gefördert vom
Bundesministerium
für Forschung
und Technologie

Verbmobil
Verbundvorhaben

From DATR to PATR via DUTR – an Interface Formalism

Markus Duda

HUB

Report No. 17
July 1994

July 1994

Markus Duda

Computerlinguistik
Institut für deutsche Sprache und Linguistik
Philosophische Fakultät II
Humboldt-Universität zu Berlin
Unter den Linden 6
10099 Berlin

Tel.: (030) 20192 - 555
e-mail: `Duda@compling.hu-berlin.de`

Gehört zum Antragsabschnitt: 5.2/4/5 Lexikonauswertung, -formalismus, -werkzeuge

Das diesem Bericht zugrundeliegende Forschungsvorhaben wurde mit Mitteln des Bundesministers für Forschung und Technologie unter dem Förderkennzeichen 01 IV 101 G gefördert. Die Verantwortung für den Inhalt dieser Arbeit liegt bei dem Autor.

Contents

1 Introduction

There are a number of different formalisms used in the field of computational linguistics like PATR as a unification formalism or DATR as an inheritance based one. One way to partition formalisms is to distinguish between process oriented formalisms and representation oriented formalisms. PATR is process oriented since it is based on unification. The unification operation is binary and closed[1]. This makes it possible to construct arbitrary sequences of unifications to define an algorithm, e.g. the procedure of how to construct a parse tree from a set of input signs. That is why, we can call the unification a procedural operation, and a unification formalism is process oriented.

DATR, on the other hand, is a theorem solver. There is no other operation than the built in inference strategy which can not be changed by the user. Thus, the DATR formalism is declarative. Since DATR uses an inheritance mechanism with defaults, it is very simple to define a hierarchy of linguistic objects and it is easy to create new objects from existing ones by similarity. By the help of DATR a linguist can simply model rules with exeptions. DATR can be refered to as representation oriented. It is designed to represent linguistic data.

Some unification formalisms make use of an inheritance based type hierarchy. By the help of a default unification operation these formalism are able to express similarity and rules with exeptions as well [18]. The advantage is that a linguist has to cope with just one formalism. The drawbacks are that firstly, there is no such formalism commonly used and secondly, these formalisms are fairly complex.

DATR and PATR are very simple and powerful, DATR for representing linguistic data and PATR for processing them. It is useful to combine both formalisms to obtain one tool for linguistic work.

2 Related works

Some related work exists concerning the problem of connecting DATR to feature term formalisms. The first approach we consider is described in [1]. The

[1] It works on two feature structures and results in a feature structure.

technique used there resembles the use of compilers in computer science. A tool converts a DATR lexicon into the PATR–like form of a processing formalism. The distinguishing feature of this approach is that DATR and PATR remain different processes. This interface has static character. The problem is the loss of information. Usually only a subset of information is converted from DATR to PATR, e.g. all the information related to a lexical entry, but no information about regularities between lexical items nor information about the lexeme's hierarchy are compiled out.

A second, very interesting approach [11] we consider, uses DATR itself as the compiler that produces strings with a PATR–like structure, i.e. the interface between DATR and PATR is constructed on string level. A DATR query results in a string in PATR syntax. This string describes a complex feature structure. The interface works during run time, but only in one direction from DATR to PATR. Since the string representing a feature structure, has to be reinterpreted into PATR, no really dynamic interaction between the DATR inference and the PATR unification is possible. The idea of constructing complex feature structures within of DATR is nice and can be used in the given approach as well.

Common to [1] and [11] are that they both use DATR to encode the lexicon and that the information flow is only directed from DATR to PATR, not the other way around.

3 Dynamic interaction between DATR and PATR

3.1 Basic idea

In order to keep all information given in a DATR theory accessible at run time, a dynamic link between DATR and PATR is defined. This approach may lead to a minimal redundance of information which has to be represented in both formalisms, DATR and PATR.

