LDB/LKB Integration

German Rigau
Horacio Rodríguez
Jordi Turmo

Report LSI–94–32–R

LDB/LKB Integration

German Rigau, Horacio Rodríguez, Jordi Turmo

Departament de Llenguatges i Sistemes Informàtics
Universitat Politècnica de Catalunya

Barcelona, July 1994

Abstract

The Lexical Data Base (LDB) and the Lexical Knowledge Base (LKB) have played, as software systems, a core role within Acquilex I Esprit 3030 project. Main limitations were derived from the fact that the LDB and the LKB are two different systems, designed to fulfil different functional specifications, in order to perform different operations. LDB software provides, basically, database-like access to lexical information. LKB software manages a lexical Knowledge Representation based on typed feature structures (FSs) allowing operations on lexical entries such as inheritance, unification, generalisation, lexical rules, etc. The central aim of the LDB/LKB integration software is to provide new tool for loading intermediate and relative stable versions of lexicons developed in the LKB into the LDB, allowing, in this way, flexible access/search to entries based on any aspect of their contents.

Content

Appendix

1 Introduction

This document is a description of the final release of the LDB/LKB integration software that constitutes the Work Part 4.1 within Acquilex II. The document covers two different aspects: a technical description of the software and a user oriented description. Most of the document content is devoted to the former. A detailed user manual is included in appendix 1 so that it can be used as a separate document.

The Lexical Data Base (LDB) and the Lexical Knowledge Base (LKB) have played, as software systems, a core role within Acquilex I. In the "Final Evaluation of LDB/LKB" (Acquilex #14 Deliverable, [Acquilex,92]) some limitations were pointed out and solutions suggested. These were included as aims for Acquilex II, in WP4.1.

Main limitations were derived from the fact that the LDB and the LKB are two different systems, designed to fulfil different functional specifications, in order to perform different operations.

LDB software provides, basically, database-like access to lexical information. LKB software manages a lexical Knowledge Representation based on typed feature structures (FSs) allowing operations on lexical entries such as inheritance, unification, generalisation, lexical rules, etc.

The central aim of WP4.1, as pointed out in the Technical Annex, is to provide tools for loading intermediate and relative stable versions of lexicons developed in the LKB into the LDB, allowing, in this way, flexible access/search to entries based on any aspect of their contents. The work within this WP has been carried out mainly by UPC with some contribution from CCL.

Initial ideas of LDB/LKB integration were discussed in a preparatory meeting held in Cambridge in December, 1992. These ideas were further discussed in a second meeting held in Barcelona in February, 1993. A first prototype of the system was distributed by November, 1993. Valuable comments on an intermediate release of the software were provide by publishers partners in a meeting held in Barcelona (in April, 1994). The present release gathers many of the suggestions received and can be considered a relatively stable final release. Nevertheless, some improvements are envisaged for adapting the software to the needs of Work Part 4.2 in the third year of the project.

The system has been developed taking into account a central guideline: The LKB lexicons [Briscoe et al. 91] or [Copestake, 92] should be expressed, loaded and stored as any other dictionary in such a way that the LDB software [Carroll 90] could be used without modifications.

The organisation of this document is as follows: After this short introduction, section 2 summarises the specification of the System describing the problems (2.1) and the proposed solutions (2.2). Section 3 is devoted to the implementation of the LDB/LKB System. Section 3.1 presents an overall description while section 3.2 and 3.3 account respectively for the main operations: indexing and recovering. As pointed out above the user manual is included as an appendix. Section 5 states further developments linking this workpart to workpart 4.2.

2 About LDB/LKB Integration

The central idea on LDB/LKB merging is to express the LKB lexicons as any other collection of lispified dictionary entries. The structure of such entries should be described and the appropriate functions tailored in order to load the lexicon into the corresponding LDB structures. In such way all the capabilities of LDB software could be used without restrictions. The original LKB entries should be susceptible of being reconstructed from their LDB representation. In fact the display capabilities of LDB have been connected with LKB display functions and the appearance of displayed FS is the same now inside LDB or LKB. This allows us to replace the LKB's lexical reading and access mechanism by the LDB functions, which gets round the current problem that reading in LKB lexicons is very slow, showing a considerable drop in performance when faced with

real-size lexicons, and in the long term will allow for efficient access to indexed.

2.1 Problems involved

The central idea of loading lexicon files as other dictionaries source files into the LDB environment looks rather straightforward but various problems arise when facing it in detail:

1 How to describe sources.

In the LDB, what we need for loading, and then querying, a dictionary is a lispified version of the MRD, a description of the dictionary entries and of the intended indexes to be built. These descriptions are given by hand, tailoring a couple of specification files and providing appropriate access functions for the different fields we need to extract.

A LDB lexicon file is not, obviously, a dictionary. Obtaining from it a lispified version is quite straightforward. Building descriptions of entries cannot, however, be accomplished in a manual way because of the great number of combinations of types and features that can appear in a lexical entry. So, we need a way to automatically generate specification files. For this task, the LKB type structure provides enough information to define a generation procedure.

2 What to index

A LKB entry consists of a headword and a list of <path, value> pairs (and <path, path> pairs if we want to deal with reentrancy) that must be loaded and indexed. Some of the information is local to the entry, while other has been inferred from psort and/or type structures by different inheritance mechanisms. Indexing unexpanded (local) LKB entries is not enough for our purposes (we want to have access in LDB to the same information coded and accessible in the LKB).

Indexing fully expanded lexical entries is impractical as well as highly redundant. Therefore, we have indexed what we call a "canonical form" of an entry, i.e. the minimum amount of information from where the system, using the Type System, is able to generate the fully expanded lexical entry. In this way the user can query the LDB for any information contained in the fully expanded lexical entry. This solution can be seen as a reasonable tradeoff in terms of space/time efficiency: no much more information, apart from the local one, is stored explicitly within the LDB. The price we must pay for this is a higher complexity to solve complex user queries.

3 How to access indexed information

A single (user) query to the LDB is converted by the system into a disjunction of (internal) sub-queries for taking into account information not explicitly indexed. A subquery is expressed as a conjunction of simple queries involving the elements (types and features) of the path. Thus, the final query involves access to a large amount of indexes. Some of these accesses can be redundant and could be saved. Several mechanisms have been provided to make more efficient the access process optimising the number of (internal) queries that must be made in order to answer an user question.

4 How to display information

LDB format (text based) is not, of course, adequate for displaying and handling LKB (feature based) structures. The system has been designed to provide the same display capabilities available in the LKB.

2.2 Fundamentals

Simply indexing on the unexpanded LKB entries in the LDB will not allow for many queries which we would wish to support.

The basic query can be expressed as follows:

< path > = type

where the value of the path in the expanded lexical entry is exactly equal to the given type (variants of this query, and other options, are considered in more detail below). We must, thus, index such paths in order to recover all the unifiable lexical entries efficiently. If only the unexpanded lexical entries are stored, supporting this sort of query will be impractical, because psort inheritance etc. would have to be carried out before the query could be evaluated.

However, storing indexes of fully expanded lexical entries is also impractical, because of the number of indexes involved. The approach we have taken is to index on the "canonical form" of the expanded lexical entry. The notion of "canonical form"[1] is illustrated below:

Given the following well-typed feature structure:

```
[ type1
 F  [ type2
      G = type3
      H = type4
      I = type5 ]
 J = type6 ]
```

and the constraint on type1:

```
[ type1
 F  [ type7
      G = top ]
 I = top ]
```

and on type2:

```
[ type2
 G = type3
 H = type4
 I = top ]
```

it is clear that the path notation for the FS could be redundant, given that we know that the FS is

[1] taken from Ann Copestake.

well typed.

i.e. the full path description would be:

<> = type1
< F > = type2
< F : G > = type3
< F : H > = type4
< F : I > = type5
< J > = type6

but all that we actually need to know is:

<> = type1
< F > = type2
< F : I > = type5
< J > = type6

Applying type inference will expand of this structure to the original one.

The information contained in the canonical representation of lexical entries is the only information to be loaded and indexed into the LDB. The idea is to include the rest of the information within the Type Structure (not in Psorts) in order to be used at query time by the system.

