PC-PATR Reference Manual

a unification based syntactic parser
version 0.99b5
October 1997

by Stephen McConnel

Published by:

Academic Computing Department
Summer Institute of Linguistics
7500 W. Camp Wisdom Road
Dallas, TX 75236
U.S.A.

Printed on 9 October 1998

The author may be reached at the address above or via email as `steve@acadcomp.sil.org`.

1 Introduction to the PC-PATR program

This document describes PC-PATR, an implementation of the PATR-II computational linguistic formalism for personal computers. It is available for MS-DOS, Microsoft Windows, Macintosh, and Unix.[1]

PC-PATR uses a left corner chart parser with these characteristics:

- bottom-up parse with top-down filtering based on the categories
- left-to-right order—after each word is added to the chart, all possible edges that can be derived up that point are computed as a side-effect

PC-PATR is still under development. The author would appreciate feedback directed to the following address:

```
Stephen McConnel                 (972)708-7361 (office)
Academic Computing Department    (972)708-7363 (fax)
Summer Institute of Linguistics
7500 W. Camp Wisdom Road
Dallas, TX 75236                 steve@acadcomp.sil.org
U.S.A.                        or steve.mcconnel@sil.org
```

[1] The Microsoft Windows implementation uses the Microsoft C QuickWin function, and the Macintosh implementation uses the Metrowerks C SIOUX function.

2 The PATR-II Formalism

The PATR-II formalism can be viewed as a computer language for encoding linguistic information. It does not presuppose any particular theory of syntax. It was originally developed by Stuart M. Shieber at Stanford University in the early 1980's (Shieber 1984, Shieber 1986). A PATR-II grammar consists of a set of rules and a lexicon. Each rule consists of a context-free *phrase structure rule* and a set of *feature constraints*, that is, *unifications* on the *feature structures* associated with the constituents of the phrase structure rules. The lexicon provides the items that can replace the terminal symbols of the phrase structure rules, that is, the words of the language together with their relevant features.

2.1 Phrase structure rules

Context-free phrase structure rules should be familiar to anyone who has studied either linguistic theory or computer science. They look like this:

```
LHS -> RHS_1 RHS_2 ...
```

'LHS' (the symbol to the left of the arrow) is a nonterminal symbol for the type of phrase that is being described. To the right of the arrow is an ordered list of the constituents of the phrase. These constituents are either nonterminal symbols, appearing on the left hand side of some rule in the grammar, or terminal symbols, representing basic classes of elements from the lexicon. These basic classes usually correspond to what are commonly called *parts of speech*. In PATR-II, the terminal and nonterminal symbols are both referred to as *categories*.

Figure 1. Context-free phrase structure grammar

```
Rule  S       -> NP VP (SubCl)
Rule  NP      -> {(Det) (AdjP) N (PrepP)} / PR
Rule  Det     -> DT / PR
Rule  VP      -> VerbalP (NP / AdjP) (AdvP)
Rule  VerbalP -> V
Rule  VerbalP -> AuxP V
Rule  AuxP    -> AUX (AuxP_1)
Rule  PrepP   -> PP NP
Rule  AdjP    -> (AV) AJ (AdjP_1)
Rule  AdvP    -> {AV / PrepP} (AdvP_1)
Rule  SubCl   -> CJ S
```

Consider the PC-PATR style context-free phrase structure grammar in figure 1. It has ten nonterminal symbols (S, NP, Det, VP, VerbalP, AuxP, PrepP, AdjP, AdvP, and SubCl), and nine terminal symbols (N, PR, DT, V, AUX, PP, AV, AJ, and CJ). This grammar describes a small subset of English sentences. Several aspects of this grammar are worth mentioning.

1. Optional constituents (or sets of constituents) on the right hand side are enclosed in parentheses.
2. Alternative constituents (or sets of constituents) on the right hand side are separated by slashes.
3. Braces are used to group alternative sets of elements together, so that alternations are not ambiguous.

4. Symbols should not be repeated verbatim within a rule. Repeated symbols should be distinguished from each other by adding a different index number to a symbol each time it is repeated. Index numbers are introduced by the underscore (_) character.

Figure 2. Parse of sample English sentence

```
            S
            /\
           /  \
          /    \
         /      \
        /        \
       /          \
      /            \
    NP              VP
    /\             /|\
   /  \           / | \
  /    \         /  |  \
Det     N  VerbalP NP   AdvP
 |      |     |     |     |
DT     man    V    PR   PrepP
 |            |     |    /\
the         sees   us   /  \
                       /    \
                      PP     NP
                       |     /\
                     with   /  \
                           /    \
                         Det     N
                          |      |
                         DT  telescope
                          |
                          a
```

Figure 3. Parse of sample sentence (PC-PATR output)

```
              S
    __________|__________
   NP                   VP
 ___|____      _________|__________
Det     N   VerbalP  NP         AdvP
 |     man     |      |           |
DT             V     PR         PrepP
the          sees    us      _____|______
                            PP          NP
                           with     ____|_____
                                   Det       N
                                    |    telescope
                                   DT
                                    a
```

A significant amount of grammar development can be done just with context-free phrase structure rules such as these. For example, parsing the sentence “the man sees us with a telescope” with this simple grammar produces a parse tree like that shown in figure 2. (In order to minimize the height of parse trees without needing to use a graphical interface, PC-PATR actually draws parse trees like the one shown in figure 3.) Parsing the similar sentence “we see the man with a telescope” produces two different parses as shown in figure 4, correctly showing the ambiguity between whether we used a telescope to see the man, or the man had a telescope when we saw him.

Figure 4. Parses of an ambiguous English sentence

```
           S_1
  __________|__________
NP_2+               VP_4
  |      _____________|_____________
PR_3+ VerbalP_5+ NP_7           AdvP_11
 we      |      ___|____           |
       V_6+  Det_8+  N_10+     PrepP_12+
        see     |     man     _____|______
              DT_9+        PP_13+     NP_14+
               the          with     ____|_____
                                  Det_15+   N_17+
                                     |    telescope
                                  DT_16+
                                     a

       S_18
  _______|________
NP_2+          VP_19
  |      ________|________
PR_3+ VerbalP_5+       NP_20
 we      |      _________|__________
       V_6+  Det_8+  N_10+     PrepP_12+
        see     |     man     _____|______
              DT_9+        PP_13+     NP_14+
               the          with     ____|_____
                                  Det_15+   N_17+
                                     |    telescope
                                  DT_16+
                                     a
```

A fundamental problem with context-free phrase structure grammars is that they tend to grossly overgenerate. For example, the sample grammar would incorrectly recognize the sentence “*he see the man with a telescope”, assigning it tree structures similar to those shown in figure 4. With only the simple categories used by context-free phrase structure rules, a very large number of rules are required to accurately handle even a small subset of a language’s grammar. This is the primary motivation behind feature structures, the basic enhancement of PATR-II over context-free phrase structure grammars.[1]

[1] Gazdar and Mellish (1989, pages 142–147) discuss why context-free phrase structure grammars are inadequate to model some human languages. The PATR-II formalism

2.2 Feature structures

The basic data structure of the PATR-II formalism is called a *feature structure*. A feature structure contains one or more *features*. A feature consists of an attribute name and a value. Feature structures are commonly written as attribute-value matrices like this (example 1):

```
(1)     [ lex: telescope
          cat: N ]
```

where *lex* and *cat* are attribute names, and *telescope* and *N* are the values for those attributes. Note that the feature structure is enclosed in brackets. Each feature occurs on a separate line, with the name coming first, followed by a colon and then its value. Feature names and (simple) values are single words consisting of alphanumeric characters.

