The ACQUILEX Lexical Database System: System Description and User Manual

John Carroll
jac@cl.cam.ac.uk

University of Cambridge Computer Laboratory
Pembroke Street, Cambridge CB2 3QG, UK

The Lexical Database System (LDB) is a computer system which provides efficient and flexible database-like access to multiple mono- and bilingual machine-readable dictionaries. It is configurable to new dictionaries, and supports a user in formulating queries to retrieve subsets of entries from one or more dictionaries. It forms part of the basic infrastructure software of the Esprit ACQUILEX project.

1. Introduction

The Lexical Database System (LDB) is a computer system which provides efficient and flexible database-like access to machine-readable dictionaries. The system has been in development at the University of Cambridge Computer Laboratory since 1986. The first version, described by Alshawi *et al.* (1988), was implemented in InterLisp-D on Xerox workstations and supported access exclusively to the Longman Dictionary of Contemporary English (LDOCE) (Procter, 1978). A number of studies which used this system are reported in Boguraev and Briscoe (1988).

In 1989 the system was re-implemented: it was necessary to move away from specialised workstations to general-purpose hardware, and to take account of the needs of the ACQUILEX project (a collaborative multilingual research project using both mono- and bilingual machine-readable dictionaries) which was about to start. The new system builds on the experience of using the first version, and also uses some of its basic implementational techniques. This second version of the LDB forms part of the basic infrastructure software of the ACQUILEX project.

As part of the re-implementation, the LDB was restructured to form a 'shell' for accessing machine-readable dictionaries (MRDs). This freed the LDB from being tied to a particular instance of a single MRD (as was the first version of the LDB to LDOCE), and enabled it to be configured to access new ones. Groups in the ACQUILEX project have set up several MRDs for access by the LDB[1]. Several further dictionaries containing information derived from these base MRDs (but augmented and enriched with new information) have been created and also set up for access.

The LDB is designed to be easy to use and user-friendly. On appropriate machines it may be driven almost entirely by mouse, using graphical displays, windows and pop-up menus to display data, results, user commands and choices. It has powerful look-up options (including conjunction, disjunction and negation operators, and wildcarding) making it easy to extract desired classes of words from an MRD. The LDB also provides a programmatic interface, allowing applications conveniently to retrieve data from dictionaries.

1.1 Looking up by Headword

The most straightforward way of accessing a dictionary with the LDB is to look up entries by headword. (Look-up by headword is similar to the traditional way of using printed dictionaries). Given a word, the LDB locates all entries which have that word as part of their homograph field, and displays them all, or just one that the user chooses. There may be several such entries since the same word may have more than one

[1] MRDs set up for access to date include LDOCE, the Longman Lexicon, the Oxford Advanced Learner's Dictionary of Current English, the MRC psycholinguistic database, monolingual Dutch and bilingual Dutch-English and English-Dutch dictionaries published by Van Dale, and the Vox Spanish, Spanish-English and English-Spanish dictionaries.

homograph, part of speech, or be part of one or more compounds. For example, the word *sound* in LDOCE has 1 adjective, 1 adverb, 2 noun and 2 verb homographs, and forms part of 5 compounds (e.g. *sound barrier*, *sound off*).

The LDB performs headword look-up efficiently using a (tree-structured) index file, since the size of MRDs (typically of the order of 10 Megabytes upwards) renders inadequate *ad hoc* methods of access such as sequential scanning of dictionary files in secondary storage. (A powerful virtual memory system might be able to manage simple modes of access by reading entire dictionaries into memory; however, this is not a general solution and is expensive in machine resources, particularly when having to deal simultaneously with more than a couple of average sized MRDs).

Look-up by headword is adequate for some applications, such as on-line extraction of syntactic or semantic information by a parser; however, a more flexible mode of access is usually required which is sensitive to the contents of entries rather than just to their headwords. Here the MRD is viewed as a *database*, from which entries may be retrieved in response to *queries*.

1.2 The Two-Level Dictionary Model

Boguraev *et al*. (1991) identify four classes of dictionary database models. The first of these follows the well-established notion of relational databases, mapping dictionary entries into a set of tables. Although this *relational* model of the lexicon can take advantage of established database technology, it is argued that it is unsuitable for mapping dictionaries into, given the intricate nature of, and subtle interactions within, lexical data. The second class is the *hierarchical* model which employs a structured representation to encode the complex structural relationships between the fields of entries (exploiting the insight that dictionary entries can naturally be regarded as shallow hierarchies with an indefinite number of attributes at each level). The third class is the *tagged* model; in contrast to the hierarchical model which fails to preserve the visual, human-readable interrelationships amongst the contents of dictionary entries, this model places the emphasis on preserving all of the information associated with the original printed form of the dictionary entry, but in the process fails to offer a natural way of making explicit statements concerning the implicit structural relationships of the elements within the entry.

The fourth class of dictionary model, the *two-level* model, combines the advantages of the hierarchical and tagged models and in doing so avoids their main drawbacks. This is the model implemented in the LDB. In the two-level model, the source dictionary is the primary repository of lexical data, and, separately from the dictionary source, sets of interrelated indices encode all statements about the structure and content of the data held in the dictionary. In the LDB, setting up (or 'mounting') a new MRD for access consists mainly of defining what these indices are, how they are to be extracted from entries, and the relationship between the information held in the indices and the high-level query interface presented to the user. The LDB automatically creates special files to hold the index information.

