DICTIONARIES, DICTIONARY GRAMMARS AND DICTIONARY ENTRY PARSING

Mary S. Neff
IBM T. J. Watson Research Center, P. O. Box 704, Yorktown Heights, New York 10598

Branimir K. Boguraev
IBM T. J. Watson Research Center, P. O. Box 704, Yorktown Heights, New York 10598;
Computer Laboratory, University of Cambridge, New Museums Site, Cambridge CB2 3QG

Computerist: ... But, great Scott, what about structure? You can't just bang that lot into a machine without structure. Half a gigabyte of sequential file ...

Lexicographer: Oh, we know all about structure. Take this entry for example. You see here italics as the typical ambiguous structural element marker, being apparently used as an undefined phrase-entry lemma, but in fact being the subordinate entry headword address preceding the small-cap cross-reference headword address which is nested within the gloss to a defined phrase entry, itself nested within a subordinate (bold lower-case letter) sense section in the second branch of a forked multiple part of speech main entry. Now that's typical of the kind of structural relationship that must be made crystal-clear in the eventual database.

from "Taking the Words out of His Mouth" —
Edmund Weiner on computerising the *Oxford English Dictionary*
(*The Guardian*, London, March, 1985)

ABSTRACT

We identify two complementary processes in the conversion of machine-readable dictionaries into lexical databases: recovery of the dictionary structure from the typographical markings which persist on the dictionary distribution tapes and embody the publishers' notational conventions; followed by making explicit all of the codified and ellided information packed into individual entries. We discuss notational conventions and tape formats, outline structural properties of dictionaries, observe a range of representational phenomena particularly relevant to dictionary parsing, and derive a set of minimal requirements for a dictionary grammar formalism. We present a general purpose dictionary entry parser which uses a formal notation designed to describe the structure of entries and performs a mapping from the flat character stream on the tape to a highly structured and fully instantiated representation of the dictionary. We demonstrate the power of the formalism by drawing examples from a range of dictionary sources which have been processed and converted into lexical databases.

1. INTRODUCTION

Machine-readable dictionaries (MRD's) are typically available in the form of publishers' typesetting tapes, and consequently are represented by a flat character stream where lexical data proper is heavily interspersed with special (control) characters. These map to the font changes and other notational conventions used in the printed form of the dictionary and designed to pack, and present in a codified compact visual format, as much lexical data as possible.

To make maximal use of MRD's, it is necessary to make their data, as well as structure, fully explicit, in a data base format that lends itself to flexible querying. However, since none of the lexical data base (LDB) creation efforts to date fully addresses both of these issues, they fail to offer a general framework for processing the wide range of dictionary resources available in machine-readable form. As one extreme, the conversion of an MRD into an LDB may be carried out by a 'one-off' program — such as, for example, used for the *Longman Dictionary of Contemporary English* (LDOCE) and described in Boguraev and Briscoe, 1989. While the resulting LDB is quite explicit and complete with respect to the data in the source, all knowledge of the dictionary structure is embodied in the conversion program. On the other hand, more modular architectures consisting of a parser and a grammar — best exemplified by Kazman's (1986) analysis of the Oxford English Dictionary (OED) — do not deliver the structurally rich and explicit LDB ideally required for easy and unconstrained access to the source data.

The majority of computational lexicography projects, in fact, fall in the first of the categories above, in that they typically concentrate on the conversion of a single dictionary into an LDB: examples here include the work by e.g. Ahlswede *et al.*, 1986, on *The Webster's Seventh New Collegiate Dictionary*; Fox *et al.*, 1988, on *The Collins English Dictionary*; Calzolari and Picchi, 1988, on *Il Nuovo Dizionario Italiano Garzanti*; van der Steen, 1982, and Nakamura, 1988, on LDOCE. Even work based on multiple dictionaries (e.g. in bilingual context: see Calzolari and Picchi, 1986) appear to have used specialized programs for each dictionary source. In addition, not an uncommon property of the LDB's cited above is their incompleteness with respect to the original source: there is a tendency to extract, in a pre-processing phase, only some fragments (e.g.

part of speech information or definition fields) while ignoring others (e.g. etymology, pronunciation or usage notes).

We have built a Dictionary Entry Parser (DEP) together with grammars for several different dictionaries. Our goal has been to create a general mechanism for converting to a common LDB format a wide range of MRD's demonstrating a wide range of phenomena. In contrast to the OED project, where the data in the dictionary is only tagged to indicate its structural characteristics, we identify two processes which are crucial for the 'unfolding', or making explicit, the structure of an MRD: identification of the structural markers, followed by their interpretation in context resulting in detailed parse trees for individual entries. Furthermore, unlike the tagging of the OED, carried out in several passes over the data and using different grammars (in order to cope with the highly complex, idiosyncratic and ambiguous nature of dictionary entries), we employ a parsing engine exploiting unification and backtracking, and using a single grammar consisting of three different sets of rules. The advantages of handling the structural complexities of MRD sources and deriving corresponding LDB's in one operation become clear below.

While DEP has been described in general terms before (Byrd *et al.*, 1987; Neff *et al.*, 1988), this paper draws on our experience in parsing the *Collins German-English / Collins English-German* (CGE/CEG) and LDOCE dictionaries, which represent two very different types of machine-readable sources vis-à-vis format of the typesetting tapes and notational conventions exploited by the lexicographers. We examine more closely some of the phenomena encountered in these dictionaries, trace their implications for MRD-to-LDB parsing, show how they motivate the design of the DEP grammar formalism, and discuss treatment of typical entry configurations.

2. STRUCTURAL PROPERTIES OF MRD'S

The structure of dictionary entries is mostly implicit in the font codes and other special characters controlling the layout of an entry on the printed page; furthermore, data is typically compacted to save space in print, and it is common for different fields within an entry to employ radically different compaction schemes and abbreviatory devices. For example, the notation *T5a,b,3* stands for the LDOCE grammar codes *T5a;T5b;T3* (Boguraev and Briscoe, 1989, present a detailed description of the grammar coding system in this dictionary), and many adverbs are stored as run-ons of the adjectives, using the abbreviatory convention **~ly** (the same convention applies to certain types of affixation in general: **~er**, **~less**, **~ness**, etc.). In CGE, German compounds with a common first element appear grouped together under it:

Kinder-: **~chor** *m* children's choir; **~dorf** *nt* children's village; **~ehe** *f* child marriage.

