# DATR: Eine flexible Sprache für Lexikalische Semantik

Eric Auer <eric@coli.uni-sb.de>

2. September 2002

Dieser Text ist die Ausarbeitung zu meinen Folien für das Hauptseminar Lexikalische Semantik (Sommer 2002). Er stellt eine Einführung in Motivation, Syntax und Anwendung von DATR dar. Die Grundlage von Vortrag und Ausarbeitung bildet *„DATR: A Language for Lexical Knowledge Representation" (1995)* von *Roger Evans* und *Gerald Gazdar*.

## Inhaltsverzeichnis

# Einleitung

DATR ( Abkürzung für *D*efault *AT*ribute *R*epresentation ) ist eine einfache Sprache um nichtmonotone Vererbungsnetze mit Pfad/Wert-Gleichungen zu definieren. Weil DATR einfach und konsequent strukturiert gehalten ist, lässt es sich besonders gut in Software implementieren. Die Autoren geben „eine Seite Prolog" als typische Länge einer einfachen Implementierung an.

Durch die einfache Struktur ist DATR nicht nur für Mensch und Computer gut lesbar. Auch die eher „mathematischen" Aspekte von Inferenz und deklarativer Semantik konnten genau definiert werden. Für weitergehende Informationen über die mathematischen Eigenschaften verweise ich auf *„An Evaluation Semantics for DATR Theories"* von *Bill Keller* – sie gehen über den Skopus dieses Textes hinaus.

# 1 Motivation und Beispiele für DATR und dessen Syntax

## 1.1 Ein einfaches Beispiel: loving und loved

Einer der Vorteile von DATR für die lexikalische Semantik ist die Möglichkeit, Generalisierungen kompakt auszudrücken. Betrachten wir dazu ein Beispiel in mehreren Schritten, von „ad hoc" zu „volles DATR". Angenommen wir wollen ein Lexikon von Verbformen abspeichern. Eine einfache Implementierung würde eine Datenstruktur mit z.B. den Einträgen „syntaktische Kategorie", „syntaktischer Typ", „syntaktische Form" und „morphologische Form" für jede einzelne Form anlegen:

```
Word1:
  <syn cat>   =  verb
  <syn type>  =  main
  <syn form>  =  present participle
  <mor form>  =  love ing.
Word2:
  <syn cat>   =  verb ← wie oben
  <syn type>  =  main ← wie oben
  <syn form>  =  passive participle
  <mor form>  =  love ed.
...
```

Die Lexikoneinträge oben sind bereits in DATR-Syntax dargestellt. Das *einzelne* Gleichheitszeichen weisst auf eine *extensionale* Gleichung hin. Die folgenden Beispiele werden überwiegend *Definitionen* zeigen, dargestellt mit *doppeltem* Gleichheitszeichen.

Am Beispiel oben kann man die starke Redundanz erkennen, die der naive Ansatz mit sich bringt. Kategorie und Typ „verb main" werden mehrfach verwendet, andere gemeinsame Eigenschaften aller „verb main" Objekte müssten ebenfalls auf alle betroffenen Lexikoneinträge kopiert werden.

Abhilfe schaffen hier Verweise: In der folgenden Version des Beispiels von oben werden syntaktische Kategorie und Typ jeweils von der allgemeinen Definition eines „VERB" Objektes geerbt. Die explizite Vererbungsanweisung für jede Eigenschaft ist noch stets nicht die optimale Lösung, wie wir weiter unten sehen werden.

```
VERB:
  <syn cat>   ==  verb ← == ist Definition
  <syn type>  ==  main.
Word1:
  <syn cat>   ==  VERB:<syn cat> ← Verweis zum Allgemeinen
  <syn type>  ==  VERB:<syn type> ← d.h. lokale Vererbung)
  <syn form>  ==  present participle
  <mor form>  ==  love ing.
Word1:
  <syn cat>   ==  VERB:<syn cat> ← wir wiederholen uns!
  <syn type>  ==  VERB:<syn type>
  <syn form>  ==  passive participle
  <mor form>  ==  love ed.
```