When we query on the canonical description, expansion of the query will be necessary to allow for type inference. For example if we want the result of the query:

< F : I > = type5

we can evaluate it against the canonical form directly, but in order to check

< F : G > = type3

the system has to convert it into

< F > = type2 OR
< F : G > = type3

(assuming that type7 and type2 are the only types with G as an appropriate feature). Note that

< F > = type2 implies < F : G > = type3

but

< F : G > = type3 does not imply < F > = type2

Not only the information consisting of values assigned to features have to be stored for further querying but reentrancy information too. In such a way the complete lexical entry can be reconstructed from its LDB representation. Issues concerning reentrancy will be explained with detail in section 3.2.

Another possibility of querying, allowed by the LDB/LKB system is to check not only the current value of a path but also to return the entries where the value of some path is subsumed by a given value.

3 Implementation of LDB/LKB System

The main goal in designing and building the LDB/LKB software has been to provide with the system all the currently available capabilities within the LDB framework without loosing the specific properties of LKB software. For instance, the way we have chosen for displaying LDB/LKB structures is the same that was used in LKB environment. The characteristics of elements to be indexed (FS instead of dictionary entries) impose, too, some additional restrictions on the way of communication between user and system. We tried, at the beginning, to maintain the same user interface than in the LDB system but when faced with real size lexicons we realised that the way of accessing was tedious and cumbersome so we decided to include in the system additional interface facilities related to the kind of objects we wanted to retrieve.

The LDB/LKB merging software has to face two main tasks:

- Building the database structures owning the necessary information for allowing a further reconstruction of the whole FS in a fully automatic way.
- Consulting the database. This task imposed several requirements:
  - All the existing capabilities of LKB software should hold in spite of the way of representing the LKB structures, i.e. the operations the external user is able to do, would not depend in any way on the representational issues.
  - The LDB/LKB system would maintain all the capabilities of LDB software.
  - Accessing mechanisms strongly adapted to FSs characteristics should be included.

Building automatically the database structures in the LDB environment from source files (till now MRD versions of conventional dictionaries) implies three different tasks:

- Building the c-spec and d-spec files in order to allow for the loading of the lexicons.
- Obtaining a lispified version of the source file, usually through a parse process of this file guided by a appropriate grammar.
- Perform the loading, creation of indexes and menus etc...

In the case of LDB/LKB system the source file consists of a set of lexical entries loaded into the LKB. At a first glance the process is quite straightforward but several difficulties arise.

- As the way of displaying and querying the information are different, and, specially in the last, more complex (querying by now could imply complex manipulation of pieces that have to be assembled prior to the final performance of the query) the graphical interface system of the current LDB has been substituted by another mechanism, more friendly and closer to LKB interface.

- Due to the features discussed in section 2.2 (subsumtion, queries to partially specified structures, etc.) a new querying mechanism has been implemented acting as an external layer over LDB.

Of course any change in the type structure or in the content of the lexicons will imply redoing the process (there is up to now no possibility of incremental loading and indexing of LDB).

3.1 System Organisation

As shown in Fig. 1, LDB/LKB system makes use of both the LDB and the LKB software, as well as its specific software. Looking at data, the information is distributed among three main blocks:

- The type system, managed by means of the LKB software (slightly enriched in some aspects but basically unchanged).

- The database content, composed by a lexicon containing the canonical forms of the lexical entries, and the corresponding index files. The LDB software is in charge of managing these structures.

- The identifier structure, needed to expand the user query into a query able to be launched.

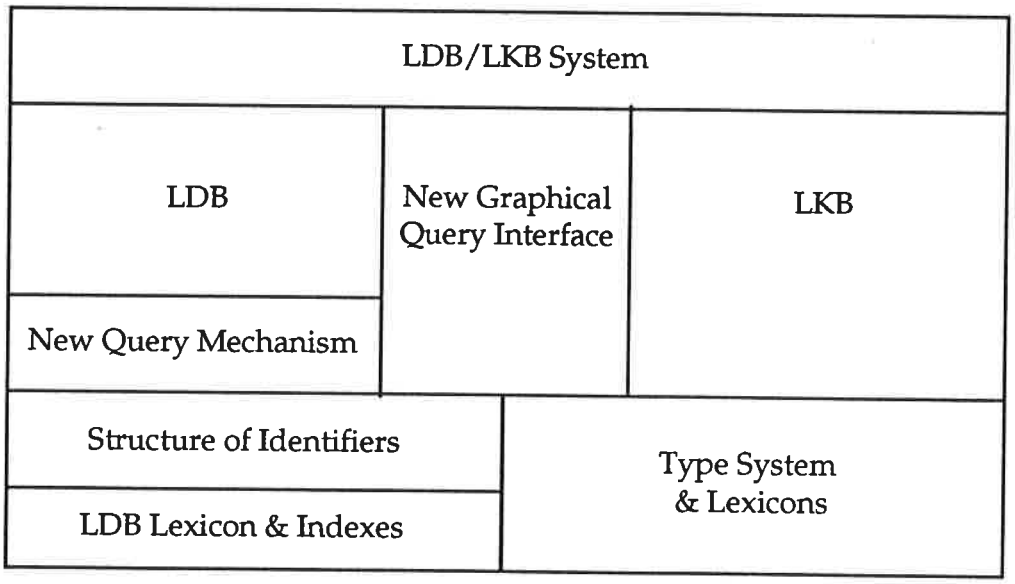


Fig. 1: System Organisation.

As we stated above, the LDB/LKB System not only has the functionalities of both the LDB and the

LKB software but also offers new features. A set of minor changes for allowing all the functionality of the LKB from the lexical entries stored in a LDB-like format have been done.

The LDB/LKB System performs the construction of the LDB lexicons and indexes from the lexical entries loaded in the LKB in a fully automatic way consulting the Type System.

A new graphical query interface has been built for allowing a LKB-like user friendly interaction with the LDB/LKB System. Thus, the query construction with the manipulation of the chuncks of FSs among the Type System (clicking on it), the lexical entries and the query windows have been carried out.

A new query mechanism has been built over the existing one allowing to the LDB/LKB System to perform the least possible number of queries over the indexes and the LDB lexicon, consulting the Type System and the Structure of Identifiers.

3.2 Representing and indexing LKB structures within LDB

If we want to represent the lexical entries loaded into the LKB in terms of the LDB System, each of them must be expanded. This process is necessary in order to know the explicit and implicit information (that appears in other lexical entries or in the Type System by means of LRL operations: inheritance, default unification, etc.). Following the Type System and the lexicon files shown in appendix 2 and 3, a lexical entry may be written in the LKB lexicon as:

horacio B_I_1
computer-scientist-man
< body : sex > = male
< body : age > = high
< body : size > = grade
< body : sport > = low
< mind : work > = grade
< mind : formation > = computer-scientist
< mind : category > = 3.

The full expanded horacio_B_I_1 lexical entry will contain the following information:

horacio B_I_1
computer-scientist-man
< body : sex > = male
< body : age > = high
< body : size > = grade
< body : sport > = low
< body : power > = grade
< mind : work > = grade
< mind : formation > = computer-scientist
< mind : category > = 3
< mind : recursive > = recursive.

where some of the path-values are redundant. The corresponding non-redundant lexical entry (the rest of the information can be found in the Type System) is:

horacio B_I_1
computer-scientist-man
< body : age > = high
< body : sport > = low
< mind : category > = 3.

The canonical lexical entry of horacio_B_I_1 in the LDB readable format will be:

```
((HORACIO_B_I)
(SN 1)
(= () computer-scientist-man)
(= (( computer-scientist-man mind )( computer-scientist-mind category )) 3)
(= (( computer-scientist-man body )( man-body sport )) LOW)
(= (( computer-scientist-man body )( man-body age )) HIGH)
(= (( computer-scientist-man sense-id )( sense-id language )) SPANISH)
(= (( computer-scientist-man sense-id )( sense-id fs-id )) "HORACIO_B_I_1")
(= (( computer-scientist-man orth )) "HORACIO")
)
```

where the redundant paths (explicitly represented in the Type System) have been removed. Thus, for each LDB lexical entry the system will index only the information that does not appear in the Type System.