Feature structures can have either simple values, such as the example above, or complex values, such as this (example 2):

```
(2)     [ lex:      telescope
          cat:      N
          gloss:    ‘telescope
          head:     [ agr:    [ 3sg: + ]
                      number: SG
                      pos:    N
                      proper: -
                      verbal: - ]
          root_pos: N ]
```

where the value of the *head* feature is another feature structure, that also contains an embedded feature structure. Feature structures can be arbitrarily nested in this manner.

Portions of a feature structure can be referred to using the *path* notation. A path is a sequence of one or more feature names enclosed in angled brackets (`<>`). For instance, examples 3–5 would all be valid feature paths based on the feature structure of example 2:

```
(3)     <head>
(4)     <head number>
(5)     <head agr 3sg>
```

Paths are used in feature templates and feature constraints, described below.

Different features within a feature structure can share values. This is not the same thing as two features having identical values. In Example 6 below, the `<head agr>` and `<subj head agr>` features have identical values, but in Example 7, they share the same value:

(unification of feature structures added to the context-free phrase structure rules) is shown to be adequate for those cases.

```
(6)     [ cat:  S
          pred: [ cat:  VP
                  head: [ agr:    [ 3sg: + ]
                          finite: +
                          pos:    V
                          tense:  PAST
                          vform:  ED ] ]
          subj: [ cat:  NP
                  head: [ agr:    [ 3sg: + ]
                          case:   NOM
                          number: SG
                          pos:    N
                          proper: -
                          verbal: - ] ] ]
```

```
(7)     [ cat:  S
          pred: [ cat:  VP
                  head: [ agr:    $1[ 3sg: + ]
                          finite: +
                          pos:    V
                          tense:  PAST
                          vform:  ED ] ]
          subj: [ cat:  NP
                  head: [ agr:    $1
                          case:   NOM
                          number: SG
                          pos:    N
                          proper: -
                          verbal: - ] ] ]
```

Shared values are indicated by the coindexing markers `$1`, `$2`, and so on.

Note that upper and lower case letters used in feature names and values are distinctive. For example, *NUMBER* is not the same as *Number* or *number*. (This is also true of the symbols used in the context-free phrase structure rules.)

2.3 Unification

Unification is the basic operation applied to feature structures in PC-PATR. It consists of the merging of the information from two feature structures. Two feature structures can unify if their common features have the same values, but do not unify if any feature values conflict.

Consider the following feature structures:

```
(8)     [ agreement: [ number: singular
                       person: first ] ]
```

```
(9)     [ agreement: [ number: singular ]
          case:      nominative ]
```

```
(10)    [ agreement: [ number: singular
                       person: third ] ]
```

```
(11)    [ agreement: [ number: singular
                       person: first ]
          case:      nominative ]
(12)    [ agreement: [ number: singular
                       person: third ]
          case:      nominative ]
```

Feature 9 can unify with either feature 8 (producing feature 11) or feature 10 (producing feature 12). However, feature 8 cannot unify with feature 10 due to the conflict in the values of their `<agreement person>` features.

2.4 Feature constraints

The feature constraints associated with phrase structure rules in PC-PATR consist of a set of unification expressions. Each expression has three parts, in this order:

1. a feature path, the first element of which is one of the symbols from the phrase structure rule
2. an equal sign (`=`)
3. either a simple value, or another feature path that also starts with a symbol from the phrase structure rule

As an example, consider the following PC-PATR rules:

```
(13) Rule S -> NP VP (SubCl)
     <NP head agr>  = <VP head agr>
     <NP head case> = NOM
     <S subj>       = <NP>
     <S head>       = <VP head>
(14) Rule NP -> {(Det) (AJ) N (PrepP)} / PR
     <Det head number> = <N head number>
     <NP head>         = <N head>
     <NP head>         = <PR head>
```

Rule 13 has two feature constraints that limit the co-occurrence of NP and VP, and two feature constraints that build the feature structures for S. This highlights the dual purpose of feature constraints in PC-PATR: limiting the co-occurrence of phrase structure elements and constructing the feature structure for the element defined by a rule. The first constraint states that the NP and VP $\langle head$ $agr\rangle$ features must unify successfully, and also modifies both of those features if they do unify. The second constraint states that NP's $\langle head$ $case\rangle$ feature must either be equal to NOM or else be undefined. In the latter case, it is set equal to NOM. The last two constraints create a new feature structure for S from the feature structures for NP and VP.

Rule 14 illustrates another important point about feature constraints. Constraints are applied only if they involve the phrase structure constituents actually found for the rule.

Figure 5. PC-PATR grammar of English subset

```
Rule  S -> NP VP (SubCl)
        <NP head agr>  = <VP head agr>
        <NP head case> = NOM
        <S subj> = <NP>
        <S pred> = <VP>
Rule  NP -> {(Det) (AdjP) N (PrepP)} / PR
        <Det head number> = <N head number>
        <NP head> = <N head>
        <NP head> = <PR head>
Rule  Det -> DT / PR
        <PR head case> = GEN
        <Det head> = <DT head>
        <Det head> = <PR head>
Rule  VP -> VerbalP (NP / AdjP) (AdvP)
        <NP head case>   = ACC
        <NP head verbal> = -
        <VP head> = <VerbalP head>
Rule  VerbalP -> V
        <V head finite> = +
        <VerbalP head>  = <V head>
Rule  VerbalP -> AuxP V
        <V head finite> = -
        <VerbalP head>  = <AuxP head>
Rule  AuxP -> AUX (AuxP_1)
        <AuxP head> = <AUX head>
Rule  PrepP -> PP NP
        <NP head case> = ACC
        <PrepP head> = <PP head>
Rule  AdjP -> (AV) AJ (AdjP_1)
Rule  AdvP -> {AV / PrepP} (AdvP_1)
Rule  SubCl -> CJ S
```

Figure 6. PC-PATR output with feature structure

```
1:
                S
      __________|__________
     NP                  VP
   ___|____       _________|__________
  Det     N   VerbalP  NP         AdvP
   |     man     |      |           |
  DT             V     PR         PrepP
  the           saw    us      _____|______
                              PP         NP
                             with     ____|_____
                                     Det       N
                                      |    telescope
                                     DT
                                      a

S:
[ cat:   S
  pred:    [ cat:   VP
             head:    [ agr:   $1[ 3sg:   + ]
                        finite:+
                        pos:   V
                        tense: PAST
                        vform: ED ] ]
  subj:    [ cat:   NP
             head:    [ agr:   $1
                        case:  NOM
                        number:SG
                        pos:   N
                        proper:-
                        verbal:- ] ] ]

1 parse found
```

Figure 5 shows the grammar of figure 1 augmented with a number of feature constraints. With this grammar (and a suitable lexicon), the parse output shown in figure 2 would include the sentence feature structure, as shown in figure 6. Note that the `<subj head agr>` and `<pred head agr>` features share a common value as a result of the feature constraint unifications associated with the rule `S -> NP VP (SubCl)`.