Eine weitere Verbesserungsmöglichkeit ist die Einführung einer weiteren Ebene im

beschriebenen Vererbungsnetz: Wir definieren alle Formen des Verbs „lieben" als Kinderknoten des Knotens „Love" welche die Grundform definiert. Im folgenden Beispiel ist auch eine weitere sinnvolle Eigenschaft von DATR erkennbar, die Pfadverlängerung bzw. Verwendung von Defaults. Die Zeile „`<> == VERB`" in „Love" legt fest, dass alle Eigenschaften von „Love" soweit keine Ausnahme definiert wurde als Eigenschaften von „VERB" zu suchen sind. Damit sind also syntaktische Kategorie und Typ von „Love" automatisch als diejenigen von „VERB" bekannt, ohne dass definiert werden musste, dass „Love" diese beiden Eigenschaften überhaupt besitzt. Dasselbe passiert bei der Vererbung von „Love" auf die beiden Lexikoneinträge:

```
VERB:
  <syn cat>   ==  verb
  <syn type>  ==  main.
Love:
  <>          ==  VERB ← default für alle Pfade
  <mor root>  ==  love.

Word2:
  <>          ==  Love
  <syn form>  ==  present participle
  <mor form>  ==  <mor root> ing.
Word1:
  <>          ==  Love
  <syn form>  ==  passive participle
  <mor form>  ==  <mor root> ed.  ← immernoch nicht optimal
```

Die letzte Verbesserungsmöglichkeit für unser Beispiel erfordert ein spezielles neues Konstrukt: *Auswertbare Pfade*. In DATR sind auswertbare Pfade daran erkennbar, dass sie von Anführungszeichen umschlossen werden. Ein auswertbarer Pfad gibt z.B. die Möglichkeit, in einem Mutterknoten die in einem Tochterknoten zu verwendenden Verweise zu definieren.

In der folgenden letzten Version des einführenden Beispiels definiert der Eintrag für „VERB" die normale Morphologie eines Verbs. Dabei wird z.B. die Morphologie des Past Tense als „Stamm plus -ed" definiert. Da nicht alle Verben denselben Stamm haben, definiert der Eintrag nur eine *Regel* in Form eines auswertbaren Pfades. Der Eintrag für das konkrete Verb „lieben" definiert einen Stamm und erbt alle allgemeinen Verb-Regeln durch die „`<> == VERB`" Zeile.

Zuletzt wird in den beiden Einträgen für die flektierten Formen die syntaktische Form definiert. Damit und mit der Vererbung vom Verb zu den Formen kann nun die Regel ausgewertet werden: Eine Abfrage nach der morphologischen Form von

„Word1" wird, da in „Word1" keine Angabe gemacht wird, mit dem gewählten Default „siehe morphologische Form von Love" und dort wiederum mit „siehe morphologische eines Verbes" beantwortet. Die resultierende Antwort ist aber ein auswertbarer Pfad: „siehe Morphologie von..." und das „..." selbst ist auch wieder ein auswertbarer Pfad, der zur syntaktischen Form verweist. Diese ist schliesslich für „Word1" definiert. Im nächsten Schritt wird dann „siehe Morphologie von Present Participle" über die global für alle Verben definierte Regel „siehe Stamm plus -ing" weiter ausgewertet. Der Stamm von „Word1" ist von „Love" als „love" geerbt.

Insgesamt liefert die Anfrage also das gewünschte Ergebnis „love plus ing" (die Ausgabe muss ggf. noch von einem weiteren Modul für englische Rechtschreibung nach „loving" korrigiert werden, dazu unten mehr). Vorteil bei der Verwendung von auswertbaren Pfaden ist also die Möglichkeit, „oben" in der Hierarchie allgemeine Regeln anzugeben, die erst weiter „unten" auf konkrete Instanzen angewandt werden. Obwohl die Regeln konkrete Instanzen bearbeiten, können sie auf allgemeiner Ebene bereits benannt werden.

```
VERB:
  <syn cat>              ==  verb
  <syn type>             ==  main
  <mor form>             ==  "<mor "<syn form>">"  ← doppelter e.p.
  <mor past>             ==  "<mor root>" ed  ← * evaluable paths *
  <mor passive>          ==  "<mor past>"  ← (für globale Vererbung)
  <mor pres>             ==  "<mor root>"  ← default für das Präfix
  <mor pres participle>  ==  "<mor root>" ing
  <mor pres tense sg 3>  ==  "<mor root>" s.


Love:          ← gemeinsamer mor root
  <>           ==  VERB
  <mor root>   ==  love.
Word1:
  <>           ==  Love
  <syn form>   ==  pres participle.  ← „aktiviert" die evaluable paths
Word2:
  <>           ==  Love
  <syn form>   ==  passive participle.
```