Thus, for example, the entry for the word *cobra* appears in the printed version of LDOCE as:

> **co·bra** /'kQbrE, 'kEu- II 'kEu / *n* a type of African or Asian poisonous snake that can spread the skin of its neck to make itself look bigger and more dangerous

The internal ASCII text representation of this in the MRD source file is:

```
((cobra) (1 C0173300 < co *80 bra) (3 "kQbrE , "kEU- = "kEU-) (5 n <)
(7 0 < < AMX- < ----A--V) (8 a type of African or Asian poisonous
snake that can spread the skin of its neck to make itself look bigger
and more dangerous))
```

Mounting the dictionary could extract the following constraints (on syntactic category, 'subject code' and definition words) from this entry; they would be added to an index file and associated with a pointer to the entry for *cobra*:

```
(@C n) (@S AM X-)
(@W A) (@W T Y P E) (@W O F) (@W A F R I C A N) (@W O R) ...
```

The two-level organisation means that the information about the definition word (`type`) occurring in the entry (for *cobra*) is stored in an index file and kept separate from the entry itself. The index explicitly represents the structural relationship between definition words and the entries in which they appear; this relationship would be absent in an implementation using the tagged model. The human-readable interrelationships within the entry stay visible, though, in contrast to an hierarchical model implementation, since the entry itself is still available.

In fact, since the original dictionary text remains available after initially mounting an MRD, the opportunity exists for the incremental addition of further indices, perhaps taking into account types of data in entries which had been ignored in previous indices, or expressing more complex structural relationships between data already encoded in them. Such additions can even be opportunistic, with more detailed analyses of the dictionary data (and the creation of the indices and associated constraints resulting from these analyses) being driven by the needs of the current research activity.

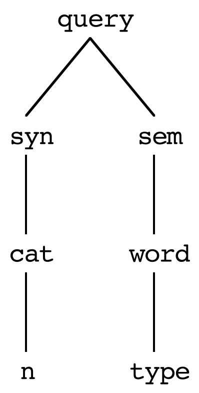
1.3 Looking up by Query

Once a dictionary has been mounted, it may be 'loaded' into an LDB session. Loading a dictionary causes its index files to be located and prepared for use, and menus of possible values for query attributes to be constructed. Once a dictionary has been loaded, the user may interactively formulate queries relating to that dictionary, and then for each query ask for all the entries which satisfy the query to be retrieved. (Section 3 outlines the extensive facilities the LDB provides for creating and modifying queries). Several dictionaries can be loaded in the same session: they are all available for access concurrently.

A query consists of a hierarchical collection of attributes with associated values; for example the query whose textual representation is

```
[[syn [cat n]]
 [sem [word type]]]
```

and graphical representation is



has two attributes at the top level: `syn` and `sem`; the attribute `cat` is beneath `syn` with value `n`, and `word` beneath `sem` with value `type`. The query means "return all entries which are nouns and whose definition contains the word `type`".

Looking up a query is a two-stage process. The LDB first maps the query onto a collection of constraints. (As mentioned above, the final stage in mounting an MRD involves defining the format of queries that the user can construct—that is, the possible attributes and their hierarchical organisation—and how these queries correspond to the indices created for the dictionary). For the query above (and following on from the previous example) the constraints would be:

```
(@C n) (@W T Y P E)
```

The LDB then determines which of these are the most discriminating (i.e. which have the lowest frequency, based on statistics which were gathered during the creation of the index files), and finds an initial set of entries which satisfy this subset of the constraints. In this example, a definition containing the word `type` would be more discriminating than just the entry being for a noun. The LDB then retrieves this set of entries from the source dictionary, checks which ones satisfy the remainder of the (less discriminating) constraints, and returns the ones which do as the final result. In the example, entries containing the word `type` might include *cobra* (as in "a type of African or Asian poisonous snake ...") and *subsume* ("to include as a member of a group or type"). The entry for *cobra* would pass the test for being a noun, but *subsume* would fail, so the final result would contain the former entry but not the latter. This process is summarised in figure 1 below. One of the index files that the LDB creates when a dictionary is mounted is a tree-structured file associating each possible constraint with its frequency of occurrence (and also with the location in a second subsidiary index file of the pointers to the entries from which it originates). Finding a particular constraint in this file involves just traversing a disc-resident discrimination network: once the constraint is found, its frequency of occurrence is immediately available. The portions of the network in the file that have been visited are cached in main memory for remainder of the LDB session, improving response time for subsequent queries involving the same constraint.