PC-PATR allows disjunctive feature constraints with its phrase structure rules. Consider rules 15 and 16 below. These two rules have the same phrase structure rule part. They can therefore be collapsed into the single rule 17, which has a disjunction in its unification constraints.

```
(15) Rule CP -> NP C'        ; for wh questions with NP fronted
    <NP type wh> = +
    <C' moved A-bar> = <NP>
    <CP type wh> = <NP type wh>
    <CP type> = <C' type>
    <CP moved A-bar> = none
    <CP type root> = +          ; root clauses
    <CP type q> = +
    <CP type fin> = +
    <CP moved A> = none
    <CP moved head> = none
```

```
(16) Rule CP -> NP C'        ; for wh questions with NP fronted
    <NP type wh> = +
    <C' moved A-bar> = <NP>
    <CP type wh> = <NP type wh>
    <CP type> = <C' type>
    <CP moved A-bar> = none
    <CP type root> = -          ; non-root clauses
```

```
(17) Rule CP -> NP C'        ; for wh questions with NP fronted
    <NP type wh> = +
    <C' moved A-bar> = <NP>
    <CP type wh> = <NP type wh>
    <CP type> = <C' type>
    <CP moved A-bar> = none
    {
    <CP type root> = + ; root clauses
    <CP type q> = +
    <CP type fin> = +
    <CP moved A> = none
    <CP moved head> = none
        /
    <CP type root> = - ; non-root clauses
    }
```

Not only does PC-PATR allow disjunctive unification constraints, but it also allows disjunctive phrase structure rules. Consider rule 18: it is very similar to rule 17. These two rules can be further combined to form rule 19, which has disjunctions in both its phrase structure rule and its unification constraints.

```
(18) Rule CP -> PP C'       ; for wh questions with PP fronted
    <PP type wh> = +
    <C' moved A-bar> = <PP>
    <CP type wh> = <PP type wh>
    <CP type> = <C' type>
    <CP moved A-bar> = none
    {
    <CP type root> = + ; root clauses
    <CP type q> = +
    <CP type fin> = +
    <CP moved A> = none
    <CP moved head> = none
        /
    <CP type root> = - ; non-root clauses
    }

(19) ; for wh questions with NP or PP fronted
Rule CP -> { NP / PP } C'
    <NP type wh> = +
    <C' moved A-bar> = <NP>
    <CP type wh> = <NP type wh>
    <PP type wh> = +
    <C' moved A-bar> = <PP>
    <CP type wh> = <PP type wh>
    <CP type> = <C' type>
    <CP moved A-bar> = none
    {
    <CP type root> = + ; root clauses
    <CP type q> = +
    <CP type fin> = +
    <CP moved A> = none
    <CP moved head> = none
        /
    <CP type root> = - ; non-root clauses
    }
```

Since the open brace ({) introduces disjunctions both in the phrase structure rule and in the unification constraints, care must be taken to avoid confusing PC-PATR when it is loading the grammar file. The end of the phrase structure rule, and the beginning of the unification constraints, is signaled either by the first constraint beginning with an open angle bracket (<) or by a colon (:). If the first constraint is part of a disjunction, then the phrase structure rule must end with a colon. Otherwise, PC-PATR will treat the unification constraint as part of the phrase structure rule, and will shortly complain about syntax errors in the grammar file.

Perhaps it should be noted that disjunctions in phrase structure rules or unifications are expanded when the grammar file is read. They serve only as a convenience for the person writing the rules.

2.5 The lexicon

The lexicon provides the basic elements (atoms) of the grammar, which are usually words. Information like that shown in feature 2 is provided for each lexicon entry. Unlike the original implementation of PATR-II, PC-PATR stores the lexicon in a separate file from the grammar rules. See Chapter 6 [Lexicon file], page 39 below for details.

3 Running PC-PATR

PC-PATR is an interactive program. It has a few command line options, but it is controlled primarily by commands typed at the keyboard (or loaded from a file previously prepared).

3.1 PC-PATR Command Line Options

The PC-PATR program uses an old-fashioned command line interface following the convention of options starting with a dash character ('`-`'). The available options are listed below in alphabetical order. Those options which require an argument have the argument type following the option letter.

`-a filename`
: loads the lexicon from an AMPLE analysis output file.

`-g filename`
: loads the grammar from a PC-PATR grammar file.

`-l filename`
: loads the lexicon from a PC-PATR lexicon file.

`-t filename`
: opens a file containing one or more PC-PATR commands. See Section 3.2 [Interactive commands], page 15.

The following options exist only in beta-test versions of the program, since they are used only for debugging.

`-/`
: increments the debugging level. The default is zero (no debugging output).

`-z filename`
: opens a file for recording a memory allocation log.

`-Z address,count`
: traps the program at the point where `address` is allocated or freed for the `count`'th time.

3.2 Interactive Commands

Each of the commands available in PC-PATR is described below. Each command consists of one or more keywords followed by zero or more arguments. Keywords may be abbreviated to the minimum length necessary to prevent ambiguity.

3.2.1 cd

`cd` *directory* changes the current directory to the one specified. Spaces in the directory pathname are not permitted.

For MS-DOS or Windows, you can give a full path starting with the disk letter and a colon (for example, `a:`); a path starting with \ which indicates a directory at the top level of the current disk; a path starting with `..` which indicates the directory above the current one; and so on. Directories are separated by the `\` character. (The forward slash `/` works just as well as the backslash `\` for MS-DOS or Windows.)

For the Macintosh, you can give a full path starting with the name of a hard disk, a path starting with `:` which means the current folder, or one starting `::` which means the folder containing the current one (and so on).

For Unix, you can give a full path starting with a `/` (for example, `/usr/pcpatr`); a path starting with `..` which indicates the directory above the current one; and so on. Directories are separated by the `/` character.

3.2.2 clear

`clear` erases all existing grammar and lexicon information, allowing the user to prepare to load information for a new language. Strictly speaking, it is not needed since the `load grammar` command erases the previously existing grammar, and the `load lexicon` and `load analysis` commands erase any previously existing lexicon.

3.2.3 close

`close` closes the current log file opened by a previous `log` command.

3.2.4 directory

`directory` lists the contents of the current directory. This command is available only for the MS-DOS and Unix implementations. It does not exist for Microsoft Windows or the Macintosh.

3.2.5 edit

`edit` *filename* attempts to edit the specified file using the program indicated by the environment variable `EDITOR`. If this environment variable is not defined, then `edlin` is used to edit the file on MS-DOS, and `vi` is used to edit the file on Unix. (These defaults should convince you to set this variable!) This command is not available for Microsoft Windows or the Macintosh.

3.2.6 exit

`exit` stops PC-PATR, returning control to the operating system. This is the same as `quit`.

3.2.7 file

The `file` commands process data from a file, optionally writing the parse results to another file. Each of these commands is described below.

3.2.7.1 file disambiguate

`file disambiguate` *input.ana [out.ana]* reads sentences from the specified AMPLE analysis file and writes the corresponding parse trees and feature structures either to the screen or to the optionally specified output file. If the output file is written, ambiguous word parses are eliminated as much as possible as a result of the sentence parsing. When finished, a statistical report of successful (sentence) parses is displayed on the screen.

3.2.7.2 file parse

`file parse` *input-file [output-file]* reads sentences from the specified input file, one per line, and writes the corresponding parse trees and feature structures to the screen or to the optionally specified output file. The comment character is in effect while reading this file. PC-PATR currently makes no attempt to handle either capitalization or punctuation. PROBABLY SOME CAPABILITY FOR HANDLING PUNCTUATION WILL BE ADDED AT SOME POINT.