## 1.2 Kompliziertere Anwendungen

DATR ist nicht nur in der Lage, regelmässige Verbformen in einem zentralen „VERB" Eintrag kompakt zu definieren: Die Art der Default-Vererbung in DATR erlaubt auch die effiziente Beschreibung von gerade in menschlicher Sprache oft vorkommenden *Subregularitäten*. Im folgenden Beispiel wird eine vom oben gezeigten Regelsatz für „VERB" abgeleitete Klasse „EN_VERB" definiert und mit „Mow" und „Sew" instanziiert:

```
EN_VERB:
  <>                      ==  VERB ← subreg. Variante von VERB
  <mor past participle>   ==  "<mor root>" en.
Mow:
  <>                      ==  EN_VERB
  <mor root>              ==  mow.
Sew:
  <>                      ==  EN_VERB
  <mor root>              ==  sew.
```

Davon abgeleitet können wir sogar „Be" als Spezialfall eines „EN_VERB" definieren: Dazu geben wir für die gegenüber „EN_VERB" unregelmässigen Formen einfach direkt im Lexikoneintrag an. Für dort nicht festgelegte Formen gilt dann jeweils wieder der Standardwert „wie für ein EN_VERB" und für die nicht für en-Verben besonders festgelegten Formen wiederum der durch die allgemeine Verbdefinition berechenbare Wert.

```
Be:
  <>                           ==  EN_VERB
  <mor root>                   ==  be
  <mor pres tense sing one>    ==  am
  <mor pres tense sing three>  ==  is
  <mor pres tense plur>        ==  are
  <mor past tense sing one>    ==  ...
  <...>                        ==  ...
```

Im Beispiel oben ist zu erkennen, dass wir keine getrennten Angaben für die erste, zweite und dritte Person Plural von „Be" gemacht haben: Die für DATR verwendete Abfragelogik gibt hier die durch das längste passende Präfix des gefragten Pfades festgelegte Antwort zurück! Implizit werden „übrige" Teile der Frage sozusagen als links und rechts der Gleichheitszeichen gleichermassen anzuhängen betrachtet, wie hier zu erkennen: Die beiden Zeilen

```
VERB:
  <mor present tense>  ==  "<mor root>"
  <mor form>           ==  <mor "<syn form>">.
```

erlauben die Ableitung der vier folgenden Abfragen:

```
  <mor present tense sing>  ==  "<mor root sing>"
  <mor present tense plur>  ==  "<mor root plur>"
  <mor form present>        ==  <mor "<syn form present>" present>
  <mor form passive>        ==  <mor "<syn form passive>" passive>
```

Nähere Details zu dieser Funktion der Pfadverlängerung finden sich in Artikeln über die Auswertung von DATR Daten wie dem oben genannten Artikel von Bill Keller. Intuitiv sollte der Effekt jedoch aus dem Beispiel oben erkennbar sein.

Der Aufbau von DATR erlaubt auch die Definition und Verwendung von *Parametern*: Damit lässt sich die Reihenfolge der zu einem Pfad zu kombinierenden Elemente elegant abstrahieren:

```
VERB:
  <syn form>    ==  "<syn tense>" "<syn number>" "<syn person>"
Word4:
  <>            ==  Sew
  <syn tense>   ==  present
  <syn person>  ==  third ← Reihenfolge ist natürlich egal
  <syn number>  ==  sing.
```

Ein weiteres, vielleicht überraschendes, Anwendungsbeispiel für DATR ist die Interpretation einfacher Ausdrücke der Booleschen Algebra in UPN (Umgekehrte Polnische Notation: Der Operator steht vor seinen Argumenten). Hier erlaubt das Default-Verhalten von DATR eine kompakte Darstellung: Standard-Antwort ist „false", wenn der Ausdruck mit „or" oder „if" beginnt jedoch „true". Am Ende werden dann die Ausnahmen vom Standard festgelegt. Statt 14 Definitionen für die kompletten Wahrheitstabellen sind in DATR nur 7 nötig. Insgesamt trotzdem ein etwas untypisches Beispiel für die Anwendung von DATR:

```
Boolean:
  <>                 ==  false  ← allgemeinster default
  <or>               ==  true   ← default für or
  <if>               ==  true
  <not false>        ==  true
  <and true true>    ==  true   ← Ausnahme vom allg. default
  <if true false>    ==  false
  <or false false>   ==  false. ← Ausnahme vom or default
```

## 1.3 Die Syntax von DATR

Nach dieser doch recht informellen Vorstellung von DATR und seinen Funktionen an Beispielen stelle ich hier kurz einige Details zur Syntax von DATR vor. Wie jede Sprache enthält auch DATR zunächst mal einige besondere „reservierte Symbole" (vergleichbar mit den Satzzeichen in menschlicher Sprache und den Zeichen mit besonderer syntaktischer Bedeutung in Programmiersprachen). Diese Symbole sind bei DATR: `: '' < > = == . ' %` und `#`.