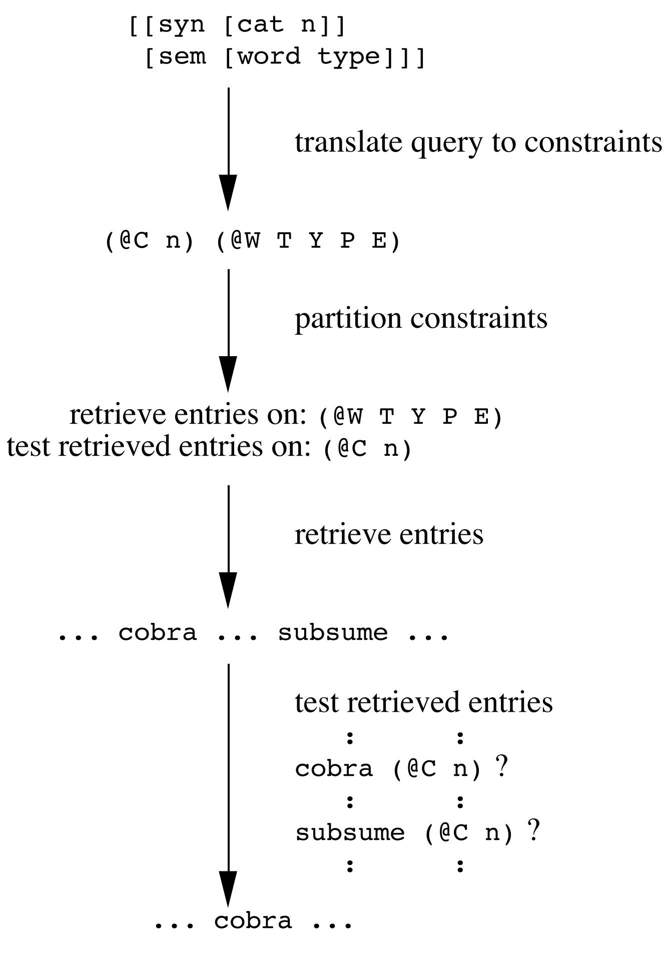


Figure 1. Steps performed during lookup of a query.

The partition of the constraints in the query into those used to form the initial candidate set of entries and those used to test these entries after they have been retrieved has a large influence on query look-up times. The most efficient strategy involves using the most specific constraints for looking up entries, more specific constraints ultimately yielding fewer entries. The optimal number of look-up constraints to use is determined by balancing the expected time taken in reading pointers (which increases with the number of look-up constraints) against the expected time taken in subsequently reading entries and testing them against the remaining constraints (which decreases with the number of look-up constraints[2]). The expected times depend on the speed of the machine running the LDB, so when the system starts up it runs a small test to gauge how fast the machine is.

It can take a significant amount of time to look up complex queries that return a large number of results. The LDB therefore allows the user to ask for just statistics giving counts of the number of entries satisfying each constraint in the query, the partitioning of the constraints into those that would be used for look-up and those for testing, and an estimate of the total number of results and time that it would take to compute them. If this estimate is satisfactory, the user can go ahead and ask for a full look-up, otherwise the query can be refined and the process repeated. The LDB, by default, computes the results in a sense-based (rather than an entry-based) fashion: that is, it returns just the senses which satisfy the query, not the whole entry (unless of course all the senses in the entry satisfy it). The LDB provides a number of possible ways of presenting results: after informing the user of how many entries and senses satisfied the query, the LDB can be asked to display the headwords of all the results, of a sample of them, of the first one, or nothing. In addition, a user-defined option allows any portion of result entries to be displayed rather than simply the headword, and results can also be returned as entry pointers to allow arbitrary set operations (e.g. union, intersection, set difference) to be applied to them.

1.4 Derived Dictionaries

The LDB allows several dictionaries to be loaded in the same session and makes them available for access concurrently. The user may at any one time have an arbitrary number of queries under construction for any number of the currently loaded dictionaries. A single query may usually only be applied to a single dictionary; however, a query may be applied to two or more dictionaries in a single operation if all of the dictionaries concerned are derived from a single 'source' dictionary. (In fact, one of the dictionaries may be the source itself). Such derived dictionaries will typically consist of an elaboration of a subset of the information in the source dictionary. Derived dictionaries make it possible to integrate information from diverse lexical (and non-lexical sources), and when several are

[2] An entry will only be read if there is a pointer to it in every pointer list. Therefore if n look-up constraints are used, returning pointer lists of length L_1, L_2, ... L_n, then the expected number of entries to be read (assuming statistical independence between lists) is $L_1L_2...L_n/D_2...D_n$, where D_i is the largest possible frequency of L_i. This decreases with n because L_i cannot exceed D_i and in fact is normally much smaller.

queried together they essentially behave as a single enriched store of lexical information.

An example of a derived dictionary (called Ldoce_inter) is a lexicon that we have derived semi-automatically from LDOCE, containing syntactic information in a format and level of detail suitable for use by a Natural Language parser. Ldoce_inter contains distinctions not explicitly present in LDOCE, for example marking verbs appropriately as ‘ergative’ or ‘unaccusative’. However, since Ldoce_inter is derived from LDOCE, a single query can refer sumultaneously both to it and to LDOCE. Thus the query

```
[[sem [box 5H]]
 [syn
   [il
      [Cat V]
      [Takes NP NP]
      [Type 2 Ergative]]]]
```

refers to ‘box code’ information in LDOCE (that the subject of a verb must be human) and to information held in Ldoce_inter (that the verb must take subject and object noun phrases, be ergative and a two-place predicate).

The LDB also makes it straightforward to create new derived dictionaries automatically based on the entries returned from looking up a query.

1.5 Hardware Requirements and Distribution

The LDB is written entirely in Common Lisp, and will run on any machine which has an implementation of this language. There are implementations of Common Lisp for most UNIX machines, the Apple Macintosh family, and IBM PC-compatibles running Windows 3 or OS/2. To date the LDB has been tested under Austin Kyoto Common Lisp, Lucid CL, Franz Allegro CL and Procyon CL. At present, it supports a graphical interface only when running under Procyon CL (on either the Macintosh or PC), or under Franz Allegro CL with Common Windows; however a textual TTY-style interface with the same functionality is available in all implementations.