This command behaves the same as `parse` except that input comes from a file rather than the keyboard, and output may go to a file rather than the screen. When finished, a statistical report of successful parses is displayed on the screen.

3.2.8 help

`help` *command* displays a description of the specified command. If `help` is typed by itself, PC-PATR displays a list of commands with short descriptions of each command.

3.2.9 load

The `load` commands all load information stored in specially formatted files. The `load ample` and `load kimmo` commands activate morphological parsers, and serve as alternatives to `load lexicon` (or `load analysis`) for obtaining the category and other feature information for words. Each of the `load` commands is described below.

3.2.9.1 load ample control

`load ample control` *xxad01.ctl xxancd.tab [xxordc.tab]* erases any existing AMPLE information (including dictionaries) and reads control information from the specified files. This also erases any stored PC-Kimmo information.

At least two and possibly three files are loaded by this command. The first file is the AMPLE *analysis data* file. It has a default filetype extension of `.ctl` but no default filename. The second file is the AMPLE dictionary code table file. It has a default filetype extension of `.tab` but no default filename. The third file is an optional dictionary orthography change table. It has a default filetype extension of `.tab` and no default filename.

`l am c` is a synonym for `load ample control`.

3.2.9.2 load ample dictionary

`load ample dictionary` *[prefix.dic] [infix.dic] [suffix.dic] root1.dic [. . .]* or `load ample dictionary` *file01.dic [file02.dic . . .]* erases any existing AMPLE dictionary information and reads the specified files. This also erases any stored PC-Kimmo information.

The first form of the command is for using a dictionary whose files are divided according to morpheme type (`set ample-dictionary split`). The different types of dictionary files must be loaded in the order shown, with any unneeded affix dictionaries omitted.

The second form of the command is for using a dictionary whose entries contain the type of morpheme (`set ample-dictionary unified`).[1]

`l am d` is a synonym for `load ample dictionary`.

3.2.9.3 load ample text-control

`load ample text-control` *xxintx.ctl* erases any existing AMPLE text input control information and reads the specified file. This also erases any stored PC-Kimmo information.

The text input control file has a default filetype extension of `.ctl` but no default filename.

`l am t` is a synonym for `load ample text-control`.

3.2.9.4 load analysis

`load analysis` *file1.ana [file2.ana . . .]* erases any existing lexicon and reads a new lexicon from the specified AMPLE analysis file(s). Note that more than one file may be loaded with the single `load analysis` command: duplicate entries are not stored in the lexicon.

The default filetype extension for `load analysis` is `.ana`, and the default filename is `ample.ana`.

`l a` is a synonym for `load analysis`.

3.2.9.5 load grammar

`load grammar` *file.grm* erases any existing grammar and reads a new grammar from the specified file.

The default filetype extension for `load grammar` is `.grm`, and the default filename is `grammar.grm`.

`l g` is a synonym for `load grammar`.

3.2.9.6 load kimmo grammar

`load kimmo grammar` *file.grm* erases any existing PC-Kimmo (word) grammar and reads a new word grammar from the specified file.

The default filetype extension for `load kimmo grammar` is `.grm`, and the default filename is `grammar.grm`.

`l k g` is a synonym for `load kimmo grammar`.

[1] This is a new feature of AMPLE version 3.

3.2.9.7 load kimmo lexicon

`load kimmo lexicon` *file.lex* erases any existing PC-Kimmo lexicon information and reads a new morpheme lexicon from the specified file. A PC-Kimmo rules file must be loaded before a PC-Kimmo lexicon file can be loaded.

The default filetype extension for `load kimmo lexicon` is `.lex`, and the default filename is `lexicon.lex`.

`l k l` is a synonym for `load kimmo lexicon`.

3.2.9.8 load kimmo rules

`load kimmo rules` *file.rul* erases any existing PC-Kimmo rules and reads a new set of rules from the specified file. This also erases any stored AMPLE information.

The default filetype extension for `load kimmo rules` is `.rul`, and the default filename is `rules.rul`.

`l k r` is a synonym for `load kimmo rules`.

3.2.9.9 load lexicon

`load lexicon` *file1.lex [file2.lex . . .]* erases any existing lexicon and reads a new lexicon from the specified file(s). Note that more than one file may be loaded with a single `load lexicon` command.

The default filetype extension for `load lexicon` is `.lex`, and the default filename is `lexicon.lex`.

`l l` is a synonym for `load lexicon`.

3.2.10 log

`log` *[file.log]* opens a log file. Each item processed by a `parse` command is stored to the log file as well as being displayed on the screen.

If a filename is given on the same line as the `log` command, then that file is used for the log file. Any previously existing file with the same name will be overwritten. If no filename is provided, then the file `pcpatr.log` in the current directory is used for the log file.

Use `close` to stop recording in a log file. If a `log` command is given when a log file is already open, then the earlier log file is closed before the new log file is opened.

3.2.11 parse

`parse` *[sentence or phrase]* attempts to parse the input sentence according to the loaded grammar. If a sentence is typed on the same line as the command, then that sentence is parsed. If the `parse` command is given by itself, then the user is prompted repeatedly for sentences to parse. This cycle of typing and parsing is terminated by typing an empty "sentence" (that is, nothing but the `Enter` or `Return` key).

Both the grammar and the lexicon must be loaded before using this command.

3.2.12 quit

`quit` stops PC-PATR, returning control to the operating system. This is the same as `exit`.

3.2.13 save

The `save` commands write information stored in memory to a file suitable for reloading into PC-PATR later. Each of these commands is described below.

3.2.13.1 save lexicon

`save lexicon` *[file.lex]* writes the current lexicon contents to the designated file. The output lexicon file must be specified. This can be useful if you are using a morphological parser to populate the lexicon.

3.2.13.2 save status

`save status` *[file.tak]* writes the current settings to the designated file in the form of PC-PATR commands. If the file is not specified, the settings are written to `pcpatr.tak` in the current directory.

3.2.14 set

The `set` commands control program behavior by setting internal program variables. Each of these commands (and variables) is described below.

3.2.14.1 set ambiguities

`set ambiguities` *number* limits the number of analyses printed to the given number. The default value is 10. Note that this does not limit the number of analyses produced, just the number printed.

3.2.14.2 set ample-dictionary

`set ample-dictionary` *value* determines whether or not the AMPLE dictionary files are divided according to morpheme type. `set ample-dictionary split` declares that the AMPLE dictionary is divided into a prefix dictionary file, an infix dictionary file, a suffix dictionary file, and one or more root dictionary files. The existence of the three affix dictionary depends on settings in the AMPLE analysis data file. If they exist, the `load ample dictionary` command requires that they be given in this relative order: prefix, infix, suffix, root(s).

`set ample-dictionary unified` declares that any of the AMPLE dictionary files may contain any type of morpheme. This implies that each dictionary entry may contain a field specifying the type of morpheme (the default is *root*), and that the dictionary code table contains a `\unified` field. One of the changes listed under `\unified` must convert a backslash code to `T`.

The default is for the AMPLE dictionary to be *split*.[2]

3.2.14.3 set check-cycles

`set check-cycles` *value* enables or disables a check to prevent cycles in the parse chart. `set check-cycles on` turns on this check, and `set check-cycles off` turns it off. This check slows down the parsing of a sentence, but it makes the parser less vulnerable to hanging on perverse grammars. The default setting is `on`.