The idea is to think of a DATR query as of a relation on feature structures within a feature structure.

For that purpose three small PATR extensions are introduced, lists of feature structures, disjunctions of feature structures, and the DATR pattern query as a relation on feature structures. On the other hand, DATR is extended with the DATR pattern query which completes a partially given DATR extensional sentence. Next, this DATR implementation allows structured values. It distinguishes between atomic values, value lists, and lists of lists.

3.2 Complex DATR values

Usually, DATR values are simple lists, maybe consisting of only one element, but they are lists. If a descriptor is a list element and it evaluates into a list itself then this list is concatenated in between the surrounding list and the result is a simple list. This makes sense since a value may become a path descriptor and a descriptor can only be a simple list of attributes.

$$value \ ::=^2 \ (atom^*)$$

On the other hand, it would be nice to have complex data structures to work with. The HUB-DATR[3] implementation makes use of structured lists as complex DATR values. It is the user's responsebility to avoid that complex DATR values are interpreted as descriptors during the inference process of DATR.

$$\begin{array}{lcl} value & ::= & atom \mid vlist \\ vlist & ::= & (\ value^*\ ) \end{array}$$

By the help of complex DATR values it is possible to construct recursive data structures, as feature structures are, to dynamically link PATR with DATR. A dynamic link requires the interchange of linguistic information between DATR and PATR not only on string level, but on the level of real data structures. Thats why we designed feature structures in PATR as well as complex values in DATR as structured lists.

[2] formula in EBNF
[3] DATR of Humboldt-University Berlin

3.3 The DATR pattern query

In order to derive information from a DATR theory, a *DATR query* [7] is resolved, i.e. a value of a given node path pair is inferred from the theory. The DATR query can be seen as a special case of an *extensional* DATR *sentence*[7]. The *extensional DATR sentence* is defined as an equation with a node path pair at its left hand side and a value at its right. (A *extensional DATR sentence* can be referred to as a theorem to be proved over a given DATR theory.) If the value is variable, the *extensional DATR sentence* represents a DATR query. (A variable position in an *extensional DATR sentence* extends the definition by *Gazdar/Evans* [7].)

Making use of another variable position in an *extensional DATR sentence*, *Langer* [13] introduced the technique of the *reverse DATR query* where the path is variable. Informally, from a node and a value of an *extensional DATR sentence*, the *reverse DATR query* resolves the path which holds the value at that node.

As an example, for the simple DATR theory

$$\begin{array}{lll}
Noun: & < orth > == & (" < stem > " \; " < ending > ") \\
 & < stem > == & " < root > " \\
 & < ending > == & () \\
 & < gender > == & masc. \\
 & & \\
N_er: & < stem\ plur > == & (" < root > " \; e \; r) \\
 & < ending\ sing\ gen > == & (e \; s) \\
 & < ending\ plur\ dat > == & (n). \\
 & & \\
Bild: & < root > == & (b \; i \; l \; d) \\
 & < gender > == & neut.
\end{array}$$

an *extensional DATR sentence* is

$$Bild :< orth\ sing\ gen > = \; (b \; i \; l \; d \; e \; s)$$

The *DATR query* for that extensional sentence is

$$Bild :< orth\ sing\ gen > =?$$

and the *reverse DATR query* is

$$Bild: \$\mathbf{Path} \;=\; (b\ i\ l\ d\ e\ s)$$

where *$Path* marks the variable position which is to be filled with a path valid for the node *Bild* and the value *(b i l d e s)*.

In general, given the set N of nodes , the set P of paths, and the set V of values of a DATR theory, the set E of all possible *extensional* DATR *sentences* over that DATR theory is the relation

$$E \;\subseteq\; N \times P \times V$$

Note, for the DATR default inference, E is either empty or infinite. Another interesting property of DATR is that it is also possible to define E as a function: $E_f : (N \times P) \;\rightarrow\; V$

Since the path of a DATR extensional sentence entails not only features but also feature values, we regard a path as consisting of a finite number of elements and not as a single element of an extensional sentence.