The LDB is currently in use at the ACQUILEX project sites, and also in a number of other university and commercial research departments. It is available for research purposes in either binary or source code form. Contact the author at the University of Cambridge Computer Laboratory for details.

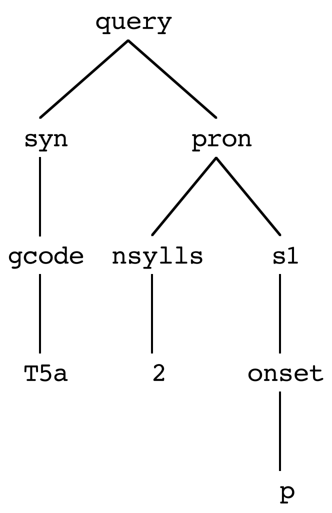
2. Queries

2.1 Query Structure

As mentioned above, in the LDB system dictionary queries can be represented in two forms: textual and graphical. In both forms they are trees whose branches are attribute names and whose leaves are attribute values. Thus in the query whose textual form is

```
[[syn
   [gcode T5a]]
 [pron
   [nsylls 2]
   [s1 [onset p]]]]
```

and whose graphical form is



the attribute names are `syn`, `gcode`, `pron`, `nsylls`, `s1` and `onset`, (of which `gcode`, `nsylls` and `onset` are terminal attributes) and the attribute values are `T5a`, `2` and `p`.

2.2 Attribute Names

In LDOCE, attribute names form a hierarchy, with `syn` (syntax), `sem` (semantics) and `pron` (pronunciation) at the top. As an example, at Cambridge when we mounted LDOCE we set up its attribute name hierarchy as follows:

`syn`			
	`cat`		syntactic category
	`c1`		first compound field
	`c2`		second compound field
	`gcode`		grammar code
	`label`		label field
	`multiple`		number of words in headword field, and whether hyphenated
`sem`			
	`antonym`		from the definition field (after '—opposite') (lower-cased)
	`box`		box codes
	`defn`		'implicit' x-refs and 'defining word' stems in definition (upper-cased)
	`order`		order of words in definition
	`subj`		subject codes
	`synonym`		taken from the definition field (lower-cased)
	`word`		actual words (excluding cross-references) in definition (lower-cased)
	`xref`		'explicit' x-refs in definition (after '—see' etc.) (upper-cased)
`pron`			
	`nsylls`		number of syllables
	`s1`, `s2`, `s3`, etc.		where *si* is the *i*th syllable
		`stress`	stress for this syllable
		`onset`	phonemes at the syllable onset
		`peak`	phonemes at the syllable peak
		`coda`	phonemes at the syllable coda

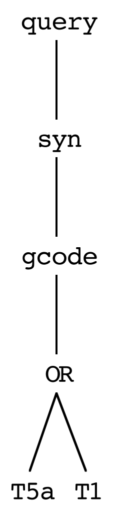
Given this hierarchy, we can now give the interpretation of the query which appears above:that the grammar code is `T5a`, there are 2 syllables, and the onset of the first syllable is the single phoneme `p`.

The basic values for attributes are usually atomic tokens (e.g. numbers, dictionary codes, or words) as in the example. More complex types of value may be made out of these basic values, however, in order to express conjunctive and disjunctive queries. A conjunction of values (where this meaningful) is indicated by a sequence of the individual values; for example

```
[[syn
   [gcode T5a T1]]]
```

means the co-occurrence of the two grammar codes, `T5a` and `T1`. (An entry which satisfies this query may, however, contain other grammar codes as well as these). A disjunction of values is indicated in the textual form of query by the values being enclosed in parentheses, prefixed by `OR`; in the graphical form a disjunction is indicated by the insertion of an node labeled with `OR`, e.g.

```
[[syn
    [gcode (OR T5a T1)]]]
```



Negation of a value is indicated by `NOT`. A conjunction of values for an attribute is meaningful only when the corresponding field in the dictionary may contain more than one value. Most fields are of this type in LDOCE, the main exceptions being the pronunciation fields. A disjunction of values, though, is meaningful for all fields.

2.3 The Constr and Headword Attributes

Independently of any dictionary, two top level attributes, `headword` and `constr` (short for constraint) are always available. The `headword` attribute may have as value a single word (possibly containing wildcards), or a negation or disjunction of words. Only entries which have a word in their headword field which matches the value of the `headword` attribute will be retrieved.

The attribute `constr` provides a way of expressing queries which cannot be formulated as a simple set of attributes and associated values. This attribute can have one or more (conjunctive) values, each either a dictionary index specification or a call to Lisp. `OR` and `NOT` can be used to modify these values. For example, the query

```
[[sem
   [subj BV]]
 [constr (OR "(@S Z A)" "(@W #\a #\l #\c #\o #\h #\o #\l #\i #\c)")]]
```

in LDOCE would return those entries which had the subject code `BV` (beverages) and which either had the subsidiary subject code `ZA` (alcoholic drinks) or contained the word-form `alcoholic` in their definition.

Lisp calls as `constr` attribute values are useful when used to test and compare the values of 'named' wildcards described in the next section. Lisp calls may also be used to accept or reject a complete entry, since during query lookup the Lisp form is called in a lexical environment containing the variables `entry` and `sense`, bound to the entry and sense number currently being considered. The call returning true indicates that the test has succeeded, and false that it has failed.