3.2.14.4 set comment

`set comment` *character* sets the comment character to the indicated value. If *character* is missing (or equal to the current comment character), then comment handling is disabled. The default comment character is `;` (semicolon).

3.2.14.5 set failures

`set failures` *value* enables or disables *grammar failure mode*. `set failures on` turns on grammar failure mode, and `set failures off` turns it off. When grammar failure mode is on, the partial results of forms that fail the grammar module are displayed. A form may fail the grammar either by failing the feature constraints or by failing the constituent structure rules. In the latter case, a partial tree (bush) will be returned. The default setting is `off`.

Be careful with this option. Setting failures to `on` can cause the PC-PATR to go into an infinite loop for certain recursive grammars and certain input sentences. WE MAY TRY TO DO SOMETHING TO DETECT THIS TYPE OF BEHAVIOR, AT LEAST PARTIALLY.

3.2.14.6 set features

`set features` *value* determines how features will be displayed.

`set features all` enables the display of the features for all nodes of the parse tree.

`set features top` enables the display of the feature structure for only the top node of the parse tree. This is the default setting.

`set features flat` causes features to be displayed in a flat, linear string that uses less space on the screen.

`set features full` causes features to be displayed in an indented form that makes the embedded structure of the feature set clear. This is the default setting.

`set features on` turns on features display mode, allowing features to be shown. This is the default setting.

`set features off` turns off features display mode, preventing features from being shown.

[2] The unified dictionary is a new feature of AMPLE version 3.

3.2.14.7 set gloss

`set gloss` *value* enables the display of glosses in the parse tree output if *value* is `on`, and disables the display of glosses if *value* is `off`. If any glosses exist in the lexicon file, then `gloss` is automatically turned `on` when the lexicon is loaded. If no glosses exist in the lexicon, then this flag is ignored.

3.2.14.8 set marker category

`set marker category` *marker* establishes the marker for the field containing the category (part of speech) feature. The default is `\c`.

3.2.14.9 set marker features

`set marker features` *marker* establishes the marker for the field containing miscellaneous features. (This field is not needed for many words.) The default is `\f`.

3.2.14.10 set marker gloss

`set marker gloss` *marker* establishes the marker for the field containing the word gloss. The default is `\g`.

3.2.14.11 set marker record

`set marker record` *marker* establishes the field marker that begins a new record in the lexicon file. This may or may not be the same as the `word` marker. The default is `\w`.

3.2.14.12 set marker word

`set marker word` *marker* establishes the marker for the word field. The default is `\w`.

3.2.14.13 set timing

`set timing` *value* enables timing mode if *value* is `on`, and disables timing mode if *value* is `off`. If timing mode is `on`, then the elapsed time required to process a command is displayed when the command finishes. If timing mode is `off`, then the elapsed time is not shown. The default is `off`. (This option is useful only to satisfy idle curiosity.)

3.2.14.14 set top-down-filter

`set top-down-filter` *value* enables or disables top-down filtering based on the categories. `set top-down-filter on` turns on this filtering, and `set top-down-filter off` turns it off. The top-down filter speeds up the parsing of a sentence, but might cause the parser to miss some valid parses. The default setting is `on`.

This should not be required in the final version of PC-PATR.

3.2.14.15 set tree

`set tree` *value* specifies how parse trees should be displayed.

`set tree full` turns on the parse tree display, displaying the result of the parse as a full tree. This is the default setting. A short sentence would look something like this:

```
   Sentence
       |
  Declarative
  _____|_____
 NP         VP
  |       ___|____
  N      V    COMP
cows    eat     |
               NP
                |
                N
              grass
```

`set tree flat` turns on the parse tree display, displaying the result of the parse as a flat tree structure in the form of a bracketed string. The same short sentence would look something like this:

```
(Sentence (Declarative (NP
    (N  cows)) (VP (V  eat) (COMP
    (NP (N  grass))))))
```

`set tree indented` turns on the parse tree display, displaying the result of the parse in an indented format sometimes called a *northwest tree*. The same short sentence would look like this:

```
Sentence
    Declarative
        NP
            N  cows
        VP
            V  eat
            COMP
                NP
                    N  grass
```

`set tree off` disables the display of parse trees altogether.

3.2.14.16 set trim-empty-features

`set trim-empty-features` *value* disables the display of empty feature values if *value* is `on`, and enables the display of empty feature values if *value* is `off`. The default is not to display empty feature values.

3.2.14.17 set unification

`set unification` *value* enables or disables feature unification. `set unification on` turns on unification mode. This is the default setting.

`set unification off` turns off feature unification in the grammar. Only the context-free phrase structure rules are used to guide the parse; the feature contraints are ignored. This can be dangerous, as it is easy to introduce infinite cycles in recursive phrase structure rules.

3.2.14.18 set verbose

`set verbose` *value* enables or disables the screen display of parse trees in the `file parse` command. `set verbose on` enables the screen display of parse trees, and `set verbose off` disables such display. The default setting is `off`.

3.2.14.19 set warnings

`set warnings` *value* enables warning mode if *value* is `on`, and disables warning mode if *value* is `off`. If warning mode is enabled, then warning messages are displayed on the output. If warning mode is disabled, then no warning messages are displayed. The default setting is `on`.

3.2.14.20 set write-ample-parses

`set write-ample-parses` *value* enables writing `\parse` and `\features` fields at the end of each sentence in the disambiguated analysis file if *value* is `on`, and disables writing these fields if *value* is `off`. The default setting is `off`.

This variable setting affects only the `file disambiguate` command.

3.2.15 show

The `show` commands display internal settings on the screen. Each of these commands is described below.

3.2.15.1 show lexicon

`show lexicon` prints the contents of the lexicon stored in memory on the standard output. THIS IS NOT VERY USEFUL, AND MAY BE REMOVED.

3.2.15.2 show status

`show status` displays the names of the current grammar, sentences, and log files, and the values of the switches established by the `set` command.

`show` (by itself) and `status` are synonyms for `show status`.

3.2.16 status

`status` displays the names of the current grammar, sentences, and log files, and the values of the switches established by the `set` command.

3.2.17 system

`system` *[command]* allows the user to execute an operating system command (such as checking the available space on a disk) from within PC-PATR. This is available only for MS-DOS and Unix, not for Microsoft Windows or the Macintosh.

If no system-level command is given on the line with the `system` command, then PC-PATR is pushed into the background and a new system command processor (shell) is started. Control is usually returned to PC-PATR in this case by typing `exit` as the operating system command.

`!` (exclamation point) is a synonym for `system`.

3.2.18 take

`take` *[file.tak]* redirects command input to the specified file.

The default filetype extension for `take` is `.tak`, and the default filename is `pcpatr.tak`.

`take` files can be nested three deep. That is, the user types `take file1`, `file1` contains the command `take file2`, and `file2` has the command `take file3`. It would be an error for `file3` to contain a `take` command. This should not prove to be a serious limitation.

A `take` file can also be specified by using the `-t` command line option when starting PC-PATR. When started, PC-PATR looks for a `take` file named '`pcpatr.tak`' in the current directory to initialize itself with.