Der Punkt dient dabei der Abgrenzung von Sätzen voneinander. Ein Satz der nur eine Zeile (Statement) enthält ist ein „einfacher Satz". Das # Zeichen leitet Compilerdirektiven ein, mit % werden Kommentare begonnen. Einfaches und doppeltes Anführungszeichen werden gleichermassen als Anführungszeichen verwendet. Die Anwendung vor Doppelpunkt, spitzen Klammern und Gleichheitszeichen sollte aus der Einführung oben klar sein.

Weiterhin kommen in der Sprache DATR Knoten, Atome und Variablen vor. Ein Atom ist zum Beispiel `love`, während `$var1` eine Variable ist. Da DATR Sätze Gleichungen definieren, müssen auch die möglichen Werte für linke und rechte Werte festgelegt werden. Oder anders gesagt: Was ist eine mögliche *lvalue*, was eine mögliche *rvalue*?

Eine lvalue ist immer ein lokaler Vererbungsdeskriptor, ein Knoten und/oder ein Pfad. Wie zum Beispiel `knoten:<deskriptor1 deskriptor2>`. Wie man sieht ist die Definition von Deskriptoren rekursiv, da in Pfaden auch wiederum Deskriptoren vorkommen. Als rvalue sind ausserdem noch Atome, Variablen und globale Vererbungsdeskriptoren zugelassen. Ein globaler Vererbungsdeskriptor ist praktisch ein auswertbarer Pfad, also z.B ein Ausdruck der Form `„deskriptor1"`. Auch hier ist wieder Rekursion möglich, ein Deskriptor der sich selbst wieder aus lokalen und globalen Deskriptoren zusammensetzt ist auch eine mögliche rvalue.

Oben haben wir bereits intensiv Gebrauch von einer abkürzenden Schreibweise

gemacht: Wenn mehrere Statements sich auf denselben Knoten beziehen, können sie in einem Satz zusammengefasst werden. Die folgenden beiden Schreibweisen drücken also beide dasselbe aus:

```
Boolean:
  <or>  ==  true
  <if>  ==  true.


Boolean:<or>  ==  true.
Boolean:<if>  ==  true.
```

Ein Satz mit einem einfachen Gleichheitszeichen wird extensionaler Satz genannt, ein Satz mit doppeltem Gleichheitszeichen definitionaler Satz. Definition impliziert dabei auch zugleich Extension. Wie oben erklärt ist die Vererbung so definiert, dass der längste/beste passende Pfad gewinnt. Informelle Beispiele dafür finden sich wieder oben, für formelle Details und weitere Erklärungen verweise ich auf den zugrundeliegenden Artikel von Evans und Gazdar (z.B. Seiten 13 und 14) sowie auf den Artikel von Keller (alle siehe oben).

## 1.4 Weitere Anwendungsbeispiele: Transducer und Automaten

Das Beispiel unten definiert endlich den weit oben angekündigten Automaten, um aus „love plus ing" die korrekte Form „loving" zu machen. In der praktischen Anwendung muss man das Lexikon welches „love ing" liefert und diesen Automaten in getrennten Modulen implementieren. Bequemerweise lassen sich also beide Module in DATR formulieren, aber die Zerlegung eines Wortes in dessen Buchstaben und umgekehrt die Kombination von Buchstaben zu einem Wort muss ausserhalb von DATR stattfinden.

Zunächst definieren wir einige Mengen über Compilerdirektiven und speichern sie in Variablen:

```
# vars $abc: a b c d e f g h i j k l m n o p q r s t u v w x y z.
# vars $vowel: a e i o u.                          ← eine Compilerdirektive
# vars $consonant: $abc - $vowel.                  ← Differenzmenge
# vars $var.                                       ← beliebiger Wert
# vars $foo: $abc - b a r.
```

Bei der Verwendung dieser Mengen müssen wir aufpassen, nicht mehrere sich mögli-

cherweise widersprechende Regeln zu definieren die alle dieselbe Frage beantworten:

```
BUGGY:
  <e>       ==  e i <>
  <$vowel>  ==  $vowel e <>.  ← zwei Regeln für e!
```

In so einem Fall ist es dem Programmierer überlassen, eine angemessene Reaktion der Software auf solche DATR Sätze festzulegen. Zum Beispiel könnte die älteste treffende Regel gewinnen, neue Regeln können ältere Regeln überschreiben, oder das Programm kann mit einer Fehlermeldung abbrechen. Dazu weiter unten mehr.