Dictionaries often factor out common substrings in data fields as in the following LDOCE and CEG entries:

in.cu.ba.tor ... a machine for **a** keeping eggs warm until they HATCH **b** keeping alive babies that are too small to live and breathe in ordinary air

Figure 1. Definition-initial common fragment

Bankrott *m* **-(e)s, -e** bankruptcy; *(fig)* breakdown, collapse; *(moralisch)* bankruptcy. **~ machen** to become *or* go bankrupt; **den ~ anmelden** *or* **ansagen** *or* **erklären** to declare oneself bankrupt.

Figure 2. Definition-final common fragment

Furthermore, a variety of conventions exists for making text fragments perform more than one function (the capitalization of "HATCH" above. for instance, signals a close conceptual link with the word being defined). Data of this sort is not very useful to an LDB user without explicit expansion and recovery of compacted headwords and fragments of entries. Parsing a dictionary to create an LDB that can be easily queried by a user or a program therefore implies not only tagging the data in the entry, but also recovering ellided information, both in form and content.

There are two broad types of machine-readable source, each requiring a different strategy for recovery of implicit structure and content of dictionary entries. On the one hand tapes may consist of a character stream with no explicit structure markings (as OED and the *Collins* bilinguals exemplify); all of their structure is implied in the font changes and the overall syntax of the entry. On the other hand, sources may employ mixed representation, incorporating both global record delimiters and local structure encoded in font change codes and/or special character sequences (LDOCE and *Webster's Seventh*).

Ideally, all MRD's should be mapped onto LDB structures of the same type, accessible with a single query language that preserves the user's intuition about the structure of lexical data (Neff *et al.*, 1988; Tompa, 1986). Dictionary entries can be naturally represented as shallow hierarchies with a variable number of instances of certain items at each level, e.g. multiple homographs within an entry or multiple senses within a homograph. The usual inheritance mechanisms associated with a hierarchical organisation of data not only ensure compactness of representation, but also fit lexical intuitions. The figures overleaf show sample entries from CGE and LDOCE and their LDB forms with explicitly 'unfolded' structure.

Within the taxonomy of normal forms (NF) defined by relational data base theory, dictionary entries are 'unnormalized relations' in which attributes can contain other relations, rather than simple scalar values; LDB's, therefore, cannot be correctly viewed as relational data bases (see Neff *et al.*, 1988). Other kinds of hierarchically structured data similarly fall outside of the relational

title [...] *n* **(a)** Titel *m (also Sport); (of chapter)* Überschrift *f; (Film)* Untertitel *m; (form of address)* Anrede *f.* **what ~ do you give a bishop?** wie redet *or* spricht man einen Bischof an? **(b)** *(Jur) (right)* (Rechts)anspruch *(to* auf + *acc)*, Titel *(spec) m; (document)* Eigentumsurkunde *f.*

```
entry
+-hdw: title
|
+-superhom
  +-pronunc: ...
  +-hom
    +-pos: n
    +-sens
    | +-sensnum: a
    | +-tran_group
    | | +-tran
    | |   +-word: Titel
    | |   +-gender: m
    | |   +-domain: also Sport
    | |
    | +-tran_group
    | | +-usage_note: of chapter
    | | +-tran
    | |   +-word: Überschrift
    | |   +-gender: f
    | |
    | +-tran_group
    | | +-domain: Film
    | | +-tran
    | |   +-word: Untertitel
    | |   +-gender: m
    | |
    | +-tran_group
    | | +-usage_note: form of address
    | | +-tran
    | |   +-word: Anrede
    | |   +-gender: f
    | |
    | +-collocat
    |   +-source: what ~ do you give a bishop?
    |   +-targ
    |     +-target
    |       +-phrase: wie redet /or/ spricht
    |                 man einen Bischof an?
    +-sens
      +-sensnum: b
      +-domain: Jur
      +-tran_group
      | +-usage_note: right
      | +-tran
      | | +-word: Rechtsanspruch
      | | +-word: Anspruch
      | | +-complement
      | | | +-engcomp: to
      | | | +-gercomp: auf + acc
      | | +-gender: m
      | |
      | +-tran
      |   +-word: Titel
      |   +-style: spec
      |   +-gender: m
      |
      +-tran_group
        +-usage_note: document
        +-tran
          +-word: Eigentumsurkunde
          +-gender: f
```

Figure 3. LDB for a CEG entry

NF mould; indeed recently there have been efforts to design a generalized data model which treats flat relations, lists, and hierarchical structures uniformly (Dadam *et al.*, 1986). Our LDB format and Lexical Query Language (LQL) support the hierarchical model for dictionary data; the output of the parser, similar to the examples in Figure 3 and Figure 4, is compacted, encoded, and loaded into an LDB.

nui.sance /'nju:səns || 'nu:-/ *n* **1** a person or animal that annoys or causes trouble, PEST: *Don't make a nuisance of yourself: sit down and be quiet!* **2** an action or state of affairs which causes trouble, offence, or unpleasantness: *What a nuisance! I've forgotten my ticket* **3 Commit no nuisance** (as a notice in a public place) Do not use this place as **a** a lavatory **b** a TIP[4]

```
entry
+-hdw: nuisance
|
+-superhom
  +-print_form: nui.sance
  +-pronunc
  | +-primary
  |   +-pron_string: 'nju:sFns || 'nu:-
  +-syncat: n
  |
  +-sense_def
  | +-sense_no: 1
  | +-defn
  | | +-implicit_xrf
  | | | +-to: pest
  | | +-def_string: a person or animal that
  | |               annoys or causes trouble;
  | |               pest
  | +-example
  |   +-ex_string: Don't make a nuisance of
  |                yourself: sit down and
  |                be quiet!
  +-sense_def
  | +-sense_no: 2
  | +-defn
  | | +-def_string: an action or state of affairs
  | |               which causes trouble, offence,
  | |               or unpleasantness
  | +-example
  |   +-ex_string: What a nuisance!
  |                I've forgotten my ticket
  +-sense_def
    +-sense_no: 3
    +-defn
      +-hdw_phrase: Commit no nuisance
      +-qualifier: as a notice in a public place
      +-sub_defn
      | +-seq_no: a
      | +-defn
      |   +-def_string: Do not use this place
      |                 as a lavatory
      +-sub_defn
        +-seq_no: b
        +-defn
          +-implicit_xrf
          | +-to: tip
          | +-hom_no: 4
          +-def_string: Do not use this place
                        as a tip
```

Figure 4. LDB for an LDOCE entry

3. DEP GRAMMAR FORMALISM

The choice of the hierarchical model for the representation of the LDB entries (and thus the output of DEP) has consequences for the parsing mechanism. For us, parsing involves determining the structure of all the data, retrieving implicit information to make it explicit, reconstructing ellided information, and filling a (recursive) template, without *any* data loss. This contrasts with a strategy that fills slots in predefined (and finite) sets of records for a relational system, often discarding information that does not fit.