Definition. A *DATR pattern query* Q over paths of length n is a relation

$$Q_n \;\subseteq\; N \times A_1 \times A_2 \times ... \times A_n \times V$$

with N - the set of nodes, A_i - the set of attributes, V - the set of values of a given DATR theory, where some places are bound to single elements or subsets of elements of its definitional sets and the other places are to be derived from the DATR theory.

To give an example, the *DATR pattern query* for the inflexion of a German noun has the following form:

$$\$Word :< orth\ \$Number\ \$Case > = \;\$Orth\_form$$

in the syntax of an *extensional DATR sentence* extended with variable positions, or

$$Q_3(\$Word,\ orth,\ \$Number,\ \$Case,\ \$Orth\_form)$$

as a relation over three element paths, where the second position is bound to the single element *orth*. A possible situation is that $\$Number$ and $\$Case$ are variable positions bound to the sets $\{sing,\ plur\}$ and $\{nom,\ gen,\ dat,\ acc\}$, and that for a particular $\$Word$ all pairs of *(number, case, form)* are to be derived from a DATR theory. An instantiation of our example pattern query is

$$Bild :< orth\ pl\ nom > =\ b\ i\ l\ d\ e\ r$$

or, as an element of Q_3

$$(Bild,\ orth,\ pl,\ nom,\ b\ i\ l\ d\ e\ r)$$

3.4 Feature structures with the DATR pattern query

In feature term formalisms, the definition of relations over feature values as an alternative to distributed disjunctions is a commonly used technique to express dependencies between the features. Combining this technique with *coreference* and *disjunction*, we can connect a DATR theory to a feature structure:

$$\begin{bmatrix} word\text{: } Bild \\ orth : \boxed{3} \\ syn : \begin{bmatrix} cat : noun \\ agr\text{: } \begin{bmatrix} gen : neut \\ num\text{: } \boxed{1} \begin{Bmatrix} sing \\ plur \end{Bmatrix} \\ cas : \boxed{2} \begin{Bmatrix} nom \\ gen \\ dat \\ acc \end{Bmatrix} \end{bmatrix} \end{bmatrix} \\ \mathbf{Bild} :< \mathbf{orth}\ \boxed{\mathbf{1}}\ \boxed{\mathbf{2}} > =\ \boxed{\mathbf{3}} \end{bmatrix}$$

The idea is that a *DATR pattern query* can be seen as a relation over feature values. Coreferences are used to define the variable positions of the *DATR pattern query*. Disjunctions bind the variable positions to a subset of all possible feature values.

Note that this connection gives a genuinely dynamic interaction between DATR inference and feature structure unification. If a unification attempt fails, a subsequent DATR inference derives new feature values, which then allow a possibly successful unification.

In the example

$$\begin{bmatrix} syn|agr|num\colon \boxed{1}\{sing,\ plur\} \\ \ldots \qquad\qquad\quad : \\ \textbf{Bild} :< \textbf{orth}\ \boxed{\mathbf{1}}\ \ \boxed{\mathbf{2}} > \ = \ \boxed{\mathbf{3}} \end{bmatrix}$$

the first variable position $\boxed{1}$ in the *DATR pattern query* is *dynamically* bound to the set of values $\{sing,\ plur\}$ of the feature *num*.