4 The PC-PATR Grammar File

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 by the `set comment` command (see Section 3.2.14.4 [set comment], page 21) is operative in the grammar file. The default comment character is the semicolon (`;`). Comments may be placed anywhere in the grammar file. Everything following a comment character to the end of the line is ignored.
- A grammar file is divided into fields identified by a small set of keywords.
  1. `Rule` starts a context-free phrase structure 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 phrase structure 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. 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 serve two primary purposes in PC-PATR: modifying the feature structures associated with lexicon entries and modifying the feature structures produced by a morphological parser.
  5. `Lexicon` starts a lexicon section. This is only for compatibility with the original PATR-II. The section name is skipped over properly, but nothing is done with it.
  6. `Word` starts an entry in the lexicon. This is only for compatibility with the original PATR-II. The entry is skipped over properly, but nothing is done with it.[1]
  7. `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.

4.1 Rules

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

[1] Would this be a useful enhancement to PC-PATR?

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
8. an optional period (`.`)

The optional rule identifier 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:

- Upper and lower case letters used in symbols are considered different. For example, `NOUN` is not the same as `Noun`, and neither is the same as `noun`.
- The symbol X may be used to stand for any terminal or nonterminal. For example, this rule says that any category in the grammar rules can be replaced by two copies of the same category separated by a CJ.

```
Rule X -> X_1 CJ X_2
        <X cat>  = <X_1 cat>
        <X cat>  = <X_2 cat>
        <X arg1> = <X_1 arg1>
        <X arg1> = <X_2 arg1>
```

  The symbol X can be useful for capturing generalities. Care must be taken, since it can be replaced by anything.
- 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 (`_`). This is illustrated in the rule for X above.
- 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.

The symbols on the right hand side of a phrase structure rule may be marked or grouped in various ways:

- Parentheses around an element of the expansion (right hand) 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 (right hand) part of a rule.

- Curly braces can be used for grouping elements. For example the following says that an S consists of an NP followed by either a TVP or an IV:

```
Rule S -> NP {TVP / IV}
```

- 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 TVP would be treated as part of the alternative containing the NP. It would not be allowed before the IV.

```
Rule S -> NP TVP / IV
```

- 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.

A rule can be followed by zero or more *feature constraints* that refer to symbols used in the rule. A feature constraint has these parts, in the order listed:

1. a feature path that begins with one of the symbols from the phrase structure 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 values of the *agr* features for the NP and VP nodes of the parse tree must unify:

```
Rule S -> NP VP
        <NP agr> = <VP agr>
```

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 NP node of the parse tree must either be originally undefined or equal to NOM:

```
Rule S -> NP VP
        <NP head case> = NOM
```

(After unification succeeds, the *head case* feature for the NP node of the parse tree 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 *tense* feature is passed from the VP node up to the S node:

```
Rule S -> NP VP
        <S tense> = <VP tense>
```

4.2 Feature templates

A PC-PATR 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 phrase structure rules), the template defines the default features for that category. Otherwise the template name serves as an abbreviation for the associated feature structure.

The characters `(){}[]<>=:` cannot be used in template names since they are used for special purposes in the grammar file. The characters `/_` can be freely used in template names. 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.

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`.

4.3 Parameter settings

A PC-PATR 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-PATR recognizes the following parameters:

`Start symbol`

defines the start symbol of the grammar. For example,

```
Parameter Start symbol is S
```

declares that the parse goal of the grammar is the nonterminal category S. The default start symbol is the left hand symbol of the first phrase structure rule in the grammar file.

`Restrictor`

defines a set of features to use for top-down filtering, expressed as a list of feature paths. For example,

```
Parameter Restrictor is <cat> <head form>
```

declares that the *cat* and *head form* features should be used to screen rules before adding them to the parse chart. The default is not to use any features for such filtering. This filtering, named *restriction* in Shieber (1985), is performed in addition to the normal top-down filtering based on categories alone.

RESTRICTION IS NOT YET IMPLEMENTED. SHOULD IT BE INSTEAD OF NORMAL FILTERING RATHER THAN IN ADDITION TO?

`Attribute order`

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

```
Parameter Attribute order is cat lex sense head
                             first rest agreement
```

declares that the *cat* attribute should be the first one shown in any output from PC-PATR, and that the other attributes should be shown in the relative order shown, with the *agreement* 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*.

4.4 Lexical rules

LEXICAL RULES ARE NOT WORKING PROPERLY; THEY NEED TO BE REIMPLEMENTED, AND SOME OTHER MECHANISM USED TO MAP PC-KIMMO FEATURES ONTO PC-PATR FEATURES.

Figure 7. PC-PATR lexical rule example

```
; lexicon entry
\w stormed
\c V
\f Transitive AgentlessPassive
   <head trans pred> = storm

; definitions from the grammar file
Let Transitive be
        <subcat first cat> = NP
        <subcat rest first cat> = NP
        <subcat rest rest> = end
        <head trans arg1> = <subcat first head trans>
        <head trans arg2> = <subcat rest first head trans>.

Define AgentlessPassive as
                <out cat> = <in cat>
                <out subcat> = <in subcat rest>
                <out lex> = <in lex> ; added for PC-PATR
                <out head> = <in head>
                <out head form> => passiveparticiple.
```

Figure 8. Feature structure before lexical rule

```
[ lex:    stormed
  cat:    V
  head:   [ trans: [ arg1:  $1 []
                     arg2:  $2 []
                     pred:  storm ] ]
  subcat: [ first: [ cat:   NP
                     head:  [ trans: $1 ] ]
            rest:  [ first: [ cat:   NP
                               head: [ trans: $2 ] ]
                     rest:  end                      ] ] ]
```

Figure 9. Feature structure after lexical rule

```
[ lex:    stormed
  cat:    V
  head:   [ trans: [ arg1: []
                     arg2: $1 []
                     pred: storm ]
            form:  passiveparticiple ]
  subcat: [ first: [ cat:  NP
                     head: [ trans: $1 ] ]
            rest:  end                     ] ]
```

A PC-PATR 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 (`=>`).[2]

Consider the information shown in figure 7. When the lexicon entry is loaded, it is initially assigned the feature structure shown in figure 8, which is the unification of the information given in the various fields of the lexicon entry. Since one of the the labels stored in the `\f` (feature) field is actually the name of a lexical rule, after the complete feature structure has been built, the named lexical rule is applied. After the rule has been applied, the feature structure has been changed to the one shown in figure 9. Note that all of the information in the output feature structure is from the input feature structure, but not all of the input feature information is found in the the output feature structure.

Figure 10. PC-PATR lexical rule for using PC-Kimmo

```
Define MapKimmoFeatures as
        <out cat>       = <in head pos>
        <out head>      = <in head>
        <out gloss>     = <in root>
        <out root_pos>  = <in root_pos>
```

Figure 11. Feature structure received from PC-Kimmo

```
[ cat:      Word
  clitic:   -
  drvstem:  -
  head:     [ agr:    [ 3sg: + ]
              finite: +
              pos:    V
              tense:  PRES
              vform:  S          ]
  root:     ‘sleep
  root_pos: V                      ]
```

Figure 12. Feature structure sent to PC-PATR

```
[ cat:       V
  gloss:     ‘sleep
  head:      [ agr:    [ 3sg: + ]
               finite: +
               pos:    V
               tense:  PRES
               vform:  S          ]
  lex:       sleeps
  root_pos: V                       ]
```

Using a lexical rule in conjunction with the PC-Kimmo morphological parser within PC-PATR is illustrated in figures 10-12. Figure 10 shows the lexical rule for mapping

[2] These two operators are equivalent in PC-PATR, 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.

from the top-level feature structure produced by the morphological parser to the bottom-level feature structure used by the sentence parser. Note that this rule must be named `MapKimmoFeatures` (unorthodox capitalization and all). Figure 11 shows the feature structure created by the PC-Kimmo parser. After the lexical rule shown in figure 10 has been applied (and after some additional automatic processing), the feature structure shown in figure 12 is passed to the PC-PATR parser.