In order to meet these needs, the formalism for dictionary entry grammars must meet at least three criteria, in addition to being simply a notational device capable of describing any particular

dictionary format. Below we outline the basic requirements for such a formalism.

3.1 Effects of context

The grammar formalism should be capable of handling 'mildly context sensitive' input streams, as structurally identical items may have widely differing functions depending on both local and global contexts. For example, parts of speech, field labels, paraphrases of cultural items, and many other dictionary fragments all appear in the CEG in italics, but their context defines their identity and, consequently, their interpretation. Thus, in the example entry in Figure 3 above, *m*, (*also Sport*), (*of chapter*), and (*spec*) acquire the very different labels of **pos**, **domain**, **usage_note**, and **style**. In addition, to distinguish between domain labels, style labels, dialect labels, and usage notes, the rules must be able to test candidate elements against a closed set of items. Situations like this, involving subsidiary application of auxiliary procedures (e.g. string matching, or dictionary lookup required for an example below), require that the rules be allowed to selectively invoke external functions.

The assignment of labels discussed above is based on what we will refer to in the rest of this paper as *global* context. In procedural terms, this is defined as the expectations of a particular grammar fragment, reflected in the names of the associated rules, which will be activated on a given path through the grammar. Global context is a dynamic notion, best thought of as a 'snapshot' of the state of the parser at any point of processing an entry. In contrast, *local* context is defined by finite-length patterns of input tokens, and has the effect of identifying typographic 'clues' to the structure of an entry. Finally, *immediate* context reflects very local character patterns which tend to drive the initial segmentation of the 'raw' tape character stream and its fragmentation into structure- and information-carrying tokens.

These three notions underlie our approach to structural analysis of dictionaries and are fundamental to the grammar formalism design.

3.2 Structure manipulation

The formalism should allow operations on the (partial) structures delivered *during* parsing, and not as separate tree transformations once processing is complete. This is needed, for instance, in order to handle a variety of scoping phenomena (discussed in section 5 below), factor out items common to more than one fragment within the same entry, and duplicate (sub-)trees as complete LDB representations are being fleshed out. Consider the CEG entry for "abutment":

abutment [...] *n (Archit)* Flügel- *or* Wangenmauer *f.*

Here, as well as in "title" (Figure 3), a copy of the gender marker common to both translations needs to migrate back to the first **tran**. In addition, a copy of the common second compound element **-mauer** also needs to migrate (note that

```
entry
+-hdw: abutment
|
+-superhom
  +-pronunc: ...
  +-hom
    +-pos: n
    +-domain: Archit
    +-sens
      +-tran_group
        +-tran
        | +-word: Flügelmauer
        | +-gender: f
        +-tran
          +-word: Wangenmauer
          +-gender: f
```

identifying this needs a separate noun compound parser augmented with dictionary lookup).

An example of structure duplication is illustrated by our treatment of (implicit) cross-references in LDOCE, where a link between two closely related words is indicated by having one of them typeset in small capitals embedded in a definition of the other (e.g. "PEST" and "TIP" in the definitions of "nuisance" in Figure 4). The dual purpose such words serve requires them to appear on at least two different nodes in the final LDB structure: **def_string** and **implicit_xrf**. In order to perform the required transformations, the grammar formalism must provide an explicit handle on partial structures, as they are being built by the parser, together with operations which can manipulate them — both in terms of structure decomposition and node migration.

In general, the formalism must be able to deal with discontinuous constituents, a problem not dissimilar to the problems of discontinuous constituents in natural language parsing; however in dictionaries like the ones we discuss the phenomena seem less regular (if discontinuous constituents can be regarded as regular at all).

3.3 Graceful failure

The nature of the information contained in dictionaries is such that certain fields within entries do not use any conventions or formal systems to present their data. For instance, the "USAGE" notes in LDOCE can be arbitrarily complex and unstructured fragments, combining straight text with a variety of notational devices (e.g. font changes, item highlighting and notes segmentation) in such a way that no principled structure may be imposed on them. Consider, for example, the annotation of "loan":

loan[2] *v* ..... *esp. AmE* to give (someone) the use of, lend ..... USAGE It is perfectly good *AmE* to use **loan** in the meaning of **lend**: *He* **loaned** *me ten dollars.* The word is often used in *BrE*, esp. in the meaning 'to lend formally for a long period': *He* **loaned** *his collection of pictures to the public* GALLERY but many people do not like it to be used simply in the meaning of **lend** in *BrE*...

Notwithstanding its complexity, we would still like to be able to process the complete entry, recovering as much as we can from the regularly encoded information and only 'skipping' over its truly unparseable fragment(s). Consequently, the formalism and the underlying processing frame-

work should incorporate a suitable mechanism for explicitly handling such data, systematically occurring in dictionaries.

The notion of graceful failure is, in fact, best regarded as 'selective parsing'. Such a mechanism has the additional benefit of allowing the *incremental* development of dictionary grammars with (eventually) complete coverage, and arbitrary depth of analysis, of the source data: a particular grammar might choose, for instance, to treat everything but the headword, part of speech, and pronunciation as 'junk', and concentrate on elaborate parsing of the pronunciation fields, while still being able to accept all input without having to assign any structure to most of it.

4. OVERVIEW OF DEP

DEP uses as input a collection of 'raw' typesetting images of entries from a dictionary (i.e. a typesetting tape with 'begin-end' boundaries of entries explicitly marked) and, by consulting an externally supplied grammar specific for that particular dictionary, produces explicit structural representations for the individual entries, which are either displayed or loaded into an LDB. The system consists of a rule compiler, a parsing engine, a dictionary entry template generator, an LDB loader, and various development facilities, all in a PROLOG shell. User-written PROLOG functions and primitives are easily added to the system. The formalism and rule compiler use the Modular Logic Grammars of McCord (1987) as a point of departure, but they have been substantially modified and extended to reflect the requirements of parsing dictionary entries.

The compiler accepts three different kinds of rules corresponding to the three phases of dictionary entry analysis: *tokenization*, *retokenization*, and *parsing proper*. Below we present informally highlights of the grammar formalism.