Thinking of a lexicon, the next step is to put all specific information about a lexeme into the DATR theory. The following feature structure defines an *abstract lexeme* for nouns which derives all concrete information from a DATR theory by unification and DATR inference:

$$\begin{bmatrix} word\colon \boxed{4} \\ orth\ \colon \boxed{3} \\ syn\ \ \colon \begin{bmatrix} cat : noun \\ agr\colon \begin{bmatrix} gen\ \ \colon \boxed{5}\begin{Bmatrix} masc \\ fem \\ neut \end{Bmatrix} \\ num\colon \boxed{1}\begin{Bmatrix} sing \\ plur \end{Bmatrix} \\ cas\ \ \colon \boxed{2}\begin{Bmatrix} nom \\ gen \\ dat \\ acc \end{Bmatrix} \end{bmatrix} \end{bmatrix} \\ \boxed{\mathbf{4}} :< \textbf{orth}\ \boxed{\mathbf{1}}\ \ \boxed{\mathbf{2}} > \ = \ \boxed{\mathbf{3}} \\ \boxed{\mathbf{4}} :< \textbf{gender} > \ = \ \boxed{\mathbf{5}} \end{bmatrix}$$

Using this technique, it is simple to extend this abstract lexeme with other non-syntactic features which can be described in DATR.

4 The DUTR project

DUTR[4] is an approach to join DATR with PATR into a single formalism. The connection between DATR and feature structures was realised on the bases of HUB-DATR and a slightly extended *Prolog PATR* [14]. Both formalisms are

[4]**D**efault and **U**nification **T**ree **R**epresentation

realized in pure Prolog and run as one Prolog process. Prolog variables are used to interchange data between HUB-DATR and PATR. Within a PATR feature structure free variables are bound by a DATR query predicate that derives the variable values from a given DATR theory. The backtracking algorithm of Prolog realizes that all possible variable bindings and thus the disjunction of all possible extensions of the feature structure are produced.

4.1 Operator definitions

Before you start to use DUTR, the very first thing to do is to load the operator definitions given in the file **operator.pl**. If the load fails, you may change the precedence of the operators but doing this you have to keep the relative order of precedence wrt. the builtin operators (see appendix A).

4.2 HUB-DATR

HUB-DATR is our own implementation of DATR [7] in Prolog. The idea of using pure Prolog inference to directly derive a theorem from a DATR theory was first presented by Gibbon [9].

A HUB-DATR theory consists of two parts: the standard DATR inference rules defined in **datr.pl** and an applikation theory file defined by the user. Both parts has to be loaded into Prolog, firstly **datr.pl**. When you have successfully loaded these files, HUB-DATR is ready to start.

4.2.1 The file datr.pl

The file **datr.pl** contains all predicates that are used for the DATR inference. It also contains the two DATR pattern query predicates of HUB-DATR that are the interface between DATR and the other parts.

The two easy to use Prolog predicates **datr(NodePathPair, Value)** and **ext_datr(NodePathPair, Value)** both realize a DATR pattern query. The difference is that **datr/2** behaves like a DATR query, thus producing exactly one result while **ext_datr/2** produces a set of valid results[5] using backtracking. Both predicates take two arguments, a DATR node path pair and a DATR value. A node path pair consists of a node name, a colon, and a list of attributes that mark the path. A value is an atom (a single value) or a list (a complex value). Both arguments can be partially or fully variable. Examples are:

```
?- datr(X, Y).
?- datr(N:P, V).
?- datr(bild:P, V).
?- datr(bild:[orth, Num, Cas], bildern).
...
```

The call of a DATR pattern query results, if it succeeds, in a binding of all free variables in its arguments. If the query doesn't succeed there is no solution for the pattern given by this query.

4.2.2 The DATR theory file

The structure of a DATR theory in HUB-DATR is the same as in Standard DATR. A theory consists of a set of sentences, a sentence consists of a node name and a set of equations, and each equation has a node path pair on its left hand side and a descriptor, a value, or a list of both on its right hand side. Nevertheless, the syntax of HUB-DATR is Prolog like and so it differs a little from Standard DATR (see appendix A). Since there is a lot of lexicon stuff written in Standard DATR, a compiler exists that makes the transformation from Standard DATR into HUB-DATR syntax automatically (see 4.2.5).

