# Building Interfaces from DATR to PATR

**Term Paper in Computational Linguistics 2**

Klaus Zechner
*klaus@cogsci.ed.ac.uk*

Lecturers: David Milward, David Beaver
Supervisor: Jo Calder
Centre for Cognitive Science
University of Edinburgh

7 April 1995

**Abstract**

This paper describes the implementation of an interface which allows runtime queries from a PATR-II grammar to a precompiled DATR lexicon during the parse of a sentence. The lexical information requested by the PATR system is then unified to the PATR parse tree. Alternatively, there is the option to create all the possible dictionary entries from a DATR lexicon before parsing starts; these entries are stored in a file and can be loaded together with a PATR-II grammar.

The main advantages of this interface are firstly, that it allows the grammar writer to make use of the powerful multiple and default inheritance mechanism (with the possibility of overriding defaults) which DATR provides — and therefore to be able to exploit all the redundancies — and secondly, that the lexical information stored as a DATR theory can be accessed at runtime of the parser, if and only if it is needed.

# Contents

# 0 Introduction

Despite the fact that one of the purposes of the DATR formalism is to provide a system which is

> intended to be complementary to ... PATR (Gazdar (1989: 11)),

and which

> has the necessary expressive power to encode the lexical entries presupposed by contemporary work in the unification grammar tradition (Gazdar (1989: 1)),

there have not been many noticeable approaches so far which deal with the issue of interfacing DATR and PATR in order to get a combined system, maximally benefitting from the advantages of both of its components.

PATR[1] is a unification based grammar formalism where lexical information can be included (and usually, this is in fact the case) but cannot be handled very economically; the only option here is to use macros (templates) to simplify the lexicon definition. However, if one uses a DATR theory for encoding lexical information, one can exploit all the features of a powerful multiple default inheritance mechanism, where regularities, subregularities and idiosyncrasies can easily be captured.

The ideal solution, then, would be to separate the grammar and the lexicon: to write the grammar using the PATR formalism and to encode all the lexical information (morphological, syntactic, semantic) in DATR; and whenever the parser needs the information from the lexicon, it accesses DATR directly to obtain it.

We shall see, that for some reasons, the approach pursued here is slightly different but comes quite close to this ideal goal.

# 1 Representation of lexical information in a PATR system

## 1.1 The syntax of lexical entries

When we use a PATR system for encoding both grammar and lexicon, a typical lexical entry would look like that:

```
dict(plays, X@[ X/syn/cat <=> v,
                X/syn/subcat/first/syn/cat <=> np,
                X/syn/subcat/rest <=> nil,
                X/mor/num <=> sing,
```

[1] Throughout this paper I will refer to the following implementations of PATR resp. DATR: the Edinburgh Sictus-Prolog PATR-II system by A.Schoeter and the Sussex Prolog DATR system by R.Evans *et al.* (Version 2.00). The general discussion in this paper, however, will also apply for any other implementations of PATR or DATR.

```
X/mor/pers <=> 3,
X/sem/name <=> play,
X/sem/type <=> activity,
X/syn/subcat/first/sem <=> X/sem/arg ]).
```

I.e., we have a Prolog clause, `dict/2`, which is called by the parser whenever it reads the next token (word) from the input string (and, of course, during backtracking as well). The first argument of `dict/2` has to match this word, the second argument returns all the unified attribute–value pairs in square brackets: these path equations are evaluated and "attached" to the tree variable `X` which forms the second argument of the `dict/2` predicate.

In principle, the path equations can have two forms:

```
(a) X/attr1/.../attrN <=> value
```

where an atomic value is assigned to a path; or

```
(b) X/attr11/.../attr1N <=> X/attr21/.../attr2M
```

where the values of two distinct paths are unified.

### 1.2 Handling of morphological ambiguities

Consider the verb TO PLAY whose present tense active paradigm has six elements (two numbers times three persons) but only two morphologically distinct forms (`plays` for third person singular and `play` elsewhere). There are two alternative ways of defining lexical entries in a PATR system: Either, one defines just two lexical entries for these two forms and ignores the number and person features in the `play` entry; in this case one has to face the problem of over-generation because the parser will not reject sentences like: *The boy play the violin.* Or, on the other hand, one explicitely encodes all the six number–person combinations, in which case, of course, the lexicon will get quite large if it consists of a lot of lexemes like that.

In the next section, we will see how this issue can be handled via the DATR formalism in a much more elegant way.

## 2 Representation and retrieval of lexical information in DATR

### 2.1 General properties of the DATR formalism

Though DATR can be used for a whole variety of things (see e.g. the examples in Evans & Gazdar (1990)), its main purpose is to be

> a formal language which is intended to facilitate the representation of lexical information. The goal ... has been to design and implement a simple language that (i) has the necessary expressive power to encode the lexical entries presupposed

by contemporary work in the unification grammar tradition, (ii) can express all the evident generalisations about the information explicit in those entries, (iii) has an explicit theory of inference, (iv) is computationally tractable and (v) has an explicit declarative semantics. DATR defines networks allowing multiple default inheritance of information through links typed by attribute paths. ... DATR has a set of seven rules of inference, a general principle of default inference, and a model-theoretic semantics inspired by that developed for auto-epistemic logic. DATR is *functional*, that is, it defines a mapping which assigns unique values to node attribute–path pairs. Recovery of these values is deterministic — no search is involved.

Linguistically, the language allows a uniform treatment of irregularity, subregularity and regularity and provides a logical basis for theories of suppletion, syncretism, declension, conjugation, and the paradigm. In addition, it offers an explicit reconstruction of informal notions such as "blocking", the "elsewhere condition", and the "proper inclusion precedence" principle. (Gazdar (1989: 1))

## 2.2 Representation of lexical information in DATR

The way information is stored in a DATR theory is very straightforward. In principle, one has to distinguish between NODES, PATHS and VALUES. The nodes can be seen as the *entry points* to the DATR network, the paths can be interpreted as links (respectively, as a list of links) which either point to other nodes, other paths or directly to a (definite) value.

I will not expand on the syntax and semantics of DATR here because this was done extensively in several papers in Evans & Gazdar (1990) and e.g. also in Erjavec (1992). I will just illustrate, how the above discussed example of storing morphologically ambiguous words can be handled within the DATR formalism (this example is taken out from the file `music2.dtr`):

```
WORD:          <> == unspecified
               <mor num $num pers $pers> == "<mor $num $pers>"
               <path1 gapin path2 gapout> == path_unify.

VERB:          <> == WORD
               <syn cat> == v
               <mor sing 1> == "<root>"
               <mor sing 2> == "<root>"
               <mor sing 3> == ("<root>" s)
               <mor plur 1> == "<root>"
               <mor plur 2> == "<root>"
               <mor plur 3> == "<root>".

VERB_NPCOMP:   <> == VERB
               <syn subcat first syn cat> == np
               <syn subcat rest> == nil.

PLAY_V:        <> == VERB_NPCOMP
               <root> == play.
```

In this example one can see several of the interesting features of the DATR language:

- The entry for the lexeme `PLAY_V` contains just two DATR equations though it comprises information about all its six forms.
- The *default* is indicated in the first line: it points to the node `VERB_NPCOMP` and there further on to the node `VERB`, where the conjugation of the relevant group of verbs is given.
- But even there, we have a reference to the global node `WORD`, where the general rules holding for all words in the lexicon are specified.
- At the more general nodes, the item `‘‘<root>’’` refers to the respective value of the `<root>`-path in the node for which a query is performed. That means in the case of this example, that the morpheme *-s* will get appended to the stem *play* (in the third person singular form).
- The first line of the `WORD` node states that all paths where no values can be found are unspecified (by default).
- The second line expands paths of the form e.g. `<mor sing 1>` to `<mor num sing pers 1>`. The reason for this is explained in section 4.
- The third line gives an example for a path unification declaration in DATR (see example (b) in section 1.1). We have developed a specific syntax for this purpose which uses reserved words. This will be explained later on, as well.

## 2.3 Querying DATR

Using the DATR language can be seen as a way of representing a formal theory. The queries, then, can be interpreted as theorems of the theory. The actual DATR system is designed as a deterministic one, i.e. the user has to question the system in a way that it can access the information without the need of backtracking.