Der Satz unten beschreibt eine korrekte Darstellung der Regel „ein e vor einer mit Vokal beginnenden Endung wird nicht geschrieben" – dabei nehmen wir „+" als Markierung für eine Schnittstelle zwischen Stamm und Endung an:

```
SPELL:
  <$abc>        ==  $abc <>
  <e + $vowel>  ==  $vowel <>.
```

Die Verwendung des <> sorgt dafür, dass die Regel auf sukzessive Reste des Abfragepfades immer wieder angewendet wird. Der Automat verwendet den Pfad als Eingabe und den Wert als Ausgabe. Siehe auch die Formulierung von Evans und Gazdar auf den Seiten 23 und 24 ihres Artikels... Nach dieser Regel sind nun zum Beispiel die folgenden extensionalen Sätze ableitbar:

```
  <l o v e>          =  l o v e
  <l o v e + s>      =  l o v e s
  <l o v e + e d>    =  l o v e d
  <l o v e + i n g>  =  l o v i n g
```

Die Idee der Anwendung von DATR für endliche Automaten lässt sich auf endliche Übersetzer erweitern: Damit kann der folgende Übersetzer (Umsetzer, Transducer) aus dem Pfad `<subj 1 sg futr obj 2 sg like>` korrekt den Swahili-Satz *„Ni ta ku penda"* konstruieren. Zugegeben ein etwas konstruiertes Beispiel...

```
S1:
  <subj 1 sg>  ==  ni S2:<>
  <subj 2 sg>  ==  u S2:<>
  <subj 3 sg>  ==  a S2:<>
  <subj 1 pl>  ==  tu S2:<>
  <subj 2 pl>  ==  m S2:<>
  <subj 3 pl>  ==  wa S2:<>.
```

```
S2:
  <past>   ==  li S3:<>.
  <futr>   ==  ta S3:<>.

S3:
  <obj 1 sg>   ==  ni S2:<>
  <obj 2 sg>   ==  ku S2:<>
  <obj 3 sg>   ==  m S2:<>
  <obj 1 pl>   ==  tu S2:<>
  <obj 2 pl>   ==  wa S2:<>
  <obj 3 pl>   ==  wa S2:<>.

S4:
  <like>   ==  penda.
```

# 2 Komplexere Themen, Lösungen und Probleme

## 2.1 Eine Stufe weiter: Regelmässige Regeln

Bisher haben wir gesehen, wie DATR Regelmässigkeiten im Lexikon kompakt darstellen kann und dass sich damit auch endliche Automaten und Transducer beschreiben lassen. Unter Verwendung von Variablen/Parametern und auswertbarer Pfade lassen sich aber sogar Regelmässigkeiten in den Regeln selbst kompakt ausdrücken!

Im Folgenden dazu das Beispiel von Subkategorisierungslisten. Damit stossen wir allerdings auch gleich an die Grenzen dessen, was mit DATR direkt darstellbar ist: Listen und Disjunktion müssen unter Zuhilfenahme externer Module verarbeitet werden.

## 2.2 Ein Beispiel: Subkategorisierung

Hier ein Vorschlag aus dem Text von Evans und Gazdar, wie man in DATR Subkategorisierungslisten implementieren kann – Details siehe Originaltext. Eine Subkategorisierungsliste gibt an, welche Arten von Argumenten ein bestimmtes Verb „konsumiert":

```
NIL:
  <>                          ==  nil  ← Ende einer Liste
  <rest>                      ==  UNDEF
  <first>                     ==  UNDEF.
NP_ARG:                                  ← eine Abstraktionshilfe
  <first syn cat>             ==  np
  <first syn case>            ==  accusative
  <rest>                      ==  NIL:<>.
VERB:                                        ← intransitives Verb
  <syn cat>                   ==  verb
  <syn args>                  ==  NP_ARG:<>
  <syn args first syn case>   ==  nominative.
TR_VERB:
  <>                          ==  VERB
  <syn args rest>             ==  NP_ARG:<>.
DI_VERB:
  <>                          ==  TR_VERB  ← erste zwei Arg. wie bisher
  <syn args rest rest>        ==  PP_ARG:<>.  ← hier nicht gezeigt
```