For expanding each lexical entry, all the information present in the Type System is taken into account. Then, any reentrancy information from the Type System will be placed into the LDB/LKB lexical entry reconstructed. The reentrancy information present in a LKB lexical entry is stored in the LDB lexical entry but it is not indexed. This information is not relevant for querying but is required for other LKB operations such as lexical rules, tlinks, etc.

The LDB lexical entries created from the LKB are indexed by four different kinds of indexes: @type, @feature, @menu and @string. The @type index deals with all the different types (except the values) of each lexical entry. For the horacio_B_I_1 lexical entry four types will be indexed:

```
(@type computer-scientist-man-24)
(@type computer-scientist-mind-34)
(@type man-body-45)
(@type sense-id-52)
```

The numbers that appear at the end of the types are identifiers. We need them to identify an item (type, feature, value) that could be addressed by different paths. These numbers are stored in the Identifiers Structure mentioned in section 3.1. The @feature index deals with all the different features of each lexical entry. For the example, nine features will be indexed:

```
(@feature mind-25)
(@feature body-35)
(@feature sense-id-53)
(@feature orth-38)
(@feature sport-67)
(@feature category-76)
(@feature age-62)
(@feature language-71)
(@feature fs-id-52)
```

The @menu index deals with all the different non-string values of each lexical entry. For our lexical entry example four menu values will be indexed:

```
(@menu 3-45)
(@menu LOW-65)
(@menu HIGH-34)
(@menu SPANISH-73)
```

The @string index deals with all the different string values of each lexical entry. Just two string values will be indexed for our example entry:

(@string #\H #\O #\R #\A #\C #\I #\O #_ #\B #_ #\I #_ #\1 #\- #\1 #\2)
(@string #\H #\O #\R #\A #\C #\I #\O #\- #\1 #\4)

3.3 Recovering LKB structures from LDB

The main problem with the query process deals with the subsumtion relationship. If the user formulates a query to be answered without subsumtion, he/she is asking for lexical entries satisfying a restriction described by means of a equation involving a path (or a set of combinations of paths with and/or logical operators) that can be found directly by the indexes. We will next illustrate the recovering process with a set of examples:

Consider the following query:

Query : (computer-scientist-man mind computer-scientist-mind work high)

We are asking for every lexical entry of type *computer-scientist-man*, owning a *mind* feature filled with a *computer-scientist-mind* type owning a *work* feature, filled in turn with the value *high*. The answer of the system will be in this case:

Looking up on these constraints:
(@MENU HIGH-57) -> 1 items
There are actually 1 results

Total of 1 results: TONI B_I (1)

This means that the only internal query that has been activated looked for the entry indexed by the value *HIGH-57* of the index *@MENU*.

If we perform a query with subsumtion, we are asking for all the combinations of paths (meaningful as regards the type system) more specific than the query path. This set of paths can be found directly by the indexes. For example, if our query is:

Query : (person mind computer-scientist-mind work high)

really we are asking for:

Query : (person mind computer-scientist-mind work high)
Query : (man mind computer-scientist-mind work high)
Query : (woman mind computer-scientist-mind work high)
Query : (computer-scientist-person mind computer-scientist-mind work high)
Query : (computer-scientist-man mind computer-scientist-mind work high)
Query : (computer-scientist-woman mind computer-scientist-mind work high)

because the types *man, woman, computer-scientist-person, computer-scientist-man and computer-scientist-woman* are subsumed by the type *person*, the only explicitely mentioned in the query not corresponding to a terminal type. The program answer will be in this case:

Looking up on these constraints:

(@MENU HIGH-17) -> 0 items
Estimated pointer+entry reading time: 0.0+0.0=0.0 seconds (0 results)
Nothing to look up

Looking up on these constraints:
(@MENU HIGH-86) -> 0 items
Estimated pointer+entry reading time: 0.0+0.0=0.0 seconds (0 results)
Nothing to look up

Looking up on these constraints:
(@MENU HIGH-67) -> 0 items
Estimated pointer+entry reading time: 0.0+0.0=0.0 seconds (0 results)
Nothing to look up

Looking up on these constraints:
(@MENU HIGH-76) -> 0 items
Estimated pointer+entry reading time: 0.0+0.0=0.0 seconds (0 results)
Nothing to look up

Looking up on these constraints:
(@MENU HIGH-53) -> 0 items
Estimated pointer+entry reading time: 0.0+0.0=0.0 seconds (0 results)
Nothing to look up

Looking up on these constraints:
(@MENU HIGH-57) -> 1 items
Estimated pointer+entry reading time: 0.0+0.0=0.0 seconds (1 results)
There are actually 1 results

Total of 1 results: TONI B_I (1)

Note that the following queries are not generated:

Query : (linguist-person mind computer-scientist-mind work high)
Query : (linguist-man mind computer-scientist-mind work high)
Query : (linguist-man-woman mind computer-scientist-mind work high)

because these are not well-formed queries following the Type System described in appendix 2.

If we are asking for a query like:

Query : (computer-scientist-woman body woman-body sex female)

no entries will be directly retrieved because this path was neither represented nor indexed in the lexicon. All the women have the sex as female and the body as woman-body. This information is constrained by the Type System and is not presented in the canonical lexical entries. The new query mechanism computes a new query consulting the Type System:

Query : (computer-scientist-woman)

Giving us the following answer:

Looking up on these constraints:
(@TYPE COMPUTER-SCIENTIST-WOMAN-1) -> 2 items
Estimated pointer+entry reading time: 0.0+0.0=0.0 seconds (2 results)

There are actually 2 results

Total of 2 results: ALICIA_B_I (1), ANN_B_I (1)

If we launch a query like:

Query : (and
(computer-scientist-woman body woman-body size low)
(computer-scientist-woman body woman-body sex female))

the results will be:

Looking up on these constraints:
(@TYPE COMPUTER-SCIENTIST-WOMAN-1) -> 2 items
Estimated pointer+entry reading time: 0.0+0.0=0.0 seconds (2 results)
There are actually 2 results

Looking up on these constraints:
(@MENU LOW-51) -> 1 items
Estimated pointer+entry reading time: 0.0+0.0=0.0 seconds (1 results)
There are actually 1 results

Total of 1 results: ALICIA_B_I (1)

Not only total query descriptions are allowed in the LDB/LKB System. Partial queries over the lexicon can be performed too. We can launch partial descriptions of the query path like:

Query : (woman-body size low)

and the result will be:

Looking up on these constraints:
(@MENU LOW-51) -> 1 items
Estimated pointer+entry reading time: 0.0+0.0=0.0 seconds (1 results)
There are actually 1 results

Total of 1 results: ALICIA_B_I (1)

4 Using LDB/LKB

Although using LDB/LKB environment does not look different from previous software systems delivered within Acquilex project, a new user manual has been written to cover all new facilities. In order to facilitate its use apart from this document, this user manual is included as Appendix 1.

5 Further development

Although this deliverable is the final one within WP 4.1, a natural continuation of our task will be carried out inside WP 4.2. These two work parts are closely related. According to the Technical Annex, technical and functional capabilities should be included in WP 4.1 while high level

interaction, including ergonomic and user-friendly aspects (facilities for manipulating lexical information, appropriate and more comprehensive presentation of data, ...) would be included in WP 4.2.

A joint meeting was held in Barcelona last April in order to discuss and decide the facilities to be included in our final interface. Some experiments are in course, involving mainly the publishers groups, in order to check the feasibility of the LKB as a tool to help the lexicographer in the task of building dictionaries. As a result of these experiments a new meeting is foreseen by Fall'94 to fix the interface requirements and as a result we will specify, design and build the desired interface.

6 References

[Acquilex,92] Acquilex: *Final evaluation of LDB/LKB System.* 30th month deliverable Num. 14, Barcelona, June. 1992

[Briscoe et al. 91] Briscoe,T., Copestake, A. y de Paiva, V., *Functionality of LKB* in Proceedings of the ACQUILEX Workshop on Default Inheritance in the Lexicon, University of Cambridge Computer Laboratory, Technical Report nº 238. Esprit BRA-3030 Acquilex working paper nº040, 1991.