2.4 Attribute Values

The attribute values that appear in queries in most cases look exactly like their counterparts in the actual dictionary. The exceptions to this rule for LDOCE are the box code, subject code and the pronunciation fields. Pronunciation fields are different because a structure has been imposed on them which is not made explicit in the dictionary; subject codes are split into two pairs of letters to reflect the fact that the LDOCE coding system interprets them as two pairs; and box codes are different to make it easier to type them in and less prone to error. In the dictionary, a box code appears as a ten-slot vector, for example

```
--I-H----T
```

In a query, the slots containing dashes are ignored. Each remaining slot has its index (a number from 1 to 10) paired with the character in it; so the textual query corresponding to the example dictionary box code value above is

```
[[sem
   [box 3I 5H 10T]]]
```

(This query would also correspond to a dictionary box code with values for some of the remaining unspecified slots, e.g. `--I-H--X-T,` a similar code except for the addition of `X` in slot 8).

Attribute values may be wildcarded: `?` as part of a value matches any single sub-part, and `*` any sequence of sub-parts. Additionally, a 'named' wildcard starting with `?` followed by a number (e.g. `?3`) acts like `?` in that it matches a single sub-part, but it also 'captures' what it matches. Further occurrences of the same named wildcard in the query will match only what the first one matched[3] .

[3] Matching is non-deterministic, in that if a given named wildcard could match in more than one possible way, then all of these ways are retained for later attempts to match further occurrences of the wildcard.

2.5 Query Interpretation

The dictionary system by default computes the answers for a query in a sense-based (rather than an entry-based) fashion: that is, it returns just the senses which satisfy the query, not the whole entry (unless of course all the senses in the entry satisfy it). This behaviour can be changed, though, by resetting the value of a flag described later.

Before displaying the results of looking up a query, the system asks how many of the entries retrieved should be printed out. There are four alternatives: none, the first one (in alphabetic order), a random sample of one hundred, or all of the entries. In any case, by default, only the headword(s), homograph number and sense number are shown, not the complete entries.

3. The User Interface

Corresponding to the two representations for queries, textual and graphical, there are two modes of interaction with the dictionary system. In the first, the 'TTY interface' (section 3.1), queries are displayed textually, and commands are issued to the system by typing them. In the second, the 'graphical interface' (section 3.2), queries are represented graphically, and commands are issued by selecting them from menus.

3.1 Using the TTY Interface

The first thing that must be done before using the TTY interface is to load a dictionary, using the 'new' command described below.

In all cases, command and attribute names may be shortened to the minimum unambiguous sequence of characters. For example, when constructing an LDOCE query, `syn` may be typed as `sy`, since it is the only attribute name which starts with the characters 'sy'. It may not be shortened to `s`, though, since there would then be an ambiguity between it and `sem`, `s1` etc.

Add <path> <value>

Adds a value for the given path through the tree representing the current query. A path is a list of attribute names (enclosed in parentheses) starting from the root of the tree and ending at a terminal attribute. The path to the value of the onset of the first syllable in

```
[[pron
   [nsylls 2]
   [s1 [onset p]]]]
```

is

```
(pron s1 onset)
```

A path may be shortened by removing successive elements from the beginning of the list, as long as it remains unambiguous. In this case, `pron` may be removed (since `s1` is always dominated by `pron` in the hierarchy), but `s1` may not be removed as well because otherwise there would be nothing to say which syllable the onset was of. To make typing them easier, a path consisting of a single attribute name does not have to be enclosed in parentheses. For example, the following commands are equivalent

```
   add (pron s1 onset) p
   add (s1 onset) p
```

Or <path> <value>

Ors a new value into one already existing for the given path. More than one new value may be specified in this command.

NOt <path>

Negates all atomic values in the given path, i.e. if the value of a path is a conjunction or disjunction, the negation appears at the lowest level, inside both the conjunction and disjunction.

Delete <path>

Deletes the given path and its associated value from the current query.

Undo

Undoes the last change made to the current query. A further undo restores the query to what it was before the first undo.

View [<path>]

Displays the current query. The argument to this command is optional: if supplied, just the path specified is displayed, otherwise the whole query.

Lookup [<path>]

Looks up the current query in the dictionary. If a path argument is given, then the lookup is performed just on that portion of the query.

FIle [<path>]

Looks up the current query, like the lookup command, but puts the results in a file. The user is prompted for the name of the file.

STatistics [<path>]

Outputs statistics concerning what would happen if the current query were looked up in the dictionary, but does not perform the actual lookup.

Entry <word>

Finds the dictionary entry for the given word, and prints the definition.

Print

Prints the definition of the last word looked up, by either the entry or the lookup command.

Template

Constructs a new query from all the information in the definition of the last word looked up.

NEw

Loads a new dictionary (together with its index and menu files) into the system. This command must be issued before a dictionary can be queried or entries looked up in it. Any number of dictionaries can be loaded in single LDB session: they will all be available for querying, but in the TTY interface only one query can be under construction at any one time. The 'switch' command specifies which (already loaded) dictionary is to be queried.

SWitch

Allows the user to switch dictionaries if more than one is available. All subsequent queries and requests for lookup of entries will apply to the dictionary that has been switched to.