<table>
<tr>
<td><pre>:- noun :: [
   [orth|X] =>> *[stem|X] +
                *[end|X],
   [stem|_] =>> *[root],
   [end |_] =>> [],
   [genus] =>> masc
].</pre></td>
<td><pre>Noun:<orth> == ("<stem>"
                "<end>")
     <stem> == "<root>"
     <end> == ()
     <gen> == masc.</pre></td>
</tr>
<tr>
<td>HUB-DATR</td>
<td>Standard DATR</td>
</tr>
</table>

[5] **?- ext_datr(X, Y), fail.** results in a set of valid pairs $(x_i,\ y_i)$.

In order to realize the DATR–PATR interface, some properties of DATR are extended or changed in HUB-DATR. This changes do not influence the power of HUB-DATR, it is equal to DATR's one. Every theory, written in DATR, is convertable into HUB-DATR.

4.2.3 Differences between HUB-DATR and Standard DATR

Lists. In Standard DATR values can be seen as symbol strings or simple lists. If there is a descriptor inside the symbol string and this descriptor is evaluated into a symbol string itself, both strings are concatenated to become a symbol string again, i.e. a structured list is made flat by this operation (see 3.2).

$$\text{(a a B:<i i> c c)} \Rightarrow_{\text{DATR}} \text{(a a (b b) c c)} \Rightarrow_{\text{DATR}} \text{(a a b b c c)}$$

Since in HUB-DATR structured lists are possible DATR values, this operation is not the default case. Concatenation is made explicite in HUB-DATR by the use of the concatenation operator $+$.

$$\begin{array}{c} \texttt{[a, a]} + \texttt{+b:[i, i]} + \texttt{[c, c]} \\ \Downarrow_{\text{HUB-DATR}} \\ \texttt{[a, a]} + \texttt{[b, b]} + \texttt{[c, c]} \\ \Downarrow_{\text{HUB-DATR}} \\ \texttt{[a, a, b, b, c, c]} \end{array}$$

Descriptors. DATR distinguishes between local and global descriptors. Both can consist of a node name, of a path, or of a node path pair. Node names start with a capital letter. In HUB-DATR it is not necessary to give a node name a initial capital letter since each descriptor is marked with an unary operator.

	Standard	HUB-DATR
local	B:<i>	+b:[i]
global	”B:<i>”	*b:[i]
local	<i>	+[i]
global	”<i>”	*[i]
local	B	+b
global	”B”	*b

Default Inheritance. In Standard DATR the path extension as a mean of inheritance is the default. This means, every path not explicitely marked as unextendable can be extended by default. Since the operation of default inheritance by path extension increases the number of all possible extensional sentences of a DATR theory more than necessary, in HUB-DATR the default is changed.

Every path, not explizitly marked as extendable, is unextendable by default.

<table>
<tr><th></th><th>Standard</th><th>HUB-DATR</th></tr>
<tr><td>extendable</td><td>N:<a> == O:<b></td><td>n:[a|X] ==> +o:[b|X]</td></tr>
<tr><td>unextendable</td><td>N:<a!> == O:<b></td><td>n:[a] ==> +o:[b]</td></tr>
<tr><td>extendable</td><td>N:<> == O</td><td>n:_ ==> +o</td></tr>
</table>

Note, the extension mechanism is more flexible then the Standard DATR one. It is not only possible to enable or disable default inheritance but also to define in detail which paths at the right hand side of an equation are to be extended.

4.2.4 An example

The following example presents a description for the language of all words that consist of the concatenation of a string of **a**, a string of **b**, and a string of **c**, and all strings have the same length **n**.

$$\mathcal{L} \;=\; \{a^n b^n c^n : n \in N\}$$

Note, $\mathcal{L}$ is context sensitive.