As a consequence of this requirement, the DATR query has the form:

```
datr_query(+Node, ?Path, ?Value)
```

where the node has to be specified and both the path and value parameters can either be specified (i.e. be a non-variable argument) or left unspecified, in which case DATR just returns the first solution it can find.

In the first Sussex DATR implementation, there were only limited possibilities of querying DATR. However, in the improved version 2.0, described in Jenkins (1990), there are ways of

- specifying those nodes that are not “relevant” for user queries
- specifying those paths that shall be shown in a “general” query (we will call these specifications now *relevant information*), and

- querying all this relevant information from either one node or all the relevant nodes (predicate: `datr_theorem`)

With these extensions, the possibilities of querying DATR have been significantly increased and we shall see in the following sections how these runtime improvements can help a great deal in the implementation of the DATR–PATR interface.

# 3 DATR–PATR interfaces and the reverse query problem

## 3.1 General issues and problems

Accessing lexical information from a DATR theory would be very easy and straightforward if there were a 1:1–correspondence between a word form in the dictionary and a node in the DATR theory. While this may be possible in isolating, non-inflectional languages like Chinese, this is quite different for most of the European languages, such as German, French and even English: here we typically have morphological paradigms for verbs, nouns and adjectives. That is, we want to generalise over different classes of declensions and conjugations and derive all morphological forms of one lexeme at one node via the inheritance mechanism of DATR (see the above example in section 2.2).

If the parser encounters an inflected word form, it has to find out from which node (stem) it was derived in order to get the necessary morphological, syntactic and semantic information it needs.

Let us look at an example in order to clarify matters: Assume the parser's current token is the word PLAYS. In the DATR theory of the example file `music2.dtr` there are two possible morphological paths which result in the value list `[play,s]`:

```
(a) <mor num plur pers 3>
(b) <mor num sing pers 3>
```

The first path corresponds to PLAYS as a noun, the second as a verb. In this case, these two paths originate at two different nodes which means that not only this morphological information differs but also the syntactic and semantic information associated with these two lexical entries. (We could call this *inter-node ambiguity* as opposed to the (more common) *intra-node ambiguity* which has been discussed in section 1.2).

This example illustrates that we need a mechanism of *reverse querying* in order to get the lexical information from DATR to the PATR system (or, to be more precise, to the parser which operates on the PATR grammar): As stated above, DATR is designed for querying for values, if the node (and the path) is known in advance. In our case, however, we know the values — to be exact: the concatenation of the value list — but neither the paths nor the nodes. Moreover, we do not only want to get access to the path "pointing" to the value list in question but to *all* the other relevant paths of the node where the value list "belongs" to, as well. That is, in fact, we need to obtain the syntactic and semantic information together with the morphological one.

### 3.2 Previous approaches

Kilbury *et al.* (1991) incorporate the syntactic characters for Prolog lists (brackets, colons, commas) directly in the DATR theory so that the result of a query can be transformed with some quite trivial intermediate steps to a PATR suitable difference list (see Zechner (1995) for a more detailed account on that). However, this idea is on the one hand not very elegant in terms of the DATR encoding and on the other hand it is only useful in a system where the name of the queried node is known in advance; i.e., this approach cannot be taken to solve the reverse query problem.

Zechner (1995) describes an on-line access system where a DATR theory is used to store semantic information which is queried directly via a Prolog predicate from the PATR lexicon. As the nodes are — and *have* to be — always known at query time, this system cannot be employed for reverse queries either.

Langer (1994) explicitely addresses the reverse query issue and provides an algorithm for a bottom–up chart parser which is able to retrieve all the possible node–path pairs[2] in a DATR theory with a value list as input query.[3] The problems with this approach for our DATR–PATR interface are however, that on the one hand, it requires a "split" list of values as a query (e.g. `[play,s]` for the word PLAYS) — which is not what we have got when the parser encounters an inflected word — and, on the other hand, only returns the nodes together with the "direct" morphological paths "pointing" to the value list of the query.–

The next section describes a system which is capable of solving these problems in an elegant and efficient way.

## 4 Description of an efficient and general DATR–PATR interface

### 4.1 Creating the dictionary "on the fly"

The first problem our system[4] has to solve is the morpheme concatenation problem. It seems as if this were a superficial matter but, as we already have seen, it is not at all. Even if we take a reverse query approach like the one described in Langer (1994) there is no way for DATR in finding out which value list corresponds to an atomic (inflected) word form.

Therefore, I decided to precompile the DATR theory in a format which has the atomic, inflected forms as values instead of value lists. The format of these `npv/3` predicates (for: NODE–PATH–VALUE) is illustrated in the following example, where all the `npv/3` facts for both the verbal and the nominal node of PLAY are listed:[5]

---

[2] The length of the paths has to be set to a predefined limit for computational reasons.

[3] Langer has implemented this algorithm in Arity Prolog.

[4] The description here is a very general one; for a more detailed account you have to look at the well–commented source code in the appendix. (See the file README for a description of the usage of the system.)

[5] These `npv/3` facts are built using the runtime improvements of the Sussex Prolog DATR implementation, Version 2.00, described in Jenkins (1990). Specifically, this procedure heavily relies on the notion of *relevant information*, explained in section 2.3; i.e., only those nodes are considered which are not "hidden" (`#hide`) and those paths which are to be "shown" (`#show`).

```
npv('PLAY_V',[syn,cat],v).
npv('PLAY_V',[syn,subcat,first,syn,cat],np).
npv('PLAY_V',[syn,subcat,rest],nil).
npv('PLAY_V',[mor,num,sing,pers,1],play).
npv('PLAY_V',[mor,num,sing,pers,2],play).
npv('PLAY_V',[mor,num,sing,pers,3],plays).
npv('PLAY_V',[mor,num,plur,pers,1],play).
npv('PLAY_V',[mor,num,plur,pers,2],play).
npv('PLAY_V',[mor,num,plur,pers,3],play).
npv('PLAY_V',[path1,gapin,path2,gapout],path_unify).
npv('PLAY_N',[syn,cat],n).
npv('PLAY_N',[mor,num,sing,pers,3],play).
npv('PLAY_N',[mor,num,plur,pers,3],plays).
npv('PLAY_N',[path1,gapin,path2,gapout],path_unify).
```

There are a couple of other advantages which you get for free when using this method:

- you can leave out all the paths which have no specified value (keyword: `unspecified`) and get a compact DATR lexicon containing only the information you want to provide for a query from the PATR system
- you need not use the DATR system and its evaluation predicates *at all* while running the PATR system; this saves time and memory
- before starting the parser, you need not load the DATR system and the relevant DATR theories because all the interface system is looking for are the `npv/3` facts which contain all the necessary information of the respective DATR theory

The second problem is to create a dictionary entry (predicate `dict/2`) which contains exactly the amount of lexical information we want to have, i.e. from a given node–word pair: the syntactic, semantic and morphological information. Moreover, our system also has to handle path equations of the form discussed in example (b) in section 1.1 which fall in neither of these categories of information.

The solution I take is to collect *all* the information stored in a node into a list (via the standard predicate `bagof`) and then separate this list into three pieces (predicate `separate_list` in the file `queries2.pl`):

1. the morphological information corresponding to the lexical entry
2. the syntactic and semantic information
3. the path–pairs which have to be unified

After the syntactic and semantic information (2) have been assigned to a PATR tree variable and the path–pairs (3) have been unified, an "intermediate tree" is created (dynamically asserted to the Prolog database) which serves as a basis for the generation of all `dict/2` entries which have identical morphological forms but distinct morphological paths (1) (all this applies for one single node; if the same word form occurs at more than one node, the whole procedure is repeated accordingly).