DATR verarbeitet Listen hier mit dem z.B. aus Prolog und LISP bekannten Ansatz von rekursiver Einteilung einer Liste in Kopf und Rest. Das Ende der Liste explizit auf den Rest „NIL" zu setzen verhindert dabei unerwünschte Treffer durch die automatische Pfadverlängerungsfunktion von DATR. Ausserdem ist gut zu sehen wie die Einträge mit $N+1$ Argumenten aus den Einträgen mit $N$ Argumenten abgeleitet werden. Der Eintrag „PP_ARG" ist hier nicht spezifiziert, die Idee ist auch ohne ihn verständlich.

## 2.3 Ein Problem: Disjunktion

DATR hat selten Mühe, auf eine Frage eine Antwort zu finden, zum Beispiel dank des Default-Mechanismus. Aber findet man auch *zwei* Antworten auf eine Frage? Das Beispiel von zwei Einträgen die beide auf die morphologische Form „bank" passen liefert bei vielen DATR verarbeitenden Systemen nur eine Fehlermeldung oder nur einen der beiden Einträge bei der Frage nach der Bedeutung des Wortes „bank":

```
Bank1:
  <>           ==  NOUN
  <mor root>   ==  bank
  <sem gloss>  ==  side of river.
Bank2:
  <>           ==  NOUN
  <mor root>   ==  bank
  <sem gloss>  ==  financial institution.
```

Ein anderes Beispiel für dieses Problemfeld ist regelmässige Polysemie: Man würde gerne von der Liste von Beschreibungen (Bedeutungen) für „cherry" ausgehend abstrahieren: Wenn ein Wort einen Obstbaum benennt, hat es im Englischen die Bedeutungen dessen Frucht, dessen Holz und des Baumes.

```
Cherry:
  <>              ==  NOUN
  <mor root>      ==  cherry
  <sem gloss 1>   ==  sweet red berry ← Aufzählung mit Nummern
  <sem gloss 2>   ==  tree bearing <sem gloss 1>
  <sem gloss 3>   ==  wood from <sem gloss 2>.
```

Im Beispiel oben wurde ein Index im Pfad verwendet um mehrere Einträge für die Bedeutung zu erlauben. Diese abzufragen ist umständlich, es wäre besser, man könnte in einem Wert mehrere Antworten verpacken. Leider bietet DATR keine entsprechende Funktion. Evans und Gazdar schlagen drei Lösungen für die Darstellung von Disjunktion und Aufzählungen vor:

```
Word71:
  <syn number>           ==  plural
  <mor form>             ==  hoof s
  <mor form alternate>   ==  hoove s.  ← ähnlich wie oben
Word72:
  <syn number>           ==  plural
  <mor forms>            ==  hoof s | hoove s.
Word72:
  <syn number>           ==  plural
  <mor forms>            ==  { hoof s , hoove s }.
```

Die „Form / alternative Form" Lösung erinnert an das vorige Beispiel und ist nicht wirklich befriedigend. Besser sind die Lösungen mit „Alternativen-Strich" und als Listendarstellung mit geschweiften Klammern. Für DATR sind jedoch `{ foo , bar }` und `foo | bar` nur Daten. Die gewählte Listendarstellung als Liste zu *interpre-*

*tieren* ist also Aufgabe eines externen Parsers. Irgendetwas ausserhalb von DATR muss also z.B. die in der Liste aufgeführten Antworten nacheinander abarbeiten. DATR gibt genau genommen immernoch nur eine Antwort.

## 2.4 Noch ein Problem: Konsistenz

Oben habe ich bereits angedeutet, dass es nicht wünschenswert ist, für eine Frage mehrere zutreffende Regeln und damit eventuell mehrere mögliche Antworten zu haben. Hier wird auf dieses Problem näher eingegangen. Meistens sind DATR Beschreibungen definitional und es gibt maximal eine Aussage die auf ein gegebenes Knoten/Pfad Paar passt. Damit beschreibt eine DATR „Datenbank" normalerweise partielle Funktionen von Pfaden zu Werten.

Die Syntax von DATR schliesst Inkonsistenzen nicht aus. Also kann man entweder neue Sätze (solche die später eingelesen werden) die alte Sätze „überlagern" so interpretieren, dass sie *Korrekturen* für die alten Sätze darstellen. Die betreffenden alten Aussagen werden also überschrieben. Es ist eventuell sinnvoll, den Benutzer darüber zu informieren und eine Warnung auszugeben, wenn solch ein Fall eintritt.