Note that the feature structure passed to the PC-PATR parser always has both a `lex` feature and a `gloss` feature, even if the `MapKimmoFeatures` lexical rule does not create them. The default value for the `lex` feature is the original word from the sentence being parsed. The default value for the `gloss` feature is the concatenation of the glosses of the individual morphemes in the word.

In contrast to the `lex` and `gloss` features which are provided automatically by default, the `cat` feature must be provided by the `MapKimmoFeatures` lexical rule. There is no way to provide this feature automatically, and it is required for the phrase structure rule portion of PC-PATR.

5 Standard format

Some of the input control files that PC-PATR reads are *standard format* files. This means that the files are divided into records and fields. A standard format file contains at least one record, and some files may contain a large number of records. Each record contains one or more fields. Each field occupies at least one line, and is marked by a *field code* at the beginning of the line. A field code begins with a backslash character (\), and contains 1 or more printing characters (usually alphabetic) in addition.

If the file is designed to have multiple records, then one of the field codes must be designated to be the *record marker*, and every record begins with that field, even if it is empty apart from the field code. If the file contains only one record, then the relative order of the fields is constrained only by their semantics.

It is worth emphasizing that field codes must be at the *beginning* of a line. Even a single space before the backslash character prevents it from being recognized as a field code.

It is also worth emphasizing that record markers *must* be present even if that field has no information for that record. Omitting the record marker causes two records to be merge into a single record, with unpredictable results.

6 The PC-PATR Lexicon File

The lexicon file is a *standard format* database file consisting of any number of records, each of which represents one word. These records are divided into fields, each of which begins with a standard format marker at the beginning of a line. These markers begin with the \ (backslash) character followed by one or more alphanumeric characters. Each record begins with a designated field. PC-PATR recognizes four different fields, with these default field markers:

`\w` the lexical form of the word, spelled exactly as it will appear in any sentences or phrases input to PC-PATR[1]

`\c` word category (part of speech)

`\g` word gloss

`\f` additional features of this word

Note that the fields containing the lexical form of the word and its category must be present for each word (record) in the lexicon. The other two fields (glosses and features) are optional, as are additional fields that may be present for other purposes.

Each word loaded from the lexicon file is assigned certain features based on the fields described above.

- The value of the *lex* feature is the lexical form of the word, taken from the lexical form field of the word's entry in the lexicon.
- The value of the *cat* feature is the lexical category of the word, for example, Noun, Verb, Adjective, and so on. This is taken from the category field of the word's entry in the lexicon. Note that the same lexical form can appear multiple times in the lexicon, with a different category for each occurrence.
- The value of the *gloss* feature is the gloss of the word, taken from the gloss field of the word's entry in the lexicon. Unlike the previous two items, this feature is optional.

These feature names should be treated as reserved names and not used for other purposes.

For example, consider these entries for the words *fox* and *foxes*:

```
\w fox
\c N
\g canine
\f <number> = singular

\w foxes
\c N
\g canine+PL
\f <number> = plural
```

When these entries are used by the grammar, they are represented by these feature structures:

[1] By default, `\w` also marks the initial field of each word's record.

```
[cat:    N
 gloss:  canine
 lex:    foxes
 number: singular]

[cat:    N
 gloss:  canine+PL
 lex:    foxes
 number: plural]
```

The lexicon entries can be simplified by defining feature templates in the grammar file. Consider the following templates:

```
Let PL be <number> = plural
Let N  be <number> = !singular
```

With these two templates, defining an abbreviation for "plural" and defining a default feature for category N (noun), the lexicon entries can be rewritten as follows:

```
\w fox
\c N
\g canine
\f

\w foxes
\c N
\g canine+PL
\f PL
```

Note that the feature (`\f`) field of the first entry could be omitted altogether since it is now empty.

7 The AMPLE Analysis File

Rather than using a dedicated lexicon file, PC-PATR can load its internal lexicon from one or analysis files produced by the AMPLE morphological analysis program. AMPLE writes a standard format database for its output, each record of which corresponds to a word of the source text. The first field of each entry contains the analysis. Other fields, which may or may not occur, contain additional information.

The utility of this command has been greatly reduced by the availability of the `load ample` and `load kimmo` commands which allow morphological analysis on demand to populate PC-PATR's word lexicon. However, the `file disambiguate` command also operates on AMPLE analysis files, so this information is still of interest.

7.1 AMPLE analysis file fields

This section describes the fields that AMPLE writes to the output analysis file. The only field that is guaranteed to exist is the analysis (`\s`) field. All other fields are either data dependent or optional.

7.1.1 Analysis: \a

The analysis field (`\a`) starts each record of the output analysis file. It has the following form:

```
\a PFX IFX PFX < CAT root CAT root > SFX IFX SFX
```

where `PFX` is a prefix morphname, `IFX` is an infix morphname, `SFX` is a suffix morphname, `CAT` is a root category, and `root` is a root gloss or etymology. In the simplest case, an analysis field would look like this:

```
\a < CAT root >
```

The `\rd` field in the analysis data file can replace the characters used to bracket the root category and gloss/etymology; see section "Root Delimiter Characters: \rd" in *AMPLE Reference Manual*. The dictionary field code mapped to `M` in the dictionary codes file controls the affix and default root morphnames; see section "Morphname (internal code M)" in *AMPLE Reference Manual*. If the AMPLE '`-g`' command line option was given, the output analysis file contains glosses from the root dictionary marked by the field code mapped to `G` in the dictionary codes file; see section "AMPLE Command Options" in *AMPLE Reference Manual*, and section "Root Gloss (internal code G)" in *AMPLE Reference Manual*.

7.1.2 Decomposition (surface forms): \d

The morpheme decomposition field (`\d`) follows the analysis field. It has the following form:

```
\d anti-dis-establish-ment-arian-ism-s
```

where the hyphens separate the individual morphemes in the surface form of the word.

The `\dsc` field in the text input control file can replace the hyphen with another character for separating the morphemes; see section "Decomposition Separation Character: \dsc" in *AMPLE Reference Manual*.

The morpheme decomposition field is optional. It is enabled either by an AMPLE '`-w d`' command line option (see section "AMPLE Command Options" in *AMPLE Reference Manual*), or by an interactive query.

7.1.3 Category (possible word or morpheme): \cat

The category field (`\cat`) provides rudimentary category information. It has the following form:

```
\cat CAT
```

where `CAT` is the proposed word category. A more complex example is

```
\cat C0 C1/C0=C2=C2/C1=C1/C1
```

where `C0` is the proposed word category, `C1/C0` is a prefix category pair, `C2` is a root category, and `C2/C1` and `C1/C1` are suffix category pairs. The equal signs (`=`) serve to separate the category information of the individual morphemes.

The `\cat` field of the analysis data file controls whether the category field is written to the output analysis file; see section "Category output control: \cat" in *AMPLE Reference Manual*.

7.1.4 Properties: \p

The properties field (`\p`) contains the names of any allomorph or morpheme properties found in the analysis of the word. It has the form:

```
\p ==prop1 prop2=prop3=
```

where `prop1`, `prop2`, and `prop3` are property names. The equal signs (`=`) serve to separate the property information of the individual morphemes. Note that morphemes may have more than one property, with the names separated by spaces, or no properties at all.