4.1 Tokenization

Unlike in sentence parsing, where tokenization (or lexical analysis) is driven entirely by blanks and punctuation, the DEP grammar writer explicitly defines token delimiters and token substitutions. Tokenization rules specify a one-to-one mapping from a character substring to a rewrite token; the mapping is applied whenever the specified substring is encountered in the original typesetting tape character stream, and is only sensitive to immediate context. Delimiters are usually font change codes and other special characters or symbols; substitutions are atoms (e.g. `ital_correction`, `field_sep`) or structured terms (e.g. `font(italic)`, `sup("1")`). Tokenization breaks the source character stream into a mixture of tokens and strings; the former embody the notational conventions employed by the printed dictionary, and are used by the parser to assign structure to an entry; the latter carry the textual (lexical) content of the dictionary. Some sample rules for the LDOCE machine-readable source, marking the beginning and end of font changes, or making explicit special print symbols, are shown below (to facilitate readability, `{*AB}` represents the hexadecimal symbol `x'AB'`).

```
delim("{*46}",  font(italic)).
delim("{*CA}",  font(begin(small_caps))).
delim("{*CB}",  font(end(small_caps))).
delim("{*64}",  ital_correction).
delim("{*80}",  hyphen_mark).
```

Immediate context, as well as local string rewrite, can be specified by more elaborate tokenization rules, in which two additional arguments specify strings to be 'glued' to the strings on the left and right of the token delimiter, respectively. For CEG, for instance, we have

```
delim(". >u4<", pd, "", ">u4<").
delim(">u4<",   font(bold)).
delim(">u5<",   font(roman)).
```

Tokenization operates recursively on the string fragments formed by an active rule; thus, application of the first two rules above to the string `"xxx. >u4< yyy"` results in the following token list: `"xxx" . pd . font(bold) . "yyy"`.

4.2 Retokenization

Longer-range (but still local) context sensitivity is implemented via retokenization, the effect of which is the 'normalization' of the token list. Retokenization rules conform to a general rewrite format — a pattern on the left-hand side defines a context as a sequence of (explicit or variable place holder) tokens, in which the token list should be adjusted as indicated by the right-hand side — and can be used to perform a range of cleaning up tasks before parsing proper.

Streamlining the token list. Tokens without information- or structure-bearing content, such as associated with the codes for italic correction or thin space, are removed:

```
ital_correction : +Seg <=> +Seg.
```

Superfluous font control characters can be simply deleted, when they follow or precede certain data-carrying tokens which also incorporate typesetting information (such as a homograph superscript symbol or a pronunciation marker indicating the beginning of the scope of a phonetic font):

```
pron_mark : font(phonetic) <=> pron_mark.
font(X)   : sup(N)         <=> sup(N).
```

(Re)adjusting the token list. New tokens can be introduced in place of certain token sequences:

```
bra : font(italic) <=> begin(restriction).
font(roman) : ket  <=> end(restriction).
```

Reconstruction of string segments. Where the initial (blind) tokenization has produced spurious fragmentation, string segments can be suitably reconstructed. For instance, a hyphen-delimited sequence of syllables in place of the print form of a headword, created by tokenization on `hyphen_mark`), can be 'glued' back as follows:

```
+Syl_1 : hyphen_mark    : +Syl_2          :
         $stringp(Syl_1) : $stringp(Syl_2) :
    <=> $join(Seg, Syl_1."-".Syl_2.nil)   :
         +Seg.
```

This rule demonstrates a characteristic property of the DEP formalism, discussed in more detail

later: arbitrary Prolog predicates can be invoked to e.g. constrain rule application or manipulate strings. Thus, the rule only applies to string tokens surrounding a hyphen character; it manufactures, by string concatenation, a new segment which replaces the triggering pattern.

Further segmentation. Often strings need to be split, with new tokens inserted between the pieces, to correct infelicities in the tapes, or to insert markers between recognizably distinct contiguous segments that appear in the same font. The rule below implements the CGE/CEG convention that a swung dash is an implicit switch to bold if the current font is not bold already.

```
font(X) : $(~X=bold)    : +E : $stringp(E) :
          $concat(A,B,E) : $concat("~",*,B)
    <=> font(X) : +A : font(bold) : +B.
```

Dealing with irregular input. Rules that rearrange tokens are often needed to correct errors in the tapes. In CEG/CGE, parentheses surrounding italic items often appear (erroneously) in a roman font. A suite of rules detaches the stray parentheses from the surrounding tokens, moves them around the font marker, and glues them to the item to which they belong.

```
+E : $stringp(E) : $concat(") ",E1,E)
   <=>  ")" : +E1.              /* detach */
font(F) : ")"
   <=>  ")" : retoken(font(F)). /* move */
+E : $stringp(E) : ")" : $concat(E,")",E1)
   <=>  +E1.                    /* glue  */
```

`retoken` invokes retokenization recursively on the sublist beginning with `font(F)` and including all tokens to its right. In principle, the three rules can be subsumed by a single one; in practice, separate rules also 'catch' other types of erroneous or noisy input.

Although retokenization is conceptually a separate process, it is interleaved in practice with tokenization, bringing improvements in performance. Upon completion, the tape stream corresponding, for instance, to the LDOCE entry

> **au.tis.tic** /ɔ:'tistik/ *adj* suffering from AUTISM[1]: *autistic child/behaviour* — **~ally** *adv* [Wa4]

```
F<autistic<F<>au(*80)tis(*80)ticP<C:"tIst
IkM<adj<S<0000<D<suffering from(*CA)autis
m(*CB)(*8A):(*46)autistic children/behavi
our(*64)R<01<R<~ally<R<><adv<Wa4<
```

is converted into the following token list:

```
head_marker              . "autistic"
fld_sep  . pf_marker     . "au-tis-tic"
pron_marker              . "C:"tIstIk"
pos_marker               . "adj"
fld_sep  . scod_marker   . "0000"
defn_marker              . "suffering from"
font(begin(small_caps)). "autism"
sup("1") . begin(exmp)   . "autistic
                           child/behaviour"
runon_marker             . "01"
begin(deriv)             . "autistically"
end(deriv) . fld_sep     . "adv"
fld_sep                  . "Wa4" . fld_sep .
```

4.3 Parsing

Parsing proper makes use of unification and backtracking to handle identification of segments by context, and is heavily augmented with some non-trivial manipulation of (partial) trees, as implicit and/or ellided information packed in the entries is being recovered and reorganized. Parsing is a top-down depth-first operation, and only the first successful parse is used. This strategy, augmented by a 'junk collection' mechanism (discussed below) to recover from parsing failures, turns out to be adequate for handling all of the phenomena encountered while assigning structural descriptions to dictionary entries.

Dictionary grammars follow the basic notational conventions of logic grammars; however, we use additional operators tailored to the structure manipulation requirements of dictionary parsing. In particular, the right-hand side of grammar rules admits the use of four different types of operators, designed to deal with *token list consumption*, *token list manipulation*, *structure assignment*, and *(local) tree transformations*. These operators suitably modify the expansions of grammar rules; ultimately, all rules are compiled into Prolog.