In the following example, we see all the `npv/3` facts for the word form PLAY and afterwards the `dict/2` entries which the interface system creates if queried with this word form:

```
npv('PLAY_V',[mor,num,sing,pers,1],play).
npv('PLAY_V',[mor,num,sing,pers,2],play).
npv('PLAY_V',[mor,num,plur,pers,1],play).
npv('PLAY_V',[mor,num,plur,pers,2],play).
npv('PLAY_V',[mor,num,plur,pers,3],play).
npv('PLAY_N',[mor,num,sing,pers,3],play).

dict(play,[syn:[cat:v,subcat:[first:[syn:[cat:np|_A]|_B],
          rest:nil|_C],tree:play|_E],sem:[name:play|_F],
          gapin:_G,gapout:_G,mor:[num:plur,pers:3|_H]|_I]@[]).
dict(play,[syn:[cat:v,subcat:[first:[syn:[cat:np|_A]|_B],
          rest:nil|_C],tree:play|_E],sem:[name:play|_F],
          gapin:_G,gapout:_G,mor:[num:plur,pers:2|_H]|_I]@[]).
dict(play,[syn:[cat:v,subcat:[first:[syn:[cat:np|_A]|_B],
          rest:nil|_C],tree:play|_E],sem:[name:play|_F],
          gapin:_G,gapout:_G,mor:[num:plur,pers:1|_H]|_I]@[]).
dict(play,[syn:[cat:v,subcat:[first:[syn:[cat:np|_A]|_B],
          rest:nil|_C],tree:play|_E],sem:[name:play|_F],
          gapin:_G,gapout:_G,mor:[num:sing,pers:2|_H]|_I]@[]).
dict(play,[syn:[cat:v,subcat:[first:[syn:[cat:np|_A]|_B],
          rest:nil|_C],tree:play|_E],sem:[name:play|_F],
          gapin:_G,gapout:_G,mor:[num:sing,pers:1|_H]|_I]@[]).
dict(play,[syn:[cat:n,tree:play|_E],sem:[name:play|_F],
          gapin:_G,gapout:_G,mor:[num:sing,pers:3|_H]|_I]@[]).
```

Remark 1: The system uses the following reserved words (to be encoded in the DATR theory):

- `unspecified`: value which is not specified (this will not be included in the list of `npv/3` facts)
- `path1, path2`: path-delimiters for path unification purposes
- `path_unify`: dummy value for path unification purposes
- `mor`: header for morphological paths

Remark 2: The morphological paths consist of the header (`mor`) and a sequence of attribute–value pairs. (This was done for simplicity of the algorithm which constructs the morphological information for the PATR tree. In principle, one could easily extend the system to more complex morphological paths.)

The `dict/2` predicates are dynamically asserted to the Prolog database so that all of them are available to the parser for backtracking or subsequent queries.

These interface procedures are triggered whenever the PATR system does *not* find a relevant `dict/2` entry in the database and therefore has to employ the generic entry (calling the query predicate) which looks like that:

```
dict( Word, Tree@[d2p_query(Tree, Word)] ).
```

If the parser reaches this generic entry *although* the current word has already been queried, then this means that all possibilities of the current lexical item have been checked without success and the parser has to backtrack to previous choice points (predicate `check_prev_query` in the file `queries2.pl`).

To summarise: In this variant of the system, whenever the parser does not find a word in the (dynamic) `dict/2` database, it executes a query to the precompiled `npv/3` lexicon and in the course of this, all relevant `dict/2` entries are created and asserted to the Prolog database. (Initially, there is *only* the generic `dict/2` entry in the database (included in the PATR file).)

### 4.2 Using a precompiled dictionary in PATR format

An alternative variant of our system, which saves time but (generally) consumes more memory, is to create all the possible `dict/2` entries from the `npv/3` facts beforehand and to use them directly, quasi as a PATR style lexicon. Of course, here you will find no elimination of redundancies whatsoever but the point is to use a DATR theory as a basic, elegant pattern for the representation of the lexicon and to do the compilation to form the PATR `dict/2` entries just before you use the PATR grammar and parser.

In this variant, there is no generic `dict/2` entry because there is no need to construct (new) `dict/2` entries any more.

### 4.3 Various useful predicates of the system

The most important predicate is `d2p_query/2` in the file `queries2.pl` which (together with its subpredicates) does all the work of creating `dict/2` entries from the `npv/3` predicates.

In the file `m_misc2.pl`, there are predicates for

- performing a demo run (`show_demo`)
- parsing sentences (`m_parse`)
- precompiling a DATR theory (`datr2db`)
- reading or writing `npv/3` facts from/to a file (`read_datr_db`, `datr2file`)
- reading or writing `dict/2` entries from/to a file (`read_dict_db`, `dicts_to_file`)
- switching from "dict"–mode to "npv"–mode (`npv_mode`)

## 5 Summary and outlook

In this paper, I have discussed an implementation of an interface system which allows a runtime access from a PATR grammar/parser to a precompiled DATR lexicon. The system operates either on a DATR style representation or on a full-fledged PATR suitable dictionary compiled from a DATR theory.

Investigations concerning the time and space complexity of this system and for other recently proposed approaches (theoretically as well as with examples of a realistic size) still remain to be issues of future research.

## 6 References


Duda, M. & Gebhardi, G. (1994): DUTR — a DATR–PATR Interface Formalism. In *Trost (1994)*, pp. 411-414

Erjavec, T. (1992): *Treatments of Slovene verb morphology in inheritance models.* MSc Thesis, Centre for Cognitive Science, University of Edinburgh.

Evans, R. & Gazdar, G. (eds.) (1990): The DATR Papers. In *Cognitive Science Research Paper* CSRP 139, University of Sussex, Brighton.

Gazdar, G. (1989): An Introduction to DATR. In *Evans & Gazdar (1990)*, pp. 1-14

Jenkins, E. A. (1990): Enhancements to the Sussex DATR implementation. In *Evans & Gazdar (1990)*, pp. 41-61

Kilbury, J., Naerger, P., Renz, I. (1991): DATR as a lexical component of PATR. In *Proceedings of the 6th Annual Meeting of the EACL*, pp. 137-141

Langer, H. (1994): Reverse Queries in DATR. In *Proceedings of COLING-94*, pp. 1089-95

Trost, H. (ed.) (1994): KONVENS '94. Berlin: Springer.

Zechner, K. (1995): *Runtime Access from a PATR-II Grammar to Lexical Information in DATR.* Project Report, Centre for Cognitive Science, University of Edinburgh.

# A Listings of the source code files

## A.1 File README

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                 %
% CL2 - term paper - 7 April 1995                                 %
%                                                                 %
% Klaus Zechner <klaus@cogsci>                                    %
%                                                                 %
% file: README   :  contains a list of the files in this          %
%                   directory and a short description of what    %
%                   you can do with the system and how to use it %
%                                                                 %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

List of Files:
**************

   README        : this file

   music2.dtr    : DATR-lexicon for music instruments plus additional
                   lexical entries of several categories

   m_gram2       : PATR-grammar which does on-line calls to DATR
                   (resp. to a database of precompiled clauses or
                    to a precompiled dictionary)

   m_gram2.links : link information, built from m_gram2

   queries2.pl   : the DATR-to-PATR query predicate

   m_misc2.pl    : predicates for the demo, file-io,
                   the call for the parser and other stuff

   music_init.pl : initialization-module

   m_demo2       : demo-file with sample sentences

   cl2tp.tex     : LaTeX source code of term paper (contains a
                   more extensive documentation section)

   datr_init.pl  : initialization of DATR (which is not necessary
                   for a standard run of the system for the queries
                   either go to a precompiled set of npv/3 clauses
                   or a precompiled set of dict/2 clauses is used)
```

```
   music2.npv    : precompiled npv/3-clauses (info from DATR about
                   node-path-value triples, used file music2.dtr)

   music2.dict   : precompiled dict/2-clauses (from DATR file
                   music2.dtr; can be consulted as a lexicon for
                   a PATR grammar; it has the suitable format)

Usage of the System:
********************

  At the PROLOG-prompt, just type:

   | ?- [music_init].

  ... and after a while, all the files will have been loaded and
  you will be given an information how to start a small demo, i.e.
  typing:

   | ?- show_demo.


  After having seen these examples you can write your own sentences
  using the predicate m_parse:

   | ?- m_parse.

  ... or you can switch to the off-line mode which is a bit faster
  (it uses a precompiled dictionary) but (possibly) consumes more
  memory:

   | ?- read_dict_db('music2.dict').

  (Type y and RETURN when asked to redefine the dict/2 predicate.)


  In order to switch back to on-line mode (which creates all dict/2
  entries during runtime of the parser as needed), type:

   | ?- npv_mode('music2.npv').