By default, the properties field is written to the output analysis file. The '`-w 0`' command option, or any '`-w`' option that does not include '`p`' in its argument disables the properties field.

7.1.5 Feature Descriptors: \fd

The feature descriptor field (`\fd`) contains the feature names associated with each morpheme in the analysis. It has the following form:

```
\fd ==feat1 feat2=feat3=
```

where `feat1`, `feat2`, and `feat3` are feature descriptors. The equal signs (`=`) serve to separate the feature descriptors of the individual morphemes. Note that morphemes may have more than one feature descriptor, with the names separated by spaces, or no feature descriptors at all.

The dictionary field code mapped to `F` in the dictionary code table file controls whether feature descriptors are written to the output analysis file; if this mapping is not defined, then the `\fd` field is not written. See section "Feature Descriptor (internal code F)" in *AMPLE Reference Manual*.

7.1.6 Underlying forms (decomposition): \u

The underlying form field (`\u`) is similar to the decomposition field except that it shows underlying forms instead of surface forms. It looks like this:

```
\u a-para-a-i-ri-me
```

where the hyphens separate the individual morphemes.

The `\dsc` field in the text input control file can replace the hyphen with another character for separating the morphemes; see section "Decomposition Separation Character: \dsc" in *AMPLE Reference Manual*.

The dictionary field code mapped to `U` in the dictionary code table file controls whether underlying forms are written to the output analysis file; if this mapping is not defined, then the `\u` field is not written. section "Underlying Form (internal code U)" in *AMPLE Reference Manual*.

7.1.7 Word (before decapitalization and orthography changes): \w

The original word field (`\w`) contains the original input word as it looks before decapitalization and orthography changes. It looks like this:

```
\w The
```

Note that this is a gratuitous change from earlier versions of AMPLE, which wrote the decapitalized form.

The original word field is optional. It is enabled either by an AMPLE '`-w w`' command line option (see section "AMPLE Command Options" in *AMPLE Reference Manual*), or by an interactive query.

7.1.8 Formatting (junk before the word): \f

The format information field (`\f`) records any formatting codes or punctuation that appeared in the input text file before the word. It looks like this:

```
\f \\id MAT 5 HGMT05.SFM, 14-feb-84 D. Weber, Huallaga Quechua\n
        \\c 5\n\n
        \\s
```

where backslashes (\) in the input text are doubled, newlines are represented by `\n`, and additional lines in the field start with a tab character.

The format information field is written to the output analysis file whenever it is needed, that is, whenever formatting codes or punctuation exist before words.

7.1.9 Capitalization flag: \c

The capitalization field (`\c`) records any capitalization of the input word. It looks like this:

```
\c 1
```

where the number following the field code has one of these values:

`1`	the first (or only) letter of the word is capitalized
`2`	all letters of the word are capitalized
`4--32767`	some letters of the word are capitalized and some are not

Note that the third form is of limited utility, but still exists because of the author's last name.

The capitalization field is written to the output analysis file whenever any of the letters in the word are capitalized; see section "Prevent Any Decapitalization: \nocap" in *AMPLE Reference Manual*, and section "Prevent Decapitalization of Individual Characters: \noincap" in *AMPLE Reference Manual*.

7.1.10 Nonalphabetic (junk after the word): \n

The nonalphabetic field (`\n`) records any trailing punctuation, bar code (see section "Bar Code Format Code Characters: \barcodes" in *AMPLE Reference Manual*), or whitespace characters. It looks like this:

```
\n |r.\n
```

where newlines are represented by `\n`. The nonalphabetic field ends with the last whitespace character immediately following the word.

The nonalphabetic field is written to the output analysis file whenever the word is followed by anything other than a single space character. This includes the case when a word ends a file with nothing following it.

7.2 Ambiguous analyses

The previous section assumed that AMPLE produced only one analysis for a word. This is not always possible since words in isolation are frequently ambiguous. AMPLE handles multiple analyses by writing each analysis field in parallel, with the number of analyses at the beginning of each output field. For example,

```
\a %2%< A0 imaika > CNJT AUG%< A0 imaika > ADVS%
\d %2%imaika-Npa-ni%imaika-Npani%
\cat %2%A0 A0=A0/A0=A0/A0%A0 A0=A0/A0%
\p %2%==%=%
\fd %2%==%=%
\u %2%imaika-Npa-ni%imaika-Npani%
\w Imaicampani
\f \\v124
\c 1
\n \n
```

where the percent sign (`%`) separates the different analyses in each field. Note that only those fields which contain analysis information are marked for ambiguity. The other fields (`\w`, `\f`, `\c`, and `\n`) are the same regardless of the number of analyses that AMPLE discovers.

The `\ambig` field in the text input control file can replace the percent sign with another character for separating the analyses; see section "Ambiguity Marker Character: \ambig" in *AMPLE Reference Manual*, for details.

7.3 Analysis failures

The previous sections assumed that AMPLE successfully analyzed a word. This does not always happen. AMPLE marks analysis failures the same way it marks multiple analyses, but with zero (`0`) for the ambiguity count. For example,

```
\a %0%ta%
\d %0%ta%
\cat %0%%
\p %0%%
\fd %0%%
\u %0%%
\w TA
\f \\v 12 |b
\c 2
\n |r\n
```

Note that only the `\a` and `\d` fields contain any analysis information, and those both have the decapitalized word as a place holder.

The `\ambig` field in the text input control file can replace the percent sign with another character for marking analysis failures and ambiguities; see section "Ambiguity Marker Character: \ambig" in *AMPLE Reference Manual*, for details.

8 Using the Embedded Morphological Parsers

Normally, PC-PATR requires the linguist to develop a full-fledged lexicon of words with their features. This may be unnecessary if a morphological analysis, and a comprehensive lexicon of morphemes, has already been developed using either PC-Kimmo (version 2) or AMPLE (version 3). These morphological parsing programs are also available from SIL.

8.1 PC-Kimmo

Version 2 of PC-Kimmo supports a PC-PATR style grammar for defining word structure in terms of morphemes. This provides a straightforward way to obtain word features as a result of the morphological analysis process. For best results, the (PC-Kimmo) word grammar and the (PC-PATR) sentence or phrase grammar should be developed together.

When using the PC-Kimmo morphological parser, PC-PATR requires a special lexical rule in the (sentence level) grammar file. This rule is named `MapKimmoFeatures` and is used automatically to map from the features produced by the word parse to the features needed by the sentence parse. For example, consider the following definition:

```
Define MapKimmoFeatures as
        <out cat>       = <in head pos>
        <out lex>       = <in lex>
        <out head>      = <in head>
```

This lexical rule uses the `<head pos>` feature produced by the PC-Kimmo parser as the `<cat>` feature for the PC-PATR parser, and passes the `<lex>` and `<head>` features from the morphological parser to the sentence parser unchanged.

8.2 AMPLE

The only thing necessary to use the AMPLE morphological parser inside PC-PATR is to load the appropriate control files and dictionaries. This will not be too useful, however, unless the AMPLE dictionaries contain feature descriptors to pass through to PC-PATR. It is also required for the AMPLE data to define the word category. (Either the word-final suffix category or the word-initial prefix category can be designated in the analysis data file). Consult the AMPLE documentation for more details on either of these issues.

9 Index

Table of Contents