In DATR this language is represented by the following single node description:

<table>
<tr>
<td><pre>
:- abc :: [
   [a,n|X] =>> [a] + +[a|X],
   [b,n|X] =>> [b] + +[b|X],
   [c,n|X] =>> [c] + +[c|X],
   [n|X] =>> +[a,n|X] +
             +[b,n|X] + +[c,n|X],
    _ =>> []
].
</pre></td>
<td><pre>
ABC:<>   == ()
    <a n> == (a <a>)
    <b n> == (b <b>)
    <c n> == (c <c>)
    <n>   == (<a n>
               <b n>
               <c n>).
</pre></td>
</tr>
<tr>
<td>HUB-DATR</td>
<td>Standard DATR</td>
</tr>
</table>

X is a Prolog variable used for path extension (default inheritance in DATR). In HUB-DATR the equations of a node are sorted according to the length of the path on its left hand side. Thus, the longest paths $<a\ n>$, $<b\ n>$, $<c\ n>$ come first, and $<>$ comes last.

In order to generate the string with $n = 3$, i.e. $(aaabbbccc)$, the following DATR query is asked:

$$ABC :< n\ n\ n >=?$$

In HUB-DATR the predicate **datr/2** is used for querying:

```
?- datr(abc:[n,n,n], Result).
 Result = [a,a,a,b,b,b,c,c,c]
```

The idea is that from the path $<n\ n\ n>$ three paths $<a\ n\ n\ n>$, $<b\ n\ n\ n>$, and $<c\ n\ n\ n>$ are generated. These paths evaluate to the strings (aaa), (bbb), and (ccc) that are concatenated to the result.

4.2.5 Tools

Compiler. In order to transform existing lexicon stuff in DATR into the HUB-DATR syntax, the program **d2d** exists. Use:

d2d *german*

means that from the DATR theory file *german.dtr* in Standard DATR syntax the file *german.pl* is produced. *german.pl* is the DATR theory in HUB-DATR syntax and directly loadable into Prolog.

Debugger. Since checking a DATR theory of correctness is a very sophisticated and complex task, there are additional predicates for debugging available then loading **dedatr.pl**:

- **node(***node***).** displays all equations of a given node.
- **extension(***node***).** dispalyes all equations that a given node inherits from other nodes, i.e. **extension/1** shows the inheritance path of a node.
- **d_datr(***NodePathPair, Value***).** displays the trace of the derivation of *Value* from *NodePathPair*. The trace distinguishes between local and global inheritance. It exactly shows which equation is used to replace a descriptor. Unresolveable descriptors are marked with an error at the point they occur.

4.3 PATR

Our implementation of PATR follows mainly [14]. In order to use PATR, the file **unify.pl** has to be loaded.

4.3.1 The interface to HUB-DATR

The interface to HUB-DATR is realized by the help of the HUB-DATR query predicate **datr/2**. The builtin[6] operator =*= uses this predicate to define a relation over feature values (see 3.4) as follows:

Node:Path == Value*

where *Node* is a DATR query node, *Path* is the query path, and *Value* is the result of the DATR query. In other terms, *Node:Path* is the left hand side of

[6] in PATR

a DATR extensional sentence and *Value* its right hand side. Each operand of =*= can be variable. Using variables as coreferences, other features and its values determine the inference process of a DATR query (see the feature structure at page 7).

4.3.2 Other features

Disjunctions. The infix operator =+= defines the disjunction of feature values as it is commonly used:

$$\mathit{feature} \;{=}{+}{=}\; [FS_1,\ FS_2,\ \ldots,\ FS_n]$$

This corresponds to:

$$[\mathit{feature}\ :\ \{FS_1,\ FS_2,\ \ldots,\ FS_n\}]$$

Lists. It is possible to define lists of feature structures as a feature value:

$$\mathit{feature} \;{=}{=}{=}\; [FS_1,\ FS_2,\ \ldots,\ FS_n]$$

This corresponds to:

$$[\mathit{feature}\ :\ \langle FS_1,\ FS_2,\ \ldots,\ FS_n\rangle]$$

4.3.3 An example

In chapter 3.4 the idea of an *abstract lexeme* is described. Similar to the type concept, an *abstract lexeme*, i.e. a feature structure that is not related to a concrete word but to a group of words e.g. regular nouns, defines a set of attributes appropriate for the linguistic object, the set of values appropriate for the attributes, and dependencies among the attributes.