Clear

Clears the current query.

Read

Reads a query in from a disc file. The user is prompted for the name of the file. If there is an already existing query, the query in the file is unified with it.

Write

Writes the current query to a disc file, c.f. the file command.

SEt

Sets the value of a dictionary system flag. There are five such flags: 'typeset entries', 'explain tests', 'sense-based retrieval', 'print query result function' and 'derived query result function' with initial values ON, OFF, ON, OFF and OFF respectively.

'Typeset entries' controls whether entries are printed in the form they appear in the dictionary source (OFF), or printed more readably (ON)

'Explain tests', when ON, causes tracing information to be output by the system about which entries fail which tests on query lookup.

'Sense-based retrieval' controls whether query lookup takes notice of sense divisions in entries. When OFF, if one sense in an entry satisfies a constraint, then the whole entry is taken as satisfying it.

'Print query result function' may be OFF, or have as value the name of a Lisp function. If the value is a function, the function is called on each entry that results from query lookup; the function is expected to print part or all of the entry. The function is passed three arguments: the actual entry, a list of the sense numbers in the entry for which the query succeeds, and the Lisp stream to which the function is meant to print.

'Derived query result function' may be OFF, or have as value the name of a Lisp function. If the value is a function, it is called in exactly the same way as a 'print query result function', the difference being that the entries it prints are incorporated into a new derived dictionary (section 3.3). The 'file' command should be used to look up the query so that the derived dictionary is output to a file, rather than to the screen.

FLags

Displays the values of the dictionary system flags.

Help

Displays a page of help information about attribute names and dictionary system commands in the current dictionary.

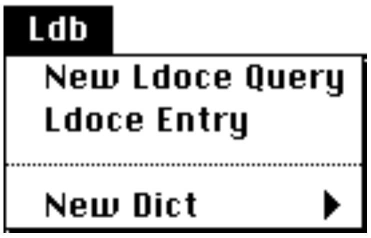
Quit

Quits from the dictionary system.

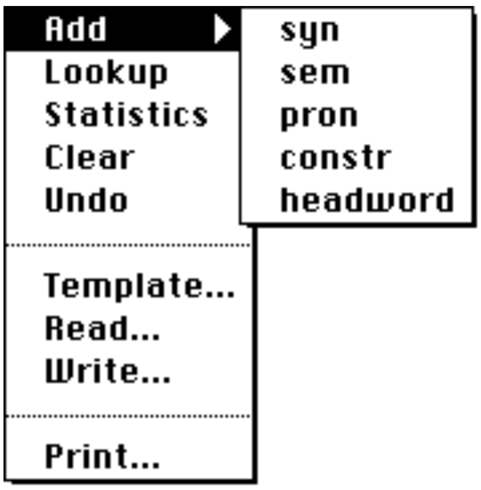
3.2 Using the Graphical Interface

The graphical interface provides an alternative, and in general, more convenient mode of working than the TTY interface (although there are facilities such as ‘set’, ‘help’ and ‘quit’ which are available only in the TTY interface).

Before using the interface, a dictionary has to be loaded: in the pull-down menu ‘Ldb’, the item ‘New Dict’ pops up a further menu asking for the name of the dictionary to be loaded. Selecting one of these names causes the dictionary of that name to be loaded. When this is completed, two new items appear on the Ldb menu: one for making a new query, and another for directly looking up an entry in that dictionary, given its headword. The ‘New Dict’ command can be issued repeatedly to make several dictionaries available in a single LDB session. For example, after selecting ‘Ldoce’ as the dictionary to be loaded, the Ldb menu will appear as follows:



Selecting ‘New Ldoce Query’ prompts for a place on the screen to put a window (by drawing a ‘ghost’ window which moves with the mouse, and waiting for the mouse button to be clicked to indicate that the user is satisfied with the position of the window). This window can contain a query. The query initially contains just a node called ‘query’. Clicking on this node pops up a menu with several items on it:



The next section describes what the commands in this menu do.

3.2.1 Graphical Interface Query Node Menu Commands

Add

Adds an attribute (selected from a further pop-up menu) to the query.

Lookup

Looks up the query in the dictionary. The user is prompted to position a new window, and lookup statistics and results are put in that window. The results window is associated with the query window, and output from subsequent lookups and calls for statistics will automatically go into the window. Double-clicking on a word in the results window causes a new window to be popped up containing the definition of that word.

Statistics

Outputs statistics concerning what would happen if the query were looked up in the dictionary, but does not perform the actual lookup. Prompts for a new window if necessary, as above.

Clear

Clears the query.

Undo

Undoes the last change to the query.

Template

Asks for a word in the dictionary, looks it up, extracts all the information in it that is indexed and makes a query out of this information. The query is the most specific one that would return this word as a result.

