LEXICON must be followed by a sublexicon name.

WARNING 307 Lexicon section *sublexicon-name* is not listed as a member of any alternations

This will not necessarily result in a processing error if this is what you intended to do.

ERROR 308 Expected continuation class or BOUNDARY symbol for *entry*

A lexical entry is missing its continuation class element.

ERROR 309 Invalid continuation class *name* for *entry*

A name appearing in the continuation class field of a lexical entry must be the name of an ALTERNATION that has already been declared.

ERROR 310 Expected gloss element for *entry*

Each lexical entry must have a gloss element.

ERROR 311 Invalid gloss element *gloss* for *entry*

The gloss element must be bracketed by matching delimiters (identical characters).

ERROR 312 Form contains character not in alphabet: *character*

Each character used in lexical items must be listed in the ALPHABET declaration of the rules file.

ERROR 313 INITIAL lexicon not found

A lexicon file must as a minimum have a sublexicon named INITIAL.

ERROR 314 Cannot nest lexicon INCLUDE files

An INCLUDE file cannot call another INCLUDE file.

ERROR 315 Missing INCLUDE file name

An INCLUDE keyword must be followed by a file name.

ERROR 316 Lexicon INCLUDE file could not be opened: *filename*

ERROR 317 Invalid lexicon file keyword: *word*

The only valid keywords in a lexicon file are ALTERNATION, FEATURES, FIELDPCODE, INCLUDE, and END.

ERROR 318 Second argument in FIELDPCODE line is not valid

ERROR 319 First argument name already used

ERROR 320 Cannot use second argument category twice

ERROR 323 Ignore FIELDPCODE *fieldcode* : no double gloss definition

ERROR 324 Expected lexical item

ERROR 325 No features

ERROR 326 Feature name already defined

ERROR 327 Feature name *name* not found

ERROR 328 Lexicon *name* already used in alternation *name*

## 4.10.4 Messages related to reading the grammar file

ERROR 500 Grammar file *filename* not found

ERROR 501 Grammar file is empty

ERROR 502 Let *abbreviation* ... has no be

ERROR 503 No rules found in grammar

ERROR 504 Unexpected *token* before *token*

ERROR 505 *Token* not defined by Let

ERROR 506 Let *abbreviation* be ... has incompatible paths

ERROR 507 Grammar does not begin with Let or rule. It begins with: *token*

ERROR 509 Features of *name* produce FAIL

ERROR 510 Too many nonterminals in rule *rule*

ERROR 511 Incompatible features *name*

ERROR 512 Feature *name* not defined by template in grammar

ERROR 513 Incompatible features on *node*

ERROR 516 + used in grammar rule (repetition not implemented yet).

ERROR 517 expand\_psr called with NULL delimiter.

ERROR 518 Unmatched *token* in the following formula:

ERROR 519 Attempt to pop\_path when current\_last is NULL.

ERROR 520 Label *label* in path not found in rule

ERROR 521 Unexpected *token* before *token*

ERROR 522 Incompatible alternatives on path *path*

ERROR 524 Empty path

ERROR 525 Path not followed by equal

ERROR 526 Invalid disjunction

ERROR 527 *structure* fails to unify

ERROR 528 Non-number *token* after \$

ERROR 529 Zero reference number *number*

ERROR 530 Reference number greater than *number*

ERROR 531 Label *name* has empty path

ERROR 532 Reference \$*label* not defined above

ERROR 533 Reference not allowed before atom

ERROR 534 Unexpected *token* ; Skipping to...

ERROR 535 *token* not defined in Let; Skipping to...

ERROR 536 End of file reached before expected ']' found

ERROR 537 End of file reached before expected '>' found

ERROR 538 Failure to unify < *path* >

ERROR 539 Let *abbreviation* redefined

ERROR 540 End of file reached before expected '}' found

ERROR 523 need\_nonterm() not found

## 4.10.5 Messages related to recognizing or generating a form

WARNING 400 Surface form not found in comparison pairs file

A lexical:surface pair in a pairs comparison file is missing the surface form.

ERROR 511 Incompatible features *features*

ERROR 512 Feature *name* not defined by template in grammar

ERROR 513 Incompatible features on *node*

ERROR 541 Cannot parse forms without a grammar

ERROR 800 Form [ *form* ] contains character not in alphabet: *character*

An input form contains a character that was not listed in the ALPHABET declaration in the rules file.

ERROR 801 RULE *number* is invalid--input *char.char* is not specified by any column

Could happen if a table does not have an ANY:ANY column.

ERROR 802 Invalid lexicon for recognizer

Probably will never occur!

ERROR 803 Lexicon section *sublexicon-name* is empty

There are no lexical entries in the named sublexicon.

ERROR 804 Cannot recognize forms without a lexicon

The lexicon is not loaded.

ERROR 805 Cannot generate forms without rules

ERROR 806 Cannot recognize forms without rules

## 4.10.6 Messages related to memory

ERROR 900 Out of memory

The rules and lexicon are too large to fit in memory.

Runtime error - stack overflow



Occurs when the generator or recognizer gets into an infinite loop due to an incorrectly written rule or lexicon continuation.