[Carroll 90] Carroll J. *Lexical Data Base System. User Manual.* Computer Laboratory, Cambridge University. October 1990.

[Copestake, 91] Copestake A., *LKB implementation outline-version 3*, Cambridge University, Computer Laboratory, ms, 1991.

[Copestake, 92] Copestake A., *The Representation of Lexical Semantic Information.* PhD thesis. Cognitive Science Research Papers 280. University of Sussex. September, 1992.

Appendix 1: LDB/LKB User manual

1.1 Installing the LDB/LKB merging system

To install the LDB/LKB merging System you must follow the next steps:

- copy the LDB/LKB folder into your LKB folder. This folder contains all the software needed for LDB/LKB performance. The content of the folder is presented in appendix 4. Some minor changes on both LDB and LKB software files are also reported in appendix 4.
- modify the lisp variables *ldb/lkb-source-dir*, *ldb/dict-dir*, *regular-exp-path-lexicons* and *path-lexicons* in the ldb/lkb-load.lsp file, placed in the LDB/LKB folder, by writing the path where you has put the LDB/LKB files and where the System will generate automatically the new LDB/LKB lexicons.

1.2 Accessing the LDB/LKB merging system

In order to use the LDB/LKB merging System you need to have previously loaded the LDB and the LKB environments (because LDB/LKB makes use of both environments). The LDB/LKB merging System must be loaded too[2]. In the top level menu of the main window, a new item named LDB/LKB appears. In order to generate new LDB/LKB lexicon files you need to have loaded in your System an LKB type system as well as an LKB lexicon. (See Fig. 2)

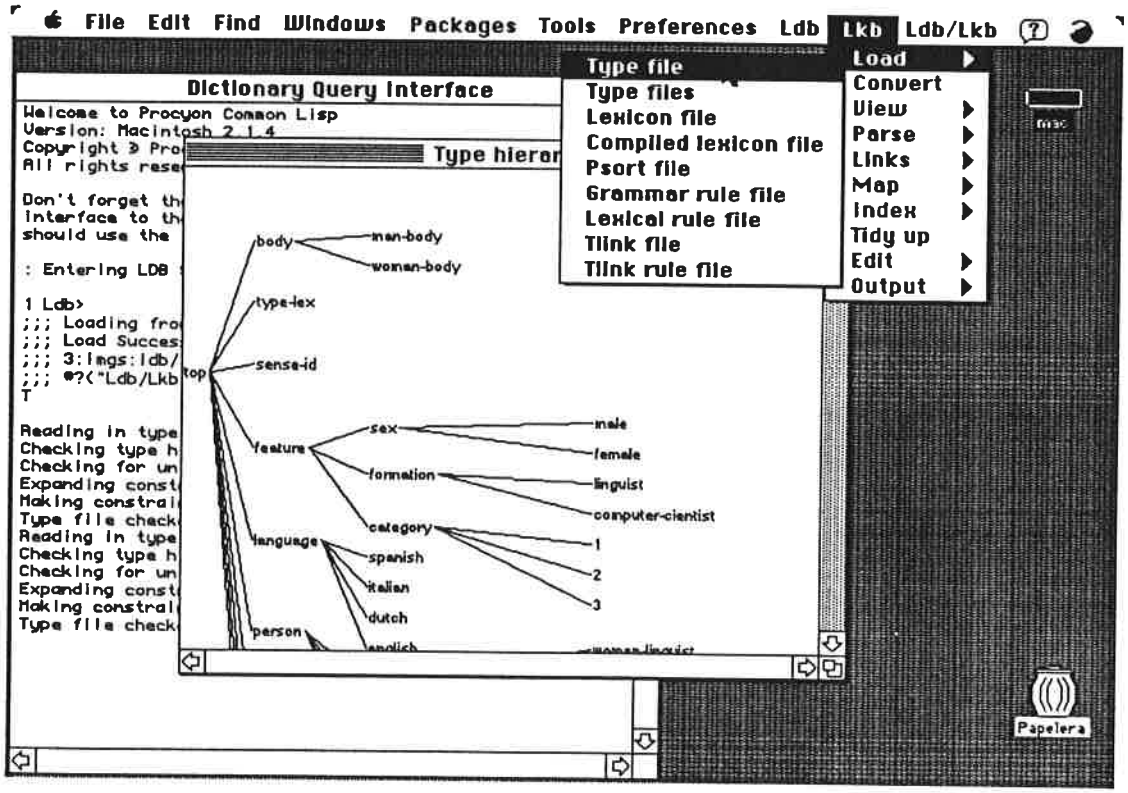


Fig. 2: LKB options.

[2] An image containing all the needed software can be generated and loaded.

The LDB/LKB interface has the following functionalities:

• **New lexicon:** This option generates the LDB/LKB lexicon from the lexicon loaded in the LKB. The program creates a new LDB/LKB interface (via the lexicon) and indexes all the generated lexical entries (see Fig. 3)

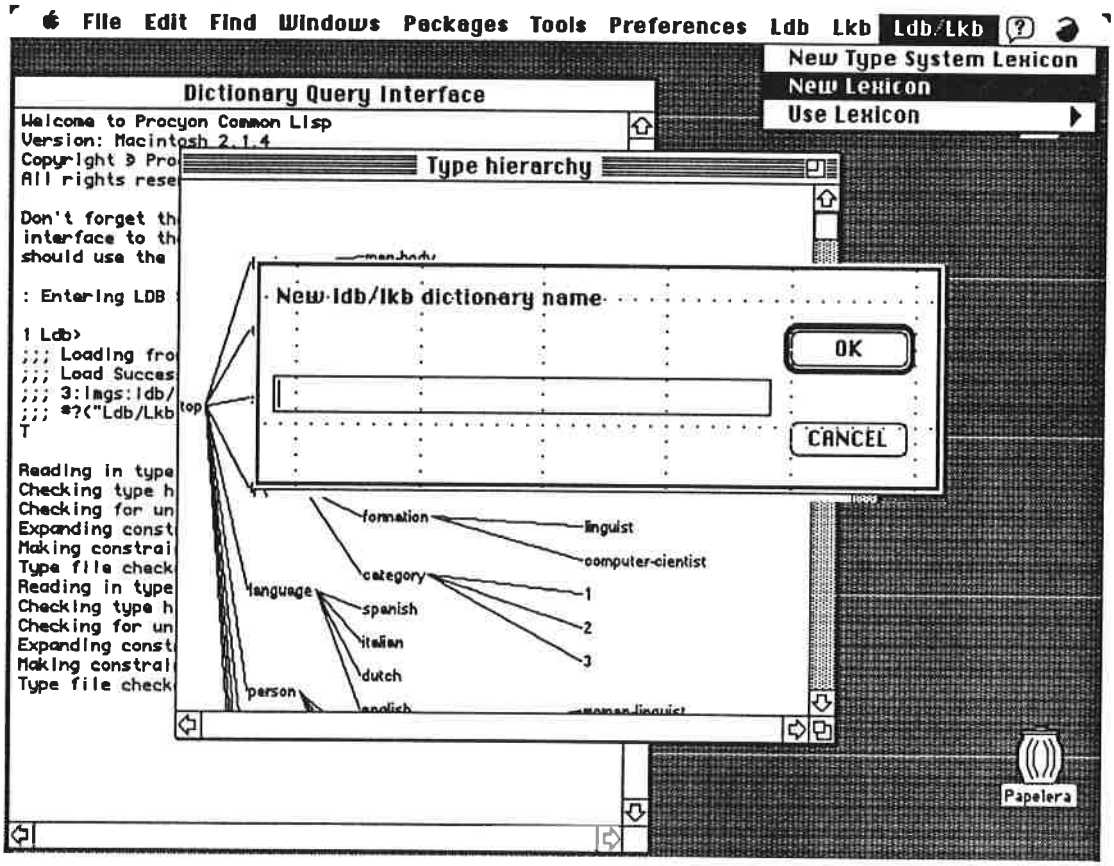


Fig.3: name of the new ldb/lkb lexicon.