List of Words in the DATR-lexicon:
**********************************

determiners:
  a, the
```

```
nouns:
  violin, viola, cello, contrabass (sing/plur)
  contrabass_1948b, contrabass_1909a,
  guarneri_1679a, stradivari_1657b (sing)
  lb_3500, lb_20000 (plur)
  play (sing/plur) (n/v ambiguity)

pp's:
  under_chin, on_ground

verbs:
  cost [subcat n]
  is_played [subcat pp]
  play [subcat np] (n/v ambiguity)

pronouns:
  i, you, it, we, they

 example sentences:
  a play costs lb_3500.
  the stradivari_1657b costs lb_20000.
  the contrabass_1909a is_played on_ground.
  the contrabasses cost lb_3500.
  i play the violin.

=============== end of file: README =================================
```

### A.2 File music2.dtr

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                %
% CL2 - term paper - 7 April 1995                                %
%                                                                %
% Klaus Zechner <klaus@cogsci>                                   %
%                                                                %
% file: music2.dtr: DATR definition of music instruments         %
%                                                                %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% In this file, a hierarchy of instruments is encoded.
% Default-inheritance and exceptions are used.
% Furthermore, some additional lexical entries are encoded for
% the use of a PATR-II grammar.
%
% reserved words:
%  unspecified: feature which is not specified (will not be included
%               in npv-list/dict/list)
%  path1,path2: path-delimiters for path unification purposes
%  path_unify : path unification pseudo (dummy) value
%  mor        : header for morphological paths
%
%
% features of the instruments
%
%  number       : individual (stock) number
%  type         : to which group of instrument it belongs
%  material     : material of instrument (characteristic)
%  position     : where it is played (on ground/under chin etc.)
%  activity     : what the player is doing
%  played_with  : what the player is using
%  sound_range  : general range (high/low)
%  ground_tone  : basic tone of instrument
%  price        : price of individual instrument (in pounds)
%  size         : size (in litre)
%
%  syntactical features:
%  cat          : word (phrase) category
%  subcat       : subcategorization info
%
%  morphological features:
%  num          : number (sing/plur)
%  pers         : person (1/2/3)

% reset definitions:
```

```
#reset.

% declare variables:
#vars $semfeat: number type material position activity
                played_with sound_range ground_tone price size.
#vars $num: sing plur.
#vars $pers: 1 2 3.

#hide INSTRUMENT STRINGS STRING_MAJ STRING_MIN
      WORD VERB VERB_NCOMP VERB_NPCOMP VERB_PPCOMP NOUN PP DET PRONOUN.

% word classes:

WORD:           <> == unspecified
                <mor num $num pers $pers> == "<mor $num $pers>"
                <path1 gapin path2 gapout> == path_unify.

VERB:           <> == WORD
                <syn cat> == v
                <mor sing 1> == "<root>"
                <mor sing 2> == "<root>"
                <mor sing 3> == ("<root>" s)
                <mor plur 1> == "<root>"
                <mor plur 2> == "<root>"
                <mor plur 3> == "<root>".

VERB_NCOMP:     <> == VERB
                <syn subcat first syn cat> == n
                <syn subcat rest> == nil.

VERB_NPCOMP:    <> == VERB
                <syn subcat first syn cat> == np
                <syn subcat rest> == nil.

VERB_PPCOMP:    <> == VERB
                <syn subcat first syn cat> == pp
                <syn subcat rest> == nil.

NOUN:           <> == WORD
                <syn cat>  == n
                <mor sing 3> == "<root>"
                <mor plur 3> == (<mor sing 3> s ).

DET:            <> == WORD
                <syn cat> == det.

PP:             <> == WORD
```

```
                <syn cat> == pp.

PRONOUN:        <> == WORD
                <syn cat> == pron.


% instrument - main node:

INSTRUMENT:     <> == NOUN
                <sem $semfeat> == ("<$semfeat>").

% instrument groups (strings-hierarchy):

STRINGS:        <> == INSTRUMENT
                <type>       == strings
                <material>   == wood
                <activity>   == bow
                <played_with>== bow.

STRING_MAJ:     <> == STRINGS
                <position>   == on_ground
                <sound_range>== low.

STRING_MIN:     <> == STRINGS
                <position>   == under_chin
                <sound_range>== high.

% specific instruments

VIOLIN:         <>           == STRING_MIN
                <root>       == violin
                <ground_tone>== g_little
                <size>       == ltr_30.

VIOLA:          <>  == STRING_MIN
                <root>       == viola
                <ground_tone>== c_little
                <size>       == ltr_45.

CELLO:          <>           == STRING_MAJ
                <root>       == cello
                <ground_tone>== c_big
                <size>       == ltr_100.

CONTRABASS:     <>           == STRING_MAJ
                <root>       == contrabass
                <mor plur 3> == contrabasses
```

```
                <ground_tone>== e_contra
                <size>       == ltr_700.


% individual instruments

CONTRABASS_1948B: <>         == CONTRABASS
                <root>       == contrabass_1948b
                <number>     == nr_12377810
                <mor plur 3> == unspecified  % single object
                <price>      == lb_5000.

CONTRABASS_1909A: <>         == CONTRABASS
                <root>       == contrabass_1909a
                <number>     == nr_17667678
                <mor plur 3> == unspecified  % single object
                <ground_tone>== c_contra  % override-feature
                <price>      == lb_5000.

GUARNERI_1679A: <>           == VIOLIN
                <mor sing 3> == guarneri_1679a
                <mor plur 3> == unspecified  % single object
                <number>     == nr_12133311
                <price>      == lb_20000.

STRADIVARI_1657B: <>         == VIOLIN
                <mor sing 3> == stradivari_1657b
                <mor plur 3> == unspecified  % single object
                <number>     == nr_14232456
                <price>      == lb_22000.

% determiners:

A:              <> == DET
                <sem type>    == indef_det
                <mor sing 3>  == a
                <mor plur 3>  == unspecified.

THE:            <> == DET
                <sem type>    == def_det
                <mor sing 3>  == the
                <mor plur 3>  == unspecified.

THE_PL:         <> == DET
                <sem type>    == def_det
                <mor sing 3>  == unspecified
                <mor plur 3>  == the.
```

```
% verbs:

COST:           <> == VERB_NCOMP
                <root> == cost.


PLAY_V:         <> == VERB_NPCOMP
                <root> == play.

IS_PLAYED:      <> == VERB_PPCOMP
                <mor sing 1> == unspecified
                <mor sing 2> == unspecified
                <mor sing 3> == is_played
                <mor plur 1> == unspecified
                <mor plur 2> == unspecified
                <mor plur 3> == are_played.


% nouns:

PLAY_N:         <> == NOUN
                <root> == play.

LB_3500:        <> == NOUN
                <sem price> == lb_3500
                <mor plur 3> == lb_3500
                <mor sing 3> == unspecified.

LB_20000:       <> == NOUN
                <sem price> == lb_20000
                <mor plur 3> == lb_20000
                <mor sing 3> == unspecified.

% pp's:

UNDER_CHIN:     <> == PP
                <mor sing 3> == under_chin.

ON_GROUND:      <> == PP
                <mor sing 3> == on_ground.


% (personal) pronouns:

I:              <> == PRONOUN
                <mor sing 1> == i.
```

```
YOU:            <> == PRONOUN
                <mor sing 2> == you
                <mor plur 2> == <mor sing 2>.

WE:             <> == PRONOUN
                <mor plur 1> == we.

THEY:           <> == PRONOUN
                <mor plur 3> == they.

IT:             <> == PRONOUN
                <mor sing 3> == it.

% specify paths which are to be shown in a general query:

#show  <syn cat>
       <syn subcat first syn cat> <syn subcat rest>
       <sem type> <sem number> <sem material> <sem position>
       <sem activity> <sem played_with> <sem sound_range>
       <sem ground_tone> <sem price> <sem size>
       <mor num sing pers 1>
       <mor num sing pers 2>
       <mor num sing pers 3>
       <mor num plur pers 1>
       <mor num plur pers 2>
       <mor num plur pers 3>
       <path1 gapin path2 gapout>.

%%%%%%%%%%%%%%%%%%%%%%%%% END OF FILE %%%%%%%%%%%%%%%%%%%%%%%%%%%
```

### A.3 File m_gram2

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                %
% CL2 - term paper - 7 April 1995                                %
%                                                                %
% Klaus Zechner <klaus@cogsci>                                   %
%                                                                %
% file: m_gram2:a small PATR-II grammar for accessing DATR in    %
%               the music-example (music2.dtr)                   %
%                                                                %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% This grammar just allows the parsing of very simple declarative
% sentences. The point of it, however, is to demonstrate the use
% of the DATR-PATR-interface, which in this example queries the
% lexical information from DATR for all occuring words in
% the parse and stores it in the resulting PATR-tree (the lexical
% information is stored in the database, npv/3-facts).
%
% note: for example sentences see file README (or the term paper)


% dict/2 has to be dynamic; at each query, the relevant
% dict-entries are added to the database (if not there):
:- dynamic dict/2.