Token consumption. Tokens are removed from the token list by the `+` and `-` operators; `+` also assigns them as terminal nodes under the head of the invoking rule. Typically, delimiters introduced by tokenization (and retokenization) are removed once they serve their primary function of identifying local context; string segments of the token list are assigned labels and migrate to appropriate places in the final structural representation of an entry. A simple rule for the part of speech fields in CEG (Figure 3) would be:

```
pos   ==> -font(italic) : +Seg.
```

A structured term `s(pos, "n".nil)` is built as a result of the rule consuming, for instance, the token `"n"`, Rule names are associated with attributes in the LDB representation for a dictionary entry; structures built by rules are pairs of the form `s(name, Value)`, where `Value` is a list of one or more elements (strings or further structures 'returned' by recursively invoked rules).

Token list manipulation. Adjustment of the token list may be required in, for instance, simple cases of recovering ellided information or reordering tokens in the input stream. This is achieved by the `ins` and `insl` operators, which respectively insert single, or sequences of, tokens into the token list at the current position; and the `++` operator, which inserts tokens (or arbitrary tree fragments) directly into the structure under construction. Assuming a global variable, `Head`, bound to the headword of the current entry, and the ability to invoke a Prolog string concatenation function from within a rule (via the `$` operator; see below), abbreviated morphological derivations stored as run-ons might be recovered by:

```
run_on ==> -runon_mark : -font(bold) : -Seg :
           $concat("~", X, Seg)      :
           $isa(X, suffix)           :
           $concat(Head, X, Deriv)   :
           ++Deriv.
```

(`isa` is separately defined to test for membership of a closed class of suffixes.)

Structure assignment. The ++ operator can only assign arbitrary structures directly to the node in the tree which is currently under construction. A more general mechanism for retaining structures for future use is provided by allowing variables to be (optionally) associated with grammar rules: in this way the grammar writer can obtain an explicit handle on tree fragments, in contrast to the default situation where each rule implicitly 'returns' the structure it constructs to its caller. The following rule, for example, provides a skeleton treatment to the situation exemplified in Figure 4, where a definition-initial substring is common to more than one sub-definition:

```
defs          ==> -Seg : $stringp(Seg) :
                  subdefs(Seg).

subdefs(X)    ==> subdef(X) : opt(subdefs(X)).
subdef(X)     ==> -font(bold) :
                  sd_letter : -font(roman)   :
                  -Seg :
                  $concat(X, Seg, DefString) :
                  ins(DefString) : def_string.

sd_letter     ==> +Seg : $verify(Seg, "abc").
def_string    ==> +Seg : $stringp(Seg).
```

The **defs** rule removes the definition-initial string segment and passes it on to the repeatedly invoked **subdefs**. This manufactures the complete definition string by concatenating the common initial segment, available as an argument instantiated two levels higher, with the continuation string specific to any given sub-definition.

Tree transformations. The ability to refer, by name, to fragments of the tree being constructed by an active grammar rule, allows arbitrary tree transformations using the complementary operators -% and +%. They can only be applied to non-terminal grammar rules, and require the explicit specification of a place-holder variable as a rule argument; this is bound to the structure constructed by the rule. The effect of these operators on the tree fragments constructed by the rules they modify is to prevent their incorporation into the local tree (in the case of -%), to explicitly splice it in (in the case of +%), or simply to capture it (%). The use of this mechanism in conjunction with the structure naming facility allows both permanent deletion of nodes, as well as their practically unconstrained migration between, and within, different levels of grammar (thus implementing node raising and reordering). It is also possible to write a rule which builds no structure (the utility of such rules, in particular for controlling token consumption and junk collection, is discussed in section 5).

Node-raising is illustrated by the grammar fragment below, which might be used to deal with certain collocation phenomena. Sometimes dictionaries choose to explain a word in the course of defining another related word by arbitrarily inserting mini-entries in their definitions:

> **lach.ry.mal** 'lækrimal *adj* [Wa5] of or concerning tears of the organ (**lachrymal gland**/'... ../) of the body that produces them

The potentially complex structure associated with the embedded entry specification does not belong to the definition string, and should be factored out as a separate node moved to a higher level of the tree, or even used to create a new tree entirely. The rule for parsing the definition fields of an entry makes a provision for embedded entries; the structure built as an **embedded_entry** is bound to the **Struc** argument in the **defn** rule. The -% operator prevents the **embedded_entry** node from being incorporated as a daughter to **defn**; however, by unification, it begins its migration 'upwards' through the tree, till it is 'caught' by the **entry** rule several levels higher and inserted (via +%) in its logically appropriate place.

```
entry          ==> head : pron : pos : code :
                   defn(Embedded) :
                   +%embedded_entry(Embedded).

defn(Struc)    ==> -Seg1 : $stringp(Seg1)  :
                   -%embedded_entry(Struc) :
                   -Seg2 : $stringp(Seg2)  :
                   $concat(Seg1,Seg2,
                           DefString) :
                   ++DefString.

embedded_entry ==> -bra : ..... : -ket.
```

Capturing generalizations / execution control. The expressive power of the system is further enhanced by allowing optionality (via the **opt** operator), alternations (|) and conditional constructs in the grammar rules; the latter are useful both for more compact rule specification and to control backtracking while parsing. Rule application may be constrained by arbitrary tests (invoked, as Prolog predicates, via a **$** operator), and a **string** operator is available for sampling local context. The mechanism of escaping to Prolog, the motivation for which we discuss below, can also be invoked when arbitrary manipulation of lexical data — ranging from e.g. simple string processing to complex morphological analysis — is required during parsing.

Tree structures. Additional control over the shape of dictionary entry trees is provided by having two types of non-terminal nodes: *weak* and *strong* ones. The difference is in the explicit presence or absence of nodes, corresponding to the rule names, in the final tree: a structure fragment manufactured by a weak non-terminal is effectively 'spliced' into the higher level structure, without an intermediate level of naming. One common use of such a device is the 'flattening' of branching constructions, typically built by recursive rules: the declaration

```
strong_nonterminals (defs . subdef . nil).
```

when applied to the sub-definitions fragment above, would lead to the creation of a group of sister **subdef** nodes, immediately dominated by a **defs** node. Another use of the distinction between weak and strong non-terminals is the effective mapping from typographically identical entry segments to appropriately named structure fragments, with global context driving the name assignment. Thus, assuming a weak **label** rule which captures the label string for further testing, analysis of the example labels discussed in 3.1 could be achieved as follows (also see Figure 3):

```
label(X)    ==> -begin(restriction)  : +X :
                $stringp(X) : -end(restriction).
tran        ==> opt ( domain | style | dial |
                      usage_note )   : word.

style       ==> label(X) : $isa(X, styl_lab).
domain      ==> label(X) : $isa(X, domn_lab).
dial        ==> label(X) : $isa(X, dial_lab).
usage_note  ==> label(X).
```

Such a mechanism captures generalities in typographic conventions employed across any given dictionary, and yet preserves the distinct name spaces required for a meaningful 'unfolding' of a dictionary entry structure.