• **Use lexicon:** This option selects an already indexed LDB/LKB lexicon. Two new options appear in the menu, one for building a query and the other for looking up a lexical entry (corresponding to the selected lexicon). This process is accumulative, so for each lexicon selected the corresponding pair of options are added to the menu (see Fig. 4).

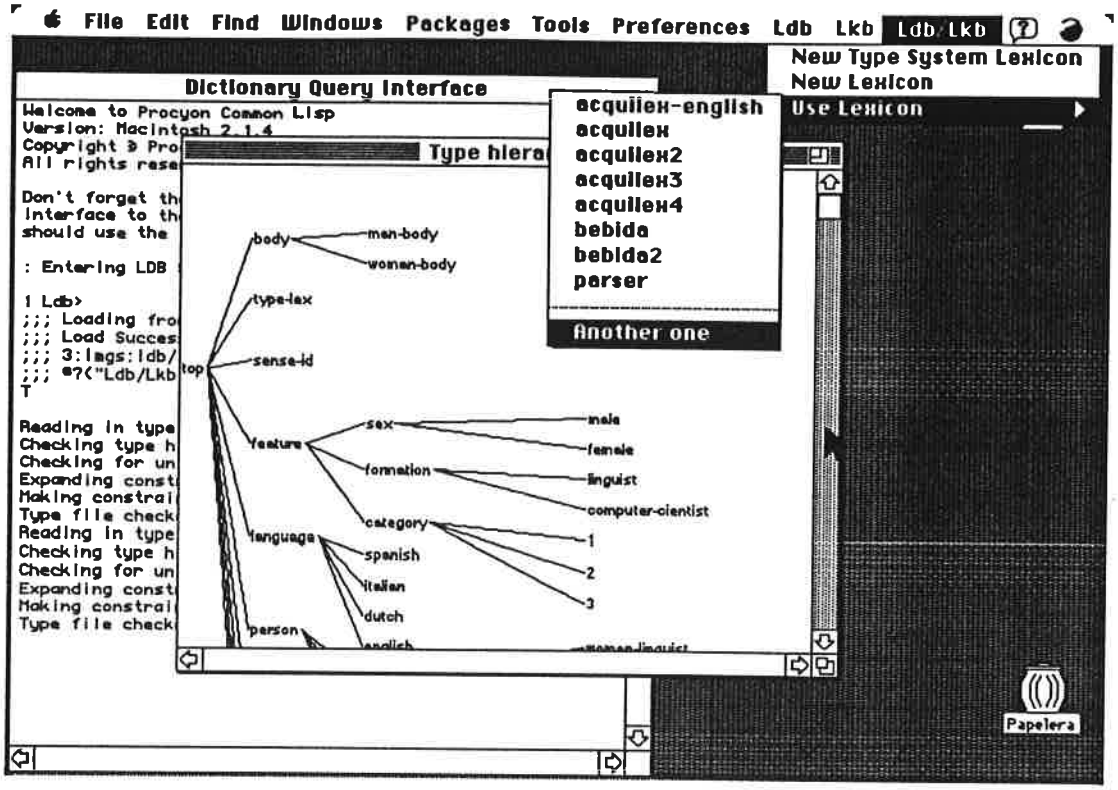


Fig. 4: LDB/LKB lexicons ready for using.

• **New type system lexicon:** A new lexicon corresponding to the current type system is created in order to allow queries against the Type System.

Appendix 2 contains a toy type system and appendix 3 a toy lexicon file. All the examples presented in this document are taken from these files.

1.3 Querying the LDB/LKB merging system

There are two ways of querying a LKB lexicon (or a LKB type system): by means of the selectors in the LDB/LKB menu or by means of clicking the item to be selected as a base for building the query, in the sensitive type system or entry windows.

1.3.1 Querying through menu selection

As we have mentioned above, for each lexicon selected a pair of menu options are added to the LDB/LKB menu:

new LDB/LKB query

LDB/LKB entry

If you choose the first one you can build queries to the LDB/LKB lexicon in a similar way as to any other dictionary loaded in the LDB. With the second option you can ask for a lexical entry (it will appear in LKB format), which will be in the lexicon generated by the LDB/LKB merging System.

When choosing the LDB/LKB entry option, a window appears and the headword must be typed. A warning is displayed if the headword matches no entry in the current lexicon. Another window will be presented for selecting the appropriate sense (if more than one senses exist).

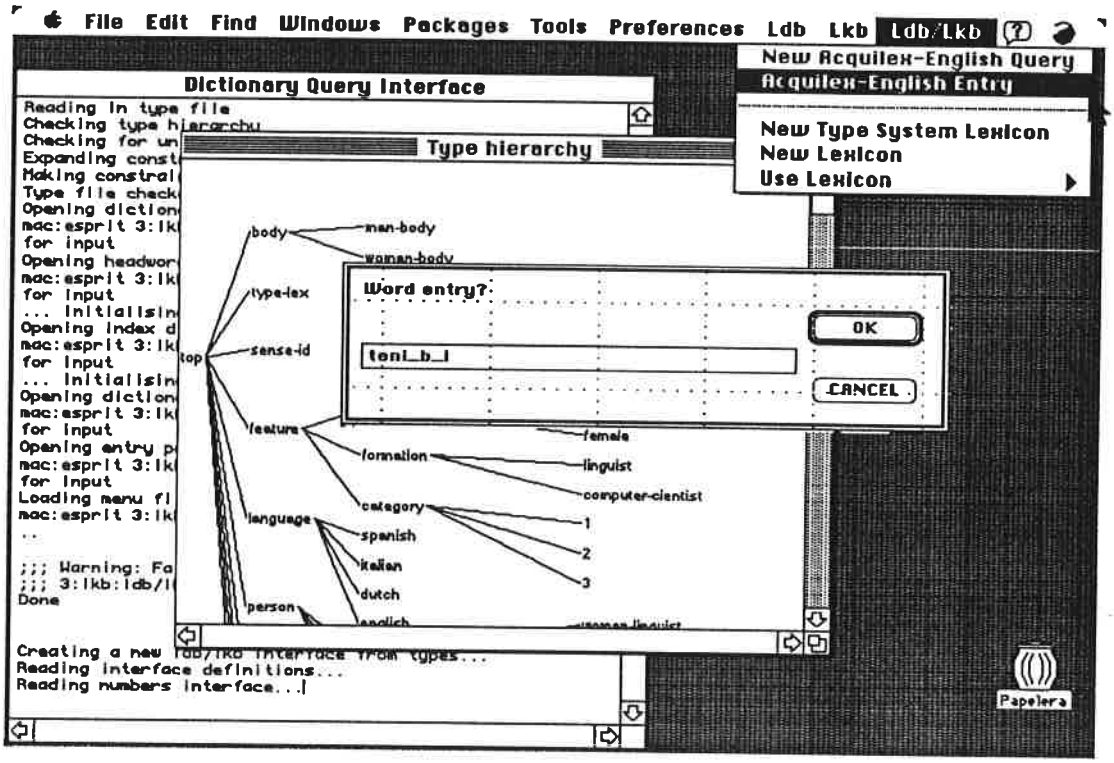


Fig. 5: lexical entry name for looking up.

When choosing the new LDB/LKB query option, a window appears for building the query tree. Fig. 6 presents this initial window. The process of construction of the query tree is described in the next section.