Using the PATR part from DUTR, the feature structure of the abstract lexeme of page 8 is written as follows:

```
W ord Word :-
   W:word === Word,
   W:orth === Orth,
   W:syn:cat === noun,
   W:syn:agr:gen === Genus,
   W:syn:agr:num === Numerus,
   W:syn:agr:cas === Casus,
   %---------------------------- disjunctions
   Genus   =+= [masc, fem, neut],
   Numerus =+= [sing, plur],
   Casus   =+= [nom, gen, dat, acc],
   %------------------------------------ DATR queries
   Word:[genus]                =*= Genus,
   Word:[orth, Numerus, Casus] =*= Orth.
```

Since the HUB-DATR value lists, as well as the PATR feature structures, are realised as Prolog lists, the interchange of complex feature structures between HUB-DATR and PATR is possible. The difference between Kilbury's approach [11] and the one presented here is that Kilbury only gives the composition of feature structures in DATR and only on the level of strings. HUB-DATR interacts with PATR on the level of data structures, interchanging feature structures between them.

5 Some pros and cons

The idea of DUTR is to use two formalisms to solve different subtasks of a system for processing linguistic data. Thus, both formalism can be lean and optimized to solve their subtasks. Next, the dynamic link, i.e. an information flow between the subtasks in both directions, guaranties an adequate encoding of linguistic data and less redundancy.

However, using two different formalisms in one system is a basic problem for all users of this system. The grammarian mostly uses PATR whereas the lexicographer DATR, but both must sometimes switch to the other formalism. It is not really a problem, but it may be inconvenient to use two small powerful formalism instead of one large formalism.

A major advantage of the approach of [11] and of our approach is the direct link to the inference system of DATR. Consider for example information about compound words in DATR. If there is no direct link, it is impossible to tackle compound words with the knowledge of the DATR–theory and it is also impossible to determine all possible compound words by converting the information of the DATR–theory.

This is the disadvantage of the approach of [1], but in this way they avoid the problem that DATR becomes a time crucial process. This problem may arrise if DATR is attached to the processing formalism. Moreover, since DATR inference becomes part of the unification operation, a special efficient encoding of that operation, e.g. type unification based on table look up, is impossible.

A The syntax of HUB-DATR

```
theory ::= sentence { sentence }
sentence ::= ":-" node ":: [" equation { "," equation } "]."
equation ::= lhs " =>> " lvalue
lhs ::= "[" atom { "," atom } [ difference ] "]"
     |  variable
difference ::= "|" variable
lvalue ::= value
        |  vallist
vallist ::= "[" lvalue { "," lvalue } "]"
         | lvalue "+" lvalue    % where lvalue has to be something that
                                % evaluates to vallist
value ::= atom
       |  "*" descriptor        % global
       |  "+" descriptor        % local
descriptor ::= node
            |  [ node ":" ] path [ "+" difference ]
path ::= "[" value { "," value } [ difference ] "]"
      |  path "+" path
node ::= atom
```

atom and **variable** are Prolog atoms and variables. **difference** realizes the optional path extension.

Precedence. The precedence of the operators can explicitly be given by parenthesis:

```
abc:<a b c> =>> +(*[a] + *[b] + *[c]).
```

The precedence list is:

$$\begin{array}{c} \texttt{:} \\ \wedge \\ * \text{ and unary}(+) \\ \wedge \\ \text{binary}(+) \\ \wedge \\ \texttt{=>>} \end{array}$$

Note. The **equations** have to be ordered wrt. the path length of its left hand side: *longest path first.*

B Where to get DUTR

DUTR is available via email from the author or via ftp from *ftp.dfki.uni-sb.de: /FTP-SERVER/vm-tps/vm5/dutr.tar.gz.*