Read, Write

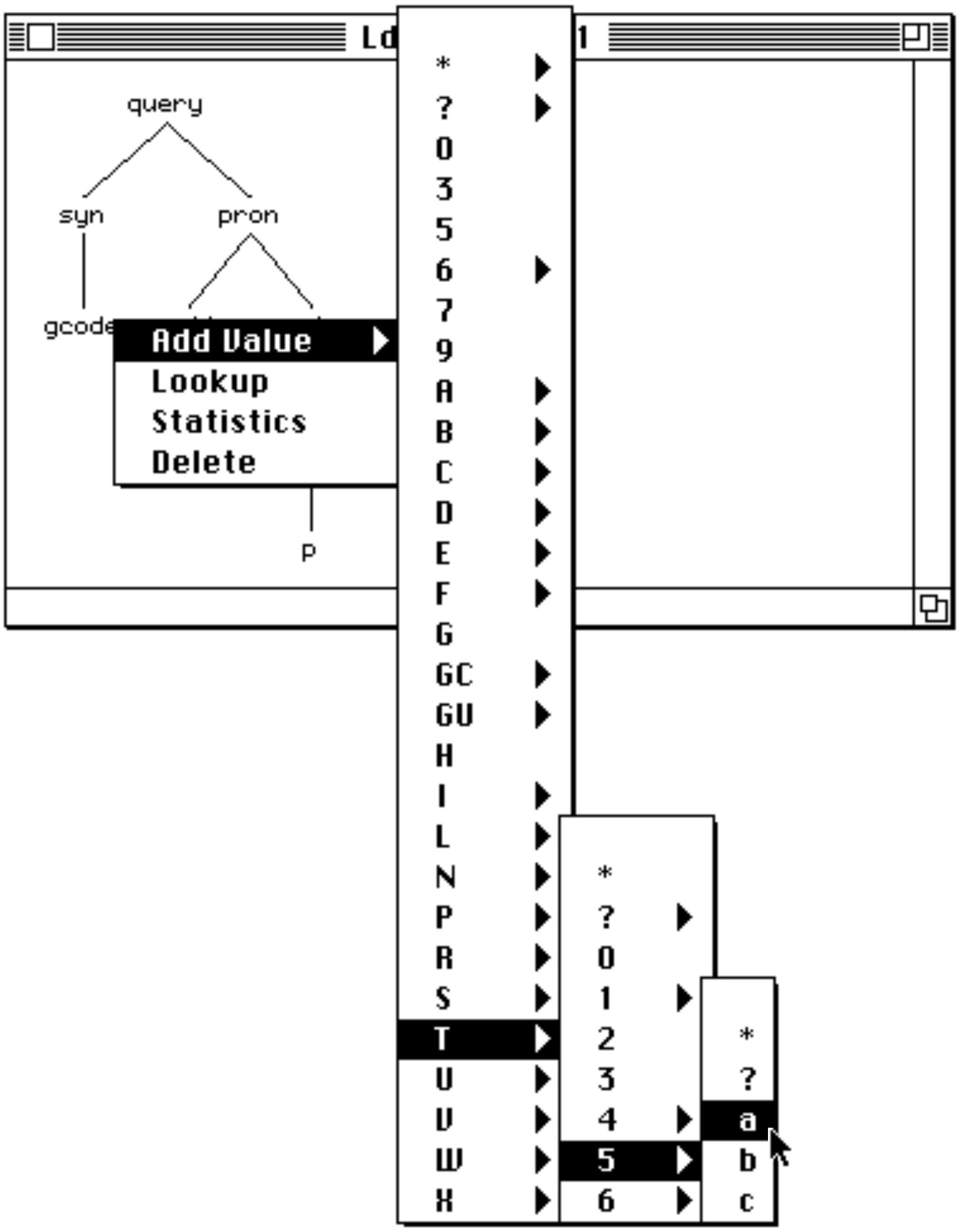
‘Read’ asks for the name of a file, which is expected to contain a query in textual form. The query in the file is then unified with the query currently in the window. ‘Write’ outputs the query to a file in textual form.

Print

Prints the query on the currently-selected system printer.

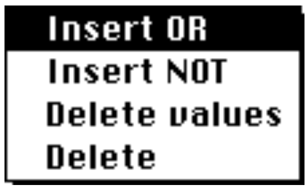
3.2.2 Graphical Interface Attribute Node Menu Commands

Attribute nodes (i.e. nodes below the top ‘query’ node) pop up similar menus when clicked, but with fewer items: just ‘add’, ‘lookup’, ‘statistics’ and ‘delete’. ‘Lookup’ and ‘statistics’ are similar to the commands on the query node, except that they apply only to the part of the query below the selected node. ‘Add’ pops up a menu of the possible values for that attribute. A value which has a sub-menu may be selected by selecting the top (blank) item in the sub-menu. ‘Delete’ deletes the node (along with all those which are below it). These menus are illustrated below:



3.2.3 Graphical Interface Value Node Menu Commands

Nodes representing the value of an attribute, when clicked, pop up the menu:



‘Insert OR’ inserts an OR node between the value node and the attribute node just above it. A disjunction of several values can then be formed by adding further values below the OR node. ‘Insert NOT’ similarly inserts a NOT node above a value node. ‘Delete Values’ deletes the value node and all its siblings; ‘Delete’ deletes just the node itself.

3.2.4 Graphical Interface Shortcuts

On the Macintosh, the generic ‘Cut’, ‘Copy’ and ‘Paste’ clipboard commands can be used in query windows: they apply to the textual form of the whole query. They allow a complete query to be copied out of a graphical window and pasted into text window, or indeed into another graphical window (forming a new query: the result of unifying the query in the clipboard with the query originally in the window).

A quick way of deleting a node is to click on it with the Command Key (‰ on the keyboard) depressed. Holding the Option Key down while clicking on a terminal attribute node pops up the menu of possible values, bypassing the menu containing the lookup and delete commands. If the Option Key is depressed when double-clicking on a word in the results window (to bring up its entry), a dialog box pops up asking for an alternative dictionary in which to look up the word.