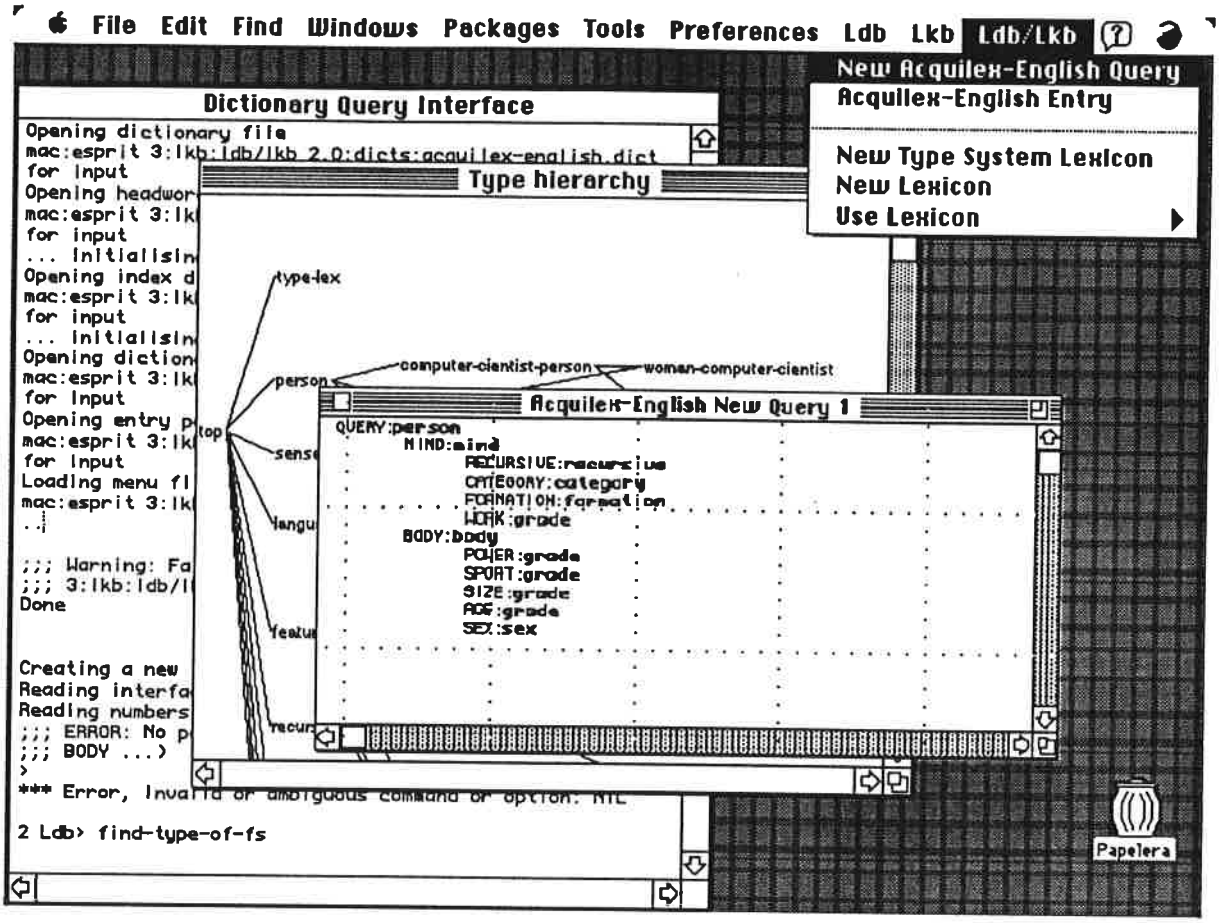
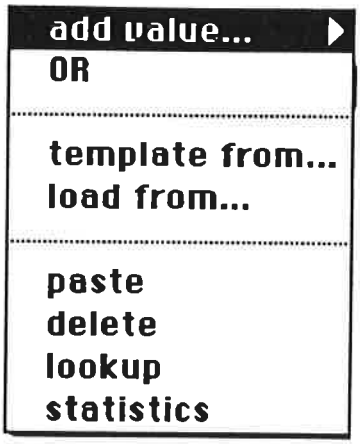


Fig. 6: a query extracted from the Type System.

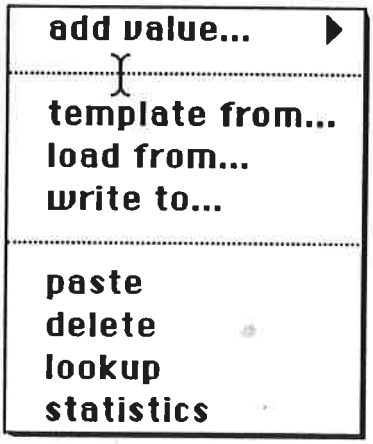
1.3.2 Building a query tree

The query tree is composed by four kinds of nodes: type nodes, feature nodes, string nodes and menu nodes. The former two are non terminal and the last two are terminals. Of course appropriateness conditions hold; so for a type node only feature nodes appropriate to this node are displayed and can be chosen and for a feature node only type nodes belonging to its domain are displayed and can be selected.

The window for building the query tree contains, at the beginning only the root node of the tree (*QUERY*). Clicking on this node the following menu is popped down:



This menu is general for any non terminal (in the type structure) clicked node without descendants in the query, except its second part (*template from* and *load from*), only used in the *query* node. The following menu is displayed when clicking on the *query* node except when it has the *OR* option selected as a daughter in the query:



The second part of this menu does not appear if the clicked node without *OR* selected descendants is not the query node.

Moreover, the first part of these menus is not displayed neither for terminal nodes nor for nodes with *OR* selected descendant.

In the case of feature nodes without *OR* selected descendant, a *copy* option is added to the menu.

We will comment briefly the different choices the user is able to select.

- **add value:** This option allows adding a new node to the query tree. Initially, any type can be chosen. Later on, only the features appropriate to the selected type or the types appropriate as domains for the selected feature are displayed in the menu.
- **or:** allows to create a disjunction of subqueries. As regard as logical operators, there are only two possibilities for a node: creating a conjunction (AND) of all the desired values (default logical operator) or creating a disjunction (OR) of them.
- **template from:** used for generating a query from a lexical entry. This one is displayed like

the unique daughter for the initial node *query*. So, all already existing daughter will be previously deleted.

- **write to:** allows the user to save the current query into a file.
- **load from:** used to recover a query previously saved into a file with the *write to* option.
- **copy:** This option permits selecting the subtree rooted in a selected type node and store it in a buffer for further pasting.
- **paste:** This option permits inserting a previously copied tree fragment as type node daughter.
- **delete:** This option allows removing the tree fragment rooted in the selected node.
- **lookup:** Triggers the searching process.
- **statistics:** Some statistical information, basically counts about the number of entries matching the query, are provided.

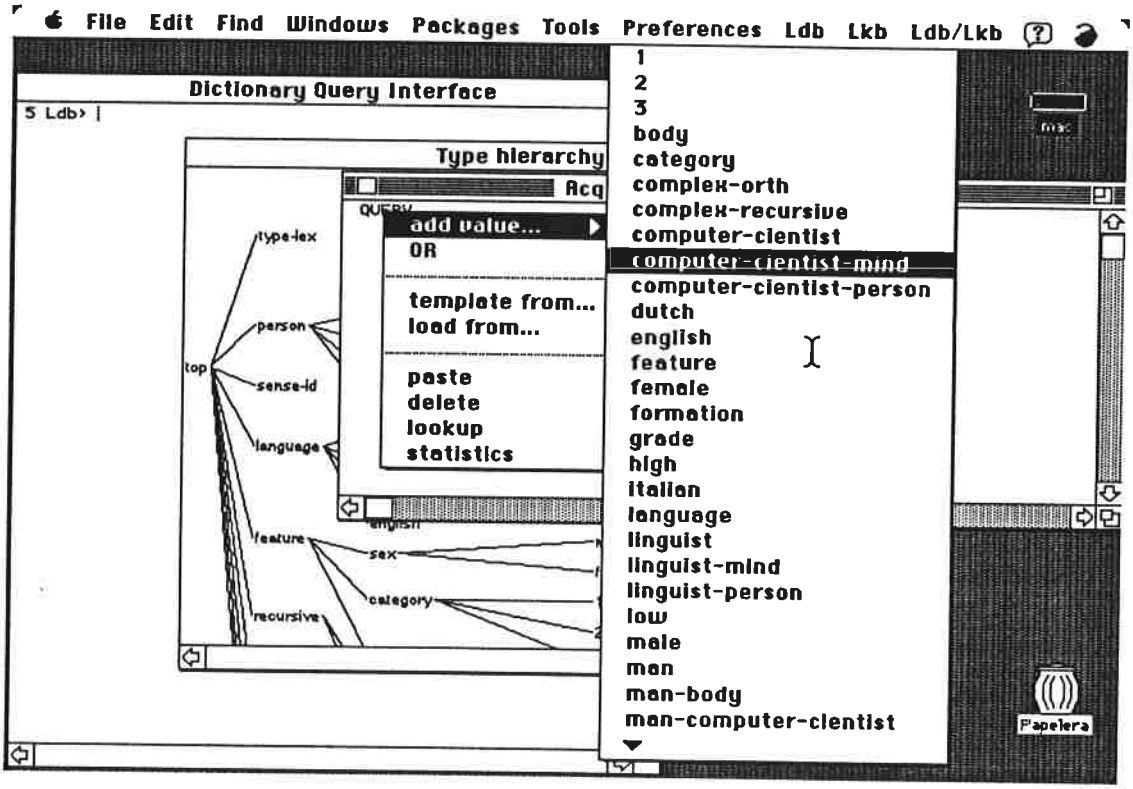
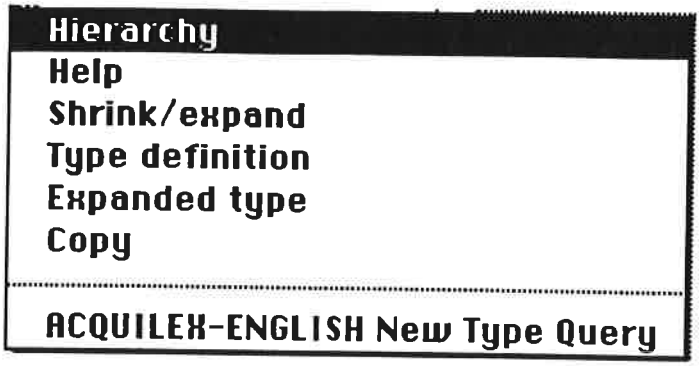