5. RANGE OF PHENOMENA TO HANDLE

Below we describe some typical phenomena encountered in the dictionaries we have parsed and discuss their treatment.

5.1 Messy token lists: controlling token consumption

The unsystematic encoding of font changes before, as well as after, punctuation marks (commas, semicolons, parentheses) causes blind tokenization to remove punctuation marks from the data to which they are visually and conceptually attached. As already discussed (see 4.2), most errors of this nature can be corrected by retokenization. Similarly, the confusing effects of another pervasive error, namely the occurrence of consecutive font changes, can be avoided by having a retokenization rule simply remove all but the last one. In general, context sensitivity is handled by (re)adjusting the token list; retokenization, however, is only sensitive to local context. Since global context cannot be determined unequivocally till parsing, the grammar writer is given complete control over the consumption and addition of tokens as parsing proceeds from left to right — this allows for motivated recovery of ellisions, as well as discarding of tokens in local transformations.

For instance, spurious occurrences of a font marker before a print symbol such as an opening parenthesis, which is not affected by a font declaration, clearly cannot be removed by a retokenization rule

```
font(roman) : bra <=> bra.
```

(The marker may be genuinely closing a font segment prior to a different entry fragment which commences with, e.g., a left parenthesis). Instead, a grammar rule anticipating a `bra` token within its scope can readjust the token list using either of:

```
... ==> ... : -font(roman) : -bra : ins(bra).
... ==> ... : -font(roman) : string(bra.*).
```

(The `string` operator tests for a token list with `bra` as its first element.)

5.2 The Peter–1 principle: scoping phenomena

Consider the entry for "Bankrott" in Figure 2. Translations sharing the label *(fig)* ("breakdown, collapse") are grouped together with commas and separated from other lists with semicolons. The restriction (context or label) precedes the list and can be said to scope 'right' to the next semicolon. We place the right-scoping labels or context under the (semicolon-delimited) `tran_group` as sister nodes to the multiple (comma-delimited) `tran` nodes (see also the representation of "title" in Figure 3). Two principles are at work here: maintaining implicit evidence of synonymy among terms in the target language responds to the "do not discard anything" philosophy; placing common data items as high as possible in the tree (the 'Peter-minus-1 principle') is in the spirit of Flickinger *et al.* (1985), and implements the notion of placing a terminal node at the highest position in the tree where its value is valid in combination with the values at or below its sister nodes. The latter principle also motivates sets of rules like

```
entry     ==> ... pron ... : homograph ... .
homograph ==> ... pron ... .
```

used to account for entries in English where the pronunciation differs for different homographs.

5.3 Tribal memory: rule variables

Some compaction or notational conventions in dictionaries require a mechanism for a rule to remember (part of) its ancestry or know its sister's descendants. Consider the problem of determining the scope of gender or labels immediately following variants of the headword:

> **Advokaturbüro** *nt (Sw)*, **Advokaturskanzlei** *f (Aus)* lawyer's office.
>
> **Tippfräulein** *nt (inf)*, **Tippse** *f* -, **-n** *(pej)* typist.
>
> **Alchemie** *(esp Aus)* , **Alchimie** *f* alchemy.

The first two entries show forms differing, respectively, in dialect and gender, and register and gender. The third illustrates other combinations. The rule accounting for labels after a variant must know whether items of like type have already been found after the headword, since items before the variant belong to the headword, different items of identical type following both belong individually, and all the rest are common to both. This 'tribal' memory is implemented using rule variables:

```
entry       ==> ... ( (dial : $(N=dial)) |
                      (N=nodial) ) :
                ... : opt(subhead(N)) ... .
subhead(N)  ==> ... opt( $(N=nodial) :
                         opt(dial) ) : ... .
```

In addition to enforcing rule constraints via unification, rule arguments also act as 'channels' for node raising and as a mechanism for controlling rule behaviour depending on invocation context.

This latter need stems from a pervasive phenomenon in dictionaries: the notational conventions for a logical unit within an entry persist across different contexts, and the sub-grammar for such a unit should be aware of the environment it is activated in. Implicit cross-references in LDOCE are consistently introduced by `font(small_caps)`, independent of whether the running text is a definition (roman font), example (italic), or an em-

bedded phrase or idiom (bold); by enforcing the return to the font active before the invocation of **implicit_xrf**, we allow the analysis of cross-references to be shared:

```
implicit_xrf(X) ==> -font(begin(small_caps))
                    : ... : -font(X).

df_txt ==> ... : implicit_xrf(roman)  : ... :
ex_txt ==> ... : implicit_xrf(italic) : ... :
id_txt ==> ... : implicit_xrf(bold)   : ... .
```

5.4 Unpacking, duplication and movement of structures: node migration

The whole range of phenomena requiring explicit manipulation of entry fragment trees is handled by the mechanisms for node raising, reordering, and deletion. Our analysis of implicit cross-references in LDOCE factors them out as separate structural units participating in the make-up of a word sense definition, as well as reconstructs a 'text image' of the definition text, with just the orthography of the cross-reference item 'spliced in' (see Figure 4).

```
defn              ==> def_segs(D_String)  :
                      def_string(D_String).

def_segs(Str_1)   ==> def_nugget(Seg)     :
                      ( def_segs(Str_0)   |
                        Str_0 = ""  )     :
                      $concat(Seg,Str_0,Str_1).

def_nugget(Ptr)   ==> %implicit_xrf
                       (s(implicit_xrf,
                         s(to, Ptr.nil).Rest)).
def_nugget(Seg)   ==> -Seg : $stringp(Seg).

def_string(Def)   ==> ++Def.
```

The rules build a definition string from any sequence of substrings or lexical items used as cross-references: by invoking the appropriate **def_nugget** rule, the simple segments are retained only for splicing the complete definition text; cross-reference pointers are extracted from the structural representation of an implicit cross-reference; and **implicit_xrf** nodes are propagated up to a sister position to the **def_string**. The string image is built incrementally (by string concatenation, as the individual **def_nuggets** are parsed); ultimately the **def_string** rule simply incorporates it into the structure for **defn**. Declaring **defn**, **def_string** and **implicit_xrf** to be strong non-terminals ultimately results in a **defn** structure similar to the one illustrated in Figure 4.