Die andere Möglichkeit ist, Datenbanken mit Inkonsistenzen völlig abzulehnen. Dazu muss das Programm welches die Datenbank einliest die *Funktionalität* der Datenbank prüfen und sicherstellen.

Funktionalität einer DATR Beschreibung ist wie folgt definiert:

*Eine DATR Beschreibung ist funktional genau dann wenn sie nur definitionale Aussagen enthält und diese eine (partielle) Funktion von Knoten/Pfad Paaren zu Deskriptorsequenzen beschreiben.*

Jede funktionale DATR Beschreibung ist auch konsistent. Intuitiv ist das so weil man aus (partiellen) Funktionen wiederum nur (partielle) Funktionen zusammenstellen kann, im Rahmen der DATR Sprache. Allerdings können auch nicht-funktionale DATR Beschreibungen konsistent sein. Das ist allerdings praktisch kaum von Nutzen, deshalb kann man ausser INCONSISTENT auch die drei NONFUNC Sätze zurückweisen und strikt Funktionalität fordern. Die drei NONFUNC Sätze sind jedoch alle konsistent.

```
INCONSISTENT:
  <syn cat>  ==  verb
  <syn cat>  ==  noun.  ← Widersprüchlich, aber Syntax erlaubt das!
```

```
NONFUNC1:
  <a>  ==  UNDEF  ← bewirkt keine constraints
  <a>  ==  1.  ← also dennoch konsistent
NONFUNC2:
  <a>  ==  b
  <a>  ==  1
  <b>  ==  1.  ← zufällig konsistent, da gleicher Wert
NONFUNC3:
  <a>  ==  b
  <b>  ==  a  ← gegenseitige Abhängigkeit! (aber konsistent)
  <a>  ==  1.
```

# 3 „The Big Picture"

## 3.1 Vererbungshierarchien

Evans und Gazdar gehen auf den Seiten 32 bis 34 ihres Textes auf die Problematik der Kohärenz im Zusammenhang mit verschiedenen Arten von Vererbungshierarchien ein: Ein einfaches Vererbungsnetzwerk ist quasi baumförmig, jeder Knoten stammt von maximal einer Quelle ab (man könnte also auch sagen „waldförmig" – das ganze Netzwerk kann in mehrere Unternetze zerfallen sein).

Ein Problem kann jedoch auftreten wenn ein Knoten *von mehreren* Elternknoten widersprüchliche Informationen erbt. In einem *orthogonalen* Vererbungsnetzwerk (OMI, Orthogonal Multiple Inheritance) ist festgelegt, dass ein Knoten von seinen Elternknoten nur jeweils verschiedene Eigenschaften erbt: Auch hier können also nicht zwei oder mehr Elternknoten einander widersprechende Informationen zur gleichen Eigenschaft an einen Tochterknoten vererben. OMI ist in konsistenten DATR Beschreibungen eine normaler Fall, der längste Subpfad gewinnt jeweils eine Abfrage.

Ohne diese Einschränkungen muss man von mehreren einander widersprechenden Aussagen eine zur Vererbung auswählen. Dabei kann man verschiedene Regeln und Heuristiken anwenden die *Evans et al.* unter dem Stichwort *Prioritized Multiple Inheritance* (PMI) in ihrem Text *„Prioritized multiple inheritance in DATR" (1993, in: „Inheritance, defaults, and the lexicon" von Brisco, de Paiva und Copestake (eds.), Seiten 38 bis 46)* näher beschreiben. PMI lässt sich also in DATR darstellen / implementieren, bietet aber laut dem vorliegenden Artikel von Evans und Gazdar nur wenige Vorteile.

## 3.2 Anwendungsmodi, Anwendungen und Implementationen

In dieser Ausarbeitung wurde bisher nur allgemein von einem Programm welches DATR Beschreibungen verarbeitet gesprochen. In der Praxis will man aber auf verschiedene Weise mit DATR arbeiten:

Ein DATR System verarbeitet Theorien (DATR Beschreibungen, Datenbanken), Abfragen (Pfade) und Werte. Jeweils zwei dieser Teile genügen um den jeweils dritten Teil zu finden:

Die „normale" Anwendung ist die Suche nach einem Wert, bei gegebener Datenbank und mit dem Pfad als Abfrage. Das ist vergleichbar mit der *Benutzung* eines Lexikons. Beispiel: `Love:<mor past participle>` liefert `love ed` mittels der Inferenzmethoden die z.B. im erwähnten Artikel von Keller verwendet werden.