Fig. 7: selecting a type from the pop-up menu.

1.3.3 Querying through type system or entry clicking

This utility allows to make queries from a template of the constraint of a clicked type. Moreover, this template can be treated in the same way that 1.3.2. So, it's possible to build complex queries against dictionaries by means of copying and pasting fragments of any query tree from type constraints.

When we click on a type node a menu like next one is popped down.



First part of this menu is a group of utilities to get a friendly way for building queries and watching type system.

Now we will comment briefly the different options.

- **hierarchy:** locates the clicked type in the type system window. Allows to get a vision of its semantic environment.
- **help:** gives a single explanation of the clicked type, consisting of the comment string attached to the lexical entry in the source lexicon.
- **shrink/expand:** hides (restore if hidden) the daughter subtree of the clicked type. It is used to provide partial views of the type system.
- **type definition:** displays the definition of the clicked type.
- **expanded type:** displays the expanded definition of the clicked type.
- **copy:** this option is not active in the type system window but it is elsewhere. It allows to buffer the daughter feature structures of the clicked type for later pasting in a query.
- **new type query:** allows building a new query initialised with the clicked type constraint.

Appendix 2: Sample Type Structure

```
top ().

person (top)
  "The person type has no constraints."
  < body > = body
  < mind > = mind.

body (top)
  "The body type has no constraints."
  < sex > = sex
  < age > = grade
  < size > = grade
  < sport > = grade
  < power > = grade.

man-body (body)
  "Everybody who is a man has sex male."
  < sex > = male.

woman-body (body)
  "Everybody who is a woman has sex female."
  < sex > = female
  < age > = < size >.

computer-scientist-mind (mind)
  "Everybody who is a computer-scientist has formation computer-scientist."
  < formation > = computer-scientist.

linguist-mind (mind)
  "Everybody who is a linguist has formation linguist."
  < formation > = linguist.

mind (top)
  "The mind type has no constraints."
  < work > = grade
  < formation > = formation
  < category > = category
  < recursive > = recursive.

man (person)
  "A person who is a man has the body of a man."
  < body > = man-body.

woman (person)
  "A person who is a woman has the body of a woman."
  < body > = woman-body.

computer-scientist-person (person)
  "A person who is a computer-scientist-person has the mind of a
  computer-scientist-mind."
  < mind > = computer-scientist-mind.
```

```
linguist-person (person)
  "A person who is a linguist-person has the mind of a
  linguist-mind."
  < mind > = linguist-mind.

computer-scientist-man (computer-scientist-person man)
  "Everybody who is a man and a computer-scientist.".

computer-scientist- woman (computer-scientist-person woman)
  "Everybody who is a woman and a computer-scientist.".

linguist-man (linguist-person man)
  "Everybody who is a man and a linguist.".

wolinguist-man (linguist-person woman)
  "Everybody who is a woman and a linguist.".

feature (top).

formation (feature)
(OR linguist computer-scientist).

sex (feature)
(OR male female).

category (feature)
(OR 1 2 3).

grade (top)
(OR low medium high).

orth (top).

complex-orth (orth)
  < orth1 > = orth
  < orth2 > = orth.

recursive (top).

complex-recursive (recursive)
  < recursive1 > = recursive
  < recursive2 > = recursive.

string (orth recursive).

sense-id (top)
  < fs-id > = string
  < language > = language.

language (top)  (OR Spanish Italian Dutch English).

type-lex (top)
  < orth > = orth
  < sense-id > = sense-id
  < type > = top.
```

Appendix 3: Sample LKB lexicon

```
ted B_I_1
linguist-man
  < body : age > = medium
  < body : size > = low
  < mind : work > = high.

mariona B_I_1
linguist-woman
  < body > < irene_B_I_1 < body >
  < body : sport > = low
  < mind : work > = high
  < mind : formation > = linguist
  < mind : category > = category
  < mind : recursive > = "a".

irene B_I_1
linguist-woman
  < body : sex > = female
  < body : age > = low
  < body : age > = < body : size >
  < body : power > = low
  < body : sport > = < body : power >
  < mind : work > = high
  < mind : formation > = linguist
  < mind : category > = 1
  < mind : recursive : recursive1 > = "a"
  < mind : recursive : recursive2 : recursive1 > = "b".

ann B_I_1
computer-scientist- woman
  < body : sex > = female
  < body : age > = low
  < body : size > = grade
  < body : sport > = grade
  < mind : work > = grade
  < mind : formation > = formation
  < mind : category > = (1 2)
  < mind : recursive : recursive1 > = "a"
  < mind : recursive : recursive2 : recursive1 > = "b"
  < mind : recursive : recursive2 : recursive2 > = recursive.

alicia B_I_1
computer-scientist- woman
  < body : sex > = female
  < body : age > = low
  < body : size > = low
  < body : sport > = high
  < mind : work > = low
  < mind : formation > = computer-scientist
  < mind : category > = 3
  < mind : recursive : recursive1 > = "a"
  < mind : recursive : recursive2 : recursive1 > = "b"
```

< mind : recursive : recursive2 : recursive2 > = "c".

toni B_I_1
linguist- woman
< body : sex > = female
< body : age > = low
< body : size > = low
< body : sport > = grade
< mind : work > = grade
< mind : formation > = linguist
< mind : category > = category.

german B_I_1
computer-scientist-man
< body : sex > = male
< body : age > = medium
< body : size > = high
< body : sport > = low
< mind : work > = grade
< mind : formation > = computer-scientist
< mind : category > = 1.

john B_I_1
computer-scientist-man
< body : age > = low
< body : size > = high
< body : sport > = high
< mind : work > = medium
< mind : category > = 2.

kiku B_I_1
computer-scientist-man
< body : sex > = male
< body : age > = low
< body : size > = high
< body : sport > = high
< mind : work > = medium
< mind : formation > = computer-scientist
< mind : category > = 2.

horacio B_I_1
computer-scientist-man
< body : sex > = male
< body : age > = high
< body : size > = grade
< body : sport > = low
< mind : work > = grade
< mind : formation > = computer-scientist
< mind : category > = 3.

toni B_I_2
computer-scientist-man
< body : sex > = male
< body : age > = low
< body : size > = high

< body : sport > = high
< mind : work > = high
< mind : formation > = computer-scientist
< mind : category > = category.

Appendix 4: Implementation outlines.