Copying and lateral migration of common gender labels in CEG translations, exemplified by "title" (Figure 3) and "abutment" (section 3.2), makes a different use of the **+%** operator. To capture the leftward scope of gender labels, in contrast to common (right-scoping) context labels, we create, for each noun translation (**tran**), a gender node with an empty value. The comma-delimited **tran** nodes are collected by a recursive weak non-terminal **trans** rule.

```
trans    ==> tran(G) : opt( -cm : trans(G) ).
tran(G)  ==> ... word ... :
             opt( -%gender(G) ) : +%gender(G).
```

The (conditional) removal of **gender** in the second rule followed by (obligatory) insertion of a **gender** node captures the gender if present and 'digs a hole' for it if absent. Unification on the last iteration of **trans** fills the holes.

Noun compound fragments, as in "abutment" can be copied and migrated forward or backward using the same mechanism. Since we have not implemented the noun compound parsing mechanism required for identification of segments to be copied, we have temporized by naming the fragments needing partners **alt_pfx** or **alt_sfx**.

5.5 Conflated lexical entries: homograph unpacking

We have implemented a mechanism to allow creation of additional entries out of a single one, for example from orthographic, dialect, or morphological variants of the original headword. Some CGE examples were given in sections 2 and 5.3 above. To handle these, the rules build the second entry inside the main one and manufacture cross reference information for both main form and variant, in anticipation of the implementation of a splitting mechanism. Examples of other types appear in both CGE and CEG:

> **vampire** [...] *n (lit)* Vampir, Blutsauger *(old) m*; *(fig)* Vampir *m*. ~ **bat** Vampir, Blutsauger *(old) m*.
>
> **wader** [...] *n* **(a)** *(Orn)* Watvogel *m*. **(b)** ~s *pl (boots)* Watstiefel *pl*.
>
> **house** *in cpds* Haus-; ~ **arrest** *n* Hausarrest *m*; ~ **boat** *n* Hausboot *nt*; ~ **bound** *adj* ans Haus gefesselt; ....
>
> **house:** ~**hunt** *vi* auf Haussuche sein; **they have started** ~**hunting** sie haben angefangen, nach einem Haus zu suchen; ~**hunting** *n* Haussuche *n*; ....

The conventions for morphological variants, used heavily in e.g. LDOCE and *Webster's Seventh*, are different and would require a different mechanism. We have not yet developed a generalized rule mechanism for ordering any kind of split; indeed we do not know if it is possible, given the wide variation in seemingly *ad hoc* conventions for 'sneaking in' logically separate entries into related headword definitions: the case of "lachrymal gland" in 4.3 is just one instance of this phenomena; below we list some more conceptually similar, but notationally different, examples, demonstrating the embedding of homographs in the variant, run-on, word-sense and example fields of LDOCE.

> **daddy long.legs** ˌdædi ˈlɔŋlegz also *(fml)* **crane fly** — *n* ... a type of flying insect with long legs
>
> **ac.ri.mo.ny** ... *n* bitterness, as of manner or language — **-nious** /ˌækriˈməuniəs/ *adj*: *an acrimonious quarrel* — **-niously** *adv*
>
> **crash**[1] ... *v* ... **6** *infml* also **gatecrash** — to join (a party) without having been invited ...
>
> **folk et.y.mol.o.gy** ˌ... *n* the changing of strange or foreign words so that they become like quite common ones: *some people say* **sparrowgrass** *instead of* ASPARAGUS: *that is an example of folk etymology*

5.6 Notational promiscuity: selective tokenization

Often distinctly different data items appear contiguous in the same font: the grammar codes of LDOCE (section 2) are just one example. Such run-together segments clearly need their own tokenization rules, which can only be applied when they are located during parsing. Thus, commas and parentheses take on special meaning in the string "X(*to be*)1,7", indicating, respectively, ellision of data and optionality of phrase. This is a different interpretation from e.g. alternation (consider the meaning of "*adj, noun*") or the enclosing of italic labels in parentheses (Figure 3). Submission of a string token to further tokenization is best done by invoking a special purpose pattern matching module; thus we avoid global (and blind) tokenization on common (and ambiguous) characters such as punctuation marks. The functionality required for selective tokenization is provided by a **parse** primitive; below we demonstrate the construction of a list of sister **syncat** nodes from a segment like "**n, v, adj**", repetitively invoking **parse** to break a string into two substrings separated by a comma:

```
syncats ==> -Seg : $stringp(Seg) :
            $parse(Hd." ,".Rest.nil, Seg) :
            insl(Hd.Rest.nil) :
            syncat : opt(syncats).
syncat  ==> +Seg : $isa(Seg, partofspeech).
```

5.7 Parsing failures: junk collection

The systematic irregularity of dictionary data (see section 3.3) is only one problem when parsing dictionary entries. Parsing failures in general are common during grammar development; more specifically, they might arise due to the format of an entry segment being beyond (easy) capturing within the grammar formalism, or requiring non-trivial external functionality (such as compound word parsing or noun/verb phrase analysis). Typically, external procedures operate on a newly constructed string token which represents a 'packed' unruly token list. Alternatively, if no format need be assigned to the input, the grammar should be able to 'skip over' the tokens in the list, collecting them under a 'junk' node.

If data loss is not an issue for a specific application, there is no need even to collect tokens from irregular token lists; a simple rule to skip over USAGE fields might be written as

```
usage     ==> -usage_mark : use_field.
use_field ==> -U_Token : $not(end_ufield) :
              opt(use_field).
```

(Rules like these, building no structure, are especially convenient when extensive reorganization of the token list is required — typically in cases of grammar-driven token reordering or token deletion *without* token consumption.)

In order to achieve skipping over unparseable input without data loss, we have implemented a **collective** rule class. The structure built by such rules the (transitive) concatenation of all the character strings in daughter segments. Coping with gross irregularities is achieved by picking up any number of tokens and 'packing' them together. This strategy is illustrated by a grammar for phrases conjoined with italic 'or' in example sentences and/or their translations (see Figure 3). The italic conjunction is surrounded by slashes in the resulting collected string as an audit trail. The extra argument to **conj** enforces, following the strategy outlined in section 5.3, rule application only in the correct font context.

```
strong_nonterminals (source . targ . nil).
collectives         (conj . nil).

source  ==> conj(bold).
targ    ==> conj(roman).
conj(X) ==> -font(X) : +Seg  : -font(ital) :
            ++" /"   : +"or" : ++"/ "      :
            -font(X) : +Seg.
```

Finally, for the most complex cases of truly irregular input, a mechanism exists for constraining junk collection to operate only as a last resort and only at the point at which parsing can go no further.