% grammar:

%  rules : s -> np vp
%          vp -> v [subcat]
%          np -> det n
%          np -> pron
%
%  note: three types of information is passed on:
%          syn : syntactic structure
%          sem : semantics
%          mor : morphology (the resulting mor-info
%                at the s node represents the projection
%                of the verbal head)

X0 ---> [X1,X2]@[X0/syn/cat <=> s,
                 X1/syn/cat <=> np,
                 X2/syn/cat <=> vp,
                 X2/syn/subcat <=> nil,
                 X1/mor <=> X2/mor,
```

```
                 X0/mor <=> X1/mor,
                 X0/syn/tree/np <=> X1/syn/tree,
                 X0/syn/tree/vp <=> X2/syn/tree,
                 X0/sem/subj <=> X1/sem,
                 X0/sem/pred <=> X2/sem].

X0 ---> [X1,X2]@[X0/syn/cat <=> vp,
                 X1/syn/cat <=> v,
                 X1/syn/subcat/first <=> X2,
                 X0/syn/subcat <=> X1/syn/subcat/rest,
                 X0/mor <=> X1/mor,
                 X0/syn/tree/verb <=> X1/syn/tree,
                 X0/syn/tree/comp <=> X2/syn/tree,
                 X0/sem <=> X1/sem,
                 X0/sem/arg <=> X2/sem].

X0 ---> [X1,X2]@[X0/syn/cat <=> np,
                 X1/syn/cat <=> det,
                 X2/syn/cat <=> n,
                 X0/mor <=> X1/mor,
                 X0/mor <=> X2/mor,
                 X0/syn/tree/det <=> X1/syn/tree,
                 X0/syn/tree/n <=> X2/syn/tree,
                 X0/sem/det <=> X1/sem,
                 X0/sem/ent <=> X2/sem].

X0 ---> [X1]@[X0/syn/cat <=> np,
                 X1/syn/cat <=> pron,
                 X0/mor <=> X1/mor,
                 X0/syn/tree/pron <=> X1/syn/tree,
                 X0/sem/perspro <=> X1/sem].



% lexicon:

% just one entry which queries all the relevant DATR-info;
% the last relevant dict-entry which was produced
% is returned to the search here:

dict( Word, Tree@[d2p_query(Tree, Word)] ).

% start symbol and link frame for lcp:

start_symbol(X) :-
        X/syn/cat <=> s.
```

```
link_frame(X,C) :-
        X/syn/cat <=> C.

%%%%%%%%%%%%%% END OF FILE %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

### A.4 File queries2.pl

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                 %
% CL2 - term paper - 7 April 1995                                 %
%                                                                 %
% Klaus Zechner <klaus@cogsci>                                    %
%                                                                 %
% file: queries2.pl:specific query predicates for the PATR/DATR  %
%                   interface                                     %
%                                                                 %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% declared was_queried/1 as dynamic (says whether this word
% has already been queried via d2p_query):
:- dynamic was_queried/1.

%%%
%%% main predicates:
%%%
%%%  d2p_query/2 (DATR-to-PATR query with PATR-tree and PATR-word;
%%%               all matching DATR values are retrieved in the PROLOG
%%%               database (npv/3-clauses) and dict/2 entries are built
%%%               from them at runtime of the parser)
%%%

%%%
%%% d2p_query(-PatrTree,+PatrWord)
%%%
%%% task: creates all the dict/2 entries according to the
%%%       current database (npv/3 facts, derived from a DATR theory)
%%%

d2p_query(PatrTree,PatrWord) :- !,
  % check if word was already queried:
     check_prev_query(PatrWord,CheckResult),
    (CheckResult == ok -> % continue (else: nothing to do)
        execute_query(PatrTree,PatrWord) ).

%%%
%%% execute_query(-PatrTree,+PatrWord)
%%%
%%% task: does the actual work for d2p_query
%%%

execute_query(PatrTree,PatrWord) :-
  % set query-switch on for subsequent queries:
```

```
     asserta( was_queried(PatrWord) ),

 % debug-info:
 % nl,write('queried for: '),write(PatrWord),nl,

  % retract possible remaining intermediate PATR-trees
  % from previous calls/runs:
    retractall( intermed_tree(_) ),!,
  % select all Nodes which are relevant to the PATR word i.e. which have
  % a path that 'leads' to a value which is identical to this word (when
  % concatenated); collection is made from the PROLOG-database, so you
  % either have to use datr2db/0 or datr_read_db/1 to build up the
  % required DB (npv/3) before you start the parse
     setof(DatrNode,Path^(npv(DatrNode,Path,PatrWord)),NodeList),
     handle_nodes(NodeList,PatrWord),
  % assign the infos of the first dict/2 entry previously created
  % to the PATR tree
     !,dict(PatrWord, PatrTree@[]).

%%%
%%% check_prev_query(+Word,-CheckResult)
%%%
%%% task: checks if word was already queried; if this is the case
%%%       that means that the parser has reached the generic dict/2
%%%       entry during backtracking and therefore it has to fail
%%%

check_prev_query(Word,prev_query_found) :-
  was_queried(Word), % all alternatives must have been
                     % tried without success: fail

  % debug info:
  % nl,write('*** All possible entries for the word: '),
  % write(Word),nl,write('*** have been tried without success.'),
  % nl,write('*** -> Parser tries backtracking...'),nl,

  !,fail.

check_prev_query(_,ok) :- !. % else (not yet queried): o.k.

%%%
%%% handle_nodes(+NodeList,+PatrWor)
%%%
%%% task: for all nodes in the node list: build up the PATR trees
%%%       and create the dict/2 entries
%%%
```

```
handle_nodes([],_) :- !. % stop when node list is empty

handle_nodes([Head|Tail],PatrWord):-
  % collect all Path-Value-Pairs in the Attribute-Feature-List:
  % Value is atomic and therefore put into list because of the
  % list-handling in patr_assign/2:
     bagof(Path:[Value],npv(Head,Path,Value),AttrFeatList),
  % separate list to: (a) SynSem-list (constant for one node),
  % (b) MorList (morphological paradigm of one node) and
  % (c) UnifyList (path unifications, i.e. X/PATH1 <=> X/PATH2):
     separate_list(AttrFeatList, SynSemList, MorList, UnifyList, PatrWord),
  % assign the synsem-values to the PATR-Tree:
     patr_assign(SynSemList,PatrTree),
  % assign phonology/semantic values to PATR tree:
     patr_assign([[syn,tree]:[PatrWord]],PatrTree),
     patr_assign([[sem,name]:[PatrWord]],PatrTree),
  % do the path-unifications:
     do_unifications(UnifyList,PatrTree),
  % store the tree WITHOUT mor-info in an intermediate DB:
     asserta(intermed_tree(PatrTree)),
  % build dict/2 entries for all matching mor-values:
     create_dict_entries(MorList,PatrWord),
  % remove the intermediate tree again:
     retract(intermed_tree(_)),
  % recurse:
     handle_nodes(Tail,PatrWord).