4.1 Changes in the LDB and LKB systems.

We have tried on one hand to modify the minimal number of functions in the LDB and the LKB systems made respectively, by John Carroll and Ann Copestake, and on the other hand to maintain a high level of compatibility within the environments already existing in order to specify the new one in terms of the old mechanism, although introducing new capabilities. The previous systems have been, thus, embedded within LDB/LKB environment.

The LDB source files are placed in two folders: client and ddb. The first one has the high level functions (query process) and the second the low level ones (index process). We have modified various functions in two files in the client folder of the LDB. These files are the following:

- gdevice.lsp

The modified functions are three:

make-active-menu-window

tr-redraw-tree

headword-sense-display1

- gmenu.lsp

The modified function is:

lookup-tree-query

We have also modified one function in a file in the ddb folder of LDB, that is

- hdr.lsp

The modified function is:

general-alphanumericp

These modified functions are in ldb-changes.lsp file inside the LDB/LKB folder. This file has also a set of new functions, all of these related to the LDB environment.

The LKB source files are in the lkb folder [Copestake 91]. The LDB/LKB folder is inside the lkb folder and we have changed two functions in the following two files:

- types.lsp

The improved function is:

get-descendants

- activefs.lsp

The modified function here is:

top-fs-action

These modified functions are in lkb-changes.lsp file inside the LDB/LKB folder. This file has also a set of new functions, all of these related to the LKB environment.

4.2 Who is who in the LDB/LKB merging System.

This current version (July'94) of LDB/LKB merging software has been implemented in the files:

- lkb-to-ldb-format.lsp

This file contains the functions needed for translating the lexical entries loaded in the lkb file to the dictionary entries in a lispified version compatible with the LDB System. These dictionary entries contain the minimal information (the canonical lexical entry explained previously) not represented in the LKB type system. This file has the functions that will create the LDB/LKB.dict file.

- cldb/lkb.lsp

This file contains the main functions related with the client folder of LDB. These functions deal with the query process.

- dldb/lkb.lsp

In this file we find the main functions related to the ddb folder of the LDB. These functions deal with the index process.

- conv-LDB/LKB-struct.lsp

This file provides to the LDB/LKB merging System with the capabilities for viewing a lkb lexical entry (in lkb format) from the LDB/LKB.dict file(in ldb format).

- graph01.lsp

We can found here the basic functions for managing a graph associated to the type system. We use this graph to index and query the LDB/LKB dictionary.

- read-entry.lsp

This file contains the functions for creating the graph structure from the lexical entry file in ldb format (LDB/LKB.dict file). In this case we have only represented in the graph those types and features that appear in the lexical entries.

- read-types.lsp

Here we find the functions for creating the graph structure from the type system loaded by the the LKB System. In this case we have represented in the graph all the possible types and features that we can represent with the type system.

- gener.lsp

In this file we can find the functions that create the ldb interface definitions variables (needed in cldb/lkb.lsp file) from the previously generated graph.

- ldb-changes.lsp

This file has the functions related to the ldb environment. Basically, these functions deal with the query process, they are, the main functions for interpreting a query made to the LDB/LKB merging System in the terms of the LDB environment.

- lkb-changes.lsp

This file has the functions related to the LDB environment.

- ldb/lkb-top.lsp

These are the top level functions that start all the process.

- ldb/lkb-load.lsp

When this file is loaded all the other files previously described are loaded, and the LDB/LKB merging System is ready to work.

Departament de Llenguatges i Sistemes Informàtics
Universitat Politècnica de Catalunya

Research Reports – 1994

LSI–94–1–R "Logspace and logtime leaf languages", Birgit Jenner, Pierre McKenzie, and Denis Thérien.

LSI–94–2–R "Degrees and reducibilities of easy tally sets", Montserrat Hermo.

LSI–94–3–R "Isothetic polyhedra and monotone boolean formulae", Robert Juan-Arinyo.

LSI–94–4–R "Una modelización de la incompletitud en los programas" (written in Spanish), Javier Pérez Campo.

LSI–94–5–R "A multiple shooting vectorial algorithm for progressive radiosity", Blanca Garcia and Xavier Pueyo.

LSI–94–6–R "Construction of the Face Octree model", Núria Pla-Garcia.

LSI–94–7–R "On the expected depth of boolean circuits", Josep Díaz, María J. Serna, Paul Spirakis, and Jacobo Torán.

LSI–94–8–R "A transformation scheme for double recursion", José L. Balcázar.

LSI–94–9–R "On architectures for federated DB systems", Fèlix Saltor, Benet Campderrich, and Manuel García-Solaco.

LSI–94–10–R "Relative knowledge and belief: SKL preferred model frames", Matías Alvarado.

LSI–94–11–R "A top-down design of a parallel dictionary using skip lists", Joaquim Gabarró, Conrado Martínez, and Xavier Messeguer.

LSI–94–12–R "Analysis of an optimized search algorithm for skip lists", Peter Kirschenhofer, Conrado Martínez, and Helmut Prodinger.

LSI–94–13–R "Bases de dades bitemporals" (written in Catalan), Carme Martín and Jaume Sistac.

LSI–94–14–R "A volume visualization algorithm using a coherent extended weight matrix", Daniela Tost, Anna Puig, and Isabel Navazo.

LSI–94–15–R "Deriving transaction specifications from deductive conceptual models of information systems", María Ribera Sancho and Antoni Olivé.

LSI–94–16–R "Some remarks on the approximability of graph layout problems", Josep Díaz, María J. Serna, and Paul Spirakis.

LSI–94–17–R "SAREL: An assistance system for writing software specifications in natural language", Núria Castell and Àngels Hernández.

LSI–94–18–R "Medición del factor modificabilidad en el proyecto LESD" (written in Spanish), Núria Castell and Olga Slávkova

LSI–94–19–R "Algorismes paral·lels SIMD d'extracció de models de fronteres a partir d'arbres octals no exactes" (written in Catalan), Jaume Solé and Robert Juan-Arinyo.

LSI–94–20–R "Una paral·lelització SIMD de la conversió d'objectes codificats segons el model de fronteres al mdel d'octrees clàssics" (written in Catalan), Jaume Solé and Robert Juan-Arinyo.

LSI–94–21–R "Clausal proof nets and discontinuity", Glyn Morrill.

LSI–94–22–R "A formal method for the synthesis of update transactions in deductive databases without existential rules", Joan A. Pastor.

LSI–94–23–R "On the sparse set conjecture for sets with low density", Harry Buhrman and Montserrat Hermo.

LSI–94–24–R "On infinite sequences (almost) as easy as π", José L. Balcázar, Ricard Gavaldà, and Montserrat Hermo.

LSI–94–25–R "Updating knowledge bases while maintaining their consistency", Ernest Teniente and Antoni Olivé.

LSI–94–26–R "Structural facilitation and structural inhibition", Glyn Morrill.

LSI–94–27–R "Formalising existential rule treatment in the automatic synthesis of update transactions in deductive databases", Joan A. Pastor.

LSI–94–28–R "Proceedings of the Fifth International Workshop on the Deductive Approach to Information Systems and Databases" (Aiguablava, 1994), Antoni Olivé (editor).

LSI–94–29–R "Towards a VRQS representation", Mariona Taulé Delor.

LSI–94–30–R "MACO: Morphological Analyzer Corpus-Oriented", Sofía Acebo, Alicia Ageno, Salvador Climent, Javier Farreres, Lluís Padró, Francesc Ribas, Horacio Rodríguez, and Oscar Soler.

LSI–94–31–R "Lexical mismatches: a semantic representation of adjectives in the LKB", Salvador Climent and Carme Soler.

LSI–94–32–R "LDB/LKB integration", German Rigau, Horacio Rodríguez, and Jordi Turmo.

LSI–94–33–R "Learning more appropriate selectional restrictions", Francesc Ribas.

LSI–94–34–R "Oracles and queries that are sufficient for exact learning", Nader H. Bshouty, Richard Cleve, Ricard Gavaldà, Sampath Kannan, and Christino Tamon.

Copies of reports can be ordered from:

Nuria Sánchez
Departament de Llenguatges i Sistemes Informàtics
Universitat Politècnica de Catalunya
Pau Gargallo, 5
08028 Barcelona, Spain
secrelsi@lsi.upc.es