If the screen is getting cluttered with windows containing entries for words, then a quick way to close them all is to select one of them and invoke ‘Close All’ from the pull-down ‘File’ menu.

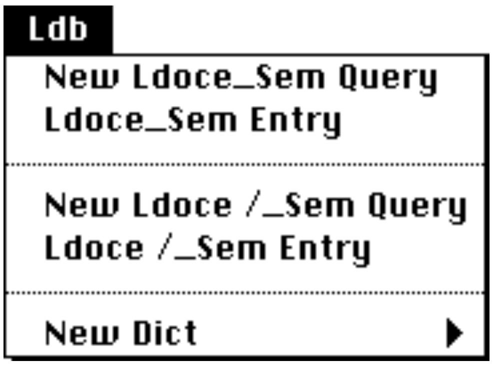
3.3 Derived Dictionaries

The LDB system (in circumstances detailed below) allows the user to apply a query to two or more dictionaries simultaneously. The facility depends on the concept of a derived dictionary.

A derived dictionary, as the name suggests, is a dictionary which has been created using another dictionary (the ‘source’ dictionary) as a starting point. Each entry in the derived dictionary will have been computed from the corresponding entry in the source dictionary (i.e. the first entry in derived dictionary must correspond to the first entry in the source, and so on; similarly with the senses in each entry). A derived dictionary entry will consist of the source entry headword and typically an elaboration of a subset of the information in the source entry, for example a parsed representation of the text in its definition fields.

To query a source – derived dictionary pair simultaneously, the source dictionary should be loaded, followed by the derived dictionary. The LDB system shows queries to the dictionary pair as being to a single, notional, dictionary called *<source name>/<derived name>*. So when using the TTY interface, you should ‘switch’ to the dictionary called this before starting to construct a query. In the graphical interface, the LDB system adds to the Ldb menu an appropriate ‘New Query’ command for this notional dictionary when the derived dictionary is loaded.

As an example, if a dictionary called ‘Ldoce_sem’ has been derived from ‘Ldoce’, then loading ‘Ldoce’ followed by ‘Ldoce_sem’ into an LDB session will result in the Ldb menu appearing as



Selecting ‘New Ldoce /_Sem Query’ from this menu will open a window in which a query can be built which will be applied simultaneously to both the ‘Ldoce’ and ‘Ldoce_sem’ dictionaries.

4. Dictionary-Specific Commands

4.1 LDOCE

4.1.1 Graphical and TTY Interface Commands

All the commands in this section are available both from the TTY interface and from the command menus of appropriate nodes in the graphical interface.

Rotate Order

Rotates the words in an order constraint. For example, rotating the value of the order attribute in the query `[[order of a type]]` would result in `[[order a type of]]`.

Ordering Constr

Automatically constructs a Lisp call as a `constr` attribute value specifying the maximum distance (in units of words in the text of a dictionary definition) between one word and the next given in the `order` part of a query. The command prompts for which words in the order part of the query are to be constrained, and the maximum distance by which they may be separated.

Broaden <path>

Works on the pronunciation part of a query in LDOCE, changing the current value of the terminal attributes onset, peak and coda into broad class. For example, the command

```
broaden s1
```

applied to the query

```
[[pron
   [s1 [onset k w]]]]
```

```
would change it into
```

```
[[pron
   [s1 [onset (OR b d g k p t) (OR r l w j)]]]]
```

The whole value of the pronunciation attribute, just one syllable, or a single onset, peak or coda may be broadened with a single command, by specifying the path in the command as `pron,s1` etc., or `onset` etc. respectively.

4.1.2 TTY Interface-Only Commands

eXplain/eXtract <attribute>

Extracts the value of the given attribute from the definition of the last word looked up, and prints it out with some explanation text. In LDOCE the attribute should be one of `gcode`, `box`, `subject`, `definition`, or `word`.

Z <attribute> <value>

Given the name of an attribute (in LDOCE, one of `gcode` , `box` or `subject`) and a value for it, the command prints an explanation of what the value means, e.g. for `z gcode T1`, the command would print

```
T1:
   transitive verb
      followed by direct and maybe indirect object
```

Acknowledgements

The LDB adopts several of the DBMS implementation techniques employed by Hiyan Alshawi, Branimir Boguraev and David Carter in the original LDOCE MRD access system. Some of the code used to mount LDOCE is a direct translation of InterLisp code written by David Carter. Thanks to Ted Briscoe, Ann Copestake and Antonio Sanfilippo, other members of the ACQUILEX project, and to Tara Davis and James McQueen for reporting bugs and suggesting improvements to the LDB.

References

Alshawi, H., B. Boguraev & D. Carter (1988) 'Placing the dictionary on-line' in B. Boguraev & E. Briscoe (eds.) *Computational Lexicography for Natural Language Processing*, Longman, London, pp. 41–63.

Boguraev, B. & E. Briscoe (eds.) (1988) *Computational lexicography for natural language processing*, Longman, London.

Boguraev, B., J. Carroll, E. Briscoe & A. Copestake (1992) "Database models for computational lexicography." In *Proceedings of EURALEX-90*, Biblograf, Spain, 59–78.

Procter, P. (ed.) (1978) *Longman dictionary of contemporary English*, Longman, London.