5.8 Augmenting the power of the formalism: escape to Prolog

Several of the mechanisms described above, such as contextual control of token consumption (section 5.1), explicit structure handling (5.4), or selective tokenization (5.6), are implemented as separate Prolog modules. Invoking such external functionality from the grammar rules allows the natural integration of the form- and content-recovery procedures into the top-down process of dictionary entry analysis. The utility of this device should be clear from the examples so far.

Such escape to the underlying implementation language goes against the grain of recent developments of declarative grammar formalisms (the procedural ramifications of, for instance, being able to call arbitrary LISP functions from the arcs of an ATN grammar have been discussed at length: see, for instance, the opening chapters in Whitelock *et al.*, 1987). However, we feel justified in augmenting the formalism in such a way, as we are dealing with input which is different in nature from, and on occasions possibly more complex than, straight natural language. Unhomogeneous mixtures of heavily formal notations and annotations in totally free format, interspersed with (occasionally incomplete) fragments of natural language phrases, can easily defeat any attempts at 'clean' parsing. Since the DEP system is designed to deal with an open-ended set of dictionaries, it must be able to confront a similarly open-ended set of notational conventions and abbreviatory devices. Furthermore. dealing in full with some of these notations requires access to mechanisms and theories well beyond the power of any grammar formalism: consider, for instance, what is involved in analyzing pronunciation fields in a dictionary, where alternative pronunciation patterns are marked only for syllable(s) which differ from the primary pronunciation (as in **arch.bish.op**: /ˌɑːtʃˈbiʃəp ‖ ˌɑr-/); where the pronunciation string itself is not marked for syllable structure; and where the assignment of syllable boundaries is far from trivial (as in **fas.cist**: /ˈfæʃəst/)!

6. CURRENT STATUS

The run-time environment of DEP includes grammar debugging utilities, and a number of options. All facilities have been implemented, except where noted. We have very detailed grammars for CGE (parsing 98% of the entries), CEG (95%), and LDOCE (93%); less detailed grammars for *Webster's Seventh* (98%), and both halves of the *Collins French Dictionary* (approximately 90%).

The Dictionary Entry Parser is an integral part of a larger system designed to recover dictionary structure to an arbitrary depth of detail, convert the resulting trees into LDB records, and make the data available to end users via a flexible and powerful lexical query language (LQL). Indeed, we have built LDB's for all dictionaries we have parsed; further development of LQL and the exploitation of the LDB's via query for a number of lexical studies are separate projects.

Finally, we note that, in the light of recent efforts to develop an interchange standard for (English mono-lingual) dictionaries (Amsler and Tompa, 1988), DEP acquires additional relevance, since it can be used, given a suitable annotation of the grammar rules for the machine-readable source, to transduce a typesetting tape into an interchangeable dictionary source, available to a larger user community.

ACKNOWLEDGEMENTS

We would like to thank Roy Byrd, Judith Klavans and Beth Levin for many discussions concerning the Dictionary Entry Parser system in general, and this paper in particular. Any remaining errors are ours, and ours only.

REFERENCES

Ahlswede, T, M Evens, K Rossi and J Markowitz (1986) "Building a Lexical Database by Parsing Webster's Seventh New Collegiate Dictionary", *Advances in Lexicology*, Second Annual Conference of the UW Centre for the New Oxford English Dictionary, 65 – 78.

Amsler, R and F Tompa (1988) "An SGML-Based Standard for English Monolingual Dictionaries", *Information in Text*, Fourth Annual Conference of the UW Centre for the New Oxford English Dictionary, 61 – 79.

Boguraev, B, and E Briscoe (Eds) (1989) *Computational Lexicography for Natural Language Processing*, Longman, Harlow.

Byrd, R, N Calzolari, M Chodorow, J Klavans, M Neff and O Rizk (1987) "Tools and Methods for Computational Lexicology", *Computational Linguistics*, vol.13(3 – 4), 219 – 240.

Calzolari, N and E Picchi (1986) "A Project for a Bilingual Lexical Database System", *Advances in Lexicology*, Second Annual Conference of the UW Centre for the New Oxford English Dictionary, 79 – 92.

Calzolari, N and E Picchi (1988) "Acquisition of Semantic Information from an On-Line Dictionary.", *Proceedings of the 12th International Conference on Computational Linguistics*, 87 – 92.

Collins (1980) *Collins German Dictionary: German – English, English – German*, Collins Publishers, Glasgow.

Garzanti (1984) *Il Nuovo Dizionario Italiano Garzanti*, Garzanti, Milano.

Longman (1978) *Longman Dictionary of Contemporary English*, Longman Group, London.

Dadam, P, K Kuespert, F Andersen, H Blanken, R Erbe, J Guenauer, V Lum, P Pistor and G Walsh (1986) "A DBMS Prototype to Support Extended NF2 Relations: An Integrated View on Flat Tables and Hierarchies", *Proceedings of ACM SIGMOD'86: International Conference on Management of Data*, 356 – 367.

Flickinger, D, C Pollard, T Wasow (1985) "Structure Sharing in Lexical Representation", *Proceedings of the 23rd Annual Meeting of the Association for Computational Linguistics*, 262 – 267.

Fox, E, T Nutter, T Alhswede, M Evens and J Markowitz (1988) "Building a Large Thesaurus for Information Retrieval", *Proceedings of the Second Conference on Applied Natural Language Processing*, 101 – 108.

Kazman, R (1986) *"Structuring the Text of the Oxford English Dictionary through Finite State Transduction"*, University of Waterloo Technical Report No. TR – 86 – 20.

McCord, M (1987) "Natural Language Processing and Prolog", in A Walker, M McCord, J Sowa and W Wilson (Eds) *Knowledge Systems and Prolog*, Addison-Wesley, Waltham, Massachusetts, 291 – 402.

Nakamura, J and Makoto N (1988) "Extraction of Semantic Information from an Ordinary English Dictionary and Its Evaluation", *Proceedings of the 12th International Conference on Computational Linguistics*, 459 – 464.

Neff, M, R Byrd and O Rizk (1988) "Creating and Querying Hierarchical Lexical Data Bases", *Proceedings of the Second Conference on Applied Natural Language Processing*, 84 – 93.

van der Steen, G J (1982) "A Treatment of Queries in Large Text Corpora", in S Johansson (Ed) *Computer Corpora in English Language Research*, Norwegian Computing Centre for the Humanities, Bergen, 49 – 65.

Tompa, F (1986) *"Database Design for a Dictionary of the Future"*, University of Waterloo, unpublished.

W7 (1967) *Webster's Seventh New Collegiate Dictionary*, C.&C. Merriam Company, Springfield, Massachussetts.

Whitelock, P, M Wood, H Somers, R Johnson and P Bennett (Eds) (1987) *Linguistic Theory and Computer Applications*, Academic Press, New York.