%%%
%%% separate_list(+AttrFeatList,-SynSemList,-MorList,-UnifyList,+PatrWord)
%%%
%%% task: separates the feature-value list to a synsem-list (constant
%%%       info), a mor-list (varying forms for one lexeme) and a
%%%       path-unify-list (path-unifications: X/PATH1 <=> X/PATH2)
%%%

separate_list([],[],[],[],_) :- !. % base case

separate_list([[mor|MorRest]:[DatrValue]|Tail], SynSemList,
              [[mor|MorRest]|Rest],UnifyList,DatrValue) :-
 % if the head is a 'mor'-one AND the value corresponds to
 % the PATR word, take this path to the mor-list and recurse:
 separate_list( Tail, SynSemList, Rest, UnifyList, DatrValue ).

separate_list([[mor|_]:[DatrValue]|Tail], SynSemList,
              MorList, UnifyList, PatrWord) :-
 % if the head is a 'mor'-one AND the value does NOT correspond to
 % the PATR word, forget this mor-path and recurse:
```

```
 DatrValue \== PatrWord,
 separate_list( Tail, SynSemList, MorList, UnifyList, PatrWord ).

separate_list([[path1|UnifyTail]:[_]|Tail], SynSemList, MorList,
              [[path1|UnifyTail]|Rest], DatrValue) :-
 % if the head is a 'path1'-one put the path to the UnifyList and recurse:
  separate_list(Tail, SynSemList, MorList, Rest, DatrValue).

separate_list([Head:[Value]|Tail], [Head:[Value]|Rest],
              MorList, UnifyList, PatrWord) :-
 % else: it's a synsem element and has to be put
 % to the synsem-list; then recurse:
 separate_list( Tail, Rest, MorList, UnifyList, PatrWord ).


%%%
%%% do_unifications(+UnifyList,-PatrTree)
%%%
%%% task: unifies two paths in a PATR tree
%%%

do_unifications([],_) :- !. % stop if list is empty

do_unifications([[path1|PathList]|Tail], PatrTree) :-
     unify_tree(PathList, PatrTree), % take first path-pair
     do_unifications(Tail, PatrTree). % recurse

%%%
%%% unify_tree(+PathPair,-PatrTree)
%%%
%%% task: does the work for do_unification/2
%%%

unify_tree(PathPair,PatrTree) :-
     make_two_paths(PathPair,Path1,Path2), % separate the path-pair
                                           % into the two paths
     create_patr_path(PatrTree,Path1,PatrPath1), % create the whole
     create_patr_path(PatrTree,Path2,PatrPath2), % PATR-paths
     PatrPath1 <=> PatrPath2.                    % ... and unify them

%%%
%%% make_two_paths(+PathPair,-Path1,-Path2)
%%%
%%% task: creates two paths from path pair (separated by keyword path2)
%%%

make_two_paths([path2|Path2],[],Path2) :- !. % put rest to 2nd path
```

```
make_two_paths([Head|Rest],[Head|Rest2],Path2) :- % else: take off head
                                                  % ... to 1st path
     make_two_paths(Rest, Rest2, Path2).

%%%
%%% create_dict_entries(+MorList, +DatrValue)
%%%
%%% task: creates the dict/2 entries using the morphology info in the
%%%       mor-list
%%%

create_dict_entries([],_) :- !. % base case

create_dict_entries([MorHead|MorTail], DatrValue) :-
    build_mor_list(MorHead, PatrMorList), % build list of attr-value pairs
    make_dict(DatrValue, PatrMorList),    % make dict/2 entries from this
    create_dict_entries(MorTail, DatrValue).

%%%
%%% build_mor_list(+MorList, -ResList)
%%%
%%% task: builds a list of attr-value pairs from the mor-list
%%%

build_mor_list([mor|Tail],ResList) :-
    create_pairs(Tail,ResList).

%%%
%%% create_pairs(+MorList, -MorPairList)
%%%
%%% task: does the job for build_mor_list/2: a pair
%%%       consists of the attribute [mor,attr] and
%%%       the value [val]
%%%

create_pairs([],[]) :- !. % base case: stop with empty list

create_pairs([A,B|Tail],[ [mor,A]:[B] | Rest ]) :-
    create_pairs( Tail, Rest ).  % recurse

%%%
%%% make_dict(+DatrValue,+MorPairList)
%%%
%%% task: builds a PATR tree from the info in the
%%%       mor-list (paired) and asserts a dict/2
%%%       clause to the database
```

```
%%%

make_dict(_,[]) :- !. % stop if list is empty

make_dict(DatrValue, MorPairList ) :- !,
    intermed_tree(Tree),          % get the mor-less tree from DB
    attach_mor(MorPairList,Tree), % attach all mor-info to tree
    !,asserta( dict(DatrValue, Tree@[]) ). % put dict/2 to db

%%%
%%% attach_mor(+MorPairList,-Tree)
%%%
%%% task: builds a PATR tree from the info in the
%%%       mor-list (paired)
%%%

attach_mor([],_) :- !. % base case

attach_mor([MorHead|MorTail],Tree) :-
    patr_assign([MorHead], Tree),  % assign head-info to tree
    attach_mor(MorTail, Tree).     % recurse

%%%
%%% patr_assign(+AttrFeatList, +PatrTree)
%%%
%%% task: assigns the feature/value-pairs from the
%%%       AttrFeatList to the PATR-Tree
%%%

patr_assign([],_) :- !. % stop if list is emptied

patr_assign([Head|Tail],Tree):- !,
      Head = PathList:ValueList,                % separate path/value
      create_patr_path(Tree,PathList,PatrPath), % get full PATR-path
      build_value(ValueList,AtomValue),         % build atomic value
      PatrPath <=> AtomValue,                   % do unification
      patr_assign(Tail,Tree).                   % recurse

%%%
%%% create_patr_path(+Tree,+PathList,-PatrPath)
%%%
%%% task: builds PATR-Path from PathList (DATR) and
%%%       attaches it to the Input-Tree (which might
%%%       have path specifications as well)
%%%

create_patr_path(Tree,[Path],Tree/Path) :- !.    % attach one-element-path
```

```
                                                   % to the PATR-Tree

create_patr_path(Tree,DatrPath,PatrPath/Last) :-   % append the...
      append(ShorterPath,[Last],DatrPath),         % ...last element to path
      create_patr_path(Tree,ShorterPath,PatrPath). % recurse

%%%
%%% build_value(+ValueList,-AtomValue)
%%%
%%% task: builds an atomic value (required for PATR) out of a list
%%%       of values (result from DATR-query)
%%%

build_value(ValueList,AtomValue) :-
      construct_value(ValueList, [], CharList ), % construct a char-list
      name(AtomValue, CharList).                 % concatenate it

%%%
%%% construct_value(+ValueList,+Accumulator,-CharList)
%%%
%%% task: builds a character-list from a value-list which contains
%%%       atoms using an accumulator
%%%

construct_value([],L,L) :- !. % move accumulator to result-list

construct_value([Head|Tail],Accumulator,NewOutList):-
       name(Head,HeadCharList),                % split atom/number to chars
       append(Accumulator,HeadCharList,OutList),  % append char-list to accu
       construct_value(Tail,OutList,NewOutList).  % recurse

%%%%%%%%%%%%%%%% END OF FILE %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

### A.5 File m_misc2.pl

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                %
% CL2 - term paper - 7 April 1995                                %
%                                                                %
% Klaus Zechner <klaus@cogsci>                                   %
%                                                                %
% file: m_misc2.pl:  miscellaneous predicates for the PATR/DATR  %
%                    interface - music-example (music2.dtr)      %
%                                                                %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%
%%% predicates:
%%%
%%%   show_demo/0
%%%          (shows a few example parses of the music-system)
%%%
%%%   m_parse/0
%%%          (parses a sentence read from the user input)
%%%
%%%   datr2db/0
%%%          (asserts all relevant node-path-value triples to
%%%           the PROLOG database (npv/3-facts))
%%%
%%%   datr2file/1
%%%          (calls datr2db/0 first and writes all the npv/3-facts
%%%           to a specified file)
%%%
%%%   read_datr_db/1
%%%          (reads in a file with npv/3-facts and asserts them to
%%%           the PROLOG database)
%%%
%%%   clean_DB/0
%%%          (retracts all transient stuff from PROLOG database)
%%%
%%%   dicts_to_file/1
%%%          (writes a complete PATR-useable dictonary to file)
%%%
%%%   read_dict_db/1
%%%          (reads in a PATR useable dictionary file to the database)
%%%
%%%   npv_mode/1
%%%          (switches to on-line npv/3-mode where the dictionary is
%%%           created at runtime of the parser; npv-file must be specified)
%%%
```