DUTR runs in UNIX and DOS environments. It is tested with Quintus-Prolog and HU-Prolog. The latter is available via ftp from *ftp.informatik.hu-berlin.de.*

If there is any question, comment, or request do not hesitate to send it to *Markus Duda, duda@compling.hu-berlin.de.*

References

[1] Andry, F.; Fraser, N. M.; McGlashan, S.; Thornton, S. and Youd, N. J. (1992): *Making* DATR *Work for Speech: Lexicon Compilation in SUNDIAL.* Computational Lingistics 18(1992)3. Assocation for Computational Linguistics.

[2] Cahill, L. J. (1993): *Morphology in the Lexicon.* In: *Proceedings of the Sixth Conference of the European Chapter of the Association for Computational Linguistics.* Utrecht.

[3] Doerre, J. (1991): *Feature Logic with Weak Subsumption Constraints.* In: *Proceedings of the 29th Annual Meeting of the Association for Computational Linguistics.* Berkeley.

[4] Emele, M. C. (1991): *Unification with Lazy Non–Redundant Copying.* In: *Proceedings of the 29th Annual Meeting of the Association for Computational Linguistics.* Berkeley.

[5] Evans, R. and Gazdar, G. (1989): *Inference in* DATR. In: *Proceedings, 4th Meeting of the European Chapter of the Association for Computational Linguistics.* Manchester.

[6] Evans, R. and Gazdar, G. (1989): *The semantics of* DATR. In: Cohn, A. (ed.): *Proceedings of the Seventh Conference of the Society for the Study of Artifical Intelligence and Simulation of Behaviour.* London, Pitman.

[7] Evans, R. and Gazdar, G. (eds.)(1990): *The* DATR *Papers.* Research Report CSPR 139, School of Cognitive and Computer Science, University of Sussex.

[8] Evans, R. (1993): *Theoretical questions about lexical access.* In: Collingham, R. J. (ed.): *Workshop on the Unified Lexicon.* Proceedings. St. Aidan's College, University of Durham.

[9] Gibbon, D. (1993): *Generalised* DATR *for flexible lexical access: Prolog Specification.* VerbMobil Report 2. Universität Bielefeld.

[10] Gibbon, D. (1993): Personal communication. Universität Bielefeld.

[11] Kilbury, J.; Naerger, P. and Renz, I. (1991): DATR *as a Lexical Component for* PATR. In: *Proceedings of the Fifth Conference of the European Chapter of the Association for Computational Linguistics.* Berlin.

[12] Kilgarriff, A. (1993): *Inheriting Verb Alternations.* In: *Proceedings of the Sixth Conference of the European Chapter of the Association for Computational Linguistics.* Utrecht.

[13] Langer, H. (1993): Personal communication. Universität Bielefeld.

[14] Gazdar, G. and Mellish, C. S. (1989): *Natural Language Processing in PROLOG.* Addison–Wesley, Wokingham.

[15] McGlashan, S.; Fraser, N. M.; Gilbert, G. N.; Bilange, E.; Heisterkamp, P. and Youd, N. J. (1992): *Dialogue Management for Telephon Information Systems.* In: *Proceedings of the 3rd Conference on Applied Natural Language Processing.* Trento.

[16] Reinhard, S. and Gibbon, D. (1991): *Prosodic Inheritance and Morphological Generalisations.* In: *Proceedings of the Fifth Conference of the European Chapter of the Association for Computational Linguistics.* Berlin.

[17] Shieber, S. M.: *An Introduction to Unification–Based Approaches to Grammar.* CSLI Lecture Notes No. 4. Stanford.

[18] Krieger, H.-U. and Nerbonne, J. (1993): *Feature-based inheritance networks for computational lexicons.* In: Brisco, T., de Pavia, V., and Copestake, A., editors: *Inheritance, Defaults, and the Lexicon.* Cambridge University Press. Cambridge.