Zweite Anwendungsmöglichkeit ist die *Inverse Abfrage*: Zum Beispiel kann man suchen, zu welchem Verb `love ed` eine Form ist und welche. Dass kann zur *Wartung* eines Lexikons nützlich sein oder zur Analyse eines Textes. Diese Art der Abfrage ist meistens computationell viel teurer als die gewöhnliche Inferenz.

Dritte und kaum implementierte Möglichkeit ist die *Theorieerzeugung*: Hier wird aus gegebenen Werten und Abfragen ein Lexikon *erzeugt*. In der Praxis ist diese Methode leider kaum brauchbar, da Daten und Korpora nicht perfekt geeigneten Abfragen zuzuordnen sind bzw. nicht geeignet annotiert sind. Ansonsten wäre dieser Einsatz von DATR natürlich enorm nützlich. Teilweise wird er aber für den groben Entwurf, die Umwandlung und die Erweiterung bestehender Lexika und Morphologien erfolgreich eingesetzt.

Evans und Gazdar nennen gegen Ende ihres Artikels eine Reihe bekannter DATR Anwendungen und Implementationen. Für weiterführende Literatur verweise ich auf die Literaturliste im Anhang des Originalartikels! Einige der Beispiele:

- Light et al. stellen die Anwendung von DATR Theorieerzeugung um Hierarchien zu verfeinern vor.
- Duda und Gebhardi verwenden ein DATR Lexikon für PATR Parser, ihre Software bietet dem PATR Parser volle Abfragemöglichkeiten im DATR Lexikon.
- Gibbon stellt eine erweiterter Abfragesprache EDQL vor, mit der man Abfragen an DATR Systeme formulieren kann.
- Eine klassische Implementierung von DATR ist die von Evans und Gazdar in Prolog.
- Andere DATR Implementierungen sind DDATR (Scheme) und NODE (Pro-

log) von Gibbon und QDATR (Prolog).

- Langer hat Versionen mit der Möglichkeiten zur Inversen Abfrage (reverse query) entwickelt.
- Barg hat aus extensionalen Daten DATR Beschreibungen abgleitet (untersucht also Theorieerzeugung).

## 3.3 Ausblick und Zusammenfassung

DATR ist eine einfache, allgemein gehaltene „Programmiersprache" insbesondere für die lexikalische Semantik, vergleichbar mit HPSG für die Syntax. DATR ist bewusst nicht an einer bestimmten „lexikalischen" Theorie orientiert und damit geeignet, um vielerlei Anwendungen von *nichtmonotonen Vererbungsnetzen* zu realisieren. Die Nichtmonotonie wird dabei durch die Arbeitsteilung von Defaults und Sonderfällen realisiert, ein für die lexikalische Semantik sehr praktischer Ansatz.

DATR ist eine „kleine" Sprache, Syntax, Inferenz und deklarative Semantik von DATR sind *klar und vollständig definiert*. Das erleichtert sowohl die Implementierung auf dem Computer als auch die mathematische Analyse von Systemen die mit DATR arbeiten und von DATR selbst. DATR wurde bereits für Systeme die auf PATR, LTAG, UCG und anderen Formalismen aufbauen erfolgreich verwendet, teilweise in grösseren Projekten und teilweise bisher nur mit Testdatensätzen.

Auch die Klasse der menschlichen Sprachen für deren Verarbeitung sich DATR Module sinnvoll einsetzen lassen ist sehr gross: Bisher wurde DATR für Arabische, Tschechische, Englische, Spanische, Russische und Lateinische computerlinguistische Projekte eingesetzt und für weitere Sprachen wie Japanisch und Sanskrit wurde mit DATR experimentiert. Dabei ist DATR *nicht* auf eine *bestimmte Ebene* der Sprachverarbeitung festgelegt: Es lässt sich für Phonologie, Orthographie, Morphologie, Syntax, Semantik und so weiter einsetzen.

Evans und Gazdar schlagen vor, ein komplettes System zur Sprachverarbeitung aus DATR Modulen zusammenzusetzen: DRT für die (Diskurs-) Semantik, HPSG für die Syntax, weitere Module für Morphologie und Ortographie, aber alle Module aufbauend auf in DATR dargestellten Daten.

Auf der Homepage `http://www.datr.org` finden sich viele Beispielfragmente und Informationen, sowie natürlich Verweise auf frei verfügbare DATR Implementationen.