```
%%%
%%% show_demo/0
%%%
%%% task: reads the demo-file (m_demo2), parses its sentences,
%%%       and pretty prints it out
%%%
%%% remark: the user can stop the demo after each sentence
%%%         by hitting q (for quit) (+ return)
%%%

show_demo :-
        open('m_demo2',read,Stream),   % open demo-file
        repeat,
                read_file_line(Stream,Atomlist,EOF), % read sentence
                do_process(Atomlist),                % process it
                (EOF \== true -> fail
                ;       true),                       % stop at EOF
        !,
        close(Stream).
%%%
%%% do_process(+AtomList)
%%%
%%% task: prints out the sentence and calls the music_parse-predicate
%%%

do_process( [] ) :- !. % do nothing with empty list (blank line)

do_process( Atomlist ) :-
     nl,write(Atomlist),nl,   % print the sentence
     music_parse(Atomlist),!.   % parse it

%%%
%%% music_parse(+AtomList)
%%%
%%% task: calls the parser, pretty prints the output,
%%%       and asks the user if he/she
%%%       wants to go on
%%%

music_parse(Sent):-
        do_parse(Sent,Tree),
        Tree/syn/tree <=> Syntax,
        nl,write('Syntax:'),nl,
        pp_dag(Syntax),
        Tree/sem <=> Semantics,
        nl,write('Semantics:'),nl,
```

```
        pp_dag(Semantics),
        Tree/mor <=> Morphology,
        nl,write('Morphology:'),nl,
        pp_dag(Morphology),
        ask_user.

%%%
%%% ask_user/0
%%%
%%% task: asks the user if he/she wants to continue the demo
%%%

ask_user :-
       nl,nl,write('hit RETURN for continue or q (plus RETURN) for quit: '),
       get0(Char),
       check_continue(Char).

%%%
%%% check_continue(+Char)
%%%
%%% task: looks if char=q (no.113), then it aborts the demo;
%%%       else the demo continues.
%%%

check_continue(113) :- !, abort. % stop the demo if 'q'

check_continue(_).               % just continue)

%%%
%%% m_parse/0
%%%
%%% task: reads in a sentence from user input,
%%%       parses it, pretty prints it
%%%

m_parse :-
        parse(Tree),
        Tree/syn/tree <=> Syntax,
        nl,write('Syntax:'),nl,
        pp_dag(Syntax),
        Tree/sem <=> Semantics,
        nl,write('Semantics:'),nl,
        pp_dag(Semantics),
        Tree/mor <=> Morphology,
        nl,write('Morphology:'),nl,
        pp_dag(Morphology).
```

```
%%%
%%% datr2db/0
%%%
%%% task: writes all (current) DATR-info to PROLOG database
%%%

datr2db :- clear_NPV,
           nl,write('asserting DATR info to PROLOG database...'),
           datr_runtime(_,node(Node)),     % get a node
           \+(datr_runtime(_,hide(Node))), % node must not be declared
                                           % as hidden (via #hide)
           datr_get_NPV(Node),             % get the infos
           fail.

datr2db :- !,write('ready.'),nl.  % finished: output message

%%%
%%% clear_NPV/0
%%%
%%% task: retracts all npv/3 clauses from PROLOG database
%%%

clear_NPV :- !, retractall(npv(_,_,_)).

%%%
%%% datr_get_NPV(+Node)
%%%
%%% task: queries DATR and writes the info to the PROLOG database
%%%


datr_get_NPV(Node) :- !,
                   datr_runtime(_,show(Path)),  % take paths declared
                                                % via #show only
                   datr_query(Node,Path,Value), % perform DATR query
                   datr_writeDB(Node,Path,Value),  % write info to DB
                   fail.

datr_get_NPV(_) :- !. % stop when no more info left

%%%
%%% datr_writeDB(+Node,+Path,+Value)
%%%
%%% task: writes the info to the PROLOG database
%%%

datr_writeDB(_,_,[unspecified]) :- !. % don't put these triples to DB
```

```
datr_writeDB(Node,Path,Value) :- !,
                       build_value(Value,AtomValue),      % make atom
                       assertz(npv(Node,Path,AtomValue)). % assert at end

%%%
%%% datr2file(+FileName)
%%%
%%% task: writes the npv/3-clauses to a specified file
%%%

datr2file(FileName) :-
             datr2db,          % make npv/3-clauses
             nl,write('writing npv/3-clauses to file: '),
             write(FileName),write(' ...'),
             open(FileName,write,Stream),
             tell(FileName),
             do_write2file,    % perform the writing
             told,
             close(Stream),
             nl,
             write('All npv-clauses written to file: '),
             write(FileName),
             nl.
%%%
%%% do_write2file/0
%%%
%%% task: gets an npv/3-clause and passes it to remove_and_write/3
%%%


do_write2file :-
             npv(Node,Path,Value),
             remove_and_write(Node,Path,Value),
             fail.

do_write2file :- !.

%%%
%%% remove_and_write(+Node,+Path,+Value)
%%%
%%% task: writes the npv/3-clauses to a specified file and retracts them
%%%

remove_and_write(Node,Path,Value) :-
             !,
             retract(npv(Node,Path,Value)),
```

```
            write('npv('''),write(Node),
            write(''','),write(Path),write(','),
            write(Value),write(').'),nl.

%%%
%%% read_datr_db(+FileName)
%%%
%%% task: reads a file with npv/3 clauses to the PROLOG database
%%%

read_datr_db(FileName) :-
     clear_NPV,             % get rid of previous npv/3 clauses
     nl,write('reading npv/3-clauses from file: '),
     write(FileName),write(' into database ...'),
     open(FileName,read,Stream),
     repeat,                % loop until EOF
         read(Stream,Term),
         term_to_db(Term),  % assert the term
         (Term \== end_of_file -> fail
                    ;  true),
         !,
     close(Stream),
     nl,write('All npv/3-clauses read from file: '),
     write(FileName),nl.

%%%
%%% term_to_db(+Term)
%%%
%%% task: asserts term (npv/3 clause) to the end of the PROLOG database
%%%

term_to_db(Term) :- !, assertz(Term).

%%%
%%% clean_DB/0
%%%
%%% task: removes the transient predicates from the PROLOG database
%%%       (npv/3-clauses stay)

clean_DB :-
 % clean all transient stuff from DB:
     retractall( was_queried(_) ),
     retractall( intermed_tree(_) ),
     retractall( dict(_,_) ),
 % add the default dict/2 entry again:
     asserta( dict(Word, Tree@[d2p_query(Tree, Word)]) ),
     nl,write('PROLOG database has been cleaned from predicates:'),
```

```
     nl,write(' was_queried/1, intermed_tree/1, dict/2 '),
     nl,write('Default dict/2 entry was asserted again.'),nl.

%%%
%%% dicts_to_file(+FileName)
%%%
%%% task: constructs a complete PATR useable dictionary from
%%%       the current npv/3 clauses and writes it to a file

dicts_to_file(FileName) :-
     clean_DB,              % remove transient stuff from db
     retractall(dict(_,_)), % get rid of default-entry
     datr2db,               % make sure we have current npv/3-clauses
  % construct the full word list from npv/3-clauses:
     setof(Word,Node^(MorPath^(npv(Node,[mor|MorPath],Word))),WordList),
     construct_dict(WordList), % construct the dictionary to db
     write_dict(FileName),     % write dictionary to file
     clean_DB.                 % create an initial state again

%%%
%%% construct_dict(+WordList)
%%%
%%% task: processes the word list and calls make_dict_entries/1
%%%

construct_dict([]) :- !. % base case: stop

construct_dict([Head|Tail]) :-
     make_dict_entries(Head),
     construct_dict(Tail).

%%%
%%% make_dict_entries(+Word)
%%%
%%% task: selects relevant nodes and processes them, building
%%%       dictionary entries (dict/2) in the PROLOG database
%%%       (see file queries2.pl, pred. execute_query/2)
%%%

make_dict_entries(path_unify) :- !. % don't put these to the dictionary

make_dict_entries(Word) :-
  % retract possible remaining intermediate PATR-trees
  % from previous calls/runs:
     retractall( intermed_tree(_) ),!,
  % select all Nodes which are relevant to the PATR word i.e. which have
  % a path that 'leads' to a value which is identical to this word (when
```

```
  % concatenated)
     setof(DatrNode,Path^(npv(DatrNode,Path,Word)),NodeList),
     handle_nodes(NodeList,Word).

%%%
%%% write_dict(+FileName)
%%%
%%% task: writes the dict/2 clauses to a file
%%%       (see predicate datr2file/1)
%%%

write_dict(FileName) :-
             nl,write('writing dict/2-clauses to file: '),
             write(FileName),write(' ...'),
             open(FileName,write,Stream),
             tell(FileName),
             nl,nl,
             write('% PATR-useable dictionary file: '),
             write(FileName),nl,nl,
             write('% dict/2 entries are dynamic:'),nl,
             write(':- dynamic dict/2.'),nl,nl,
             do_writedicts,
             told,
             close(Stream),
             nl,
             write('All dict/2-clauses written to file: '),
             write(FileName),
             nl.

%%%
%%% do_writedicts/0
%%%
%%% task: matches a dict/2 clause and calls remove_and_write_dict/2
%%%

do_writedicts :-
             dict(Word,Tree),
             remove_and_write_dict(Word,Tree),
             fail.

do_writedicts :- !.

%%%
%%% remove_and_write_dict(+Word,+Tree)
%%%
%%% task: writes a dict/2 clause to the file and retracts it
%%%
```

```
remove_and_write_dict(Word,Tree) :-
             !,
             retract(dict(Word,Tree)),
             write('dict('),write(Word),
             write(','),write(Tree),write(').'),nl.

%%%
%%% read_dict_db(+FileName)
%%%
%%% task: reads a file which contains dict/2 clauses to the database
%%%

read_dict_db(FileName) :-
     clean_DB,                % remove transient predicates
     retractall(dict(_,_)),   % remove default dict/2 entry
     nl,nl,
     nl,write('reading dict/2-clauses from file: '),
     write(FileName),write(' into database ...'),nl,nl,
     reconsult(FileName),     % reconsult the file
     nl,write('All dict/2-clauses read from file: '),
     write(FileName),nl.

%%%
%%% npv_mode(+FileName)
%%%
%%% task: switches to npv/3 mode (on-line) getting npv/3 clauses
%%%       from a file
%%%

npv_mode(FileName) :-
     clean_DB,               % remove transient predicates
     read_datr_db(FileName). % read in the npv/3 clauses

%%%%%%%%%%%%%%%%%%%%%%% END OF FILE %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

## A.6 File music_init.pl

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                %
% CL2 - term paper - 7 April 1995                                %
%                                                                %
% Klaus Zechner <klaus@cogsci>                                   %
%                                                                %
% file: music_init.pl: initial calls for the PATR (DATR) run     %
%                      with the info from the music-file         %
%                      (music2.dtr), read from a npv-file, as    %
%                      lookup table for the grammar (m_gram2)    %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% load the PATR files:
:- compile('/import/public/teach/msc/compling/CL2/project/parser/load_patr').

% compile the interface query file:
:- compile('queries2.pl').

% compile the file for file-io, parsing, demo and other purposes:
:- compile('m_misc2.pl').

% compile the PATR grammar and build the links:
:- build_links(m_gram2).

% load the npv/3 clauses to the PROLOG database:
:- read_datr_db('music2.npv').

% set tracer off:
:- set_tracer(off).

% inform the user how to perform a short demo and
% how to switch to off-line mode and back:
:- nl,nl,write('==============================================='),nl,nl,
   write('You can perform a short demo now if you type in:'),nl,
   write(' show_demo. '),nl,nl,
   write('You can switch to off-line mode (using a precompiled dictionary)'),
   nl,write('by typing:'),nl,
   write(' read_dict_db(''music2.dict'').'),nl,
   write(' (Type y and RETURN for redefinition of dict/2 predicate.)'),nl,nl,
   write('... and return to on-line mode (dynamic creation of dictionary'),
   nl,write('entries) by typing:'),nl,
   write(' npv_mode(''music2.npv'').'),nl,nl,
   write('==============================================='),nl,nl.

%%%%%%%%%%%% END OF FILE %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

### A.7 File datr_init.pl

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                %
% CL2 - term paper - 7 April 1995                                %
%                                                                %
% Klaus Zechner <klaus@cogsci>                                   %
%                                                                %
% file: datr_init.pl: loads DATR system,                         %
%                     datr-compiles music2.dtr and asserts       %
%                     npv/3 clauses to DB                        %
%                                                                %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% load the DATR files:
:- compile('/usr/local/share/lib/datr/load.pl').

% compile the music2.dtr file:
:- nl,nl,write('DATR compiles: music2.dtr'),nl.
:- datr_compile('music2.dtr').

% assert npv/3 clauses to PROLOG database:
:- datr2db.

%%%%%%%%%%%% END OF FILE %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

## A.8 File m_demo2

```
a play costs lb_3500.
the plays costs lb_20000.
the guarneri_1679a cost lb_3500.
the stradivari_1657b costs lb_20000.
the contrabass_1909a is_played on_ground.
the contrabass_1909a are_played on_ground.
the contrabasses cost lb_3500.
i play the violin.
```

## B Sample (demo) run of the system

Note: The following demo run shows the parse of the eight sentences in the demo file `m_demo2`. Those sentences which cannot be parsed due to feature clashes are just listed without the relevant trees.

```
| ?- [music_init].

 % ... all files are loaded ...

| ?- show_demo.

[a,play,costs,lb_3500]

Syntax:

np : det : a
     n : play
vp : verb : costs
     comp : lb_3500

Semantics:

subj : det : type : indef_det
             name : a
       ent : name : play
pred : name : costs
       arg : price : lb_3500
             name : lb_3500

Morphology:

num : sing
pers : 3


hit RETURN for continue or q (plus RETURN) for quit:

[the,plays,costs,lb_20000]

[the,guarneri_1679a,cost,lb_3500]

[the,stradivari_1657b,costs,lb_20000]

Syntax:
```

```
np : det : the
     n : stradivari_1657b
vp : verb : costs
     comp : lb_20000

Semantics:

subj : det : type : def_det
             name : the
       ent : type : strings
             number : nr_14232456
             material : wood
             position : under_chin
             activity : bow
             played_with : bow
             sound_range : high
             ground_tone : g_little
             price : lb_22000
             size : ltr_30
             name : stradivari_1657b
pred : name : costs
       arg : price : lb_20000
             name : lb_20000

Morphology:

num : sing
pers : 3


hit RETURN for continue or q (plus RETURN) for quit:

[the,contrabass_1909a,is_played,on_ground]

Syntax:

np : det : the
     n : contrabass_1909a
vp : verb : is_played
     comp : on_ground

Semantics:

subj : det : type : def_det
             name : the
       ent : type : strings
```

```
             number : nr_17667678
             material : wood
             position : on_ground
             activity : bow
             played_with : bow
             sound_range : low
             ground_tone : c_contra
             price : lb_5000
             size : ltr_700
             name : contrabass_1909a
pred : name : is_played
       arg : name : on_ground

Morphology:

num : sing
pers : 3


hit RETURN for continue or q (plus RETURN) for quit:

[the,contrabass_1909a,are_played,on_ground]

[the,contrabasses,cost,lb_3500]

Syntax:

np : det : the
     n : contrabasses
vp : verb : cost
     comp : lb_3500

Semantics:

subj : det : type : def_det
             name : the
       ent : type : strings
             material : wood
             position : on_ground
             activity : bow
             played_with : bow
             sound_range : low
             ground_tone : e_contra
             size : ltr_700
             name : contrabasses
pred : name : cost
       arg : price : lb_3500
```

```
             name : lb_3500

Morphology:

num : plur
pers : 3


hit RETURN for continue or q (plus RETURN) for quit:

[i,play,the,violin]

Syntax:

np : pron : i
vp : verb : play
     comp : det : the
            n : violin

Semantics:

subj : perspro : name : i
pred : name : play
       arg : det : type : def_det
                   name : the
             ent : type : strings
                   material : wood
                   position : under_chin
                   activity : bow
                   played_with : bow
                   sound_range : high
                   ground_tone : g_little
                   size : ltr_30
                   name : violin

Morphology:

num : sing
pers : 1


hit RETURN for continue or q (plus RETURN) for quit:

yes
| ?-
```